FUNDAMENTALS OF OPERATING SYSTEM

Dr. Praveen Gujjar
Dr. G. S. Vijaya

BOOKS ARCADE

Fundamentals of Operating System

Fundamentals of Operating System

Dr. Praveen Gujjar
Dr. G. S. Vijaya

BOOKS ARCADE
KRISHNA NAGAR, DELHI

Fundamentals of Operating System

Dr. Praveen Gujjar
Dr. G. S. Vijaya

BOOKS ARCADE

Regd. Office:
F-10/24, East Krishna Nagar, Near Vijay Chowk, Delhi-110051
Ph. No: +91-11-79669196, +91-9899073222
E-mail: info@booksarcade.co.in, booksarcade.pub@gmail.com
Website: www.booksarcade.co.in

Year of Publication 2023

International Standard Book Number-13: 978-81-19199-42-6

CONTENTS

CHAPTER 1

OPERATING-SYSTEM STRUCTURES

Dr. Praveen Gujjar, Associate Professor,
Department of Business Analytics,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id:-dr.praveengujjar@cms.ac.in

Operating Systems

An operating system acts as a conduit between a computer's user and its hardware. An operating system's objective is to offer a setting in which a user can efficiently and conveniently execute arbitrary.

A piece of software called an operating system supervises computer hardware. In order to safeguard the proper operation of the computer system and prevent software components from interfering with that operation, the hardware must offer the infrastructure needed.

1. An operating system is a component of software that manages the execution of application programmes and serves as a conduit between a computer's user and its hardware.
2. The operating system is typically referenced to as the kernel and is the only program that runs persistently on a computer. All other programs are application programs, in accordance with a more widely accepted definition.
3. The distribution of services and resources including as memory, processors, devices, and information, is a concern of an operating system. To handle these resources, the operating system has programs like a traffic controller, a scheduling, a memory management component, I/O programs, and a file system.

Operating-System Structure

The majority of enterprise systems lack a clear structure, and their operating systems are typically compact, uncomplicated, and rudimentary. Only a limited number of people can use MS-straightforward DOS's operating system framework for a particular effect. For instance:

MS-DOS

1. The OS's above straightforward structure is typically applied to desktop computers, where users can carry out a variety of functions.
2. Levels of functionality and terminals are not typically distinguished. Access to I/O routines allows application programmes to write directly to the disc and display drivers.
3. As a result, MS-DOS is sensitive to rogue software, which can lead to system crashes when a user programme malfunctions.

The OS's structure and functioning.

1. The capacity to multiprogramming is one of the key ideas in an OS.
2. The OS may execute a variety of procedures, and each operation is utilised to accomplish a particular duty because inputs and outputs cannot carry out any work without the help of OS instructions.
3. The memory setup procedures might wait for the primary storage to accumulate on disc.

The Operating System uses a structure and that without one it cannot function because it needed to handle things like the CPU, disc, USB, graphics, and memory, all of which are interconnected to the OS by means of a structure. The operating control framework is shown in the following Figure 1.1:

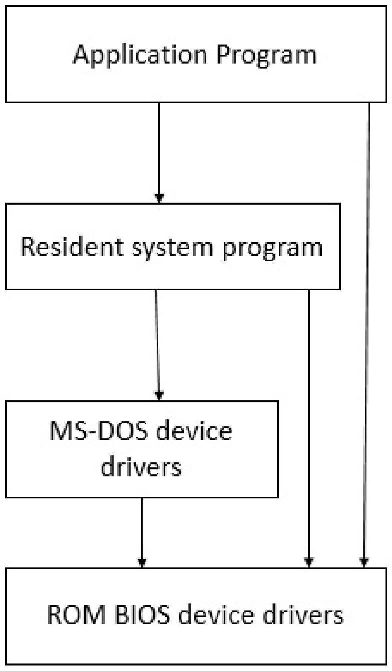

Figure 1.1: Represented that the structure of the Operating System.

Operating System Operation

Storage operations, I/O operations, process performance management, and file management operations are the many categories of operations. All of these actions are carried out by the OS's processes, which are all essential (Figure 1.2).

Memory and Storage Management in Operation System

A group of data in a certain format might be referred to as memory. It processes data and stores instructions. A sizable array or collections of words or bytes, each with a wonderful environment, make up the memory. A computer system's fundamental goal is to run applications. These programs should be performed from the main memory, along with the data they access. Depending upon the value of the program counter, the CPU retrieves instructions from memory.

Memory management is critical for multiprogramming success and in terms of memory use. The efficiency of each algorithm varies depending on many factors, and there are many memory management algorithms reflecting distinct interpretations.

Main Memory

A contemporary computer's primary memory is fundamental to its functionality. The Main Memory is a large collection of characters or bytes, with sizes ranging from tens of thousands to billions. The CPU and I/O devices share a store of quickly readily available information called main memory. When the CPU is successfully using them, the program and data are preserved in the main memory. Because the main memory is integrated into the CPU, transporting data between and within the processor happens very quickly. RAM is a second name for the main memory (Random Access Memory). Volatile memory represents what this memory is. When there is a power disruption, RAM loses its data.

Memory Management

The operating system occupies a limited portion of memory in a multiprogramming computer, whereas other regions are shared by many programs. Memory management is the procedure of distributing memory across so many activities. The system's operating system employs memory management as a methodology to control disc and main memory accesses even though a process is running. The appropriate use of memory is the primary goal underlying memory management.

Requirement of Memory Management

1. Pre- and post-process virtual memory and de-allocation.
2. To monitor how much RAM certain processes are using.
3. To reduce fragmentation-related complications.
4. To make more effective use of main memory.
5. To preserve data integrity even though a procedure is being used.

Storage Management

Storage management is described as the maintenance of the hardware utilized for the storage of user/computer-produced data. To protect your data and storage equipment, an administrator employs a technique or set of procedures. Storage management is a process that then enables users to utilize the most efficient use of storage devices and better protect the integrity of the information for any media on which it resides. The category of storage management traditionally includes a variety of subcategories that includes aspects like security, virtualization, and more, as well as multiple kinds of provisioning or intelligent systems, which together constitute the majority of the storage management system market.

Storage Management Key Attributes

Key characteristics of storage management are often designed to control the system's storage capacity. Here are some of them:

1. Capacity
2. Recoverability

3. Reliability
4. Performance

Feature of Storage Management

Storage management has only a few features that are presented for storage capacity. Listed below are some of them:

1. The strongest quality usage of storage devices is provided via the use of storage management.
2. To effectively benefit the company firm, storage management must also be assigned and managed as a resource.
3. A fundamental system component of information systems is often storage management.
4. It helps them run their data storage resources more efficiently.

Advantages of Storage Management

Storage management has additional benefits, among which are listed below:

1. Managing a storage capacity becomes easy.
2. In most cases, it takes less time.
3. The system's performance is enhanced.
4. It may aid a company in enhancing its agility using virtualization and automation technology.

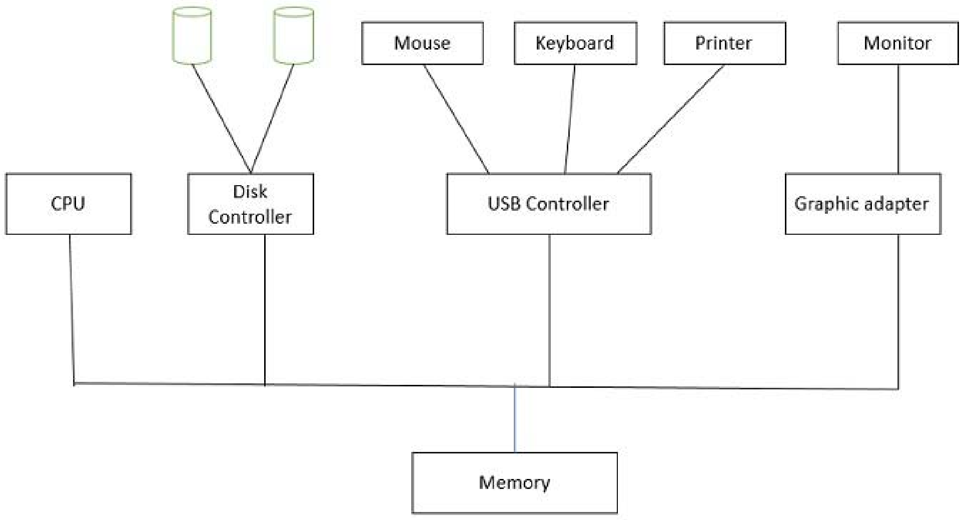

Figure 1.2: Represented that the Operating System Operation.

Open-Source Operating Systems

Open source relates to computer software or programs that let users or intermediaries access, view, and alter the manufacturer's source code thanks to the authors or copyright holders. An open-source operating system's source code is readable and changeable by anyone anywhere. Most operating systems, particularly Microsoft Windows, Apple's Mac OS, and Apple's iOS, are restricted platforms. It is legal to make however many copies of and use open-source software wherever you choose to owe to the way it is licensed. Because it includes any code for licensing, advertising, promoting other goods, identifying, attaching adverts, etc., it often requires less infrastructure than its commercial equivalent. The open-source operating framework enables the use of freely searchable, freely distributable code for advice to customers. An open-source OS's source code is accessible given that it is an open-source application or software. According to their need, the user may alter or edit certain programs and create new apps. Open-source operating systems include, but are not limited to, Linux, Open Solaris, Free RTOS, Open BDS, Free BSD, etc.

The first piece of Open-Source software was published in 1997. Despite the industry, every Software application today does have an Open-Source alternative. Since the start of the twentieth century, several Open-Source Operating Systems have been created attributable to technical advances and breakthroughs.

Working of Open Source Operating System

It behaves similarly to a closed operating system, except the user is allowed to change the program's or application's source code. Even if there is no capacity difference, there may be a distinction to be made in the function. For instance, the knowledge is compressed and kept under such a closed-source operating system. The same problem occurred with open-source software. However, because then you can see the source code, anyone may be able to better appreciate the procedure and alter when data is handled. Both operating systems might well be customized to boost performance, but only the latter required technical expertise. The former is secured and hassle-free. The open-source OS has no set structure or foundation, but it may be customized to meet the requirements of the user.

Best Open Source Operating System

1. Linux Kernel

Linus Torvalds designed the Linux kernel. It includes the fundamental operations fundamental for an operating system, such as data deletion, memory processing, and hardware connections because it is open-source software, many programmers have investigated the source code and constructed a broad range of useful plug-ins and operating systems to meet their specific needs.

2. Linux Lite

Another free and open-source computer system that can function on less expensive machines is Linux Lite. It's a simplified version of windows designed for people who might not be acquainted with Linux-based ones. The operating system is supplied with all the necessary desktops, tools, applications, and capacities. It is based on the Ubuntu operating system and includes a simple UI. The operating system seems to have been reliable and updated periodically during the last five years. Soon after installation, it is successfully operating. Users are not needed to install any further drivers following the first installation. Choose Linux Lite if you want an open-source operating system that isn't excessively heavy for your PC.

3. Linux Mint

Linux Mint is a potent operating system built upon that Linux platform that radiates strength and sophistication. It is an open-source operating system that is user-friendly given that it is straightforward to use and has full multimedia. It is a distribution built on Ubuntu that is appreciated by both newcomers and professional users. It has one of the most efficient software managers and is predicated on the Debian platform. Compared to Ubuntu, it is more dependable and has superior looks.

4. Fedora

Another well-liked Linux-based OS is Fedora, which is acknowledged as the second-best open-source OS after Ubuntu. It is a regular operating system produced by the Fedora Project community and sponsored by Red Hat. Its purpose is to develop and freely distribute cutting-edge open-source technologies. This is why Fedora developers favor upstream advancements over fixes well-being for Fedora. Updates provided by Fedora development companies are accessible to all Linux distributions. It provides a customizable desktop developed on the GNOME platform. A customizable GNOME-based desktop is supplied with Fedora. You may run and personalize a variety of desktop applications using its Fedora Spins function.

5. React OS

Another free and open-source operating system, ReactOS, has been downloaded about 1 million times throughout more than 100 nations. This open-source operating system is a great complement to Windows since it can execute Graphical interfaces. ReactOS is still developing, although consumers who want operating systems that offer numerous customization options may choose it. The computer system, however, is geared toward professionals.

6. Solus

For your desktop PC, Solus is a free and open-source operating system. It is a brand-new Linux operating system that was developed in 2012. Currently, the technology is being used by more than 6000 registered users. Solus comes pre-installed with VLC, XChat, Transmission, Thunderbird, Open Shot Video Editor, Firefox, Budgie desktop environment, and Libre Office Suite. Solus 3, the previous release, was published in August 2017.

7. Chrome OS

A partially open-source operating system with a multitude of appealing features is called Chrome OS. It belongs to the Chromium and Linux families and has features that include enhanced security, support for certified Android and Chrome applications, the Aura window manager, Google cloud print, an integrated media player, connection to virtual desktops, and cloud-based administration. The operating system's only limitation is that it only works with Nexus technology or devices. This means that Chrome OS on a Chromebook will be appealing to Google fans.

Advantages and Disadvantages of Open-Source Operating Systems

Various advantages and disadvantages of the open-source operating system are as follows:

Advantages

Reliable and Efficient

The most dependable and productive operating systems are open-source ones. These are monitored by thousands of people since their source code is available. As an outcome, the world's top coders endeavor to correct any faults or mistakes that may exist.

Cost-efficient

Most open-source operating environments may be downloaded for nothing. Additionally, some products cost far less than commonly available goods.

Flexibility

The fact that you could modify it to accommodate your requirements is a huge benefit. Additionally, there is room for creativity.

Disadvantages

Complicated

In comparison to the ones that seem to be closed, it is less user-friendly. You need to have a rudimentary grasp of technology to fully utilize this programmer.

Security risk

Even though the weaknesses have been found, cyber-attacks are still a possibility since the adversaries have the source code in their possession.

No support

There is no customer assistance accessible to support you if you encounter a problem.

CHAPTER 2

OPERATING-SYSTEM SERVICES

Dr. Rupesh Kumar Sinha, Assistant Professor,
Department of Business Analytics,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - rupesh.sinha@cms.ac.in

The interface between the user, as well as computer hardware, is thought up of software. It is software that can be utilized to execute a wide range of applications. It is the only implementation that is always running. An operating system must be installed for every computer to properly run other applications.

The OS manages how different individuals operate the hardware and application software. It offers workplace conditions for other application programs. An operating system is a collection of specialized applications that execute on a computer system to enable instruments. It manages files, algorithms, input-output devices, and many other things.

Services of Operating System

1. Program Execution
2. Input Output Operations
3. File Management
4. Error Handling
5. Resource Management
6. Communication between Processes

Program Execution

The Operating System is in charge of overseeing a program's development. The program is run after being downloaded into memory. The CPU Scheduling Algorithms establish the order in which they are carried out. A few include SJF, FCFS, etc. The operating system also maintains deadlock, which occurs when no two mechanisms are available for performance at the same moment, while the program is executing. The efficient implementation of user and system programs is the obligation of the operating system. The Operating System employs a wide range of tools to make sure that all operations perform smoothly.

Input Output Operations

The operating system regulates input-output processes and creates a connection between the user interface components and device drivers. Device drivers are programs that are part of the hardware that the operating system regulates and are necessary for experimentally determined synchronization. Additionally, it enables program access to input-output devices as appropriate.

File Management

The operating system also aids in file management. The operating system authorizes access to a file when an application requested it. These privileges ranged from read-only to read-write, and some others. Additionally, it offers a platform on which consumers may add and remove files. All choices about just the storage of data or files, including on floppy discs, hard drives, pen drives, etc., are performed by the computer system. How data should be handled and maintained is decided by the computer system.

Error Handling

The operating system also regulates errors that occurred in input-output devices, the CPU, etc. Additionally, it addresses the problems while also making sure they don't occur often. Additionally, it protects the process from reaching a stalemate. Additionally, it monitors for any errors or defects that could arise while performing any job. The well-secured OS sometimes serves as a defensive measure for preventing and perhaps mitigating any kind of intrusion into the Software System from any external source.

Resource Management

Several processes share available resources. The operating system is in charge of coordinating resource sharing. Using CPU Scheduling Algorithms also controls however much CPU time is given to each application. Additionally, it aids the service's memory management. Secondly, it manages input-output devices. The OS also insures optimal use of all components by allocating which resources to which user.

Communication between Processes

The operating system determines how processes converse with one another. Transferring data among processes is a component of their communication. When processes are interconnected through a computer server rather than being on the same equipment, the operating system itself is charged with controlling its communications.

Security and Protection

Protection in a computer system is a method that typically refers to any computer system components by processes, applications, or users. All-access to system resources should always be tracked and controlled thanks to the operating system. Secondly, it guarantees that connections or external resources must be controlled against security breaches. By employing usernames and passwords, it offers authorization.

User and Operating-System Interface

An operating system is a piece of software that acts as little more than a conduit between the hardware and the user. Without an operating system, a machine can perform any function. The computer hardware is governed and under management by the operating system.

Operating system advancements have rendered PCs more adaptable, dependable, and simple to administer. For desktop environments, there are Microsoft Windows, macOS, Ubuntu Unity, and GNOME Shell; for mobile platforms, there are many Android, Apple's iOS, BlackBerry OS, Palm OS-WebOS, Windows 10 Mobile, and Firefox OS.

Every interactive application, spanning ATMs, video games, cellphones, self-service checkouts, airline self-ticketing, check-in, and PCs, incorporates contemporaneous operating systems and graphical user interfaces.

Some operating systems have command-line interfaces, while others have graphical user interfaces (GUI).

Graphical User Interface OS

GUI, or graphic user interface, stands for. It is a graphical presentation of communication that is displayed to the user for simple machine interaction. Typically, a GUI's operations are carried out by directly manipulating graphical components like buttons and icons. Rather than using the typical text- or command-based communication, interactions with these pictures may be used to communicate.

There are different visual programming languages, each with special benefits for producing a graphical user interface. Because they can execute GUIs concurrently in a browser and as a desktop program, C # or Java may be considered. Other languages like Python, HTML5/Javascript, and C/C++ are also preferred.

To create a user-friendly GUI, the consumer may interact with the program using a wide range of objects and objects.

1. **Button:** A button that, when pushed, launches a program in graphical form.
2. **Dialogue box:** A sort of window that shows extra information and solicits interaction from the user.
3. **Icon:** A cursor-like graphic that represents a program, a feature, or a file.
4. **Menu:** A list of options or commands made available to the user on the menu bar.
5. **Ribbon:** An alternative to the file menu and toolbar that combines application functions.
6. **Tab:** A window's top clickable region that displays a different page or area.
7. **Toolbar:** A row of buttons that controls the operation of software, often located towards the top of an application window.
8. **Window:** A rectangular area of the computer screen that displays the active software.

Types of GUI-based Operating Systems

Ivan Sutherland of MIT created Sketchpad in 1962, which is said to be the first visual computer-aided design application. It included a lighted pen that allowed users to create as well as alter things in engineering drawings in real time with synchronized graphics. The following are the many kinds of operating systems that depend on GUI:

i. Microsoft Windows

One of the most widely used GUI-based operating systems is Microsoft Windows. Microsoft created it and conducts the marketing. Windows 10 is the most recent generation of Microsoft Windows. It also comprises several older versions, like Windows XP, Windows 8, Windows 7, etc. Additionally, Windows Home and Windows Professional are the most extensively used versions for home PCs.

ii. Linux

A well-liked operating system is Linux. It is separate from any particular business or group. However, it was created in 1991 by programmer Linus Torvalds. This is an open-source operating system. Today, LINUX is used by many businesses, individuals, and even in mobile devices, supercomputers, and the internet.

It already has essentially all of the UNIX OS's functionality. However, it offers a few extra features. It encompasses a variety of user interfaces, including GNOME, KDE, Mate, Cinnamon, and some others. Additionally, it has a variety of systems, including UBUNTU, DEBIAN, SOLAS, LINUX MINT, and some others.

iii. Android

It is one of the most generally used kinds of operating systems in use today. It is used on mobile devices and tablets. The majority of its generations use an open-source operating system.

iv. Apple iOS

It is one of the most generally used OS after Android. It is intended to be used with Apple computers including iPhones, iPad tablets, and many other mobile devices [3].

Working of GUI

The model-view-controller software pattern, which separates internal representations of personal data from how information is presented to the user, is used to create graphical user interface design principles. This pattern establishes a framework where users are shown whether the functions are possible instead of having to enter command codes.

1. Users manipulate visual components that are built to answer the kind of data they represent and support the activities necessary to complete the user's job when they communicate with information.
2. The graphical user interfaces are independent of application functionality, an operating system or applications software's look may be customized.
3. Applications generally leverage pre-existing user interface graphical display components on the operating system and design their own distinctive graphical user experience elements.
4. Standard formats for displaying images and text are also found in a typical graphical user interface, allowing information to be exchanged across programs that implement the same graphical user interface design software.
5. Graphical user interface testing is the meticulous procedure of creating test cases to assess the functional and design aspects of the system.
6. Many different agreements and platforms support the various graphical user interface testing methodologies, whether they are manual or automated.

Advantages of Graphical User Interfaces

The most prevalent characteristics of operating systems incorporating graphical interfaces are as follows:

1. A user interface with graphics has the benefit of significantly increasing usability for something like the typical user.
2. The features of a graphical user interface make use of common iconography and metaphors, also including drag-and-drop for transferring files, to make computer procedures intuitive and simple to use regardless of those who have no previous education or experience with scripting languages or hardware.
3. Applications with graphical user experiences are self-explanatory, feedback is often quick, and signals promote and direct information quality.

System Calls

A user application should interact with the operating system through a system call. Several services are demanded by the software, and the OS reacts by performing several systems calls to accomplish the request. A system call may be programmed in high-level languages like C or Pascal or assembly code. If a high-level language is implemented, the operating system may directly invoke batch processing, which is an established function.

A system call is a way for something like a desktop application to ask the operating system's kernel for a service whilst it is executing. A system call is a way for applications to interface with the operating system. A system call is a letter written to the kernel of an operating system by computer software.

The operating system's functionalities are linked to application components using the Application Program Interface (API). It acts as a bridge between a process and the operating system, enabling application services to ask for operating system services. System calls are the sole method of interaction with the kernel system. Any software that uses resources must utilize system calls.

Making System Cell

A system call is made by computer software when something needs to interact with the kernel of the operating system. To make the operating system's services available to user applications, the system call makes use of an API. There is no other method of getting into the kernel system. System calls provide an interface between the operating system and user applications, hence they must be executed by any programs or processes that need capabilities to run. System calls provide an interface between the operating system and user applications, hence they must be employed by any programs or processes that need capabilities to run.

The differences between a system call and then a user function are shown here.

1. To carry out the possibility of integrating, a system call mechanism may generate and activate kernel processes.
2. A system call is more powerful than a typical subroutine. In the kernel protection domain, a system call with kernel-mode privileges is executed.
3. Shared libraries and some other symbols that are not part of something like the kernel protection domain might not be used by system calls.
4. Global kernel memory seems to be where the system calls data and codes are kept.
5. System cell requirement for operating systems

Needs of System Cell in Operating System

You may need system calls in the operational system in a variety of circumstances. The following are the circumstances:

1. When a file system wishes to create or remove a file, it must be required.
2. System calls are necessary for transmitting and receiving data packets through network connections.
3. System calls are necessary if you wish to read or write a file.
4. A system call is required to access hardware, such as a printer or scanner.
5. New processes are created and managed via system calls.

Working of System Cell

The Applications operate in the operating system, a portion of memory. The operating system's kernel, which runs in kernel space, is reached by a system call. An application must request permission from the kernel before it can generate a system call. This is accomplished by sending an interrupt request, which stops the program counter and gives the kernel control.

The kernel carries out another requested action, such as establishing or removing a file, if the request has been approved. The output of the kernel is sent to the programmer as input. Once the information is received, the programmer resumes the process. The kernel transfers data from kernel space to user space in memory when the operation has finished and provides the results to the application.

A straightforward operating system, such as getting the computer date and time, could only provide a return in a few nanoseconds. It could take a few seconds to complete a somewhat more complex system function, such as connecting to a particular network. To prevent bottlenecks, the preponderance of operating systems starts a unique kernel thread for each system calls. Because modern computer systems are multi-threaded, they can manage several batch processing concurrently.

Types of System Calls

System calls often fall under one of five categories. These are listed below:

1. Process Control

The system call used to guide programs is called process control. Examples of process control include launching, loading, stopping, ending, executing, processing, canceling the process, etc.

2. File Management

A system linear function "file management" is used to manage files. Examples of file management include establishing, deleting, opening, closing, reading, writing, etc.

3. Device Management

A system call called help the process is used to manage devices. Read, device, write, retrieve device performance, release device, etc. are a few applications of device management.

4. Information Maintenance

A system call called "information maintenance" is used to maintain confidentiality current. Acquiring system data, setting time or date, getting time or date, setting database information, etc. are a few instances of information maintenance.

5. Communication

A system call used for communicating is called communication. Examples of communication included messaging and receiving messages, creating and deleting network connections, etc.

System Program

The act of creating systems software using system programming languages is based on system programming. Hardware is the element that is last in the computer hierarchy. Next enters the operating system, followed by system programs and thereafter application programs. System programs make it simple to develop and run programs. Other System Programs are comprehensive, while others are only human tendencies. It often sits between function calls and the user interface. In this case, the user can only monitor System Calls but not System Programs. These characteristics may be used to categorize platform programs:

i. File Management:

A file is a grouping of certain data kept in a computer system's memory. The process of establishing, changing, and disk fragmentation is included in the definition of file management, which is the manipulation of files inside a computer system.

A. Adding new files to the computer system and organizing them in certain places helps.

B. It facilitates fast and attempting to find these files from inside the computer system.

C. It makes illegal downloading between users very simple as well as user-friendly.

D. Keeping files structured in distinct directories is essential.

E. These folders enable users may handle files in conformity with their original purpose or conduct rapid searching on files.

F. It enables individuals to modify the name of files in subdirectories or the data of files.

ii. Status Information:

Some users request parameters such as the date, time, amount of available RAM, or disc space. Others further provide comprehensive and detailed performance, logging, and debugging information. Every bit of this data is prepared before becoming printed or shown on peripheral devices. The program output is also shown via a GUI window, a terminal, or other control circuits or files.

iii. File Modification:

We utilize this to change the contents of files. We deployed several sorts of editors for files maintained on discs or other storage media. We utilize specialized commands to accomplish file manipulations or search their contents.

iv. Programming-Language support:

We utilize compilers, assemblers, debuggers, and interpreters for popular languages of programming that are currently available to consumers. It offers consumers all kinds of guidance. Any programming language could be employed by us. All relevant countries are already supplied.

v. Program Loading and Execution:

After assembly and construction, the program must be loaded into RAM before it can be run. An operating node on the network called a loader is in charge of loading frameworks and applications. It's one of the important steps in beginning a program. The service includes loaders, relocatable loaders, linkage editing, and overlay loaders.

vi. Communications:

Programs create virtual links throughout processes, people, and computer systems. Consumers have the opportunity to transfer files from one user to another, transmit a signal to other users on their screens, search the web, log in remotely, and send messages.

Some examples of system programs in O.S. are:

1. Windows 10
2. Mac OS X
3. Ubuntu
4. Linux
5. Unix
6. Android
7. Anti-virus
8. Disk formatting
9. Computer language translators

Purpose of using the System Program

The actions and functionalities of a system's hardware and software are coordinated and exchanged by the system program, which also supervises the hardware's operations. One illustration of system software is an operating system. The operating system supervises the hardware of the computer and provides an interface for business applications.

Types of System Programs

The types of system programs are as follows:

1. Utility program

It oversees, administers, and manages a broad range of computer resources. Utility applications generally are more technical and are targeted at individuals with deep technological backgrounds. Software for backup, disc tools, and antivirus are only a few examples of utilities.

2. Device drivers

It manages the specific tool that is linked to a software system. In essence, device drivers operate as a translator between the operating system and wearable technology. Drivers for storage devices, scanners, printers, etc. are a few examples.

3. Directory reporting tools

An operating system requires to contain these tools in order therefore for the computer system's management to be made easier.

CHAPTER 3

OPERATING-SYSTEM DESIGN AND IMPLEMENTATION

Dr. Moovendhan.V, Associate Professor,
Department of Business Analytics,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.moovendhan_v@cms.ac.in

An extensive and intricate discipline, the design of an operating system touches upon several facets of computer science. This essay must first discuss operating system design broadly before concentrating on implementing it.

Design Goals

The operating system's aims are design goals. They could be utilized to assess the design and must be completed to satisfy design criteria. Although they may not necessarily be professional, these objectives often have a great influence on how users see their interaction with an operating system. While identifying and prioritizing each project objective, designers must also make sure that these objectives are consistent with customers' expectations or professional guidance.

Additionally, designers must document any potential conflicts between their designs and other motherboard chipsets and then rank those potential conflicts according to cost-benefit analyses. This methodology makes it easier to decide which functionalities should be included in the manufactured product and which ones will need a lot of effort in the future. It's also crucial to keep in mind that CBA encompasses something beyond simply monetary costs; it may also be taken into consideration things like user experience, time to market, and the implications on other systems.

The term "goal-driven design" is often used to refer to the process of determining design specifications, conflicts, and priorities. This method's objective is to ensure that every design choice is made with users and other parties involved in mind.

Mechanisms and Policies

A computer's resources are managed by a collection of software components known as an operating system, which also mainly provides system administration. An operating system's two foundational pillars are mechanisms and guidelines. Policies address higher-level functions like resource development, security, and dependability whilst also mechanisms handle regular ones like scheduling, memory management, and interrupt handling. For an OS to be competent at its purpose, it must supply both strategies and guidelines for each component:

Applications should be allowed access to the proper computer resources via mechanisms. Likewise, they must ensure that applications don't conflict with one another during using these capabilities. What processor affinity might happen during multitasking operations? Policies dictate how applications will interact with each other while they are operating independently on several CPUs inside a regular computer.

These are just a few of the plethora of queries that policies must address. These controls and requirements must be followed, and the OS is in charge of managing any violations that may arise. Additionally, the operating system provides a variety of services to programs, including networking and file access.

Additionally, the operating system has the task of ensuring that each of these duties is completed promptly and efficiently. Applications are given access to the underlying computer resources, and the OS makes sure they are used effectively by the applications. Additionally, it manages any objections that arise during implementation to prevent catastrophic failure.

Implementation:

Writing source code in some kind of a high-level programming language, compiling something into object code, and then understanding (running) this object code using interpreters are all steps in the system implementation. An operating system's mission is to provide services to individuals when they utilize their workstations to execute programs. An operating system's main job is to manage how the program is run. Additionally, it offers services like memory management, interrupts handling, and root file system access features so that users or other connected devices may make substantial use of the applications. A program or software application called a version of windows manages the hardware and resources of a computer. It serves as a bridge between programs, users, and indeed the hardware of the computers. Without any human input, it controls the behavior of any software installed on a computer.

The operating system oversees a variety of tasks, including memory management, data security enforcement, and peripheral device management. Secondly, it offers a user interface because then people may converse with their computers. To be accessible when the computer turns on, the operating system is often stored in ROM or flash memory. Multiple servers were intended to be controlled by the first operating systems. They required many people to create and were extraordinarily massive and expensive, with millions of lines of code.

Operating systems of yesteryear are far more user-friendly and compact. They have been developed to be modular so that users or engineers may completely change them.

There are many different types of operating systems:

1. **Graphical user interfaces** (GUIs) like Microsoft Windows and Mac OS.
2. **Command line interfaces** like Linux or UNIX
3. **Real-time operating systems** that control industrial and scientific equipment
4. **Embedded operating systems** are designed to run on a single computer system without needing an external display or keyboard.

An operating system is a piece of software that governs how computer applications are run as well as offers user services. It is in charge of overseeing the hardware resources available to the computer and offering standard services to all processes operating on the system. Additionally, an operating system makes it easier for individuals to connect with systems.

An operating system oversees resources including memory, input/output devices, file systems, and many other parts of a computer system's physical design in addition to any of these fundamental tasks (hardware). This obligation rests with the different applications themselves or respective developers through to the APIs made available by each program's interfaces with its environment. It does not manage mobile applications or their data.

The operating system comprises the most crucial part of a computer since it enables users to interface with every other part. The operating system gives users access to computing resources like printers and storage devices in addition to ensuring that all running applications are integrated and functioning correctly. An operating system's design and installation are the results of a difficult process that requires several disciplines. The objective is to provide users with a dependable, effective, and easy processing environment to increase the efficiency of their job.

Real-Time Embedded Systems

A system called a "real-time system" is utilized to carry out certain specific behavior. It is a computing system that is employed for a variety of real-time, hard, and soft activities. These particular tasks have a time component. Real-time systems have been given tasks that must be performed in a certain amount of time. Embedded Systems are wide area networks made up of a certain function's software and components from a computer. It may be described as a special computer system created with a particular purpose in mind. However, they constitute embedded systems rather than computational methods, which could operate on their own or be interconnected to bigger systems to achieve a limited number of specified tasks. These embedded systems are capable of working with little to no human participation. Real-time Embedded Systems or Embedded Real-time Systems are the terms used to describe embedded systems that were already intended to carry perform instructional methods.

Real-time systems are technologies that operate under rigorous time limitations and produce a worst-case time estimate within urgent circumstances. A specialized function is performed by embedded systems inside a much wide variety of sizes. Whenever a real-time system has an embedded feature, it is referred to as a real-time embedded system.

Types of Embedded Real-time Systems:

There are two types of embedded real-time systems:

Hard Embedded Real-time System:

Hard real-time activities have been carried out by these embedded real-time systems. These systems have some very intricate designs. These algorithms are precise.

Soft Embedded Real-time System:

Soft real-time responsibilities are carried out by these embedded real-time systems. Due to their framework comprises, these systems have quite a potential of someone being inaccurate.

Structure of Embedded Real-time System

Different system components of an embedded real-time system are by their very existence widely dispersed. Real-time embedded Linux, whether hard or soft, shares this very same structure. A real-time system's architecture comprises a variety of both hardware and software components so that certain operations may be executed within the allotted time frames. The embedded real-time system's composition is shown in the following Figure 3.1:

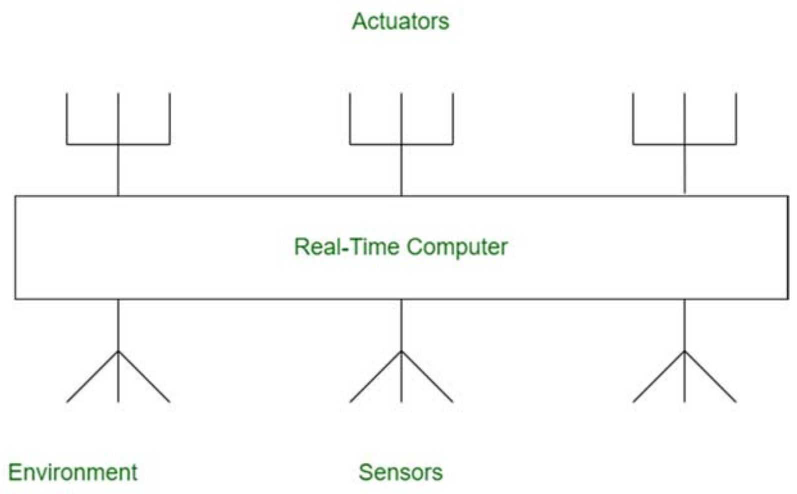

Figure 3.1: Represented the Structure of the Embedded Real-Time System.

1. **Actuator:**

The device that is opposing a sensor is an actuator. While the sensor does the opposite, the actuator transfers electrochemical stimulation into physical signals. Depending on the specifications of the user, it may transform electrical impulses into physiological factors or characteristics. It receives input from the network and outputs results to the immediate neighborhood. The actuator's output would take the shape of any aerobic exercise. Motors and heaters are a couple of something like the widely utilized actuators.

2. **Sensor:**

The opposite of an accelerator is a sensor. On occasion, a sensor is utilized to perceive the atmosphere. It is used to transform quantum systems into electrical signals. This piece of hardware provides the computer with the output after receiving the input from the environment. To identify the appropriate remedial steps, the acquired atmospheric data is analyzed.

Applications of Real-Time Embedded Systems

There are various applications of real-time embedded systems. Some of these are −

A. Vehicle control systems for cars, trains, aircraft, and other vehicles.
B. Satellite communications, radio, and telephones.
C. Medical devices for cardiac treatments, radiation therapy, patient monitoring, etc.
D. Military actions including missile launches, military command posts, etc.
E. Robotic and artificially intelligent systems.
F. Multimedia systems with text, audio, video, and graphic user interfaces.
G. Building management systems that regulate the temperature, doors, elevators, etc.
H. Space operations, such as spacecraft launch and observation, control of space stations, etc.

Real-Time Embedded Systems

A system called a "real-time system" is utilized to carry out certain specific behavior. It is a computing system that is employed for a variety of real-time, hard, and soft activities. These particular tasks have a time component. Real-time systems have been given tasks that must be performed in a certain amount of time. Embedded Systems are wide area networks made up of a certain function's software and components from a computer. It may be described as a special computer system created with a particular purpose in mind. However, they constitute embedded systems rather than computational methods, which could operate on their own or be interconnected to bigger systems to achieve a limited number of specified tasks. These embedded systems are capable of working with little to no human participation. Real-time Embedded Systems or Embedded Real-time Systems are the terms used to describe embedded systems that were already intended to carry perform instructional methods.

Real-time systems are technologies that operate under rigorous time limitations and produce a worst-case time estimate within urgent circumstances. A specialized function is performed by embedded systems inside a much wide variety of sizes. Whenever a real-time system has an embedded feature, it is referred to as a real-time embedded system.

CHAPTER 4

OPERATING SYSTEMS' HISTORY

Dr. Lakshmi Sevukamoorthy, Assistant Professor,
Department of Business Analytics,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.lakshmi@cms.ac.in

One or more CPUs, some main memory, discs, printers, a keyboard, a mouse, a display, network interfaces, and several other input/output devices make up a contemporary computer. All in all, it's a complicated system, and no code would ever be developed if each application programmer had to fully comprehend how everything works. Additionally, it is a very difficult task to manage all of these components and use them properly. The operating system is a layer of software that is built into computers as a result. Its role is to provide user programmes a better, simpler, cleaner model of the machine and to manage managing all the resources listed above. In this book, operating systems are the focus.

While looks might be misleading, the majority of readers will have some familiarity with an operating system like Windows, Linux, FreeBSD, or OS X. While it requires the operating system to do its tasks, the programme that users interact with is not a part of it. It is often referred to as the shell when it is text-based and as the GUI (Graphical User Interface), which is pronounced "gooey," when it employs icons. Chips, boards, discs, a keyboard, a monitor, and other tangible items make up the hardware. The software is placed on top of the hardware. The two operating modes on the majority of computers are kernel mode and user mode. The most essential component of software, the operating system, operates in kernel mode (also called supervisor mode). It can carry out any command that the machine is capable of carrying out in this mode since it has full access to all hardware. The remaining programme operates in user mode, where a limited number of machine instructions are accessible. User-mode applications cannot execute any instructions, in particular ones that modify the machine's control or perform input and output (I/O). In this book, we shall revisit the distinction between kernel mode and user mode many times. It is essential to the operation of operating systems.

The user may launch additional programmes, such as a Web browser, email reader, or music player, from the user interface programme, often known as the shell or GUI, which is the lowest level of user-mode software. These apps also use the operating system quite a bit. If a user does not like a particular email reader, he is free to get a different one or write his own if he so chooses; however, he is not free to write his own clock interrupt handler, which is a component of the operating system and is safeguarded by hardware against attempts by users to modify it. This is an important distinction between the operating system and normal (user- mode) software. Nevertheless, in embedded systems (which may not have kernel mode) or interpreted systems, this line can occasionally be blurred (such as Java-based systems that use interpretation, not hardware, to separate the components). Other than stating that it is the software that runs in kernel mode, it is difficult to define what an operating system is, and even that isn't necessarily accurate. The fact that operating systems manage these hardware resources and provide application programmers (and application programmes, naturally) with a clean abstract set of resources rather than the clumsy hardware ones is a contributing factor in the issue. You could hear mostly about one function or the other, depending on who is speaking.

An Expanded Machine: The Operating System

At the machine-language level, the architecture (instruction set, memory organisation, I/O, and bus structure) of the majority of computers is rudimentary and difficult to programme, particularly for input/output. Consider the contemporary SATA (Serial ATA) hard drives found in the majority of PCs to further illustrate this idea. A book (Anderson, 2007) with almost 450 pages that described an early interface to the disc and what a programmer would need to know to utilise the disc. The interface has undergone several revisions since then and is much more complex than it was in 2007. No rational programmer would logically wish to work with this disc at the hardware level that is obvious. Instead, a programme known as a disc driver takes care of the hardware and offers a user interface for reading and writing disc blocks without diving into the specifics. Several drivers are included in operating systems to manage I/O devices.

Even still, this level is much too low for the majority of applications. Because of this, all operating systems provide files as a further layer of abstraction for utilising drives. Programs may create, write, and read files using this abstraction without having to worry about the complex intricacies of how the hardware really operates.

The secret to controlling all of this complexity is one abstraction. An almost difficult job is broken down into two doable ones by using effective abstractions. The first step is creating and using abstractions. The second is tackling the issue at hand utilising these abstractions. The file, as previously indicated, is one abstraction that practically every computer user is familiar with. Such as a digital picture, email message that has been saved, music, or Web page, it is a piece of information that is helpful. Photos, emails, music, and Web sites are far simpler to manage than the specifics of SATA (or other) drives. The operating system's role is to provide high-quality abstractions, which are subsequently used to implement and manage the resulting abstract objects. Abstractions will be a major topic in this book. One of the secrets to comprehending operating systems is their use.

It is worthwhile to reiterate this argument in various phrases since it is so crucial. Hardware is unsightly, with all due respect to the industrial experts who so painstakingly developed the Macintosh. The individuals who have to build software to leverage real processors, memory, discs, and other devices encounter challenging, inconvenient, peculiar, and inconsistent user interfaces. This may sometimes be attributed to the need for hardware backward compatibility. Sometimes it's an effort to cut costs. Nevertheless, often the hardware designers are unaware of (or don't care about) the problems they are creating for the software. Hiding the hardware and providing programmes (and their programmers) with pleasant, clean, elegant, consistent abstractions to deal with instead is one of the main goals of the operating system. It should be remembered that application programmes serve as the operating system's true clients (via the application programmers, of course). They are the ones who interact directly with the abstractions of the operating system. End users, on the other hand, work with the abstractions that the user interface, whether a command-line shell or a graphical interface, provides. Although there are times when the operating system's abstractions and those at the user interface are comparable, this is not always the case. Consider the standard Windows desktop and the line-oriented command prompt to further understand this idea. Both are Windows operating system applications that make use of Windows' abstractions, but they have quite different user interfaces. Similar to this, a Linux user working directly on top of the underlying X Window System sees a vastly different interface than a Linux user working with Gnome or KDE, although the underlying operating system abstractions are the same in both situations.

Operating System Manages Resources

A top-down perspective is the idea that an operating system serves largely as abstractions for application applications. An alternate, bottom-up perspective asserts that the operating system's purpose is to control every component of a complicated system. Modern computers are made up of a number of components, including processors, memory, timers, drives, mouse, network interfaces, printers, and many more. According to the bottom-up approach, the operating system's responsibility is to provide a systematic and controlled distribution of the processors, memory, and I/O devices among the numerous applications that use them.

