Computer Operating System

Floyd Richmond

COMPUTER OPERATING SYSTEM

COMPUTER OPERATING SYSTEM

Floyd Richmond

BIBLIOTEX
Digital Library

Computer Operating System
by Floyd Richmond

www.bibliotex.com

Ebook ISBN: 9781984664242

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

1

Operating System

An operating system (OS) is software, consisting of programmes and data, that runs on computers, manages computer hardware resources, and provides common services for execution of various application software.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between application programmes and the computer hardware, although the application code is usually executed directly by the hardware and will frequently call the OS or be interrupted by it. Operating systems are found on almost any device that contains a computer—from cellular phones and video game consoles to supercomputers and web servers.

Examples of popular modern operating systems for personal computers are: Microsoft Windows, Mac OS X, GNU/Linux, and Unix.

Types of Operating Systems

Real-time Operating System: It is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behaviour. The main object of real-time operating systems is their quick and predictable response to events. They either have an event-driven or a time-sharing design. An event-driven system switches between tasks based on their priorities while time-sharing operating systems switch tasks based on clock interrupts.

Multi-user and Single-user Operating Systems: The operating systems of this type allow a multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. Single-user operating systems, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to have multiple accounts on a Windows operating system does not make it a multi-user system. Rather, only the network administrator is the real user. But for a Unix-like operating system, it is possible for two users to login at a time and this capability of the OS makes it a multi-user operating system.

Multi-tasking and Single-tasking Operating Systems: When a single programme is allowed to run at a time, the system is grouped under a single-tasking system, while in case the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi-tasking can be of two types namely, pre-

emptive or co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programmes. Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. MS Windows prior to Windows 95 used to support cooperative multitasking.

Distributed Operating System: An operating system that manages a group of independent computers and makes them appear to be a single computer is known as a distributed operating system. The development of networked computers that could be linked and communicate with each other, gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

Embedded System: The operating systems designed for being used in embedded computer systems are known as embedded operating systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

Summary

Early computers were built to perform a series of single tasks, like a calculator. Operating systems did not exist in their modern and more complex forms until the early

1960s. Some operating system features were developed in the 1950s, such as monitor programmes that could automatically run different application programmes in succession to speed up processing. Hardware features were added that enabled use of runtime libraries, interrupts, and parallel processing.

When personal computers by companies such as Apple Inc., Atari, IBM and Amiga became popular in the 1980s, vendors added operating system features that had previously become widely used on mainframe and mini computers. Later, many features such as graphical user interface were developed specifically for personal computer operating systems.

An operating system consists of many parts. One of the most important components is the kernel, which controls low-level processes that the average user usually cannot see: it controls how memory is read and written, the order in which processes are executed, how information is received and sent by devices like the monitor, keyboard and mouse, and decides how to interpret information received from networks. The user interface is a component that interacts with the computer user directly, allowing them to control and use programmes.

The user interface may be graphical with icons and a desktop, or textual, with a command line. Application programming interfaces provide services and code libraries that let applications developers write modular code reusing well defined programming sequences in user space libraries or in the operating system itself. Which features are

considered part of the operating system is defined differently in various operating systems. For example, Microsoft Windows considers its user interface to be part of the operating system, while many versions of Linux do not.

History

In the early 1950s, a computer could execute only one programme at a time. Each user had sole use of the computer and would arrive at a scheduled time with programme and data on punched paper cards and tape.

The programme would be loaded into the machine, and the machine would be set to work until the programme completed or crashed. Programmes could generally be debugged via a front panel using toggle switches and panel lights. It is said that Alan Turing was a master of this on the early Manchester Mark 1 machine, and he was already deriving the primitive conception of an operating system from the principles of the Universal Turing machine.

Later machines came with libraries of software, which would be linked to a user's programme to assist in operations such as input and output and generating computer code from human-readable symbolic code.

This was the genesis of the modern-day operating system. However, machines still ran a single job at a time. At Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different colored clothes-pegs to indicate job-priority.

Examples of Operating Systems

Microsoft Windows

Bootable Windows To Go USB flash drive

Microsoft Windows is a family of proprietary operating systems designed by Microsoft Corporation and primarily targeted to Intel architecture based computers, with an estimated 88.9 percent total usage share on Web connected computers. The newest version is Windows 7 for workstations and Windows Server 2008 R2 for servers. Windows 7 recently overtook Windows XP as most used OS.

Microsoft Windows originated in 1985 as an application running on top of MS-DOS, which was the standard operating system shipped on most Intel architecture personal computers at the time. In 1995, Windows 95 was released which only used MS-DOS as a bootstrap. For backwards compatibility, Win9x could run real-mode MS-DOS and 16 bits Windows 3.x drivers. Windows Me, released in 2000, was the last version in the Win9x family. Later versions have all been based on the Windows NT kernel. Current versions of Windows run on IA-32 and x86-64 microprocessors, although Windows 8 will support ARM architecture. In the past, Windows NT supported non-Intel architectures.

Server editions of Windows are widely used. In recent years, Microsoft has expended significant capital in an effort to promote the use of Windows as a server operating

environment. However, Windows' usage on servers is not as widespread as on personal computers, as Windows competes against Linux and BSD for server market share.

Other

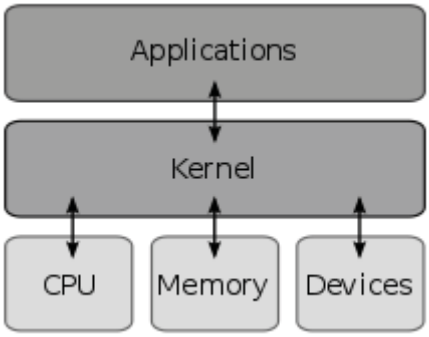
There have been many operating systems that were significant in their day but are no longer so, such as AmigaOS; OS/2 from IBM and Microsoft; Mac OS, the non-Unix precursor to Apple's Mac OS X; BeOS; XTS-300; RISC OS; MorphOS and FreeMint. Some are still used in niche markets and continue to be developed as minority platforms for enthusiast communities and specialist applications.OpenVMS formerly from DEC, is still under active development by Hewlett-Packard. Yet other operating systems are used almost exclusively in academia, for operating systems education or to do research on operating system concepts. A typical example of a system that fulfills both roles is MINIX, while for example Singularity is used purely for research.

Other operating systems have failed to win significant market share, but have introduced innovations that have influenced mainstream operating systems, not least Bell Labs' Plan 9.

Components

The components of an operating system all exist in order to make the different parts of a computer work together. All software—from financial databases to film editors—needs to go through the operating system in order to use any of the hardware, whether it be as simple as a mouse or keyboard or complex as an Internet connection.

Kernel



A kernel connects the application software to the hardware of a computer.

With the aid of the firmware and device drivers, the kernel provides the most basic level of control over all of the computer's hardware devices. It manages memory access for programs in the RAM, it determines which programs get access to which hardware resources, it sets up or resets the CPU's operating states for optimal operation at all times, and it organizes the data for long-term non-volatile storage with file systems on such media as disks, tapes, flash memory, etc.

Program Execution

The operating system provides an interface between an application program and the computer hardware, so that an application program can interact with the hardware only by obeying rules and procedures programmed into the operating system. The operating system is also a set of services which simplify development and execution of application programs. Executing an application program involves the creation of a process by the operating system kernel which assigns memory space and other resources, establishes a priority for the process in multi-tasking

systems, loads program binary code into memory, and initiates execution of the application program which then interacts with the user and with hardware devices.

Interrupts

Interrupts are central to operating systems, as they provide an efficient way for the operating system to interact with and react to its environment. The alternative — having the operating system "watch" the various sources of input for events (polling) that require action — can be found in older systems with very small stacks (50 or 60 bytes) but are unusual in modern systems with large stacks.Interrupt-based programming is directly supported by most modern CPUs. Interrupts provide a computer with a way of automatically saving local register contexts, and running specific code in response to events. Even very basic computers support hardware interrupts, and allow the programmer to specify code which may be run when that event takes place.

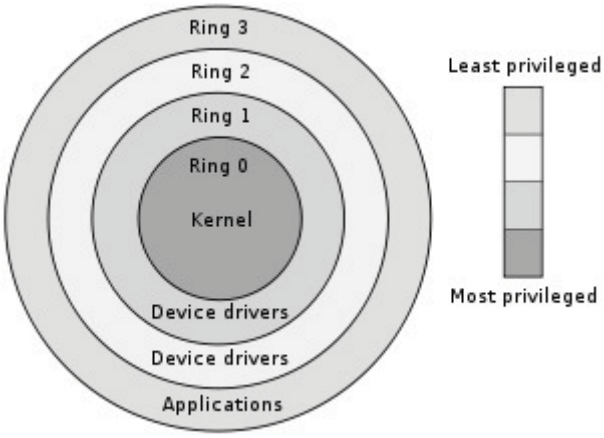
When an interrupt is received, the computer's hardware automatically suspends whatever program is currently running, saves its status, and runs computer code previously associated with the interrupt; this is analogous to placing a bookmark in a book in response to a phone call. In modern operating systems, interrupts are handled by the operating system's kernel. Interrupts may come from either the computer's hardware or from the running program.

When a hardware device triggers an interrupt, the operating system's kernel decides how to deal with this event, generally by running some processing code. The

amount of code being run depends on the priority of the interrupt (for example: a person usually responds to a smoke detector alarm before answering the phone). The processing of hardware interrupts is a task that is usually delegated to software called device driver, which may be either part of the operating system's kernel, part of another program, or both. Device drivers may then relay information to a running program by various means.

A program may also trigger an interrupt to the operating system. If a program wishes to access hardware for example, it may interrupt the operating system's kernel, which causes control to be passed back to the kernel. The kernel will then process the request. If a program wishes additional resources (or wishes to shed resources) such as memory, it will trigger an interrupt to get the kernel's attention.

Modes



Privilege rings for the x86 available in protected mode. Operating systems determine which processes run in each mode.

Modern CPUs support multiple modes of operation. CPUs with this capability use at least two modes: protected mode and supervisor mode. The supervisor mode is used by the

operating system's kernel for low level tasks that need unrestricted access to hardware, such as controlling how memory is written and erased, and communication with devices like graphics cards. Protected mode, in contrast, is used for almost everything else. Applications operate within protected mode, and can only use hardware by communicating with the kernel, which controls everything in supervisor mode. CPUs might have other modes similar to protected mode as well, such as the virtual modes in order to emulate older processor types, such as 16-bit processors on a 32-bit one, or 32-bit processors on a 64-bit one.

When a computer first starts up, it is automatically running in supervisor mode. The first few programs to run on the computer, being the BIOS or EFI,bootloader, and the operating system have unlimited access to hardware - and this is required because, by definition, initializing a protected environment can only be done outside of one. However, when the operating system passes control to another program, it can place the CPU into protected mode.

In protected mode, programs may have access to a more limited set of the CPU's instructions. A user program may leave protected mode only by triggering an interrupt, causing control to be passed back to the kernel. In this way the operating system can maintain exclusive control over things like access to hardware and memory.

The term "protected mode resource" generally refers to one or more CPU registers, which contain information that the running program isn't allowed to alter. Attempts to alter these resources generally causes a switch to supervisor

mode, where the operating system can deal with the illegal operation the program was attempting (for example, by killing the program).

Memory Management

Among other things, a multiprogramming operating system kernel must be responsible for managing all system memory which is currently in use by programs. This ensures that a program does not interfere with memory already in use by another program. Since programs time share, each program must have independent access to memory.

Cooperative memory management, used by many early operating systems, assumes that all programs make voluntary use of thekernel's memory manager, and do not exceed their allocated memory. This system of memory management is almost never seen any more, since programs often contain bugs which can cause them to exceed their allocated memory. If a program fails, it may cause memory used by one or more other programs to be affected or overwritten. Malicious programs or viruses may purposefully alter another program's memory, or may affect the operation of the operating system itself. With cooperative memory management, it takes only one misbehaved program to crash the system.

Memory protection enables the kernel to limit a process' access to the computer's memory. Various methods of memory protection exist, including memory segmentation and paging. All methods require some level of hardware support (such as the 80286 MMU), which doesn't exist in all computers.

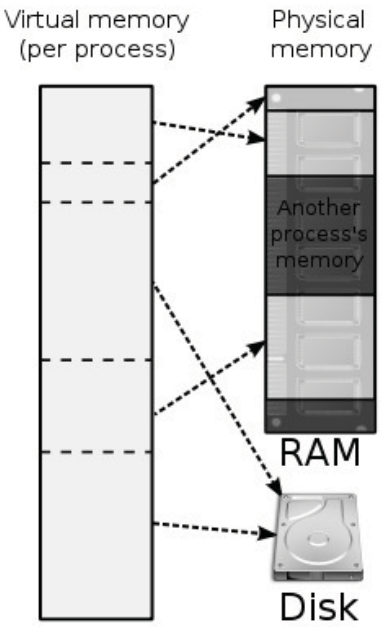
In both segmentation and paging, certain protected mode registers specify to the CPU what memory address it should allow a running program to access. Attempts to access other addresses will trigger an interrupt which will cause the CPU to re-enter supervisor mode, placing the kernel in charge. This is called a segmentation violation or Seg-V for short, and since it is both difficult to assign a meaningful result to such an operation, and because it is usually a sign of a misbehaving program, the kernel will generally resort to terminating the offending program, and will report the error.

Windows 3.1-Me had some level of memory protection, but programs could easily circumvent the need to use it. A general protection fault would be produced, indicating a segmentation violation had occurred; however, the system would often crash anyway.

Virtual Memory

The use of virtual memory addressing (such as paging or segmentation) means that the kernel can choose what memory each program may use at any given time, allowing the operating system to use the same memory locations for multiple tasks.

If a program tries to access memory that isn't in its current range of accessible memory, but nonetheless has been allocated to it, the kernel will be interrupted in the same way as it would if the program were to exceed its allocated memory. Under UNIX this kind of interrupt is referred to as a page fault.



Many operating systems can "trick" programs into using memory scattered around the hard disk and RAM as if it is one continuous chunk of memory, called virtual memory.

When the kernel detects a page fault it will generally adjust the virtual memory range of the program which triggered it, granting it access to the memory requested. This gives the kernel discretionary power over where a particular application's memory is stored, or even whether or not it has actually been allocated yet.

In modern operating systems, memory which is accessed less frequently can be temporarily stored on disk or other media to make that space available for use by other programs. This is called swapping, as an area of memory can be used by multiple programs, and what that memory area contains can be swapped or exchanged on demand.

"Virtual memory" provides the programmer or the user with the perception that there is a much larger amount of RAM in the computer than is really there.

Multitasking

Multitasking refers to the running of multiple independent computer programs on the same computer; giving the appearance that it is performing the tasks at the same time. Since most computers can do at most one or two things at one time, this is generally done via time-sharing, which means that each program uses a share of the computer's time to execute.

An operating system kernel contains a piece of software called a scheduler which determines how much time each program will spend executing, and in which order execution control should be passed to programs. Control is passed to a process by the kernel, which allows the program access to the CPU and memory. Later, control is returned to the kernel through some mechanism, so that another program may be allowed to use the CPU. This so-called passing of control between the kernel and applications is called a context switch.

An early model which governed the allocation of time to programs was called cooperative multitasking. In this model, when control is passed to a program by the kernel, it may execute for as long as it wants before explicitly returning control to the kernel. This means that a malicious or malfunctioning program may not only prevent any other programs from using the CPU, but it can hang the entire system if it enters an infinite loop.

Modern operating systems extend the concepts of application preemption to device drivers and kernel code, so that the operating system has preemptive control over internal run-times as well.

The philosophy governing preemptive multitasking is that of ensuring that all programs are given regular time on the CPU. This implies that all programs must be limited in how much time they are allowed to spend on the CPU without being interrupted. To accomplish this, modern operating system kernels make use of a timed interrupt. A protected mode timer is set by the kernel which triggers a return to supervisor mode after the specified time has elapsed. On many single user operating systems cooperative multitasking is perfectly adequate, as home computers generally run a small number of well tested programs. The AmigaOS is an exception, having pre-emptive multitasking from its very first version. Windows NTwas the first version of Microsoft Windows which enforced preemptive multitasking, but it didn't reach the home user market untilWindows XP (since Windows NT was targeted at professionals).

Disk Access and File Systems

Access to data stored on disks is a central feature of all operating systems. Computers store data on disks using files, which are structured in specific ways in order to allow for faster access, higher reliability, and to make better use out of the drive's available space. The specific way in which files are stored on a disk is called afile system, and enables files to have names and attributes. It also allows them to be stored in a hierarchy of directories or folders arranged in a directory tree.

Early operating systems generally supported a single type of disk drive and only one kind of file system. Early file systems were limited in their capacity, speed, and in the

kinds of file names and directory structures they could use. These limitations often reflected limitations in the operating systems they were designed for, making it very difficult for an operating system to support more than one file system.

While many simpler operating systems support a limited range of options for accessing storage systems, operating systems like UNIX and GNU/Linux support a technology known as a virtual file system or VFS. An operating system such as UNIX supports a wide array of storage devices, regardless of their design or file systems, allowing them to be accessed through a commonapplication programming interface (API). This makes it unnecessary for programs to have any knowledge about the device they are accessing. A VFS allows the operating system to provide programs with access to an unlimited number of devices with an infinite variety of file systems installed on them, through the use of specific device drivers and file system drivers. A connected storage device, such as a hard drive, is accessed through a device driver. The device driver understands the specific language of the drive and is able to translate that language into a standard language used by the operating system to access all disk drives. On UNIX, this is the language of block devices.

When the kernel has an appropriate device driver in place, it can then access the contents of the disk drive in raw format, which may contain one or more file systems. A file system driver is used to translate the commands used to access each specific file system into a standard set of commands that the operating system can use to talk to all file systems. Programs can then deal with these file systems

on the basis of filenames, and directories/folders, contained within a hierarchical structure. They can create, delete, open, and close files, as well as gather various information about them, including access permissions, size, free space, and creation and modification dates.

Various differences between file systems make supporting all file systems difficult. Allowed characters in file names, case sensitivity, and the presence of various kinds of file attributes makes the implementation of a single interface for every file system a daunting task. Operating systems tend to recommend using (and so support natively) file systems specifically designed for them; for example, NTFSin Windows and ext3 and ReiserFS in GNU/Linux. However, in practice, third party drives are usually available to give support for the most widely used file systems in most general-purpose operating systems (for example, NTFS is available in GNU/Linux through NTFS-3g, and ext2/3 and ReiserFS are available in Windows through FS-driver and rfstool).

Support for file systems is highly varied among modern operating systems, although there are several common file systems which almost all operating systems include support and drivers for. Operating systems vary on file system support and on the disk formats they may be installed on. Under Windows, each file system is usually limited in application to certain media; for example, CDs must use ISO 9660 or UDF, and as of Windows Vista, NTFS is the only file system which the operating system can be installed on. It is possible to install GNU/Linux onto many types of file systems. Unlike other operating systems, GNU/Linux and UNIX allow any

file system to be used regardless of the media it is stored in, whether it is a hard drive, a disc (CD,DVD...), a USB flash drive, or even contained within a file located on another file system.

Device Drivers

A device driver is a specific type of computer software developed to allow interaction with hardware devices. Typically this constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. It is a specialized hardware-dependent computer program which is also operating system specific that enables another program, typically an operating system or applications software package or computer program running under the operating system kernel, to interact transparently with a hardware device, and usually provides the requisite interrupt handling necessary for any necessary asynchronous time-dependent hardware interfacing needs.

The key design goal of device drivers is abstraction. Every model of hardware (even within the same class of device) is different. Newer models also are released by manufacturers that provide more reliable or better performance and these newer models are often controlled differently. Computers and their operating systems cannot be expected to know how to control every device, both now and in the future. To solve this problem, operating systems essentially dictate how every type of device should be

controlled. The function of the device driver is then to translate these operating system mandated function calls into device specific calls. In theory a new device, which is controlled in a new manner, should function correctly if a suitable driver is available. This new driver will ensure that the device appears to operate as usual from the operating system's point of view.

Under versions of Windows before Vista and versions of Linux before 2.6, all driver execution was co-operative, meaning that if a driver entered an infinite loop it would freeze the system. More recent revisions of these operating systems incorporate kernel preemption, where the kernel interrupts the driver to give it tasks, and then separates itself from the process until it receives a response from the device driver, or gives it more tasks to do.

Networking

Currently most operating systems support a variety of networking protocols, hardware, and applications for using them. This means that computers running dissimilar operating systems can participate in a common network for sharing resources such as computing, files, printers, and scanners using either wired or wireless connections. Networks can essentially allow a computer's operating system to access the resources of a remote computer to support the same functions as it could if those resources were connected directly to the local computer. This includes everything from simple communication, to using networked file systems or even sharing another computer's graphics or sound hardware. Some network services allow the

resources of a computer to be accessed transparently, such as SSH which allows networked users direct access to a computer's command line interface.

Client/server networking allows a program on a computer, called a client, to connect via a network to another computer, called a server. Servers offer (or host) various services to other network computers and users. These services are usually provided through ports or numbered access points beyond the server's network address. Each port number is usually associated with a maximum of one running program, which is responsible for handling requests to that port. A daemon, being a user program, can in turn access the local hardware resources of that computer by passing requests to the operating system kernel.

Many operating systems support one or more vendor-specific or open networking protocols as well, for example, SNA on IBM systems,DECnet on systems from Digital Equipment Corporation, and Microsoft-specific protocols (SMB) on Windows. Specific protocols for specific tasks may also be supported such as NFS for file access. Protocols like ESound, or esd can be easily extended over the network to provide sound from local applications, on a remote system's sound hardware.

Security

A computer being secure depends on a number of technologies working properly. A modern operating system provides access to a number of resources, which are available to software running on the system, and to external devices like networks via the kernel.

The operating system must be capable of distinguishing between requests which should be allowed to be processed, and others which should not be processed. While some systems may simply distinguish between "privileged" and "non-privileged", systems commonly have a form of requester *identity*, such as a user name.

To establish identity there may be a process of *authentication*. Often a username must be quoted, and each username may have a password. Other methods of authentication, such as magnetic cards or biometric data, might be used instead. In some cases, especially connections from the network, resources may be accessed with no authentication at all (such as reading files over a network share). Also covered by the concept of requester identity is *authorization*; the particular services and resources accessible by the requester once logged into a system are tied to either the requester's user account or to the variously configured groups of users to which the requester belongs.

In addition to the allow/disallow model of security, a system with a high level of security will also offer auditing options. These would allow tracking of requests for access to resources (such as, "who has been reading this file?"). Internal security, or security from an already running program is only possible if all possibly harmful requests must be carried out through interrupts to the operating system kernel. If programs can directly access hardware and resources, they cannot be secured.