Many applications may run simultaneously in memory thanks to modern operating systems. Consider what would happen if three computer programmes attempted to print their results to the same printer at the same time. The printout's first few lines may come from programme 1, then its next few lines from programme 2, then its following few lines from programme 3, and so on. Absolute pandemonium would be the outcome. By buffering all output going for the printer on the disc, the operating system can control the potential mayhem. After one programme is completed, the operating system may transfer the output from the disc file where it was saved for the printer, while the other programme can continue to generate additional output while being unaware that the output is not really going to the printer (yet).

The requirement for controlling and safeguarding the memory, I/O devices, and other resources is increased when a computer (or network) has several users since they may otherwise interfere with one another. Moreover, users often need to exchange information (files, databases, etc.) in addition to hardware. Simply put, according to this theory of the operating system, its main duties include keeping track of which applications are utilising which resources, approving resource requests, recording consumption, and mediating disputes between programmes and users. Multiplexing (sharing) resources in two distinct ways, including time and space, is a component of resource management. When a resource is time multiplexed, several users or programmes utilise it alternately. The resource is first used by one of them, then by another, and so on. For instance, when there is only one CPU available and several programmes compete for its use, the operating system first assigns the CPU to the first programme. Once it has had a chance to run for a while, however, another programme is given the opportunity to use it, and eventually the first programme is given the CPU once more. The operating system is responsible for determining how the resource is time multiplexed, including who goes next and for how long. The sharing of a printer is another example of time multiplexing. A choice must be made about which print job will be produced next when many are in the queue to print on a single printer. Space multiplexing is the other kind of multiplexing. Each consumer receives a portion of the resource rather than having to wait their turn. To allow many running applications to share main memory, for instance, each one may be resident at once (for example, in order to take turns using the CPU). If there is adequate memory to accommodate numerous applications, it is more effective to run several of them simultaneously than than giving one of them, access to the whole memory, particularly if it only requires a tiny portion of the total. Naturally, this causes concerns with justice, protection, and other issues, and the operating system must address them. The disc is another resource that uses space multiplexing.

A single disc may store data from several users at once in numerous systems. A common operating system duty is allocating disc space and keeping track of who is utilising which disc blocks.

Operating Systems' History

Throughout the years, operating systems have changed. We'll take a quick look at a few of the highlights in the sections that follow. As the architecture of the computers on which they operate has traditionally been strongly correlated with operating systems, we will examine a series of computer generations to see what their operating systems were like. While the comparison between operating system and computer generations is clumsy, it does provide structure where there otherwise wouldn't be any.

The following evolution is mostly chronological, although it hasn't always been smooth sailing. Each new development began without waiting for the one before it to be completely accomplished. Along with a lot of overlap, there were also a number of false beginnings and dead ends. Consider this to be a guide only, not the last word. English mathematician Charles Babbage created the first genuine digital computer (1792–1871). Babbage spent the majority of his life and career trying to construct his "analytical engine," but he was never able to get it to function properly because it was entirely mechanical and the available technology at the time was unable to produce the necessary wheels, gears, and cogs with the level of precision he required. The analytical engine obviously lacked an operating system. As a fascinating historical aside, Babbage employed Ada Lovelace, the daughter of the renowned British poet Lord Byron, as the first programmer when he realised he would require software for his analytical engine.

The First Generation (1945–1955)

Up until World War II, which sparked an explosion of activity, little progress had been achieved in building digital computers after Babbage's disastrous attempts. At Iowa State University, Professor John Atanasoff and his graduate student Clifford Berry created what is today recognised as the world's first operational digital computer. 300 vacuum tubes were employed. Konrad Zuse in Berlin constructed the Z3 computer using electromechanical relays about the same time. The Mark I was created by Howard Aiken at Harvard, the Colossus by a team of scientists at Bletchley Park in England, and the ENIAC by William Mauchley and his doctoral student J. Presper Eckert at the University of Pennsylvania in 1944. While some were programmable, some utilised vacuum tubes, some were binary, and all were very basic and required several seconds to do even the simplest computation.

In those early times, each machine was conceived, constructed, programmed, operated, and maintained by a separate team of people (often engineers). The only way to programme anything was in pure machine language, or much worse, by wiring electrical circuits by attaching many wires to plugboards in order to regulate the computer's fundamental operations. Languages for programming weren't known (even assembly language was unknown). Operating systems didn't exist. The typical procedure was for the programmer to sign up for a block of time on the wall-mounted signup sheet, then go to the machine room, connect their plugboard into the computer, and wait for the next several hours praying that none of the 20,000 or so vacuum tubes would blow during the run. Almost majority of the issues included easy mathematical and numerical computations, such as determining artillery trajectories or calculating sine, cosine, and logarithm tables. The process had been slightly enhanced by the advent of punched cards in the early 1950s. Instead of utilising plugboards, programmes could now be written on cards and read in; otherwise, the process remained the same.

Generation II (1955–1965): Batch systems and transistors

The image was drastically altered by the transistor's invention in the middle of the 1950s. As time went on, computers' dependability increased to the point that they could be produced and sold to paying clients with the knowledge that they would operate for long enough to do certain valuable tasks. The distinction between designers, builders, operators, programmers, and maintenance personnel was now quite evident.

Mainframes, as these computers are now known, were kept locked up in huge, particularly air-conditioned computer rooms, with teams of qualified operators to manage them. Only huge businesses, significant government entities, or institutions could afford the price tag of several million dollars. A programmer would first create the programme on paper (in FORTRAN or assembly language) and then punch it on cards to execute a job (i.e., a programme or collection of programmes). After the output was prepared, he would go drink coffee after bringing the card deck down to the input room and giving it to one of the operators. An operator would walk over to the printer, pull out the output, and transport it over to the output room after the computer completed whatever task it was working on so that the programmer could later retrieve it. He would then read in a deck of cards that had been brought from the input room. The operator would have to read in the FOR- TRAN compiler from a file cabinet if it were required. Operators were wasting a lot of computer time by moving around the machine room.

It is not surprise that individuals sought for methods to cut down on the amount of time spent given the expensive nature of the equipment. The batch system was the solution that was most often used. The concept was to use a tiny (relatively) inexpensive computer, like the IBM 1401, which was fairly excellent at reading cards, duplicating tapes, and printing output but not at all strong at numerical computations, to gather a tray full of tasks in the input room and then read them onto a magnetic tape.

After gathering a batch of tasks for roughly an hour, the cards were read and written onto a magnetic tape. This tape was then taken into the machine room and installed on a tape drive. The first task was then run by the operator after loading a specific software (the forerunner of today's operating system). Instead of printing, the result was written onto a second cassette. The operating system automatically read the next task from the cassette and started executing it once each job concluded. The operator removed the input and output tapes after the whole batch had been processed, swapped the input tape for the subsequent batch, and took the output tape to a 1401 for off-line printing (i.e., not connected to the main computer).

It began with a $JOB card containing the name of the programmer, the account number to be charged, and the maximum run duration in minutes. The operating system was then instructed to load the FORTRAN compiler from the system tape via a $FORTRAN card. The programme to be compiled and a $LOAD card that instructed the operating system to load the recently compiled object programme were directly behind it. (Compiled programmes required to be explicitly loaded and were often written on scratch cassettes.) The $RUN card followed next, instructing the operating system to start the programme with the data after it. The $END card signalled the completion of the task at the end. The ancestors of contemporary shells and command-line interpreters were these archaic control cards. The majority of computations performed on large second-generation computers were in the sciences and engineering, such as the solution of partial differential equations that often occur in physics and engineering. They were mostly written in assembly language and FORTRAN. FMS (the Fortran Monitor System) and IBSYS, IBM's operating system for the 7094, were typical operating systems.

The Third Generation: Multiprogramming and ICs (1965–1980)

Most computer manufacturers had two independent, incompatible product lines by the early 1960s. On the one hand, there were the word-oriented, massively parallel scientific computers, like the 7094, which were used in research and engineering for industrial-strength numerical computations. On the other side, there were the character-oriented, business computers, such the 1401, which were extensively used by banks and insurance firms for tape sorting and printing.

It cost the manufacturers a lot of money to create and maintain two whole distinct product lines. In addition, many first-time computer customers found that they required a little system at first but soon outgrew it and desired a larger machine that could run all of their previous applications quicker. IBM introduced the System/360 in an effort to address both of these issues simultaneously. The 360 was a line of software-compatible computers that included 1401-sized units as well as considerably bigger ones that were more potent than the formidable 7094. Just the pricing and capabilities (i.e., allowed I/O devices, maximum RAM, CPU speed) varied amongst the computers. Theoretically, programmes created for one computer could run on all the others as they shared the same architecture and instruction set. (Yet, Yogi Berra is quoted as saying that although theory and practise are equivalent in principle, they are not in reality.) A single family of computers could meet the demands of all clients since the 360 was designed to handle both scientific (i.e., numerical) and commercial computing. The 370, 4300, 3080, and 3090 are backward-compatible upgrades to the 360 range that IBM released in the next years with more advanced technology. The most current member of this series is the zSeries, despite the fact that it has significantly changed since the beginning. In contrast to second-generation computers, which were constructed from individual transistors, the IBM 360 was the first major computer line to employ (small-scale) ICs (Integrated Circuits), giving it a significant price/performance advantage. It was an instant hit, and all the other big manufacturers quickly embraced the concept of a family of interoperable computers. These machines' offspring are still in use in data centres today. These days, they are often used as servers for World Wide Web sites that must handle thousands of requests per second or for handling enormous databases (such as those used by airline reservation systems). The "single-family" concept's greatest strength was also its biggest vulnerability. Every software, including the OS/360 operating system, was supposed to be compatible with all models in the original design. It had to function both on extremely big systems, which often replaced 7094s for doing heavy computing tasks like weather forecasting, and on tiny systems, which frequently merely replaced 1401s for transferring cards to tape. Both systems with few peripherals and systems with many of peripherals needed to function well with it. It had to function both in professional and academic settings. Above all, it had to be effective for each of these many applications. There was no way IBM, or anybody else, could create a piece of software that satisfied all those opposing demands. The end product was a two to three orders of magnitude bigger than FMS operating system that was also very sophisticated. It was composed of millions of lines of assembly code created by tens of thousands of programmers, and it was riddled with many problems that required a steady stream of updates to try to fix. The number of bugs likely stayed consistent over time since every new version both added and removed problems. Fred Brooks, one of the OS/360 creators, later wrote a sharp and funny book (Brooks, 1995) about his interactions with the operating system. It is hard to characterise the book in this space, but suffice it to say that the cover features a herd of extinct animals trapped in a tar pit. Operating systems are described as "dinosaurs" on the cover of Silberschatz et al. (2012). Despite their vast size and issues, OS/360 and other third-generation operating systems manufactured by other computer manufacturers really provided a fair amount of satisfaction for the majority of its consumers. They also helped to promote a number of crucial methods that were not included in second-generation operating systems. The most significant of them was probably multiprogramming. On the 7094, the CPU just sat idle while the present work was put on hold while it waited for a tape or other I/O operation to finish. I/O is uncommon in highly CPU-bound scientific computations, therefore this lost time is not important. Something had to be done to stop the (expensive) CPU from being idle for such a long period since with commercial data processing, the I/O wait time may often be 80 or 90% of the entire time. Memory was divided into numerous parts, each of which had a separate task as the eventual answer. A task may be utilising the CPU while another was waiting for I/O to finish. The CPU could be kept active almost constantly if enough tasks could be held in main memory at once. The 360 and other third-generation systems were equipped with this technology, making it possible to run numerous processes securely in memory at once. This needs specialised hardware to shield each programme from spying and mischief by the others.

Third-generation operating systems also had the capacity to read tasks from cards onto the disc as soon as they were brought into the computer room, which was a significant feature. The operating system could then load a new task from the disc into the now-empty partition and execute it whenever an existing job terminated. Spooling, which stands for Simultaneous Peripheral Operation on Line, is a technique that was also utilised for output. The 1401s were no longer required with spooling, and carrying tapes significantly decreased. Third-generation operating systems were remained mostly batch systems, despite being well suited for large-scale commercial data processing operations and complex scientific computations. Many programmers yearned for the first-generation period, when they could swiftly debug their programmes since they had the computer to themselves for a few hours. With third-generation systems, it may take several hours between submitting a task and receiving the results, thus one incorrectly placed comma could ruin a compilation and cost the programmer a whole day's work. It did not sit well with programmers. Due to this need for speedy responses, timesharing—a kind of multiprogramming in which each user has an online terminal—was made possible. If 20 people are enrolled in to a timesharing system and 17 of them are chatting, thinking, or sipping coffee, the CPU may be distributed to the three active tasks in turn. Since short commands like "compile a five-page procedure" are more common when debugging software than longer ones like "sort a million-record file," the computer can offer quick, interactive service to many users while also possibly working on large batch tasks in the background when the CPU is otherwise idle. At M.I.T., using a specially modified 7094, the first general-purpose timesharing system, CTSS (Compatible Time Sharing Sys- tem), was created. Unfortunately, timesharing did not fully take off until the third generation, when the required hardware for protection became widely available.

After the CTSS system's success, M.I.T., Bell Laboratories, and General Electric (at the time, a significant computer manufacturer) made the decision to start developing a "computer utility," or a device that would handle a large number of concurrent timesharing users. They used the electrical system as their example; if you need electricity, all you have to do is put a plug into the wall, and, within reason, you'll have all the power you need. The MULTICS (Multiplexed Information and Computing Service) system's creators had an enormous computer in mind that would provide computing capacity for everyone in the Boston region. It was pure science fantasy to think that just 40 years hence, computers 10,000 times faster than their GE-645 mainframe would be marketed in the millions (for far under $1000). Similar to the concept of the current supersonic submarine trains crossing the Atlantic.

Success for MULTICS was uneven. While it had substantially higher I/O capacity, it was built to accommodate hundreds of users on a system that was only marginally more powerful than an Intel 386-based PC. This is not exactly as absurd as it sounds since back then, programmers were able to create compact, effective programs—a talent that has now virtually vanished. There are several reasons why MULTICS did not become the dominant force in the world, not the least of which is that it was created using the PL/I programming language and that the PL/I compiler, which was ultimately released years later, hardly functioned. MULTICS was also, like Charles Babbage's analytical engine in the nineteenth century, a very ambitious project for its time.

To cut a long tale short, MULTICS offered a number of groundbreaking concepts into the field of computer literature, but it was far more difficult than anybody had anticipated to transform it into a significant product and commercial success. Bell Laboratories pulled out of the project, and General Electric completely gave up on the computer industry. But, M.I.T. persevered, and MULTICS ultimately began to function. It was eventually turned into a commercial product and deployed by roughly 80 major corporations and colleges throughout the globe by the firm (Honeywell) that acquired GE's computer business. Users of MULTICS were fervently devoted despite their very tiny numbers. After years of attempting to convince Honeywell to upgrade the hardware, companies including General Motors, Ford, and the U.S. National Security Agency, for instance, finally shut down their MULTICS systems in the late 1990s, 30 years after MULTICS was first introduced. The idea of a computer utility had vanished by the end of the 20th century, but it may resurface in the form of cloud computing, in which comparatively small computers (such as smartphones, tablets, and the like) are connected to servers located in enormous and remote data centres, with the local computer only managing the user interface. The rationale for this is that most individuals would prefer to have a team of professionals, such as those working for the corporation hosting the data centre, handle the administration of an increasingly complicated and fussy computer system. In line with the MULTICS architecture, several businesses are already moving e-commerce in this way by executing emails on multiprocessor servers that simple client PCs connect to.

Generation IV (1980–Present): Individual computers

The era of the personal computer began with the creation of LSI (Large Scale Integration) circuits, processors with thousands of transistors on a square centimetre of silicon. While personal computers, formerly known as microcomputers, did not change much in architecture from minicomputers of the PDP-11 class, they did differ significantly in price. A department in a business or institution might now have its own computer thanks to the minicomputer, but a single person can now have a personal computer thanks to the microprocessor chip. When Intel released the 8080 in 1974, the first all-purpose 8-bit Processor, it sought an operating system for it so that it could test it. Gary Kildall, one of Intel's consultants, was tasked with creating one. Kildall and a buddy created the first microcomputer with a disc when they constructed a controller for the recently announced Shugart Associates 8-inch floppy disc and connected it to the 8080. Then Kildall created CP/M (Control Program for Microcomputers), a disk-based operating system, for it. Kildall's request for the rights to CP/M was approved by Intel since the company did not see much of a future for disk-based microcomputers. Kildall subsequently established a business, Digital Research, to continue to create and market CP/M.

To make CP/M compatible with operating on the many microcomputers utilising the Zilog Z80, 8080, and other CPU processors, Digital Research redesigned it in 1977. For around five years, CP/M absolutely dominated the microcomputer industry thanks to the abundance of application programmes that were created to operate on it. The IBM PC was created by IBM in the early 1980s, and IBM scoured the market for software to run on it. Bill Gates was approached by IBM about licencing his BASIC interpreter. They also questioned him about whether he was familiar with any Computer operating systems. Gates advised IBM to get in touch with Digital Research, the world's leading operating systems provider at the time. Kildall sent a subordinate in place of himself when he declined to meet with IBM, making what is unquestionably the worst business mistake ever made in history. Even worse, his attorney refused to sign the nondisclosure agreement that IBM had prepared for the upcoming PC. As a result, IBM contacted Gates once again to ask whether he could provide them an operating system.

When IBM returned, Gates discovered that Seattle Computer Products, a local computer maker, had a suitable operating system, DOS (Disk Operat- ing System). They quickly agreed when he contacted them and offered to purchase it (supposedly for $75,000). Gates then made an offer to IBM, which IBM accepted: a DOS/BASIC bundle. Tim Paterson, the author of DOS, was recruited by Gates to work for his startup business, Microsoft, to implement the adjustments IBM requested. The updated system was given the new moniker MS-DOS (Microsoft Disk Operating System), and it immediately dominated the market for IBM PCs. As opposed to Kildall's effort to sell CP/M to end customers one at a time, Gates made the (in hindsight, very savvy) choice to sell MS-DOS to computer firms for bundling with their hardware (at least initially). Following all of this, Kildall passed away abruptly and suddenly from circumstances that have not yet been completely explained.

By the time the IBM PC/AT, which replaced the IBM PC and had an Intel 80286 Processor, was released in 1983, MS-DOS had gained widespread adoption and CP/M was nearing the end of its useful life. Later, the 80386 and 80486 were often utilised with MS-DOS. While MS-first DOS's iteration was very archaic, later iterations included more sophisticated capabilities, many of which were derived from UNIX. (During the early years of the corporation, Microsoft even offered a microcomputer version of UNIX called XENIX.) Early microcomputer operating systems like CP/M and MS-DOS all relied on keyboard input from users to execute instructions. It later changed as a result of Doug Engelbart's 1960s study at Stanford Research Institute. The Graphical User Interface, which includes windows, icons, menus, and a mouse, was created by Engelbart. Researchers at Xerox PARC accepted these concepts and used them in the creation of their machinery.

The co-inventor of the Apple computer, Steve Jobs, visited PARC one day, saw a GUI, and immediately saw its potential usefulness, unlike Xerox management, who notoriously failed to do so. This massive strategic error inspired the writing of the book Fumbling the Future (Smith and Alexander, 1988). Jobs started work on an Apple with a GUI after that. The Lisa, a project that sprang from this one, was overpriced and a commercial failure. The Apple MacIntosh, Jobs' second try, was a big success not just because it was far less expensive than the Lisa but also because it was designed for customers who not only had no prior computer experience but also had no intention of learning. Macintoshes are extensively utilised and well-liked in the creative industries of graphic design, professional digital photography, and professional digital video production. Originally created to replace the BSD UNIX kernel, the Mach microkernel from Carnegie Mellon University was adopted by Apple in 1999. So, although having a highly different user interface, Mac OS X is a UNIX-based operating system. The popularity of the Macintosh had a big impact on Microsoft's decision to create a replacement for MS-DOS. It created Windows, a GUI-based system that was first built on top of MS-DOS (i.e., it was more like a shell than a true operating system). From 1985 to 1995, Windows was only a graphical interface on top of MS-DOS for nearly ten years. However beginning in 1995, a standalone version called Windows 95 was made available. This version had numerous operating system capabilities and only used the underlying MS-DOS system to boot and execute legacy MS-DOS applications. A significantly altered version of operating system known as Windows 98 was launched in 1998. Yet, there was still a significant amount of 16-bit Intel assembly code in Windows 95 and Windows 98. Windows NT, another Microsoft operating system, was essentially completely rewritten from scratch but was internally compatible with Windows 95 to a certain extent (the NT stands for New Technology). A complete 32-bit system was used. As David Cutler served as the principal developer of Windows NT and was also one of the inventors of the VAX VMS operating system, NT incorporates several concepts from VMS. In fact, it included so many concepts from VMS that DEC, the owner of VMS, sued Microsoft. A settlement was reached outside of court for a sum of money that would need numerous numbers to represent. As the original edition of NT was a considerably improved system, Microsoft anticipated that it would annihilate MS-DOS and all earlier iterations of Windows, but it fizzled. It didn't really take off until Windows NT 4.0, particularly on business networks. Early in 1999, Windows NT version 5 underwent a name change to Windows 2000. It was created with the intention of replacing both Windows 98 and Windows NT 4.0. It also did not quite work out, so Microsoft released Windows Me, a new version of Windows 98. (Millennium Edition). Windows XP, a somewhat improved version of Windows 2000, was introduced in 2001. With a lifespan of six years, the version essentially replaced all earlier iterations of Windows.

The creation of new variants carried on unabatedly. Microsoft divided the Windows family into a client and a server line with the release of Windows 2000. Although Windows Server 2003 and Windows 2008 were part of the server line, XP and its successors served as the foundation for the client line. Later, a third line for the embedded world was added. Service packs were the vehicle via which each of these Windows iterations split off its variants. Several administrators (and authors of operating systems textbooks) were sufficiently enraged by it. Eventually, in January 2007, Microsoft officially unveiled Windows Vista, the replacement for XP. It comes with several new or upgraded user applications, an updated graphical user interface, and enhanced security. Microsoft had hoped that it would totally replace Windows XP, but it never did. Instead, because of the stringent licence conditions, high system requirements, and support for Digital Rights Management—techniques that made it more difficult for consumers to duplicate copyrighted content—it garnered a lot of criticism and negative attention.

Several others choose not to use Vista at all with the release of Windows 7, a new and far less resource-hungry version of the operating system. While Windows 7 didn't include many new features, it had a manageable size and was fairly stable. Windows 7 gained greater market share in less than three weeks than Vista did in seven. Microsoft released Windows 8, a new operating system designed specifically for touch displays, as its replacement in 2012. The business anticipates that a considerably larger range of devices, including desktops, laptops, notebooks, tablets, phones, and home theatre PCs, would adopt the new design as the standard operating system. Nevertheless, compared to Windows 7, the market penetration is now modest. UNIX is another top competitor in the field of personal computers (and its various derivatives). Although though network and business servers are where UNIX performs best, it is often found on desktops, laptops, tablets, and smartphones as well. Students and a growing number of business users are switching from Windows to Linux on x86-based systems. As a side note, we shall refer to all modern processors built on the family of instruction-set architectures that began with the 8086 in the 1970s as "x86" throughout this book. There are numerous of these processors, produced by companies like AMD and Intel, and they often vary significantly from one another in terms of their internal architecture: processors might be 32 bits or 64 bits, have a few or many cores, deep or shallow pipelines, and so on. Even Nevertheless, they all seem quite similar to a coder and can still execute 8086 code created 35 years ago. When the difference is significant, we shall refer to explicit models instead, designating the 32-bit and 64-bit versions with the abbreviations x86-32 and x86-64. Another well-known UNIX variant that emerged from the Berkeley BSD project is FreeBSD. A customised version of FreeBSD is installed on every new Macintosh machine (OS X). Workstations powered by high-performance RISC CPUs also come standard with UNIX. On mobile devices running iOS 7 or Android, its variants are often utilised.

As the majority of UNIX users, particularly seasoned programmers, prefer a command-based interface over a graphical user interface (GUI), practically all UNIX systems support the X Window System (commonly known as X11), a windowing system developed at M.I.T. Users may create, remove, move, and resize windows with a mouse thanks to this system, which also handles basic window management. For UNIX users who want it, a full GUI, like Gnome or KDE, is often offered to run on top of X11 and provide UNIX a look and feel similar to the Macintosh or Microsoft Windows. The expansion of personal computer networks using distributed operating systems and network operating systems is an intriguing phenomenon that started happening in the middle of the 1980s (Tanenbaum and Van Steen, 2007). Users of a network operating system are aware that there are many computers present, and they may log in to distant computers and transfer data from one computer to another. Each device has its own local user and runs its own local operating system (or users). Operating systems for single processors and networks are not fundamentally different. Although it is evident that they need a network interface controller, some low-level software to operate it, as well as applications to enable remote login and remote file access, these additions do not alter the operating system's fundamental design. Although really consisting of numerous processors, a distributed operating system presents itself to its users as a conventional uniprocessor system. Users shouldn't be aware of where their applications are being executed or where their data are stored; the operating system should take care of everything automatically and effectively. As distributed and centralised systems differ, true distributed operating systems need much more than just adding a little amount of code to a uniprocessor operating system different in a few crucial respects. For instance, distributed systems often let applications to operate concurrently on several processors, necessitating the use of more complicated processor scheduling methods to maximise the degree of parallelism. These (and other) algorithms often have to operate with partial, out-of-date, or even inaccurate data due to network communication delays. With a single-processor system, the operating system has comprehensive knowledge of the system state. This condition is quite different from that.

Mobile computers, Fifth Generation (1990-Present)

People have yearned for a portable communication gadget since since detective Dick Tracy in the 1940s comic strip began conversing with his "two-way radio wrist watch." In 1946, a true mobile phone made its debut and it weighed around 40 kg. As long as you had a vehicle to transport it in, you could take it anywhere you went. The first real portable phone debuted in the 1970s and was positively featherweight at around one kilogramme. It was jokingly referred to as "the brick." Soon, everyone was clamouring for one. Almost 90% of people on the planet have a cell phone nowadays. In the near future, eyeglasses and other wearable devices will be able to make calls in addition to our portable phones and wrist watches. Also, the phone component is no longer all that fascinating. We do not give it a second thought while we get email, browse the web, text pals, play games, and manoeuvre through traffic. Although the concept of integrating computer and telephony in a phone-like device has existed since the 1970s, the first true smartphone did not emerge until Nokia unveiled the N9000 in the middle of the 1990s. This device physically integrated two, mostly independent devices: a phone and a PDA (Personal Digital Assistant). With its GS88 "Penelope" model, Ericsson invented the word smartphone in 1997.

The battle between the different operating systems is severe now that cellphones are commonplace, and the result is even less certain than in the PC industry. At the time of writing, Apple's iOS and Google's Android are the two most popular operating systems, but this was not always the case. In a few years, things might change drastically. In the world of smartphones, it is evident that it is difficult to hold the top spot for very long. After all, the majority of cellphones in the first ten years following their introduction used Symbian OS. Popular companies like Samsung, Sony Ericsson, Motorola, and notably Nokia used it as their preferred operating system. Nevertheless, Symbian's market share began to decline as alternative operating systems, including Apple's iOS and RIM's Blackberry OS, which were first debuted in 2002 for smartphones. Many predicted that RIM would rule the corporate sector while iOS would rule the consumer electronics industry. The market share of Symbian decreased. Nokia abandoned Symbian in 2011 and said that Windows Phone will become its main platform instead. While not quite as popular as Symbian had been, Apple and RIM were once the talk of the town. But, Android, a Linux-based operating system introduced by Google in 2008, quickly surpassed all of its competitors. The fact that Android was open source and offered under a permissive licence was advantageous for phone makers. They could thus easily modify it and fit it to their own hardware. It also has a sizable developer community that mostly writes applications in the well-known Java programming language. The domination may not continue, however, and Android's rivals are desperate to reclaim part of its market share, as history has proven.

CHAPTER 5

HARDWARE OF COMPUTER SYSTEM

Naveen Kumar.V, Assistant Professor,
Department of Business Analytics,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - naveenkumar_v@cms.ac.in

An operating system is closely related to the computer's hardware. It handles resources and expands the computer's instruction set. It has to be very knowledgeable about the hardware, or at least how the hardware appears to the programmer, in order for it to function. Let's take a quick look at the computer hardware that may be found in contemporary personal computers. After that, we may begin discussing the specifics of what operating systems do and how they function. A system bus connects and allows communication between the CPU, memory, and I/O devices. We shall examine the more complex bus architecture of modern personal computers in a later section. This model will be enough for the moment. We will quickly go through each of these parts in the sections that follow, as well as look at a few hardware problems that worry operating system developers. It goes without saying that this will be a very brief overview. On the topic of computer organisation and hardware, several books have been published.

Processors

The CPU is the "brain" of the computer. It retrieves and executes instructions from memory. Every CPU goes through a basic cycle in which it retrieves the initial instruction from memory, determines its type and operands, decodes it, executes it, and then retrieves, decodes, and executes following instructions. Up to the program's conclusion, the cycle is repeated. Programs are carried out in this manner. Each CPU can only carry out a certain set of instructions. As a result, neither an x86 CPU nor an ARM processor can run x86 applications. All CPUs include a few registers within to retain important variables and temporary results since accessing memory to get an instruction or data word takes far longer than actually executing an operation. In order to load a word from memory into a register and save a word from a register into memory, the instruction set typically contains these instructions. Some instructions, such adding two words and putting the outcome in a register or memory, combine two operands from registers, memory, or both into a result. Most computers feature a number of special registers that are accessible to programmers in addition to the basic registers used to store variables and temporary results. The programme counter is one of them and it holds the memory address of the next instruction that has to be fetched. The programme counter is then adjusted to refer to that instruction's successor once that instruction has been fetched. The stack pointer, which corresponds to the top of the active stack in memory, is another register. Each process that has been started but not yet finished has a frame in the stack. Input parameters, local variables, and temporary variables that aren't stored in registers are retained in a procedure's stack frame. The PSW is yet another register (Program Status Word). The CPU priority, the mode (user or kernel), the condition code bits that are set by comparison instructions, and numerous other control bits are all included in this register. The PSW may be read in its entirety by user applications, but they generally cannot write to all of its fields. System calls and I/O heavily rely on the PSW. All registers must be completely known to the operating system. The operating system often stops a current application to (re)start another one during time multitasking the CPU. The operating system must save all the registers each time it terminates a running programme in order to restore them when the application resumes. CPU designers have long abandoned the straightforward concept of retrieving, decoding, and processing one instruction at a time in order to increase speed. Several contemporary CPUs have the ability to execute multiple instructions at once. In most pipeline architectures, even if the previous instruction was a conditional branch that was taken, it must be performed after it has been fetched into the pipeline. Since they reveal the complexity of the underlying machine to them and force them to deal with them, pipelines give compiler authors and operating system developers a lot of trouble.

A number of execution units are featured in this architecture, including ones for integer, floating-point, and boolean operations. A holding buffer is used to store instructions until they may be executed after they have been simultaneously fetched, decoded, and dumped. When an execution unit becomes available, it immediately scans the holding buffer to check whether there is an instruction it can handle before removing it and executing it. Its architecture has the consequence that software instructions are often carried out out of order. Most of the time, it is up to the hardware to ensure that the output matches what a sequential implementation would have delivered, but as we will see, an unpleasant amount of the complexity is imposed on the operating system.

The majority of CPUs, with the exception of the very basic ones used in embedded systems, have two operating modes: kernel mode and user mode. Typically, the mode is controlled by a bit in the PSW. The CPU may utilise all of the hardware features and execute every instruction in its instruction set while it is operating in kernel mode. The operating system often runs in kernel mode on desktop and server computers, allowing it access to all hardware. For the majority of embedded systems, a small portion of the operating system operates in kernel mode while the other portions do so in user mode.

User applications always operate in user mode, which limits the number of features that can be accessed and the number of instructions that may be performed. In user mode, most instructions requiring I/O and memory protection are forbidden. Of course, setting the PSW mode bit to kernel mode is likewise prohibited. A user application must make a system call, which traps into the kernel and activates the operating system, in order to access the operating system's services. The TRAP instruction launches the operating system by transitioning from user mode to kernel mode. Control is transferred back to the user application at the instruction following the system call after the task has been finished.

Procedure call that also has the added feature of toggling between kernel mode and user mode. We shall use the lower-case Helvetica typeface to denote system calls in running text, such as this, as a remark on typography. It is important to remember that traps on computers exist in addition to the instructions for executing a system call. The majority of the additional traps are brought on by the hardware in order to alert the user of a rare event, such an attempt to divide by 0 or a floating-point underflow. In every situation, the operating system takes over and must make decisions. Sometimes it's necessary to end a programme with an error. Sometimes a mistake might be overlooked (an underflowed number can be set to 0). Eventually, control may be returned to the programme so that it can tackle the issue after having indicated in advance that it wishes to handle certain conditions.

Devices with many threads and cores

According to Moore's law, a chip's transistor count doubles every 18 months. This "law" is not a physical principle, such as the conservation of momentum; rather, it is Intel cofounder Gordon Moore's observation of how quickly process engineers at semiconductor firms are able to reduce transistor size. Moore's law has been consistent for more than three decades and is predicted to continue for at least another. Following that, there won't be enough atoms in each transistor to function, and quantum mechanics will start to dominate, prohibiting future reductions in transistor size.

What to do with all of the transistors is a dilemma brought on by their overabundance. One strategy was shown above using superscalar topologies with several functional units. Yet, as the number of transistors rises, more is theoretically feasible. The obvious solution is to increase the cache size on the CPU chip. It is undoubtedly taking place, but ultimately the benefits will stop increasing.

The logical next step is to duplicate part of the control logic in addition to the functional units. A number of other CPU chips, such as the SPARC, the Power5, the Intel Xeon, and the Intel Core series, also contain this feature, which is known as multithreading or hyperthreading (Intel's term for it). It enables the Processor to roughly maintain the state of two separate threads while switching between them in a nanosecond time frame a thread is a kind of lightweight process that, in turn, represents a running programme.) A multithreaded Processor, for instance, may simply switch to another thread if one of the processes has to read a word from memory (which requires several clock cycles). True parallelism is not provided by multithreading. There is just one process operating at once, yet the time it takes to switch threads is on the order of a nanosecond.

Since each thread appears to the operating system as a distinct CPU, multithreading has effects on the operating system. Think of a system with two real CPUs and two threads per CPU. This will appear to the operating system as four CPUs if at some point in time there is just enough work to keep two Processors active,

On sometimes, it could accidentally schedule two threads on one CPU while leaving the other CPU inactive. Using one thread on each CPU instead of this option would be far more efficient. Many CPU chips now contain four, eight, or more complete processors or cores on them in addition to multithreading. The multicore processors in Fig. 1-8 are really four little chips, each with an own CPU. The caches are described in further detail below. There are certain processors with more than 60 cores on a single chip, such as the Intel Xeon Phi and the Tilera TilePro. A multiprocessor operating system will unquestionably be needed in order to operate such a multicore hardware. Interestingly, nothing tops a contemporary GPU in terms of raw numbers (Graph- ics Processing Unit). A GPU is a processor made up of thousands of very small cores. They are excellent for several little concurrent operations like drawing polygons in graphics programmes. They do serial tasks poorly. Also challenging to programme are they. It is unlikely that a large portion of the operating system will run on GPUs, despite the fact that GPUs may be beneficial for operating systems (for example, encryption or processing of network data).

Memory

The memory is a computer's second important component. To ensure that the CPU is not slowed down by the memory, a memory should ideally be very quick (faster than performing an instruction), massively inexpensive, and quick to access. As no existing technology can achieve all of these objectives, a new strategy is used. The memory system is built as a hierarchy of levels.

The highest layers often outperform the lower ones by a factor of a billion or more in terms of speed, capacity, and cost per bit. The CPU's internal registers are located at the upper layer. They are equally as quick as the CPU since they are constructed from the same material. As a result, accessing them is instantaneous. The amount of storage space they have is the cache memory follows, which is mostly managed by the hardware. With addresses 0 to 63 in cache line 0, 64 to 127 in cache line 1, and so on, main memory is split into cache lines, which are generally 64 bytes. A high-speed cache that is either within the CPU or placed relatively nearby stores the most frequently utilised cache lines. The cache hardware determines if the required line is present in the cache when the application wants to read a memory word. If so, this is referred to as a cache hit, and the request is completed without sending a memory request across the bus to the main memory. Typically, cache hits take two clock cycles. Missing a cache requires going to memory, which takes a lot of time. Since cache memory is expensive, its capacity is limited. One, two, or even three layers of cache are available on certain computers, and each one is slower and larger than the one before it. Beyond only caching RAM lines, caching plays a significant role in many fields of computer science. Caching is often used to enhance performance if a resource can be separated into portions, some of which are utilised considerably more extensively than others. It is often used by operating systems. For instance, most operating systems save (parts of) frequently used files in main memory rather than continuously fetching them from the disc. Similar to how lengthy path names may be translated into the disc location where a file is stored, the results of this conversion can also be cached to save needless lookups. Lastly, the outcome of the transformation of a Web page's URL into an IP address may be stored for later use.

Every query is not appropriate for every caching circumstance. On average, a new item will be recorded on every cache miss when storing lines of main memory in the CPU cache. Some of the high-order bits of the memory address being addressed are often used to calculate the cache line to be used. For instance, bits 6 through 17 might be used to describe the cache line and bits 0 to 5 could be used to specify the byte inside the cache line for 4096 cache lines of 64 bytes and 32 bit addresses. The item to be deleted in this instance is the same item into which the new data is entered, but in other systems it may not be. The address in question also serves as a unique identifier for the location in memory to which a cache line should be stored when it is rebuilt to main memory (if it has been updated since it was cached).

Modern CPUs feature two caches because they believe they are a good idea. The decoded instructions are typically sent into the CPU's execution engine from the first level or L1 cache, which is always located within the CPU. The majority of chips contain a second L1 cache for data words that are utilised a lot. The L1 caches generally have a size of 16 KB. Moreover, a second cache known as the L2 cache that stores several gigabytes of recently utilised memory words is often present. The time is what distinguishes the L1 and L2 caches. Although there is no delay when using the L1 cache, there is a one- to two-clock-cycle delay when accessing the L2 cache. Each tactic has benefits and drawbacks. For instance, the AMD method makes maintaining the L2 caches constant more challenging whereas the Intel shared L2 cache necessitates a more complex cache controller.

Often referred to as RAM, main memory (Random Access Memory). Old-timers sometimes refer to it as core memory since primary memory in computers from the 1950s and 1960s was made up of small magnetizable ferrite cores. Despite their long-term disappearance, the name lives on. Memory sizes range from several gigabytes to hundreds of megabytes at the moment and are expanding quickly. Any CPU demands that the cache is unable to handle are sent to main memory.

Many computers also feature a small quantity of non-volatile random-access memory in addition to the primary memory. Nonvolatile memory, in contrast to RAM, retains its data even when the power is turned off. ROM (Read Only Memory) is factory-programmed and cannot be altered afterwards. It is inexpensive and quick. The bootstrap loader that is used to start various computers is stored in ROM. Additionally, some I/O cards come with ROM for managing low-level de- vice control.

Both flash memory and EEPROM (Electrically Erasable PROM) are non-volatile, however unlike ROM, they can be wiped and overwritten. They are utilized in the same manner as ROM, but since writing those takes orders of magnitude longer than creating RAM, it is now possible to fix problems in the programmes they store by rewriting them in the field. Another popular storage option for portable electronic devices is flash memory. For example, it may be used as the film in digital cameras and the disc in portable music players. In terms of speed, flash memory falls between RAM and HDD. Also, unlike disc memory, it deteriorates if it is deleted too often. A different volatile kind of memory is CMOS. Many computers save the time and date in CMOS memory. Even when the computer is disconnected, the time is accurately updated because the CMOS memory and the clock circuit that advances the time in it are powered by a tiny battery. The setup settings, such as the boot disc to use, may also be stored in the CMOS memory. Since CMOS consumes so little power, the first battery installed at the manufacturer often lasts for many years. As a computer starts to malfunction, it might start to act like it has Alzheimer's disease and start to forget things that it has known for years, such which hard drive to boot from.

Disks

Magnetic disc is next in the hierarchy (hard disk). Disk storage is often two orders of magnitudes bigger and costs two orders of magnitude less per bit than RAM. The only issue is that it takes over three orders of magnitude longer to randomly access data on it. One or more metal platters rotating at a speed of 5400, 7200, 10,800 RPM, or higher make up a disc. Similar to the pickup arm of a vintage 33-RPM phonograph used to play vinyl records, a mechanical arm pivots over the platters from the corner. Concentric circles are used to write information onto the disc. Each head may read a track, an annular area, at any given arm position. All of the tracks for a certain arm position come together to create a cylinder.

Every track has a certain amount of sectors, usually 512 bytes every second. On contemporary discs, the outer cylinders are larger than the inner ones in terms of sector density. It takes around 1 msec for the arm to go from one cylinder to the next. Depending on the drive, moving it to a random cylinder normally takes 5 to 10 msec. After the arm is on the proper track, the drive must wait an additional 5 to 10 milliseconds—depending on the drive's RPM—for the required sector to revolve under the head. For slower drives, reading or writing happens at a rate of 50 MB/sec to 160 MB/sec after the sector is beneath the head. Sometimes, you'll hear individuals refer to discs that aren't really discs at all, like SSDs (Solid State Disks). SSDs store data in (Flash) memory and lack moving components and platters in the form of discs. They merely resemble discs in the sense that they can also store a lot of data that is not lost when the power is turned off. Virtual memory is a technology that is supported by many computers. By storing programmes bigger than physical memory on the disc and utilising main memory as a kind of cache for the most frequently used portions, this strategy enables the use of programmes that are larger than available memory. In order to translate the address the programme generated to the actual physical location in RAM where the word is placed, this approach necessitates reallocating memory addresses on the fly. The MMU, a component of the CPU, does this mapping (Memory Management Unit), Performance might be significantly impacted by the MMU's and caching's existence. When moving between programmes in a multiprogramming system—a process known as a context switch—it may be required to flush all modified blocks from the cache and modify the mapping registers in the MMU. Both of these are costly procedures that programmers make every effort to avoid. Later, we'll witness some of their strategy's repercussions.

I/O Equipment

The operating system has other resources to handle in addition to the CPU and memory. I/O devices and the operating system communicate often. I/O devices typically have two components: the device itself and a controller. The device is physically controlled by a chip or group of chips known as the controller. It takes and executes instructions from the operating system, such as reading data from the device. As the actual operation of the device is often intricate and detailed, the controller's purpose is to provide the operating system a simpler—though still quite complex—interface. A disc controller, for instance, could agree to a request to read sector 11,206 from disc 2. A cylinder, sector, and head must then be created by the controller using this linear sector number. The fact that the outer cylinders contain more sectors than the inner ones and that some problematic sectors have been remapped onto other sectors might make this conversion more difficult. The controller must then ascertain which cylinder the disc arm is currently on before instructing it to move the necessary number of cylinders in or out. Before reading and storing the bits that are pulled off the drive, eliminating the preamble, and calculating the checksum, it must wait till the correct sector has rotated beneath the head. It must then put the incoming bits together into words and store them in memory. Controllers often have tiny embedded processors that are preprogrammed to do all of this job. The gadget itself is the second component. Since they are limited in functionality and to ensure uniformity, devices have very minimal user interfaces. The latter is required, for instance, so that any SATA disc controller may manage any SATA disc. Serial ATA is referred to as SATA, whereas AT Attachment is referred to as ATA. If you're wondering what AT stands for, it was IBM's second generation of "Personal Computer Advanced Technology," which was based on the company's 1984 6-MHz 80286 CPU, which was at the time very powerful. We may infer from this that the computer sector has a propensity for constantly adding new prefixes and suffixes to established acronyms. We also learnt that using an adjective like "advanced" carelessly might result in embarrassing situations thirty years from now.