External security involves a request from outside the computer, such as a login at a connected console or some

kind of network connection. External requests are often passed through device drivers to the operating system's kernel, where they can be passed onto applications, or carried out directly.

Security of operating systems has long been a concern because of highly sensitive data held on computers, both of a commercial and military nature. The United States Government Department of Defense (DoD) created the *Trusted Computer System Evaluation Criteria* (TCSEC) which is a standard that sets basic requirements for assessing the effectiveness of security. This became of vital importance to operating system makers, because the TCSEC was used to evaluate, classify and selecttrusted operating systems being considered for the processing, storage and retrieval of sensitive or classified information.

Network services include offerings such as file sharing, print services, email, web sites, and file transfer protocols (FTP), most of which can have compromised security. At the front line of security are hardware devices known as firewalls or intrusion detection/prevention systems. At the operating system level, there are a number of software firewalls available, as well as intrusion detection/prevention systems. Most modern operating systems include a software firewall, which is enabled by default. A software firewall can be configured to allow or deny network traffic to or from a service or application running on the operating system. Therefore, one can install and be running an insecure service, such as Telnet or FTP, and not have to be threatened by a security breach because the firewall would deny all traffic trying to connect to the service on that port.

An alternative strategy, and the only sandbox strategy available in systems that do not meet the Popek and Goldberg virtualization requirements, is the operating system not running user programs as native code, but instead either emulates a processor or provides a host for a p-code based system such as Java.

Internal security is especially relevant for multi-user systems; it allows each user of the system to have private files that the other users cannot tamper with or read. Internal security is also vital if auditing is to be of any use, since a program can potentially bypass the operating system, inclusive of bypassing auditing.

User Interface

Every computer that is to be operated by an individual requires a user interface. The user interface is not actually a part of the operating system—it generally runs in a separate program usually referred to as a shell, but is essential if human interaction is to be supported.

The user interface requests services from the operating system that will acquire data from input hardware devices, such as a keyboard, mouse or credit card reader, and requests operating system services to display prompts, status messages and such on output hardware devices, such as a video monitor or printer.

The two most common forms of a user interface have historically been the command-line interface, where computer commands are typed out line-by-line, and the graphical user interface, where a visual environment (most commonly a WIMP) is present.

Graphical user Interfaces

Most of the modern computer systems support graphical user interfaces (GUI), and often include them. In some computer systems, such as the original implementation of Mac OS, the GUI is integrated into the kernel.

While technically a graphical user interface is not an operating system service, incorporating support for one into the operating system kernel can allow the GUI to be more responsive by reducing the number of context switches required for the GUI to perform its output functions. Other operating systems are modular, separating the graphics subsystem from the kernel and the Operating System. In the 1980s UNIX, VMS and many others had operating systems that were built this way. GNU/Linux and Mac OS X are also built this way. Modern releases of Microsoft Windows such asWindows Vista implement a graphics subsystem that is mostly in user-space; however the graphics drawing routines of versions between Windows NT 4.0 and Windows Server 2003 exist mostly in kernel space.Windows 9x had very little distinction between the interface and the kernel.

Many computer operating systems allow the user to install or create any user interface they desire. The X Window System in conjunction with GNOME or KDE is a commonly found setup on most Unix and Unix-like (BSD, GNU/Linux, Solaris) systems. A number of Windows shell replacements have been released for Microsoft Windows, which offer alternatives to the included Windows shell, but the shell itself cannot be separated from Windows. Numerous Unix-

based GUIs have existed over time, most derived from X11. Competition among the various vendors of Unix (HP, IBM, Sun) led to much fragmentation, though an effort to standardize in the 1990s to COSE and CDE failed for various reasons, and were eventually eclipsed by the widespread adoption of GNOME and KDE. Prior to free software-based toolkits and desktop environments, Motif was the prevalent toolkit/desktop combination (and was the basis upon which CDE was developed).

Graphical user interfaces evolve over time. For example, Windows has modified its user interface almost every time a new major version of Windows is released, and the Mac OS GUI changed dramatically with the introduction of Mac OS X in 1999.

Real-time Operating Systems

A real-time operating system (RTOS) is a multitasking operating system intended for applications with fixed deadlines (real-time computing). Such applications include some small embedded systems, automobile engine controllers, industrial robots, spacecraft, industrial control, and some large-scale computing systems.

An early example of a large-scale real-time operating system was Transaction Processing Facility developed by American Airlines andIBM for the Sabre Airline Reservations System.

Embedded systems that have fixed deadlines use a real-time operating system such as VxWorks, PikeOS, eCos, QNX, MontaVista Linux and RTLinux. Windows CE is a real-time operating system that shares similar APIs to

desktop Windows but shares none of desktop Windows' codebase. Symbian OS also has an RTOS kernel (EKA2) starting with version 8.0b.

Some embedded systems use operating systems such as Palm OS, BSD, and GNU/Linux, although such operating systems do not support real-time computing.

Operating System Development as a Hobby

Operating system development is one of the most complicated activities in which a computing hobbyist may engage. A hobby operating system may be classified as one whose code has not been directly derived from an existing operating system, and has few users andactive developers.

In some cases, hobby development is in support of a "homebrew" computing device, for example, a simple single-board computerpowered by a 6502 microprocessor. Or, development may be for an architecture already in widespread use. Operating system development may come from entirely new concepts, or may commence by modeling an existing operating system. In either case, the hobbyist is his/her own developer, or may interact with a small and sometimes unstructured group of individuals who have like interests.

Examples of a hobby operating system include ReactOS and Syllable.

Diversity of Operating Systems and Portability

Application software is generally written for use on a specific operating system, and sometimes even for specific hardware. When porting the application to run on another OS, the functionality required by that application may be

implemented differently by that OS (the names of functions, meaning of arguments, etc.) requiring the application to be adapted, changed, or otherwise maintained.

This cost in supporting operating systems diversity can be avoided by instead writing applications against software platforms like Java, or Qt for web browsers. These abstractions have already borne the cost of adaptation to specific operating systems and their system libraries.

Another approach is for operating system vendors to adopt standards. For example, POSIX and OS abstraction layers provide commonalities that reduce porting costs.

2

Functions of Operating System

The operating system (sometimes referred to by its abbreviation *OS*), is responsible for creating the link between the material resources, the user and the applications (word processor, video game, etc.). When a programme wants to access a material resource, it does not need to send specific information to the peripheral device but it simply sends the information to the operating system, which conveys it to the relevant peripheral via its driver. If there are no drivers, each programme has to recognise and take into account the communication with each type of peripheral! The operating system thus allows the "dissociation" of programmes and hardware, mainly to simplify resource management and offer the user a simplified Man-machine interface (MMI) to overcome the complexity of the actual machine.

An operating system is a software component that acts as the core of a computer system. It performs various

functions and is essentially the interface that connects your computer and its supported components. In this article, we will discuss the basic functions of the operating system, along with security concerns for the most popular types. Drivers play a major role in the operating system. A driver is a programme designed to comprehend the functions of a particular device installed on the system.

The operating system performs other functions with system utilities that monitor performance, debug errors and maintain the system. It also includes a set of libraries often used by applications to perform tasks to enable direct interaction with system components. These common functions run seamlessly and are transparent to most users. Types of operating systems: There are several types of operating systems, with Windows, Linux and Macintosh suites being the most widely used.

The operating system has various roles:

- *Management of the processor*: The operating system is responsible for managing allocation of the processor between the different programmes using a scheduling algorithm. The type of scheduler is totally dependent on the operating system, according to the desired objective.
- *Management of the random access memory*: The operating system is responsible for managing the memory space allocated to each application and, where relevant, to each user. If there is insufficient physical memory, the operating system can create a memory zone on the hard drive, known as “virtual memory”.

The virtual memory lets you run applications requiring more memory than there is available RAM on the system. However, this memory is a great deal slower.

- *Management of input/output*: the operating system allows unification and control of access of programmes to material resources via drivers (also known as peripheral administrators or input/output administrators).
- *Management of execution of applications*: The operating system is responsible for smooth execution of applications by allocating the resources required for them to operate. This means an application that is not responding correctly can be "killed".
- *Management of authorisations*: The operating system is responsible for security relating to execution of programmes by guaranteeing that the resources are used only by programmes and users with the relevent authorisations.
- *File management*: The operating system manages reading and writing in the file system and the user and application file access authorisations.
- *Information management*: The operating system provides a certain number of indicators that can be used to diagnose the correct operation of the machine.

Charateristics of Operating System

The definition of an operating system is "the software that controls the hardware". However, today, due to microcode we need a better definition. We see an operating system as the programs that make the hardware useable.

In brief, an operating system is the set of programs that controls a computer. Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM. Controlling the computer involves software at several levels. We will differentiate kernel services, library services, and application-level services, all of which are part of the operating system. Processes run Applications, which are linked together with libraries that perform standard services. The kernel supports the processes by providing a path to the peripheral devices. The kernel responds to service calls from the processes and interrupts from the devices. The core of the operating system is the kernel, a control programme that functions in privileged state (an execution context that allows all hardware instructions to be executed), reacting to interrupts from external devices and to service requests and traps from processes. Generally, the kernel is a permanent resident of the computer. It creates and terminates processes and responds to their request for service.

Objectives

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

Hiding of Hardware

An abstraction is software that hides lower level details and provides a set of higher-level functions. An operating system transforms the physical world of devices, instructions,

memory, and time into virtual world that is the result of abstractions built by the operating system. There are several reasons for abstraction.

- *First*, the code needed to control peripheral devices is not standardized. Operating systems provide subroutines called device drivers that perform operations on behalf of programs for example, input/ output operations.
- *Second*, the operating system introduces new functions as it abstracts the hardware. For instance, operating system introduces the file abstraction so that programs do not have to deal with disks.
- *Third*, the operating system transforms the computer hardware into multiple virtual computers, each belonging to a different programme. Each programme that is running is called a process. Each process views the hardware through the lens of abstraction.
- *Fourth*, the operating system can enforce security through abstraction.

Manage Resources

An operating system controls how processes (the active agents) may access resources (passive entities).

Effective user Interface

The user interacts with the operating systems through the user interface and usually interested in the "look and feel" of the operating system. The most important components of the user interface are the command interpreter, the file system, on-line help, and application integration. The recent

trend has been towards increasingly integrated graphical user interfaces that encompass the activities of multiple processes on networks of computers.

Types of Operating Systems

As computers have progressed and developed so have the types of operating systems. Below is a basic list of the different types of operating systems and a few examples of operating systems that fall into each of the categories. Many computer operating systems will fall into more than one of the below categories.

GUI

Graphical User Interface, a GUI Operating System contains graphics and icons and is commonly navigated by using a computer mouse. Below are some examples of GUI Operating Systems.

System 7.x

Mac OS 9 is the latest public release of the Apple operating system, which includes new and unique features not found in any other operating system. Below are some of the new features found with this new operating system.

- *Sherlock* 2: Which offers the capability of quickly searching and purchasing online.
- 3*D acceleration*: Support for technologies such as OpenGL, allowing improved video and a wider gaming experience.
- *Share files*: Share files and folders over the Internet with other Mac users.

- *Colorsync* 3.0: Manages colour even more efficiently.
- *Synchronize*: Synchronizes with Palm computing products using HotSync software.
- *TCP/IP*: Provides access to TCP/IP networks.
- *Lock system*: Ensures that System Folders and Applications do not accidentally get reconfigured by having the capability of locking the system.

Windows 98

Microsoft Windows 98 is the upgrade to Microsoft Windows 95. While this was not as big as release as Windows 95, Windows 98 has significant updates, fixes and support for new peripherals. Below is a list of some of its new features.

- *Protection*: Windows 98 includes additional protection for important files on your computer such as backing up your registry automatically.
- *Improved support:* Improved support for new devices such as AGP, DirectX, DVD, USB, MMX,
- *FAT32*: Windows 98 has the capability of converting your drive to FAT32 without losing any information.
- *Interface*: Users of Windows 95 and NT will enjoy the same easy interface.
- *PnP*: Improved PnP support, to detect devices even better than Windows 95.
- *Internet Explorer 4.0*: Included Internet Explorer 4.0
- *Customizable Taskbar*: Windows adds many nice new features to the taskbar that 95 and NT do not have.
- *Includes Plus!*: Includes features only found in Microsoft Plus! free.

- *Active Desktop*: Includes Active Desktop that allows for users to customise their desktop with the look of the Internet.

Includes the same additional features as Windows 98; however, includes additional fixes and all of Year 2000 patches have been included in Windows 98 Second Edition. Below is a listing of the various new features Windows 98 SE includes.

Windows CE

Microsoft Windows CE 1.0 was originally released in 1996 to compete in the Palm Device Assistant Category. Windows CE, as shown below, has many of the same features as Windows 95.

In addition to the look of Windows 95, Windows CE also includes similar applications such as Pocket Excel, Pocket Word, and Pocket Power.

Multi-user

A multi-user operating system allows for multiple users to use the same computer at the same time and/or different times. See our multi-user dictionary definition for a complete definition for a complete definition. Below are some examples of multi-user operating systems.

Linux

Unix, which is not an acronym, was developed by some of the members of the Multics team at the bell labs starting in the late 1960's by many of the same people who helped create the Cprogramming language. The Unix today, however, is not just the work of a couple of programmers. Many other

organizations, institutes and various other individuals contributed significant additions to the system we now know today.

Unix

Unix, which is not an acronym, was developed by some of the members of the Multics team at the bell labs starting in the late 1960's by many of the same people who helped create the Cprogramming language. The Unix today, however, is not just the work of a couple of programmers. Many other organizations, institutes and various other individuals contributed significant additions to the system we now know today.

Windows 2000

Windows 2000 is based of the Windows NT Kernel and is sometimes referred to as Windows NT 5.0. Windows 2000 contains over 29 Million lines of code, mainly written in C++. 8 Million of those lines alone are written for drivers. Currently, Windows 2000 is by far one of the largest commercial projects ever built.

Some of the significant features of Windows 2000 Professional are:

- Support for FAT16, FAT32 and NTFS.
- Increased uptime of the system and significantly fewer OS reboot scenarios.
- Windows Installer tracks applications and recognizes and replaces missing components.
- Protects memory of individual apps and processes to avoid a single app bringing the system down.

- Encrypted File Systems protects sensitive data.
- Secure Virtual Private Networking (VPN) supports tunneling in to private LAN over public Internet.

Multithreading

Operating systems that allow different parts of a software programme to run concurrently.

Operating systems that would fall into this category are:

- Linux
- Unix
- Windows 2000.

Operating System

An operating system (OS) is a set of software that manages computer hardwareresources and provides common services for computer programs. The operating system is a vital component of the system software in a computer system. Application programs require an operating system to function.

Time-sharing operating systems schedule tasks for efficient use of the system and may also include accounting for cost allocation of processor time, mass storage, printing, and other resources.

For hardware functions such as input and output and memory allocation, the operating system acts as an intermediary between programs and the computer hardware, although the application code is usually executed directly by the hardware and will frequently make a system call to an OS function or be interrupted by it. Operating systems can be found on almost any device that contains a

computer—from cellular phones andvideo game consoles to supercomputers and web servers.

Examples of popular modern operating systems include Android, BSD, iOS, Linux, Mac OS X, Microsoft Windows, Windows Phone, and IBM z/OS. All these, except Windows and z/OS, share roots in UNIX.

Types

Real-time

A real-time operating system is a multitasking operating system that aims at executing real-time applications. Real-time operating systems often use specialized scheduling algorithms so that they can achieve a deterministic nature of behavior.

The main objective of real-time operating systems is their quick and predictable response to events. They have an event-driven or time-sharing design and often aspects of both. An event-driven system switches between tasks based on their priorities or external events while time-sharing operating systems switch tasks based on clock interrupts.

Multi-user

A multi-user operating system allows multiple users to access a computer system concurrently. Time-sharing system can be classified as multi-user systems as they enable a multiple user access to a computer through the sharing of time. Single-user operating systems, as opposed to a multi-user operating system, are usable by a single user at a time. Being able to use multiple accounts on a Windows operating system does not make it a multi-user

system. Rather, only the network administrator is the real user. But for a UNIX-like operating system, it is possible for two users to login at a time and this capability of the OS makes it a multi-user operating system.

Multi-tasking vs. Single-tasking

When only a single program is allowed to run at a time, the system is grouped under a single-tasking system. However, when the operating system allows the execution of multiple tasks at one time, it is classified as a multi-tasking operating system. Multi-tasking can be of two types: pre-emptive or co-operative. In pre-emptive multitasking, the operating system slices the CPU time and dedicates one slot to each of the programs.

Unix-like operating systems such as Solaris and Linux support pre-emptive multitasking, as does AmigaOS. Cooperative multitasking is achieved by relying on each process to give time to the other processes in a defined manner. 16-bit versions of Microsoft Windows used cooperative multi-tasking. 32-bit versions, both Windows NT and Win9x, used pre-emptive multi-tasking. Mac OS prior to OS X used to support cooperative multitasking.

Distributed

A distributed operating system manages a group of independent computers and makes them appear to be a single computer. The development of networked computers that could be linked and communicate with each other gave rise to distributed computing. Distributed computations are carried out on more than one machine. When computers in a group work in cooperation, they make a distributed system.

Embedded

Embedded operating systems are designed to be used in embedded computer systems. They are designed to operate on small machines like PDAs with less autonomy. They are able to operate with a limited number of resources. They are very compact and extremely efficient by design. Windows CE and Minix 3 are some examples of embedded operating systems.

Summary

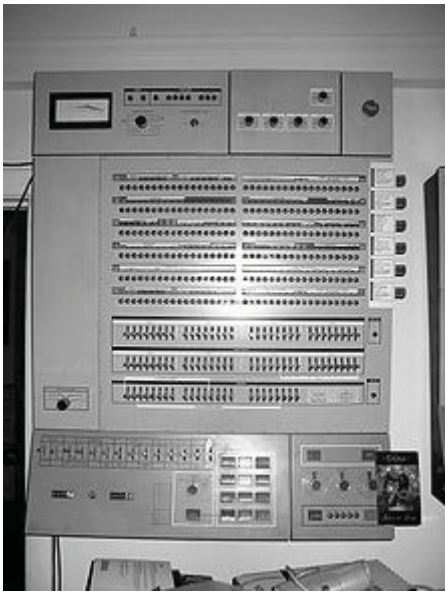
Early computers were built to perform a series of single tasks, like a calculator. Operating systems did not exist in their modern and more complex forms until the early 1960s. Basic operating system features were developed in the 1950s, such as resident monitorfunctions that could automatically run different programs in succession to speed up processing. Hardware features were added that enabled use of runtime libraries, interrupts, and parallel processing. When personal computers by companies such as Apple Inc., Atari,IBM and Amiga became popular in the 1980s, vendors included operating systems in them that had previously become widely used on mainframe and mini computers.

History

In the 1940s, the earliest electronic digital systems had no operating systems. Electronic systems of this time were so primitive compared to those of today that instructions were often entered into the system one bit at a time on rows of mechanical switches or by jumper wires on plug boards. These were special-purpose systems that, for example, generated ballistics tables for the military or controlled the printing of payroll checks from data on punched paper cards.

After programmable general purpose computers were invented, machine languages (consisting of strings of the binary digits 0 and 1 on punched paper tape) were introduced that sped up the programming process (Stern, 1981). In the early 1950s, a computer could execute only one program at a time. Each user had sole use of the computer for a limited period of time and would arrive at a scheduled time with program and data on punched paper cards and/or punched tape.

The program would be loaded into the machine, and the machine would be set to work until the program completed or crashed. Programs could generally be debugged via a front panel using toggle switches and panel lights. It is said that Alan Turing was a master of this on the early Manchester Mark 1 machine, and he was already deriving the primitive conception of an operating system from the principles of the Universal Turing machine.



OS/360 was used on most IBM mainframe computers beginning in 1966, including the computers that helped NASA put a man on the moon.

Later machines came with libraries of programs, which would be linked to a user's program to assist in operations such as input and output and generating computer code from human-readable symbolic code. This was the genesis of the modern-day computer system. However, machines still ran a single job at a time. At Cambridge University in England the job queue was at one time a washing line from which tapes were hung with different colored clothes-pegs to indicate job-priority.

Mainframes

Through the 1950s, many major features were pioneered in the field of operating systems, including batch processing, input/output interrupt, buffering, multitasking, spooling, runtime libraries, link-loading, and programs for sorting records in files.

These features were included or not included in application software at the option of application programmers, rather than in a separate operating system used by all applications. In 1959 the SHARE Operating System was released as an integrated utility for the IBM 704, and later in the 709 and 7090 mainframes, although it was quickly supplanted by IBSYS/IBJOB on the 709, 7090 and 7094.

During the 1960s, IBM's OS/360 introduced the concept of a single OS spanning an entire product line, which was crucial for the success of the System/360 machines. IBM's current mainframe operating systems are distant descendants of this original system and applications written for OS/360 can still be run on modern machines.

OS/360 also pioneered the concept that the operating system keeps track of all of the system resources that are used, including program and data space allocation in main memory and file space in secondary storage, and file locking during update. When the process is terminated for any reason, all of these resources are re-claimed by the operating system.

The alternative CP-67 system for the S/360-67 started a whole line of IBM operating systems focused on the concept of virtual machines.

Other operating systems used on IBM S/360 series mainframes included systems developed by IBM: COS/360 (Compatibility Operating System), DOS/360 (Disk Operating System), TSS/360 (Time Sharing System), TOS/360 (Tape Operating System), BOS/360 (Basic Operating System), and ACP (Airline Control Program), as well as a few non-IBM systems: MTS (Michigan Terminal System), MUSIC (Multi-User System for Interactive Computing), and ORVYL (Stanford Timesharing System).

Control Data Corporation developed the SCOPE operating system in the 1960s, for batch processing. In cooperation with the University of Minnesota, the Kronos and later the NOS operating systems were developed during the 1970s, which supported simultaneous batch and timesharing use. Like many commercial timesharing systems, its interface was an extension of the Dartmouth BASIC operating systems, one of the pioneering efforts in timesharing and programming languages. In the late 1970s, Control Data and the University of Illinois developed the PLATO operating system, which

used plasma panel displays and long-distance time sharing networks. Plato was remarkably innovative for its time, featuring real-time chat, and multi-user graphical games. Burroughs Corporation introduced the B5000in 1961 with the MCP, (Master Control Program) operating system. The B5000 was a stack machine designed to exclusively support high-level languages with no machine language or assembler, and indeed the MCP was the first OS to be written exclusively in a high-level language – ESPOL, a dialect of ALGOL. MCP also introduced many other ground-breaking innovations, such as being the first commercial implementation of virtual memory. During development of the AS400, IBM made an approach to Burroughs to licence MCP to run on the AS400 hardware. This proposal was declined by Burroughs management to protect its existing hardware production. MCPis still in use today in the Unisys ClearPath/MCP line of computers.