Several PCs today use SATA as the default kind of disc. The operating system only sees the interface to the controller, which may be quite different from the interface to the device since the actual device interface is concealed behind the controller. Since each kind of controller is unique, it requires a particular piece of software to operate it. Device drivers are pieces of software that communicate with controllers, sending instructions and receiving replies. Each controller maker is required to provide a driver for each supported operating system. For example, a scanner may have drivers for OS X, Windows 7, Windows 8, and Linux. The driver must be integrated into the operating system for kernel mode operation in order to be utilised. In fact, drivers may function independently of the kernel, and modern operating systems like Linux and Windows do provide some support for this. Nonetheless, the great majority of drivers operate beyond the kernel border. Just a very small number of systems today, like MINIX 3, run all drivers entirely in user space. It is difficult to enable user space drivers to access the device in a regulated manner. The driver may be inserted into the kernel in one of three methods. The first method is to reboot the machine after relinking the kernel with the new driver. This is how many legacy UNIX systems operate. The second method is to add a line stating that the operating system requires the driver, reboot the computer, and repeat. The operating system searches for and loads the necessary drivers at boot time. This is how Windows works. The third method entails the ability of the operating system to accept new drivers while it is already running and to instantly install them without the need to restart. This method was uncommon in the past but is now much more prevalent. Drivers must always be dynamically loaded for hot-pluggable devices, such as USB and IEEE 1394 devices (described below).

A modest number of registers are utilised by each controller to communicate with it. A basic disc controller, for instance, might comprise registers for the disc address, memory address, sector count, and direction (read or write). The driver receives a command from the operating system, translates it into the proper values to put into the device registers, and then executes them to activate the controller. The I/O port space is made up of a collection of all device registers. On certain systems, the device registers may be read and written to just like regular memory words since they are mapped into the operating system's address space (the addresses it can utilise). No special I/O instructions are required on such systems, and user applications may be kept out of the hardware's grasp by not giving them access to these memory locations (e.g., by using base and limit registers). On other systems, each device register has a port address and is placed in a unique I/O port area. On these devices, the kernel mode has specific IN and OUT instructions that enable drivers to read and write registers. The old technique saves address space but does away with the requirement for specialised I/O instructions. The latter needs specific instructions but utilises no address space. Both platforms have a large user base.

Three methods may be used for input and output. The simplest approach involves sending a system call from a user application to the kernel, which the kernel then converts into a procedure call to the right driver. The driver then initiates I/O and executes a close loop while continually polling the device to determine when it is finished (usually there is some bit that in- dicates that the device is still busy). The driver places any necessary data once the I/O is finished before returning. The operating system then gives the caller back control. This approach, known as busy waiting, has the drawback of keeping the CPU busy until the task is complete by polling the device. The second way involves starting the machine and instructing it to interrupt when it is done. The motorist then makes a turnback. If necessary, the operating system then stops the caller and seeks for alternative tasks to complete. The controller produces an interrupt to notify completion after it has determined that the transfer has ended a three-step I/O procedure in the first phase, the driver instructs the controller by writing to its device registers. The gadget is then turned on by the controller. In step 2, the controller notifies the interrupt controller chip using specific bus lines after it has done reading or writing the number of bytes it was instructed to transmit. In step three, the interrupt controller sets a pin on the CPU chip to inform it whether it is prepared to take the interrupt (which it may not be if it is processing a higher-priority one). the interrupt controller in step 4 places the device's identification number on the bus so that the CPU can see it and determine which device has just ended (many devices may be running at the same time).

The programme counter and PSW are generally then placed onto the current stack and the CPU is switched into kernel mode once the CPU has chosen whether to accept the interrupt. The address of the interrupt handler for this device may be found by using the device number as an index into a section of memory. The interrupt vector is the name of this region of memory. The stacking programme counter and PSW are removed and saved after the interrupt handler (a component of the driver for the interrupting device) has begun. After that, it requests the device to find out its state. After the handler has completed, it goes back to the user programme that was previously executing and executes the first instruction that has not yet been carried out. The third I/O approach takes use of specialised hardware: a DMA (Direct Memory Access) chip that can regulate the movement of bits between memory and a controller without routine CPU interaction. The DMA chip is configured by the CPU, which also gives it instructions on the direction to proceed in, the device and memory locations to use, and the number of bytes to transfer. An interrupt is generated by the DMA chip when it is finished, and it is handled as previously said hardware for DMA and I/O in general. Interrupts may occur at very inopportune times, such as while another interrupt handler is active. Because of this, the CPU contains a feature that allows interruptions to be turned off and then turned on again. Any finished devices continue to assert their interrupt signals when interrupts are off, but the CPU is not interrupted until interrupts are enabled once again. The interrupt controller chooses which device to allow through first if many devices complete while interrupts are disabled, often based on static priorities given to each device. The gadget with the greatest priority wins and is serviced first. Everyone else must wait.

CHAPTER 6

OPERATING SYSTEMS FOR MANY PROCESSORS

Dr. L. Sudershan Reddy, Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - sudershan.reddy@cms.ac.in

For many years, both the original IBM PC and minicomputers used the structure. Yet, as processors and storage became faster, a single bus's capacity to carry all the traffic—certainly the IBM PC bus—was pushed to its limit. A sacrifice has to be made. As a consequence, more buses were introduced, both for CPU-to-memory traffic and for faster I/O devices. The present appearance of a huge x86 system is a result of this progression. This system contains several buses, each having a distinct transfer rate and purpose (such as the cache, RAM, PCIe, PCI, USB, SATA, and DMI). For setup and administration, the operating system has to be aware of every single one of them. The PCIe (Peripheral Component Interconnect Express) bus is the primary bus. Intel developed the PCIe bus to replace the earlier PCI bus, which had been developed to replace the original ISA (Industry Standard Architecture) bus. PCIe is far faster than its forerunners, with the ability to transport tens of gigabits per second. Its nature is also quite unique. The majority of buses were parallel and shared prior to its inauguration in 2004. With a shared bus design, data is transferred between devices using the same cables. As a result, an arbiter is required to decide who may utilise the bus when numerous devices need to deliver data. PCIe, on the other hand, uses exclusive, point-to-point connectivity. The classic PCI parallel bus design calls for sending each word of data across many lines. For instance, a single 32-bit value is delivered across 32 parallel lines on typical PCI buses. PCIe, in contrast, employs a serial bus design and delivers all information in one direction. A message over a single link, referred to as a lane, similar to how a network packet works. Since you do not need to guarantee that all 32 bits arrive at the destination precisely at the same moment, this is significantly easier. Due to the possibility of having numerous lanes running simultaneously, parallelism is still in use. For instance, we could send 32 messages simultaneously using 32 lanes. The PCIe standard is updated every three to five years due to the quick growth in speed of peripheral devices like network cards and graphics adapters. For example, PCIe 2.0's 16 lanes provide 64 gigabits per second. You can double the speed by upgrading to PCIe 3.0, and PCIe 4.0 will further quadruple it. These components are connected to a different hub processor. It is feasible that all PCI devices will link to yet another hub that in turn connects them to the main hub, establishing a tree of buses, in the future when we regard PCI to be not just old but ancient. With this setup, the CPU communicates with the memory through a quick DDR3 connection, with an external graphics card via PCIe, and with every other device via a hub via a DMI (Direct Media Interface) bus. The hub then links all the other components, communicating with hard drives and DVD players through the SATA bus, USB devices via the Universal Serial Bus, and Ethernet frames via PCIe. The older PCI devices that utilise the conventional PCI bus have already been described. Moreover, each core has its own dedicated cache in addition to a much bigger cache that is shared by all of the cores. These caches all introduce new buses. In order to connect all the sluggish I/O devices, including the keyboard and mouse, to the computer, the USB (Universal

Serial Bus) was created. But, for the generation that grew up with 8-Mbps ISA serving as the primary bus in the earliest IBM PCs, labelling a modern USB 3.0 device that hums along at 5 Gbps "slow" may not come easily. Depending on the version, USB employs a tiny connection with four to eleven wires, some of which link to ground or provide power to the USB devices. With a centralised bus like USB, all I/O devices are polled every 1 msec to check whether there is any traffic. A combined load of 12 Mbps could be handled by USB 1.0, 480 Mbps by USB 2.0, and no less than 5 Gbps by USB 3.0. Any USB device may be attached to a computer and begin working immediately without the need for a reboot, which was a need for pre-USB devices, much to the chagrin of a previous generation of disgruntled users. Small Computer System Interface, or SCSI, is a high-performance bus designed for quick discs, scanners, and other devices that need a lot of band- width. These days, we usually see them in servers and workstations. They have a maximum speed of 640 MB/sec. The operating system must be able to setup peripheral devices linked to the computer. This need prompted Intel and Microsoft to create the plug and play PC system, based on a similar idea initially used in the Apple Macintosh. Each I/O card had a defined interrupt request level and fixed ad- dresses for its I/O registers before to plug and play. For instance, the floppy disc controller was interrupt 6 and used I/O numbers 0x3F0 to 0x3F7, the printer was interrupt 7 and used I/O addresses 0x378 to 0x37A, and so on. The keyboard was interrupt 1 and used I/O values 0x60 to 0x64.

Good news thus far, the issue arose when the user purchased a sound card and a modem card, both of which were put to use. They wouldn't cooperate because of their disagreements. Every I/O card was given DIP switches or jumpers as a solution, and users were asked to configure them to choose an interrupt level and I/O device addresses that did not clash with any other I/O devices in the user's system. Sometimes, teenagers who focused their whole life on learning the nuances of Computer hardware might do this without making mistakes. Sadly, no one else could, which caused a mess. Using plug and play, the system automatically records data about the I/O devices, assigns interrupt levels and I/O addresses centrally, and then notifies each card of its respective numbers. Let's take a closer look at that since it is closely related to this work. It's not entirely unimportant.

Start-up of the computer

The boot procedure is summarised as follows. There is a parentboard in every PC (formerly called a motherboard before political correctness hit the computer indus- try). The system BIOS programme is located on the parentboard (Basic Input Out- put System). Low-level I/O software, like as commands to read the keyboard, write to the screen, and perform disc I/O, are all included in the BIOS. Nowadays, it is stored in a flash RAM, a nonvolatile memory that may be updated by the operating system if a BIOS flaw is discovered. The BIOS starts when the computer boots. The amount of RAM that is installed and the functionality of the keyboard and other fundamental components are checked first. To find every device connected to them, it first scans the PCIe and PCI buses. The new devices are setup if the devices present vary from when the system was last started. After that, the BIOS chooses the boot device from a list of devices that are stored in the CMOS memory. By opening a BIOS configuration application shortly after booting, the user may modify this list. Normally, if a CD-ROM or USB device is available, an attempt is made to boot from it. When it fails, the hard drive is used to start the system. The boot device's first sector is read into memory and run. A programme that ordinarily checks the partition table at the end of the boot sector to decide which partition is active is present in this sector. Then data from that partition is read to load a backup boot loader. This loader launches the operating system by reading it from the active partition. The operating system then requests the configuration information from the BIOS. It verifies if the device driver is present for every device. If not, it prompts the user to either download the driver from the Internet or insert a CD-ROM that has it (provided by the device's manufacturer). The operating system loads the device drivers into the kernel once it has all of them. Next it runs a login software or GUI, initialises its tables, and creates any necessary background processes.

The system in use

More than 50 years have passed since the invention of operating systems. Several of them have been created throughout this period, however not all of them are well-known. We will touch on nine of them in this part. Later in the book, we shall revisit some of these various systems. Systems for Mainframes Operating systems for mainframes, those room-sized computers still present in significant corporate data centres, are at the top end. The I/O capability of these computers sets them apart from personal computers. A desktop computer with these characteristics would be the envy of its friends; a mainframe with 1000 drives and millions of terabytes of data is not rare. As high-end Web servers, servers for large electronic commerce sites, and servers for business-to-business operations, mainframes are also making something of a resurgence. The operating systems for mainframes are largely focused on handling several concurrent workloads, the majority of which need enormous quantities of I/O. Batch, transaction processing, and timesharing are the three types of services that are generally provided. A batch system is one that performs regular tasks in the absence of an interacting user. Batch mode is frequently used for tasks like processing claims for an insurance business or reporting sales for a network of retailers. Large numbers of tiny requests are handled by transaction-processing systems, such as processing bank checks or making airline reservations. While each work unit is tiny, the system must be able to process hundreds or thousands of them per second. Several distant users may conduct tasks on the computer simultaneously with the help of timesharing technologies, such as searching a large database. All of these tasks are performed often by mainframe operating systems since they are all closely connected. An ancestor of OS/360 is OS/390, a mainframe operating system. Nonetheless, UNIX derivatives like Linux are progressively displacing main-frame operating systems.

Operating Systems for servers

The server operating systems are one step down. They function on servers, which may be mainframes, workstations, or even extremely large personal computers. They let users to share hardware and software resources while serving several users simultaneously across a network. Print, file, and web services are all available from servers. Websites utilise servers to store the Web pages and respond to incoming requests, and Internet service providers maintain a large number of server computers to support their users. Windows Server 201x, Solaris, FreeBSD, and Linux are common server operating systems.

Operating Systems for Many Processors

Connecting numerous CPUs to a single system is a means to acquire major-league computing capacity that is becoming more and more popular. These systems are referred to as parallel computers, multi-computers, or multiprocessors depending on how precisely they are coupled and what is shared. They need unique operating systems, however these are often modifications of server operating systems with additional communication, connection, and consistency capabilities. Even traditional desktop and notebook operating systems are beginning to cope with multiprocessors, if only on a limited scale, thanks to the recent introduction of multicore CPUs for personal computers, and the number of cores is certain to increase over time. Fortunately, there is a lot of information on multiprocessor operating systems that has been gleaned from years of prior study, so applying this information to multicore systems shouldn't be difficult. Using all this computing power will be difficult for apps. Multiprocessors are used to operate a number of well-known operating systems, including Windows and Linux.

Operating Systems for Personal Computers

The operating system for personal computers is the following categories. All contemporary ones enable multiprogramming, sometimes launching hundreds of applications at boot time. Their responsibility is to provide one user excellent service. For word processing, spreadsheets, gaming, and Internet access, they are frequently utilised. Linux, FreeBSD, Windows 7, Windows 8, and Apple's OS X are typical instances. Operating systems for personal computers are so well-known that minimal introduction is likely necessary. In fact, a lot of individuals aren't even aware that there are other varieties.

Operating Systems for mobile devices

We reach tablets, cellphones, and other portable computers as we continue to descend to smaller and smaller systems. A portable computer, sometimes referred to as a PDA (Personal Digital Assistant), is a little computer that you may use while holding it in your hand. The most well-known examples are smartphones and tablets. As we have previously seen, Google's Android and Apple's iOS presently have the majority of the market share, but they are not alone. The majority of these devices have multicore CPUs, GPS, cameras, and other sensors, as well as enough memory and advanced operating systems. And you can shake a (USB) stick at how many third-party software (or "apps") each of them has.

Operating Systems Embedded

On machines that are not often thought of as computers and that do not accept user-installed software, embedded systems are executed. Microwaves, TVs, automobiles, DVD recorders, conventional phones, and MP3 players are typical examples. The fundamental characteristic that sets embedded systems apart from portable devices is the assurance that no unreliable software will ever be installed on them. All of the software in your microwave oven is in ROM, therefore you cannot download new programmes. Because of this, there is no requirement for shielding between uses, which simplifies the design. In this area, prominent systems include Embedded Linux, QNX, and VxWorks.

Operating Systems for Sensor-Nodes

Many applications involve the deployment of networks of small sensor nodes. These nodes are small computers that use wireless communication to connect to a base station and interact with one another. Sensor networks are used for a variety of purposes, including perimeter protection for structures, border surveillance, fire detection in forests, temperature and precipitation measurement for weather forecasting, gathering intelligence on enemy activities on battlegrounds, and much more. The sensors are tiny computers with radios that run on batteries. Workers must labour for extended amounts of time outside, unaccompanied, and usually in challenging environmental conditions with minimal electricity. Once the batteries start to deplete, there will be an increase in the frequency of individual node failures, hence the network must be resilient enough to withstand them. With a CPU, RAM, ROM, and one or more environmental sensors, each sensor node functions as a true computer. It runs a modest but functional operating system, often an event-driven one that reacts to outside events or takes periodic measurements based on an internal clock. The nodes have limited Memory, and battery life is a significant concern, thus the operating system has to be compact and straightforward. Also, similar to embedded systems, all the applications are loaded beforehand; users do not abruptly launch apps they got from the Internet, simplifying the architecture. An established sensor node operating system is called TinyOS.

Operating Systems in Real-Time

The real-time system is a different kind of operating system. Time is a crucial characteristic in these systems, which is how they are distinguished. For instance, real-time computers must gather information on the manufacturing process in order to run the factory's machinery in industrial process-control systems. There are often strict deadlines that must be fulfilled. For instance, certain tasks must be performed at specific times while a vehicle is travelling down an assembly line. The automobile will be destroyed, for instance, if a welding robot welds either too early or too late. We have a hard real-time system if the activity definitely must take place at a certain time (or within a defined window).

Many of these may be found in aviation, military, industrial process control, and related application fields. These systems must provide unwavering assurances that a certain event will take place by a specific time.

Although sometimes missing a deadline is undesirable, it is acceptable in a soft real-time system and does not have any negative long-term effects. They include digital audio or multimedia systems. Moreover, smartphones are soft real-time systems. As meeting deadlines is critical in (hard) real-time systems, the operating system may sometimes be reduced to little more than a library linked in with the application programmes, with everything being closely connected and without any kind of security eCos is an example of this kind of real-time system. There is significant overlap between the categories of handhelds, embedded systems, and real-time systems. They all have at least a few soft real-time components. Just the software included by the system designers is used by embedded and real-time systems, which makes protection simpler. Users cannot install their own software. Real-time systems are better suited to industrial use, whilst handhelds and embedded systems are designed for consumers. They do, however, have certain similarities.

Operating Systems for Smart Cards

The smallest operating systems are powered by smart cards, which are CPU-equipped devices the size of a credit card. They are severely limited in terms of memory and processing power. Some are powered via the reader's contacts, but contactless smart cards are inductively powered, which severely restricts their functionality. Electronic payments is one example of a single function that some of them can only manage, whilst others can handle several purposes. They are often proprietary systems. Several smart cards are focused on Java. This indicates that a Java Virtual Machine interpreter is stored in the smart card's ROM (JVM). Little applications called Java applets are downloaded to the card and run on the JVM interpreter. Several of these cards have the capacity to manage numerous Java applets concurrently, resulting in multiprogramming and the need for scheduling.

When two or more applets are active at once, resource management and protection also become a problem. The operating system on the card, which is often quite basic, must deal with these problems.

Concepts for Operating Systems

Some fundamental ideas and abstractions, such processes, address spaces, and files, are provided by the majority of operating systems and are essential to comprehending them. As an introduction, we'll take a quick look at a few of these fundamental ideas in the parts that follow. Later in this book, we shall revisit each of them in great depth. We will sometimes use examples, often taken from UNIX, to demonstrate these principles. But, similar instances are often seen in other systems as well, and we shall examine some of them later.

Processes

The process is a fundamental idea in all operating systems. A process is essentially a software that is being run. Each process has an address space, which is a set of memory addresses from 0 to a certain maximum that the process may read from and write to. The executable programme, the program's data, and its stack are all located in the address space. Each process also has a collection of resources attached to it, which often include registers (such as the programme counter and stack pointer), a list of open files, a list of ongoing alarms, a list of connected processes, and all the additional data required to operate the programme. Fundamentally, a process is a container that houses all the data required to perform a programme.

The process notion will be covered in much greater detail later on. For the time being, visualising a multiprogramming system can help you get a good intuitive sense of a procedure. It's possible that the user launched a video editing programme, gave it instructions to convert a one-hour movie to a specific format (which might take hours), and then left to browse the Internet. A background task that regularly checks for new email may have already begun operating in the meanwhile. Hence, the video editor, web browser, and email recipient are all (at least) active processes. Every now and again, the operating system chooses to switch from one process to another, maybe because the previous one used more CPU time recently than was necessary. When a process is momentarily halted in this manner, it must be resumed precisely in the same condition as when it was stopped. This implies that during the suspension, all process-related data must be explicitly preserved. For instance, the procedure can have a number of files open for reading simultaneously. Each of these files has a pointer with the current location attached to it (i.e., the number of the byte or record to be read next). All these pointers must be kept when a process is momentarily halted so that a read call made after the process is restarted will read the correct data. The process table is an operating system table that contains an array of structures, one for each process that is presently running, and stores all the data about each process—aside from the contents of its own address space—in many operating systems.

A (suspended) process thus consists of its address space, often known as the core image (in honour of the magnetic core memory used in earlier times), and its process table entry, which stores the information required to resume the process later as well as the contents of its registers. The calls to the process-management system that deal with starting and stopping processes are the most important ones. Think about a common instance. Commands from a terminal are read by a process known as the command interpreter or shell. The user has just entered a command to request the compilation of a programme. A new process must now be created by the shell to execute the compiler. After the compilation is complete, that process makes a system call to end itself. We swiftly reach the process tree structure if a process has the ability to generate one or more processes (referred to as child processes), and if these child processes may also generate further child processes. When related processes are working together to complete a task, they often need to communicate and coordinate their operations.

Additional process system calls may be used to wait for a child process to end, request extra memory (or release unneeded memory), and overlay the application running in that process with another. On sometimes, it becomes necessary to provide information to a process that is already in motion and not waiting for it. For instance, a process may communicate with another process running on a separate machine by sending messages via a computer network to the distant process. To protect against the possibility that a message or its reply would be lost, the sender may ask for notification from its own operating system after a certain amount of time, allowing it to resend the message if no acknowledgment has been received. The software may go on with other tasks after starting this timer. The operating system alerts the process when the predetermined number of seconds has passed. When a signal is received, the process momentarily suspends its current activity, saves its registers to the stack, and begins to conduct a special signal-handling routine, such as retransmitting a message that has been presumed lost. The operating process is resumed in the same condition it was in shortly before the signal when the signal handler completes. In addition to timers running out, signals—the software equivalent of hardware interrupts—can be caused by a number of other factors. Several traps found by hardware, including utilising an incorrect address or executing an illegal instruction, are also turned into signals for the responsible process. The system administrator issues a UID (User IDentification) to each user who has been granted access to a system. Every process that is launched contains the UID of the initiator. The UID of a child process matches that of its parent. Users may join groups, and each group has a GID (Group Identification). One UID, known as the super user in UNIX or the Administrator in Windows, has exceptional authority and may ignore numerous security restrictions. In big installations, the only person who knows the password to become a super user is the system administrator, but many regular users (particularly students) put a lot of work into finding system faults that let them become super users without the password.

Spaces in Address

Each computer has main memory, which it utilises to store running programmes. Just one programme may be running at once with a very basic operating system. The first application must be deleted in order to launch the second, which must then be loaded into memory. Several programmes may be running simultaneously on more advanced operating systems. Some kind of protection mechanism is required to prevent them from interacting with one another (and with the operating system). While this mechanism must be in the hardware, the operating system is in charge of it. The management and defence of the computer's main memory are the subjects of the aforementioned point of view. Managing the address space of the processes is a distinct but equally significant memory-related challenge. Each process normally has a range of addresses it may utilise, usually starting at 0 and going all the way up to a certain limit. In the simplest scenario, a process's maximal address space is smaller than main memory. In this manner, a process may use its whole address space and main memory will have adequate capacity to accommodate it. In many systems, however, addresses are either 32 or 64 bits long, resulting in address spaces that are 232 or 264 bytes long, respectively. What happens if a process wants to utilise all of its available address space but the computer's main memory is full? Such a technique in the early computers was just chance. Nowadays, a method called "led virtual memory" exists, as was already noted, in which the operating system maintains portions of the address space in main memory and on disc and switches between them as necessary. In essence, an address space is created by the operating system as a collection of addresses that a process may use. The address space is independent of the machine's physical memory and has the flexibility to be bigger or smaller.

Files

The file system is yet another essential idea that is supported by almost all operating systems. The operating system's primary job, as previously said, is to mask the quirks of discs and other I/O devices and provide programmers with a beautiful, tidy abstract model of device-independent files. It goes without saying that system calls are required to add, delete, read, and write files. A file must be found on the disc, opened, and closed before it can be read, thus calls are supplied to carry out these tasks.

Most PC operating systems contain the idea of a directory as a manner of grouping files together in order to give a place to put them. A student may, for instance, have a directory for each course he is enrolled in (for the programmes required for that course), another directory for his email, and still another directory for his personal homepage on the World Wide Web. Then, in order to create and delete folders, system calls are required. Moreover, calls are supplied to transfer a file from a directory and to add an existing file to it.

While both the process and file hierarchies are structured as trees that is where the similarities end. In contrast to file hierarchies, which can have four, five, or even more layers, process hierarchies are seldom particularly deep (more than three levels). Process hierarchies normally last for a few seconds, at most, although directory hierarchies might last for many years. Processes and files have different ownership and protection. Usually, only a parent process is able to supervise or even access a child process, although methods almost always exist to let more than just the owner read files and directories. The path name from the root directory, which is at the top of the directory hierarchy, may be used to specify any file inside it. Slashes are used to delineate the components of such absolute path names, which are a list of the directories that must be passed through before reaching the file from the root directory. The path is absolute, beginning at the root directory, as shown by the leading slash. A file must first be opened, at which point the permissions are verified, before it can be read or written to. The system returns a tiny integer called a file descriptor to be used in further actions if the access is allowed. An error code is issued if access is restricted. The mounted file system is another crucial idea in UNIX. The majority of desktop computers feature one or more optical drives that can accept CD-ROMs, DVDs, and Blu-ray discs. Most computers include USB connections that may be used to connect USB memory sticks, which are really solid state disc drives, and some also have floppy disc drives or external hard drives. UNIX enables the file system on the optical disc to be attached to the main tree in order to offer an elegant method of handling these portable media. The root file system on the hard drive and a second file system on a CD-ROM are independent and unrelated before the mount function. Nevertheless, as there is no method to define path names on the CD-file ROM's system, it cannot be utilised. As it would be the exact kind of device reliance that operating systems should do rid of, UNIX does not permit the prefixing of path names with a drive name or number. The CD-ROM file system may instead be mounted to the root file system wherever the software desires thanks to the mount system function.

Several hard drives may also be mounted into a single tree if a system has them. The special file is another crucial idea in UNIX. To make I/O devices seem to be files, special files are provided. The same system calls that are used to read and write files may be used to read and write these objects. Block special files and character special files are the two categories of special files.

Devices like discs, which are made up of a number of randomly ad- dressable blocks, are modelled using block special files. A software may directly access the fourth block on the device by opening a block special file and reading. Similar to how printers, modems, and other devices that take or output a character stream are modelled, character special files are employed in this instance as well. A pipe may be used to link two processes. It is a kind of pseudofile. Processes A and B must prepare the pipe in advance if they want to communicate via it. Process a writes to the pipe as if it were an output file when it needs to deliver data to Process B. In reality, a pipe's implementation resembles a file's in many ways. By reading from the pipe as if it were an input file, Process B is able to read the data. As a result, file reads and writes across processes under UNIX are remarkably similar to one another. And to make matters worse, a process may only learn that the output file it is writing to is a pipe and not a file by using a specific system call.

Input/output

Physical input and output devices are present on every computer. After all, what use would a computer be if its users couldn't instruct it what to perform or weren't able to access the findings after it completed the task? Keyboards, displays, printers, and other input and output devices come in a variety of shapes and sizes. The operating system is in charge of controlling these devices. In order to manage its I/O devices, every operating system needs an I/O subsystem. Certain I/O software can be applied to many or all I/O devices equally since it is device neutral. Device drivers, among other components of it, are specialised to various I/O devices.

Protection

Users often desire to protect and keep secret the vast quantities of information that computers carry. Emails, company ideas, tax filings, and a lot more might be included in this data. The operating system is responsible for controlling system security so that, for example, only users with permission may access certain files. Consider UNIX as a straightforward illustration to gain a sense of how security may operate. UNIX protects files by giving each one a unique 9-bit binary protection code. Three 3-bit fields make up the protection code: one for the owner, one for other users in the owner's group (the system administrator divides users into groups), and one for everyone else. For read, write, and execute access, there are separate bits for each field. The rwx bits are these three bits. For instance, the protection code rwxr-x—x indicates that the owner, other group members, and everyone else have access to the file only for execution (but not for reading or writing). X denotes a directory's search authorization. The absence of the appropriate per-mission is indicated by a dash. Together with file protection, there are other more security concerns. One of them is defending the system from unauthorised users, both human and nonhuman (such as viruses).

A Shell

The programme that executes system calls is known as the operating system. Although though they are crucial and useful, editors, compilers, assemblers, linkers, utility programmes, and command interpreters are definitely not a component of the operating system. At the risk of somewhat confounding matters, in this part we will take a quick look at the shell, the UNIX command interpreter. It heavily utilises several operating system functions while not being a part of the operating system, making it a useful illustration of how system calls are used. The primary interface is also there.

Unless the user is utilising a graphical user interface, there is no communication between a user sitting at his terminal and the operating system. There are several shells, such as sh, csh, ksh, and bash. They all provide the following functionality, which comes from the original shell (sh). Every time a user signs in, a shell is launched. The terminal serves as both standard input and standard output for the shell. It begins by entering the prompt, which shows the user that the shell is ready to take a command and looks like a dollar symbol. The shell generates a child process and launches the date programme as the child if the user inputs date, for instance. The shell waits for the child process to finish operating while it is still active. The shell retypes the prompt when the kid is finished and attempts to read the next input line.

The majority of modern personal computers use a GUI. In actuality, the GUI is really a programme that runs over the operating system, much like a shell. This reality is made clear on Linux systems since the user has the option of (at least) two GUIs, Gnome and KDE, or none at all (using a terminal window on X11). By altering specific registry variables in Windows, it is also feasible to replace the default GUI desktop (Windows Explorer) with an alternative software, however not many people do this.

Operating Systems' History

The German naturalist Ernst Haeckel said that "ontogeny recapitulates phylogeny" after the publication of Charles Darwin's book On the Origin of Species. By this, he meant that the evolution of a species is repeated (or recapitulated) in the development of an embryo (ontogeny) (phylogeny). In other words, a human egg through phases of becoming a fish, a pig, and so on after fertilisation before developing into a human kid. While this is considered a crude simplification by contemporary biologists, there is some truth to it. The computing sector has experienced something roughly similar. Every new species (mainframe, minicomputer, personal computer, portable, embedded computer, smart card, etc.) seems to undergo the same hardware and software development as its relatives. We sometimes overlook how much the computer industry and many other professions rely on technology. It's not because they like to walk so much that the ancient Romans didn't have vehicles. It's because they had no idea how to construct automobiles. Personal computers exist because it is now feasible to build them affordably rather than because millions of people have a long-standing desire to possess one. It is important to sometimes remember how much technology influences the way we see systems. In instance, it commonly occurs that a concept becomes outmoded due to a shift in technology and disappears overnight. Perhaps another technological advancement could bring it back to life. This is particularly true when the change has to do with how well various system components function in relation to one another. Caches, for instance, were crucial to speed up the "slow" memory when CPUs overtook memories in speed. Caches will disappear if new memory technology ever makes memories quicker than CPUs. Caches will reemerge if new CPU technology makes them quicker than memory once again. Extinction in nature lasts forever, but it sometimes only lasts a few years in computer science. Because of its impermanence, this book may sometimes examine "obsolete" notions, or those that are not compatible with modern technology. Yet, technological advancements may revive some of the supposedly "obsolete principles." Because of this, it's critical to comprehend why a notion has been outmoded and what alterations to the external environment can make it relevant once again. Let's look at a simple example to further illustrate this notion. Early computers featured instruction sets that were hardwired. The instructions could not be modified since they were immediately carried out by the hardware. Eventually came microprogramming, in which an underlying interpreter executed the "hardware instructions" included in software (first made widely available with the IBM 360). Hardwired execution was no longer necessary. It lacked sufficient flexibility. Microprogramming (i.e., interpreted execution) was then rendered obsolete by the development of RISC computers since direct execution was quicker. Interpretation is now making a comeback in the shape of Java applets that are delivered over the Internet and interpreted when they arrive. Since network delays are so significant, they often dominate, execution speed is not always essential. As a result, the pendulum between direct execution and interpretation has already swung multiple times and may do so again in the future.

Enduring memories

Now let's look at some past hardware advancements and how they have consistently impacted software. Early mainframes had a little amount of memory. Just over 128 KB of memory was available on a fully loaded IBM 7090 or 7094, which served as the king of the mountain from late 1959 until 1964. To save scarce memory, it was mostly coded in assembly language, and its operating system was also written in assembly language.

Assembly language was declared to be extinct as compilers for languages like FORTRAN and COBOL improved over time. But, the PDP-1, the first commercial minicomputer, only had 4096 18-bit words of memory when it was originally produced, and assembly language made an unexpected reappearance. Little computers eventually gained more memory, and high-level languages started to be used often on them. The earliest microcomputers had 4-KB memories when they were introduced in the early 1980s, and assembly-language programming came back to life. Assembler was first used to programme embedded computers, which often utilised the same Processor chips as microcomputers (8080s, Z80s, and subsequently 8086s). Nowadays, their descendants—the personal computers—have a lot of memory and are written in high-level languages like C, C++, and Java. However above a certain size, smart cards often feature a Java interpreter and run Java applications interpretively rather than having Java compiled to the smart card's machine language. Smart cards are now experiencing comparable development.

Protection Hardware

The IBM 7090/7094 and other early mainframes did not have protection hardware, therefore they could only run one application at a time. A flawed software might quickly crash the computer and destroy the operating system. With the release of the IBM 360, a basic kind of hardware security was made accessible. Then, these devices could store several programmes in memory at once and execute them sequentially (multiprogramming). Monoprogramming was deemed unnecessary. Multiprogramming was not conceivable, at least not until the first minicomputer appeared, which lacked security hardware. The PDP-11 finally included protection hardware, although the PDP-1 and PDP-8 did not, and this feature paved the way for multiprogramming and, ultimately, UNIX. As the Intel 8080 CPU chip, which was used in the earliest microcomputers, lacked hardware protection, monoprogramming—running one programme at a time in memory—was once again the norm. Protection hardware wasn't installed and multiprogramming wasn't made feasible until the Intel 80286 CPU. Many embedded devices even today just execute a single software and lack any kind of hardware for safety. Let's now examine operating systems. A single manually loaded programme was processed at a time by the rudimentary operating systems that the early mainframes ran since they lacked protective hardware and multiprogramming capability. Eventually, they required full timesharing capabilities in addition to hardware and operating system support for running many applications simultaneously. Although while multiprogramming was already well-established in the mainframe world by the time minicomputers first debuted, they also lacked protective hardware and only ran one manually loaded programme at a time. They gradually gained hardware for protection and the capacity to run two or more applications simultaneously. The original microcomputers could only execute one programme at once, but they subsequently developed the capacity to multitask. Smart cards and handheld computers both followed the same path. Technology always ruled the software development process. For instance, the original microcomputers had no hardware for protection and just 4 KB of memory. Multiprogramming and high-level languages were just too much for such a little system to manage. The hardware and software needed to handle increasingly complex capabilities were both gained when microcomputers turned into contemporary personal computers. This process will most certainly continue for years to come. This cycle of reincarnation may also exist in other sectors, although it tends to spin more quickly in the computer business.

Disks

Most early mainframes relied heavily on magnetic tape. They would assemble, execute, and write the results of a programme that was read in from tape to another cassette. Disks and the idea of a file system didn't exist. When IBM unveiled the RAMAC (RandoM ACcess), the first hard disc, in 1956, this started to alter. It had a floor area of around 4 square metres and had storage capacity for 5 million 7-bit characters, which is sufficient for one medium-resolution digital picture. But putting together enough of them to hold the equivalent of a roll of film quickly became expensive due to the $35,000 yearly leasing rate. Nonetheless, costs subsequently decreased and simple file systems were created. The CDC 6600, debuted in 1964 and for many years the world's fastest computer, is an example of these revolutionary advancements. By assigning files names and trusting that no other users had already chosen names like, say, "data," users may establish so-called "permanent files." This residence only had one storey. A complicated hierarchical file system for mainframes eventually emerged, perhaps culminating in the MULTICS file system. Minicomputers ultimately acquired hard drives as they became more prevalent. When the PDP-11 was first released in 1970, the RK05 disc, which had a 2.5 MB capacity and was just 40 cm in diameter and 5 cm height, was the standard disc. Yet at first, it also had a single-level directory. When microcomputers first appeared, CP/M was the primary operating system. Like DOS, it supported just one directory on the (floppy) drive.

CHAPTER 7

VIRTUAL MEMORY

Dr. Navaneetha Kumar, Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.navaneethakumar@cms.ac.in

Virtual memory allows for faster switching of data between RAM and disc, allowing for the execution of applications with a greater memory footprint than the machine's physical memory. It developed similarly, initially showing up on mainframes before going on to minis and micros. A programme may dynamically link in a library at runtime rather than needing to compile it in thanks to virtual memory. The first system to support this was MULTICS. The concept eventually spread across the system and is currently used by the majority of UNIX and Windows systems. In all of these advances, concepts that were first developed in one setting and then abandoned when the setting changed (assembly-language programming, mono-programming, single-level directories, etc.) sometimes resurface in a different setting a decade or more later. Because to this, there may be concepts and techniques in this book that appear archaic on gigabyte PCs today but may be revived in the future on embedded systems and smart cards.

The System Calls

As we've seen, operating systems serve two primary purposes: managing the computer's resources and giving abstractions to user applications. The interaction between user applications and the operating system mostly affects the former; for instance, generating, writing, reading, and deleting files are examples of the former. The process of managing the sources is automated and mostly visible to the users. As a result, dealing with abstractions is largely what the interface between user applications and the operating system is about. We need to carefully explore this interface in order to fully comprehend what operating systems accomplish. There are several system calls accessible in the interface depending on the operating system (although the underlying concepts tend to be similar). As a result, we are left with two options: (1) nebulous generalizations (e.g., "operating systems have system calls for reading files"), or (2) a more specific system (e.g., "UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read"). The latter strategy is what we've selected. Even though it takes more effort, doing it that way reveals more about what operating systems really accomplish. The majority of other current operating systems contain system calls that accomplish the same duties, even if the specifics vary, even while this talk particularly refers to POSIX (International Standard 9945-1), therefore also referring to UNIX, System V, BSD, Linux, MINIX 3, and so forth. A procedure library is supplied to allow system calls to be made from C programmes and often from other languages as well, despite the fact that the actual mechanics of making a system call are extremely machine dependant and frequently must be described in assembly code.

The following should be kept in mind. Every computer with a single Processor can only execute one instruction at a time. When a process requires a system service, like as reading data from a file while executing in user mode, it must execute a trap instruction to hand over control to the operating system. The operating system then examines the parameters to determine what the calling process wants. After that, it executes the system call before handing over control to the next instruction. Making a system call is analogous to making a specific form of procedure call, with the difference being that procedure calls do not reach the kernel whereas system calls do. Let's take a brief look at the read system call to better understand how system calls work. It contains three arguments, as was already mentioned: the first one specifies the file, the second one points to the buffer, and the third one indicates how many bytes should be read.

The number of bytes actually read in count is returned by the system call and the library method. Normally, this number is equal to nbytes, but it might be less if, for instance, end-of-file is met when reading. The count is set to 1 and the error number is stored in the global variable errno if the system call cannot be executed because of an incorrect argument or a disc error. A system call's results should always be checked by programmes to discover whether an error occurred. System calls are carried out in stages. Let's look at the read call mentioned before to help make this notion more obvious. The calling application puts the arguments onto the stack before invoking the read library method, which initiates the read system call.

For historical purposes, C and C++ compilers place the arguments onto the stack in the opposite order (having to do with making the first parameter to printf, the format string, appear on top of the stack). The second argument is supplied by reference, which means that the address of the buffer (marked by &) rather than its contents is given. The first and third parameters are called by value. The actual call to the library operation follows (step 4). The standard procedure-call instruction used to invoke all procedures is this one. The system-call number is often placed by the library method, which may be written in assembly language, in a location that the operating system anticipates, such as a register (step 5). In order to go from user mode to kernel mode and begin execution at a defined location inside the kernel, it then executes a TRAP instruction (step 6). In fact, the TRAP instruction and the procedure-call instruction are rather similar in that they both accept the instruction after them from a remote place and keep the return address on the stack for later use.

Yet, the TRAP command also varies fundamentally in two ways from the procedure-call instruction. Initially, it enters kernel mode as a side consequence. The directive to call a procedure does not alter the mode. Second, the TRAP instruction cannot jump to an arbitrary location; it must instead specify an absolute or relative address at which the process is located. Depending on the architecture, the instruction either leaps to a single fixed place or provides an equivalent index into a table of jump locations in memory through an 8-bit field in the instruction. The system-call number is examined by the kernel code that launches after the TRAP, and it then directs execution to the appropriate system-call handler, often using a database of pointers to system-call handlers indexed by system-call number (step 7). The system-call handler then begins to operate (step 8). Control may be transferred back to the user-space library function at the instruction after the TRAP instruction once it has finished its task (step 9). Then, in the manner customary for procedures calling return, this procedure returns to the user programme (step 10).

The user application must clear the stack to complete the task, as it does after every method call (step 11). Assuming that the stack descends as it often does when the arguments that were put before the call to read are removed, the produced code precisely increases the stack pointer. Now the programme may proceed in whatever way it pleases. For a valid reason, we said in step 9 above that "may be returned to the user-space library method." The system call could stop the caller from continuing and block it. The caller must be prevented, for instance, if it is attempting to read the keyboard while nothing has been written. The operating system will then search its surroundings to see whether any other processes may be started after this one. The system will be alerted by this procedure, which will then perform steps 9 through 11 when the necessary input becomes available.

The most popular POSIX system calls and, more precisely, the library procedures that implement those system calls, will be discussed in the sections that follow. Over 100 procedure calls are used in POSIX. List of some of the most significant ones, conveniently divided into four groups. We will quickly go through each call's function in the text to see what it performs. Because to the limited resource management on personal computers, the operating system is largely determined by the services provided through these calls (at least compared to big machines with multiple users). The functions offered by the services range from starting and stopping processes to generating, deleting, reading, and writing files, managing directories, and handling input and output. As a side note, it is important to note that POSIX procedure calls and system calls are not 1:1 mapped. A conformant system must include a number of procedures that are specified by the POSIX standard, although it is not specified whether these methods are system calls, library calls, or something else entirely. Performance-wise, a process will often be carried out in user space if it can be done without using a system call (i.e., without trapping to the kernel). The majority of POSIX procedures do, however, call system calls; typically, one procedure maps directly to one system call. One system call may deal with many library calls in certain circumstances, particularly when the needed procedures for multiple situations are simply slight variants of one another.