UNIVAC, the first commercial computer manufacturer, produced a series of EXEC operating systems. Like all early main-frame systems, this was a batch-oriented system that managed magnetic drums, disks, card readers and line printers. In the 1970s, UNIVAC produced the Real-Time Basic (RTB) system to support large-scale time sharing, also patterned after the Dartmouth BC system.

General Electric and MIT developed General Electric Comprehensive Operating Supervisor (GECOS), which introduced the concept of ringed security privilege levels. After acquisition by Honeywell it was renamed to General Comprehensive Operating System (GCOS).

Digital Equipment Corporation developed many operating systems for its various computer lines, including TOPS-10 and TOPS-20 time sharing systems for the 36-bit PDP-10 class systems. Prior to the widespread use of UNIX, TOPS-10 was a particularly popular system in universities, and in the early ARPANET community.

In the late 1960s through the late 1970s, several hardware capabilities evolved that allowed similar or ported software to run on more than one system. Early systems had utilized microprogramming to implement features on their systems in order to permit different underlying computer architectures to appear to be the same as others in a series. In fact most 360s after the 360/40 (except the 360/165 and 360/168) were microprogrammed implementations. But soon other means of achieving application compatibility were proven to be more significant.

The enormous investment in software for these systems made since 1960s caused most of the original computer manufacturers to continue to develop compatible operating systems along with the hardware. The notable supported mainframe operating systems include:

- Burroughs MCP – B5000, 1961 to Unisys Clearpath/MCP, present.
- IBM OS/360 – IBM System/360, 1966 to IBM z/OS, present.
- IBM CP-67 – IBM System/360, 1967 to IBM z/VM, present.
- UNIVAC EXEC 8 – UNIVAC 1108, 1967, to OS 2200 Unisys Clearpath Dorado, present.

Microcomputers

Mac OS by Apple Computer became the first widespread OS to feature a graphical user interface. Many of its features such as windows and icons would later become commonplace in GUIs.

The first microcomputers did not have the capacity or need for the elaborate operating systems that had been developed for mainframes and minis; minimalistic operating systems were developed, often loaded from ROM and known as *monitors*.

One notable early disk operating system was CP/M, which was supported on many early microcomputers and was closely imitated by Microsoft's MS-DOS, which became wildly popular as the operating system chosen for the IBM PC (IBM's version of it was called IBM DOS or PC DOS). In the '80s, Apple Computer Inc. (now Apple Inc.) abandoned its popular Apple II series of microcomputers to introduce the Apple Macintosh computer with an innovative Graphical User Interface (GUI) to the Mac OS.

The introduction of the Intel 80386 CPU chip with 32-bit architecture and pagingcapabilities, provided personal computers with the ability to run multitasking operating systems like those of earlier minicomputers and mainframes. Microsoft responded to this progress by hiring Dave Cutler,

who had developed the VMS operating system forDigital Equipment Corporation. He would lead the development of the Windows NToperating system, which continues to serve as the basis for Microsoft's operating systems line. Steve Jobs, a co-founder of Apple Inc., started NeXT Computer Inc., which developed the Unix-like NEXTSTEP operating system. NEXTSTEP would later be acquired by Apple Inc. and used, along with code from FreeBSD as the core of Mac OS X.

The GNU Project was started by activist and programmer Richard Stallman with the goal of a complete free software replacement to the proprietary UNIX operating system. While the project was highly successful in duplicating the functionality of various parts of UNIX, development of the GNU Hurd kernel proved to be unproductive. In 1991, Finnish computer science student Linus Torvalds, with cooperation from volunteers collaborating over the Internet, released the first version of the Linux kernel.

It was soon merged with the GNU user space components and system software to form a complete operating system. Since then, the combination of the two major components has usually been referred to as simply "Linux" by the software industry, a naming convention that Stallman and the Free Software Foundation remain opposed to, preferring the name GNU/Linux. The Berkeley Software Distribution, known as BSD, is the UNIX derivative distributed by the University of California, Berkeley, starting in the 1970s. Freely distributed and ported to many minicomputers, it eventually also gained a following for use on PCs, mainly as FreeBSD, NetBSD and OpenBSD.

Examples of operating systems

UNIX and UNIX-like Operating Systems

Ken Thompson wrote B, mainly based on BCPL, which he used to write Unix, based on his experience in the MULTICS project. B was replaced by C, and Unix developed into a large, complex family of inter-related operating systems which have been influential in every modern operating system.

The *UNIX-like* family is a diverse group of operating systems, with several major sub-categories including System V, BSD, and GNU/Linux. The name "UNIX" is a trademark of The Open Group which licenses it for use with any operating system that has been shown to conform to their definitions. "UNIX-like" is commonly used to refer to the large set of operating systems which resemble the original UNIX.

Unix-like systems run on a wide variety of computer architectures. They are used heavily for servers in business, as well as workstations in academic and engineering environments. Free UNIX variants, such asGNU/Linux and BSD, are popular in these areas.

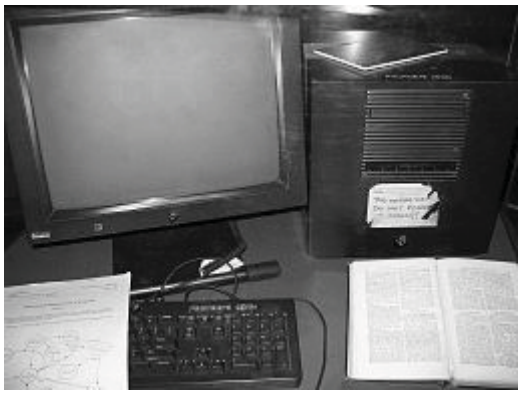
Four operating systems are certified by the The Open Group (holder of the Unix trademark) as Unix. HP's HP-UX and IBM's AIX are both descendants of the original System V Unix and are designed to run only on their respective vendor's hardware. In contrast, Sun Microsystems's Solaris Operating System can run on multiple types of hardware, including x86 and Sparc servers, and PCs. Apple'sMac OS X, a replacement for Apple's earlier (non-Unix) Mac OS, is

a hybrid kernel-based BSD variant derived from NeXTSTEP, Mach, and FreeBSD.

Unix interoperability was sought by establishing the POSIX standard. The POSIX standard can be applied to any operating system, although it was originally created for various Unix variants.

BSD and its Descendants

A subgroup of the Unix family is the Berkeley Software Distribution family, which includes FreeBSD, NetBSD, and OpenBSD, PC-BSD. These operating systems are most commonly found on webservers, although they can also function as a personal computer OS. The Internet owes much of its existence to BSD, as many of the protocols now commonly used by computers to connect, send and receive data over a network were widely implemented and refined in BSD. The world wide web was also first demonstrated on a number of computers running an OS based on BSD calledNextStep.



The first server for the World Wide Web ran on NeXTSTEP, based on BSD.

BSD has its roots in Unix. In 1974, University of California, Berkeley installed its first Unix system. Over time, students

and staff in the computer science department there began adding new programs to make things easier, such as text editors. When Berkely received new VAX computers in 1978 with Unix installed, the school's undergraduates modified Unix even more in order to take advantage of the computer's hardware possibilities. The Defense Advanced Research Projects Agency of the US Department of Defense took interest, and decided to fund the project. Many schools, corporations, and government organizations took notice and started to use Berkeley's version of Unix instead of the official one distributed by AT&T.

Steve Jobs, upon leaving Apple Inc. in 1985, formed NeXT Inc., a company that manufactured high-end computers running on a variation of BSD called NeXTSTEP. One of these computers was used by Tim Berners-Lee as the first webserver to create the World Wide Web.

Developers like Keith Bostic encouraged the project to replace any non-free code that originated with Bell Labs. Once this was done, however, AT&T sued. Eventually, after two years of legal disputes, the BSD project came out ahead and spawned a number of free derivatives, such as FreeBSD and NetBSD.

Mac OS X

The standard user interface of Mac OS X

Mac OS X is a line of open core graphical operating systems developed, marketed, and sold by Apple Inc., the latest of which is pre-loaded on all currently shippingMacintosh computers.

Mac OS X is the successor to the original Mac OS, which had been Apple's primary operating system since 1984. Unlike its predecessor, Mac OS X is a UNIX operating system built on technology that had been developed at NeXTthrough the second half of the 1980s and up until Apple purchased the company in early 1997.

The operating system was first released in 1999 as Mac OS X Server 1.0, with a desktop-oriented version (Mac OS X v10.0 "Cheetah") following in March 2001.

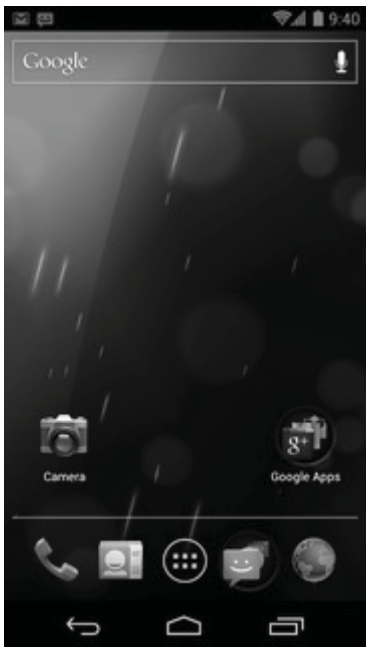
Since then, six more distinct "client" and "server" editions of Mac OS X have been released, the most recent being OS X 10.8 "Mountain Lion", which was first made available on February 16, 2012 for developers, and to be released to the public late summer 2012. Releases of Mac OS X are named after big cats.

The server edition, Mac OS X Server, is architecturally identical to its desktop counterpart but usually runs on Apple's line of Macintoshserver hardware. Mac OS X Server includes work group management and administration software tools that provide simplified access to key network services, including a mail transfer agent, a Samba server, an LDAP server, a domain name server, and others.

In Mac OS X v10.7 Lion, all server aspects of Mac OS X Server have been integrated into the client version.

Linux and GNU

Ubuntu, desktop Linux distribution



Android, a popular mobile operating system using the Linux kernel

Linux (or GNU/Linux) is a Unix-like operating system that was developed without any actual Unix code, unlike BSD and its variants. Linux can be used on a wide range of devices from supercomputers to wristwatches. The Linux kernelis released under an open source license, so anyone can read and modify its code.

It has been modified to run on a large variety of electronics. Although estimates suggest that Linux is used on 1.82% of all personal computers, it has been widely adopted for use in servers and embedded systems (such as cell phones). Linux has superseded Unix in most places, and is used on the 10 most powerful supercomputers in the world. The Linux kernel is used in some popular distributions, such as Red Hat, Debian, Ubuntu, Linux Mint and Google's Android.

The GNU project is a mass collaboration of programmers who seek to create a completely free and open operating system that was similar to Unix but with completely original code. It was started in 1983 by Richard Stallman, and is responsible for many of the parts of most Linux variants.

Thousands of pieces of software for virtually every operating system are licensed under the GNU General Public License. Meanwhile, the Linux kernel began as a side project of Linus Torvalds, a university student from Finland. In 1991, Torvalds began work on it, and posted information about his project on a newsgroup for computer students and programmers. He received a wave of support and volunteers who ended up creating a full-fledged kernel. Programmers from GNU took notice, and members of both projects worked to integrate the finished GNU parts with the Linux kernel in order to create a full-fledged operating system.

Google Chrome OS

Chrome is an operating system based on the Linux kernel and designed by Google. Since Chrome OS targets

computer users who spend most of their time on the Internet, it is mainly a web browser with no ability to run applications. It relies on Internet applications(or Web apps) used in the web browser to accomplish tasks such as word processing and media viewing, as well as online storage for storing most files.

3

Linux and other Operating Systems

It is important to understand the differences between Linux and other operating systems, like MS-DOS, OS/2, and the other implementations of UNIX for personal computers. First of all, Linux coexists happily with other operating systems on the same machine: you can run MS-DOS and OS/2 along with Linux on the same system without problems. There are even ways to interact between various operating systems, as we'll see.

Why use Linux

Why use Linux, instead of a well known, well tested, and well documented commercial operating system? We could give you a thousand reasons. One of the most important, however, is that Linux is an excellent choice for personal UNIX computing. If you're a UNIX software developer, why use MS-DOS at home? Linux allows you to develop and test

UNIX software on your PC, including database and X Window System applications. If you're a student, chances are that your university computing systems run UNIX. You can run your own UNIX system and tailor it to your needs. Installing and running Linux is also an excellent way to learn UNIX if you don't have access to other UNIX machines. But let's not lose sight. Linux isn't only for personal UNIX users. It is robust and complete enough to handle large tasks, as well as distributed computing needs. Many businesses--especially small ones--have moved their systems to Linux in lieu of other UNIX based, workstation environments. Universities have found that Linux is perfect for teaching courses in operating systems design. Large, commercial software vendors have started to realise the opportunities which a free operating system can provide.

Linux vs. MS-DOS

It's not uncommon to run both Linux and MS-DOS on the same system. Many Linux users rely on MS-DOS for applications like word processing. Linux provides its own analogs for these applications, but you might have a good reason to run MS-DOS as well as Linux. If your dissertation is written using WordPerfect for MS-DOS, you may not be able to convert it easily to TeX or some other format. Many commercial applications for MS-DOS aren't available for Linux yet, but there's no reason that you can't use both.

MS-DOS does not fully utilize the functionality of 80386 and 80486 processors. On the other hand, Linux runs completely in the processor's protected mode, and utilizes all of its features. You can directly access all of your available

memory (and beyond, with virtual RAM). Linux provides a complete UNIX interface which is not available under MS-DOS. You can easily develop and port UNIX applications to Linux, but under MS-DOS you are limited to a subset of UNIX functionality.

Linux and MS-DOS are different entities. MS-DOS is inexpensive compared to other commercial operating systems and has a strong foothold in the personal computer world. No other operating system for the personal computer has reached the level of popularity of MS-DOS, because justifying spending $1,000 for other operating systems alone is unrealistic for many users. Linux, however, is free, and you may finally have the chance to decide for yourself. You can judge Linux vs. MS-DOS based on your expectations and needs. Linux is not for everybody. If you always wanted to run a complete UNIX system at home, without the high cost of other UNIX implementations for personal computers, Linux may be what you're looking for.

Linux vs. The Other Guys

A number of other advanced operating systems have become popular in the PC world. Specifically, IBM's OS/2 and Microsoft Windows have become popular for users upgrading from MS-DOS. Both OS/2 and Windows NT are full featured multitasking operating systems, like Linux. OS/2, Windows NT, and Linux support roughly the same user interface, networking, and security features. However, the real difference between Linux and The Other Guys is the fact that Linux is a version of UNIX, and benefits from contributions of the UNIX community at large.

What makes UNIX so important? Not only is it the most popular operating system for multiuser machines, it is a foundation of the free software world. Much of the free software available on the Internet is written specifically for UNIX systems.

There are many implementations of UNIX from many vendors. No single organization is responsible for its distribution. There is a large push in the UNIX community for standardization in the form of open systems, but no single group controls this design. Any vendor (or, as it turns out, any hacker) may develop a standard implementation of UNIX. OS/2 and Microsoft operating systems, on the other hand, are proprietary. The interface and design are controlled by a single corporation, which develops the operating system code. In one sense, this kind of organization is beneficial because it sets strict standards for programming and user interface design, unlike those found even in the open systems community.

Several organizations have attempted the difficult task of standardizing the UNIX programming interface. Linux, in particular, is mostly compliant with the POSIX.1 standard. As time goes by, it is expected that the Linux system will adhere to other standards, but standardization is not the primary goal of Linux development.

Linux vs. Other Implementations of UNIX

Several other implementations of UNIX exist for 80386 or better personal computers. The 80386 architecture lends itself to UNIX, and vendors have taken advantage of this.

Oher implementations of UNIX for the personal computer are similar to Linux. Almost all commercial versions of UNIX support roughly the same software, programming environment, and networking features. However, there are differences between Linux and commercial versions of UNIX.

Linux supports a different range of hardware than commercial implementations. In general, Linux supports most well-known hardware devices, but support is still limited to hardware which the developers own. Commercial UNIX vendors tend to support more hardware at the outset, but the list of hardware devices which Linux supports is expanding continuously. We'll cover the hardware requirements for Linux in Section.

Many users report that Linux is at least as stable as commercial UNIX systems. Linux is still under development, but the two-pronged release philosophy has made stable versions available without impeding development.

The most important factor for many users is price. Linux software is free if you can download it from the Internet or another computer network. If you do not have Internet access, you can still purchase Linux inexpensively via mail order on diskette, tape, or CD-ROM.

Of course, you may copy Linux from a friend who already has the software, or share the purchase cost with someone else. If you plan to install Linux on a large number of machines, you need only purchase a single copy of the software--Linux is not distributed with a ``single machine'' license.

The value of commercial UNIX implementations should not be demeaned. In addition to the price of the software itself, one often pays for documentation, support, and quality assurance. These are very important factors for large institutions, but personal computer users may not require these benefits. In any case, many businesses and universities have found that running Linux in a lab of inexpensive personal computers is preferable to running a commercial version of UNIX in a lab of workstations. Linux can provide workstation functionality on a personal computer at a fraction of the cost.

Linux systems have travelled the high seas of the North Pacific, and manage telecommunications and data analysis for an oceanographic research vessel. Linux systems are used at research stations in Antarctica. Several hospitals maintain patient records on Linux systems.

Other free or inexpensive implementations of UNIX are available for the 80386 and 80486. One of the best known is 386BSD, an implementation of BSD UNIX for the 80386. The 386BSD package is comparable to Linux in many ways, but which one is better depends on your needs and expectations. The only strong distinction we can make is that Linux is developed openly, and any volunteer can aid in the development process, while 386BSD is developed by a closed team of programmers. Because of this, serious philosophical and design differences exist between the two projects. The goal of Linux is to develop a complete UNIX system from scratch (and have a lot of fun in the process), and the goal of 386BSD is in part to modify the existing BSD code for use on the 80386.

NetBSD is another port of the BSD NET/2 distribution to several machines, including the 80386. NetBSD has a slightly more open development structure, and is comparable to 386BSD in many respects.

Another project of note is HURD, an effort by the Free Software Foundation to develop and distribute a free version of UNIX for many platforms. Contact the Free Software Foundation (the address is given in Appendix C) for more information about this project. At the time of this writing, HURD is still under development.

Other inexpensive versions of UNIX exist as well, like Minix, an academic but useful UNIX clone upon which early development of Linux was based. Some of these implementations are mostly of academic interest, while others are full fledged systems.

Hardware Requirements

You must be convinced by now of how wonderful Linux is, and of all the great things it can do for you. However, before you rush out and install Linux, you need to be aware of its hardware requirements and limitations.

Keep in mind that Linux is developed by users. This means, for the most part, that the hardware supported by Linux is that which the users and developers have access to. As it turns out, most popular hardware and peripherals for personal computers are supported. Linux supports more hardware than some commercial implementations of UNIX. However, some obscure devices aren't supported yet.

Another drawback of hardware support under Linux is that many companies keep their hardware interfaces

proprietary. Volunteer Linux developers can't write drivers for the devices because the manufacturer does not make the technical specifications public. Even if Linux developers could develop drivers for proprietary devices, they would be owned by the company which owns the device interface, which violates the GPL. Manufacturers that maintain proprietary interfaces write their own drivers for operating systems like MS-DOS and Microsoft Windows. Users and third-party developers never need to know the details of the interface.

In some cases, Linux programmers have attempted to write hackish device drivers based on assumptions about the interface. In other cases, developers work with the manufacturer and try to obtain information about the device interface, with varying degrees of success.

In the following sections, we attempt to summarize the hardware requirements for Linux. The Linux Hardware HOWTO contains a more complete listing of hardware supported by Linux.

Disclaimer: Much hardware support for Linux is in the development stage. Some distributions may or may not support experimental features. This section lists hardware which has been supported for some time and is known to be stable. When in doubt, consult the documentation of your Linux distribution. for more information about Linux distributions.

Linux is available for many platforms in addition to Intel 80x86 systems. These include Macintosh, Amiga, Sun SparcStation, and Digital Equipment Corporation Alpha

based systems. In this book, however, we focus on garden-variety Intel 80386, 80486, and Pentium processors, and clones by manufacturers like AMD, Cyrix, and IBM.

Motherboard and CPU Requirements

Linux currently supports systems with the Intel 80386, 80486, or Pentium CPU, including all variations like the 80386SX, 80486SX, 80486DX, and 80486DX2. Non-Intel clones work with Linux as well. Linux has also been ported to the DEC Alpha and the Apple PowerMac.

If you have an 80386 or 80486SX, you may also wish to use a math coprocessor, although one isn't required. The Linux kernel can perform FPU emulation if the machine doesn't have a coprocessor. All standard FPU couplings are supported, including IIT, Cyrix FasMath, and Intel.

Most common PC motherboards are based on the PCI bus but also offer ISA slots. This configuration is supported by Linux, as are EISA and VESA-bus systems. IBM's MicroChannel (MCA) bus, found on most IBM PS/2 systems, is significantly different, and support has been recently added.

Memory Requirements

Linux requires very little memory, compared to other advanced operating systems. You should have 4 megabytes of RAM at the very least, and 16 megabytes is strongly recommended. The more memory you have, the faster the system will run. Some distributions require more RAM for installation.

Linux supports the full 32-bit address range of the processor. In other words, it uses all of your RAM

automatically. Linux will run with only 4 megabytes of RAM, including bells and whistles like the X Window System and emacs.

However, having more memory is almost as important as having a faster processor. For general use, 16 megabytes is enough, and 32 megabytes, or more, may be needed for systems with a heavy user load.

Most Linux users allocate a portion of their hard drive as swap space, which is used as virtual RAM. Even if your machine has more than 16 megabytes of physical RAM, you may wish to use swap space. It is no replacement for physical RAM, but it can let your system run larger applications by swapping inactive portions of code to disk. The amount of swap space that you should allocate depends on several factors;

Hard Drive Controller Requirements

It is possible to run Linux from a floppy diskette, or, for some distributions, a live file system on CD-ROM, but for good performance you need hard disk space. Linux can co-exist with other operating systems--it only needs one or more disk partitions. Linux supports all IDE and EIDE controllers as well as older MFM and RLL controllers. Most, but not all, ESDI controllers are supported. The general rule for non-SCSI hard drive and floppy controllers is that if you can access the drive from MS-DOS or another operating system, you should be able to access it from Linux. Linux also supports a number of popular SCSI drive controllers. This includes most Adaptec and Buslogic cards as well as cards based on the NCR chip sets.