System Requests Management of Process

With POSIX, forking is the sole technique to start a new process. The whole original process is precisely recreated, down to the registers and file descriptors. The original process and the copy (the parent and kid) split off after the fork. At the moment of the fork, all the variables have the same values, but since the data from the parent are duplicated to construct the child, changes made to one of them in the future have no impact on the other. (Parents and children both use the programme text, which cannot be changed.) The fork call returns a value that is equal to the child's PID (Process IDentifier) in the parent and zero in the child. The two processes may identify which one is the parent process and which one is the child process using the returned PID. After a fork, the child must typically run separate code from the parent. Take the shell as an example. It takes in a command from the terminal, forks off a child process, waits for the child to execute it, and then reads the next command when the child process ends. The parent uses a waitpid system call to wait for the child to complete, which just keeps waiting until the child finishes (any child if more than one exists). By changing the first argument to 1, Waitpid may wait for any old kid or for a particular child. The address indicated by the second argument, statloc, will be set to the child process' exit status after waitpid is finished (normal or abnormal termination and exit value). Also, a variety of alternatives are offered, as indicated by the third parameter. Upon returning, for instance, if no children have previously left. The name of the file to be executed, a pointer to the argument array, and a pointer to the environment array are the three arguments that make up the most basic form of execve. Soon, they will be explained. Execl, execv, execle, and execve are just a few of the library functions that are available to enable the arguments to be supplied differently or omitted altogether. In this book, the system call that is triggered by all of these shall be referred to as "exec." Cp's main programme, as well as the majority of other C programmes' main programmes, include the declaration main (argc, argv, envp), where argc is a count of the number of items on the command line, including the programme name. In the aforementioned example, argc is 3. A pointer to an array, argv, is the second argument. The ith string on the command line is pointed to by element I of that array. In an example, argv [0] would refer to the string "cp," argv[1] to the string "file1," and argv [2] to the string "file2."

Envp, the third argument of main, is a pointer to the environment, which consists of an array of strings with assignments of the form name = value and is used to send data to programmes such the terminal type and the name of the user's home directory. Programs may utilise library procedures to access environment variables, which are often used to tailor how a user wishes to carry out certain operations (e.g., the default print- er to use). Do not give up if exec appears challenging; it is (semantically) the most challenging of all POSIX system calls. The others are all lot easier. Consider the exit command, which processes should utilise after they are through running, as an example of a straightforward one. It only takes one argument, the exit status (0 to 255), and the waitpid system call uses statloc to return it to the parent. In UNIX, processes' memory is split into three sections: the text segment, which contains the programme code, the data segment, which contains the variables, and the stack segment. There is a space in the address between them. The stack expands to fill up the space as required, but the data segment must be explicitly expanded using the system function brk, which specifies the new address at which it should finish. However, since programmers are advised to use the malloc library procedure for dynamically allocating storage, this call is not defined by the POSIX standard. Additionally, the underlying implementation of malloc was not deemed to be a suitable subject for standardisation since few programmers use it directly, and it is unlikely that anyone even notices that brk is not in POSIX.

The file system is involved in many system calls. We will study calls that affect individual files in this part; calls that affect directories or the whole file system will be covered in the next section. A file must be opened before it can be read or written to. This call gives the file name to open, either as an absolute path name or relative to the working directory, along with an O RDONLY, O WRONLY, or O RDWR code that denotes whether the file is open for reading, writing, or both operations. The O CREAT option is used to create a new file. The file descriptor that is returned May then be read or written to. The file descriptor will then be available for reuse on a later open when the file is closed by close.

Without a doubt, read and write are the most frequently utilised calls. We already watched read. The same criteria apply to write. While the majority of programmes read and write files in a sequential order, certain applications need programmes to be able to randomly access any region of a file. Each file has a pointer that identifies its present location inside the file. In a sequential read (write), it often leads to the next byte to be read (written). The lseek command modifies the location pointer's value, allowing future read and write requests to start wherever in the file. Lseek takes three inputs: the file descriptor for the file, the file position, and a third argument that indicates whether the file position is relative to the files beginning, middle, or end. Lseek returns the absolute location in the file (in bytes) after a pointer change. UNIX maintains track of a file's size, latest modification time, file mode (normal file, special file, di- rectory, etc.), and other details. Programs may use the stat system call to request to access this data. The first argument designates the file that will be examined, and the second is a reference to a structure that will hold the information. For an open file, the fstat call performs the same function.

System Calls for Directory Management A few system calls have more of a focus on directories or the whole file system than they do on a single file, as was the case in the preceding section. The first two calls, mkdir and rmdir, respectively create and delete empty directories. Link is the next call. Its function is to enable the appearance of the same file under two or more names, often in several directories. A frequent use is to let many programmers on the same team to share a same file while having each of them have the file appear in his or her own directory, maybe with a distinct name. Having a shared file implies that any modifications made by any team member are immediately accessible to the other team members as there is only one file, which is different from providing each team member a private copy. Changes made to one copy of a file after it has been copied do not impact the other copies.

Knowing how link works will probably make it more obvious what it accomplishes. The i-number assigned to each file under UNIX serves as its unique identification. An index into a database of i-nodes, one for each file, this i-number indicates who owns the file, where its disc blocks are located, and other information. Simply put, a directory is a file with a list of (i-number, ASCII name) pairs. Each directory entry in the early versions of UNIX was 16 bytes long—2 bytes for the i-number and 14 bytes for the name. While lengthier file names now need a more complex format, a directory is still conceptually a collection of (i-number, ASCII name) pairs. The other one is left behind if either one is subsequently deleted using the unlink system function. The file gets deleted from the disc if both are relocated because UNIX detects that there are no entries referring to it (a field in the i-node maintains track of the number of directory entries pointing to the file). Two file systems may be combined into one using the mount system function, as we have already stated. The root file system, which includes the executable (binary) versions of frequently used commands and other files, is often located on one (sub) partition of a hard drive while user files are located on a different (sub) partition. The user may then insert a USB drive with files for reading. A file on drive 0 may be accessed after the mount call by simply using its path from the root directory or the working directory, regardless of which disc it is on. In actuality, additional drives may be installed anywhere along the tree. Without having to worry about which device a file is on, removable media may be included into a single integrated file hierarchy thanks to the mount function. Although while this example uses CD-ROMs, the same technique may be used to mount USB sticks, external hard drives, and small devices—also known as partitions—on hard discs. Using the umount system call, a file system may be unmounted when it is no longer required.

Different System Calls

There are also other additional system calls. Here, we'll focus on only four of them. The current working directory is modified by the chdir command. Following the call, /usr/ast/test/xyz will be opened with an open on the file xyz. It is no longer necessary to constantly type (long) absolute path names thanks to the idea of a working directory. Every file in UNIX has a protection mode. The read-write-execute bits for the owner, group, and others are included in the mode. A file's mode may be changed using the chmod system function. For instance, one may run chmod ("file", 0644); to make a file read-only for everyone except the owner.

Users and user processes may notify one another by using the kill system function. A signal handler is executed when a procedure is ready to receive a certain signal. When a signal arrives and the process is unprepared to handle it, the process is terminated (hence the name of the call). Many time management techniques are defined by POSIX. For instance, time simply gives the current time in seconds, with 0 being midnight on January 1, 1970. The greatest value time may return on systems employing 32-bit words is 232 + 1 seconds (assuming an unsigned integer is used). This number translates to little over 136 years. Hence, 32-bit UNIX systems will malfunction in the year 2106, much like the infamous Y2K bug, which would have caused havoc with the world's computers in 2000 if not for the enormous amount of work the IT industry put into repairing it. It is recommended that you upgrade your 32-bit UNIX system to a 64-bit one before the year 2106 if you currently own one.

The Win32 API for Windows

We have mostly concentrated on UNIX thus far. It's time to take a quick look at Windows now. The relative programming paradigms used by Windows and UNIX are fundamentally different from one another. A UNIX application is made up of code that performs various tasks and makes system calls to request the delivery of certain services. Windows programmes, on the other hand, are often event-driven. The primary programme waits for an event to occur before invoking a method to deal with it. Common occurrences include keystrokes, mouse movements, mouse button presses, and USB drive insertions. The event is then processed, the screen is updated, and the internal programme state is updated by calling handlers. Overall, this results in a somewhat different programming style than UNIX, but because this book's concentration is on operating system function and structure, these various programming models won't worry us too much. Naturally, Windows contains system calls as well. In UNIX, the library methods (like read) used to call the system calls have a nearly 1:1 connection to the system calls themselves. In other words, there is typically one library operation invoked to initiate each system call. Moreover, there are only roughly 100 procedure calls in POSIX.

The situation is much different with Windows. First off, there is a significant degree of decoupling between library calls and real system calls. Programmers are required to utilise the Win32 API (Application Programming Interface), which Microsoft has developed, to access operating system services. With Windows 95, all Windows versions (partially) use this interface. Microsoft is still able to alter the actual system calls over time (even from release to release), without affecting already-written applications, because to the decoupling of the API interface from the system calls itself. Because of the many additional calls that current versions of Windows have that weren't previously accessible, it is unclear exactly what Win32 is. Win32 refers to the interface that is supported by all Windows versions in this area. Win32 offers cross-version Windows interoperability.

There are thousands of Win32 API calls, which is a very significant amount. Also, even while many of them do involve system calls, a sizeable portion of them are exclusively executed in user space. Because of this, it is hard to distinguish between a system call (carried out by the kernel) and a simple user-space library call while using Windows. In reality, what is a user space operation in one version of Windows may be a system call in another, and vice versa. As Microsoft promises that the Win32 procedures will be stable throughout time, we shall utilise them (where applicable) while discussing the Windows system calls in this book. Yet, it is important to keep in mind that not all of these are genuine system calls (i.e., traps to the kernel). There are a tonne of calls in the Win32 API for controlling windows, geometric objects, text, typefaces, scrollbars, dialogue boxes, menus, and other GUI elements. These are system calls if the graphics subsystem is running in the kernel (which is true for certain Windows versions but not all); otherwise, they are purely library calls. Should we or shouldn't we address these calls in this book? Although though they may be handled by the kernel, we have chosen not to as they are not actually linked to how an operating system works.

A System's Operating Structure

It is time to examine inside operating systems now that we have seen how they seem from the outside (i.e., the programmer's interface). In order to gain a sense of the range of options, we will look at six distinct architectures that have been tested in the following sections. While by no means comprehensive, they provide a sense of some concepts that have been put to the test in real-world settings. These six designs—monolithic systems, layered systems, microkernels, client-server systems, virtual machines, and exokernel.

Integrated Systems

The monolithic technique, by far the most typical structure, executes the whole operating system as a single application in kernel mode. The operating system is composed of several operations that are connected to form a single sizable executable binary software. Any procedure in the system is free to call any other one when this strategy is used if the latter offers the helpful calculation that the former requires. Having allowed to call whatever process you want is tremendously useful, but having thousands of procedures that are free to call one another may result in a system that is cumbersome and hard to comprehend. Moreover, if any of these processes fail, the operating system as a whole will crash. By employing this approach, one first compiles each individual procedure (or the files containing the procedures), and then uses the system linker to tie them all together into a single executable file. This creates the real object programme of the operating system. There is basically no information concealing since every operation can see every other procedure (as opposed to a struc- ture containing modules or packages, in which much of the information is hidden away inside modules, and only the officially designated entry points can be called from outside the module). Yet, it is still feasible to have some structure in monolithic systems. The operating system's services (system calls) are requested by placing the arguments in a specific location (such as on the stack) and then performing a trap instruction. This command changes the computer from user mode to kernel mode, giving the operating system control. According to this architecture, a single service process handles and executes each system call. The service procedures depend on the utility procedures to perform tasks like retrieving data from user applications. Several operating systems allow loadable extensions, such as I/O device drivers and file systems, in addition to the core operating system that is loaded when the machine boots. On demand, these components are loaded. They are referred to as shared libraries in UNIX. Windows refers to them as DLLs (Dynamic-Link Libraries). On Windows computers, there are well over 1000 of these in the C:Windowssystem32 directory, and they all have the.dll file extension.

Stratified Systems

The method may be extended to structure the operating system as a hierarchy of levels, with each layer built upon the one below it. The first system designed in this manner was the THE system created by E. W. Dijkstra and his students in 1968 at the Technische Hoge-school Eindhoven in the Netherlands. The Electrolog-ica X8, a Dutch computer with 32K of 27-bit words, was used with the THE system, which was a straightforward batch system (bits were expensive back then). The system has six levels. When interrupts or timers occurred, Layer 0 handled CPU allocation and switching between tasks. The system's upper layers were made up of sequential processes, each of which could be designed without having to take into account the fact that there was more than one processor in use. In other words, layer 0 enabled the CPU's fundamental multiprogramming. Memory management was handled by Layer 1. It allotted space for processes in the main memory as well as on a 512K word drum used to store the pages of processes for cases the main memory was insufficient. The layer 1 software took care of making sure that pages were brought into memory when they were required and discarded when they were not needed, so operations above layer 1 did not have to worry about whether they were in memory or on the drum. Communication between each process and the operator console was handled by Layer 2. Each process basically had its own operator console on top of this layer. I/O device management and information stream buffering were handled by Layer 3 for these devices. Instead of actual devices with many peculiarities, processes above layer 3 might deal with abstract I/O devices with desirable features. The user applications were located in layer 4. They were free from having to worry about managing processes, memory, consoles, or I/O. Layer 5 housed the system operator procedure. The MULTICS system included a further expansion of the layering idea. MULTICS was characterised as having concentric rings rather than tiers, with the inner rings having greater privileges than the outer ones (which is ef- fectively the same thing). A TRAP instruction, whose parameters were rigorously examined for validity before the call was permitted to continue, was required whenever a process in an outer ring attempted to call a procedure in an inner ring. With MULTICS, each user process had access to the complete operating system, but the hardware allowed for the designation of some procedures (really memory segments) as being protected from reading, writing, or execution. While every component of the system was eventually connected to create a single executable programme, the THE layering method was essentially simply a design assistance. In contrast, in MULTICS, the hardware enforced the ring mechanism, which was very much present at run time. The ring approach has the benefit of being easily expanded to organise user subsystems. To prevent students from changing their marks, a professor may, for instance, build a programme that tests and scores student programmes and execute it in ring n while the student programmes run in ring n 1.

Microkernels

The designers may choose where to draw the line separating the kernel from the user when using the layered method. The kernel used to include all the layers, however that is no longer essential. In fact, there is a compelling argument for keeping as little code as possible in kernel mode since kernel defects have the potential to completely crash the system. User processes, however, may be configured to have less power so that a defect there might not be catastrophic.

Many academics have routinely examined how many problems there are for every 1000 lines of code (e.g., Basilli and Perricone, 1984; and Ostrand and Weyuker, 2002). While problem density varies depending on module size, age, and other factors, an approximate number for important industrial systems is two to ten defects per thousand lines of code. Thus, a five million line monolithic operating system is likely to have between 10,000 and 50,000 kernel flaws. Of course, not all of them are catastrophic because some flaws could do things like send an inaccurate error message in a condition that happens very seldom. Yet, despite the vast quantity of software in these products, TV sets, stereos, and autos do not come with reset buttons, although computer makers do. This is because operating systems are sufficiently problematic. The fundamental concept behind the microkernel architecture is to achieve high reliability by segmenting the operating system into manageable, well-defined modules, of which only one the microkernel runs in kernel mode and the others as comparatively power-efficient standard user processes. A defect in one of them may specifically crash the device driver or file system, but not the whole system, since each is executed as a distinct user process. In other words, a defect in the audio driver will make the sound distorted or halt, but it won't cause the computer to crash. A flawed audio driver, on the other hand, might simply refer to an erroneous memory location and instantly cause the system to a complete stop in a monolithic system where all the drivers reside in the kernel. Just 12,000 lines of C and 1400 lines of assembler are used in the MINIX 3 microkernel to perform relatively basic tasks like switching processes and handling interrupts. The operating system's remaining functions are carried out by the C code, which also handles interprocess communication (by transferring messages between processes) and provides a set of roughly 40 kernel calls. These calls carry out tasks including establishing memory mappings for new processes, transferring data across advertisement spaces, and hooking handlers to interrupts, among others contains the Sys call handlers for the kernel. The scheduler interacts closely with the clock device driver, hence it is also included in the kernel. Several user processes are used to operate the other device drivers. The system is composed of three layers of processes that all operate in user mode outside of the kernel. Device drivers are located on the lowest layer. As they operate in user mode, they lack direct access to the I/O port space and are unable to send I/O instructions. Instead, the driver creates a structure that specifies which values should be written to which I/O ports and then makes a kernel call instructing the kernel to do the write in order to programme an I/O device. By using this method, the kernel may verify that the driver is writing from or reading from I/O that it is permitted to use. As a result, an unreliable audio driver cannot unintentionally write to the disc (unlike in a monolithic architecture). The servers, which carry out the majority of the operating system's tasks, are located in another user-mode layer above the drivers. A process manager generates, kills, and manages processes, while one or more file servers handle the file system(s). By sending brief messages to the servers and requesting the POSIX system calls, user applications may access operating system services. One of the file servers receives a message from a process that needs to read something, for instance, telling it what to read. The reincarnation server, whose responsibility it is to verify that all other servers and drivers are operating properly, is one intriguing server. In the case that one is found to be defective, it is automatically changed without user input. The system may attain great dependability in this fashion and is self-healing. The power of each process is constrained by the system's many limitations. As previously mentioned, drivers may only access approved I/O ports, but access to kernel calls and the capacity to send messages to other processes are also restricted per-process. Other processes may be given a limited amount of permission to allow the kernel access to their address spaces by other processes. A file system could let the disc driver to allow the kernel to place a freshly read-in disc block at a particular location inside the file system's address space, as an example. The total of all these restrictions means that each driver and server only has the power necessary to carry out their specific tasks. This significantly reduces the amount of harm that a defective component may do. Putting the method for doing something in the kernel but not the policy is a concept that is somewhat linked to having a minimal kernel. Consider the scheduling of processes to help illustrate this notion better. Every process should be given a numerical priority, and the kernel should then execute the process with the greatest priority that can be executed. This is a very straightforward scheduling technique. The kernel has a system that finds the process with the greatest priority and starts it. User-mode processes are capable of carrying out the policy, which assigns precedence to processes. The kernel can be made smaller and policy and mechanics can be separated in this fashion.

Model for Client-Server

Differentiating between two kinds of processes—the servers, each of which offers a different service, and the clients, who make use of these services—represents a modest modification of the microkernel concept. The client-server model is referred to as this one. Microkernels are often the lowest layer, however they are not necessary. The existence of client and server processes is essential. The most common method of communication between clients and servers is message passing. A client process creates a message describing the service it needs and delivers it to the right service in order to acquire it. The service then completes the task and replies with the solution. While there may be certain improvements if the client and server are running on the same computer, fundamentally we are still talking about message forwarding. The clients and servers running on various computers linked by a local or wide-area network is an obvious expansion of this concept. Clients and servers interact by exchanging messages, therefore clients do not need to be aware of whether the messages are processed locally on their own machines or forwarded across a network to servers on a different system. The client experiences both situations in the same way: requests are submitted, and responses are returned. The client-server paradigm is a generalisation that may be used to both a single system and a network of machines. An increasing number of systems employ users' home Computers as clients and huge machines operating elsewhere as servers. In actuality, this is how a lot of the Web works. A PC requests a Web page from a server, and the server responds with the requested Web page. In a network, this is a common use of the client-server architecture.

Virtual Machines: Initially, OS/360 was only available as batch systems. Nevertheless, many 360 users desired the ability to work interactively at a terminal, therefore a number of organisations, both within and outside of IBM, chose to develop timesharing solutions for it. Few sites switched to the official IBM timesharing system, TSS/360, since it was so large and sluggish when it eventually arrived after being delivered late. After spending almost $50 million on its development, it was finally banned. But, a team at IBM's Cambridge, Massachusetts, Scientific Center created a fundamentally different technology, which IBM finally approved as a product. On IBM's current mainframes, the zSeries, which are heavily used in large corporate data centres, for example, as e-commerce servers that handle hundreds or thousands of transactions per second and use databases with sizes that reach millions of gigabytes, a linear descendant of it, called z/VM, is now widely used.

VM/370: This system was based on a wise observation: a timesharing system offers (1) multiprogramming and (2) an expanded machine with a more user-friendly interface than the raw hardware. It was initially known as CP/CMS and subsequently changed to VM/370 (Seawright and MacKinnon, 1979). To totally separate these two roles is the goal of VM/370. A number of virtual machines are provided to the layer above by the virtual machine monitor, the brains of the system, which operates on the system's bare hardware and multiprogramming. These virtual computers are not expanded machines with files and other good features, however, unlike all other operating systems. These are identical replicas of the machine's basic hardware, including kernel/user mode, I/O, interrupts, and all other features. Each virtual machine may run any operating system that can run directly on the bare hardware since each one is similar to the real hardware. Various operating systems may operate on distinct virtual machines, and this happens often. In the original IBM VM/370 system, some users used the single-user, interactive CMS (Conversational Monitor System) system for interactive timesharing users while others used OS/360 or another big batch or transaction-processing operating system. Programmers preferred the latter. When a system call was made by a CMS programme, the call was made to the operating system on its own virtual machine rather than to VM/370, precisely as it would be if the CMS programme were executing on a physical computer as opposed to a virtual one. After that, CMS sent the standard hardware I/O commands to read its virtual disc or whatever else was required to complete the call. VM/370 captured these I/O instructions and then performed them as part of its emulation of the actual hardware. Each of the components might be made far more straightforward, adaptable, and easy to maintain by totally separating the multiprogramming capabilities and offering an expanded machine. Instead of stripped-down single-user systems like CMS, z/VM is often utilised to run several full operating systems in its present iteration. As an example, the zSeries may run one or more virtual Linux computers alongside conventional IBM operating systems.

CHAPTER 8

VIRTUAL MACHINES

Dr. Jaykumar Padmanabhan, Associate Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - p.jaykumar@cms.ac.in

Although a few other businesses, such as Oracle and Hewlett-Packard, have recently added virtual machine support to their high-end enterprise servers, and although IBM has had a virtual machine product available for four decades, the concept of virtualization has largely been ignored in the PC world until recently. Yet in recent years, it has become a hot subject as a result of a confluence of new requirements, new software, and new technology initial needs It has long been customary for many businesses to operate their mail servers, Web servers, FTP servers, and other servers on separate machines, sometimes using different operating systems. They see virtualization as a method to operate them all on the same system without risking the failure of all of them in the event that one server crashes.

The field of web hosting is also a prominent place for virtualization. Customers of web hosting are only given the option of shared hosting (which just grants them a login account on a web server but offers them no control over the server software) or dedicated hosting in the absence of virtualization (which gives them their own machine, which is very flexible but not cost effective for small to medium Websites). A single physical system may operate several virtual machines that each seem to be a full machine when a web hosting provider makes virtual machines available for rent. Virtual machine renters may use whatever operating system and programme them like, but for a fraction of the price of a dedicated server (because the same physical machine supports many virtual machines at the same time).

The term "virtual machine monitor" has been renamed to "type 1 hypervisor," which is commonly used these days because "virtual machine monitor" requires more keystrokes than people are willing to put up with right now. Another use of virtualization is for end users who want to be able to run two or more operating systems at the same time, say Windows and Linux, because some of their favourite application packages run on one and some run on the other. Be aware that that the phrases are often used synonymously. Although the appeal of virtual machines now is undeniable, implementation was the issue back then. A computer's CPU must be virtualizable if virtual machine software is to be executed on it (Popek and Goldberg, 1974). This is the issue in a nutshell. It is crucial that the hardware trap to the virtual-machine monitor when an operating system running on a virtual machine (in user mode) performs a privileged instruction, such as changing the PSW or performing I/O, so the instruction may be emulated in software. The Pentium, its predecessors, and its clones stand out among CPUs that simply disregard efforts to execute privileged instructions in user mode. The lack of interest in the x86 world is due to this characteristic, which rendered it difficult to run virtual machines on this technology. There were interpreters for the Pentium that operated on the processor, such as Bochs, but due to their performance loss of one to two orders of magnitude, they were not suitable for use in serious work.

By translating code blocks on the fly, storing them in an internal cache, and then reusing them if they were performed again, several of these early research initiatives enhanced performance over inter- preters like Bochs. This greatly increased performance and produced what we shall refer to as machine simulators. Nonetheless, even though this method, known as binary translation, helped to ameliorate the situation, the resultant computers were still too slow to be used in commercial settings where speed is crucial.

The addition of a kernel module to handle some of the labor-intensive tasks was the next step in enhancing performance. Nowadays, this hybrid approach is used by all commercially available hypervisors, such as VMware Workstation (and have many other improvements as well). Even while we would like to call them type 1.7 hypervisors to reflect the fact that they are not wholly user-mode programmes, we will (rather reluctantly) go along and use this nomenclature for the remainder of this book. In reality, a type 2 hypervisor differs from a type 1 hypervisor in that it leverages the host operating system and file system to create processes, store files, and perform other operations. A type 1 hypervisor must carry out all of these tasks on its own since it lacks any underlying support. The guest operating system is installed on a virtual disc, which is just a large file in the file system of the host operating system, once a type 2 hypervisor has begun reading the installation CD-ROM (or CD-ROM image file) for the selected guest operating system. Since there is no host operating system to store files on, Type 1 hypervisors are unable to do this. They have a raw disc partition that they are responsible for managing. The guest operating system starts some background operations and then normally launches a GUI when it boots up, just as it would on real hardware. While not really being running on bare metal, the guest operating system appears to the user to operate in the same manner. Control instructions may be handled differently by altering the operating system to eliminate them. This strategy is paravirtualization rather than genuine virtualization. We shall go into greater depth about virtualization.

Virtual Machine for Java

Running Java applications is another purpose for virtual machines, but in a somewhat different method. Sun Microsystems created the Java programming language along with the JVM virtual machine, which is a kind of computer architecture (Java Virtual Machine). A software JVM interpreter generally runs the JVM code that is generated by the Java compiler. This method has the advantage that any machine with a JVM interpreter may receive and execute JVM code sent over the Internet. These could not have been delivered and ran elsewhere as readily if the compiler had produced binary programmes for the SPARC or x86 architectures, for example. Sun could have, of course, created a compiler that generated SPARC binaries and then released a SPARC interpreter, but JVM is a lot easier architecture to comprehend.

Exokernels

Partitioning the computer, or providing each user with a portion of the resources, is an alternative technique to replicating the real system as is done with virtual machines. The exokernel technique has the benefit of saving a layer of mapping. The virtual machine monitor must maintain tables to remap disc addresses because in the previous systems, each virtual machine believes it has its own disc, with blocks ranging from 0 to a maximum (and all other resources). The exokernel eliminates the requirement for this remapping. The only thing the exokernel needs to monitor is which virtual machine has been given access to which resource. As the exokernel just needs to keep the virtual machines out of each other's hair, this solution still has the benefit of isolating the multiprogramming (in the exokernel) from the user operating system code (in user space).

The World As Seen By C

Operating systems are typically big C (or sometimes C++) programmes made up of several components and built by numerous programmers. When people (like students) write little Java applications, the environment is considerably different from what is utilised to construct operating systems. This section aims to provide novice Java or Python programmers with a very basic introduction to the realm of creating operating systems.

Programming in C

This is not a tutorial in C; rather, it is a quick review of some of the major distinctions between C and other programming languages, particularly Python and Java. There are numerous parallels between Java and C as they are both based on C. While somewhat different, Python is nonetheless quite comparable. We concentrate on Java for ease of use. As examples of imperative languages having data types, variables, and control statements, consider Java, Python, and C. Integers (both short and long), characters, and floating-point numbers are the basic data types in C. Arrays, structures, and unions may be used to create composite data types. The if, switch, for, and while statements are examples of control statements that are comparable to those in Java in C. Both languages have nearly the same functions and parameters. Explicit pointers are a feature of C that Java and Python do not have. which declares c1 and c2 to be character variables and p to be a variable that refers to (i.e., contains the address of) a character. A pointer is a variable that points to (i.e., contains the address of) a variable or data structure. The variable c1 is initialised with the ASCII code for the character "c" in the first assignment. The second one gives the pointer variable p the location of c1. After the execution of these statements, the variable c2 also contains the ASCII code for the letter "c" since the third one sends the contents of the variable referenced to by p to the variable c2. The address of a floating-point integer should not be assigned to a character pointer since, in principle, pointers are typed. Yet, in reality, compilers allow such assignments, although sometimes with a warning. Pointers are a tremendously potent tool, but when used irresponsibly, they may also be a major cause of mistakes. Built-in strings, threads, packages, classes, objects, type safety, and garbage collection are a few features that C does not offer. The last one puts an end to operating systems. The library methods malloc and free are often used to directly allocate and release all storage in C programmes. Explicit pointers and the latter characteristic, which gives the programmer complete control over memory, are what make C a desirable language for creating operating systems. Even general-purpose operating systems are, in a sense, real-time systems. The operating system may only have a few microseconds after an interrupt before losing crucial data or performing some action. It is unbearable for the trash collector to start working at random times.

File Headers

An operating system project typically consists of a number of folders, each of which has numerous.c files that each contain the code for a different component of the system, as well as a few.h header files that include declarations and definitions needed by one or more code files. Simple macros may also be used in header files, such as #define BUFFER SIZE 4096, which enables programmers to name constants such that 4096 is substituted for BUFFER SIZE in the code when it is needed. It's best practise in C programming to give each constant a name, except 0, 1, and 1 and sometimes even those three. The bigger of j and k+1 may be stored in I by writing I = max(j, k+1) and getting I = (j > k+1? j: k+1). Macros can also take arguments, such as #define max(a, b) (a > b? a: b). Moreover, headers may include conditional compilation, which, if the macro X86 is specified, compiles into a call to the function intel int ack and nothing else. In order to isolate architecture-dependent code, conditional compilation is often employed. As a result, some code is only inserted when the system is built on an X86 processor, other code is only inserted when the system is compiled on a SPARC processor, and so on. With the #include directive, a.c file may physically include 0–1 header files. Also, there are a large number of header files that are shared by almost all.c programmes and are kept in a single directory.

Major Programming Initiatives

Each.c file is converted by the C compiler into an object file in order to construct the operating system. Binary instructions for the target computer are included in object files, which end with the.o suffix. Afterwards, the CPU will execute them directly. In the realm of C, there is nothing comparable to Java or Python bytes of code. The C preprocessor is the first iteration of the C compiler. Every time it encounters a #include directive while reading a.c file, the compiler retrieves the header file mentioned in the directive, processes it, expands macros, takes care of conditional compilation, and passes the results to the next run of the compiler as if they were physically included.

Operating systems are fairly huge (five million lines of code is not uncommon), therefore it would be inconvenient to have to completely recompile them if a single file is altered. On the other hand, it does need recompiling those files when a crucial header file that is present in hundreds of other files is changed. Without assistance, it is totally impossible to keep track of which header files each object file depends on. Thankfully, computers are excellent at doing just this kind of thing. The make software, which has several variations including gmake, pmake, etc., is used on UNIX systems to read the Makefile, which specifies which files are dependent on which other files. In order to construct the operating system binary, make first determines which object files are required. Next, for each of these, it checks to see whether any of the files it relies on (the code and headers) have changed since the previous time the object file was built. If so, recompiling that object file is necessary. Make uses the C compiler to recompile the necessary.c files after it has decided which ones must be done so, keeping the overall number of compiles to a minimal. There are programmes that automatically create the Makefile since it is error-prone while working on big projects.

After every.o file is prepared, it is sent to a software known as the linker so that it may merge all of the.o files into a single executable binary file. At this stage, interfunction references are resolved, machine addresses are moved as necessary, and any library functions that were called are also included. After the linker is completed, an executable programme is produced. On UNIX systems, this programme is often referred to as a.out.

The Run Time Model

The machine may be restarted and the new operating system launched once the operating system binary has been linked. Once it's operating, it could dynamically load components like device drivers and file systems that weren't statically included in the binary. The text (the programme code), the data, and the stack are three segments that make up the operating system during run time. Normally, the text segment is immutable and does not change while being executed. The size and initialization values of the data segment are predetermined, but they may be altered and expanded as necessary. Initially empty, the stack expands and contracts as functions are called and returned from. However, different systems operate differently. Typically, the text segment is positioned close to the bottom of memory, the data segment is positioned immediately above it and has the ability to grow upward, and the stack segment is positioned at a high virtual address and has the ability to grow downward. In each scenario, the hardware immediately executes the operating system code without the need of an interpreter or just-in-time compilation, as is typical for Java.

Operating Systems Research

The discipline of computer science is developing quickly, making future predictions difficult. Researchers at academic institutions and commercial research facilities are always coming up with new concepts, some of which are never implemented but others of which become the pillars of later products and have a significant influence on the market and customers. In retrospect, it is simpler to identify which is which than it was at the time. The fact that it often takes 20 to 30 years from concept to effect makes it particularly challenging to separate the wheat from the chaff. In 1958, for instance, President Eisenhower established the Department of Defense's Advanced Research Projects Agency (ARPA) in an effort to prevent the Army from sabotaging the Navy and the Air Force over the Pentagon's research budget. He had no intention of creating the Internet. The ARPANET, the first experimental packet-switched network, was created as a result of some academic research on the then-unknown idea of packet switching that was funded by ARPA. In 1969, it went live. The ARPANET was soon joined by additional ARPA-funded research networks, and the Internet was created. Academic scholars then joyfully exchanged emails with one another through the Internet for 20 years. The World Wide Web was created by Tim Berners-Lee at the CERN research facility in Geneva in the early 1990s, and Marc Andreesen created a graphical browser for it at the University of Illinois. Teenagers were tweeting all of a sudden, populating the Internet. President Eisenhower is likely squirming in his grave right now.

Operating system research has also significantly altered real-world systems. Until M.I.T. developed general-purpose timesharing in the early 1960s, the earliest commercial computer systems were all batch systems, as we previously explained. Until Doug Engelbart created the mouse and the graphical user interface at Stanford Research Institute in the late 1960s, all computers were text-based. We will briefly review some of the operating system research that has occurred over the last five to ten years in this part and similar sections throughout the book to give readers a taste of what may be in store. Certainly not everything is covered in this introduction. As these concepts have to at least make it through a rigorous peer review process in order to be published, it is mostly based on papers that have been presented at prestigious scientific conferences. The majority of computer science research is published through conferences rather than journals, in contrast to other scientific disciplines. ACM, the IEEE Computer Society, or USENIX published the majority of the papers mentioned in the research sections, and their (student) members may access them online.

Threads and Processes

We are ready to start a thorough investigation of the planning and development of operating systems. Every operating system's most fundamental idea is the process, which is an abstraction of an active programme. The operating system designer (and student) should have a clear understanding of what a process is as early as feasible since everything else is dependent on it. One of the first and most significant abstractions offered by operating systems is the concept of processes.

Even with just one CPU available, they enable the capacity to have (pseudo) concurrent operation. They convert a single physical CPU into several virtual CPUs. The process abstraction is essential to contemporary computing.

Processes

All contemporary computers often do many tasks at once. This truth may not be entirely understood by those used to using computers, thus a few examples may help to clarify the issue. Think about a Web server first. There are requests for Web sites coming in from all around. The server determines if the requested page is in the cache when a request is received. If it is, it is returned back; if not, a disc request to retrieve it is launched. Yet from the viewpoint of the CPU, disc requests consume a lot of time. Several additional requests could arrive while you're waiting for one to finish reading from the disc. If there are many discs available, some or all of the more recent ones can be transmitted to other discs before the first request is fulfilled. There must be a mechanism to model and manage this concurrency. Here, processes—and threads in particular—can be useful. Think of a user PC next. Many processes are secretly launched as the system boots, often without the user's knowledge. A process could be established, for instance, to watch for incoming email. The antivirus application may be supported by another process that regularly checks for fresh virus definitions. Moreover, when a user is browsing the Internet, explicit user processes may be running, printing documents, and backing up the user's images to a USB stick. A multiprogramming system supporting several processes is quite helpful in managing all of this activity. The CPU swiftly changes between processes in any multiprogramming system, with each lasting for tens or hundreds of milliseconds. While, technically speaking, the CPU only runs one process at a time, it may operate on numerous of them simultaneously for one second, creating the impression of parallelism. In order to contrast it with the genuine hardware parallelism of multiprocessor systems, individuals may sometimes use the term "pseudoparallelism" in this context (which have two or more CPUs shar- ing the same physical memory). People find it challenging to keep track of several, concurrent activities. As a result, operating system designers have developed a conceptual model (sequential processes) throughout time that makes dealing with parallelism simpler. The model, its applications, and some of its effects make up the topic.

Procedure Model

In this architecture, every piece of executable software on the computer, sometimes even the operating system, is arranged into a number of linked processes, or simply processes. A process is just an instance of a programme that is now running, complete with the counter, register, and variable values that are in effect at the time. Theoretically, every process has a separate virtual CPU. While the actual CPU changes back and forth between processes, it is much simpler to grasp the system if you imagine a number of processes operating in (pseudo) parallel rather than trying to keep track of the CPU's actual programme switching patterns. Multiprogramming refers to this quick movement back and forth between tasks.

Nevertheless, that assumption is no longer accurate since contemporary processors often have multicore architectures with two, four, or more cores. Multicore chips and multiprocessors in general, but for the time being, it is easier to focus on a single CPU at a time. Hence, if there are two cores (or CPUs), each of them can only execute one process at a time when we state that a CPU can only run one process at a time in reality. The pace at which a process completes its calculation will not be consistent and likely not even replicable if the identical processes are run again due to the CPU moving back and forth between the processes. As a result, processes must not be programmed with time assumptions. Imagine an audio process that plays music in the background of a high-definition video that is being played by another device. It tells the video server to begin playing, then performs an idle loop 10,000 times before playing back the audio since the audio should begin a bit later than the video. The video and audio will be frustratingly out of sync if the loop is a dependable timer, but if the CPU chooses to switch to another process while the idle loop is running, the audio process may not begin again until the matching visual frames have already come and gone. Special precautions must be taken to guarantee that essential real-time requirements, such as this one where certain events must take place within a predetermined number of milliseconds, are met. Most processes, however, are not impacted by the CPU's underlying multiprogramming or the relative speeds of several processes. While minor, the distinction between a process and a programme is vitally essential. Here, an analogy could be useful. Think about a computer scientist who enjoys preparing cakes for his small daughter's birthday. He has a recipe for birthday cake and a well-stocked kitchen with all the ingredients: flour, eggs, sugar, vanilla essence, and so on. The recipe serves as the analogy's programme, which is an algorithm defined in a suitable notation, and the computer scientist serves as the analogy's processor (CPU), and the input values are the ingredients for the cake. Our baker's reading of the recipe, gathering of the ingredients, and baking of the cake constitute the procedure. Imagine for a moment that the computer scientist's kid runs inside screaming like a little boy after being stung by a bee. The computer scientist pulls up a first aid manual and starts by following the instructions in it after recording where he was in the recipe (the state of the current procedure is stored). Here, we see the CPU switching between a lower-priority operation (baking) and a higher-priority process (providing medical care), each of which has a unique programme (recipe versus first aid book). The computer scientist returns to his cake and picks off where he left off after the bee sting has been treated. The main concept is that a process is a kind of action. It has an input, an output, a programme, and a state. Several processes may share a single processor, and a scheduling mechanism may be used to decide when to stop working on one process and start working on another. A programme, on the other hand, is something that may be saved on a disc but does nothing. It is important to keep in mind that a programme counts as two processes if it is executing twice. For instance, if two printers are available, it is often feasible to run a word processor twice or print two files simultaneously. It makes no difference when two processes are concurrently executing the same software; they are separate processes. While the operating system may be able to share the code such that only one copy is in memory, this is a technical feature that has no bearing on the conceptual scenario of two processes running simultaneously.

Process Development

Process creation has to be supported by operating systems. It may be conceivable to have all the processes that will ever be required be there when the system starts up in extremely basic systems or in systems built for executing just a single application (for example, the controller in a microwave oven). Yet, general-purpose systems must include a mechanism for starting and stopping operations as necessary. We'll now examine a few of the problems. Usually, several processes are started when an operating system is booted. Several of these are foreground processes, meaning they work for and interact with users (humans). Others operate in the background, are not connected to individual users, but serve a particular purpose. As an example, one background process may be created to receive incoming email, resting the most of the day but waking up immediately when email comes. When a request for a Web page hosted on that system is received, another background process may be set up to accept the request and wake up to handle it. Daemons are processes that run in the background to perform tasks like email, Web sites, news, printing, and other activities. Many them are often seen in large systems. The ps application in UNIX may be used to display a list of active processes. The task manager is available in Windows. New processes may be generated after the initial ones, in addition to those created at boot time. In order to accomplish its tasks, a running process often makes system calls to start one or more new processes. When the job to be done can be simply defined in terms of numerous connected, but otherwise separate interacting processes, developing new processes is very useful. It may be convenient to construct one process to collect the data and place them in a shared buffer while a second process removes the data items and processes them, for instance, if a huge volume of data is being retrieved over a network for further processing. Allowing each task to execute on a distinct CPU on a multiprocessor may help speed up the process. Users may launch a programme in interactive systems by entering a command or (doubly) clicking on an icon. Any of these operations launches a new process with the chosen application running within it. The newly begun process takes control of the window on command-based UNIX systems that run X. When a Windows process is launched, it lacks a window; nevertheless, it has the ability to generate one (or many), and the majority of them do. Users may have many windows open at once in both systems, each executing a different task. The user may pick a window and engage with the process using the mouse, for instance, by giving input as required.

Only the batch systems seen on huge mainframes are affected by the final scenario in which processes are generated. At the end of the day, consider inventory management at a chain of businesses. Here, users may send the system batch tasks (possibly remotely). The next task in the input queue is executed by a new process that the operating system launches when it determines that it has the resources to perform another job. Theoretically, in each of these situations, a new process is formed by having an already-running process make a system call for process creation. It might be a user process that is already running, a system process that has been started using the keyboard or mouse, or a batch manager process. To start the new process, that process performs a system call. This system call instructs the operating system to start a new process and, directly or indirectly, specifies which application should execute inside of it. Forking a new process in UNIX simply requires one system call: fork. This call duplicates the calling procedure exactly. After the fork, the parent and child processes share the same memory image, environment strings, and open files. There is no more programming. For instance, the shell forks off a child process and has the child run the command sort when a user inputs the command into the shell. This two-step procedure is necessary to enable the child to modify its file descriptors before the execve but after the fork in order to achieve standard input, standard output, and standard error redirection.

In contrast, Windows uses a single Win32 function call called CreateProcess to manage the creation of new processes and loading the appropriate programmes into them. A pointer to a structure that contains information about the newly created process is returned to the caller in this call's ten parameters, which also include the programme to be executed, command-line parameters to feed that programme, various security attributes, bits that control whether open files are inherited, priority information, and a specification of the window to be created for the process (if any). Win32 provides roughly 100 other functions for managing and synchronising processes and associated issues in addition to CreateProcess. Once a process is established, the parent and child have separate address spaces on UNIX and Windows systems. If each process modifies a word in its address space, the change is not apparent to the other process. There are clearly two separate address spaces involved in UNIX; no writable memory is shared; the child's initial address space is a duplicate of the parent's. The programme text is shared across the two in certain UNIX implementations since it cannot be changed. Alternatively, the child may share all of the parent’s memory, but in that case the memory is shared copy-on-write, which means that whenever either of the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area. Again, no writable memory is shared. It is, nevertheless, feasible for a freshly started process to share some of its creator’s other resources, such as open files. In Windows, the parent's and child's address spaces are different from the start.