Hard Drive Space Requirements

Of course, to install Linux, you need to have some amount of free space on your hard drive. Linux will support more than one hard drive on the same machine; you can allocate space for Linux across multiple drives if necessary.

How much hard drive space depends on your needs and the software you're installing. Linux is relatively small, as UNIX implementations go. You could run a system in 20 megabytes of disk space. However, for expansion and larger packages like X, you need more space. If you plan to let more than one person use the machine, you need to allocate storage for their files. Realistic space requirements range from 200 megabytes to one gigabyte or more. Each Linux distribution comes with literature to help you gauge the precise amount of storage required for your software configuration. Look at the information which comes with your distribution or the appropriate installation section in.

Monitor and Video Adaptor Requirements

Linux supports standard Hercules, CGA, EGA, VGA, IBM monochrome, Super VGA, and many accelerated video cards, and monitors for the default, text-based interface. In general, if the video card and monitor work under an operating system like MS-DOS, the combination should work fine under Linux. However, original IBM CGA cards suffer from ``snow'' under Linux, which is not pleasant to view. Graphical environments like X have video hardware requirements of their own. Rather than list them here, we relegate that discussion to Section Popular video cards are supported and new card support is added regularly.

Miscellaneous Hardware

You may also have devices like a CD-ROM drive, mouse, or sound card, and may be interested in whether or not this hardware is supported by Linux.

Mice and Other Pointing Devices

Typically, a mouse is used only in graphical environments like X. However, several Linux applications that are not associated with a graphical environment also use mice. Linux supports standard serial mice like Logitech, MM series, Mouseman, Microsoft (2-button), and Mouse Systems (3-button). Linux also supports Microsoft, Logitech, and ATIXL bus mice, and the PS/2 mouse interface. Pointing devices that emulate mice, like trackballs and touchpads, should work also.

CD-ROM Drives

Many common CD-ROM drives attach to standard IDE controllers. Another common interface for CD-ROM is SCSI. SCSI support includes multiple logical units per device so you can use CD-ROM ``jukeboxes.'' Additionally, a few proprietary interfaces, like the NEC CDR-74, Sony CDU-541 and CDU-31a, Texel DM-3024, and Mitsumi are supported. Linux supports the standard ISO 9660 file system for CD-ROMs, and the High Sierra file system extensions.

Tape Drives

Any SCSI tape drive, including quarter inch, DAT, and 8MM are supported, if the SCSI controller is supported. Devices that connect to the floppy controller like floppy tape drives are supported as well, as are some other interfaces, like QIC-02.

Printers

Linux supports the complete range of parallel printers. If MS-DOS or some other operating system can access your printer from the parallel port, Linux should be able to access it, too. Linux printer software includes the UNIX standard lp and lpr software. This software allows you to print remotely via a network, if you have one. Linux also includes software that allows most printers to handle PostScript files.

Modems

As with printer support, Linux supports the full range of serial modems, both internal and external. A great deal of telecommunications software is available for Linux, including Kermit, pcomm, minicom, and seyon. If your modem is accessible from another operating system on the same machine, you should be able to access it from Linux with no difficulty.

Ethernet Cards

Many popular Ethernet cards and LAN adaptors are supported by Linux. Linux also supports some FDDI, frame relay, and token ring cards, and all Arcnet cards. A list of supported network cards is included in the kernel source of your distribution.

Sources of Linux Information

Many other sources of information about Linux are available. In particular, a number of books about UNIX in general will be of use, especially for readers unfamiliar with UNIX. We suggest that you peruse one of these books before

attempting to brave the jungles of Linux. Information is also available online in electronic form. You must have access to an online network like the Internet, Usenet, or Fidonet to access the information. A good place to start is If you do not, you might be able to find someone who is kind enough to give you hard copies of the documents.

Online Documents

Many Linux documents are available via anonymous FTP from Internet archive sites around the world and networks like Fidonet and CompuServe. Linux CD-ROM distributions also contain the documents mentioned here. If you are can send mail to Internet sites, you may be able to retrieve these files using one of the FTP e-mail servers that mail you the documents or files from the FTP sites. for more information on using FTP e-mail servers.

Linux on World Wide Web

The Linux Documentation Project Home Page is on the World Wide Web at http://sunsite.unc.edu/LDP This web page lists many HOWTOs and other documents in HTML format, as well as pointers to other sites of interest to Linux users, like ssc.com, home of the Linux Journal, a monthly magazine.

Comparison of Linux with MS-DOS and Windows

Linux Vs MS-DOS

DOS was the first operating system I learned to use. I remember a test by my tutor in which one had to create a hierarchical set of directories. This probably sounds trivial to anyone familiar with graphical user interfaces, but then

the scenario was very different. It even looks ancient by today's standards. Windows 95 hadn't been released at that time, and Linux was unheard of in India. I liked DOS much better than Windows 3.1, which looked quite flimsy and unstable, not to mention that it was a big memory hog. DOS is quite different from Linux in many ways.

DOS does not provide any graphical user interface and you have to learn at least a dozen commands with its numerous options to do some basic tasks like copying a file or moving between the directories. Even a minor spelling mistake can result in a "Bad command or file name" error.

DOS does not support the concept of multi-users; each and every user has to customise the system according to his need every time he wants to work on it. It was also not a multitasking system. This meant that you could not check out the value of a calculation when typing a letter without closing that application first. DOS also does not have any in built security features. This was acceptable as long as you did not want a networking system. There was other variants of MS-DOS, like PC-DOS from IBM and some others, which tried to add the missing features. Some of the deficiencies have been resolved using third party utilities but basic limitations like the arcane 640kb-memory limit and single-tasking were not acceptable to many.

Now in case you are wondering why anyone would care to use DOS, I will point out some advantages. Basically DOS has had very different goals from that of Linux. It was a very cheap system (as far as cost is concerned) and it was quite usable with its minimalist set of features. It was a

simple system to work with. There weren't too many complications to worry about if you didn't want to develop anything on it. It was arguably the world's most popular operating system and it had a comfortable number of applications for common tasks.

Files and Directories

The files in Linux can be very long, up-to 255 characters like Windows, and they do not always have extensions. The executable files are identified through an attribute rather than the extension. File extensions are less important to Linux than for DOS and Windows, since Linux usually identifies files by a unique identification code called the magic number that depends on the file type. Directories are similar to that of DOS and follow a hierarchical structure. The path names are separated by forward slashes (/)in Linux whereas DOS and Windows uses back slashes (\). *For example*:

```
% cd /mnt/cdrom
```

A / denotes the root and .. stands for the parent directory, similar to DOS.

In bash shells, the ~ symbol maybe used to jump to the home directory quickly. *For example*:

```
% cd ~
```

Linux Shell

Several of the DOS commands have Linux equivalents. The Linux shell is similar to the DOS command line but is far more powerful, and I found that it was also more workable with features like colour highlighting and friendlier navigation capabilities, depending on the particular shell

you are using. Most Linux distributions come with the Bash (Bourne Again SHell) as the default. There are several others, like the Korn shell and the C shell. They are usually similar. It's recommended that you learn to work with one shell completely before trying out the others. Things usually get complicated if you want to run shell scripts, which are similar to batch files (files with the .bat extension) under DOS.

Running DOS Programmes Under Linux

There is a DOS emulator called dosemu www.dosemu.org for Linux that is capable of running DOS programmes under the Linux operating system. This software is still under development; you may wish to try it out though. It is known to be fairly usable at least for some applications. If you are looking for Linux just to use DOS programmes for free then try using FreeDOS www.freedos.org. That should be much better than Linux.

Linux vs. Windows

As I have said before Windows is more or less similar to Linux. When people are introduced to Linux they are at first intimidated by the system. It has different kinds of graphical interfaces and things don't always work as they are expected to.

When users look at me in a puzzled manner I demonstrate in some easy ways how common tasks like changing the desktop wallpaper or playing a song is similar to Windows. The problem with this kind of approach is that the users complain very soon that Linux doesn't offer them much more than Windows does :-).

I agree with them to a certain extent on this. There are some limitations to what you can expect from an operating system. You just can't expect Linux to work like a 3D-shooter game or something. Of course, there are many differences in the shell, the choice of user interfaces and the philosophy and goals of the operating system. Linux is developed as a open system in which the source code of the core Linux system (kernel) is available for anyone for free but how this could affect the end user is difficult to explain initially.

The user interface is probably the first thing you notice when you begin to use the Linux system. Windows offers a single, monolithic user interface, which is more or less the same across all the versions. In contrast, Linux has two major desktop environments called KDE and Gnome. KDE has a built-in window manager, while Gnome is supported by many, such as Sawfish and Enlightenment. The decision of choosing one among the desktop environments and windows managers is left to you. Some of them can run efficiently in a system with low amounts of memory and some of them are designed to look like a game console. KDE would be more similar to Windows, and Gnome with the Enlightenment window manager was fancy enough for me. Try out some of the popular ones before making the decision. Let's take a look at Windows in more detail so that you can clearly make out the differences.

Windows 9x Series

Before Windows 95 was released, all versions of Windows until version 3.1 were graphical platforms on top of DOS.

This offered limited capability for multitasking and the Programme Manager interface was cluttered with no distinct hierarchy. Windows 95 was a 32-bit operating system and a major improvement in user interface with its "Desktop" concept adapted from the Macintosh user interface.

It also offered limited compatibility with previous versions of Windows and DOS. Stability was also improved Windows 98 and Windows ME offered some more features though nothing major was added. The more recent version called Windows XP is considerably more stable due to incorporating the Windows 2000 kernel, and is comparatively friendlier and easier due to an attractive interface.

Windows NT Series

Windows NT is considerably stable but demands more resources. It supports the Intel architecture, and at one time the Digital alpha and MIPS processors, but I believe those have been dropped now. It managed to replace UNIX in small-scale networks due to the similarity to the popular Windows 95 interface. The latest incarnation called Windows 2000 provides a few more administrative utilities and services.

4

Modern Network Devices and Operating System

Modern network devices are complex entities composed of both silicon and software. Thus, designing an efficient hardware platform is not, by itself, sufficient to achieve an effective, cost-efficient and operationally tenable product. The control plane plays a critical role in the development of features and in ensuring device usability.

Although progress from the development of faster CPU boards and forwarding planes is visible, structural changes made in software are usually hidden, and while vendor collateral often offers a list of features in a carrier-class package, operational experiences may vary considerably. Products that have been through several generations of software releases provide the best examples of the difference made by the choice of OS. It is still not uncommon to find routers or switches that started life under older, monolithic

software and later migrated to more contemporary designs. The positive effect on stability and operational efficiency is easy to notice and appreciate.

However, migration from one network operating system to another can pose challenges from non-overlapping feature sets, noncontiguous operational experiences and inconsistent software quality. These potential challenges make it is very desirable to build a control plane that can power the hardware products and features supported in both current and future markets.

Developing a flexible, long-lasting and high-quality network OS provides a foundation that can gracefully evolve to support new needs in its height for up and down scaling, width for adoption across many platforms, and depth for rich integration of new features and functions. It takes time, significant investment and in-depth expertise.

Most of the engineers writing the early releases of Junos OS came from other companies where they had previously built network software. They had firsthand knowledge of what worked well, and what could be improved.

These engineers found new ways to solve the limitations that they'd experienced in building the older operating systems.

Resulting innovations in Junos OS are significant and rooted in its earliest design stages. Still, to ensure that our products anticipate and fulfil the next generation of market requirements, Junos OS is periodically reevaluated to determine whether any changes are needed to ensure that it continues to provide the reliability, performance and resilience for which it is known.

Contemporary network operating systems are mostly advanced and specialized branches of POSIX-compliant software platforms and are rarely developed from scratch. The main reason for this situation is the high cost of developing a world-class operating system all the way from concept to finished product. By adopting a general purpose OS architecture, network vendors can focus on routing-specific code, decrease time to market, and benefit from years of technology and research that went into the design of the original (donor) products.

First-Generation OS: Monolithic Architecture

Typically, first-generation network operating systems for routers and switches were proprietary images running in a flat memory space, often directly from flash memory or ROM. While supporting multiple processes for protocols, packet handling and management, they operated using a cooperative, multitasking model in which each process would run to completion or until it voluntarily relinquished the CPU.

All first-generation network operating systems shared one trait: They eliminated the risks of running full-size commercial operating systems on embedded hardware. Memory management, protection and context switching were either rudimentary or nonexistent, with the primary goals being a small footprint and speed of operation.

Nevertheless, first-generation network operating systems made networking commercially viable and were deployed on a wide range of products. The downside was that these systems were plagued with a host of problems associated with resource management and fault isolation; a single runaway process could easily consume the processor or

cause the entire system to fail. Such failures were not uncommon in the data networks controlled by older software and could be triggered by software errors, rogue traffic and operator errors.

Legacy platforms of the first generation are still seen in networks worldwide, although they are gradually being pushed into the lowest end of the telecom product lines.

Second-Generation OS: Control Plane Modularity

The mid-1990s were marked by a significant increase in the use of data networks worldwide, which quickly challenged the capacity of existing networks and routers. By this time, it had become evident that embedded platforms could run full-size commercial operating systems, at least on high-end hardware, but with one catch: They could not sustain packet forwarding with satisfactory data rates. A breakthrough solution was needed. It came in the concept of a hard separation between the control and forwarding plane—an approach that became widely accepted after the success of the industry's first application-specific integrated circuit (ASIC)-driven routing platform, the Juniper Networks M40. Forwarding packets entirely in silicon was proven to be viable, clearing the path for next generation network operating systems, led by Juniper with its Junos OS.

Today, the original M40 routers are mostly retired, but their legacy lives in many similar designs, and their blueprints are widely recognized in the industry as the second-generation reference architecture.

Second-generation network operating systems are free from packet switching and thus are focused on control plane

functions. Unlike its first-generation counterparts, a second-generation OS can fully use the potential of multitasking, multithreading, memory management and context manipulation, all making systemwide failures less common. Most core and edge routers installed in the past few years are running second-generation operating systems, and it is these systems that are currently responsible for moving the bulk of traffic on the Internet and in corporate networks. However, the lack of a software data plane in second-generation operating systems prevents them from powering low-end devices without a separate (hardware) forwarding plane. Also, some customers cannot migrate from their older software easily because of compatibility issues and legacy features still in use.

These restrictions led to the rise of transitional (generation 1.5) OS designs, in which a first-generation monolithic image would run as a process on top of the second-generation scheduler and kernel, thus bridging legacy features with newer software concepts. The idea behind "generation 1.5" was to introduce some headroom and gradually move the functionality into the new code, while retaining feature parity with the original code base. Although interesting engineering exercises, such designs were not as feature-rich as their predecessors, nor as effective as their successors, making them of questionable value in the long term.

Third-Generation OS: Flexibility, Scalability and Continuous Operation

Although second-generation designs were very successful, the past 10 years have brought new challenges.

Increased competition led to the need to lower operating expenses and a coherent case for network software flexible enough to be redeployed in network devices across the larger part of the end-to-end packet path. From multiple terabit routers to Layer 2 switches and security appliances, the "best-in-class" catchphrase can no longer justify a splintered operational experience—true "network" operating systems are clearly needed. Such systems must also achieve continuous operation, so that software failures in the routing code, as well as system upgrades, do not affect the state of the network. Meeting this challenge requires availability and convergence characteristics that go far beyond the hardware redundancy available in second-generation routers.

Another key goal of third-generation operating systems is the capability to run with zero downtime (planned and unplanned). Drawing on the lesson learned from previous designs regarding the difficulty of moving from one OS to another, third-generation operating systems also should make the migration path completely transparent to customers. They must offer an evolutionary, rather than revolutionary upgrade experience typical to the retirement process of legacy software designs.

Basic OS Design Considerations

Choosing the right foundation (prototype) for an operating system is very important, as it has significant implications for the overall software design process and final product quality and serviceability. This importance is why OEM vendors sometimes migrate from one prototype platform to another midway through the development process, seeking

a better fit. Generally, the most common transitions are from a proprietary to a commercial code base and from a commercial code base to an open-source software foundation.

Regardless of the initial choice, as networking vendors develop their own code, they get further and further away from the original port, not only in protocol-specific applications but also in the system area. Extensions such as control plane redundancy, in-service software upgrades and multi chassis operation require significant changes on all levels of the original design.

However, it is highly desirable to continue borrowing content from the donor OS in areas that are not normally the primary focus of networking vendors, such as improvements in memory management, scheduling, multi core and symmetric multiprocessing (SMP) support, and host hardware drivers. With proper engineering discipline in place, the more active and peer-reviewed the donor OS is, the more quickly related network products can benefit from new code and technology.

This relationship generally explains another market trend—only two out of five network operating systems that emerged in the routing markets over the past 10 years used a commercial OS as a foundation.

Juniper's main operating system, Junos OS, is an excellent illustration of this industry trend. The basis of the Junos OS kernel comes from the FreeBSD UNIX OS, an open-source software system. The Junos OS kernel and infrastructure have since been heavily modified to accommodate advanced and unique features such as state

replication, nonstop active routing and in-service software upgrades, all of which do not exist in the donor operating system.

Nevertheless, the Junos OS tree can still be synchronized with the FreeBSD repository to pick the latest in system code, device drivers and development tool chains, which allows Juniper Networks engineers to concentrate on network-specific development.

Commercial Versus Open-Source Donor OS

The advantage of a more active and popular donor OS is not limited to just minor improvements—the cutting edge of technology creates new dimensions of product flexibility and usability. Not being locked into a single-vendor framework and roadmap enables greater control of product evolution as well as the potential to gain from progress made by independent developers.

This benefit is evident in Junos OS, which became a first commercial product to offer hard resource separation of the control plane and a real-time software data plane. Juniper-specific extension of the original BSD system architecture relies on multicore CPUs and makes Junos OS the only operating system that powers both low-end software-only systems and high-end multiple-terabit hardware platforms with images built from the same code tree. This technology and experience could not be created without support from the entire Internet-driven community. The powerful collaboration between leading individuals, universities and commercial organizations helps Junos OS stay on the very edge of operating system development. Further, this collaboration works both ways:

Juniper donates to the free software movement, one example being the Juniper Networks FreeBSD/MIPS port.

Functional Separation and Process Scheduling

Multiprocessing, functional separation and scheduling are fundamental for almost any software design, including network software. Because CPU and memory are shared resources, all running threads and processes have to access them in a serial and controlled fashion. Many design choices are available to achieve this goal, but the two most important are the memory model and the scheduling discipline.

Memory Model

The memory model defines whether processes (threads) run in a common memory space. If they do, the overhead for switching the threads is minimal, and the code in different threads can share data via direct memory pointers. The downside is that a runaway process can cause damage in memory that does not belong to it.

In a more complex memory model, threads can run in their own virtual machines, and the operating system switches the context every time the next thread needs to run. Because of this context switching, direct communication between threads is no longer possible and requires special Inter Process Communication (IPC) structures such as pipes, files and shared memory pools.

Scheduling Discipline

Scheduling choices are primarily between cooperative and preemptive models, which define whether thread switching happens voluntarily. A cooperative multitasking model allows

the thread to run to completion, and a preemptive design ensures that every thread gets access to the CPU regardless of the state of other threads.

Virtual Memory/Preemptive Scheduling Programming Model

Virtual memory with preemptive scheduling is a great design choice for properly constructed functional blocks, where interaction between different modules is limited and well defined. This technique is one of the main benefits of the second-generation OS designs and underpins the stability and robustness of contemporary network operating systems. However, it has its own drawbacks.

Notwithstanding the overhead associated with context switching, consider the interaction between two threads, A and B, both relying on the common resource R. Because threads do not detect their relative scheduling in the preemptive model, they can actually access R in a different order and with varying intensity. For example, R can be accessed by A, then B, then A, then A and then B again. If thread B modifies resource R, thread A may get different results at different times—and without any predictability. For instance, if R is an interior gateway protocol (IGP) next hop, B is an IGP process, and A is a BGP process, then BGP route installation may fail because the underlying next hop was modified midway through routing table modification. This scenario would never happen in the cooperative multitasking model, because the IGP process would release the CPU only after it finishes the next-hop maintenance. This problem is well researched and understood within software design theory, and special solutions such as resource locks

and synchronization primitives are easily available in nearly every operating system. However, the effectiveness of IPC depends greatly on the number of interactions between different processes. As the number of interacting processes increases, so does the number of IPC operations. In a carefully designed system, the number of IPC operations is proportional to the number of processes (N). In a system with extensive IPC activity, this number can be proportional to N2.

Exponential growth of an IPC map is a negative trend not only because of the associated overhead, but because of the increasing number of unexpected process interactions that may escape the attention of software engineers.

In practice, overgrown IPC maps result in systemwide "IPC meltdowns" when major events trigger intensive interactions. For instance, pulling a line card would normally affect interface management, IGP, exterior gateway protocol and traffic engineering processes, among others. When interprocess interactions are not well contained, this event may result in locks and tight loops, with multiple threads waiting on each other and vital system operations such as routing table maintenance and IGP computations temporarily suspended. Such defects are signatures of improper modularization, where similar or heavily interacting functional parts do not run as one process or one thread. The right question to ask is, "Can a system be too modular?" The conventional wisdom says, "Yes." Excessive modularity can bring long-term problems, with code complexity, mutual locks and unnecessary process interdependencies. Although none of these may be severe enough to halt development,

feature velocity and scaling parameters can be affected. Complex process interactions make programming for such a network OS an increasingly difficult task.

On the other hand, the cooperative multitasking, shared memory paradigm becomes clearly suboptimal if unrelated processes are influencing each other via the shared memory pool and collective restartability. A classic problem of first-generation operating systems was systemwide failure due to a minor bug in a nonvital process such as SNMP or network statistics. Should such an error occur in a protected and independently restartable section of system code, the defect could easily be contained within its respective code section.

This brings us to an important conclusion. No fixed principle in software design fits all possible situations. Ideally, code design should follow the most efficient paradigm and apply different strategies in different parts of the network OS to achieve the best marriage of architecture and function. This approach is evident in Junos OS, where functional separation is maintained so that cooperative multitasking and preemptive scheduling can both be used effectively, depending on the degree of IPC containment between functional modules.

Generic Kernel Design

Kernels normally do not provide any immediately perceived or revenue-generating functionality. Instead, they perform housekeeping activities such as memory allocation and hardware management and other system-level tasks. Kernel threads are likely the most often run tasks in the entire system. Consequently, they have to be robust and

run with minimal impact on other processes. In the past, kernel architecture largely defined the operating structure of the entire system with respect to memory management and process scheduling. Hence, kernels were considered important differentiators among competing designs.