Processing is finished

A process begins operating once it is formed and performs its function. Yet nothing—not even processes—lasts forever. The majority of processes end because they have completed their tasks. A compiler conducts a system call to notify the operating system that it has completed compiling the programme that was sent to it. In UNIX, this call is known as exit, and in Windows, Exit Process. Programs that focus on screens also allow for voluntary termination. The user may always instruct word processors, web browsers, and other similar applications to close any open temporary files by clicking an icon or menu item on the toolbar. The procedure finds a fatal error, which is the second cause for termination.

For instance, the compiler will simply indicate the fact that no such file exists and quit if a user enters the command cc foo.c to build the programme foo.c. When provided incorrect settings, screen-oriented interactive programmes often do not terminate. Instead, a dialogue box appears with a request for the user to try again. The third reason for termination is a process fault, often brought on by a programming flaw. Examples include using a forbidden instruction, referring to memory that doesn't exist, or dividing by zero. In certain systems (like UNIX), a process may instruct the operating system that it wants to handle specific faults on its own. In this instance, when one of the errors occurs, the process is notified (interrupted) rather than being terminated. The fourth possible cause for a process to end is when it issues a system call instructing the operating system to kill another process. UNIX calls this operation kill. TerminateProcess is a Win32 function that is equivalent. In all situations, the murderer needs the required consent to harm the killee. Some systems automatically destroy any processes that a process has launched when it ends, whether it did so intentionally or not. Nevertheless, neither UNIX nor Windows operate in this manner.

Hierarchies in Process

As a process generates another process, the parent process and child process may continue to be connected in certain ways in some systems. A process hierarchy may be formed by the child process by itself producing further processes. Keep in mind that a process only has one parent, unlike plants and animals which employ sexual reproduction (but zero, one, two, or more children). A process is thus more akin to a hydra than, say, a cow. A process group in UNIX is made up of a process and all of its offspring and future grandchildren. All members of the process group presently connected to the keyboard get any signals sent by users from the keyboard (usually all active processes that were created in the current window). Each process may respond independently and either catch the signal, ignore the signal, or do the de-fault action, which is to allow the signal to terminate the process.

Let's examine how UNIX initialises itself when it is launched, immediately after the machine being booted, as another instance of where the process hierarchy plays a crucial role. In the boot image, a unique process with the name of init is present. It reads a file containing information about the number of terminals before it may execute. A new process is then started for each terminal. Some procedures hold off till a user logs in. When a login attempt is successful, a shell that can receive commands is launched. These instructions could launch more processes, and so forth. As a result, the whole system's processes may be grouped into a single tree with init at the root. Windows, in contrast, has no understanding of a process hierarchy. Every procedure is equivalent. The sole indication of a process hierarchy is the special token (referred to as a handle) that is supplied to the parent when a process is established and may be used to manage the child. The hierarchy may be invalidated by freely passing this token to another process. In UNIX, processes cannot disinherit their offspring.

Continuity States

While every process is a separate entity with its own programme counter and internal state, interactions with other processes are often necessary. It's possible for one operation to produce some output that another utilises as input. The second phase, which is grep in action, picks out any lines that include the term "tree." It might happen that grep is prepared to execute but there is no input awaiting it, depending on the relative speeds of the two processes (which rely on both the relative com- laxity of the programmes and how much CPU time each one has got). After then, it must block until input is available. When a process stops, it is usually because it is waiting for input that is not yet accessible, for which it is logically unable to proceed. A process that is theoretically prepared and capable of running could also be terminated because the operating system has chosen to temporarily provide the CPU to another process. These two circumstances are quite different. Since you can't process the user's command line until it has been input, the suspension occurs in the first scenario. In the second instance, it is a system technicality (not enough CPUs to give each process its own private processor).

The first two statements make sense together logically. The process is ready to execute in both situations, but in the second one there isn't a CPU available for it right now. The third stage differs significantly from the prior two in that the process is unable to operate even when the CPU is idle and unoccupied. As shown, there are four conceivable transitions between these three phases. When the operating system determines that a process cannot proceed at this time, transition 1 takes place. On certain systems, a system call like pause may be used by the process to enter a blocked state. In other operating systems, such as UNIX, a process is immediately halted if it reads from a pipe or special file (such as a terminal) without any input being available. The operating system's process scheduler, which is responsible for transitions 2 and 3, initiates them without the process being aware of it. When the scheduler determines that a process has completed its runtime and it is ready to give another process access to the CPU, transition 2 takes place. After the other processes have used up their fair share of the CPU and it is time for the first process to start using it again, transition 3 takes place. Scheduling, or choosing which process to run. To attempt to strike a balance between the opposing goals of efficiency for the system as a whole and justice to individual operations, several algorithms have been developed. When a process receives the external event for which it was waiting (such as the receipt of some input), transition 4 takes place. If no other processes are currently active, transition 3 will be activated and the process will begin to execute. Alternatively, it may have to wait a little time in the ready state until the CPU becomes available and it is turned around. It is considerably simpler to think about what is happening within the system when using the process model. Some of the processes execute software that executes user-typed instructions. Other system-wide processes take care of things like processing requests for file services or controlling the specifics of operating a disc or a tape device. The system decides to switch from running the currently running process to the disc process that was stalled while waiting for the interrupt when a disc interrupt occurs. As a result, we may consider user processes, disc processes, terminal processes, and other processes instead of interruptions since these processes block while they are waiting for anything to happen. The process that was waiting for it is unblocked and may resume running after the disc has been read or the character has been inputted.

Application of Procedures

The operating system maintains a table (an array of structures) called the process table, with one entry per process, to implement the process model. These items are sometimes referred to as process control blocks. This entry contains crucial information about the state of the process, such as its programme counter, stack pointer, memory allocation, the status of any open files, accounting and scheduling information, and everything else that needs to be saved when the process is changed from the running state to the ready or blocked state so that it can later be restarted as if it had never been stopped. The first column's fields have to do with process management. The other two are, respectively, file management and memory management. It should be emphasised that the exact fields the process table contains vary greatly on the system, however this figure provides an overview of the types of data required. We can now explain a bit more about how the appearance of several sequential processes is maintained on one (or each) CPU after looking at the process table. Each I/O class has an associated location known as the interrupt vector, which is normally located at a fixed point at the bottom of memory. It includes the interrupt service procedure's advertisement. Assume a disc interrupt occurs when user process 3 is active. The interrupt hardware pushes the programme counter, programme status word, and sometimes one or more registers from User Process 3 into the (current) stack. After that, the computer jumps to the location specified in the interrupt vector. All the gear can do is that. From this point on, the software, in particular the interrupt service method, is in charge. All interruptions begin by saving the registers, often in the entry for the current process in the process table. The data that the interrupt had previously placed onto the stack is then deleted, and the stack pointer is changed to refer to a makeshift stack that the interrupt handler uses. Since tasks like saving the registers and setting the stack pointer cannot even be expressed in high-level languages like C, they must instead be carried out by a short assembly-language routine, which is typically used for all interrupts since saving the registers always involves the same work, regardless of the reason for the interrupt.

Upon completion of this function, a C procedure is called to complete the remaining tasks for this particular interrupt type. (We presumptively write the operating system in C, the language of choice for all genuine operating systems.) The scheduler is summoned to determine who should run next after it has completed its task and may have perhaps made some process ready. Control is then returned to the assembly-language code, which starts the currently executing process by loading its registers and memory map. It is important to note that the details differ somewhat across systems. The important point is that after every interruption, the interrupted process returns to exactly the same condition it was in before the interruption. A process may be interrupted thousands of times while it is being executed.

CHAPTER 9

MODELING MULTIPROGRAMMING

Dr. Yavana Rani. S, Associate Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.yavanarani@cms.ac.in

The employment of many programmes might increase CPU use. Simply said, with five processes running in parallel and the average process only computing 20% of the time it is in memory, the CPU should be always occupied. Nevertheless, since it implicitly predicts that none of the five processes would ever be waiting for I/O at the same time, this model is too optimistic.

Threads

Each process in conventional operating systems has its own address space and control thread. In actuality, it is almost how a process is defined. Yet, it is sometimes preferable to operate many control threads in a single address space in a manner similar to parallel processing, almost as if they were independent processes (except for the shared address space). We will go through these scenarios and their ramifications in the sections that follow.

Use of Threads

It turns out that there are a number of benefits to having tiny threads, or little processes. So let's look at a few of them. The fundamental benefit of threads is the simultaneous operation of several tasks in various applications. Some of these could sometimes block. The programming approach is simplified by breaking down such an application into several sequential threads that operate in a quasi-parallel fashion. This debate has already occurred. That is exactly the case in favour of procedures. We may consider parallel processes instead of interrupts, timers, and context switches. The capacity for the simultaneous entities to share an address space and all of its data among themselves is only now added with threads. Several processes (each with its own address space) will not function because of this need for certain applications.

The fact that threads may be created and destroyed more quickly than processes due to their lesser weight than processes is a second justification for having threads. A thread may be created 10–100 times quicker than a process in many systems. This property is advantageous to have when the required number of threads fluctuates quickly and dynamically. A performance argument is thus a third justification for using threads. When all of the threads are CPU-bound, there is no performance benefit from having them, but when there is significant compute and significant I/O, having threads enables these activities to overlap, accelerating the programme. Lastly, threads are helpful on systems with many CPUs because they enable true parallelism. We'll discuss this problem again.

The best way to understand the value of threads is to look at some specific instances. Consider a word processor as a starter. The document being written is often formatted precisely as it would appear on the printed page when using a word processor. To remove widows and orphans, which are unfinished top and bottom lines on a page that are deemed aesthetically unpleasant, all line breaks and page breaks are specifically in their proper and final locations. Assume the user is writing a book. From the author's perspective, it is simplest to maintain the whole book in a single file to make subject searches, global replacements, and other operations simpler. As an alternative, may be its own file. When the whole book has to be changed globally, having every part and subsection as a distinct file is a major pain since hundreds of files must then be manually altered, one at a time. For instance, all instances of "Draft Standard xxxx" must be changed to "Standard xxxx" at the last minute if proposed standard xxxx is adopted shortly before the book goes to print. If the whole book is one file, the replacements may usually be completed with only one command. If the book is divided into 300 files, however, each one has to be edited separately.

Think about what would happen if a user abruptly deleted one phrase from page 1 of an 800-page book. He punches in a command instructing the word processor to proceed to page 600 in order to make another adjustment after verifying the accuracy of the revised page (possibly by searching for a phrase occurring only there). Since the word processor cannot predict the first line of page 600 until it has processed all the preceding pages, it is now compelled to reformat the whole book up to page 600 immediately. Page 600 may not be shown for some time, which might make the user irate. Here, threads may be useful. Let's assume that the word processor is a two-threaded software. When one thread communicates with the user, the other conducts background reformatting. The interactive thread instructs the reformatting thread to reformat the whole book as soon as the sentence on page 1 is removed. While the other thread is working furiously in the background, the interactive thread is listening to the keyboard and mouse and responds to basic requests like scrolling to page 1. It's possible that the reformatting will be finished before the user requests page 600, in which case it will be shown right away. If the software were single-threaded, keyboard and mouse instructions would be disregarded anytime a disc backup began until the backup was complete. The user would undoubtedly consider this to be a slow performance. The disc backup might also be interrupted by keyboard and mouse events, which would allow for high performance but result in a complicated interrupt-driven programming style. The programming paradigm is significantly easier with three threads. The user is the only thing that the first thread does. When instructed, the second thread reformats the document. Periodically, the third thread copies the contents of Memory to disc. It should be obvious that in this case, having three distinct processes would be ineffective since each of the three threads has to work on the page. Instead of having three separate processes, there are three threads, which share a memory and may all access the same page. Three procedures would make this impossible. There are several additional interactive applications that operate in a similar manner. An electronic spreadsheet, as an example, is a software that enables a user to maintain a matrix, some of whose members are user-provided data. Additional factors are calculated using possibly complicated calculations depending on the provided data. Several additional items may need to be recalculated when a user changes one element. The interactive thread may enable the user to make further modifications while the calculation is taking place by delegating the precomputation to a background thread. Similar to that, a third thread may manage routine disc backups by itself.

Now take a look at another scenario in which threads are advantageous: a server hosting a website. When a page request is received, the requested page is provided back to the client. Most websites have pages that are visited more often than others. For instance, Sony's main page is visited far more often than a page buried deep in the tree that lists the technical details of any given camera. This fact is leveraged by web servers to boost speed by keeping a collection of frequently used pages in main memory to avoid having to access the disc. Caches are utilised in many different settings and are the name given to this kind of collection. Here, the dispatcher, a single thread, receives incoming network requests for tasks. It analyses the request, selects an idle (i.e., blocked) worker thread, and passes the request to it. This may be accomplished by putting a pointer to the message into a specific word associated with each thread. The dispatcher then awakens the dozing employee, switching its status from blocked to ready.

When waking up, the worker determines whether it can fulfil the request using the Web page cache, to which all threads have access. If not, a read operation to get the page from the disc is started, and the process stalls until the disc operation is finished. When a thread stops during a disc operation, another thread possibly the dispatcher so it may pick up additional work or perhaps another worker who is now prepared to run is picked to continue processing. Using this paradigm, the server may be created as a group of sequential threads. The software that the dispatcher uses to receive job requests and pass them on to employees is an unending loop. The code for each worker is an endless loop that accepts a request from the dispatcher and checks the Web cache to see whether the requested page is there. If so, the client is notified, and the worker blocks until a fresh request comes in. If not, it retrieves the page from the disc, sends it back to the client, and then waits for another request before continuing. Think about a possible Web server implementation without threads. It may run as a single thread, for instance. The Web server's main loop receives a request, analyses it, and executes it until it is finished before receiving the next. The server is idle and doesn't handle any more incoming requests while it waits for the disc. The CPU merely sits idle while the Web server waits for the disc if it is operating on a dedicated computer, which is often the case. As a consequence, far fewer requests per second can be handled. As a result, threads perform far better, yet they are still often programmed sequentially. A single-threaded Web server and a multi-threaded Web server are the two architectures we have seen so far. Even if the system designers believe the performance loss caused by single threading is unacceptable, let's say threads are not accessible. A third strategy is feasible if a nonblocking read system call is provided. The only thread is responsible for processing each request that is received. If the cache can handle it, great; if not, a nonblocking disc operation is initiated. The server then goes and retrieves the next event after recording the current request's status in a table. A request for new work or a disc response to a prior activity might be the next event. If it is fresh labour, it is begun. If there is a reply from the disc, the reply is analysed and the pertinent data is retrieved from the database. A response with nonblocking disc I/O will most likely be in the form of a signal or interrupt. The "sequential process" paradigm that we had in the previous two situations is gone in this approach. Every time the server moves from working on one request to another, the status of the calculation must be explicitly saved and restored in the database. We are essentially replicating the threads and their stacks manually. A finite-state machine is a design like this, in which each computation has a stored state and there is a set of events that may take place to modify the state. The field of computer science makes extensive use of this notion. What threads provide ought to be now be evident. They enable parallelism while maintaining the concept of sequential processes that make blocking calls (such as for disc I/O). Programming is made simpler by blocking system calls, and performance is enhanced by parallelism. While speed is sacrificed, the single-threaded server retains the simplicity of blocking system calls.

Applications that need to handle enormous volumes of data are a third situation in which threads are advantageous. A block of data is typically read in, processed, and then written out again. The issue here is that the process will halt while data are being received and sent if only blocking system calls are available. It is obviously inefficient to let the CPU sit idle when there is a lot of work to be done, thus it should be avoided whenever feasible. Threads provide a remedy. An input thread, a processing thread, and an output thread might be used to organise the process. Data is read into an input buffer by the input thread. Data from the input buffer are removed by the processing thread, which then processes them and stores the outcomes in an output buffer. These outcomes are written back to disc via the output buffer. Input, output, and processing may all occur concurrently in this manner. Of course, for this paradigm to operate, just the calling thread and not the whole process must be blocked by a system call.

Traditional Thread Model

After examining the potential benefits and practical applications of threads, let's take a closer look at the concept. The resource grouping and execution ideas serve as the foundation for the process model. Threads are handy in this situation since it might be useful to separate them at times. We will first study the traditional thread model before moving on to the Linux thread model, which obfuscates the distinction between processes and threads. A process may be seen as a technique to gather together materials that are connected. An address space for a process contains programme text, data, and other resources. Open files, child processes, pending alarms, signal handlers, accounting data, and more might be among these resources. They can be controlled more readily if they are combined into a process. The execution thread, sometimes abbreviated to simply thread, is the other idea a process might have. A programme counter in the thread maintains track of which instruction should be executed next. It has registers where its active working variables are stored. The execution history is stored in a stack, with one frame for each procedure that has been called but has not yet been called back from. Even though a thread must run inside a process, the thread and the process in which it runs are distinct concepts and may be handled differently. Threads are the entities scheduled for execution on the CPU, whereas processes are utilised to combine resources together.

The process paradigm is improved by threads because they enable numerous executions to occur in the same process environment, mostly independently of one another. The parallel operation of many threads inside a single process is comparable to the parallel operation of several processes within a single computer. The threads share an address space and other resources in the first scenario. Processes share physical memory, discs, printers, and other resources in the latter scenario. Threads are frequently referred to be lightweight processes since they share certain characteristics with processes. The practise of permitting several threads in a single process is sometimes referred to as multithreading.

A process's several threads are not as independent of one another as its many processes are. As every thread uses the exact same address space, they all use the same global variables. One thread may read, write, or even wipe out the stack of another thread since each thread has access to every memory address in the process' address space. As it is both impossible and unnecessary, there is no protection between threads. A process is always owned by a single user who, unlike numerous processes, which may be from different users and which may be antagonistic to one another, has probably generated many threads so that they may cooperate rather than fight. Process properties, not thread attributes, are listed in the first column. For instance, if one thread opens a file, all other threads in the process may read and write to that file. This makes sense given that the process, not the thread, serves as the unit of resource management. Each thread would be a different process if it had its own address space, open files, pending alarms, and other features. With the thread notion, we want to enable the sharing of resources across numerous threads of execution so that they may collaborate closely to complete a job.

A thread may be in any of the following states, similar to a typical process (a process with just one thread): running, blocked, ready, or terminated. The CPU is now being used by an active thread. A blocked thread, on the other hand, is awaiting an event that will unblock it. For instance, until input is entered, a thread performing a system call to read from the keyboard is stopped. A thread may stall as it waits for an outside event to occur or for another thread to unblock it. When it is time for one thread to run, another is already prepared to do so.

Processes typically begin with a single thread when multithreading is available. By using a library function like thread create, this thread has the power to start new threads. The name of a procedure that the new thread will execute is specified by an argument to thread create. As the new thread automatically uses the address space of the thread that created it, no address space specification is required (or even possible). Sometimes, threads have a parent-child connection and are hierarchical; however, this relationship is often absent, making all threads equal. The generating thread typically receives a thread identification that identifies the new thread, whether or not there is a hierarchical connection.

A thread may terminate itself after completing its task by using a library operation, such as thread exit. It then ceases to exist and cannot be scheduled. In certain threading architectures, a thread may wait for another thread to finish running by using a process like thread join. Until a (specific) thread exits, this method stops the thread that called it. In this sense, starting and ending a thread is quite similar to starting and ending a process, and both have roughly the same choices. Thread yield is a typical thread command that enables a thread to voluntarily cede control of the CPU to another thread. This call is crucial because, unlike with processes, there is no clock interrupt to truly enforce multiprogramming. Consequently, it is crucial for threads to be courteous and sometimes give up the CPU so that other threads may execute. Additional calls enable a thread to wait for a task to be completed by another thread, for a thread to declare that it has completed a task, and so forth. While threads are often helpful, they can cause a variety of difficulties in the programming approach.

Threads in POSIX

The IEEE has established a standard for threads in IEEE standard 1003.1c to enable the creation of portable threaded applications. Pthreads is the name of the threads package it defines. Most UNIX systems have it available. There are way too many function calls defined in the standard—over 60—to list them all here. The threads in Pthreads share a few characteristics. Every single one of them is comprised of an identifier, a collection of registers (including the programme counter), and a collection of characteristics that are stored in a structure. The properties include the thread's requirements, scheduling constraints, and the size of the stack. The pthread create function generates a new thread. The value of the function is the thread identification of the just started thread. With the exception of having arguments, this call purposely looks a lot like the fork system call, with the thread identifier serving as the PID and mostly being used to identify threads that are mentioned in other calls. A thread may end by using pthread exit after it has completed the task it was given. The thread is terminated by this call, and its stack is released. A thread often has to wait for another thread to perform its task and close before moving on. In order to wait for a certain other thread to end, the waiting thread uses the pthread join command. The argument specifies the thread identification of the thread to wait for. Sometimes a thread believes it has ran long enough and wishes to give another thread a chance to continue running despite not being logically blocked. It can do this by using the pthread yield command. As it is assumed that processes are extremely competitive and each seeks as much CPU time as possible, there is no such call for processes. Yet, since a process's threads collaborate and the same programmer always creates their code, the programmer sometimes requests that they give one another another shot. The two thread calls that follow discuss characteristics. The attribute structure connected to a thread is created by pthread attinit, which initialises it with default values. Fields in the attribute structure may be altered to alter these values (such as the priority).

Thread Implementation in User Space

User space and the kernel are the two primary implementation locations for threads. The decision is rather debatable, and a hybrid approach is also an option. We will now discuss these techniques in detail, including their benefits and drawbacks. The threads package may be totally located in user space like the first approach. They are unknown to the Ker- Nel. The kernel is responsible for controlling common, single-threaded processes. The first and most apparent benefit is the ability to construct a user-level threads package on an operating system without thread support. This category formerly included all operating systems, and some still do today. This method uses a library to implement threads.

Each process requires its own private thread table to keep track of the threads it contains while managing threads in user space. This database is comparable to the kernel's process table, with the exception that it only records information specific to each thread, such as its programme number, stack pointer, registers, state, and so on. The run-time system controls the thread table. Similar to how the kernel maintains information about processes in the process table, when a thread is switched to ready status or blocked state, the information required to restart it is saved in the thread table. When a thread performs an action that might result in local blocking, such as waiting for another thread in the same process to do some job, it invokes a run-time system function. This technique determines if the thread needs to be blocked. If so, it saves the thread's registers (including its own) in the thread table, searches for an available thread to run from the table, and then reloads the machine registers with the new thread's stored values. The new thread immediately restarts when the stack pointer and programme counter have been changed. The full thread transition may be completed in a small number of instructions if the machine has one instruction to save all the registers and another to load them all. This method of thread switching is at least a factor of ten—possibly much more—faster than trapping to the kernel, which is a compelling case for user-level threads packages.

With processes, there is one significant distinction, however. When a thread calls thread yield, for instance, after it has done executing for the time being, the thread table itself may save the thread's data in the thread yield code. To choose a different thread to execute in further, it may then make a call to the thread scheduler. Invoking the scheduler and the operation that preserves the thread's state are essentially local functions, therefore doing so is significantly more effective than using a kernel function. No trap is required, no context switch is required, and the memory cache does not need to be cleared, and so on. This speeds up thread scheduling considerably. There are further benefits to user-level threads. They enable the use of individualized scheduling algorithms for each activity. It may be beneficial for certain applications, such as those that include a garbage-collector thread, to not have to worry about a thread being killed at an inopportune time. Also, they scale well since kernel threads always need some table and stack space, which might become an issue if there are a lot of threads. User-level threads packages, although having greater performance, have several serious issues. The first of them is the issue with the implementation of blocking system calls. Consider the scenario when a thread scans the keyboard before any keys are pressed. It is unacceptable to allow the thread to actually perform the system call since doing so would halt all of the threads. Allowing each thread to utilise blocking calls while preventing one blocked thread from interfering with the others was one of the key reasons for creating threads in the first place. It is difficult to see how this aim may be easily accomplished using blocking system calls. All of the system calls might be made nonblocking (a keyboard read, for instance, would simply return 0 bytes if no characters were already buffered), but doing so would need altering the operating system, which is undesirable. Moreover, one defence of user-level threads was that they could function under current operating systems. Also, several user applications will need to be modified in order to update the semantics of read. If it is feasible to predict whether a call will be blocked in advance, there is another option. Select is a system call that may be used in the majority of UNIX implementations to determine if a potential read will block. When this call is available, a new procedure that first executes a select call and then executes the read call only if it is safe may be used to replace the library method read (i.e., will not block). The call is not made if the read request will stall. Another thread is instead started. When the run-time system regains control the next time, it may reevaluate whether the read is now secure. There isn't much of an option except to use this method, which necessitates rewriting some of the system call library and is inefficient and ugly. A jacket or wrapper is the code that is placed around the system call to do the checks. For now, suffice enough to mention that computers may be configured such that not every application is running simultaneously in main memory. The operating system will retrieve the missing instruction (and its neighbours) from disc if the application calls or jumps to an instruction that is not in memory due to a page fault. We refer to this as a page fault. When the required instruction is being found and read in, the procedure is halted. Even if other threads may be permitted to execute, if one thread creates a page fault, the kernel, which is not even aware that there are threads, will automatically stop the whole process until the disc I/O is finished.

Another issue with user-level thread packages is that once one thread begins to run, none of the others will ever be able to do so until the first one willingly cedes the CPU. There are no clock interruptions inside a single process, making round-robin scheduling of processes impossible (taking turns). A thread will never have an opportunity to join the run-time system unless it voluntarily does so. Having the run-time system ask for a clock signal (interrupt) once per second to take control is one potential fix for the issue of threads running indefinitely, but this is also a primitive and difficult programming solution. Higher frequency periodic clock interruptions are not always feasible, and even then the overall overhead could be significant. Moreover, a thread may need a clock interrupt, which would prevent the run-time system from using the clock.

The fact that programmers often prefer threads in systems where the threads frequently block, such as, for example, a multithreaded Web server, is one more, and in reality the most damaging, argument against user-level threads. These threads make system calls all the time. The necessity for continuously making select system calls that check to determine whether read system calls are safe is removed by having the kernel switch threads if the previous one has stalled after a trap has happened to the kernel to execute the system call. What use does having threads serve in apps that are fundamentally CPU constrained and seldom block? There is nothing to be gained by utilising threads to play chess or compute the first n prime numbers, hence no one would really suggest doing any of those things in that manner.

Thread Implementation in the Kernel

Let's now examine allowing the kernel to monitor and control threads. In neither, a run-time system is required. Each process also lacks a thread table. The system's threads are instead all tracked by a thread table in the kernel. A thread may start a new thread or kill an existing thread by making a kernel call, which updates the kernel thread database and performs the creation or killing. Each thread's registers, status, and other information are stored in the kernel's thread table. The data is still retained in user space as it was with user-level threads, but now it is kept in the kernel (inside the run-time system). This information is a component of the process state, which classical kernels keep track of for their single-threaded processes. The conventional process table is also kept up to date by the kernel to keep track of processes.

Any call that might potentially stall a thread is implemented as a system call, incurring a cost that is noticeably higher than calling a run-time system method. The kernel has the choice to launch either a thread from the same process (if one is ready) or a thread from a separate process when a thread stalls. While using user-level threads, the run-time system continues to operate its own process's threads until the kernel frees the CPU for another task (or there are no ready threads left to run). Some systems recycle their threads in an effort to be more ecologically friendly since it is more expensive to create and delete threads in the kernel. While a thread's kernel data structures are left unaffected when it is killed, it is designated as non-runnable. An old thread is revived later when a new one has to be generated, reducing overhead. User-level threads can also recycle threads, but because there is less motivation to do so since the thread-management cost is so much lower.

There are no new, nonblocking system calls needed to support kernel threads. Moreover, the kernel may quickly determine if a process has any other runnable threads and, if so, run one of them while waiting for the necessary page to be fetched in from the disc if one of the threads generates a page fault. Its primary drawback is that because system calls are expensive, if thread operations (creation, termi- nation, etc.) are often performed, there will be a significant increase in overhead.

Hybrid Application

The benefits of user-level threads and kernel-level threads have been combined in a variety of ways. Use of kernel-level threads followed by multiplexing of some or all user-level threads is one method. Using this method, the kernel only manages and schedules the threads that are running at the kernel level. A number of user-level threads may be multiplexed on top of some of those threads. Similar to user-level threads in a process running on an operating system without multithreading, these user-level threads are generated, deleted, and scheduled. In this paradigm, a number of user-level threads alternately utilise each kernel-level thread.

Activations of the scheduler

Kernel threads are unquestionably slower than user-level threads, while being superior to them in certain significant aspects. Because of this, scientists have tried to find methods to make things better without sacrificing their positive attributes. The scheduler activation work aims to emulate kernel thread functionality with the improved performance and more flexibility often associated with user space threads packages. In particular, user threads shouldn't have to perform unique nonblocking system calls or determine in advance if a certain system call is safe to execute. But, if any other threads within the same process are available and ready to start, it should be allowed to do so when a thread stops due to a system call or a page fault. By minimising needless transfers between user and kernel space, efficiency is obtained. There is no need to engage the kernel, for example, if a thread is blocked while waiting for another thread to complete its task. This avoids the cost associated with the kernel-user transition. The synchronisation thread may be blocked, and a new one may be automatically scheduled, by the user-space run-time system.

By using scheduler activations, the kernel gives each process a certain number of virtual processors and leaves the (user-space) run-time system to allocate threads to processors. This approach is also applicable to multiprocessor systems, where the virtual CPUs may be actual CPUs. A process is originally given one virtual processor, but it has the ability to request more and return any extra processors it doesn't need. Moreover, the kernel has the ability to reassign allotted virtual processors to programmes that are in greater need of them. This scheme's fundamental premise is that when the kernel detects that a thread has stalled (for example, by executing a blocking system call or causing a page fault), it alerts the process' run-time system, giving the thread's number and a description of the incident as parameters on the stack. Similar to a signal in UNIX, the notification is initiated by the kernel activating the run-time system at a predefined beginning address. The upcall is the term for this process. The run-time system may reschedule its threads after it has been enabled, usually by blocking the currently running thread and selecting a different thread from the ready list, setting up its registers, and resuming it. The kernel sends another upcall to the run-time system to let it know when it discovers that the original thread can resume execution (for example, when the pipe it was attempting to read from now has data or when the page it faulted over has been loaded from disc). The blocked thread may either be instantly restarted by the run-time system or it can be added to the ready list and launched at a later time.

The interrupted CPU enters kernel mode when a hardware interrupt occurs while a user thread is active. As the interrupt handler completes, it returns the interrupted thread to its previous state if the interrupt was brought on by an event that was unimportant to the interrupted process, such as the conclusion of another process' I/O. Nevertheless, the interrupted thread is not resumed if the process is interested in it, such as when a page arrives that one of the process' threads needs. As a substitute, it is suspended, and the run-time system is launched on that virtual CPU with the interrupted thread's state still in effect. The run-time system will then have to choose whether to schedule the interrupted thread, the one that has just become ready, or a third option on that CPU. The essential dependency on upcalls, a notion that contradicts the structure inherent in every layered system, is a criticism against scheduler activations. Normal- ly, layer n provides certain services that layer n 1 may call on, but layer n is not always allowed to call layer n 1 procedures. This essential rule is broken by upcalls.

Pop-Up Topics

Distributed systems commonly find usage for threads. How incoming communications, such as service requests, are handled is a crucial illustration. The conventional method involves blocking a process or thread on a receive system call while it waits for an incoming message. It receives messages, unpacks them, looks over their contents, and then processes them.

But, an entirely alternative strategy is also feasible, in which the system creates a new thread as a message arrives to process the message. Each one is similar to the others and begins from scratch. This enables the speedy creation of such a thread. The incoming message is sent to the new thread to process. Using pop-up threads has the advantage of reducing the amount of time between when a message arrives and when processing begins.

Multithreading Single-Threaded Code

Several applications in use today were created for single-threaded operations. They are considerably difficult to multithread than they would first seem. We will just look at a handful of the hazards below.

Starting out, much like a process, the code of a thread often consists of many procedures. They could include parameters, global variables, and local variables. Local variables and parameters are trouble-free, but variables that are global to a thread but not to the whole programme are problematic. They are global variables in the sense that several threaded processes utilise them (like they would any global variable), but other threads should logically avoid interfering with them.

The fact that many library operations are not reentrant is the second issue with converting a single-threaded application to a multithreaded one. In other words, it was not intended for a process to receive a second call while the first call was still in progress. For instance, it may be coded to construct the message in a specified library buffer before triggering the kernel to transmit it over the network. What happens if a clock interrupt triggers a switch to a second thread, which immediately overwrites the buffer with its own message after one thread has composed its message. Similar to this, memory-allocation programmes like UNIX's malloc keep important tables on memory use, such as a linked list of memory chunks that are currently accessible. These lists could momentarily be unreliable and include pointers that go nowhere while malloc is actively updating them. A software crash may result from the usage of an erroneous pointer if a thread transition takes place while the tables are inconsistent and a new call is received from a different thread. Rewriting the whole library is necessary to adequately address all of these issues. It is a nontrivial task with a genuine chance of unnoticeable faults being introduced. Another option is to provide each operation a jacket that sets a bit to indicate that the library is currently in use. Every attempt to access a library procedure by another thread while a prior call is still running is prevented. This approach can be made to work, but it severely reduces the possibility of parallelism. Consider signals next. Although some are not, certain signals are logically thread-specific. For instance, it makes sense for the signal to reach the thread that called alert if the call was made. Nevertheless, when threads are wholly implemented in user space, the kernel is completely unaware of them and has no chance of directing the signal to the appropriate thread.

When many threads independently call alarm while a process can only have one alarm active at once, things get more complicated. Some signals are not thread-specific, such keyboard interrupt. This problem may occur if some threads execute user-written code while others execute standard library routines. These desires are obviously incompatible. In general, managing signals in a single-threaded environment is challenging enough. It does not make them any simpler to manage to switch to a multithreaded environment. The management of stacks is the last issue brought on by threads. When a process' stack overflows, the kernel will often just give it extra stack on its own. A process must have several stacks in addition to its many threads. The kernel cannot automatically increase any of these stacks upon a stack fault if it is unaware of all of them. In fact, it could not even be aware that a memory error is connected to the expansion of the stack of one or more threads. These issues are by no means insurmountable, but they do demonstrate that adding threads to an existing system without also substantially redesigning the system will not work. It may be necessary to, at the very least, rewrite libraries and modify the semantics of system calls. And for the limiting scenario of a process with just one thread, all of these things need to be done in a manner that is backwards compatible with already-existing applications. For further details about threads.

CHAPTER 10

TRANSCRIPTIVE COMMUNICATION

Dr. G.S.Vijaya, Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.vijayags@cms.ac.in

The necessity for communication between processes is common. The output of the first process, for instance, in a shell pipeline must be transferred to the second process, and so on. Thus, there is a need for interposes communication, ideally without the use of interruptions and in a systematic manner. We'll examine a few of the problems with this interposes communication, or IPC, in the sections that follow. There are three difficulties here, to put it simply. The first was mentioned earlier: how information might be sent from one procedure to another. The second involves preventing two or more processes from interfering with one another, such as when two processes in an airline reservation system are competing for the final available seat for a separate passenger. The third is about correct sequencing in the presence of dependencies: if process A creates data and process B prints them, B must wait till A has generated some data before beginning to print. In the part after that, we will start looking at all three of these problems. It's also crucial to note that threads are equally affected by two of these problems. As they share an address space, threads may easily transmit information in the first case (threads in different address spaces that need to communicate fall under the heading of communicating processes). The other two, appropriate sequencing and staying out of each other's hair, however, also apply to threads. The same issues exist, and the same fixes work. While the issue will be addressed in terms of processes in the sections that follow, the issues and fixes also apply to threads.

Conditions of Race

Some operating systems allow processes to share shared storage that each process may read from and write to. The placement of the shared memory has no bearing on the form of communication or the issues that develop. The shared storage may be in main memory (perhaps in a kernel data structure), or it may be a shared file. Let's now think about a simple yet typical scenario to illustrate how interprocess communication works in practise: a print queue. A process inputs the file name in a unique spooler directory when it wishes to print a certain file. The printer daemon is another process that checks on a regular basis to determine whether any files need to be printed. If so, it prints the files and then deletes their names from the directory. Assume that our spooler directory contains a very big number of slots, each one able to carry a file name, numbered 0, 1, 2,... Consider that there are two common variables as well: out, which links to the next file to be printed, and in, which refers to the directory's next open slot. It is possible to save these two variables in a two-word file that is accessible to all programmes. Slots 0 to 3 are vacant (the files have previously been printed), while slots 4 to 6 are filled, at that precise moment (with the names of files queued for printing). Processes A and B choose to queue a file for printing roughly concurrently. The following might occur in areas where Murphy's Law is relevant. Process A reads in the value 7 and puts it in the local variable next free slot. The CPU judges that process A has operated for enough time after a clock interrupt and switches to process B. Process B reads in as well and receives a 7. It also saves it in the next available local variable slot. Both processes currently believe that the next available time window is 7.

Crucial areas

Finding a means to prevent several processes from simultaneously accessing and writing to the shared data can help you avoid problems in this scenario as well as many others involving shared memory, shared files, and shared everything else. In other words, we need mutual exclusion, or a mechanism to prevent other processes from engaging in the same activity as one that is accessing a shared variable or file. Since process B began utilising one of the shared variables before process A had completed using it, the aforementioned problem occurred. Any operating system faces a significant design challenge when deciding which primitive operations to use to achieve mutual exclusion. In the sections that follow, we will go into great detail on this topic. It is also possible to state the issue of avoiding racial situations abstractly. A process may spend some time doing internal calculations and other tasks that may not result in race circumstances. However, a process may occasionally need to access shared memory or files or perform other crucial actions that may result in races. The critical region or critical section of the programme is where the shared memory is accessed. We could prevent races if we could set up the situation so that no two processes were ever in the same critical region at the same time. . While it prevents race problems, this criterion is insufficient to ensure that parallel processes work together appropriately and effectively when utilising shared data. To have a good solution, the following four requirements must be met: No assumptions about speeds or the number of CPUs are allowed, no process operating outside its critical zone is allowed to obstruct any other process, and no process should have to wait an eternity to reach its critical region. Abstractly speaking, that conduct. At time T1 in this instance, process a reaches its critical area. Process B attempts to reach its critical zone at time T2 but is unsuccessful since another process is already there and we only allow one process to be in a critical region at once. As a result, B is momentarily put on hold until time T3, when A exits its crucial zone and B may enter right away. At some point (T4), B departs, and we return to the initial state with no processes in their crucial areas.

Blocking Interruptions

The easiest option for a system with just one processor is to have each process disable all interrupts as soon as it enters its critical zone and enable them again as soon as it exits. No clock interrupts may occur while interrupts are deactivated. After all, the CPU only switches between processes as a consequence of clock or other interrupts, and if interrupts are disabled, the CPU won't move between processes. As a result, once interruptions are turned off, a process may update and inspect shared memory without worrying that another process would interfere. Given that it is dangerous to allow user processes the ability to disable interrupts, this strategy is typically undesirable. That may spell the death of the system if one of them did it and never switched them on again. Also, if the system has many CPUs (two or more), disabling interrupts only affects the CPU that carried out the disable command. The other ones can still access the shared memory and keep functioning. Contrarily, it is usually practical for the kernel to turn off interrupts for a short period of time while updating variables, particularly lists. Race circumstances could happen, for instance, if an interrupt occurs while the list of ready processes is inconsistent. The resulting conclusion is that, although interrupt disablement is often a practical operating system strategy, it is ineffective as a broad mutual exclusion mechanism for user programmes. The ability to achieve mutual exclusion by silencing interrupts—even inside the kernel—becomes less and less likely as multicore chips—even in low-end PCs—become more prevalent. On many devices, there are now two cores, and eight, sixteen, or even 32 cores are not far behind. Disabling one CPU's interrupts in a multicore (i.e., multiprocessor system) does not stop other CPUs from interfering with the tasks being carried out by the first CPU. Hence, more advanced strategies are required.

Variable Locks

Let's make another try at finding a software solution. Think about having a single, shared (lock) variable with a starting value of 0. A process initially checks the lock before attempting to access its crucial region. If the lock is set to 1, the process moves into the crucial zone. The method just waits until the lock reaches 0 if it is already at 1. As a result, a value of 0 indicates that no process is in its critical zone, while a value of 1 indicates that a process is in its critical region. Sadly, this concept also has the exact same fatal problem as the spooler directory. Let's say a process reads the lock and determines it to be 0. Prior to it setting the lock to 1, another scheduled operation starts, sets the lock to two processes will be in their crucial zones simultaneously when the first process restarts since it will also set the lock to 1. You may believe that by first reading the lock value out and then double-checking it just before saving it, we can get around this issue, but that is not the case. If the second process updates the lock right after the first process has done its second check, the race will now happen.

Alternating strictly

A third method of solving the mutual exclusion issue. This programme fragment is written in C, much like the majority of the others in this book. The reason C was selected for this project is since genuine operating systems are almost usually developed in C (or sometimes C++), but seldom in Java, Python, or Haskell. C has the strength, effectiveness, and predictability that are essential for creating operating systems. Since it could run out of storage at a crucial time and need to use the garbage collector to free up memory at an inconvenient time, Java is an example of a programme that is unpredictable. Because C doesn't have a trash collection, this cannot occur. C, C++, Java, and four additional languages are compared quantitatively.

Initially set to 0, the integer variable turn maintains track of who is next to enter the crucial zone and read from or write to the shared memory. Process 0 first examines turn, determines that it is 0, and then moves into its crucial zone finds it to be 0 and therefore sits in a close loop, checking the turn to see whether it changes to 1. Busy waiting is the process of repeatedly checking a variable until a value arrives. Typically, it should be avoided since it uses much CPU time. Busy waiting is only used when there is a solid anticipation that the wait will be brief. Using busy waiting, a lock is referred to as a spin lock. Turn is set to 1 when process 0 exits the crucial zone to make room for process 1 to enter. With turn set to 0, let's assume that process 1 swiftly completes its critical area, leaving both processes in their noncritical regions. Process 0 now swiftly completes its whole loop, leaving its critical zone, and setting turn to 1. Both processes are now operating in their noncritical zones as the turn counter reads 1. Process 0 completes its noncritical section and abruptly returns to the start of its loop. Sadly, since it is turn 1 and process 1 is occupied with its noncritical zone, it is not allowed to access its critical region at this time. While loop, it remains stationary until process 1 sets turn to 0. To put it another way, it is not a good idea to switch off when one activity is much slower than the other. The above-mentioned requirement 3 is broken in this instance: A process that is not in process 0's vital zone is obstructing it. Returning to the spooler directory from before, if we now link the crucial area to reading and writing the spooler directory, process 0 would be prevented from printing further files since process 1 was engaged in another task. The two processes must really rigorously alternate when entering their key zones, such as when spooling files, in order for this strategy to work. Both of them would not be allowed to spool two in a row. Although this approach does prevent all races, condition 3 is broken, making it a weak option for a solution.

Peterson's Approach

The first person to provide a software solution to the mutual exclusion issue that does not need rigorous alternation was a Dutch mathematician named T. Dekker. He did this by fusing the concept of taking turns with the concepts of lock variables and warning variables to talk about Dekker's algorithm. Each process calls enter area with its own process number, 0 or 1, as an argument before utilising the common variables (i.e., before entering its crucial region). If necessary, this call will make it wait until it is safe to enter. The process calls exit area to signal that it is completed and to enable the other process to enter, if it so chooses, when it has finished working with the shared variables. Let's test the efficacy of this solution. Both processes are not initially in their crucial regions. Process 0 incoming calls at this time. It sets turn to 0 and its array element to show interest. Process 1 instantly returns after entering the area since it has no interest. Process 0 can only use leave region to leave the crucial zone, therefore if process 1 now calls enter region, it will hang there until interested [0] changes to FALSE. Now imagine that both processes call enter area practically at the same time. Each will keep track of their own process number. It doesn't matter which store is finished last; the first one gets overwritten and lost. Consider that turn 1 is because process 1 saves last. When the while statement is reached by both processes, process 0 runs it 0 times and enters its crucial zone. Process 0 must depart its critical area before process 1 may begin its loop and enter its critical region.

The TSL Directive

Let's now examine a suggestion that needs some assistance from the hardware. There are certain computers that feature instructions like (Test and Set Lock) that operate as follows, particularly those that were built with multiple processors in mind. It writes a nonzero value at the memory address lock after reading the contents of the memory word lock into register RX. No other processor may access the memory word until the instruction is complete, ensuring that the actions of reading the word and saving into it remain indivisible. The memory bus is locked by the CPU that is performing the TSL instruction, preventing other CPUs from accessing memory until it is finished. It's crucial to understand that locking the memory bus differs significantly from turning off interrupts. A second processor on the bus may still access the memory word in between a read and a write when interrupts are off and a memory word is read, followed by a write. In reality, processor 2 is completely unaffected by processor 1's disablement of interrupts. Locking the bus is the only means to prevent processor 2 from accessing memory before processor 1 is completed, which necessitates a unique piece of hardware (basically, a bus line asserting that the bus is locked and not avail- able to processors other than the one that locked it). We will manage access to shared memory by using the shared variable lock to employ the TSL instruction. Any process may use the TSL instruction to change lock from 0 to 1 and then read or write to shared memory. After it is finished, the process uses a standard move command to reset lock to 0. This is a demonstration of a fictional (though typical) assembly language four-in-traction subroutine. The first instruction sets lock to 1 after copying lock's previous value to the register. The previous value is then contrasted with 0. If it is nonzero, the lock has already been established, therefore the programme just starts again and checks it once more. It will eventually reach 0 (when the process that is now in its critical zone has finished with it), at which point the function returns with the lock set. The lock can be easily cleared. In lock, the application only saves a 0. There are no unique synchronisation instructions required. The critical-region issue now has a straightforward remedy. A process first uses enter region to prepare for entering its crucial area. Enter region performs busy waiting until the lock becomes available, then it obtains the lock and exits. The process calls exit region after leaving the crucial area, which stores a 0 in lock. For the approach to function, as with other solutions based on crucial areas, the processes must call enter region and depart region at the proper moments. The mutual exclusion will not work if one process cheats. In other words, vital areas only function when the processes are in sync. TSL has a substitute instruction called XCHG that swaps the contents of two places atomically, such as a register and a memory word, which does the same thing as TSL. The XCHG instruction is used for low-level synchronisation by all Intel x86 Processors.

Awaken and Sleep

The flaw with Peterson's method and those using TSL or XCHG is that they both call for busy waiting even if they are both right. These solutions essentially check to determine whether a process is permitted to access its crucial zone before allowing it to do so. If not, the procedure just waits in a close loop until it is. This method not only wastes CPU time, but it could also provide unexpected results. Think of a computer with two processes: L has a low priority and H has a high priority. Because of the scheduling policies, H is always executed when it reaches the ready state. As L reaches its critical zone, H eventually becomes ready to execute (for example, after an I/O operation is finished). H now starts to wait when busy, but because L is never L is never given the opportunity to escape its critical zone while H is operating, therefore H loops indefinitely. The priority inversion dilemma is another name for this circumstance. Let's look at some primitives for interprocess communication that block when denied access to their crucial areas rather than wasting CPU time. The pair of sleep and waking is among the simplest. A system call called "sleep" causes the calling process to stall, or be paused, until another process wakes it up. The process to be woken is the only variable in the wake-up call. Alternately, the memory location used to match up sleeps and wakeups is the only parameter shared by both sleep and wakeup.

The Producer-Consumer Problem

Let's have a look at the producer-consumer situation to see how these primitives may be applied (also known as the bounded-buffer problem). There is a shared, fixed-size buffer between two processes. Information is inputted into the buffer by one of them, the producer, and removed by the other, the consumer. (It is also theoretically possible to expand the issue to include m producers and n consumers, but for simplicity's sake we will just discuss the scenario of one producer and one consumer.) When the producer wishes to add a new item to the buffer but it is already full, trouble might result. The answer is to put the producer to sleep and have them wake up after they've taken one or more things away. Similar to this, if a consumer wishes to take anything from the buffer but notices that it is empty, it goes to sleep until the producer fills the buffer with something to wake it up. This method seems straightforward enough, however it causes the same types of race situations that the spooler directory did previously. We need a variable called count to record the number of items in the buffer. The producer's code will first check to see whether count is N if the buffer's maximum item capacity is N. The producer will add a new item and increase the count if it is, else, the producer will go to sleep. Similar code is used by the consumer: first check the count to determine whether it equals 0. Go to sleep if it is; if it is not, remove something and decrease the counter. Each procedure also determines if the other needs to be woken and, if so, wakes it up both the producer's and the consumer's code. We will demonstrate how to express system calls like sleep and awaken in C as calls to library procedures. These are not a part of the default C library, although any system that genuinely had these system functions would probably make them accessible. The accounting for adding things to the buffer and removing items from the buffer is handled by the operations insert item and remove item, which are not shown. Let's return to the racing situation at this time. As there are no restrictions on access to count, it is possible. Thus, it's possible that the following scenario will develop. The consumer has just read the count to determine whether it is zero and the buffer is empty. The scheduler determines right away to temporarily pause operating the consumer and begin running the producer. The creator increases the count after adding an item to the buffer and sees that it has changed to 1. The producer calls awaken to rouse the consumer up on the theory that because count was only 0, the customer must be dozing. Sadly, the wakeup signal is missing since the consumer is not yet conceptually asleep. When the consumer starts up again, it will check the value of the count it previously read, discover that it is zero, and then shut down. The producer will eventually finish filling the buffer and go to bed. Both will slumber endlessly. The main issue here is that a wakeup signal delivered to a process that isn't (yet) sleeping gets misplaced. It wouldn't be lost, and everything would function. The rules may be changed to include a wakeup waiting bit as a simple workaround. This bit is set whenever a wakeup is delivered to an active process. The wakeup waiting part will be switched off later when the process attempts to go to sleep, but it won't, thus the process will remain awake. For storing wakeup signals, the wakeup waiting bit functions as a piggy bank. At each loop iteration, the consumer clears the wakeup waiting bit. Even if the wakeup waiting bit saves the day in this straightforward example, it is trivial to create instances involving three or more processes where just one wakeup waiting bit is enough. We could create a new patch and include a second wakeup waiting bit, or even 8 or 32 of them, but the issue would still exist in theory.

Semaphores When E. W. Dijkstra (1965) proposed utilising an integer variable to tally the number of wakeups stored for later use, this was the scenario. He created a brand-new form of variable in his proposal that he dubbed a semaphore. A semaphore might be set to a positive value if there were any pending wakeups or a value of 0 if none were saved. Dijkstra suggested semaphores perform two actions, currently known as down and up (generalizations of sleep and wakeup, respectively). A semaphore's down action verifies if the value is larger than zero. If so, it just keeps going while decreasing the value (i.e., using up one stored wakeup). If the value is 0, the procedure is suspended without temporarily finishing the down. A single, indivisible atomic operation is performed to check the value, change it, and perhaps go to sleep. No other process is permitted to access the semaphore once a semaphore action has begun until it is finished or blocked. To solve synchronisation issues and prevent race circumstances, this atomicity is a must. Atomic actions, in which a collection of connected processes is either carried out whole or not at all, are also crucial in many other areas of computer science. The semaphore addressed's value is increased by the up operation. The system chooses one process (for example, at random) and permits it to finish its down if one or more processes were asleep on that semaphore and unable to complete an earlier down operation. As a result, following an up, a semaphore with processes sleeping on it will still remain 0, but one less process will be doing so. Moreover, the act of increasing the semaphore and arousing one process occurs simultaneously. No process ever stops an up from happening, just as no process stopped a wakeup from happening in the older paradigm. As a side note, in Dijkstra's original article, he referred to down and up, respectively, as P and V. We shall use the phrases down and up in their place since Proberen (try) and Verhogen (raise, make higher) have no mnemonic relevance to individuals who do not know Dutch and very little significance to those who do. They were were presented in the computer language Algol 68.

How to Fix the Producer-Consumer Issue Semaphore use

It is crucial that they be implemented in an indivisible manner for them to function properly. Up and down are often implemented as system calls, with the operating system momentarily turning off all interrupts while it tests the semaphore, updates it, and, if required, puts the process to sleep. Disabling interruptions has no negative effects since each of these activities only requires a few instructions. Each semaphore should be secured by a lock variable if many CPUs are being utilised, and the TSL or XCHG instructions should be used to ensure that only one CPU at a time checks the semaphore. Make sure you get the difference between the producer and consumer being occupied waiting for the other to empty or fill the buffer and utilising TSL or XCHG to prevent several CPUs from accessing the semaphore at the same time. Although the producer or consumer may take indefinitely long, the semaphore action will only take a few microseconds.

This solution makes use of three semaphores: a full semaphore for counting the number of filled slots, an empty semaphore for counting the number of empty slots, and a mutex semaphore to prevent concurrent access to the buffer by the producer and consumer. Empty is originally equal to the number of slots in the buffer, full is initially 0, and mutex is initially 1. Binary semaphores are semaphores that are initialised to 1 and utilised by two or more processes to make sure that only one of them may approach their critical zone simultaneously. Mutual exclusion is ensured if each process performs a down immediately before entering its crucial zone and an up shortly after leaving it. Let's revisit the interrupt sequence in Fig. 2-5 now that we have a reliable interprocess communication primitive available. Having a semaphore, initially set to 0, connected with each I/O device in a system that uses semaphores is the obvious approach to conceal interruptions. The management process performs a down on the related semaphore shortly after initiating an I/O device, instantly blocking. When an interrupt occurs, the interrupt handler performs an up on the corresponding semaphore to prepare the process to resume execution. In this architecture, step 5 in Fig. 2-5 entails performing an up on the device's semaphore in order to enable the scheduler to launch the device manager in step 6. Of However, the scheduler could decide to execute a process that is even more crucial first if numerous processes are now available. Semaphores have been employed by mankind in two distinct ways. This distinction is significant enough to be mentioned explicitly. Mutual exclusion is accomplished via the mutex semaphore. It is made to ensure that only one process at a time will read from or write to the buffer and the related variables. To avoid anarchy, this mutual exclusion is necessary. In the part after this one, we will look at mutual exclusion and how to accomplish it. Semaphores are also used for synchronisation. It is necessary to use the full and empty semaphores to ensure that certain event sequences happen or do not take place. In this instance, they make sure that the consumer and producer both end their operations when the buffer is filled. In contrast to mutual exclusion, this usage.

Mutexes

A mutex, a condensed form of the semaphore, is occasionally used when the capacity to count is not required. Just managing mutual exclusion to a shared resource or piece of code is a valid use for mutexes. They are particularly helpful in thread packages that are totally implemented in user space since they are simple and effective to construct. A shared variable known as a mutex has two possible states: unlocked and locked. In order to represent it, just 1 bit is needed; nevertheless, in reality, an integer is often used, with 0 denoting unlocked and all other values denoting locked. Mutexes are utilised together with two methods. A thread (or process) requests for mutex lock when it requires access to a crucial area. The call succeeds and the caller thread is free to reach the critical zone if the mutex is presently unlocked (indicating that the critical region is available). The calling thread is halted until the thread in the crucial zone completes its work and invokes mutex unlock, which happens if the mutex is already locked. If many threads are stopped by the mutex, one of them is randomly selected and permitted to get the lock. It constantly tests the lock when entry region is unable to enter the crucial area (busy waiting). The timer eventually expires and another task is set to execute. The process that is holding the lock eventually completes its execution and releases it. The problem is different with (user) threads since there is no clock to terminate threads that have ran too long. Since it never permits another thread to execute and release the lock, a thread that attempts to acquire a lock through busy waiting would loop endlessly and never succeed. That is where the distinction between a mutex lock and an entry region comes into play. The latter invokes thread surrender to offer the CPU to another thread when it is unable to get a lock. There is thus no busy waiting. The lock is tested once again when the thread resumes execution. Thread yield is very quick since all it does is make a call to the user space thread scheduler. As a result, neither mutex locking nor unlocking calls the kernel. They allow user-level threads to fully synchronise in user space with procedures that only call a small number of instructions. The above-mentioned mutex system is a basic collection of calls. There is always a need for greater functionality in software, and synchronisation primitives are no different. For instance, a thread package may sometimes include a call mutex trylock that does not stall and either obtains the lock or provides an error code in the event of failure. If there are other options than waiting, this call allows the thread to choose what to do next. Up to this point, we have skirted over a nuance that is worth at least making apparent. As all the threads run in the same address space, there is no issue with multiple threads having access to the same mutex when using a user-space threads package. Nevertheless, there is an implicit assumption that many processes have access to at least some shared memory, even if it is only one word, with the majority of the older methods, such as Peterson's algorithm and semaphores. How can processes share a shared buffer or semaphores in Peterson's method if they have separate address spaces, as we have repeatedly stated. Two options exist. First off, the kernel may keep certain shared data structures, like semaphores, and only allow system functions to access them. The issue is resolved using this strategy. In addition, the majority of contemporary operating systems, such as Windows and UNIX, allow processes to share a part of their address space with other processes. Buffers and other data structures may be transferred in this fashion. In the worst case scenario, if there is no other option, a shared file may be utilised. The difference between processes and threads gets somewhat muddled but still exists when two or more processes share most or all of their address spaces. Although the threads inside a single process share open files, alarm timers, and other per-process attributes, two processes that share an address space do not. Yet because the kernel is heavily engaged in managing them, it is always true that numerous processes sharing a similar address space will never be as efficient as user-level threads.

Futexes

Efficiency in synchronisation and locking is crucial for performance as parallelism increases. Spin locks consume CPU cycles if the wait is prolonged but are quick if it is. So, blocking the process and allowing the kernel to unblock it only when the lock is free is more effective when there is a lot of contention. However, this has the opposite issue: although it performs well in situations of high contention, frequent switching to the kernel is costly in situations when there is little contention to begin with. Even worse, it could be difficult to forecast how much lock contention would be there. Futex, sometimes referred to as a "rapid user space mutex," is one intriguing option that attempts to blend the best of both worlds. Similar to a mutex, a futex is a Linux feature that supports basic locking but stays out of the kernel until absolutely necessary. Performance is much enhanced since it is costly to move to and from the kernel. A futex is made up of two components: a user library and a kernel service. Several processes may wait on a lock simultaneously thanks to the wait queue provided by the kernel service. Until the kernel specifically unblocks them, they won't operate. An (cost) system call is necessary to add a process to the wait queue, hence this should be avoided. Hence, the futex operates entirely in user space when there is no conflict. In particular, the processes share a shared lock variable, which is a fancy term for the lock, a 32-bit aligned integer. Let's say the lock's initial value is 1, which we take to signify that it is unlocked. The lock is taken by a thread using an atomic "decrement and test" (atomic functions in Linux consist of inline assembly wrapped in C functions and are defined in header files). The thread then examines the outcome to determine if the lock was released or not. Everything will be OK if it wasn't locked, and our thread was successful in grabbing the lock. Nevertheless, if another thread is holding the lock, our thread will have to wait. In such situation, the futex library utilises a system call to place the thread on the kernel's wait queue rather than spinning it. Because the thread was stopped in the first place, maybe the cost of the transition to the kernel is now justified. After a thread has finished using a lock, it releases it using an atomic "increment and test" and examines the outcome to determine whether any processes are still waiting on the kernel wait queue. If so, it will inform the kernel that one or more of these processes may be unblocked. The kernel is not at all engaged if there is no conflict.

Pthreads' mutexes

A variety of synchronisation functions are available in Pthreads. Each crucial zone is protected by a mutex variable in the basic mechanism, which may be locked or freed. Before entering a crucial zone, a thread attempts to lock the relevant mutex. The lock is atomically set and the thread may enter right away if the mutex is unlocked, blocking entry by other threads. The calling thread is stopped until the mutex is unlocked if it has already been locked. When a mutex is unlocked while many threads are waiting on it, only one of them is permitted to proceed and relock it. These are not mandatory locks. Programmers are responsible for ensuring that threads utilise them properly. The main mutex-related calls. Mutexes can be built and destroyed, as intended. Pthread mutex init and pthread mutex destroy are the calls used to carry out these tasks. Moreover, they may be locked using the pthread mutex lock, which attempts to acquire the lock and fails if it is already locked. If a mutex is already blocked, it may also be attempted to lock but fail with an error code rather than blocking. That is a pthread mutex trylock call. In the event that busy waiting is ever required, this call enables a thread to do so successfully. Lastly, if one or more threads are waiting on it, pthread mutex unlock releases precisely one thread when unlocking a mutex. While mutexes may also contain properties, they are only utilised in certain circumstances. Pthreads provides condition variables as a second synchronisation method in addition to mutexes. For granting or preventing access to a crucial sector, mutexes are useful. Threads may get stuck if a condition is not fulfilled thanks to condition variables. Nearly often, the two approaches are combined. Let's now take a closer look at how threads, mutexes, and condition variables interact with one another. Consider the producer-consumer situation once again as a straightforward illustration: one thread adds items to a buffer, while another thread removes them. The producer must block until a free slot is available if it discovers that the buffer is empty of any more empty spaces. Mutexes allow for the check to be completed atomically and without the intervention of other threads, but once the producer has determined that the buffer is full, they need a mechanism to block and be woken up later. This is made possible using conditional variables.

The crucial calls concerned conditional variables. There are calls to construct and remove condition variables, as you would anticipate. There are many calls for controlling them, and they may have characteristics (not shown). Pthread cond wait and pthread cond signal are the two main operations on condition variables. Unless another thread indicates it, the former prevents the caller thread from continuing (using the latter call). Of fact, the waiting and signalling protocol does not include the causes of blocking and waiting. The blocking thread often awaits the signalling thread's completion of a task, the release of a resource, or the performance of another action. The blocked thread may then proceed only after that. The waiting and blocking may be done atomically thanks to the condition variables. When many threads are possibly stuck and waiting for the same signal, the pthread cond broadcast call is utilised. Mutexes and condition variables are usually used in tandem. When a thread can't obtain what it needs, the pattern is for it to lock a mutex and then wait on a conditional variable. It will eventually get a signal from another thread and may proceed. The mutex it is holding is atomically released by the pthread cond wait call. The mutex is one of the parameters for this reason. It is also important to remember that condition variables have no memory, unlike semaphores. A signal is lost if it is sent to a condition variable on which no thread is waiting. Programmers must take care to avoid signal loss.

Shows a very basic producer-consumer dilemma with a single buffer as an illustration of the usage of mutexes and condition variables. After filling the buffer, the producer must wait until the consumer uses it all before moving on to the next item. Similar to when the customer removes an item, the manufacturer must wait until a replacement has been made. Even though it's quite basic, this sample shows the fundamental workings. A thread may be woken by a UNIX signal or for other reasons, therefore the statement that puts it to sleep should always verify the condition to make sure it is met before moving on.

Monitors

Interprocess communication seems simple with semaphores and mutexes, right? Ignore it. Before to adding or deleting items from the buffer pay particular attention to the sequence of the downs. Consider reversing the sequence of the two downs in the producer's function such that mutex was decremented before empty rather than after it. When the buffer is entirely filled, the producer will block and the mutex will be set to 0. As a result, the consumer would conduct a down on the mutex, which was now set to 0, and block the next time it attempted to access the buffer. No additional work could ever be completed since both processes would be halted indefinitely. A dead-lock is the unpleasant circumstance that exists here. This issue is brought up to highlight how cautious you must be while utilising semaphores. Everything grinds to a screeching end with only one little mistake. Since the mistakes involve race conditions, deadlocks, and other types of unexpected and irreproducible behaviour, it is like programming in assembly language but worse. Brinch Hansen and Hoare (1973) developed a higher-level synchronisation primitive called a monitor to make it simpler to construct proper algorithms. The following details how little their proposals varied. A monitor is a grouping of procedures, variables, and data structures in a particular sort of module or package. The procedures in a monitor may be called anytime a process desires, however procedures declared outside of a monitor cannot directly access the monitor's internal data structures. A computer monitor created in the fictional language Pidgin Pascal. Since monitors are a language notion and C doesn't have them, C cannot be utilised in this situation.

One crucial feature of monitors that makes them helpful for accomplishing mutual exclusion is the ability for just one process to be active in a monitor at any one time. As monitor procedures are built into programming languages, the compiler is aware of their unique nature and may treat calls to them differently from regular procedure calls. In most cases, when a process runs a monitor procedure, the program's first few instructions check to determine whether any other processes are already running within the monitor. If so, until the other process has left the monitor, the calling process will be put on hold. The calling process may enter if no other process is utilising the monitor. Mutual exclusion on monitor entries is left to the compiler, although a typical approach is to employ a mutex or a binary semaphore. It is considerably less probable that anything will go wrong since the compiler, and not the programmer, is setting up for the mutual exclusion. In any case, the compiler's arrangements for mutual exclusion are not required to be known to the person building the monitor. Knowing that no two processes will ever execute their crucial regions concurrently by converting all critical regions into monitor procedures is sufficient. As we have shown above, monitors do provide a simple method for achieving mutual exclusion, but that is insufficient. We also need a method for processes to halt when they are unable to continue. It is simple enough to include all of the checks for buffer-full and buffer-empty in monitor methods in the producer-consumer issue, but it is more difficult to determine how the producer should block when it detects a full buffer. The introduction of condition variables and the two operations wait and signal on them are the answer. A monitor process performs a wait on some condition variable, such as full, when it realises it cannot proceed (for example, when the producer finds the buffer full). The calling procedure becomes stuck as a result of this action. Also, it permits another process to access the monitor now that was before forbidden. Conditions variables and these procedures were previously seen in the context of Pthreads. By sending a signal to the condition variable that its mate is awaiting, for instance, the consumer in this other process may wake up its sleeping companion. We need a rule that specifies what occurs following a signal in order to prevent having two active processes in the monitor at once. Hoare suggested stopping the other process and allowing the freshly awakened one to operate. Brinch Hansen suggested solving the issue by mandating that every operation that generates a signal must leave the display right away. Thus, only the last statement in a monitor method may contain a signal statement. We'll adopt Brinch Hansen's suggestion since it makes sense philosophically and practically. Just one process is restarted when a signal is sent on a condition variable that many processes are waiting on, according to the system scheduler.

In addition, there is a third option that neither Hoare nor Brinch Hansen have mentioned. This is done so that the signaler may keep operating and the waiting procedure can begin after the signaler has left the display. Variables used in conditions are not counters. Unlike semaphores, they do not store signals for subsequent use. The signal is thus permanently lost if a condition variable is signalled and no one is waiting for it. To put it another way, the signal must arrive before the wait. The application is made considerably easier by this rule. Because it is simple to monitor each process's status using variables, it is not an issue in practise. By examining the variables, a process that would ordinarily perform a signal may determine that this operation is not required. An outline of the monitor producer-consumer issue. Pidgin Pascal, a language of the imagination. The benefit of utilising Pidgin Pascal in this situation is that it is straightforward and adheres completely to the Hoare/Brinch Hansen paradigm. You could be considering how sleep and awakening, which we saw previously have catastrophic race conditions, resemble the operations wait and signal. They are extremely similar, but there is one key distinction between them: wakeup and sleep failed because one process attempted to go to sleep while the other attempted to wake it up. Monitors prevent it from happening. The producer within a monitor process, for example, could see that the buffer is full and be able to finish the wait operation without worrying that the scheduler might move to the consumer just before the wait is finished thanks to the automatic mutual exclusion on monitor procedures. Once the wait is over and the producer has been designated as unrunnable, the customer won't even be permitted to enter the monitor.

Although being an invented language, Pidgin Pascal is supported by certain genuine programming languages, but not necessarily in the manner that Hoare and Brinch Had intended. Java is one of such languages. User-level threads and class-based grouping of methods (procedures) are features of the object-oriented language Java. Java ensures that once a thread has begun running a method after including the term synchronised in its declaration, no other thread will be permitted to begin executing any additional synchronised methods of that object. There are no assurances for interleaving without synchronization.

The producer and consumer threads are similar to their counterparts in all of our earlier examples in terms of functionality. The producer generates data and deposits it into the shared buffer using an endless loop. The consumer has an endless loop that it uses to extract data from the shared buffer and perform enjoyable operations on it. The class our monitor, which has the buffer, the administration variables, and two synchronized methods, is what makes this programme interesting. It is safe to update the variables and the buffer without concern about race situations while the producer is active within insert since it is certain that the consumer cannot be active inside removal. The number of objects in the buff- er is kept track of by the variable count. It may have any value between 0 and N 1, inclusive. The index of the buffer slot where the next item is to be retrieved is included in the variable lo. In a similar vein, hi is the index of the buffer slot that will hold the next item. It is acceptable for lo to be greater than hi, which indicates that there are either 0 or N items in the buffer. Which situation is true is determined by count's value. The absence of condition variables in Java makes synchronized methods in Java fundamentally different from traditional monitors. Instead, it provides two procedures, wait and notify, which are comparable to sleep and awake but are not prone to race situations when used within synchronized methods. The purpose of the code around it is to theoretically interrupt the method wait. Java mandates the explicit handling of exceptions. Let's pretend for the sake of this discussion that sleeping is the only option. Monitors significantly reduce the likelihood of errors in parallel programming compared to semaphores by automating the mutual exclusion of crucial sections. Yet, they also have certain shortcomings. Just by chance, unlike the other examples in this book, our two instances of monitors were written in Pidgin Pascal rather than C. Monitors are a notion found in computer languages, as we previously said. The compiler must be able to identify them and set up their mutual exclusion in some way. It is absurd to expect the compilers of C, Pascal, and the majority of other languages to implement any mutual exclusion restrictions since these languages lack monitors.

Even though these same languages lack semaphores, adding them is simple: all you need to do is add two quick assembly-code functions to the library that execute the up and down system calls. The compilers don't even need to be aware of their existence. Of course, operating systems must be aware of semaphores, but at least with a semaphore-based operating system, user applications may still be created in C or C++ (or even assembly language if you are masochis- tic enough). You need a language that supports monitors when working with them. The fact that semaphores and monitors were created to solve the mutual exclusion problem on one or more CPUs that all have access to the same memory presents another issue with them. Races may be prevented by storing the semaphores in shared memory and safeguarding those using TSL or XCHG instructions. These primitives lose their usefulness when we switch to a distributed system made up of many CPUs, each with its own private memory and linked by a local area network. The conclusion is that monitors are not useful outside of a few computer languages and that semaphores are too low level. Moreover, none of the primitives provide machine-to-machine communication. Further action is required.

CHAPTER 11

MESSAGE-PASSING SYSTEM

Dr. S.Yogananthan, Adjunct Faculty,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.s_yogananthan@cms.ac.in

The transmission of messages is that something else. This interposes communication technique makes use of two primitives, send and receive, which, unlike monitors but like semaphores, are system calls. As a result, they are simple to include into library operations. For example, send (destination, &message) and receive (source, &message) transmit messages to specific destinations and respectively receive messages from specific sources (or from ANY, if the receiver does not care). In the absence of a message, the receiver may block until one does. It may also instantly return with an error code as an alternative.

Problems with Message-Passing System Design

Semaphores and monitors do not have the same difficulties or design challenges that message-passing systems have, particularly if the communication processes are running on separate computers linked by a network. For instance, the network may lose messages. The sender and receiver might agree that as soon as a message is received, the receiver will send back a certain acknowledgment message in order to prevent lost communications. The message is sent again if the sender does not get the acknowledgment within a certain window of time.

Now examine what would transpire if the communication was successfully delivered but the acknowledgment to the sender was misplaced. The message will be sent again, ensuring that the recipient receives it twice. The receiver must be able to differentiate between a new message and a retransmission of an older message. The typical solution to this issue is to include successive sequence numbers in each original message. Receivers are aware that a message is a duplicate and might be disregarded if it has the same sequence number as a prior message. A significant portion of the research of computer networks is how to communicate effectively in the face of inconsistent message conveyance.

So that the process mentioned in a send or receive call is clear, message systems must also address the issue of how processes are named. Authentication is a problem in messaging systems as well. When the transmitter and receiver are on the same computer, there are additional design considerations that are crucial. Performance is one of them. A semaphore action or accessing a monitor are always faster than copying messages from one process to another. Making message forwarding efficient has required a great deal of effort.

The Message Passing Producer-Consumer Issue

Let's now examine how message forwarding and no shared memory may be used to address the producer-consumer dilemma. Suppose that all messages have the same size and that the operating system automatically buffers transmitted but unreceived messages. This technique uses N messages altogether, which is comparable to the N slots in a shared memory buffer. The customer first sends the producer N empty messages. Every time the producer has anything for the consumer to receive, it takes an empty message and replies with a full one. As a result, the total quantity of messages in the system stays constant over time, allowing them to be stored in a predetermined amount of memory. All messages will be full and waiting for the consumer if the producer operates more quickly than the consumer; in this case, the producer will be blocked and waiting for an empty message to return. The opposite will occur if the consumer works more quickly: all messages will be empty while they wait for the producer to fill them up, blocking the consumer as they wait for a complete message. Using message passing, several variations are available. Let's start by examining the manner in which communications are addressed. Messages may be addressed to processes by giving each one a distinct address. Creating a new data structure called a mailbox is an alternative method. A mailbox is a location where a certain amount of messages may be buffered, usually one that is defined when the mailbox is formed. By using mailboxes, the send and receive calls' address arguments are mailboxes rather than processes. In order to make place for a new message, a procedure that attempts to transmit to a mailbox that is already full is halted. Both the producer and the consumer would build mailboxes big enough to house N messages for the producer-consumer issue. The consumer would send empty messages to the producer's mailbox, and the producer would respond by sending messages with real data to the consumer's inbox. Using mailboxes makes the buffering mechanism obvious: messages transmitted to the destination process but not yet accepted are stored in the destination mailbox.

The opposite of having mailboxes is to completely disable buffering. While using this method, the sending process is halted if the send occurs before receive. At that point, the message may be copied directly from the sender to the receiver without buffering. The receiver is also blocked until a send occurs if the receive is completed first. This tactic is often referred to as a rendezvous. As the transmitter and receiver must operate in lockstep, it is less flexible than a buffered message method but simpler to construct.

Barriers

Instead of two-process producer-consumer scenarios, our final synchronisation method is designed for groups of processes. There are certain applications that are divided into phases, and one of the rules is that no process may go on to the next phase unless all other processes are prepared to do so. A barrier might be added at the conclusion of each step to encourage this behaviour. As a process crosses the barrier, it is halted until every process has done so. As a result, many processes may synchronise.

Consider a typical relaxation issue in physics or engineering as an example of a problem needing barriers. Usually, there is a matrix with certain beginning values. The numbers might reflect the temperatures at different locations on a metal sheet. Calculating how long it takes for the impact of a flame set in one corner of the sheet to spread across may be the concept. To create the second version of the matrix, a transformation is done to the original matrix starting with the present values, for instance by using the laws of thermodynamics to determine what the final temperatures are, T. Then, when the sheet warms up, the procedure is repeated several times to provide the temperatures at the sample spots as a function of time. Each matrix created by the method is for a specific moment in time and is produced in a series across time. Consider a situation where the matrix is exceedingly huge (say, 1 million by 1 million), necessitating the use of parallel processes (perhaps on a multiprocessor) in order to accelerate the computation. In accordance with the rules of physics, several processes operate on various matrix components, calculating the new matrix elements from the old ones. But until iteration n is ended, that is, until all processes have finished their present job, no process may begin on iteration n 1. Programming each process to carry out a barrier operation after completing its portion of the current iteration is the best technique to accomplish this aim. The new matrix, which serves as the input for the next iteration, will be completed once they are all complete, at which point all processes will be concurrently released to begin the following iteration.

Preventing Locks Read-Copy-Update

The simplest locks to use are none at all. It is unclear whether concurrent read and write operations to shared data structures can be made possible without locking. The response is categorically no in the usual situation. Take the example of process A sorting a list of numbers while process B computes the average. B could come across certain values several times and others not at all as a result of A shifting the values back and forth throughout the array. Any outcome is possible, but it would almost definitely be incorrect. But, in certain circumstances, we may let a writer to update a data structure while it is still in use by other processes. The challenge is to make sure that each reader reads either the old or the new version of the data, not some odd combination of the two. As an example, think about the tree. From the tree's roots to its leaves, readers go up and down it. A new node named X is added to the upper portion of the diagram. To do this, we initialise all variables in node X, including its child pointers, before making it visible in the tree. Then, with a single atomic write, we add X to A's family. A version that is inconsistent won't be read by anybody. We then exclude B and D from the bottom half of the diagram. Then, we set C as the left child pointer of A. Nodes B and D will never be seen by any readers who were in node A as they go on to node C. They will only view the updated version, in other words. All readers who are now in B or D will see the outdated version and continue to follow the original data structure points. Everything is OK; we never have to lock anything. The fact that RCU (Read-Copy-Update) decouples the removal and reclamation stages of the update is the primary factor in why removing B and D doesn't lock the data structure. Undoubtedly, there is an issue. We can't actually release B or D until we are certain that there are no more readers left. The length of time a reader may keep a reference to the data structure is governed by RCU care- fully. After that time, it is safe to take the memories back. Readers access the data structure specifically via a part of code called as the read-side crucial section, which may include any code as long as it does not block or sleep. In such situation, we are aware of how long we can wait. We specifically define a grace period as any period of time during which we are confident that each thread will at least occasionally be outside the read-side critical section. If we wait at least as long as the grace period before reclaiming, everything will be OK. A straightforward criterion is to wait until all threads have executed a context switch because code in a read-side critical section is not permitted to block or sleep.

Scheduling

Several programmes or threads fight for the CPU at once when a computer is multiprogrammed. This condition occurs when two or more of them are all at once in the ready stage. Which process to execute next must be decided when there is only one CPU available? The decision-making operating system component is referred to as the scheduler, and the method it uses is referred to as the scheduling algorithm. On these subjects, the material of the following parts is based. Many of the same issues that affect process scheduling also affect thread scheduling, despite minor distinctions. As the kernel controls threads, scheduling is often performed per thread, with little to no regard for the process that the thread belongs to. We will begin by concentrating on scheduling problems that affect both processes and threads. We will specifically examine thread scheduling and a few of the peculiar problems it poses later. We'll talk about multicore processors.

Overview of Scheduling

The scheduling method was straightforward in the earlier days of batch systems with input in the form of card images on a magnetic tape: merely perform the following task on the tape. The scheduling method got more complicated in multiprogramming systems since there were often many customers waiting for service. On certain mainframes, batch and timesharing services are still combined, thus the scheduler must choose between a batch operation and an interactive user at a terminal to determine which should come next. (As a side note, we will only assume that a batch job is a request to execute a single programme; nevertheless, a batch job may be a request to run many programmes in sequence.) As CPU time is a limited resource on these computers, a competent scheduler may significantly impact user happiness and perceived performance. So much effort has been put into creating clever and effective scheduling algorithms. The situation altered in two ways with the introduction of personal computers. Initially, there is often just one active process. It seems implausible that a user of a word processor would be developing a programme in the background while inputting data into the document. The word processor is the sole option, so the scheduler doesn't have to do any effort to decide which process to start when the user inputs a command into the word processor. Second, the CPU is no longer often a limited resource since computers have become so much quicker over time. Most computer programmes are limited by how quickly the user can enter data (by typing or clicking), not by how quickly the CPU can do it. Even compilations, formerly a significant drain on CPU resources, now often only take a few seconds. It barely matters which software starts first when two are really running at once, such as a word processor and a spreadsheet since the user is likely waiting for both of them to finish. Scheduling hence has little impact on basic Computers. There are programmes, of course, that virtually devour the CPU. Industrial-strength computer power is needed, for example, to render an hour of high-resolution video while adjusting the colours in each of the 107,892 (for NTSC) or 90,000 (for PAL) frames. Such applications do exist, but they are the exception rather than the norm. We notice a significant shift in the scenario when we look at networked servers. Here, numerous programmes often fight for CPU time, therefore scheduling is important. The users will be more pleased if the latter receives priority over the former when the CPU must decide between executing a process that fulfils user requests and one that collects the daily statistics, for instance. The "plenty of resources" argument is also untrue for many mobile devices, including sensor network nodes and smartphones (with the possible exception of the most powerful versions). The RAM may be little and the Processor remain unreliable in this case. However, as battery life is one of these devices' most significant limits, several schedulers work to reduce their power usage. Since moving between processes is costly, the scheduler must also be concerned with using the CPU as effectively as possible. The first step is to transition from user mode to kernel mode. The state of the running process must then be stored, including the registers, so that they may be subsequently loaded. Certain systems also need the storage of the memory map, such as the page table's memory reference bits. The scheduling algorithm must then be executed in order to choose a new process. The memory map of the new process must then be reloaded into the memory management unit (MMU). The new procedure must then be launched. On top of everything else, the process transition may invalidate the memory cache and associated tables, necessitating a dynamic reloading of the main memory twice (while entering and exiting the kernel) (upon entering the kernel and upon leaving it). Therefore, it is advisable to exercise care since performing too many process transitions per second might use a significant amount of CPU time.

Process Conduct

Almost all processes alternate short bursts of computation with requests for (disc or network) I/O. A system call to read from or write to a file is often performed after the CPU has continued to operate for a time. After the system call is finished, the CPU continues to calculate until more data is required or additional data has to be written, and so on. Be aware that certain I/O operations qualify as computation. For instance, the CPU is calculating, not doing I/O, when it transfers bits to a video RAM to refresh the screen. I/O in this context refers to the blocked state that a process enters as it waits for an external device to finish its task. They wait for I/O the majority of the time. The first are known as compute-bound or CPU-bound, while the second are known as I/O-bound. I/O-bound processes have frequent I/O waits and short CPU bursts, while compute-bound processes often have large CPU bursts and rare I/O waits. Not the duration of the I/O burst, but the length of the CPU burst, is what matters most. I/O-bound processes are I/O-bound not because their I/O requests are particularly lengthy, but rather because they do not compute much in between them. No matter how long or short it takes to process the data once they arrive, the hardware request to read a disc block takes the same amount of time to send. It is important to keep in mind that processes become increasingly I/O-bound as CPU speeds increase. Since CPUs are developing much more quickly than drives, this impact happens. As a result, scheduling I/O-bound operations will probably gain in significance in the future. The fundamental principle behind this is that an I/O-bound activity should be given the opportunity to execute as soon as possible in order to send its disc request and keep the disc active.

Making judgements on when to schedule is a crucial scheduling challenge. It turns out that scheduling is necessary in a number of different circumstances. The first choice to be made when a new process is formed is whether to run the parent process or the child process. Given that both processes are in the ready state, the option to execute the parent process or the child process next is a legal scheduling choice that may be made in any direction by the scheduler.

Second, once a process ends, a scheduling choice must be made. As that process is no longer available, another one must be selected from the list of prepared processes. A system-supplied idle process is typically launched if no processes are available. Third, another process has to be chosen to run when one is blocked by I/O, a semaphore, or for some other reason. Occasionally the decision may be influenced by the cause of obstruction. For instance, letting B run next will enable it to depart its critical area and allow A to continue if A is an essential process and it is waiting for B to exit its critical zone. The issue, however, is that the scheduler often lacks the knowledge required to take this reliance into account.

Fourth, a scheduling choice may be made in response to an I/O interrupt. If the I/O device that caused the interruption has finished its task, certain halted processes that were awaiting the I/O may now be able to proceed. The scheduler will determine whether to execute the recently ready process, the process that was already executing when the interrupt occurred, or a third process. A scheduling choice may be made at each clock interrupt or every kth clock interrupt if a hardware clock generates periodic interruptions at 50 or 60 Hz or another frequency. In terms of how they handle clock interruptions, scheduling algorithms may be split into two groups. An algorithm for nonpreemptive scheduling chooses a process to start and then merely lets it run until it stalls (either on I/O or while waiting for another process to finish) or willingly releases the CPU. It won't be forced stopped even if it runs for a long time. Clock interruptions essentially result in no scheduling choices being made. Except in cases where a higher priority process was waiting for a timeout that has already been fulfilled, after clock-interrupt processing is complete, the process that was active previous to the interrupt is restarted.

A preemptive scheduling method, in contrast, selects a process and allows it to operate for a maximum of a certain amount of time. It is suspended if it is still running at the end of the time interval, and the scheduler chooses another process to execute in its place (if one is avail- able). In order to return control of the CPU to the scheduler during preemptive scheduling, a clock interrupt must take place at the conclusion of the time period. Non-preemptive scheduling is the sole choice if there is no clock.

Scheduling Algorithms by Types

It should come as no surprise that different scheduling techniques are required in various situations. The reason for this condition is that various application domains (as well as various operating systems) have various objectives. In other words, not all systems need the scheduler to optimise for the same things. There are three distinct habitats that are Payroll, inventories, accounts receivable, accounts payable, interest computation (at banks), claims processing (at insurance firms), and other recurring operations are still often handled via batch systems in the corporate sector. There are no users eagerly waiting at their terminals for a prompt answer to a brief request in batch systems. Thus, it is often permissible to use non-preemptive algorithms or preemptive algorithms with lengthy time windows between processes. This strategy eliminates process switching and hence increases performance. Even for those who are not engaged in commercial mainframe computing, the batch methods are really rather broad and often useful in other contexts. Preemption is necessary to prevent one process from consuming the CPU and depriving other processes of service in an environment with interactive users. Even if no process ever ran endlessly on purpose, a computer fault may cause one process to permanently block all others. To stop this behaviour, preemption is required. Servers also fit into this category since they often provide services to many busy (remote) consumers. Computer users are always pressed for time. Preemption is surprisingly sometimes not required in systems with real-time limitations since the processes often complete their task and block swiftly because they are aware that they may not operate for lengthy periods of time. Real-time systems vary from interactive systems in that they only execute programmes that are meant to advance the application at hand. Interactive systems may execute arbitrary, uncooperative, and perhaps harmful applications since they are general-purpose devices.

Objectives for Scheduling Algorithms

Fairness is critical in any situation. Services should be provided using similar procedures. It is unfair to provide one process a lot more CPU time than an equivalent one. Of course, different process categories may be handled in various ways. Consider the computer centre of a nuclear reactor, where payroll and safety controls are handled.