Historically, the disputes between the proponents and opponents of lightweight versus complex kernel architectures came to a practical end when most operating systems became functionally decoupled from their respective kernels.

Once software distributions became available with alternate kernel configurations, researchers and commercial developers were free to experiment with different designs.

For example, the original Carnegie-Mellon Mach microkernel was originally intended to be a drop-in replacement for the kernel in BSD UNIX and was later used in various operating systems, including mkLinux and GNU FSF projects. Similarly, some software projects that started life as purely microkernel-based systems later adopted portions of monolithic designs.

Over time, the radical approach of having a small kernel and moving system functions into the user-space processes did not prevail. A key reason for this was the overhead associated with extra context switches between frequently executed system tasks running in separate memory spaces.

Furthermore, the benefits associated with restart ability of essentially all system processes proved to be of limited value, especially in embedded systems. With the system code being very well tested and limited to scheduling, memory management and a handful of device drivers, the potential errors in kernel subsystems are more likely to be related to

hardware failures than to software bugs. This means, for example, that simply restarting a faulty disk driver is unlikely to help the routing engine stay up and running, as the problem with storage is likely related to a hardware failure (for example, uncorrectable fault in a mass storage device or system memory bank).

Another interesting point is that although both monolithic and lightweight kernels were widely studied by almost all operating system vendors, few have settled on purist implementations. For example, Apple's Mac OS X was originally based on microkernel architecture, but now runs system processes, drivers and the operating environment in BSD-like subsystems. Microsoft NT and derivative operating systems also went through multiple changes, moving critical performance components such as graphical and I/O subsystems in and out of the system kernel to find the right balance of stability, performance and predictability. These changes make NT a hybrid operating system. On the other hand, freeware development communities such as FSF, FreeBSD and NetBSD have mostly adopted monolithic designs (for example, Linux kernel) and have gradually introduced modularity into selected kernel sections (for example, device drivers).

So what difference does kernel architecture make to routing and control?

Analysts of Traditional Physical Security Systems

Analysts of traditional physical security systems have suggested two further design principles which, unfortunately, apply only imperfectly to computer systems.

Work Factor

Compare the cost of circumventing the mechanism with the resources of a potential attacker. The cost of circumventing, commonly known as the "work factor," in some cases can be easily calculated. For example, the number of experiments needed to try all possible four letter alphabetic passwords is 26^4 = 456 976.

If the potential attacker must enter each experimental password at a terminal, one might consider a four-letter password to be adequate. On the other hand, if the attacker could use a large computer capable of trying a million passwords per second, as might be the case where industrial espionage or military security is being considered, a four-letter password would be a minor barrier for a potential intruder.

The trouble with the work factor principle is that many computer protection mechanisms are *not* susceptible to direct work factor calculation, since defeating them by systematic attack may be logically impossible. Defeat can be accomplished only by indirect strategies, such as waiting for an accidental hardware failure or searching for an error in implementation. Reliable estimates of the length of such a wait or search are very difficult to make.

Compromise Recording

It is sometimes suggested that mechanisms that reliably record that a compromise of information has occurred can be used in place of more elaborate mechanisms that completely prevent loss. For example, if a tactical plan is known to have been compromised, it may be possible to

construct a different one, rendering the compromised version worthless. An unbreakable padlock on a flimsy file cabinet is an example of such a mechanism.

Although the information stored inside may be easy to obtain, the cabinet will inevitably be damaged in the process and the next legitimate user will detect the loss. For another example, many computer systems record the date and time of the most recent use of each file. If this record is tamperproof and reported to the owner, it may help discover unauthorized use.

In computer systems, this approach is used rarely, since it is difficult to guarantee discovery once security is broken. Physical damage usually is not involved, and logical damage (and internally stored records of tampering) can be undone by a clever attacker. As is apparent, these principles do not represent absolute rules—they serve best as warnings. If some part of a design violates a principle, the violation is a symptom of potential trouble, and the design should be carefully reviewed to be sure that the trouble has been accounted for or is unimportant.

Considerations Surrounding Protection

Briefly, then, we may outline our discussion to this point. The application of computers to information handling problems produces a need for a variety of security mechanisms. We are focusing on one aspect, computer protection mechanisms—the mechanisms that control access to information by executing programmes. At least four levels of functional goals for a protection system can be identified: all-or-nothing systems, controlled sharing, user-programmed sharing controls, and putting strings on

information. But at all levels, the provisions for dynamic changes to authorization for access are a severe complication.

Since no one knows how to build a system without flaws, the alternative is to rely on eight design principles, which tend to reduce both the number and the seriousness of any flaws: Economy of mechanism, fail-safe defaults, complete mediation, open design, separation of privilege, least privilege, least common mechanism, and psychological acceptability. Finally, some protection designs can be evaluated by comparing the resources of a potential attacker with the work factor required to defeat the system, and compromise recording may be a useful strategy.

Technical Underpinnings

The Development Plan

At this point we begin a development of the technical basis of information protection in modern computer systems. There are two ways to approach the subject: from the top down, emphasizing the abstract concepts involved, or from the bottom up, identifying insights by, studying example systems. We shall follow the bottom-up approach, introducing a series of models of systems as they are, (or could be) built in real life. The reader should understand that on this point the authors' judgment differs from that of some of their colleagues. The top-down approach can be very satisfactory when a subject is coherent and self-contained, but for a topic still containing *ad hoc* strategies and competing world views, the bottom-up approach seems safer.

Our first model is of a multiuser system that completely isolates its users from one another. We shall then see how

the logically perfect walls of that system can be lowered in a controlled way to allow limited sharing of information between users.

The mechanics of sharing using two different models: the capability system and the access control list system. It then extends these two models to handle the dynamic situation in which authorizations can change under control of the programmes running inside the system. Further extensions to the models control the dynamics. The final model (only superficially explored) is of protected objects and protected subsystems, which allow arbitrary modes of sharing that are unanticipated by the system designer. These models are not intended so much to explain the particular systems as they are to explain the underlying concepts of information protection.

Our emphasis throughout the development is on direct access to information (for example, using LOAD and STORE instructions) rather than acquiring information indirectly (as when calling a data base management system to request the average value of a set of numbers supposedly not directly accessible).

Control of such access is the function of the protected subsystems developed near the end of the paper. Herein lies perhaps the chief defect of the bottom-up approach, since conceptually there seems to be no reason to distinguish direct and indirect access, yet the detailed mechanics are typically quite different. The beginnings of a top-down approach based on a message model that avoids distinguishing between direct and indirect information access may be found in a paper by Lampson.

The Essentials of Information Protection

For purposes of discussing protection, the information stored in a computer system is not a single object. When one is considering direct access, the information is divided into mutually exclusive partitions, as specified by its various creators. Each partition contains a collection of information, all of which is intended to be protected uniformly. The uniformity of protection is the same kind of uniformity that applies to all of the diamonds stored in the same vault: any person who has a copy of the combination can obtain any of the diamonds. Thus the collections of information in the partitions are the fundamental objects to be protected.

Conceptually, then, it is necessary to build an impenetrable wall around each distinct object that warrants separate protection, construct a door in the wall through which access can be obtained, and post a guard at the door to control its use.

Control of use, however, requires that the guard have some way of knowing which users are authorized to have access, and that each user have some reliable way of identifying himself to the guard. This authority check is usually implemented by having the guard demand a match between something he knows and something the prospective user possesses. Both protection and authentication mechanisms can be viewed in terms of this general model.

Before extending this model, we pause to consider two concrete examples, the multiplexing of a single computer system among several users and the authentication of a user's claimed identity. These initial examples are complete isolation systems—no sharing of information can happen.

Later we will extend our model of guards and walls in the discussion of shared information.

An Isolated Virtual Machine

A typical computer consists of a processor, a linearly addressed memory system, and some collection of input/output devices associated with the processor. It is relatively easy to use a single computer to simulate several, each of which is completely unaware of the existence of the others, except that each runs more slowly than usual.

Such a simulation is of interest, since during the intervals when one of the simulated (commonly called *virtual*) processors is waiting for an input or output operation to finish, another virtual processor may be able to progress at its normal rate. Thus a single processor may be able to take the place of several. Such a scheme is the essence of a multiprogramming system.

To allow each virtual processor to be unaware of the existence of the others, it is essential that some isolation mechanism be provided.

One such mechanism is a special hardware register called a *descriptor register*. In this figure, all memory references by the processor are checked by an extra piece of hardware that is interposed in the path to the memory. The descriptor register controls exactly which part of memory is accessible. The descriptor register contains two components: a *base* value and a *bound* value.

The base is the lowest numbered address the programme may use, and the bound is the number of locations beyond the base that may be used. We will call the value in the

descriptor register a *descriptor*, as it describes an object (in this case, one programme) stored in memory. The programme controlling the processor has full access to everything in the base-bound range, by virtue of possession of its one descriptor. As we go on, we shall embellish the concept of a descriptor: it is central to most implementations of protection and of sharing of information.

So far, we have not provided for the dynamics of a complete protection scheme: we have not discussed who loads the descriptor register. If any running programme could load it with any arbitrary value, there would be no protection. The instruction that loads the descriptor register with a new descriptor must have some special controls—either on the values it will load or on who may use it.

It is easier to control who may use the descriptor, and a common scheme is to introduce an additional bit in the processor state. This bit is called the *privileged state* bit. All attempts to load the descriptor register are checked against the value of the privileged state bit; the privileged state bit must be ON for the register to be changed. One programme runs with the privileged state bit ON, and controls the simulation of the virtual processors for the other programmes. All that is needed to make the scheme complete is to ensure that the privileged state bit cannot be changed by the user programmes except, perhaps, by an instruction that simultaneously transfers control to the supervisor programme at a planned entry location. (In most implementations, the descriptor register is not used in the privileged state.)

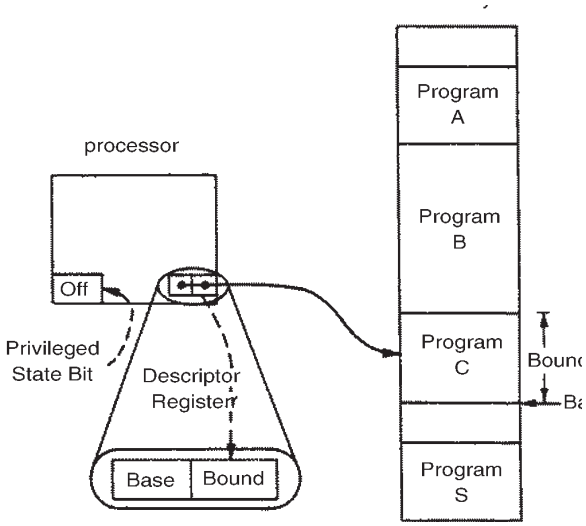


Fig. Use of a Descriptor Register to sSimulate Multiple Virtual Machines. Programme C is in Control of tho Processor. The Privileged State bit has Value OFF, indicating that Programme C is a User Programme. When Programme S is Running, the Privileged State bit has Value ON. In this (and later) Figures, Lower Addresses are Nearer the Bottom of the Figure.

One might expect the supervisor programme to maintain a table of values of descriptors, one for each virtual processor. When the privileged state bit is OFF, the index in this table of the programme currently in control identifies exactly which programme—and thus which virtual processor—is accountable for the activity of the real processor. For protection to be complete, a virtual processor must not be able to change arbitrarily the values in the table of descriptors. If we suppose the table to be stored inside the supervisor programme, it will be inaccessible to the virtual processors. We have here an example of a common strategy and sometime cause of confusion: the protection mechanisms not only protect one user from another, *they may also protect their own implementation*. We shall encounter this strategy again.

So far, this virtual processor implementation contains three protection mechanisms that we can associate with our

abstractions. For the first, the information being protected is the distinct programmes. The guard is represented by the extra piece of hardware that enforces the descriptor restriction. The impenetrable wall with a door is the hardware that forces all references to memory through the descriptor mechanism. The authority check on a request to access memory is very simple. The requesting virtual processor is identified by the base and bound values in the descriptor register, and the guard checks that the memory location to which access is requested lies within the indicated area of memory.

The second mechanism protects the contents of the descriptor register. The wall, door, and guard are implemented in hardware, as with the first mechanism. An executing programme requesting to load the descriptor register is identified by the privileged state bit. If this bit is OFF, indicating that the requester is a user programme, then the guard does not allow the register to be loaded. If this bit is ON, indicating that the requester is the supervisor programme, then the guard does allow it.

The third mechanism protects the privileged state bit. It allows an executing programme identified by the privileged state bit being OFF (a user programme) to perform the single operation "turn privileged state bit ON and transfer to the supervisor programme." An executing programme identified by the privileged state bit being ON is allowed to turn the bit OFF. This third mechanism is an embryonic form of the sophisticated protection mechanisms required to implement protected subsystems. The supervisor programme is an example of a protected subsystem, of which more will be said later.

The supervisor programme is part of all three protection mechanisms, for it is responsible for maintaining the integrity of the identifications manifest in the descriptor register and the privileged state bit. If the supervisor does not do its job correctly, virtual processors could become labelled with the wrong base and bound values, or user programmes could become labelled with a privileged state bit that is ON, The supervisor protects itself from the user programmes with the same isolation hardware that separates users, an example of the "economy of mechanism" design principle.

With an appropriately sophisticated and careful supervisor programme, we now have an example of a system that completely isolates its users from one another. Similarly isolated permanent storage can be added to such a system by attaching some longterm storage device (*e.g.*, magnetic disk) and developing a similar descriptor scheme for its use. Since long-term storage is accessed less frequently than primary memory, it is common to implement its descriptor scheme with the supervisor programmes rather than hardware, but the principle is the same. Data streams to input or output devices can be controlled similarly. The combination of a virtual processor, a memory area, some data streams, and an isolated region of long-term storage is known as a virtual machine.

Long-term storage does, however, force us to face one further issue. Suppose that the virtual machine communicates with its user through a typewriter terminal. If a new user approaches a previously unused terminal and requests to use a virtual machine, which virtual machine (and, therefore, which set of long-term stored information)

should he be allowed to use? We may solve this problem outside the system, by having the supervisor permanently associate a single virtual machine and its long-term storage area with a single terminal.

Then, for example, padlocks can control access to the terminal. If, on the other hand, a more flexible system is desired, the supervisor programme must be prepared to associate any terminal with any virtual machine and, as a result, must be able to verify the identity of the user at a terminal. Schemes for performing this authentication are the subject of our next example.

Authentication Mechanisms

Our second example is of an authentication mechanism: a system that verifies a user's claimed identity. The mechanics of this authentication mechanism differ from those of the protection mechanisms for implementing virtual machines mainly because not all of the components of the system are under uniform physical control. In particular, the user himself and the communication system connecting his terminal to the computer are components to be viewed with suspicion.

Conversely, the user needs to verify that he is in communication with the expected computer system and the intended virtual machine. Such systems follow our abstract model of a guard who demands a match between something he knows and something the requester possesses. The objects being protected by the authentication mechanism are the virtual machines. In this case, however, the requester is a computer system user rather than an executing

programme, and because of the lack of physical control over the user and the communication system, the security of the computer system must depend on either the secrecy or the unforgeability of the user's identification. In time-sharing systems, the most common scheme depends on secrecy. The user begins by typing the name of the person he claims to be, and then the system demands that the user type a password, presumably known only to that person.

There are, of course, many possible elaborations and embellishments of this basic strategy. In cases where the typing of the password may be observed, passwords may be good for only one use, and the user carries a list of passwords, crossing each one off the list as he uses it. Passwords may have an expiration date, or usage count, to limit the length of usefulness of a compromised one. The list of acceptable passwords is a piece of information that must be carefully guarded by the system.

In some systems, all passwords are passed through a hard-to-invert transformation before being stored, an idea suggested by R. Needham. When the user types his password, the system transforms it also and compares the transformed versions. Since the transform is supposed to be hard to invert (even if the transform itself is well known), if the stored version of a password is compromised, it may be very difficult to determine what original password is involved. It should be noted, however, that "hardness of inversion" is difficult to measure. The attacker of such a system does not need to discern the general inversion, only the particular one applying to some transformed password he has available.

Passwords as a general technique have some notorious defects. The most often mentioned defect lies in choice of password—if a person chooses his own password, he may choose something easily guessed by someone else who knows his habits. In one recent study of some 300 self-chosen passwords on a typical time-sharing system, more than 50 percent were found to be short enough to guess by exhaustion, derived from the owner's name, or something closely associated with the owner, such as his telephone number or birth date. For this reason, some systems have programmes that generate random sequences of letters for use as passwords.

They may even require that all passwords be system-generated and changed frequently. On the other hand, frequently changed random sequences of letters are hard to memorize, so such systems tend to cause users to make written copies of their passwords, inviting compromise. One solution to this problem is to provide a generator of "pronounceable" random passwords based on digraph or higher order frequency statistics to make memorization easier.

A second significant defect is that the password must be exposed to be used. In systems where the terminal is distant from the computer, the password must be sent through some communication system, during which passage a wiretapper may be able to intercept it. An alternative approach to secrecy is unforgeability.

The user is given a key, or magnetically striped plastic card, or some other unique and relatively difficult-to-fabricate object. The terminal has an input device that examines the object and transmits its unique identifying code to the

computer system, which treats the code as a password that need not be kept secret. Proposals have been made for fingerprint readers and dynamic signature readers in order to increase the effort required for forgery.

The primary weakness of such schemes is that the hard-to-fabricate object, after being examined by the specialized input device, is reduced to a stream of bits to be transmitted to the computer. Unless the terminal, its object reader, and its communication lines to the computer are physically secured against tampering, it is relatively easy for an intruder to modify the terminal to transmit any sequence of bits he chooses. It may be necessary to make the acceptable bit sequences a secret after all. On the other hand, the scheme is convenient, resists casual misuse, and provides a conventional form of accountability through the physical objects used as keys.

A problem common to both the password and the unforgeable object approach is that they are "one-way" authentication schemes. They authenticate the user to the computer system, but not *vice-versa*. An easy way for an intruder to penetrate a password system, for example, is to intercept all communications to and from the terminal and direct them to another computer—one that is under the interceptor's control. This computer can be programmed to "masquerade," that is, to act just like the system the caller intended to use, up to the point of requesting him to type his password. After receiving the password, the masquerader gracefully terminates the communication with some unsurprising error message, and the caller may be unaware that his password has been stolen. The same attack can be

used on the unforgeable object system as well. A more powerful authentication technique is sometimes used to protect against masquerading. Suppose that a remote terminal is equipped with enciphering circuitry, such as the LUCIFER system, that scrambles all signals from that terminal. Such devices normally are designed so that the exact encipherment is determined by the value of a key, known as the *encryption* or*transformation* key.

For example, the transformation key may consist of a sequence of 1000 binary digits read from a magnetically striped plastic card. In order that a recipient of such an enciphered signal may comprehend it, he must have a deciphering circuit primed with an exact copy of the transformation key, or else he must cryptanalyse the scrambled stream to try to discover the key. The strategy of encipherment/decipherment is usually invoked for the purpose of providing communications security on an otherwise unprotected communications system. However, it can simultaneously be used for authentication, using the following technique, first published in the unclassified literature by Feistel.

The user, at a terminal, begins bypassing the enciphering equipment. He types his name. This name passes, unenciphered, through the communication system to the computer. The computer looks up the name, just as with the password system. Associated with each name, instead of a secret password, is a secret transformation key. The computer loads this transformation key into its enciphering mechanism, turns it on, and attempts to communicate with the user. Meanwhile, the user has loaded his copy of the

transformation key into his enciphering mechanism and turned it on. Now, if the keys are identical, exchange of some standard hand-shaking sequence will succeed. If they are not identical, the exchange will fail, and both the user and the computer system will encounter unintelligible streams of bits. If the exchange succeeds, the computer system is certain of the identity of the user, and the user is certain of the identity of the computer.

The secret used for authentication—the transformation key—has not been transmitted through the communication system. If communication fails (because the user is unauthorized, the system has been replaced by a masquerader, or an error occurred), each party to the transaction has immediate warning of a problem.

Relatively complex elaborations of these various strategies have been implemented, differing both in economics and in assumptions about the psychology of the prospective user. For example, Branstad explored in detail strategies of authentication in multinode computer networks. Such elaborations, though fascinating to study and analyse, are diversionary to our main topic of protection mechanisms.

Shared Information

The virtual machines are totally independent, as far as information accessibility was concerned. Each user might just as well have his own private computer system. With the steadily declining costs of computer manufacture there are few technical reasons not to use a private computer. On the other hand, for many applications some sharing of information among users is useful, or even essential. For

example, there may be a library of commonly used, reliable programmes. Some users may create new programmes that other users would like to use. Users may wish to be able to update a common data base, such as a file of airline seat reservations or a collection of programmes that implement a biomedical statistics system. In all these cases, virtual machines are inadequate, because of the total isolation of their users from one another. Before extending the virtual machine example any further, let us return to our abstract discussion of guards and walls.

Implementations of protection mechanisms that permit sharing fall into the two general categories described by Wilkes:

- "List-oriented" implementations, in which the guard holds a list of identifiers of authorized users, and the user carries a unique unforgeable identifier that must appear on the guard's list for access to be permitted. A store clerk checking a list of credit customers is an example of a list-oriented implementation in practice. The individual might use his driver's license as a unique unforgeable identifier.
- "Ticket-oriented" implementations, in which the guard holds the description of a single identifier, and each user has a collection of unforgeable identifiers, or tickets, corresponding to the objects to which he has been authorized access. A locked door that opens with a key is probably the most common example of a ticket-oriented mechanism; the guard is implemented as the hardware of the lock, and the

matching key is the (presumably) unforgeable authorizing identifier.

Authorization, defined as giving a user access to some object, is different in these two schemes. In a list-oriented system, a user is authorized to use an object by having his name placed on the guard's list for that object. In a ticket-oriented system, a user is authorized by giving him a ticket for the object.

We can also note a crucial mechanical difference between the two kinds of implementations. The list-oriented mechanism requires that the guard examine his list at the time access is requested, which means that some kind of associative search must accompany the access. On the other hand, the ticket-oriented mechanism places on the user the burden of choosing which ticket to present, a task he can combine with deciding which information to access. The guard only need compare the presented ticket with his own expectation before allowing the physical memory access. Because associative matching tends to be either slower or more costly than simple comparison, list-oriented mechanisms are not often used in applications where traffic is high.

On the other hand, ticket-oriented mechanisms typically require considerable technology to control forgery of tickets and to control passing tickets around from one user to another. As a rule, most real systems contain both kinds of sharing implementations—a list-oriented system at the human interface and a ticket-oriented system in the underlying hardware implementation. This kind of arrangement is accomplished by providing, at the higher

level, a list-oriented guard whose only purpose is to hand out temporary tickets which the lower level (ticket-oriented) guards will honor. Some added complexity arises from the need to keep authorizations, as represented in the two systems, synchronized with each other. Computer protection systems differ mostly in the extent to which the architecture of the underlying ticket-oriented system is visible to the user.

Finally, let us consider the degenerate cases of list- and ticket-oriented systems. In a list-oriented system, if each guard's list of authorized users can contain only one entry, we have a "complete isolation" kind of protection system, in which no sharing of information among users can take place. Similarly, in a ticket-oriented system, if there can be only one ticket for each object in the system, we again have a "complete isolation" kind of protection system.

Thus the "complete isolation" protection system turns out to be a particular degenerate case of both the list-oriented and the ticket-oriented protection implementations. These observations are important in examining real systems, which usually consist of interacting protection mechanisms, some of which are list-oriented, some of which are ticket-oriented, and some of which provide complete isolation and therefore may happen to be implemented as degenerate examples of either of the other two, depending on local circumstances.

We should understand the relationship of a user to these transactions. We are concerned with protection of information from programmes that are executing. The user is the individual who assumes accountability for the actions of an executing programme. Inside the computer system, a programme is executed by a virtual processor, so one or more

virtual processors can be identified with the activities directed by the user. In a list-oriented system it is the guard's business to know whose virtual processor is attempting to make an access.

The virtual processor has been marked with an unforgeable label identifying the user accountable for its actions, and the guard inspects this label when making access decisions. In a ticket-oriented system, however, the guard cares only that a virtual processor present the appropriate unforgeable ticket when attempting an access. The connection to an accountable user is more diffuse, since the guard does not know or care how the virtual processor acquired the tickets. In either case, we conclude that in addition to the information inside the impenetrable wall, there are two other things that must be protected: the guard's authorization information, and the association between a user and the unforgeable label or set of tickets associated with his virtual processors.

Since an association with some user is essential for establishing accountability for the actions of a virtual processor, it is useful to introduce an abstraction for that accountability—the*principal*. A principal is, by definition, the entity accountable for the activities of a virtual processor. In the situations discussed so far, the principal corresponds to the user outside the system. However, there are situations in which a one-to-one correspondence of individuals with principals is not adequate. For example, a user may be accountable for some very valuable information and authorized to use it. On the other hand, on some occasion he may wish to use the computer for some purpose unrelated

to the valuable information. To prevent accidents, he may wish to identify himself with a different principal, one that does not have access to the valuable information—following the principle of least privilege. In this case there is a need for two different principals corresponding to the same user.

Similarly, one can envision a data base that is to be modified only if a committee agrees. Thus there might be an authorized principal that cannot be used by any single individual; all of the committee members must agree upon its use simultaneously.

Because the principal represents accountability, that authorizing access is done in terms of principals. That is, if one wishes a friend to have access to some file, the authorization is done by naming a principal only that friend can use.

For each principal we may identify all the objects in the system which the principal has been authorized to use. We will name that set of objects the *domain* of that principal. Summarizing, then, a principal is the unforgeable identifier attached to a virtual processor in a list-oriented system.

When a user first approaches the computer system, that user must identify the principal to be used. Some authentication mechanism, such as a request for a secret password, establishes the user's right to use that principal. The authentication mechanism itself may be either list- or ticket-oriented or of the complete isolation type.

Then a computation is begun in which all the virtual processors of the computation are labelled with the identifier of that principal, which is considered accountable for all further actions of these virtual processors. The

authentication mechanism has allowed the virtual processor to enter the domain of that principal. That description makes apparent the importance of the authentication mechanism. Clearly, one must carefully control the conditions under which a virtual processor enters a domain. Finally, we should note that in a ticket-oriented system there is no mechanical need to associate an unforgeable identifier with a virtual processor, since the tickets themselves are presumed unforgeable. Nevertheless, a collection of tickets can be considered to be a domain, and therefore correspond to some principal, even though there may be no obvious identifier for that principal. Thus accountability in ticket-oriented systems can be difficult to pinpoint.

Now we shall return to our example system and extend it to include sharing. Consider for a moment the problem of sharing a library programme—say, a mathematical function subroutine. We could place a copy of the math routine in the long-term storage area of each virtual machine that had a use for it.

This scheme, although workable, has several defects. Most obvious, the multiple copies require multiple storage spaces. More subtly, the scheme does not respond well to changes. If a newer, better math routine is written, upgrading the multiple copies requires effort proportional to the number of users. These two observations suggest that one would like to have some scheme to allow different users access to a single *master copy* of the programme.

The storage space will be smaller and the communication of updated versions will be easier. In terms of the virtual

machine model of our earlier example, we can share a single copy of the math routine by adding to the real processor a second descriptor register, placing the math routine somewhere in memory by itself and placing a descriptor for it in the second descriptor register.

Following the previous strategy, we assume that the privileged state bit assures that the supervisor programme is the only one permitted to load either descriptor register.

In addition, some scheme must be provided in the architecture of the processor to permit a choice of which descriptor register is to be used for each address generated by the processor. A simple scheme would be to let the high-order address bit select the descriptor register. Thus, all addresses in the lower half of the address range would be interpreted relative to descriptor register 1, and addresses in the upper half of the address range would be relative to descriptor register 2. An alternate scheme, suggested by Dennis, is to add explicitly to the format of instruction words a field that selects the descriptor register intended to be used with the address in that instruction.

The use of descriptors for sharing information is intimately related to the addressing architecture of the processor, a relation that can cause considerable confusion. The reason why descriptors are of interest for sharing becomes apparent by comparing parts a and b. When programme A is in control, it can have access only to itself and the math routine; similarly, when programme B is in control, it can have access only to itself and the math routine. Since neither programme has the power to change the descriptor register, sharing of the math routine has been

accomplished while maintaining isolation of programme A from programme B.

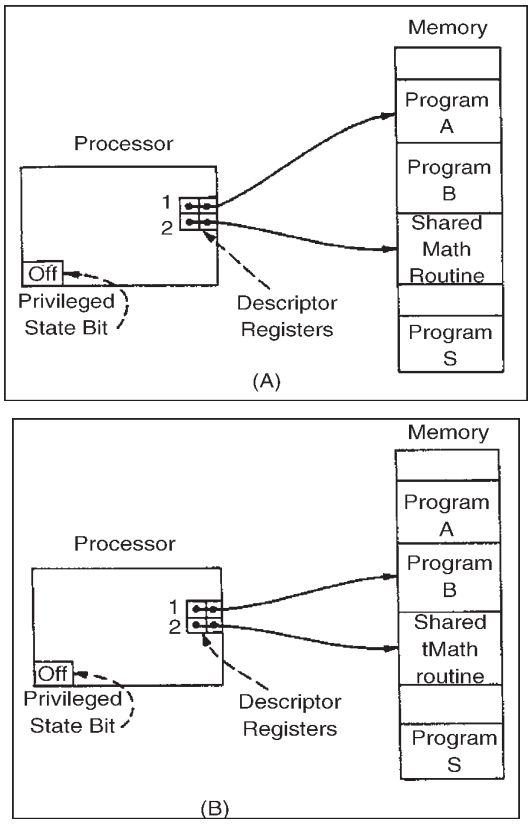


Fig. Sharing of a Math Routine by use of Two Descriptor Registors. (a) Programme A in Control of Processor. (b) Programme B in Control of Processor.

The effect of sharing is shown even more graphically redrawn with two virtual processors, one executing programme A and the other executing programme B.

Whether or not there are actually two processors is less important than the existence of the conceptually parallel access paths. Every virtual processor of the system may be viewed as having its own real processor, capable of access to the memory in parallel with that of every other virtual processor. There may be an underlying processor multiplexing facility that distributes a few real processors

among the many virtual processors, but such a multiplexing facility is essentially unrelated to protection. Recall that a virtual processor is not permitted to load its own protection descriptor registers. Instead, it must call or trap to the supervisor programme S which call or trap causes the privileged state bit to go ON and thereby permits the supervisor programme to control the extent of sharing among virtual processors. The processor multiplexing facility must be prepared to switch the entire state of the real processor from one virtual processor to another, including the values of the protection descriptor registers.

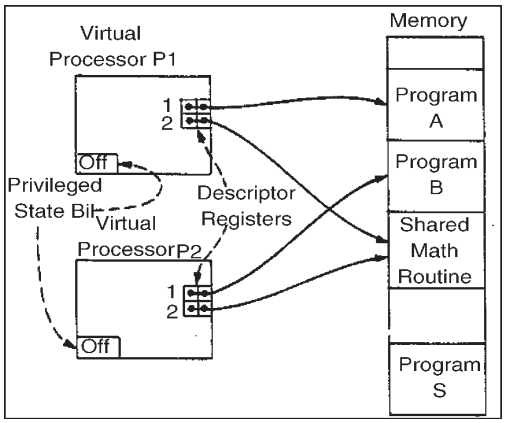


Fig. Redrawn to Show Sharing of a Math Routine by Two Virtual Processors Simultaneously.

Although the basic mechanism to permit information sharing is now in place, a remarkable variety of implications that follow from its introduction require further mechanisms.

These implications include the following.

- If virtual processor P_1 can overwrite the shared math routine, then it could disrupt the work of virtual processor P_2.
- The shared math routine must be careful about making modifications to itself and about where in

memory it writes temporary results, since it is to be used by independent computations, perhaps simultaneously.

- The scheme needs to be expanded and generalized to cover the possibility that more than one programme or data base is to be shared.
- The supervisor needs to be informed about which principals are authorized to use the shared math routine (unless it happens to be completely public with no restrictions).

Let us consider these four implications in order. If the shared area of memory is a procedure, then to avoid the possibility that virtual processor P_1 will maliciously overwrite it, we can restrict the methods of access. Virtual processor P_1 needs to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants embedded in the programme, but it has no need to write into any part of the shared procedure. We may accomplish this restriction by extending the descriptor registers and the descriptors themselves to include*accessing permission*, an idea introduced for different reasons in the original Burroughs B5000 design. For example, we may add two bits, one controlling permission to read and the other permission to write in the storage area defined by each descriptor. In virtual processor P_1, descriptor 1 would have both permissions granted, while descriptor 2 would permit only reading of data and execution of instructions. An alternative scheme would be to attach the permission bits directly to the storage areas containing the shared programme or data. Such a scheme is less satisfactory because, unlike the

descriptors so far outlined, permission bits attached to the data would provide identical access to all processors that had a descriptor. Although identical access for all users of the shared math routine might be acceptable, a data base could not be set up with several users having permission to read but a few also having permission to write.

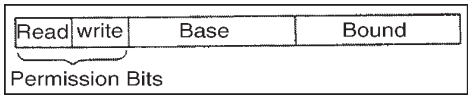


Fig. A Descriptor Containing READ and WRITE Permission Bits

The second implication of a shared procedure, mentioned before, is that the shared procedure must be careful about where it stores temporary results, since it may be used simultaneously by several virtual processors. In particular, it should avoid modifying itself. The enforcement of access permission by descriptor bits further constrains the situation. To prevent programme A from writing into the shared math routine, we have also prohibited the shared math routine from writing into itself, since the descriptors do not change when, for example, programme A transfers control to the math routine. The math routine will find that it can read but not write into itself, but that it can both read and write into the area of programme A. Thus programme A might allocate an area of its own address range for the math routine to use as temporary storage.

As for the third implication, the need for expansion, we could generalize our example to permit several distinct shared items merely by increasing the number of descriptor registers and informing the supervisor which shared objects should be addressable by each virtual processor. However, there are two substantially different forms of this

generalization—*capability systems* and *access control list systems*. The capability systems are ticket-oriented, while access control list systems are list-oriented. Most real systems use a combination of these two forms, the capability system for speed and an access control list system for the human interface. Before we can pursue these generalizations, and the fourth implication, authorization, more groundwork must be laid.

The development of protection continues with a series of successively more sophisticated models. The initial model, of a capability system, explores the use of encapsulated but copyable descriptors as tickets to provide a flexible authorization scheme.

In this context we establish the general rule that communication external to the computer must precede dynamic authorization of sharing. The limitations of copyable descriptors—primarily lack of accountability for their use—lead to analysis of revocation and the observation that revocation requires indirection. That observation in turn leads to the model of access control lists embedded in indirect objects so as to provide detailed control of authorization.

The use of access control lists leads to a discussion of controlling changes to authorizations, there being at least two models of control methods which differ in their susceptibility to abuse. Additional control of authorization changes is needed when releasing sensitive data to a borrowed programme, and this additional control implies a nonintuitive constraint on where data may be written by the borrowed programme. The concept of implementing arbitrary

abstractions, such as extended types of objects, as programmes in separate domains.

Research Networking

Research Networking (RN) is about using web-based tools to discover and use research and scholarly information about people and resources. Research Networking tools (RN tools) serve as knowledge management systems for the research enterprise.

RN tools connect institution-level/enterprise systems, national research networks, publicly available research data (*e.g.*, grants and publications), and restricted/proprietary data by harvesting information from disparate sources into compiled expertise profiles for faculty, investigators, scholars, clinicians, community partners, and facilities.

RN tools facilitate the development of new collaborations and team science to address new or existing research challenges through the rapid discovery and recommendation of researchers, expertise, and resources.

RN tools differ from search engines such as Google in that they access information in databases and other data not limited to web pages.

They also differ from social networking systems such as LinkedIn or Facebook in that they represent a compendium of data ingested from authoritative and verifiable sources rather than predominantly individually-asserted information, making RN tools more reliable.

Yet, RN tools have sufficient flexibility to allow for profile editing. RN tools also provide resources to bolster human connector systems: they can make non-intuitive matches, they

do not depend on serendipity, and they do not have a propensity to return only to previously identified collaborations/ collaborators. RN tools also generally have associated analytical capabilities that enable evaluation of collaboration and cross-disciplinary research/scholarly activity, especially over time. Importantly, data harvested into robust RN tools is accessible for broad repurposing, especially if available as linked open data (RDF triples). Thus RN tools enhance research support activities by providing data for customized, up-to-date web pages, CV/biosketch generation, and data tables for grant proposals.

The Developing Trend of Computer Network Management System

Computer network management system is now starting to enter the application layer. Traditionally, computer network management system mainly concerned about the various network equipment, which was in the network layer. Centered around equipment or equipment assemblage, it used SNMP to control and manage equipment. Web users have higher demand on network as well as network bandwidth.

Some demands concern the transmission of time sensitive data, such as real-time audio and video, but some data are not time sensitive. Therefore, considering the limit current network bandwidth, it is imperative to change the previous practice of not differentiating service content, but to provide high quality service to all individual users according to the service contents, so as to better utilize the resources of bandwidth. This is called QOS (Quality of

Services). With this idea, network management starts to move the controlling force from network layer to application layer. R1MON2 has tried this way, which was an important change to network management system. In spite of all the versatile technologies used in network management system, as a result of standardization activities and the need for system interconnection.

Distributed Network Management

The key of distributed object is to solve the problem of cross-platform connection and interaction, and to realise distributed application system. The CORBA presented by OMG is quite an ideal platform. Distributed network management is to set up multi domain management processes. Domain management process takes charges of the objects in the domain, and at the same time, different domains coordinates and interacts with each other, so as to perform the management of global area network.

Thus, not only is the load of central network management reduced, but time lag for transmission of information on network management is decreased as well, which makes management more effective. Distributed technologies mainly have two aspects: one utilizes CORBA, and the other mobile agent technology. In the near future, centralized distributed network management model can be used to realise the functions of centralization and data acquisition distribution.

Integrated Network Management

Integrated network management requires network management system provide the multitier management

support. It can keep all the sub networks in perspective through one operating platform, understanding its operating businesses, identifying and eliminating failures.

Thus, the multi interlinked networks management is fulfilled. With network management having become more and more important, many different network management systems have emerged, including those that manage SDH networks and IP networks.

The networks managed by these systems interlinks with and interdependent on with each other. There are multi network management systems at the same time. They are independent, in charge of different parts of the network. There can even be a few network management systems with same contents existing concurrently. They come from different manufacturers, and manage their equipment respectively, which has greatly made network management more complex.

Businesses Monitoring

Traditionally, network management aims at network equipment, and could not directly reflect the impact of equipment failure on businesses. Up until now, some network products have realised the monitoring of processes. However, for some services, even though the services end, but the processes still exist.

The monitoring of services cannot be clearly shown. For customers, they are concerned more about the services they get, such as quantity and quality of programs. Therefore, the monitoring of services and businesses is the further goal of management.

Intelligent Management

It supports strategic management and network management system self diagnosis and self adjustment. Network management is the method of managing the tools that belong to a network and maintaining, administering all the systems that are connected in the network. For one to be able to efficiently manage a network that person should be a qualified network administrator and should have in depth knowledge of the functionalities of the network and different topologies of network. There are two aspects in any network, one is the logical aspect and the other is the physical level. The network administrator should be good at both the logical and the physical aspects to be able to troubleshoot efficiently.

Network Administration Functions

The main task of network administration is to keep the network running smoothly 24 hours a day. The main function is to monitor the network constantly and look for possible trouble, detect and rectify the trouble before the network gets affected.

The network administration part of the job involves the resources available on the network and their functioning. The administrator is required to keep a track of all the resources available in the network and ensure they are functioning properly.

Network maintenance part of the job involves installing updates frequently, updating the service packs for the network software's and also applying patches for the routers when needed. The software and the hardware for the network

must be constantly monitored for updates and maintained in order for the network to function properly.

The provisioning part of the job is accommodating extra resources on the network or upgrading the devices on the network. Suddenly if an entirely new device is introduced to the network, the network may stop functioning, so in order to avoid this the administrator needs to make extra provisions for possible devices at the beginning itself with precision.

Network Architecture

The network administrator needs to have a thorough understanding of the network architecture in order to be able to troubleshoot when there are problems. Most of the networks follow a similar architecture and function in the same way. The network architecture is designed in such a way that if the normal functioning is disrupted in any way then the network will send out an e-mail to the administrator, of any unwanted event, or shutdowns.

Since the administrator is notified of the problem it gets taken care of immediately. The disaster recovery is also apart of network planning and management. The network architecture itself plays a pro-active role in the network by aiding in the functioning of the network. The other crucial part of the network is the network protocol. Most networks use the Simple Network Management Protocol and some others use the Common Management Information Protocol.

Network Management History

By the time computers gained popularity, the demand for networks has also been increasing. Companies and people

started needing systems which would work in an environment and still required the controls to be in one place. As and when more devices were introduced it became difficult to add one individual device for each computer. Companies needed an easier way of managing the resources. A Network was the perfect answer; however networks were not easy and needed advanced working technology.

When the network initially began it was difficult, but today networks have advanced through software and topologies and there are many efficient methods to handle them.

There are many kinds of networks today like cable networks, wireless networks, digital networks, Satellite connections and all these networks work on similar network topologies. Companies use a combination of these networks for their functionalities. Based on these networks internet and intra company networks have promoted business to a large extent.

Fault Management-State of the Art

The terms *fault, error* and *failure* have often been confused. Incorrect interpretation of these terms may lead to their misuses. Definitions distinguishing them can be found in Wang's paper. A *fault* is a software or hardware defect in a system that disrupts communication or degrades performance. An *error* is the incorrect output of a system component. If a component presents an error, we say the component fails. This is a *component failure.*

Failure or component failure corresponds to the production of an error by this component. It is essential that

we distinguish the terms. The *fault* is the direct or indirect cause of the errors. The *errors* are manifestations of the *fault.* The *failure* is the overall result of the *fault.* However, if a component produces errors, we cannot conclude that there is a fault within the component.

Network faults can be characterised by several aspects such as their symptoms, propagation (transmission of an error from a component to another), duration in the network and severity. Though network faults can be distinguished through their characteristics, it is worth noting that it is difficult to measure these properties accurately since they are subjected to the manner in which they are controlled and managed. The occurrence of a fault may be detected by users through symptoms that may be produced by some network components as a result of error. Fault symptoms can be associated to four types of error-timing errors, timely errors, commission errors and omission errors.

These symptoms may take one of the following forms:

- *An output with an expected value comes either too early or too late*: This situation is due to a timing error. It is usually seen by users as a slow response or a time-out, when their applications are indirectly influenced by the effects of the faults.
- *An output with an unexpected value within the specified time interval*: This situation is due to a timely error which usually indicates a minor fault in the applications or underlying software and hardware.
- *An output with an unexpected value outside the specified time interval*: This is due to a commission error. If no response is produced, it is associated with

an omission error. An omission error can be regarded as a special case of commission error. A commission error usually implies a severe fault has occurred in the network.

If these symptoms are observed, there is a possibility that a fault has occurred somewhere in the network resulting in the components to produce erroneous outputs (errors). A fault in one component may have consequences on other associated components. Besides failing its own system, it may produce errors which could be transmitted to other components and degrade other systems as well. In this way, a fault in a single component may have global effects on the network. This phenomenon is referred to as fault propagation. A fault which occurs in an isolated systems may not affect other systems because there is no interaction between them. However, when the isolated systems are connected together by communications, errors produced by a fault may travel in a packet to other systems. A component can fail as a result of faults within it and the erroneous input produced by faults in other components.

This is one of the major characteristics of network faults. Media for fault propagation include parameter, data and traffic. The duration of a network fault, though recognised as one of its important criteria is somewhat difficult to measure.

This is due to three reasons. Firstly, a fault will not be perceived until it produces errors. Secondly, it may take a long time for a particular fault to be isolated. Finally, the effects of network faults may not be eliminated automatically when the faults are removed. They will remain for some time

until the operation is completely restored. Therefore, network faults can only be generally divided according to their duration into three groups: permanent, intermittent and transient. A permanent fault will exist in the network until a repair action has been taken.

This results in permanent maximum degradation of the service. An intermittent fault occurs in a discontinuous and a periodic manner. Its outcome will be failures in current processes. This implies maximum degradation of the service level for a short period of time. A transient fault will momentarily cause a minor degradation of the service. Faults of the first type will cause an event report to be sent out and changes made in the network configuration to prohibit further utilisation of this resource.

For a fault of the second type, the severity of the fault may transfer from being intermittent to being permanent if an excessive occurrence of this kind of fault becomes significant. Finally, a transient fault will usually be masked by the error recovery procedures of network protocols and therefore may not be observed by the users. It is fundamental for a designer of a fault management system to have knowledge of fault characteristics. This is because not all faults will have the same priority. The fault management system designer will have to decide which faults must be managed.

Fault Management Process

Recent literature suggests that a comprehensive fault management system is composed of monitoring, reporting, logging, trouble ticketing, filtering, correlating, diagnosis and recovery activities. The domain in which the fault

management system will operate. For the purpose, we have chosen to divide the activities associated with fault management into four major categories, namely detection, isolation, correction and administration. Error detection provides the capability to recognise faults. It consists of monitoring and reporting activities. Information provided by monitoring devices must be current, timely, accurate, relevant and complete. Reporting activity include investigation of critical criteria which require notification and a mechanism for report generation. It also involves determining appropriate destination for sending notifications.