Fairness and system enforcement are somewhat connected. The scheduler must ensure that the local policy is followed if it allows safety control operations to run whenever they like, even if it means payroll will be 30 seconds late. Another overarching objective is, wherever feasible, to keep the whole system active. More work is completed per second if the CPU and all I/O devices can be kept active than if some of them are idle. For instance, the scheduler determines which tasks are brought into memory and executed in a batch system. It is preferable to execute certain CPU-

and I/O-bound tasks simultaneously in memory as opposed to loading and performing all CPU-bound activities first, then all I/O-bound jobs when they are done. The CPU-bound programmes will compete for the CPU while operating under the latter method, leaving the disc inactive. Later, when the I/O-bound tasks arrive, the CPU will be idle while they compete for disc space. It is preferable to maintain continuous system operation by carefully balancing processes. Throughput, turnaround time, and CPU usage are the three indicators that major computer centres' administrators commonly use to gauge the efficiency of their systems. The system's throughput is the number of tasks it completes per hour. When everything is taken into account, doing 50 tasks in an hour is better than finishing 40 tasks in an hour. Turnaround time is the length of time, on average, that passes between the submission and completion of a batch task. It gauges how long it takes the typical user to get the result. Here's the guideline: Tiny is lovely. An method for scheduling that aims to improve throughput may not always reduce turnaround time. A scheduler that always ran short tasks and never ran lengthy jobs, for instance, may achieve an excellent throughput (many small jobs per hour) at the price of a bad turnaround time for the long jobs if there was a mix of short and long jobs. The lengthy tasks may never run, making the mean turnaround time indefinite while attaining a high throughput, if small jobs continued coming in at a reasonably constant pace. In batch systems, CPU usage is often employed as a statistic. Yet in reality, it is a bad measure. What actually important are the output (number of jobs produced per hour) and turnaround times for jobs (turnaround time). It would be comparable to grading vehicles based on how often their engines spin over an hour to use CPU usage as a criterion. Yet knowing when the CPU usage is approaching 100% might help you decide whether it's time to upgrade your computer's processing capacity. Several objectives are relevant for interactive systems. The most crucial one is to reduce reaction time, or the interval between giving a command and receiving the outcome. A user request to launch a programme or access a file should take priority over background work on a personal computer while one is active (for instance, reading and saving email from the network). Any interactive requests coming first will be viewed as providing excellent service. What is often referred to as proportionality is a slightly similar topic. People have a preconceived notion of how long things should take, which is frequently erroneous. Users understand when a request they view as hard takes a while, but when a request they see as easy takes a while, consumers get impatient. For instance, if pressing a button to begin uploading a 500 MB film to a cloud server takes 60 seconds, the user would likely accept it as the norm as he does not anticipate the upload to take 5 seconds. On the other hand, the user has other expectations when he clicks the symbol that cuts off the connection to the cloud server once the movie has been uploaded. By 30 seconds, the user will likely be cursing a blue streak, and after 60 seconds, he will be frothing at the mouth if it has not finished. This behaviour results from users' general belief that transferring large amounts of data should take much longer than just cutting the connection. The scheduler may influence the response time in certain circumstances (like this one), but not in all of them, particularly when the delay is the result of a poor choice of process order.

Interactive systems have distinct characteristics from real-time systems, and as a result, different scheduling objectives. They are distinguished by having deadlines that must be reached or at the very least should. For instance, if a computer is in charge of a device that generates data on a regular basis, data loss may occur if the data gathering procedure is not completed on schedule. Meeting all (or the majority of) deadlines is therefore the primary need for a real-time system. Predictability is crucial in certain real-time systems, particularly those using multimedia. Although sometimes missing a deadline is not deadly, the sound quality will quickly decrease if the audio process is too unpredictable. Video is a problem as well, although jitter is far more noticeable to the ear than to the sight. Process scheduling has to be very predictable and regular in order to prevent this issue. The supplemental content on multimedia operating systems on the book's website instead covers real-time scheduling.

Organizing Tasks in Batch Systems

It's time to move on from generic scheduling concerns to precise scheduling formulas. We shall examine batch system algorithms in this part. The ones after that will look at interactive and real-time systems. A few methods are used in both batch and interactive systems, it is important to note. They will be studied later.

First Arrival, Initial Service

Nonpreemptive first-come, first-served is perhaps the simplest scheduling algorithm ever created. The CPU is allotted to processes using this technique in the order in which they make their requests. In essence, there is only one queue of prepared processes. The first work that comes into the system in the morning is instantly begun and given the freedom to run for whatever long it wants to. It continues because it has gone on for too long. They are sent to the back of the line when new positions are added. The first process in the queue is executed after a current process blocks. A stopped process gets moved to the back of the line, after all waiting processes, when it is ready, much like a freshly arriving task. This algorithm's major strength is that it is simple to comprehend and similarly simple to develop. It's also justifiable in the same way as giving away expensive concert tickets or brand-new iPhones to those who are prepared to wait in line beginning at 2 a.m. This technique maintains track of all running processes in a single linked list. Just taking one out of the front of the queue allows you to choose which process to execute. Just connecting a new task or unblocked process to the end of the queue will add it. However, first-come, first-served also has a significant drawback. Pose- pose there are several I/O-bound processes that each need 1000 disc reads but utilise minimal CPU time, but one compute-bound process runs for one second at a time. After running for one second, the compute-bound process reads a disc block. Now that every I/O activity has started, disc reads have begun. After a fast succession of all the I/O-bound processes, the CPU-bound process continues for another second after receiving its disc block. Overall, each I/O-bound operation will read 1 block each second and take 1000 seconds to complete. The I/O-bound processes would complete in 10 seconds as opposed to 1000 seconds with a scheduling technique that preempted the compute-bound process every 10 msec, and without significantly slowing down the compute-bound process.

First the shortest job

Let's now have a look at another no preemptive batch method that presumptively knows the run timings. Someone at an insurance firm, for instance, can estimate with reasonable accuracy how long it will take to process a batch of 1000 claims since this kind of work is done every day. The scheduler chooses the shortest work first when numerous equally critical jobs are waiting to be started in the input queue.

Next in Least Time Left

Shortest task first is a preventive variant of shortest remaining time next. The scheduler always selects the process with the least remaining run time using this approach. Again, the runtime has to be determined beforehand. The entire duration of a new work is compared to the remaining time of the present process. The current process is paused and the new job is begun if the new task takes less time to complete than the existing one. This plan enables new, brief tasks to get top-notch servicing.

Interactive System Scheduling

We'll now take a look at a few interactive systems-friendly algorithms. They are typical on servers, personal computers, and other types of devices.

Round-Robin Planning

Round robin is one of the most well-known, fairest, and easiest algorithms. Each process is given a time window known as its quantum within which it is permitted to operate. The CPU is preempted and assigned to another process if the process is still active at the conclusion of the quantum. Naturally, the Processor switches off when the process stalls if it has stalled or finished before the quantum has passed. Implementing round robin is simple. The scheduler just has to keep track of a list of processes that are ready to execute. The procedure is moved to the end of the list after it has used up all of its quantum. The quantum's length is the only genuinely intriguing aspect of round robin. The administration involved in switching from one process to another takes some time, including saving and loading registers and memory mappings, updating different tables and lists, flushing and reloading the memory cache, and other tasks. Consider that this process transition, or context switch as it is commonly referred as, takes 1 msec to complete. This would include swapping memory mappings, flushing and reloading the cache, among other things. Let's assume that the quantum is set to 4 msec as well. At these settings, the CPU will have to squander 1 msec on process switching after 4 msec of productive work. Due to administrative overhead, 20% of CPU time will be wasted. This is obviously too much. We might increase CPU efficiency by setting the quantum to, say, 100 msec. Just 1% of the time is now lost. Now imagine if 50 requests with wildly different CPU needs arrive on a server system in a relatively short period of time. The list of executable processes will include 50 processes. The first one will start right away if the CPU is idle, the second one may not start for 100 songs, and so on. If all the other players utilise their whole quanta, the unfortunate last player could have to wait five seconds before getting a chance. The majority of users will consider a 5-sec response to a brief command to be slow. If some of the requests at the back of the queue used just a few milliseconds of CPU time, this scenario is much worse. They would have obtained better service with a smaller quantum. Preemption won't happen very often if the quantum is configured to be longer than the average CPU burst. Instead, most processes will transition to another process by performing a blocking action before the quantum expires. Performance is improved by eliminating preemption since switches in processes only occur when they are logically required, or when a process stops and cannot continue.

The conclusion may be stated as follows: setting the quantum too long may result in poor response to brief interactive requests, while setting it too short may result in too frequent process transitions and reduced CPU efficiency. A suitable compromise is often a quantum of 20–50 msec.

Priority Planning

The underlying premise of round-robin scheduling is that each activity is of equal importance. Owners and operators of multiuser computers often have quite diverse viewpoints on the matter. For instance, in a university, the president may come first, followed by the deans of the faculty, then the professors, secretaries, janitors, and lastly the students. Priority scheduling results from the need to account for outside circumstances. The fundamental concept is simple: each process is given a priority rating, and only the runnable process with the highest rating is permitted to proceed. There may be several processes running on a Computer with a single owner, some of which are more crucial than others. For instance, a process showing a video film on the screen in real time should be given a lower priority than a daemon process delivering electronic mail in the background.

The scheduler may reduce the priority of the presently running process at each clock tick to prevent high-priority processes from running endlessly (i.e., at each clock interrupt). A process transition takes place if this causes its priority to fall below that of the next highest process. As an alternative, a maximum time quantum might be allocated to each process. The next-highest-priority process is given a chance to run once this quantum has been used up. Processes may be given statically assigned or dynamically assigned priorities. In a military computer, a general's process may start at priority 100, a colonel's process at 90, a major's process at 80, a captain's process at 70, a lieutenant's process at 60, and so on down the totem pole. In contrast, in a for-profit computer centre, high-priority tasks can be charged at $100 per hour, medium-priority tasks at $75 per hour, and low-priority tasks at $50 per hour. In the UNIX operating system, the nice command enables a user to deliberately lower the priority of his process in an effort to be considerate to other users. It is never used.

The system has the ability to dynamically allocate priorities in order to accomplish certain system objectives. For instance, some processes are very dependent on I/O and spend the majority of their time doing nothing but waiting. When one of these processes requests the CPU, the CPU should be delivered to it right away so it may begin its subsequent I/O request and work in parallel with another process that is really doing the computation. Letting the I/O-bound process wait for the CPU for an excessively lengthy period of time will only result in it lingering in memory. Setting the priority to 1/f, where f is the fraction of the most recent quantum consumed by a process, is a straightforward approach for providing excellent service to I/O-bound processes. A process would get priority 50 if it utilised just 1 millisecond of its 50-millisecond quantum, priority 2 if it ran for 25 milliseconds before stalling, and priority 1 if it consumed the whole quantum. Grouping processes into priority classes, using priority scheduling among the classes but round-robin scheduling inside each class, is often useful. The scheduling strategy is as follows: never worry about lower-priority classes as long as there are runnable processes in priority class 4, simply run each one for one quantum in a round-robin way. Run the class 3 processes in a round-robin fashion if priority class 4 is empty. Run class 2 round robin if classes 4 and 3 are both vacant, and so on. Lower priority classes risk starvation if priorities are not periodically changed.

Several queues

One of the early priority schedulers was found in CTSS, the IBM 7094-based M.I.T. Compatible Time Sharing System. Since the 7094 could only retain one process in memory, CTSS had an issue with delayed process switching. At each switch, a new process was read in from disc and the existing one was swapped to disc. The CTSS designers rapidly understood that it was more effective to sometimes offer CPU-bound programmes a big quantum as opposed to doing it regularly (to reduce swapping). On the other hand, as we have previously shown, giving all processes a big quantum would result in slow reaction times. They came up with the idea of creating priority classes. The most advanced processes were conducted for one quantum. The next-highest class processes were run for two quanta. The next one ran processes for four quanta, etc. A process was demoted one class once it used all of the quanta allotted to it. Think of a procedure that required to continually calculate 100 quanta, for instance. It would get one quantum at first before being exchanged. The next time, it would get two quanta before being replaced. It would get 4, 8, 16, 32, and 64 quanta on subsequent runs, albeit it would only have needed 37 of the 64 final quanta to do the job. With a pure round-robin method, only 7 swaps would be required (including the initial load), as opposed to 100. Also, when the procedure descended further and farther down the priority queues, it would be executed less and fewer times, freeing up the CPU for quick, interactive tasks. The following rule was implemented to prevent penalising processes that initially required a lot of time to execute before becoming interactive. The process bound to a terminal was shifted to the highest priority class if a carriage return (Enter key) was typed, on the premise that the process was going to become interactive. One lovely day, a user with a process that was significantly CPU-bound found that just sitting at the terminal and randomly entering carriage returns every few seconds improved his response time. He told all of his pals. They informed each of their pals. The moral of the tale is that it is far more difficult to do anything properly in reality than it is in theory.

Next fastest process

It would be wonderful if shortest job first could be utilised for interactive processes as well since it consistently delivers the lowest average response time for batch systems. It can be, up to a point. Generally speaking, interactive processes go as follows: wait for command, execute command, wait for command, execute command, etc. If we think of each command's execution as a distinct "task," we may reduce the total response time by starting with the smallest one. Finding the shortest process among those that are now able to execute is the issue. Making predictions based on historical performance and executing the procedure with the least predicted running time is one method. Let's say that T0 represents the anticipated time per command for some process. Let's say that the subsequent run is measured as T1. By considering the weighted total of these two figures, or aT0 (1 a)T1, we might revise our estimate. By selecting a, we can determine whether the estimating method should swiftly forget previous runs or retain them for a lengthy period.

The weight of T0 in the revised estimate has decreased to 1/8 after three more runs. Often referred to as "led ageing," this method involves estimating the next value in a series by averaging the prior estimate and the current measured value. It may be used in a variety of circumstances when making a forecast based on historical data is necessary. When a 1/2, ageing is very simple to accomplish. All that is required is to multiply the total by two after adding the new value to the existing estimate (by shifting it right 1 bit).

CHAPTER 12

RELIABLE SCHEDULING

Dr. Ravishankar S Ulle, Assistant Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.ravishankarulle@cms.ac.in

Making genuine performance promises to consumers and then keeping those promises is a fundamentally different approach to scheduling. One achievable promise both achievable and simple to live: You will get around 1/n of the CPU power if there are n people signed in at the same time you are working. Similar to this, if all else is equal, each of the n processes operating on a single-user system should get 1/n of the CPU cycles. It looks reasonable. The system must keep track of how much CPU each process has used since its inception in order to deliver on this promise. The time since creation divided by n is then used to calculate how much CPU each one is entitled to. Calculating the ratio of real CPU time utilised to CPU time entitled is not too difficult since we also know how much CPU time each process has actually got. A ratio of 0.5 indicates that a process received just half of what it was due, while a ratio of 2.0 indicates that the process received twice as much as it was allowed. The algorithm will then keep doing the procedure with the lowest ratio up until it surpasses the ratio of its nearest rival. Then that candidate is picked to run second.

Lottery Timetables

Although it is a good idea to make promises to consumers and then keep them, doing so might be challenging. A different approach, nevertheless, may be used to get results that are equally predictable and need a considerably simpler implementation. It's referred to as lottery scheduling. Giving processes lottery tickets for different system resources, such CPU time, is the underlying principle. The process holding the winning lottery ticket is given the resource whenever a scheduling decision has to be made. In the context of CPU scheduling, the system may conduct a lottery 50 times per second, awarding 20 msec of CPU time to each winner. All processes are equal, but some processes are more equal, to quote George Orwell. To boost their chances of winning, more significant processes may be granted more tickets. A process will have a 20% chance of winning each lottery if it retains 20 of the 100 unclaimed tickets. It will eventually get 20% of the CPU. Thus, the rule is quite clear: a process holding a fraction f of the tickets will get about a fraction f of the resource in question. This is in contrast to a priority scheduler, where it is extremely difficult to define what having a priority of 40 truly implies. There are various intriguing aspects to lottery scheduling. For instance, if a new process appears and is given some tickets, it will have a chance of winning the very next lottery in proportion to the amount of tickets it possesses. Hence scheduling for lotteries is quite responsive. Whenever they like, cooperative processes may trade tickets. To improve the likelihood of the server running next, a client process that delivers a message to a server process and then stalls could give the server all of its tickets. The tickets are returned by the server after it is completed, allowing the client to restart. In fact, servers don't even need tickets when there are no customers. Problems that are difficult to tackle using existing techniques may be resolved with lottery scheduling. One example is a video server where many processes are delivering video streams to customers at various frame rates. Assume that the processes need 10, 20, and 25 frames per second. These processes will automatically split the CPU in roughly the right proportion, i.e., 10: 20: 25, by being given 10, 20, and 25 tickets, respectively.

Equitable Scheduling

We have assumed up to this point that each process is scheduled independently, regardless of who its owner is. With round robin or equal priority, user 1 will get 90% of the CPU and user 2 will receive just 10% of it if user 1 starts nine processes and user 2 begins one. Some systems consider the user who owns a process before scheduling it to avoid this scenario. In this paradigm, a certain percentage of the CPU is allotted to each user, and the scheduler chooses processes to enforce this allocation. No of how many processes are running, if two users are each guaranteed 50% of the CPU, they will get that amount. Consider a system with two users who were each promised 50% of the CPU as an example. Whereas user 2 only has one process, E, user 1 has four processes: A, B, C, and D. If round-robin scheduling is used, the following scheduling arrangement may be utilised that satisfies all the requirements:

Real-Time Scheduling in Systems

A system that incorporates time as a key component is known as real-time. Normally, a computer must respond properly to stimuli within a certain length of time that are generated by one or more physical devices outside of the computer. As an example, the computer inside a CD player receives the bits as they leave the drive and has a very limited amount of time to turn them into music. The music will sound odd if the computation takes too long. Such real-time systems include the autopilot in an aeroplane, the robot control in an automated factory, and patient monitoring in a hospital critical care unit. In each of these scenarios, having the correct solution at a terrible time may be just as detrimental as not having one at all. Hard real-time systems, in which there are strict deadlines that must be fulfilled at all costs, and soft real-time systems, in which occasional deadline violations are undesirable but tolerated. In both situations, a programme is divided into a number of processes, each of whose behaviour is predictable and known in advance, to achieve real-time activity. These processes may run their course in well under a second and are typically brief in duration. It is the scheduler's responsibility to arrange the operations so that all deadlines are fulfilled when an external event is identified. The types of events that a real-time system could be required to handle include periodic (meaning they happen at regular intervals) and aperiodic events (meaning they occur unpredictably). Several periodic event streams may need a system to react. Handling all of them may not even be feasible, depending on how much time each event takes to process. A real-time system is considered to be schedulable if it fits this requirement. For instance, if there are m periodic events and event I occurs with period Pi and needs Ci sec of CPU time to process each event. This indicates that it can be put into practise. A process that fails to pass this test cannot be scheduled because it requests more CPU time than the CPU can provide individually. Consider, for instance, a soft real-time system with three periodic events that occur at intervals of 100, 200, and 500 msec. The system may be scheduled if each of these events requires 50, 30, and 100 msec of CPU time, respectively, since 0. 5 0. 15 0. 2 1. The system will continue to be scheduleable even if a fourth event with a period of one second is introduced, provided that this event requires no more than 150 msec of CPU time per event. This computation makes the implicit assumption that the overhead associated with context switching is so negligible that it can be disregarded. Algorithms for real-time scheduling might be static or dynamic. Before the system is put into operation, the former schedules its activities. The latter choose their scheduling during runtime, after the beginning of execution. Static scheduling only works when there is complete knowledge of the work to be done in advance and the deadlines that must be reached. Algorithms for dynamic scheduling are not constrained by these limitations.

Regulation vs. Mechanism

Up until this point, we have implicitly assumed that every process in the system belongs to a distinct user and is vying for CPU power. Although this is often the case, it may also happen that one process has a large number of dependent processes operating behind it. A database management system process, for instance, could produce several offspring. Each youngster may be working on a separate request or may each be assigned a particular task (query parsing, disc access, etc.). It is absolutely likely that the primary process has a great understanding of which of its offspring are the most crucial or vital in terms of timing, and which are the least important. Sadly, none of the schedulers covered above allow suggestions for scheduling from user processes. The scheduler thus seldom chooses well. This issue may be resolved by adhering to the tried-and-true concept of separating the scheduling method from the scheduling policy. This implies that while the scheduling method is somewhat parameterized, user processes may fill in the parameters. Again, let's think about the database scenario. Consider a scenario in which the kernel implements a priority-scheduling mechanism but also provides a system call that enables a process to establish (and modify) the priorities of its offspring. The parent may therefore manage the scheduling of its offspring even when it does not carry out the scheduling itself. In this case, the policy is determined by a user process, but the mechanism is in the kernel. The principle of separating policy mechanisms is crucial.

Scheduling threads

There are two layers of parallelism present when many processes have several threads—processes and threads. Depending on whether user-level threads or kernel-level threads (or both) are allowed, scheduling in such systems varies significantly. Let's start by thinking about user-level threads. The kernel continues to function as usual, choosing a process, let's say A, and handing A control for its quantum since it is unaware that there are threads. A's internal thread scheduler chooses which thread to execute, let's say A1. This thread may run indefinitely since multiprogramming threads are not subject to clock interruptions. The kernel will choose another process to execute if it exhausts the process' quantum. Thread A1 will start operating after Process A has fully restarted. A's time will be completely consumed by it till it is done. Its antisocial conduct won't have an impact on other processes, however. No of what happens inside process A, they will get whatever the scheduler deems to be their proper portion.

Any of the scheduling algorithms mentioned above may be employed by the run-time system. The most typical scheduling methods in use are round-robin scheduling and priority scheduling. The only restriction is the lack of a clock to stop a thread when it has gone on for too long. This is often not a problem since threads cooperate.

Now think about the threads running at the kernel level. In this case, the kernel selects a specific thread to run. It is not required to consider the process to which the thread belongs, but it may do so if it so chooses. A quantum is assigned to the thread, and if the quantum is exceeded, the thread is forcibly suspended. With these parameters and user-level threads, the thread order for some period of 30 msec might be A1, B1, A2, B2, A3, B3, but with a 50-msec quantum and threads that block after 5 msec, this is not possible.

The performance of user-level threads and kernel-level threads differs significantly. With user-level threads, switching threads requires a number of machine instructions. When using kernel-level threads, the process necessitates a full context switch, which is much slower and involves altering the memory map and invalidating the cache. However, unlike user-level threads, kernel-level threads do not suspend the entire process when a thread block occurs on I/O. The kernel can consider this knowledge when making a decision because it is aware that running a second thread in process A is more expensive than switching from a thread in process A to a thread in process B (due to having to change the memory map and having the memory cache spoiled). For instance, if there are two otherwise equally important threads, one of them is part of the same process as a thread that just blocked, and the other is part of a different process, the former could be given preference. User-level threads' ability to use an application-specific thread scheduler is another significant aspect. Think about it, for instance. Assume that the dispatcher thread, two worker threads, and a worker thread have just blocked and are ready. Who should enter the next election? Knowing what each thread does allows the run-time system to choose which dispatcher to run next so that a new worker can be started. This method maximises parallelism in a situation where workers frequently experience disc I/O blockages. The kernel would never be aware of what each thread did with kernel-level threads (although they could be assigned different priorities). However, application-specific thread schedulers are typically better at tuning an application than the kernel is.

IPC Classic Problems

Numerous intriguing issues have been thoroughly discussed and examined in the operating systems literature using a range of synchronisation techniques. We will look at three of the more well-known issues in the sections that follow.

The Problem of the Dining Philosophers

The dining philosophers problem was a synchronisation issue that Dijkstra posed in 1965 and then resolved. Since then, everyone who has created a new synchronisation primitive has felt compelled to show how beautifully it resolves the dining philosopher's problem in order to prove how great the new primitive is. The issue is easily explained as follows. The group of five philosophers is gathered around a globular table. The spaghetti is on a plate for each philosopher. One needs two forks to eat the spaghetti because it is so slick. One fork sits between each set of two plates. A philosopher's day is divided into intervals of eating and contemplation. (Even for philosophers, this is a bit of an abstraction, but those other pursuits are unimportant at this point.) A philosopher will attempt to obtain her left and right forks one at a time, in either order, once she has become sufficiently hungry. If she is successful in getting two forks, she eats for a while, puts the forks down, and keeps thinking. Can you create a programme for each philosopher that accomplishes its goals without ever becoming stuck? The two-fork rule has been criticised as being somewhat artificial; perhaps we should switch from Italian to Chinese cuisine, using rice instead of spaghetti and chopsticks instead of forks.) The take fork procedure watches for the available fork before grabbing it. Sadly, the obvious answer is incorrect. Assume that each of the five philosophers uses their left fork at the same time. There will be a deadlock because nobody will be able to take their right fork. It would be simple to change the programme so that it looks to see if the right fork is available after taking the left fork. If not, the philosopher sets the left one down, waits a while, and then goes through the process again. The failure of this proposal is also due to a different factor. By simultaneously picking up their left forks, noticing that their right forks were unavailable, putting them down, waiting, picking up their left forks simultaneously once more, and so on indefinitely, all the philosophers could start the algorithm with a little bit of bad luck. Starvation is what happens when all of the programmes continue to run indefinitely without ever moving forward. Even if the issue does not arise in an Italian or a Chinese restaurant, it is still referred to as starvation.

You might be thinking right now that the likelihood that everything would continue in lockstep for even an hour is extremely slim if the philosophers simply waited a random time instead of the same time after failing to obtain the right-hand fork. This observation is accurate, and it is generally not a problem to try again later. For instance, in the well-known Ethernet local area network, if two computers send a packet simultaneously, each one waits a random amount of time before attempting again; in practise, this solution is effective. In some situations, though, it would be preferable to have a solution that always succeeds and cannot go wrong because of an unlikely set of random numbers. Consider the safety measures at a nuclear power plant. The five statements that follow the command to think are protected from starvation and deadlock by a binary semaphore. A philosopher would perform a down on mutex before beginning to acquire forks. She would perform an up on mutex after changing the forks. Theoretically speaking, this solution works well. From a performance standpoint, it has a flaw: only one philosopher can be eating at any given time. We should be able to accommodate two philosophers eating simultaneously with the five forks we have available. deadlock-free and enables the greatest amount of parallelism for any quantity of philosophers. It monitors whether a philosopher is eating, thinking, or hungry using an array called state (trying to acquire forks). Only if neither of their neighbours is currently eating may a philosopher enter the eating state. The macros LEFT and RIGHT define the boundaries of philosopher i's neighbours. For example, if I is 2, LEFT is 1 and RIGHT is 3, respectively. One semaphore is used per philosopher in the program's array so that hungry philosophers can block if the necessary forks are occupied. Take note that while take forks, put forks, and test are all regular procedures and not separate processes, each process runs philosopher as its main code.

The Issue with Readers and Writers

Modeling processes that compete for exclusive access to a finite number of resources, like I/O devices, is made easier by using the dining philosopher's problem. The Readers and Writers Problem, which simulates database access, is another well-known problem. Consider a system for making airline reservations where there are numerous competing processes that want to read from and write to it. While multiple processes can read the database concurrently, no processes—not even readers—are permitted to access the database while one process is updating (writing) it. In this solution, the semaphore db is downed by the first reader who gains access to the database. Readers after them simply add to a counter, rc. A blocked writer, if there is one, can enter by doing an up on the semaphore as readers leave, decrementing the counter.

The solution given here implicitly makes a deft choice that should be noted. Let's say a reader is using the database when another reader walks by. The second reader is allowed because it is not a problem to have two readers reading at once. If more readers arrive, they can also be admitted. Imagine that a writer now appears. Since writers must have exclusive access to the database, the writer is suspended because they may not be allowed access. Later, more readers come in. Additional readers are permitted as long as there is still at least one active reader. As a result of this strategy, everyone will be admitted as soon as they arrive so long as there is a steady flow of readers. The writer will remain out of commission until no readers are present. The writer will never enter if a reader is added, say, every two seconds and each reader takes five seconds to complete its task. The programme could be written a little differently to prevent this situation:

when a reader arrives and a writer is waiting, the reader is suspended behind the writer rather than being admitted right away. In this way, a writer must wait for readers who were present when it began, but need not wait for readers who arrived later. This solution's drawback is that it produces lower performance and less con- currency. The solution put forth by Courtois et al. prioritises writers. We direct you to the paper for more information.

There are some topics that are much more settled than others, as will become apparent over time. Instead of topics that have been around for decades, new topics tend to receive the majority of research attention. As an example of something that is largely established, consider the idea of a process. The idea of a process as a container for gathering together related resources, such as an address space, threads, open files, protection permissions, and so forth, is present in almost every system. The grouping is carried out slightly differently by various systems, but these are purely technical variations. There isn't much new research on the subject of processes, and the fundamental idea is no longer particularly contentious. Though they are a more recent concept than processes, threads have also received considerable thought. A paper on threads does, however, occasionally appear. For instance, Tam et al. discussed thread clustering on multiprocessors and how well modern operating systems like Linux scale with numerous threads and cores. The ability to record and playback a process's execution is one area of research that is currently very active. Replaying aids security experts in investigating incidents and developers in locating elusive bugs. Similar to this, a lot of current operating systems research focuses on security-related issues. Numerous incidents have shown that users require better defence against attackers (and, occasionally, from themselves). Some researchers suggest moving the process to a more powerful cloud server as needed to accommodate the growing amount of computation on underpowered, battery-constrained smartphones. However, it doesn't seem like many actual system designers are spending their days wringing their hands over the lack of a good thread-scheduling algorithm. It seems that this kind of research is more driven by the researcher than by consumer demand. Overall, research on processes, threads, and scheduling is not as popular as it once was. Topics like power management, virtualization, clouds, and security are now being studied.

Memory administration

Primary memory (RAM) is a crucial resource that has to be handled with extreme care. Despite the fact that the typical home computer now has 10,000 times more memory than the IBM 7094, which was the world's biggest computer in the early 1960s, applications are growing more quickly than memories. Programs "grow to fill the memory available to retain them," to paraphrase Parkinson's Law. The creation and management of memory-based abstractions by operating systems. Every programmer wants a memory that is private, indefinitely big, infinitely quick, and nonvolatile—that is, one that doesn't lose its data when the power is turned off. Why not also make it affordable while we're at it? Unfortunately, such memories are not yet available thanks to technology. Maybe you'll figure out how to accomplish it. Over time, people learned about the idea of a memory hierarchy, in which computers have a small amount of very quick, expensive, volatile cache memory, a small amount of medium-speed, affordable, volatile main memory, and a small amount of slow, inexpensive, nonvolatile magnetic or solid-state disc storage, not to mention removable storage like DVDs and USB sticks. The operating system's responsibility is to manage the abstraction and turn this hierarchy into a useable model. The memory manager is a component of the operating system that controls (part of) the memory hierarchy. Its responsibility is to effectively manage memory by keeping track of which areas are being used, allocating memory to processes as needed, and freeing up space after usage. It will examine a variety of memory management strategies, from the most basic to the most complex. Concept of main memory and how it may be controlled because controlling the lowest level of cache memory is often handled by the hardware. The administration of persistent storage the disk by looking at the most straightforward plans before moving on to ever-more complex ones.

Having no memory abstraction

The absence of any memory abstraction is the most basic. Early main- frame computers (before 1960), early minicomputers (before 1970), and early per- sonal computers (before 1980) had no memory abstraction. The physical memory was all that any software saw. The contents of physical memory address 1000 were simply relocated to register 1 when a program's command MOV REGISTER1, 1000 was performed. As a result, the model of memory given to the programmer was only physical memory, which consisted of a set of addresses ranging from 0 to a maximum and corresponded to cells with various numbers of bits, most often eight. Two active programmes could not coexist in memory at the same time under these circumstances. The value that the second programme was storing at position 2000 would be overwritten if the first programme wrote a new value to, say, location 2000. Nothing would function, and both applications would instantly crash. There are several alternatives even when memory is represented as merely physical memory. a trio of variants. The device drivers might be at the top of memory in a ROM, with the operating system being at the bottom in RAM (Random Access Memory), or vice versa. Once utilised on mainframes and minicomputers, the first model is now seldom used. Several portable computers and embedded systems employ the second model. Early personal computers (running MS- DOS, for example) employed the third model, in which the BIOS is the part of the system that is stored in ROM (Basic Input Output System). The drawback of models (a) and (c) is that a defect in the user software might potentially destroy the operating system, with devastating consequences. When the system is set up this manner, often only one process can be active at once. The operating system transfers the desired software from disc to memory and starts it running as soon as the user inputs a command. The operating system shows a prompt character and awaits a user-new command after the procedure is finished. The operating system loads a new programme into memory, overwriting the old one, when it gets the command. Programming with many threads is one technique to achieve some parallelism in a system without memory abstraction. The fact that they are compelled to do not present an issue since each thread in a process should view the same memory picture. While this concept works, it is only of limited value since what users often desire is for unrelated programmes to execute concurrently, which the threads abstraction does not provide. Additionally, it is improbable that a system that lacks a memory abstraction would also have a threads abstraction.

Several Processes Executed Without Memory Abstraction

Nonetheless, it is still feasible to execute numerous programmes at once even in the absence of memory abstraction. The operating system must first save all of the contents of memory to a disc file before loading and executing the subsequent application. There are no conflicts as long as there is just one application running at once in memory. The discussion of this idea (swapping) will follow. Even without swapping, it is feasible to run numerous programmes simultaneously with the inclusion of certain specialised hardware. The issue was resolved in the following way by the original IBM 360 models. Each 2-KB block of memory had a 4-bit protection key that was stored in specialised registers within the CPU. For a total of 256 bytes of key storage on a machine with a 1-MB memory, just 512 of these 4-bit registers were required. Besides a 4-bit key, the PSW (Program Status Word) also had one. Every attempt by a running process to access memory using a protection code other than the PSW key was blocked by the 360 hardware. User processes were kept from interfering with one another and the operating system as only the operating system was able to modify the protection keys. Yet, there was a significant flaw in this approach. Shaded lines indicate that the former has a different memory key than the latter. Jumping to address 24, which includes a MOV instruction, is where the first programme begins. The CMP instruction is located at location 28, where the second programme first jumps. The directions that are not discussed here are not shown. When the two applications are loaded one after the other, beginning at address 0, in memory. In this example, we'll suppose that the operating system is in high memory and won't be visible.

The apps may be executed once they have loaded. They cannot harm one other since their memory keys are different. But, the issue is of a different kind. The JMP 24 instruction, which jumps to the instruction as predicted, is executed by the first programme when it launches. This software runs properly. The operating system, however, can choose to execute the second programme, which was loaded above the first one, at address 16,384 once the first programme has run for a sufficient amount of time. The first instruction to be carried out is JMP 28, which instead of jumping to the intended CMP instruction in the first programme, jumps to the ADD instruction instead. Very likely, the software will crash in little under a second. The two applications' shared use of absolute physical memory is the main issue in this situation. We absolutely do not want that. The ability for any programme to make private, local references is what we want. We'll quickly demonstrate how to do this. As a workaround, the IBM 360 used a process called as static relocation to dynamically adjust the second application as it was being put into memory. It operated in this way. Every programme address throughout the loading process was increased by the constant 16,384 when a programme was loaded at address 16,384 (for example, "JMP 28" became "JMP 16,412," etc.). While this approach works well, it is not a particularly universal fix and slows down loading. All executable programmes must also include additional information indicating which words include (relocatable) addresses and which do not. For all, an instruction like MOV REGISTER1,28, which changes the integer 28 to REGISTER1, cannot be moved yet the "28" in Fig. 3-2(b) must. The loader requires a mechanism to identify an address. Lastly, in the domain of computers, history often repeats itself. On mainframes, minicomputers, desktop computers, laptop computers, and smartphones, direct addressing of physical memory is a thing of the past. Nevertheless, embedded and smart card systems still often lack a memory abstraction. The software (in ROM) in modern appliances like radios, washing machines, and microwave ovens almost always refers to absolute memory. Because consumers are not permitted to run their own software on their toasters, this works since all the applications are known in advance. Complex operating systems are present in high-end embedded devices (such as cellphones), but not in basic ones. There may be an operating system present in certain circumstances, but it is only a library that is coupled with the application software and offers system functions for carrying out I/O and other typical operations. An operating system that doubles as a library is the e-Cos operating system.

Overall, there are a number of significant disadvantages to exposing physical memory to processes. Secondly, since user applications had access to every byte of memory, they might easily damage the operating system accidentally or on purpose, putting the machine to a complete standstill (unless there was specialised hardware like the lock-and-key mechanism on the IBM 360). Even if just one user programme (application) is running, this issue still occurs. Second, executing numerous applications simultaneously is challenging with this technique (taking turns, if there is only one CPU). With personal computers, it is typical to have many open applications (a word processor, an email client, a web browser), with one taking the current focus and the others responding to mouse clicks. It was necessary to take action since this circumstance is challenging to accomplish without any abstraction from physical memory.

CHAPTER 13

ADDRESS SPACE CONCEPT

Dr. Vinoth Kumar. V, Assistant Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.vinothkumar_v@cms.ac.in

On the IBM 360, a crude solution to the first problem included labelling sections of memory with a protection key and comparing it to each memory word read. The latter issue cannot be resolved by this method alone, while it can be done by moving applications as they are loaded, but this is a sluggish and difficult solution. A new abstraction for memory called the address space offers a more effective approach. The ad- hoc space produces a form of abstract memory for programmes to reside in, just as the process notion generates an abstract CPU to execute programmes. The range of addresses that a process may utilise to address memory is known as an address space. Each process has a unique address space that is separate from those used by other processes (except in some special circumstances where processes want to share their address spaces). An address space may be thought of in many different ways and has a very broad definition. Think about phone numbers. A local phone number is often a seven-digit number in the US and many other nations. As a result, the address space for telephony numbers ranges from 0,000,000 to 9,999,999, however certain numbers, including those starting with 000, are not utilised. This area is becoming too tiny due to the proliferation of cellphones, modems, and fax machines, thus more digits must be utilised. In the x86, the I/O ports' address space ranges from 0 to 16383. The address space for IPv4 addresses ranges from 0 to 232 1 since they are 32-bit values (again, with some reserved numbers). It's not required for address spaces to be numeric. Another address space is the collection of Internet domains ending in.com. This address space comprises of all strings that can be created using letters, digits, and hyphens that range in length from 2 to 63 characters, followed by the dot com. You should now understand the concept. That is quite easy. How to provide each programme its own address space is a little trickier; for example, address 28 in one programme may refer to a different physical place than address 28 in another. We'll go through a straightforward method that was once popular but has since become obsolete as a result of the availability of far more intricate (and superior) strategies on contemporary CPU processors.

Limit and Base Registers

This easy fix makes advantage of a particularly simple kind of dynamic relocation. It simply maps the address space of each process onto a distinct region of physical memory. The traditional way is to provide each CPU two specialised hardware registers, often referred to as the base and limit registers. This method was utilised on machines ranging from the CDC 6600 (the first supercomputer in history) through the Intel 8088 (the brain of the first IBM PC). Programs are loaded into successive memory regions whenever there is space and without relocating during loading when these registers are utilised. When a process is executed, the physical location in memory at which its programme starts is loaded into the base register, and the program's length is placed into the limit register. When the first programme is executed, the base and maximum values that would be placed into these hardware registers are 0 and 16,384, respectively. When the second programme is executed, the values utilised are 16,384 and 32,768, respectively. The base and limit registers would be 32,768 and 16,384 if a third 16-KB programme were loaded and executed immediately above the second one. The CPU hardware automatically adds the base value to the address created by the process before sending the address out on the memory bus whenever a process contacts memory, whether to fetch an instruction or read or write a data word. The address given must be equal to or larger than the value in the limit register for it to pass; otherwise, a fault is created and the access is terminated. Hence, in the case of the second program's first instruction Since every memory address created automatically has the base-register contents appended to it before being transmitted to memory, using base and limit registers is a simple approach to provide each process its own private address space. The base and limit registers are often safeguarded such that only the operating system is able to make changes to them. On the CDC 6600, this was the case, but not on the Intel 8088 since it lacked the limit register. While it did include a number of base registers, enabling programme text and data to be moved separately, for example, it did not provide any protection against out-of-range memory accesses. Relocation utilising base and limit registers has the drawback of requiring an addition and a comparison to be performed for each memory reference. With the use of specific addition circuits, comparisons may be performed quickly, but adds take longer because of carry-propagation time.

Swapping

The techniques discussed so far will more or less work if the computer's physical memory is big enough to accommodate all of the processes. In actuality, however, the total amount of RAM required by all the operations is sometimes far more than what can be stored in memory. As soon as the computer boots, 50–100 processes or more may begin running on a normal Windows, OS X, or Linux machine. For instance, when a Windows programme is loaded, it often provides instructions that cause a process to be created the next time the system boots and that process will only check for updates to the current application. A procedure of this kind may easily use 5–10 MB of RAM. Several other items, including incoming network connections and mail, are checked by other background programmes. All of this has happened prior to the launch of the first user programme. These days, serious user application applications like Photoshop might easily need 500 MB merely to launch and several terabytes once they begin processing data. As a result, maintaining all processes in memory at all times needs a substantial amount of memory and is impossible with inadequate memory. Throughout time, there have been two main strategies for handling memory overflow. The simplest method, known as swapping, is loading each process into memory in its entirety, executing it for a time, and then putting it back on the disc. Although idle processes are often saved on disc, when they are not in use, they do not consume RAM (although some of them wake up periodically to do their work, then go to sleep again). The alternative approach, known as virtual memory, enables programmes to function even if they only occupy a portion of main memory. Swapping is the topic of the section below, whereas virtual memory.

By relocating all the processes as far lower as feasible, memory gaps caused by swapping may be combined into a single large one. The term "memory compaction" refers to this method. Since it takes a lot of CPU time, it is often not completed. For instance, to condense the whole memory on a 16 GB system that can copy 8 bytes in 8 nsec would take roughly 16 sec. How much memory should be allotted for a process when it is created or swapped in is an important argument to make?

The allocation is straightforward if processes are established with a set size that never changes: the operating system allo- cates precisely what is required, no more and no less. Yet if processes' data segments are expandable, as in many programming languages, for instance by dynamically allocating memory from a heap, then a problem arises anytime a process wants to expand. If if the procedure is the expanding process, on the other hand, will need to be transferred to a hole in memory big enough for it, or one or more processes will need to be switched out to provide a big enough hole, if the process is next to another process. A process must be halted until more space is available if it is unable to expand in memory and the swap area on the disc is full (or it can be killed). In order to avoid the overhead associated with moving or switching processes that can no longer fit in their assigned memory, it is usually a good idea to allocate a little more memory whenever a process is swapped in or relocated if it is anticipated that most processes will expand as they run. Therefore, when switching processes to disc, only the memory currently in use should be changed; it is unnecessary to shift the additional memory as well. A different configuration, namely, suggests itself if processes are capable of having two growing segments, such as the data segment serving as a heap for variables that are dynamically allocated and released and the stack segment housing the usual local variables and return addresses. In this diagram, each process is shown with a building stack at the top of its allotted memory and an upward-growing data segment just beyond the programme text. Both segments may utilise the memory that exists between them. If it does, the process will need to be either terminated, transferred to a hole with appropriate capacity, or switched out of memory until a hole big enough can be made.

Paging

The majority of virtual memory systems contain paging, which we will now discuss. Any computer's programmes utilise a list of memory addresses. A software executes an instruction like MOV REG, 1000 to move data from memory address 1000 to REG (or vice versa, de- pending on the computer). Addresses may be made using a variety of techniques, including indexing, base registers, segment registers, and others.