Its purpose is to isolate the actual fault, given a number of possible hypotheses of faults. Testing may be the most appropriate way of isolating the fault at this stage. Isolation comprises four activities: filtration, interpretation, correlation and diagnosis. Filtration involves analysing management information in order to identify new faults or if the fault has occurred before, to update its count.

Filtration discards management information notifications that are of no significance and routes applicable notifications to their appropriate destinations within the system. Important information contained in the event reports must be extracted. Here the nature, structure and significance of the event report are examined.

Important information such as the name of the event report which normally represents the predefined condition that was met and triggers the generation of the event report itself. Other useful information is the time when the event report was generated. This information is important when performing correlation activity so that we may distinguish

which events are related, and which are not. Correlation proves to be helpful, when two or more notifications received are actually due to a single fault. Through correlation, some faults may be indirectly detected.

Hypotheses can be drawn from alarm correlation giving possible causes of fault. The objective of the diagnosis process is to isolate the cause of a fault down to a network resource. Given a set of probable causes, the diagnosis process is carried out. It involves identification and analysis of problems by gathering, examining and testing the symptoms, information and facts.

Once isolated, the effect of the fault must be minimized through bypass and recovery, and permanent repair instituted. Where applicable, steps must be taken to ensure that problems do not recur. This procedure consists of three activities: reconfiguration, recovery and restoration.

Reconfiguration or bypassing involves activating redundant resources specifically assigned to backup critical entities, suspending services, or re-allocating resources to more important uses. The objective is to reduce the immediate impact of the failure.

This function may be in a mixture of manual, semi-auto and automatic procedures. It may be possible for a fault to recover before any reconfiguration attempt is made. This depends on the nature of the failure, the criticality of the service and the expected time required to recover/ reconfigure. Once a fault has been rectified, the repair needs to be tested and the entity returned to service. This needs to be scheduled at an appropriate time and depends on the expected service disruption in doing so.

Fault administration service ensures that faults are not lost or neglected, but they are solved in a timeous fashion. This involves monitoring fault records, maintaining an archive of fault information, analysing trends, tracking costs, educating personnel and enforcing company policy with regards to problem resolution. It consists of three activities: logging, tracking and trend analysis. Logging maintains a log of event reports on faults that have occurred in the network.

This will be used for trend analysis, reporting and future diagnosis of the same type of or similar failures. Tracking keeps track of existing problems and persons responsible for and/or working on each one, facilitates communication between problem solving entities and prevents duplicate problem solving efforts.

This includes prioritisation of open faults due to their severity, and escalation of fault isolation or correction processes based on duration and severity of the faults. The current open fault records need to be ordered according to a priority scheme such that the most costly, or potentially costly failures are timeously resolved.

The whole activities may be accomplished using a trouble ticket system. Important information includes the frequency of occurrence of a particular failure and how much down time the various users are experiencing. Trends may indicate a need to redesign areas of the communication environment, replace inferior equipment, enhance problem solving expertise, acquire new problem solving tools, improve problem solving procedures, improve education, renegotiate service level agreements. In this project, all activities in fault

detection and isolation procedures and some aspects in recovery and administration procedures are implemented.

Typical Problems

One of the most critical problem associated with fault monitoring as given by Dupuy and Stallings is u*nobservable faults.* In this situation, certain faults are inherently unobservable through local observation. For example, the existence of a deadlock between co-operating distributed processes may not be observable locally. Other faults may not be observable because the vendor equipment is not instrumented to record the occurrence of a fault.

Other problems are defined by Fried and Tjong as follows. *Too many related observation:* A single failure can affect many active communication paths. The failure of a WAN back-bone will affect all active communication between the token-ring stations and stations on the Ethernet LANs, as well as voice communication between the PBXs.

Furthermore, a failure in one layer of the communications architecture can cause degradation or failures in all the dependent higher layers. This kind of failure is an example of propagation of failures. Because a single failure may generate many secondary failures, they may occur around the same time and may often obscure the single underlying problem.

Absence of automated testing tools: Testing to isolate faults is difficult, expensive and time consuming. It requires significant expertise in device behaviours and tools to pursue testing. Even such a simple task as tracing the progressions of packets along a virtual circuit is typically impossible to

accomplish. This leads to empirical rules of operation as "the only way to test if a virtual circuit is up, is to take it down".

The process of recovery typically involves a combination of automatic local resetting combined with manual activation of recovery procedures. Recovery presents a number of interesting technical challenges. Which include a*utomatic recovery as a source of fault:* Since most network devices or processes are designed to recover automatically from local failures, this can also be a source of faults. The problem in fault administration is the maintenance of fault reports log which has been made difficult due to the lack of functionalities for creating or deleting records. The logging facility is usually performed in a static manner, where logging characteristics are stagnant. Therefore, some important fault occurrences are unable to be recorded. If log attribute values can be changed, the logging behaviour may also be altered.

FMS: A Fault Management System Based on the OSI Standards

In order to overcome the problems, we have designed and implemented FMS. It offers three types of fault management applications as depicted namely the Fault Maintenance Application (FMA), the Log Maintenance Application (LMA), and the Diagnostic Test

Application (DTA). These fault management applications work together with the OSI Agent to perform fault management tasks on the network resources. The OSI Agent serves as a fault-monitoring agent. It has the capability to independently report errors to FMA. It can also issue a report when a monitored variable crosses a threshold. This allows

the FMS to anticipate faults. In addition, it maintains a log of events. These logs can be accessed and manipulated by LMA. DTA provides a set of diagnostic tests which may be invoked by the user. This facility is beneficial to the network administrator whenever there is any suspicion that some of the resources are not functioning as desired.

The paragraphs that follow explain how these facilities help in increasing fault management efficiency. The FMA solves the problems of unobservable fault due to ill-equipped vendor equipment to record the occurrence of a fault. This is done by monitoring the real resource critical properties and when significant events involving these properties occur, these events are reported to FMA so that further investigation is initiated.

FMA acts upon the receipt of these events by performing other fault management processes on them. These processes include event interpretation and filtration, event correlation, invocation of predefined diagnostic tests and initiating recovery process. Event or alarm correlation is used by the FMA to solve the problems of too many related observation. In addition, the reporting criteria is designed to be reasonably tight to reduce the volume of alarms received by the FMA. In accomplishing this objective, only events that require attention are reported.

Thus, in the FMS implementation, event reports are equal to alarms. Nevertheless, the objective to anticipate failure is neither neglected nor sacrificed. Hence, a number of managed objects are designed to issue event reports when the monitored attributes cross thresholds. Automated testing is provided in FMA by scheduling the execution of predefined

diagnostic tests for every event report that is received, so that the source of failure becomes apparent.

Subsequently, recovery action pertaining to the diagnosis is carried out automatically. On the other hand, the DTA provides a function that allows diagnostic tests to be invoked by the user. This facility proves to be beneficial to the experienced user who wants to skip trivial tests and choose only specific ones.

This results in lower consumption of the network resources for the purpose of management. The lack of adequate tools for systematic auditing is overcome by the supports provided by the LMA. The LMA has the capability to initiate error condition (events) logging. Furthermore, the LMA can access these logs and control logging behaviour by setting their log attribute values. Other supports include facilities for deleting logs and log records, and reviewing events (by reviewing log records) for diagnostic purpose or trend analysis.

Monolithic Versus Microkernel Network Operating System Designs

In the network world, both monolithic and microkernel designs can be used with success.

However, the ever-growing requirements for a system kernel quickly turn any classic implementation into a compromise. Most notably, the capability to support a real-time forwarding plane along with stateful and stateless forwarding models and extensive state replication requires a mix of features not available from any existing monolithic or microkernel OS implementation.

This lack can be overcome in two ways.

First, a network OS can be constrained to a limited class of products by design. For instance, if the OS is not intended for mid- to low-level routing platforms, some requirements can be lifted. The same can be done for flow-based forwarding devices, such as security appliances. This artificial restriction allows the network operating systems to stay closer to their general-purpose siblings—at the cost of fracturing the product lineup. Different network element classes will now have to maintain their own operating systems, along with unique code bases and protocol stacks, which may negatively affect code maturity and customer experience.

Second, the network OS can evolve into a specialized design that combines the architecture and advantages of multiple classic implementations.

This custom kernel architecture is a more ambitious development goal because the network OS gets further away from the donor OS, but the end result can offer the benefits of feature consistency, code maturity, and operating experience. This is the design path that Juniper selected for Junos OS.

Junos OS Kernel

According to the formal criteria, the Junos OS kernel is fully customizable. At the very top is a portion of code that can be considered a microkernel. It is responsible for real-time packet operations and memory management, as well as interrupts and CPU resources. One level below it is a more conventional kernel that contains a scheduler, memory manager and device drivers in a package that looks more like a monolithic design.

Finally, there are user-level (POSIX) processes that actually serve the kernel and implement functions normally residing inside the kernels of classic monolithic router operating systems. Some of these processes can be compound or run on external CPUs (or packet forwarding engines). In Junos OS, examples include periodic hello management, kernel state replication, and protected system domains (PSDs).

The entire structure is strictly hierarchical, with no underlying layers dependent on the operations of the top layers.

This high degree of virtualization allows the Junos OS kernel to be both fast and flexible.

However, even the most advanced kernel structure is not a revenue-generating asset of the network element.

Uptime is the only measurable metric of system stability and quality. This is why the fundamental difference between the Junos OS kernel and competing designs lies in the focus on reliability.

Coupled with Juniper's industry-leading nonstop active routing and system upgrade implementation, kernel state replication acts as the cornerstone for continuous operation. In fact, the Junos OS redundancy scheme is designed to protect data plane stability and routing protocol adjacencies at the same time. With in-service software upgrade, networks powered by Junos OS are becoming immune to the downtime related to the introduction of new features or bug fixes, enabling them to approach true continuous operation. Continuous operation demands that the integrity of the control and forwarding planes remains intact in the event

of failover or system upgrades, including minor and major release changes. Devices running Junos OS will not miss or delay any routing updates when either a failure or a planned upgrade event occurs.

This goal of continuous operation under all circumstances and during maintenance tasks is ambitious, and it reflects Juniper's innovation and network expertise, which is unique among network vendors.

Process Scheduling in Junos OS

Innovation in Junos OS does not stop at the kernel level; rather, it extends to all aspects of system operation.

As mentioned before, there are two tiers of schedulers in Junos OS, the topmost becoming active in systems with a software data plane to ensure the real-time handling of incoming packets. It operates in real time and ensures that quality of service (QoS) requirements are met in the forwarding path.

The second-tier (non-real-time) scheduler resides in the base Junos OS kernel and is similar to its FreeBSD counterpart. It is responsible for scheduling system and user processes in a system to enable preemptive multitasking.

In addition, a third-tier scheduler exists within some multithreaded user-level processes, where threads operate in a cooperative, multitasking model. When a compound process gets the CPU share, it may treat it like a virtual CPU, with threads taking and leaving the processor according to their execution flow and the sequence of atomic operations. This approach allows closely coupled threads to run in a cooperatively multitasking environment and avoid being entangled in extensive IPC and resource-locking activities.

The UNIX System

The UNIX system has been around for a long time, and many people may remember it as it existed in the previous decades. Many IT professionals who encountered UNIX systems in the past found it uncompromising. While its power was impressive, its command-line interface required technical competence, its syntax was not intuitive, and its interface was unfriendly.

Moreover, in the UNIX system's early days, security was virtually nonexistent. Subsequently, the UNIX system became the first operating system to suffer attacks mounted over the nascent Internet. As the UNIX system matured, however, the organization of security shifted from centralized to distributed authentication and authorization systems. Now, a single Graphical User Interface is shipped and supported by all major vendors has replaced command-line syntax, and security systems, up to and including B1, provide appropriate controls over access to the UNIX system.

The Value of Standards

The UNIX system's increasing popularity spawned the development of a number of variations of the UNIX operating system in the 1980s, and the existence of these caused a mid-life crisis. Standardization had progressed slowly and methodically in domains such as telecommunications and third-generation languages; yet no one had addressed standards at the operating system level. For suppliers, the thought of a uniform operating environment was disconcerting. Consumer lock-in was woven tightly into the fabric of the industry. Individual consumers, particularly

those with UNIX system experience, envisioned standardized environments, but had no way to pull the market in their direction.

However, for one category of consumer-governments-the standardization of the UNIX system was both desirable and within reach. Governments have clout and are the largest consumers of information technology products and services in the world. Driven by the need to improve commonality, both US and European governments endorsed a shift to the UNIX system.

The Institute of Electrical and Electronic Engineers POSIX family of standards, along with standards from ISO, ANSI and others, led the way. Consortia such as the X/Open Company (merged with the Open Software Foundation in 1995 to form The Open Group) hammered out draft standards to accelerate the process.

In 1994, the definitive specification of what constitutes a UNIX system was finalized through X/Open Company's consensus process. The Single UNIX Specification was born-not from a theoretical, ivory tower approach, but by analysing the applications that were in use in businesses across the world. With the active support of government and commercial buyers alike, vendors began to converge on products that implement the Single UNIX Specification, and now all major vendors have products labeled UNIX 95, which indicates that the vendor guarantees that the product conforms to the Single UNIX Specification.

Vendors continue to add value to the UNIX system, particularly in areas of new technology, however that value will always be built upon a single, consensus standard.

Meanwhile, the functionality of the UNIX system was established and the mid-life crisis was resolved. Suppliers today provide UNIX systems that are built upon a single, consensus standard. It is also important to remember that even when variance among UNIX systems was at its worst, IT professionals agreed that migration among UNIX system variants was far easier than migration among the proprietary alternatives.

Now with UNIX 95 branded products available from all major systems vendors, the buyer can for the first time buy systems from different manufacturers, safe in the knowledge that each one is guaranteed to implement the complete functionality of the Single UNIX Specification and will continue to do so. UNIX system suppliers can assure customers that they own a standards-based system by registering them to use the Open Brand. Below is a list of suppliers who give users this guarantee.

UNIX and Windows NT

It is common these days to read analysts' accounts and IS professionals' experiences that compare and contrast the UNIX system with Microsoft Corporation's latest operating system, called Windows NT. Opinions vary, of course, but a number of common themes have emerged. The key differences between these operating environments are as follows: The UNIX system today is more robust, reliable and scalable. Analysts say this observation, which is widely reported from many different viewpoints, makes practical sense. Engineers at Microsoft are retracing the steps that the UNIX system has completed. How else could it be?

In sharp contrast to the open standards that define the UNIX system, Windows NT technology remains fiercely proprietary. Microsoft remains ambivalent to the world of standards. Choosing NT entangles customers with nonstandard utilities, directories, and software tools that do not conform to any de jure or consensus standards. The UNIX system today is available on a wide spectrum of computer hardware.

Particularly when high performance is at issue, hardware suppliers suggest the UNIX system, rather than Windows NT. The primary appeal of NT is for low-end, office-centered, departmental applications. Unit shipment growth rates for Windows NT exceed the rates for the UNIX system, which is to be expected for a new product. However, revenue growth in UNIX systems sales is much higher than NT. It is reasonable to expect Windows NT to take a share in the operating systems market, along with other more specialized operating systems. There is no evidence today to indicate that NT will be dominant; in fact, most IT professionals predict that it will not.

Windows NT Server 4.0 is still not a full-function server operating system. While it does support multi-user computing via third-party add-on tools, it lacks certain fundamental features that the UNIX system is known for providing, such as directory services for managing user access and peripherals over a distributed enterprise network. The presence of the UNIX system in the marketplace has been good for Windows NT. The UNIX system established the market for cross-platform client and server operating environments that NT seeks to address. In turn, NT will

improve the market for UNIX systems in the future. That is, competition among UNIX system providers will be enhanced by competition with NT. The choice between open and proprietary products will be quite crisp.

Today's UNIX System

The key to the continuing growth of the UNIX system is the free-market demands placed upon suppliers who produce and support software built to public standards. The "open systems" approach is in bold contrast to other operating environments that lock in their customers with high switching costs. UNIX system suppliers, on the other hand, must constantly provide the highest quality systems in order to retain their customers. Those who become dissatisfied with one UNIX system implementation retain the ability to easily move to another UNIX system implementation.

The continuing success of the UNIX system should come as no surprise. No other operating environment enjoys the support of every major system supplier. Mention the UNIX system and IT professionals immediately think not only of the operating system itself, but also of the large family of hardware and application software that the UNIX system supports. In the IT marketplace, the UNIX system has been the catalyst for sweeping changes that have empowered consumers to seek the best-of-breed without the arbitrary constraints imposed by proprietary environments. The market's pull for the UNIX system was amplified by other events as well. The availability of relational database management systems, the shift to the client/server architecture, and the introduction of low-cost UNIX system

servers together set the stage for business applications to flourish. For client/server systems, the networking strengths of the UNIX system shine. Standardized relational database engines delivered on low-cost high-performance UNIX system servers offered substantial cost savings over proprietary alternatives.

UNIX System Existence

There is every reason to believe that the UNIX system will continue to be the platform of choice for innovative development. In the near term, for example, UNIX system vendors will define the scope of Java and provide the distributed computing environment into which the Network Computer terminal will fit and enable it to thrive and grow.

How will Java and the Network Computer terminal manifest themselves? The exact answer is unknown; however, in open computing, the process for finding that answer is well understood. The UNIX system community has set aside (via consensus standards) the wasteful task of developing arbitrary differences among computing environments. Rather than building proprietary traps, this community is actively seeking ways to add value to the UNIX system with improved scalability, reliability, price/performance, and customer service. Java and the Network Computer terminal offer several potential advantages for consumers. One key advantage is a smaller, lighter, standards-based client. A second advantage is a specification that is not controlled by one company, but is developed to the benefit of all by an open, consensus process. Thirdly, greater code reuse and a component software market based

on Object technology, such as CORBA and Java. All of these options and more are being deployed first by members of the UNIX system community.

Industrial Strength UNIX System

Today's UNIX system is robust, scalable, and it continues to provide uniform access to a wide variety of computing hardware. For these reasons the UNIX system continues to be the operating system of choice for mission-critical systems. The UNIX system is the key enabler for enterprises that wish to keep switching costs as low as possible. That is, the UNIX system remains the only open alternative to locking in on a proprietary operating system.

Scalability is here today, enabling application to run on small-scale systems through to the largest servers necessary. The UNIX system is available on hardware ranging from low-cost PC-class servers on through parallel architectures that harness together 60 or more processors. This range is wider and the choices of hardware more cost effective than any other system. The UNIX system is the only option for Massively Parallel Processing (MPP).

A robust operating system is tough enough to perform successfully under a variety of different operating conditions. By virtue of its worldwide deployment by an international community of system vendors, the UNIX system has earned the reputation for robustness. Uniform operating system services are at the heart of the standardized UNIX system. Many enterprise systems are assembled with hardware from several different sources. Atop these different hardware platforms, the UNIX operating system provides a uniform

platform for database management systems and application software. The market for the UNIX system continues to expand.

IDC estimates the market at US$ 39 billion in 1996 and forecasts the market to be US$ 50 billion in the year 2000. In addition, the installed base of the UNIX system has an estimated value of US$ 122 billion. These market estimates lead to several conclusions about the UNIX system, as follows: An annual market of US$ 39 billion is large enough to remain attractive to many suppliers and to provide sufficient revenue to fund continuing high levels of investment in support and product enhancement.

The UNIX system's growth rates, which appear modest in comparison to the unit shipment growth of newer products, are anchored by an enormous installed base. High unit shipment growth rates are typical of new entries in a marketplace. In key benchmarks and mission-critical applications, the UNIX system consistently performs better. The UNIX system is the dominant software platform for Relational Database Management Systems. Investment in developing and enhancing UNIX system products is significantly larger than in any other operating environment.

Single UNIX Specification

Today, The Open Group's UNIX 95 brand may be applied to any operating system product that is guaranteed to meet the Single UNIX Specification. The Single UNIX Specification is designed to give software developers a single set of APIs to be supported by every UNIX system. The most significant

consequence of the Single UNIX Specification initiative is that it shifts the focus of attention away from incompatible UNIX system product implementations on to compliance with a single, agreed-upon set of APIs. If an operating system meets the specification, and commonly available applications can run on it, then it can reliable viewed as open.

So, the future looks as though it will be about a set of sturdy and dependable specifications standing as a firm foundation upon which many competing product implementations will be built.

By developing a single specification for the UNIX system, The Open Group and the computer industry have completed the foundation of open systems. The next version of the Single UNIX Specification, known as Version 2 was announced in March 1997. Products guaranteed to conform to this specification will carry the label UNIX 98.

Single UNIX Specification

The Single UNIX Specification, Version 2 contains the following enhancements:

- Year 2000 Alignment-changes to minimize the impact of the Millennium Rollover.
- *Threads*: POSIX 1003.1c-1995. The Threads extensions permit development of applications to make significant performance gains on multiprocessor hardware.
- Large File Summit extensions to permit UNIX systems to support files of arbitrary sizes, this is of particular relevance to database applications.

- *Networking Services*: The specifications are aligned with the POSIX 1003.1g standard.
- *MSE*: The Multibyte Support Extension is now aligned with ISO C amendment 1, 1995.
- Dynamic linking extensions to permit applications to share common code across many applications, and ease maintenance of bug fixes and performance enhancements for applications.
- N-bit cleanup (64 bit and beyond), to remove any architectural dependencies in the Single UNIX Specification. This is of particular relevance with the ongoing move to 64 bit CPUs.
- The real-time extensions are an optional feature group, allowing procurement of X/Open real-time systems with predictable, bounded behaviour.
- Inclusion of the existing specifications for the graphical user interface, CDE as an option in the UNIX 98 brand.

Benefits for Application Developers

- Improved portability.
- Faster development through the increased number of standard interfaces.
- More innovation is possible, due to the reduced time spent porting applications.

Benefits for Users

The Single UNIX Specification will evolve and develop in response to market needs protecting users investment in existing systems and applications. The availability of the

UNIX system from multiple suppliers gives users freedom of choice rather than being locked in to a single supplier. And the wide range of applications-built on the UNIX system's strengths of scalability, availability and reliability-ensure that mission critical business needs can be met.

5

Windows Operating System

Microsoft first started developing the Interface Manager (later renamed Microsoft Windows) in September 1981. While the first prototypes used Multiplan and Word-like menus at the bottom of the screen, the interface was modified in 1982 to utilize pull-down menus and dialogues, as used on the Xerox Star.

On November 10, 1983, at the Plaza Hotel in New York City, Microsoft Corporation officially announced Microsoft Windows, a next-generation operating system that would supply a graphical user interface (GUI) and a multitasking environment for IBM computers.

This was after the release of the Apple Lisa, and prior to Digital Research announcing GEM, and DESQ from Quarterdeck and the Amiga Workbench , or GEOS/GeoWorks Ensemble, IBM OS/2, NeXTstep or even DeskMate from

Tandy. Windows assured an easy-to-use graphical interface, device-independent graphics and multitasking support. The development was held up several times, however, and the Windows 1.0 arrive at store shelves in November 1985. The selection of applications was thin, however, and Windows sales were moderate.

Windows 1.0 package, included: MS-DOS Executive, Calendar, Cardfile, Notepad, Terminal, Calculator, Clock, Reversi, Control Panel, PIF (Programme Information File) Editor, Print Spooler, Clipboard, RAMDrive, Windows Write, Windows Paint.

Microsoft Windows version 1.0 was viewed as buggy, rough, and sluggish. This bumpy start was made sorrier by a threatened lawsuit from Apple Computers. In September 1985, Apple lawyers warned Bill Gates that Windows 1.0 infringed on Apple copyrights and patents, and that his corporation had stolen Apple's trade secrets.

Microsoft Windows had similar drop-down menus, tiled windows and mouse support. Bill Gates and his head counsel Bill Neukom, chose to make an offer to license features of Apple's operating system. Apple concurred and a contract was drawn up.

The key to this is: Microsoft wrote the licensing agreement to include use of Apple features in Microsoft Windows version 1.0 and all future Microsoft software programmes. As it turned out, this move by Bill Gates was as intelligent as his decision to purchase QDOS from Seattle Computer Products and his persuading IBM to let Microsoft keep the licensing rights to MS-DOS.

Windows 1.0 fluttered on the market until January 1987, when a Windows-compatible programme called Aldus PageMaker 1.0 was brought out. PageMaker was the first WYSIWYG desktop-publishing programme for the PC. Later that year, Microsoft released a Windows-compatible spreadsheet called Excel.

Other popular and functional software like Microsoft Word and Corel Draw helped advance Windows, yet, Microsoft recognized that Windows required further development. Microsoft Windows version 2.0 came out in December 1987, and evidenced somewhat more popular than its predecessor. A great deal of the popularity for Windows 2.0 came through its inclusion as a "run-time version" with Microsoft's new graphical applications, Excel and Word for Windows.

They could be operated from MS-DOS, executing Windows for the length of their activity, and shutting down Windows upon exit.

Microsoft Windows incurred a major boost about this time when Aldus PageMaker came out in a Windows version, having previously run exclusively on Macintosh. Some computer historians date this, the introduction of a significant and non-Microsoft application for Windows, as the start of the success of Windows.

Versions 2.0x used the real-mode memory model, which held it to a maximum of 1 megabyte of memory. In such a form, it could run under a different multitasker like DESQview, which used the 286 Protected Mode. Subsequently, two new versions were published: Windows/286 2.1 and Windows/386 2.1.Like preceding versions of

Windows, Windows/286 2.1 used the real-mode memory model, but was the first version to support the HMA.

Windows/386 2.1 had a protected mode kernel with LIM-standard EMS emulation, the predecessor to XMS which would at last shift the topology of IBM PC computing. All Windows and DOS-based applications at the time were real mode, running over the protected mode kernel by using the virtual 8086 mode, which was new with the 80386 processor. After Versions 2.0 and beyond were released, Apple began filing suit against IBM for various copyright infringements. In their defence, Microsoft claimed that their licensing agreement actually afforded them the rights to use Apple features.

Apple claimed that Microsoft had encroached on 170 of their copyrights. After a four-year court case, Microsoft won. The courts stated that the licensing agreement gave Microsoft the rights to use all but nine of the copyrights, and Microsoft later convinced the courts that the other nine copyrights shouldn't be covered by copyright law. Bill Gates asserted that Apple had acquired ideas from the graphical user interface developed by Xerox for Xerox's Alto and Star computers.

On June 1, 1993, Judge Vaughn R. Walker of the U.S. District Court of Northern California ruled in Microsoft's favour in the Apple vs. Microsoft & Hewlett-Packard copyright suit. The judge granted Microsoft's and Hewlett-Packard's motions to dismiss the last remaining copyright infringement claims against Microsoft Windows versions 2.03 and 3.0, as well as HP NewWave.

What would have occurred if Microsoft had lost the lawsuit? Microsoft Windows might never have become the predominant operating system that it is today.

Features

Some individuals with disabilities require assistive technology (AT) in order to access computers. Hundreds of Windows AT third-party products are available, making it possible for almost anyone to use Windows® applications, regardless of their disabilities. The Microsoft® Windows® operating systems also provides a core set of basic accessibility features and AT applications, which can be deployed on all computers in a computer lab or classroom without additional cost. These applications provide students with basic accessibility features from any workstation, maximizing the inclusiveness of the learning environment.

It should be noted that the AT applications that are bundled with Windows provide only a minimum level of accessibility, not the full set of features that many users require for equal access to the operating system, educational programmes, and other software applications. Therefore, many educational entities deploy the standard set of Windows AT on all workstations by default, but addditionally 1) provide a small number of dedicated workstations that are equipped with commonly requested third party AT, and 2) are prepared to purchase and install additional AT as needed by specific students.

It should also be noted that the availability of AT does not itself guarantee accessibility. Software applications must be designed in a way that is compatible with AT and other

accessibility features of the operating system. The following is a list of basic accessibility features that are included with Windows XP. Previous versions of Windows also included several of these same features.

Display and Readability

These features are designed to increase the visibility of items on the screen.

- Font style, colour, and size of items on the desktop—using the Display options, choose font colour, size and style combinations.
- Icon size—make icons larger for visibility, or smaller for increased screen space.
- Screen resolution—change pixel count to enlarge objects on screen.
- High contrast schemes—select colour combinations that are easier to see.
- Cursor width and blink rate—make the cursor easier to locate, or eliminate the distraction of its blinking.
- Microsoft Magnifier—enlarge portion of screen for better visibility.

Sounds and Speech

These features are designed to make computer sounds easier to hear or distinguish - or, visual alternatives to sound. Speech-to-text options are also available.

- *Sound Volume*: Turn computer sound up or down.
- *Sound Schemes*: Associate computer sounds with particular system events.
- *ShowSounds*: Display captions for speech and sounds.

- *SoundSentry*: Display visual warnings for system sounds.
- *Notification*: Get sound or visual cues when accessibility features are turned on or off.
- *Text-to-Speech*: Hear window command options and text read aloud.

Keyboard and Mouse

These features are designed to make the keyboard and mouse faster and easier to use.

Mouse Options:

- *Double-Click Speed*: Choose how fast to click the mouse button to make a selection.
- *ClickLock*: Highlight or drag without holding down the mouse button.
- *Pointer Speed*: Set how fast the mouse pointer moves on screen.
- *SnapTo*: Move the pointer to the default button in a dialog box.
- *Cursor Blink Rate*: Choose how fast the cursor blinks—or, if it blinks at all.
- *Pointer Trails*: Follow the pointer motion on screen.
- *Hide Pointer While Typing*: Keep pointer from hiding text while typing.
- Show Location of Pointer: Quickly reveal the pointer on screen.
- *Reverse the function of the right and left mouse buttons*: Reverse actions controlled by the right and left mouse buttons.

- *Pointer schemes*: Choose size and colour options for better visibility.

Keyboard Options

- *Character Repeat Rate*: Set how quickly a character repeats when a key is struck.
- *Dvorak Keyboard Layout*: Choose alternative keyboard layouts for people who type with one hand or finger.
- *Sticky Keys*: Allow pressing one key at a time (rather than simultaneously) for key combinations.
- *Filter Keys*: Ignore brief or repeated keystrokes and slow down the repeat rate.
- *Toggle Keys*: Hear tones when pressing certain keys.
- *MouseKeys*: Move the mouse pointer using the numerical keypad.
- *Extra Keyboard Help*: Get ToolTips or other keyboard help in programmes that provide it.

Accessibility Wizard

The Accessibility Wizard is designed to help new users quickly and easily set up groups of accessibility options that address visual, hearing and dexterity needs all in one place.

The Accessibility Wizard asks questions about accessibility needs. Then, based on the answers, it configures utilities and settings for individual users.

The Accessibility Wizard can be run again at any time to make changes, or changes can be made to individual settings through Control Panel.

Structure

System Components

Even though, not all systems have the same structure many modern operating systems share the same goal of supporting the following types of system components.

Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, registers, OS resources in use etc.). It is important to note that a process is not a programme. A process is only ONE instant of a programme in execution. There are many processes can be running the same programme.

The five major activities of an operating system in regard to process management are:

- Creation and deletion of user and system processes.
- Suspension and resumption of processes.
- A mechanism for process synchronization.
- A mechanism for process communication.
- A mechanism for deadlock handling.

Main-Memory Management

Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be access directly by the CPU. That is to say for a programme to be executed, it must in the main memory.

The major activities of an operating in regard to memory-management are:

- Keep track of which part of memory are currently being used and by whom.
- Decide which process are loaded into memory when memory space becomes available.
- Allocate and deallocate memory space as needed.

File Management

A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, data transfer rate and access methods. A file systems normally organized into directories to ease their use. These directories may contain files and other directions.

The five main major activities of an operating system in regard to file management are:

- The creation and deletion of files.
- The creation and deletion of directions.
- The support of primitives for manipulating files and directions.
- The mapping of files onto secondary storage.
- The back up of files on stable storage media.

I/O System Management

I/O subsystem hides the peculiarities of specific hardware devices from the user. Only the device driver knows the peculiarities of the specific device to whom it is assigned.

Secondary-Storage Management

Generally speaking, systems have several levels of storage, including primary storage, secondary storage and cache storage. Instructions and data must be placed in primary storage or cache to be referenced by a running programme. Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide secondary storage to back up main memory. Secondary storage consists of tapes, disks, and other media designed to hold information that will eventually be accessed in primary storage (primary, secondary, cache) is ordinarily divided into bytes or words consisting of a fixed number of bytes. Each location in storage has an address; the set of all addresses available to a programme is called an address space.

The three major activities of an operating system in regard to secondary storage management are:

- Managing the free space available on the secondary-storage device.
- Allocation of storage space when new files have to be written.
- Scheduling the requests for memory access.

Networking

A distributed systems is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network. The communication-network design must consider routing and connection strategies, and the problems of contention and security.

Protection System

If a computer systems has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by a computer systems.

Command Interpreter System

A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode.

Operating Systems Services

Following are the five services provided by an operating systems to the convenience of the users.

Programme Execution

The purpose of a computer systems is to allow the user to execute programs. So the operating systems provides an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems.

Running a programme involves the allocating and deallocating memory, CPU scheduling in case of

multiprocess. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

I/O Operations

Each programme requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So the operating systems by providing I/O makes it convenient for the users to run programs. For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

File System Manipulation

The output of a programme may need to be written into new files or input taken from some files. The operating systems provides this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his her task accomplished. Thus operating systems makes it easier for user programs to accomplished their task.

This service involves secondary storage management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence I think it is best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this service s left with operating system.

Communications

There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes.

In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user programme may be customised to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Error Detection

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

System Calls and System Programs

System calls provide an interface between the process an the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. In handling the trap, the operating system will enter in the kernel mode, where it has access to privileged instructions, and can perform the desired service on the behalf of user-level process.

It is because of the critical nature of operations that the operating system itself does them every time they are needed. For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

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

Layered Approach Design

In this case the system is easier to debug and modify, because changes affect only limited portions of the code, and programmer does not have to know the details of the other layers.

Information is also kept only where it is needed and is accessible only in certain ways, so bugs affecting that data are limited to a specific module or layer.

Mechanisms and Policies

The policies what is to be done while the mechanism specifies how it is to be done. For instance, the timer construct for ensuring CPU protection is mechanism. On the other hand, the decision of how long the timer is set for a particular user is a policy decision.

The separation of mechanism and policy is important to provide flexibility to a system. If the interface between mechanism and policy is well defined, the change of policy may affect only a few parameters. On the other hand, if

interface between these two is vague or not well defined, it might involve much deeper change to the system.

Structure of Operating Systems

As modern operating systems are large and complex careful engineering is required. There are four different structures that have shown in this document in order to get some idea of the spectrum of possibilities. These are by no mean s exhaustive, but they give an idea of some designs that have been tried in practice.

Monolithic Systems

This approach well known as "The Big Mess". The structure is that there is no structure. The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

For constructing the actual object programme of the operating system when this approach is used, one compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file with the linker.

In terms of information hiding, there is essentially none-every procedure is visible to every other one *i.e.* opposed to a structure containing modules or packages, in which much of the information is local to module, and only officially designated entry points can be called from outside the module.

Layered System

A generalization of the approach for organizing the operating system as a hierarchy of layers, each one constructed upon the one below it. The system had 6 layers. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

Layer 1: Did the memory management. It allocated space for processes in main memory and on a 512k word drum used for holding parts of processes (pages)for which there was no room in main memory. Above layer 1, processes did not have to worry about whether they were in memory or on the drum; the layer 1 software took care of making sure pages were brought into memory whenever they were needed.

Layer 2: Handled communication between each process and the operator console. Above this layer each process effectively had its own operator console. Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities. Layer 4 was where the user programs were found. They did not have to worry

about process, memory, console, or I/O management. The system operator process was located I layer 5.

Virtual Machines

The heart of the system, known as the virtual machine monitor, runs on the bare hardware and does the multiprogramming, providing not one, but several virtual machines to the next layer up. However, unlike all other operating systems, these virtual machines are not extended machines, with files and other nice features. Instead, they are exact copies of the bare hardware, including kernel/ user mod, I/O, interrupts, and everything else the real machine has.

For reason of Each virtual machine is identical to the true hardware, each one can run any operating system that will run directly on the hard ware. Different virtual machines can, and usually do, run different operating systems. Some run one of the descendants of OF/360 for batch processing, while other ones run a single-user, interactive system called CMS (conversational Monitor System) fro timesharing users.

Client-server Model

A trend in modern operating systems is to take this idea of moving code up into higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel.

The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (presently

known as the client process) sends the request to a server process, which then does the work and sends back the answer.

In client-Server Model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one fact of the system, such as file service, process service,

Terminal service, or memory service, each part becomes small and manageable; furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Types of Operating System

Microsoft Windows isn't the only operating system for personal computers, or even the best... it's just the best-distributed. Its inconsistent behaviour and an interface that changes with every version are the main reasons people find computers difficult to use. Microsoft adds new bells and whistles in each release, and claims that this time they've solved the countless problems in the previous versions... but the hype is never really fulfilled. Windows 7 offers little new: it's basically Vista without quite so many mistakes built into it. The upgrade prices serve primarily to keep the cash flowing to Microsoft, to subsidize their efforts to take over other markets. A slew of intrusive "features" in the recent versions benefit Microsoft at the

expensive of both your privacy and your freedom. Switching to Windows Vista or Windows 7 requires buying new hardware and learning a new system, so instead consider switching to something *better*. More than 1 in 10 people on the web already have.

- ≅ ç If you can't say "no" to Windows (which is understandable in many cases), you can still say "no more". The simplest alternative to Windows Vista/7 is a previously-installed version of Windows. Windows Vista/7 isn't a simple upgrade; it's a drastically different operating system, which may not even run your existing software, or work properly on hardware just a few years old, so installing the "upgrade" is a risk.

  The Mac OS user interface inspired the creation of Windows, and is still the target Microsoft is trying to equal. As a popular consumer product, there's plenty of software available for it, and it's moving beyond its traditional niches of graphic design, education, and home use, into general business use (after all, Apple Corp. runs on it). OS X (ten), uses Unix technology, which makes it more stable and secure than Windows. But the real star is OS X's visual interface, which shows the difference between Microsoft's guesswork in this area and Apple's innovative *design work*: it's both beautiful and easy to use.
- ç⊘ Linux ("LIH-nux") is a free Unix-like operating system, originally developed by programmers who who simply love the challenge of solving problems and producing quality software… even if that means giving the resulting product away. Not coincidentally, there's

also a wealth of free software for it. Unlike proprietary operating systems, which are usually controlled in every detail by a single company, Linux has a standard consistent core (called the "kernel") around which many varieties (known as "distributions") have been produced by various companies and organisations. Some are aimed at geeks, some focus on the needs of business users, and some are designed with typical home users in mind. Businesses might prefer RedHat/ Fedora, Novell/SUSE, or CentOS. Geeks should check out Debian, Slackware, and Gentoo. Linux is a first-rate choice for servers; this site is a Linux system

- ç⊘ Google's Chrome OS is still vapourware so far, and it's arguably just another flavour of Linux, but it promises to be a viable alternative to Windows on small portable "netbooks" which will come with it preinstalled. The user interface is going to be based on Google's web browser of the same name, and take advantage of technology to make online apps like Google Docs work even if you're not online.
- ≅ ç BeOS was designed with multimedia in mind, including the kinds of features that Microsoft is just recently tacking onto Windows. Although Microsoft successfully drove Be Corp. out of business through illegal interference with their marketing efforts, reports of BeOS's death are exaggerated: The source code for BeOS has been licensed to a European software firm whose Zeta is effectively the much-longed-for BeOS R6. The free BeOS R5 Personal Edition is still available to download, and has been packaged with all the

latest drivers and free add-ons as BeOS Max Edition. And the Haiku project is creating an open duplicate of BeOS R5, which will then be enhanced

- çø FreeBSD is commonly called "the free Unix". It's descended from the classic 1970's Berkeley Software Distribution of Unix (from before the OS became "UNIX"®), making it one of the most mature and stable operating systems around. It's "free" as in "free beer" (you can download it for nothing) and as in "free speech" (you can do pretty much whatever you like with it... like when Microsoft took code from it to add better networking to Windows NT).
- çø OpenBSD is "the other free Unix". It's similar to FreeBSD both in the Berkeley code it's based on, and the licensing terms. One key advantage it has over its BSD siblings (and nearly any other OS) is that it's incredibly secure from attack, as implied by its blowfish mascot, and made explicit by their boast of only one remotely-exploitable hole-ever-in their default installation. (Compare that to Windows' hundreds.) "Open" is a reference to their code auditing process, not a welcome-mat forcrackers. It's not as speedy as FreeBSD, but it's safer. It's also available for some hardware plaforms FreeBSD doesn't support, including Mac 68K, PPC.
- çø NetBSD is "the *other* other free Unix". It's the work of another group of volunteer developers using the net to collaborate (hence the name of their product). Their mission is to get the OS to run-and run *well*-on hardware platforms no other Unix supports. In addition

to most of the usual suspects above, it's been ported to run on the NeXT box, MIPSmachines, the good Atari computers, the BeBox, WinCE-compatible handhelds, ARM processors, and even game machines like the Playstation 2 or the orphaned Sega Dreamcast.

- ç⃠ Darwin is a cousin of Free/Open/NetBSD, and the free foundation on which the commercial Mac OS X is built. Although its development was originally managed rather tightly by Apple (understandable, because their business depends on it) they've loosened the leash, making participation in the development more open. Darwin is making progress towards becoming an open-source OS in its own right.
- ç Syllable is a free alternative OS for standard PCs. It uses some of the better ideas from Unix, BeOS, AmigaOS, and others, and is compatible enough with portable software written for Unix that many have already been ported over to it. It's not a full-featured OS yet, but it's functional enough to be used with built-in web and e-mail clients, and media players.
- Amiga owners used to taunt PC and Mac users with their smoothly-multitasking graphical operating system, back when the Macs couldn't multitask, and PCs weren't even graphical. Even though the "classic" Amiga machines are no longer being produced, there's been a lot of activity in Amigaspace in the meantime: The OS has been updated to support current technology with Amiga OS 4, emulation layers called AmigaOS XL and AMIthlon were created to run Amiga OS on modern PC hardware, Amiga Forever is an emulator for

Windows and other operating systems, and a new hardware platform and OS called AmigaOne have been introduced to try to carry on the Amiga legacy.

- MorphOS began as a project to port the Amiga OS to the then-new PowerPC architecture, but has since morphed into an OS in its own right. It runs on certain PowerPC/G3/G4-based systems, and has better-than-standard-emulation support for Amiga OS 3.1 applications as well as native apps built for MorphOS.
- RISC OS is the operating system of the former Acorn line of computers (best known in the UK), which has been revived and updated for faster performance and to meet current OS standards (*e.g.* long filenames, large hard drives). It doesn't run on standard PCs, but on systems specifically designed for it (such as the RiscPC and A7000), using the high-speed StrongARM processors. The OS itself is stored in electronic ROM rather than having to be loaded into RAM from a hard drive
- ç⃠ GNU's Not Unix. In fact, that phrase is what G.N.U. is a (recursive) abbrevation for. It is a Unix-like operating system being developed as a long-term project by the Free Software Foundation to offer a fully-free alternative to the commercial and BSD versions of Unix. Although you'll find many key components of GNU used in Linux and BSD packages under the GNU General Public license (GPL), a fully GNU system will use the Hurd, GNU's own free-software kernel. The Hurd has some design advantages over the Linux kernel, but is still far from finished, and requires serious expertise with OS development to install.

- ç⊘ Minix is an open-source Unix-like operating system originally developed for educational purposes. Because of its relative simplicity and ample documentation, its creator says that a few months studying the source code should teach you most of how such things work. (It inspired Linus Torvalds to create Linux.) Versions 1 and 2 serve primarily as teaching examples, but version 3 has also become useful in its own right, intended for highly reliable uses on low-end 386-level hardware.
- ⊘ There are also a bunch of commercial UNIX systems, which are typically customised to run on expensive, high-end, proprietary hardware sold by the same vendor. Most of them have names other than "Unix" due to old trademark issues. They're better as alternatives to the server versions of Windows, not the desktop versions of Windows such as 98/XP/Vista.
- ≅ IBM's OS/2warp was once supposed to replace MS Windows, back when Emperor IBM and Darth Microsoft were planning to rule the galaxy together. Then Darth decided he didn't need the Emperor, struck confidential deals with other hardware vendors and software developers, and made Windows (just barely) powerful enough to fill OS/2's intended role. Windows didn't really beat OS/2 technically, but it won the Marketing Wars, which is what mattered. Unfortunately, IBM has given up on OS/2's future. A third-party package called eComStation is a licensed effort to update and maintain OS/2.

- ç Believe it or not, DOS (with or without Windows 3.1) is still a viable option for many uses. There was an incredible amount of software developed for it, and it still works. Plus, DOS runs like a champ, on old hardware that no one else wants. You can even fit it on a diskette, to boot it on nearly any PC anywhere.