These program-generated addresses, sometimes referred to as virtual addresses, make up the virtual address space. On devices without virtual memory, when the virtual address is put directly into the memory bus, the physical memory word at the same location is read or written. When virtual memory is used, the virtual dresses do not transfer straight to the memory bus. As an alternative, they go to an MMU, which transforms virtual addresses into physical memory addresses. In this example, a computer generates 16-bit addresses with a range of 0 to 64K. These are the virtual addresses although but on this is just this memory, this memory is only 32. This memory on this machine. This memory, but this memory is only 32. As a result, 64-KB programmes may be written but cannot be run since they cannot be loaded into memory completely. Nevertheless, a complete duplicate of a program's core image, up to 64 KB, must be kept on the disc in order for components to be brought in as needed. The pages that make up the virtual address space are fixed-size objects. The physical memory's equivalent components are called page frames. Pages and page frames are typically the same size. In this example, they are 4 KB, but in real systems, page sizes have ranged from 512 bytes to a gigabit. Using 64 KB of virtual address space and 32 KB of physical memory, we are able to create 16 virtual pages and 8 page frames. Memory to disc transfers are always carried out in complete pages. The operating system may mix and match different page sizes on several processors as it sees fit. As the x86-64 architecture supports 4-KB, 2-MB, and 1-GB pages, we could, for example, employ 1-GB pages for the kernel and 4-KB pages for user programmes. Later, we'll see why it's often advantageous to use a single, large page rather than multiple smaller ones. According to the range 0K-4K, the virtual or physical addresses on the page are 0 through 4095. The range 4K-8K includes the addresses 4096 to 8191 and so on. Each advertising starts at a multiple of 4096 and ends one short of a multiple of 4096, thus there are exactly 4096 adverts on each page. The MMU receives virtual address 0 anytime the programme requests access to address 0, such as when using the MOV REG instruction. The MMU detects that this virtual address is on page 0 (0 to 4095), which, according to its mapping, is page frame 2. (8192 to 12287). In order to output the address to the bus as address 8192, the address is changed to 8192. The memory is completely unaware of the MMU; it just detects a request for reading or writing address 8192 and complies. The result is that the MMU has successfully mapped the virtual addresses 0 to 4095 onto the real values 8192 to 12287.

The problem of the virtual address space being larger than the physical memory is not resolved by the fact that the 16 virtual pages may be mapped into any of the eight page frames by correctly setting the MMU's map. Eight physical page frames, which is all we have, are mapped into physical memory. The cross-shaped group of others in the image is not mapped. A Present/Absent bit in the actual hardware keeps track of which pages are really in memory.

What happens if the programme executes, for example, the MOV REG,32780 instruction, which is found in byte 12 of virtual page 8 (starting at 32768)? When the MMU determines that the page is unmapped, as shown by a cross in the image, the CPU sends a trap to the operating system. This trap is called a page fault. A page frame that hasn't been used lately is chosen by the operating system, and its data is copied back to the disc (if it is not al- ready there). Then it restarts the trapped instruction and changes the map before bringing the previously referenced page into the newly freed page frame (also from the disc).

If the operating system chose to evict page frame 1, for example, it would load virtual page 8 at physical address 4096 and make two changes to the MMU map. To prevent any further visits to virtual addresses between 4096 and 8191, the first step would be to mark the record on virtual page 1 as unmapped. As a result, when the trapped instruction is executed once again, it will change the cross in the entry of virtual page 8 to a 1. p., transferring the virtual address 32780 to the physical address 4108 (4096 + 12).

In a straightforward implementation, the mapping of virtual addresses into real addresses may be stated as follows: An offset plus a virtual page number (high-order bits) make up the virtual address (low-order bits). For instance, the top 4 bits of a 16-bit address may identify one of the 16 virtual pages, while the bottom 12 bits could identify the byte offset (0 to 4095) inside the selected page. It is possible to divide the page into 3 or 5 bits or another number of bits, however. Page sizes match different divides. The virtual page number is used as an index into the page database to find the record for that virtual page. The page table item may be used to get the page frame number, if it is present. The page frame number is connected to the high-order end of the off-set in lieu of the virtual page number to produce a physical address that may be sent to the memory. The purpose of the page table is to connect virtual pages to page frames as a consequence. The physical frame number is the output and the virtual page number is the parameter of the mathematical function known as the page table. The output of this function may be used to convert a virtual address's virtual page field into a page frame field, resulting in a physical memory address merely virtual memory, not full virtualization. As a result, no virtual machines exist at this time. The structure of page tables becomes substantially more complex as a result of each virtual machine having its own virtual memory, necessitating the use of shadow or stacked page tables among other things. As we'll see, even without such mysterious arrangements, paging and virtual memory are rather sophisticated.

Speeding Up Paging

We've just gone through the foundations of virtual memory and paging. It's time to get into further detail about possible implementations. These two critical issues must be addressed by any paging system: The process of converting a virtual address into a physical place must be swift. If the virtual address space is enormous, the page table will also be quite large.

The virtual-to-physical mapping must be performed for every memory reference, which leads to the initial point. All commands must ultimately come from memory, and many of them make use of operands that are already stored there.

The second reason stems from the fact that virtual addresses of at least 32 bits are utilized by all modern computers, with 64-bit addresses becoming more typical for desktop and laptop systems. A 64-bit address space has more pages than you want to imagine, but a 32-bit address space has 1 million pages with, say, a 4-KB page size. For every million pages in the virtual address space, there must be one million entries in the page table. Moreover, remember that each process needs its own page table (because it has its own virtual address space). The need for large, rapid page mapping is a significant barrier to the design of computers. The simplest design (at least conceptually) consists of a single page table, which is composed of a group of fast hardware registers with a single entry for each virtual page that is indexed by the virtual page number. This architecture is shown in Fig. 3-10. When a process is started, the operating system fills the registers with a copy of the page table that is kept in main memory. The page table doesn't need any extra memory references while the process is running. The simplicity of this method and the absence of memory accesses during mapping are advantages. A drawback is that it is extremely expensive if the page table is large; it is often unworkable. Another defence is that performance would be completely ruined if each context switch required loading the whole page database. On the other hand, the page table could exist entirely in main memory. Hence, the hardware only needs one register that points to the start of the page table. This design allows the virtual-to-physical map to be changed during a context switch by reloading one register.

The operating system would load virtual page 8 at physical address 4096 and make two modifications to the MMU map, for instance, if it opted to evict page frame 1. The first step would be to flag the record on virtual page 1 as unmapped in order to block any upcoming visits to virtual addresses between 4096 and 8191. Therefore, when the trapped instruction is performed again, it will transfer virtual address 32780 to physical address 4108 (4096 + 12), by replacing the cross in virtual page 8's entry with a 1.

Accelerating Paging

The fundamentals of virtual memory and paging have just been covered. It's time to discuss potential implementations in greater depth. Every paging system must address these two crucial problems: It must be quick to translate a virtual address to a real location. The page table will be huge if the virtual address space is vast.

The first point results from the need to do the virtual-to-physical mapping for each memory reference. In the end, all instructions must originate from memory, and many of them also make references to operands there. As a result, each instruction requires one, two, or even more page references to tables. To prevent the mapping from turning into a significant bottleneck, the page table search must be completed in less than 0.2 nanoseconds if, for example, an instruction execution takes 1 n sec.

The second argument arises from the fact that 64-bit virtual addresses are becoming common for desktop and laptop computers, with virtual addresses of at least 32 bits being used by all current machines. A 32-bit address space contains 1 million pages with, let's say, a 4-KB page size, while a 64-bit address space has more pages than you want to think about. The page table must have one million entries for each of the million pages in the virtual address space. Furthermore, keep in mind that each procedure requires an own page table (because it has its own virtual address space). The necessity for big, quick page mapping is a major limitation on how computers are constructed. As illustrated in Fig. 3-10, the simplest architecture (at least theoretically) is a single page table made up of a collection of quick hardware registers with a single entry for each virtual page that is indexed by the virtual page number. A copy of the process' page table is stored in main memory, and when a process is launched, the operating system fills the registers with that copy. No additional memory references are required for the page table during process execution. The benefits of this strategy are that it is easy and needs no memory references dur- ing mapping. The fact that it is very costly if the page table is huge is a drawback; generally speaking, it is not feasible. Another argument is that speed would be utterly destroyed if the whole page table had to be loaded with each context transition. The page table might, on the other hand, reside wholly in main memory. The hardware therefore only requires a single register that refers to the page table's beginning. By reloading one register, this architecture enables the virtual-to-physical map to be updated during a context transition. Of course, it has the drawback of being very sluggish since each instruction requires one or more memory accesses to read page table information.

Lookaside Translation Buffers

Let's now examine commonly used techniques for managing massive virtual address spaces and for quickening paging. Most optimization methods begin with the assumption that the page table is in memory. Its design might have a significant effect on performance. Take a 1-byte instruction that duplicates one register to another, for instance. This instruction only uses one memory reference to retrieve the instruction when paging is not present. To access the page table while using paging, at least one extra memory reference is required. Making two memory references for every memory reference affects performance by 50% because the pace at which the CPU can get instructions and data from memory often determines how quickly programmes can be executed. No one would utilise paging in this scenario. This issue has been well-known to computer designers for a long time, and they have developed a remedy. Their technique is based on the fact that, rather than the other way around, most programmes prefer to make several references to a limited number of pages. As a result, only a tiny percentage of the page table items are extensively read, while the remainder are hardly ever utilised. A tiny hardware component for translating virtual addresses to physical addresses without using the page table has been added to computers as a solution. The system is sometimes referred to as an associative memory or a TLB (Translation Lookaside Buffer). It typically has eight entries, for example, and is located within the MMU. Seldom does it have more than 256 entries. The virtual page number, a bit that is set when a page is edited, the protection code (read/write/execute rights), and the physical page frame that the page is situated are all included in each entry. With the exception of the virtual page number, which is unnecessary in the page table, these values match exactly to the fields in the page table. Whether the entry is legitimate (i.e., in use) or not, it is indicated by another bit.

Application TLB Management

Up until this point, we have assumed that every computer with paged virtual memory has a TLB and hardware-recognized page tables. With this arrangement, the MMU hardware handles all aspects of TLB management and TLB failure handling. Operating system traps only appear when a page is not in memory. This supposition was accurate in the past. Nevertheless, the majority of this page management is done in software by several RISC computers, notably the SPARC, MIPS, and (the now-dead) HP PA. The operating system explicitly loads the TLB entries on these computers. Instead of searching the page tables for and retrieving the required page reference when a TLB miss occurs, the MMU simply issues a TLB fault and hands the issue off to the operating system. The procedure is to locate the page, delete an item from the TLB, replace it with the new entry, and resume the instruction that failed. Naturally, all of this must be accomplished in a small number of instructions due to the fact that TLB misses happen far more often than page faults.

Interestingly, software administration of the TLB turns out to be acceptable efficient if the TLB is relatively big (let's say, 64 entries), to lower the miss rate. The biggest benefit is a considerably simpler MMU, which frees up a lot of space on the CPU chip for caches and other performance-enhancing features. Uhlig et al. talk about the administration of the software TLB (1994). Long ago, a number of techniques were created to enhance the functionality of software TLB management machines. One strategy is to cut down on both TLB misses and the expense of a TLB miss when it does happen (Bala et al., 1994). The operating system may sometimes use intuition to determine which pages are likely to be utilised next and preload entries for them in the TLB in order to decrease TLB misses. For instance, it is quite probable that the server will need to start shortly when a client process on the same computer sends a message to a server process. With this knowledge, the system may check to see where the server's code, data, and stack pages are and map them in before they have a chance to result in TLB faults while processing the trap to do the send.

Whether in hardware or software, the standard procedure for handling a TLB miss is to access the page table and carry out the indexing operations necessary to find the page being referred. The issue with doing this search in software is that it's possible that the pages containing the page table aren't in the TLB, which would result in more TLB errors while the search is being processed. By keeping a sizable (e.g., 4-KB) software cache of TLB entries at a set place whose page is always preserved in the TLB, these errors may be minimised. TLB misses may be significantly decreased by the operating system by first verifying the software cache.

Knowing the differences between various miss types is crucial when using software TLB management. When the page being addressed resides in memory rather than the TLB, it is referred to as a soft miss. The TLB only has to be changed in this case. Disk I/O is not required. A soft miss typically requires 10 to 20 machine instructions to process and may be finished in a few nanoseconds. A hard miss, on the other hand, happens when the page itself is not remembered (and of course, also not in the TLB). The page must be brought in through a disc access, which might take a few milliseconds depending on the disc being utilised. An easy million times slower than a soft miss is a hard miss. A page table walk is the process of navigating the page table hierarchy to find the mapping. Indeed, it goes beyond that. A miss isn't simply soft or hard; it's both. A few misses are a little bit softer (or a little bit harsher) than others. Consider the scenario when the programme encounters a page fault because the page walk was unable to locate the page in the process' page table. Three options are available. First, the page could really be in memory, but not in the page table for this process. For instance, another process could have read the page from disc.

In such situation, we only need to map the page correctly in the page tables rather than re-accessing the disc. This is considered a small page error and is a rather light mistake. Second, if the page has to be loaded from disc, a significant page fault occurs. Lastly, it's conceivable that the application just visited an incorrect address, in which case the TLB doesn't even need a mapping to be inserted. The operating system normally terminates the programme with a segmentation fault in such scenario. Just in this instance did the software make a mistake. The hardware and/or operating system will automatically correct any additional issues, although at the expense of some performance.

Huge Memories Page Tables

Compared to the original page-table-in-memory system, virtual-to-physical address translation may be sped up using TLBs. Nevertheless, it's not the only issue we have to deal with. How to handle very huge virtual address spaces is another issue. We'll go through two approaches of dealing with them below.

Tables on inverted pages

Inverted page tables are a paging structure alternative to ever-increasing levels. They were first implemented by processors like the PPC, UltraSPARC, and Itanium (sometimes known as "Itanic" since it was not nearly as successful as Intel had intended). Instead than having one entry per page of virtual address space, this architecture has one entry per page frame in physical memory. For instance, an inverted page table only needs 1,048,576 entries when using 4 GB of Memory, 64-bit virtual addresses, and 4-KB pages. Which (process, virtual page) is present in the page frame is recorded in the entry.

Inverted page tables have a significant drawback: it becomes much more difficult to translate data from virtual to physical memory, at least when the virtual advertisement space is substantially greater than the real memory. The hardware can no longer locate the physical page by utilising p as an index into the page table when process n refers to virtual page p. Instead, it must look for an entry across the full inverted page table (n, p). Moreover, all memory references must be searched for, not just page faults. Making your system blindingly fast shouldn't involve searching a 256K table for every memory reference. The TLB may be used as a solution to this conundrum. Translation may proceed as quickly as with conventional page tables if the TLB has enough space to contain all of the frequently used pages. On a TLB miss, however, a software search of the inverted page table is required. A hash table hashed on the virtual address is one practical approach to carry out this search. Since there are so many page table entries even with a relatively high page size, inverted page tables are often used on 64-bit processors. For instance, 242 page table entries are required for 4-MB pages and 64-bit virtual addresses.

Algorithms for Page Replacement

In order to create space for the incoming page when a page fault occurs, the operating system must decide which page to evict (remove from memory). The page that has to be relocated must be rewritten to the disc in order to update the disc copy if it was changed while it was in memory. Although it is technically feasible to choose a random page to evict at each page fault, system performance is significantly improved if a seldom utilised page is selected. If a frequently utilized page is deleted, it will likely need to be rapidly put back in, adding additional overhead. Page repositioning methods have been the focus of much research, both theoretical and experimental. Following we shall discuss some of the more prominent ones. It is important to note that other aspects of computer architecture are also affected by the "page replacement" issue. As an example, the majority of computers feature one or more memory caches that include recently utilised 32-byte or 64-byte memory blocks. It is necessary to choose a block to remove when the cache is full. Similar to page replacement, but with a shorter time frame, is this issue (it has to be done in a few nanoseconds, not milliseconds as with page replacement). The reason for the reduced time scale is that main memory, which has no seek time and no rotational delay, is used to fill up cache block misses. Web servers serve as a second illustration. A set number of frequently accessed Web pages may be stored in the server's memory cache. Yet, a judgement must be made on which Web page to remove when the memory cache is full and a new page is referred. As the Web pages are never changed while they are in the cache, there is always a new copy "on disc," and the issues are identical to those for pages of virtual memory. Pages in main memory in a virtual memory system may either be clean or filthy.

Algorithm for Optimum Page Replacement

It is simple to define but difficult to use the optimal page replacement method. It proceeds as follows. Some set of pages are in memory at the time a page fault happens. The very next instruction will make a reference to one of these sites (the page containing that instruction). It's possible that other pages won't be mentioned for 10, 100, or even 1000 instructions. The number of instructions that will be carried out before a page is first referenced might be written on each page as a label. According to the best page replacement algorithm, the page with the highest label needs to be deleted. Removing the first page moves the page fault that will retrieve it as far into the future as feasible if one page will not be utilised for 8 million instructions and another page will not be used for 6 million instructions. Like individuals, computers also attempt to delay unpleasant occurrences as long as they can. This algorithm's unreliability is its sole drawback. The time of the system has no way of the system has no way of the system has no way of the system has no way of the system has no way of the system has no way of the system has no way of the system has no How can the system determine which work is the shortest? (We encountered a similar issue before with the shortest-job-first scheduling technique.) Nonetheless, by executing a programme on a simulator and recording all page references, optimum page replacement may be implemented on the second run by making use of the page-reference data gathered during the first run. This makes it easy to contrast the effectiveness of the best algorithm with that of realisable algorithms. For example, if an operating system performs only 1% worse than the best algorithm, searching for a better algorithm will only result in a 1% improvement. It should be made clear that this list of page references only pertains to the one programme that was just assessed and then with a single unique input in order to prevent any potential misunderstandings. This makes the page replacement method that was created from its unique to that one programme and set of input data. It serves no purpose in the fact that which means nothing in the fact that which means nothing in the fact that which means nothing in the fact that which means nothing in the fact that which means nothing in the fact that we shall examine methods below that are helpful for real-world systems.

Replacement of Not Recently Used Pages Algorithm

Most systems with virtual memory have two status bits, R and M, connected with each page, allowing the operating system to gather important page use information. When the page is referenced, R is set (read or written). When the page is written, M is set to (i.e., modified). Each page table item contains the bits. It is crucial to understand that these bits must be set by the hardware since they must be updated on each memory access. As soon as a bit is set to 1, it remains set to that value until the operating system resets it. If the hardware lacks these bits, the operating system's page fault and clock interrupt techniques may be used to imitate them. All of a process' page table entries are flagged as being out of memory when the process is started. A page fault will happen as soon as any page is referenced. The operating system then sets the R bit (in its internal tables), modifies the page table entry to refer to the right page, with mode read only, and resumes the instruction. The operating system may set the M bit and switch the page's mode to READ/WRITE if the page is later updated, causing another page fault. The following is a straightforward paging technique that uses the R and M bits. The operating system sets both page bits for each of a process's pages to 0 when it starts up. The R bit is periodically cleared to differentiate between pages that have recently been referenced and those that haven't (for example, on each clock interrupt). The operating system examines every page after a page failure and categorises them into four groups depending on the current values of their R and M bits:

Class 0: not updated or referenced. Class 1: not cited; altered.

Class 2: referenced, not changed. Class 3: changed and referenced.

While class 1 pages first seem to be impossibile, they may happen when a clock interrupt causes the R bit of a class 3 page to be cleared. Since this information is required to determine whether or not the page has to be rewritten to disc, clock interrupts do not clear the M bit. A class 1 page may be reached by clearing R but not M. The NRU (Not Recently Used) method randomly selects a page from the nonemptiest class with the lowest number. This approach assumes that a changed page that has not been referred in at least one clock tick (usually 20 msec) should be removed rather than a clean page that is often accessed. The fundamental appeal of NRU is that it is simple to comprehend, reasonably efficient to implement, and provides performance that, although undoubtedly subpar, could be sufficient.

Page replacement using the First-In, First-Out (FIFO) algorithm

The FIFO (First-In, First-Out) method is another low-overhead paging mechanism. Consider a supermarket with precisely k shelves, or items, to display to demonstrate how this works. One day, a manufacturer releases a brand-new convenience meal called instant, microwaveable, freeze-dried organic yoghurt. Our limited supermarket had to get rid of one outdated item since it was an instant hit in order to make room for it. One option is to locate the item that the store has been carrying the longest (i.e., something it started selling 120 years ago) and remove it on the justification that no one is interested in it any more. The previous one gets discarded, and the new one is added to the list's back. The same concept applies to page replacement algorithms. A list of all pages that are presently in memory is kept by the operating system, with the most recent arrival at the tail and the least recent arrival at the head. When a page error occurs, the page at the top of the list is dropped, and the new page is placed at the bottom. FIFO may eliminate moustache wax from shops, but it may also remove wheat, salt, or butter. The similar issue exists when it comes to computers: the oldest page could still be helpful. For of reasons, true FIFO is seldom used.

Algorithm for Second-Chance Page Replacement: Inspecting the R bit of the oldest page is a straightforward tweak to FIFO that solves the issue of not discarding a heavily utilised page. If it is 0, the page is instantly replaced since it is both outdated and unusable. The bit is cleared, the page is added to the end of the list of pages, and the load time is updated to reflect that the page has just entered memory if the R bit is set to 1. The hunt then goes on. Assume a page fault happens at time 20. The procedure began at time 0, when the oldest page, A, arrived. If A's R bit is cleared, it is removed from memory by being either abandoned or written to the disc (if it is filthy) (if it is clean). If the R bit is set, on the other hand, A is moved to the end of the list and its "load time" is changed to the current time (20). Likewise, the R bit is cleared. B continues to look for an appropriate page.

The Replacement Algorithm for the Clock Page: Second chance is a good method, but since it continuously moves pages about in its list, it is excessively inefficient. A better strategy is to maintain all the page frames on a clock-shaped circular list. The page being pointed to by the hand is examined when a page fault occurs. If the page's R bit is set to 0, it is removed from the clock, a new page is put in its place, and the hand is moved forward one position. R is cleared if it equals 1, and the hand moves on to the next page. Until a page is located with R 0, this procedure is repeated. The name clock for this method is not surprising.

Using the Least Recently Used (LRU) Page Replacement Algorithm

Based on the fact that pages that have been substantially utilised in the most recent instructions will likely be heavily used again soon, one may approximate the ideal method well. On the other hand, pages that have not been accessed in a while are likely to do so for a very long period. This concept provides a workable algorithm: when a page fault occurs, discard the page that hasn't been utilised in the longest. This method of paging is known as LRU (Least Recently Used). While LRU is theoretically possible, it is by no means inexpensive. A linked list of all the pages in memory must be kept, with the most recently used page at the front and the least recently used page at the back, in order to completely implement LRU. The list has to be updated after each memory reference, which presents a challenge. Even in hardware, finding a page in the list, removing it, and then pushing it to the front takes a lot of effort (assuming that such hardware could be built). LRU may also be implemented without specialised hardware in various methods. First, let's think about the simplest approach. For this technique to work, the hardware must have a 64-bit counter named C that is constantly increased. Additionally, a field big enough to hold the counter must be included in each page table entry. The current value of C is kept in the page table entry for the page that was just accessed after each memory reference. The operating system searches through all of the counters in the page table to locate the lowest one when a page failure occurs. The least frequently used page is that one.

CHAPTER 14

SOFTWARE LRU SIMULATION

Dr. Abhinav Tiwari, Assistant Professor,
Department of Decision Sciences,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - dr.abhinav_tiwari@cms.ac.in

While the preceding LRU technique can be implemented (in theory), very few computers have the necessary hardware. As an alternative, a software-based solution is required. The NFU (Not Frequently Used) algorithm is one option. Each page must have a programme counter that starts off at zero. The operating system examines every memory page at a clock interrupt. The R bit, which may be either 0 or 1, is added to the counter on every page. The numbers basically record how often each page has been cited. The page with the lowest counter is selected as a replacement when a page fault occurs. The fundamental issue with NFU is that it never forgets anything, much like an elephant. For instance, pages that were widely utilised during pass 1 of a multipass compiler may continue to have a high count long into following passes. In reality, the code pages for succeeding passes may always have lower counts than the pass-1 pages if pass 1 occurs to have the longest execution time of all the runs. As a result, the operating system will delete helpful pages rather than pages that are no longer needed. Thankfully, NFU can replicate LRU fairly effectively with a little adjustment. The change consists of two sections. The R bit is introduced after the counters have each been pushed rightward by one bit. Second, the rightmost bit is not added; the leftmost bit is.

The page with the lowest counter is eliminated when a page fault occurs. It is obvious that a page with four clock ticks of inactivity will have four leading zeros in its counter and, as a result, will have a lower value than a counter with three clock ticks of inactivity. This algorithm has two key distinctions from LRU. Both were referred to in the tick before the two clock ticks in which neither has been mentioned. If a page has to be changed, we should choose one of these two, says LRU. The issue is that we do not know which of them was last referred to between ticks 1 and 2. We have now lost the capacity to differentiate references happening earlier in the clock period from those occurring later since we are only recording one bit per time interval. The counters in ageing contain a limited amount of bits (8 bits in this case), which restricts its past horizon. This is the second distinction between ageing and LRU. Let's say that the counter values of two pages are both 0. We are only able to choose one of them at random. One of the pages may have last been mentioned nine ticks ago, while the other may have last been referenced a thousand ticks ago. There is no way for us to perceive that. In actual use, however, 8 bits are often sufficient if a clock tick lasts for around 20 msec. A page is presumably not that essential if it hasn't been referenced in 160 milliseconds.

Algorithm for Working Set Page Replacement

Processes are started up without any of their pages in memory in the purest form of paging. The operating system loads the page holding the first instruction as soon as the Processor attempts to retrieve the first instruction and encounters a page fault. The stack and other page errors for global variables often follow. The procedure eventually has the majority of the pages it requires and settles down to operate with just a small number of page defects. Demand paging is the name of this tactic since pages are only loaded when needed and not beforehand. In all, it is simple enough to create a test programme that repeatedly reads every page in a big address space and generates so many page faults that memory can't handle them all. Luckily, this is not how most processes operate. They display a locality of reference, which indicates that only a tiny portion of its pages are really referred to by the process during any given stage of execution. For instance, just a portion of the total number of pages are referred to during each pass of a multipacks compiler, and that portion is always different.

A process's working set is the collection of pages it is presently utilising (Denning, 1968a; Denning, 1980). While the complete working set is in memory, the process won't generate many errors up until it enters the next execution phase (e.g., the next pass of the compiler). As it takes a few nanoseconds to execute an instruction and 10 milliseconds to read in a page from the disc, the operation will run slowly if the available memory is insufficient to contain the whole working set. It will take a very long time to complete at a pace of one or two instructions per 10 msec. A programme is considered to be thrashing if it generates page faults every few instructions (Denning, 1968b). To give other programmes a chance at the CPU in a multiprogramming system, processes are often transferred to disc. The issue of what to do when a procedure is reintroduced emerges. There is technically no necessity for action. Page faults will be all the process does until its working set has loaded. The issue is that because it takes the operating system a few milliseconds to handle a page fault, having several page faults every time a process is loaded is sluggish and wastes a significant amount of CPU time. To ensure that the working set for each process is in memory before allowing it to execute, several paging systems attempt to keep track of it. The working set model is the name given to this method (Denning, 1970). It is intended to significantly reduce the rate of page errors. Preparing also refers to loading the pages before enabling procedures. It should be noted that the working set is dynamic. Programs seldom and typically cluster their address space references on a limited number of pages, as has long been understood. A memory reference may retrieve data or instructions or store data. There is a collection of all the pages utilised by the k most recent memory accesses at any given moment, t. The working set is this set, w(k, t) is a monotonically non-decreasing function of k because the k 1 most recent references must have consumed all the pages used by the k > 1 most recent references, as well as perhaps others. Since a programme cannot refer to more pages than its address space can hold and because few programmes utilise every page, the limit of w(k, t) as k increases is limited. The first quick increase of the curve and the subsequent considerably slower growth for high k are explained by the fact that most programmes only randomly visit a limited selection of pages, but that this set changes slowly over time. For instance, a programme running a two-page loop with data on four pages may refer to all six of those pages every 1000 instructions, yet the most recent reference to a different page might have been made a million instructions ago, during the startup phase. The working set's contents are not affected by the choice of k because of this asymptotic nature. To put it another way, the working set remains unaffected over a large range of k numbers. Since the working set doesn't change much over time, one may infer from the working set at the time the programme was last halted which pages will be necessary when it restarts. Before starting the procedure again, prepare by loading these pages. To implement the working set concept, it is required for the operating system to keep track of which pages are in the working set. Possessing this information also instantly leads to a viable page replacement algorithm: when a page failure occurs, discover a page not in the working set and evict it. We need a precise method of identifying which pages are in the working set in order to develop such an algorithm. By definition, the working set is the collection of pages utilised in the k most recent memory accesses (some authors use the k most recent page references, but the choice is arbitrary). Every working set algorithm requires a predetermined value of k to be implemented. The collection of pages utilised by the most recent k memory accesses is thus determined in a unique manner after each memory reference. Of fact, just because the working set has an operational definition does not guarantee that computing it effectively during programme execution is possible. An example of a shift register with every memory reference would be one of length k. The most recent page number should be inserted on the right after moving the register left by one place. The working set would be the set of all k page numbers in the shift register. The shift register's contents might theoretically be read out and sorted during a page fault. Then, duplicate pages might be deleted. The working set would be the outcome. This approach is never used because it would be prohibitively costly to retain the shift register and process it at a page fault.

Several approximations are employed as a substitute. Dropping the concept of counting back k memory accesses in favour of using execution time is one typical approximation. For instance, we might define the working set as the collection of pages utilised within the last 100 milliseconds of execution time rather than the set of pages used throughout the preceding 10 million memory accesses. In actuality, such a definition is considerably more practical and just as good. Keep in mind that each process only counts the time spent executing itself. So, for working set purposes, a process's time is 40 msec if it begins executing at time T and has received 40 msec of CPU time at actual time T + 100 msec. The amount of CPU time a process has really consumed since it began is typically dubbed its current virtual time. According to this estimate, a process's working set is the collection of pages it has consulted during the last number of virtual seconds. Let's now examine a page replacement technique built using the working set. Each entry includes (at least) two essential pieces of data: the R (Refer-enced) bit and the approximate time the page was last used. The other fields, such as the page frame number, the protection bits, and the M (Modified) bit, are represented by an empty white rectangle. The algorithm operates as shown below. The R and M bits are presumed to be set by the hardware, as was previously mentioned. Similar to this, it is expected that a periodic clock interrupt triggers the execution of code that clears the Referenced bit every clock tick. The page table is searched for an appropriate page to evict on every page fault. The R bit is inspected for each entry as it is processed. If it is 1, the page table's Time of Last Use column is updated to reflect the current virtual time, indicating that the page was being used when the problem occurred. time of, time of time of time slug slug slug out slug of slug out for span of time of,, the time,, and the time time to, and,, and that If R is zero, the page could be suitable for deletion since it hasn't been cited during the current clock tick. Its age (the current virtual time minus its Time of last usage) is calculated and compared to to determine whether or not it should be deleted. The page is no longer in the working set and is replaced by the new page if the age is higher than. The remaining entries are updated as the scan goes on. Nonetheless, the page is still in the working set if R is 0 but the age is less than or equal to. The page is momentarily spared, but it is recorded which page is the oldest (smallest value of Time since last usage). All pages must be in the working set if the whole table is scanned without producing a candidate for eviction. The oldest page is then removed if one or more pages with R = 0 were discovered. The worst-case scenario is that all pages have been referred during the current clock tick (and as a result, all have R = 1), thus one is randomly selected for removal, preferably a clean page, if one exists.

The fundamental working set approach is laborious because, for each page defect, the whole page table must be searched until a suitable candidate is found. WS Clock is a better approach that employs both the working set information and the clock algorithm as a foundation. It is frequently utilised in practise as a result of its ease of use and strong performance.

A circular list of page frames, as in the clock method, serves as the necessary data structure. This list is empty at first. The first page is added to the list after it has loaded. When additional pages are added, a ring-shaped pattern forms in the list. The basic working set algorithm's Time of Last Use field, the R bit (shown), and the M bit are all included in each entry (not shown).

The page that the hand is pointing to is inspected first, just as with the clock algorithm, if a page failure occurs. The page is not a good candidate for removal if the R bit is set to 1, since it has been utilised during the current tick. After that, the R bit is set to 0, the hand advances to the next page, and the procedure is repeated. It is not in the working set and a valid copy is present on the disc if the age is larger than and the page is clean. The page frame is simply claimed and the new page placed there. Nevertheless, if the page is unclean, it cannot be instantly claimed since there isn't a valid copy on the disc. The write to disc is scheduled in order to prevent a process transition, however the hand is advanced and the algorithm moves on to the next page. After all, there could be a previous, unmarked page that can be utilised right away. All pages may theoretically be scheduled for disc I/O in a single cycle around the clock. A restriction that permits a maximum of n pages to be written back may be established in order to reduce disc usage. No more writing would be planned once this threshold has been met. No writes have been planned, but at least one has been. In the first instance, the hand just continues to search for a blank page. Because one or more writes have been planned, one or more of them will finally finish, and the page will then be designated as clean. The first clean page that is found gets removed. The disc driver may reorganise writes in order to improve disc efficiency, therefore this page is not always the first write planned. In the second scenario, every page is part of the working set; otherwise, a write would have been scheduled at least once. The easiest course of action is to take ownership of any empty page and utilise it in the absence of further information. During the sweep, the position of a clean page might be tracked. The current page is picked as the victim and written back to disc if there are no clean pages available. The page that will be referred the furthest in the future is removed by the best algorithm. This approach cannot be utilised in reality since there is no way to tell which page this is. Yet, it may be used as a standard to compare other algorithms against. The R and M bits' states determine how the NRU algorithm splits pages into four classifications. The lowest-numbered class's random page is picked. While this method is simple to use, it is really basic. There are superior ones. By storing the pages in a linked list, FIFO maintains track of the order in which they were loaded into memory. The oldest page can therefore be easily deleted, however FIFO is a poor option since the oldest page can still be in use. A second chance update to FIFO ensures that a page is still being used before being removed. If so, the page is not affected. This tweak considerably improves the performance. Simply said, clock is another way to say second chance. While it takes a bit less time to run the algorithm, it has the same performance characteristics. While LRU is a great algorithm, it cannot be used without specialised gear. This hardware cannot be utilised if it is not readily accessible. A poor attempt to imitate LRU is NFU. It's not all that great. But, ageing may be effectively used and is a far better approximation to LRU. That is a wise decision. The working set is used by the last two algorithms. The working set approach performs quite well but has some implementation costs. A variation that provides high performance and is easy to implement is WSClock. Overall, ageing and WSClock are the two best algorithms. They are founded on the working set and LRU, respectively. Both have effective paging performance and may be used effectively. There are a few additional useful algorithms, but these two are likely the most significant in actual use.

Paging System Design Concerns

You need to know a lot more in order to build a system and make it function properly. That is comparable to the difference between being a competent chess player and just understanding how to move the rook, knight, bishop, and other pieces. To obtain the most performance out of a paging system, operating system designers must carefully consider a number of additional factors, which we shall examine in the sections that follow.

Charge Control

It is possible for the system to crash even with the best page replacement technique and efficient global allocation of page frames to processes. Thrashing is really anticipated whenever the total working sets of all processes exceed the memory available. The PFF algorithm's indication that certain processes require more memory but none need less memory is one sign of this problem. There is currently no method to increase RAM for processes without negatively affecting other processes. Eliminating certain procedures temporarily is the only true option. Swapping some of the processes to the disc and freeing up all the pages they are holding is an effective technique to lower the number of processes that are contending for memory. One process, for instance, may be switched to disc and its page frames distributed among other thrashing processes. The system may function in this manner for a time if the thrashing ceases. If the thrashing continues, another process must be replaced, and so on, until it stops. Hence, swapping may still be required even with paging; the difference is that swapping is now used to lower the potential demand for memory rather than to reclaim pages. In two-level scheduling, some processes are placed on disc and a short-term scheduler is used to schedule the remaining processes, switching processes out to reduce the pressure on memory is similar to that. It is obvious that the two concepts may be integrated, with just enough processes changed to keep the page-fault rate within acceptable bounds. Certain processes are periodically loaded from disc, while others are switched off. Nevertheless, the level of multiprogramming should also be taken into account. The CPU may be idle for long periods of time when there are not enough processes in main memory. This argument makes the case for taking into account a process's characteristics, such as whether it is CPU constrained or I/O bound, in addition to its size and paging rate when determining which process to swap out.

Page Dimensions

The operating system has control over a number of parameters, including page size. The operating system may simply treat page pairs 0 and 1, 2 and 3, 4 and 5, and so forth as 8-KB pages by always allocating two consecutive 8192-byte page frames for each, even though the hardware, for example, was intended for 4096-byte pages. A number of conflicting elements must be balanced in order to determine the ideal page size. There is thus no overarching ideal. First, there are two arguments in favour of lower page sizes. A text, data, or stack segment picked at random won't take up an exact number of pages. On average, the last page will be vacant for half of it. The excess space in that page is wasted. The term "internal fragmentation" refers to this waste. With n segments in memory and a page size of p bytes, internal fragmentation will consume np/2 bytes. This argument favours a modest page size. When we consider a programme with eight successive stages of 4 KB each, another case for tiny page sizes is made clear. A 32-KB page size means that the software must always have 32 KB available. It simply need 16 KB for a page size of 16 KB. It only needs 4 KB at any one time for pages that are 4 KB or less in size. A big page size will often result in greater memory wastage than a small page size. On the other side, programmes will need numerous pages and a huge page table if the pages are short. Just four 8 KB pages, but 64 512 byte pages, are required for a 32 KB application. A page is typically transferred to or from the disc at a time, with the seek and rotational delays taking up the majority of the time. As a result, transferring a tiny page takes virtually as long as transferring a big page. 64 512-byte pages could load in 64 10 ms, whereas four 8 KB pages could load in 4 12 ms.

Public Pages

Sharing is another difficulty with design. It is typical for several users to execute the same programme simultaneously in a big multiprogramming system. Several apps that utilise the same library can be running simultaneously for even a single user. Sharing the pages and avoiding having two copies of the same page in memory at once is undoubtedly more efficient. The inability of certain sites to be shared is one issue. Pages that are read-only, such programme text, may be shared specifically, but sharing data pages is more challenging. If separate I- and D-spaces are enabled, sharing programmes across two or more processes using the same page table for their I-space but distinct page tables for their D-spaces is quite simple. Page tables are often separate from the process table in an implementation that enables sharing in this manner. Then, in the process table of each process, there are two pointers: one to the I-space page table and the other to the D-space page table. These pointers are used by the scheduler to find the appropriate page tables and set up the MMU when it selects a process to execute. Processes can exchange programmes (or sometimes libraries) even without distinct I- and D-spaces, although the technique is more difficult. A issue arises with the shared pages when two or more processes share certain code. Assume that processes A and B are sharing the editor's pages and executing the editor simultaneously. If the scheduler chooses to remove A from memory, evicting all of its pages and putting another programme in the vacant page frames would require B to produce a lot of page faults in order to put them back in. Similar to this, it is crucial to be able to determine if the pages are still in use after A terminates so that their disc space is not unintentionally released. As it is often extremely costly to search through all of the page tables to determine if a page is shared, special data structures are required to keep track of shared pages, particularly when the sharing is restricted to a single page (or run of pages) rather than a whole page table. While it is more difficult than sharing code, exchanging data is still doable. Particularly in UNIX, the parent and child are needed to communicate both programme text and data following a fork system call. Giving each of these processes its own page table and having both of them link to the same set of pages is a common practise in a paged system. As a result, no page copies are made at fork time. All of the data pages are, however, mapped as READ ONLY into both processes. This state may persist as long as both processes are merely reading their data, not altering it. The read-only protection is violated as soon as either process alters a memory word, which triggers an operating system trap. The problematic page is then copied, giving each process access to its own private copy. Now that both copies are configured to READ/WRITE, any additional writes to either copy may be made without encountering any traps. This method eliminates the requirement for copying all programme pages and other pages that are never updated. It is only necessary to copy the data pages that have been really edited. This strategy, known as copy on write, enhances performance by minimising copying.

Shared Libraries

More levels of sharing beyond individual pages are possible. Most operating systems will automatically share all the text pages if a programme is launched again so that only one copy is kept in memory. As text pages are always read-only, nothing is wrong. Each process may get a private copy of the data pages, or they may be shared and designated as read-only, depending on the operating system. A private copy will be created for each process that alters a data page, or copy on write will be used. There are several huge libraries utilised by numerous processes in contemporary systems, such as various I/O and graphics libraries. These libraries would become much more obese if they were statically bound to every executable application on the disc.

An alternative strategy is to utilise shared libraries (which are called DLLs or Dynamic Link Libraries on Windows). Let's first analyse conventional linking to help understand the concept of a shared library. When a programme is linked, one or more object files and potentially certain libraries are specified in the linker's command. For example, the UNIX command ld *.o -lc -lm links all the.o (object) files in the current directory and then scans the two libraries /usr/lib/libc.a and /usr/lib/libm.a. Undefined externals are functions that are called in object files but are not existent there (like printf), and are searched for in libraries. They are added to the executable binary if they are discovered. They also create undefined externals for any functions they call that are not yet existent. For instance, the linker will search for and include write if it is not already present since printf depends on it. After the linker is finished, a disk-based executable binary file containing all the required functions is created. Functions that are available in the libraries but are not used are excluded. All the functions required by the programme are available when it is loaded into memory and run. Imagine that most apps need 20–50 MB of graphics and user interface features. As the system wouldn't know it could share them, statically linking hundreds of applications with all these libraries would waste a significant amount of space on the disc and RAM when they were loaded. Shared libraries have a role in this. Instead of include the actual function called when a programme is linked with shared libraries (which are somewhat different from static ones), the linker instead inserts a short stub procedure that binds to the called function at run time. Shared libraries may be loaded when the application is launched or when their functions are first invoked, depending on the system and configuration parameters. Of course, there is no need to load the shared library again if another application has already done so; that is its whole purpose. Shared libraries provide another significant benefit in addition to reducing the size of executable files and freeing up memory: if a function in a shared library is modified to fix a defect, the programmes that use it do not need to be recompiled. The previous binaries still function. This functionality is particularly crucial for commercial software since the buyer does not get the source code. For instance, if Microsoft discovers and corrects a security flaw in a common DLL, Windows Update will download the updated DLL and overwrite the outdated one, and all programmes that rely on the DLL will use the updated version by default the next time they are run. However there is a little issue with shared libraries that has to be fixed. Here, two processes are shown sharing a 20 KB library (assuming each box is 4 KB). Yet in each process, the library resides at a different location, probably because the programmes are not of the same size. The library begins at address 36K in process 1 and at address 12K in process 2. Assume that the library's first function must first jump to address 16 in order to complete its task. The jump (in process 1) might be to virtual address 36K + 16 if the library weren't shared since it could be moved there instantly as it loaded. Since all the pages are mapped from virtual to physical addresses by the MMU hardware, it is important to note that the actual location in the RAM where the library is placed is irrelevant. Moving somewhere else on the fly won't work, however, since the library is shared. After all, the jump instruction has to move to 12K + 16, not 36K + 16, when process 2 calls the first function (at location 12K). The little issue is this. Using copy on write to produce new pages for each process sharing the library and moving them as they are created might be one solution, however this would obviously undermine the point of sharing the library. A more effective approach is to generate shared libraries with a specific compiler flag instructing the compiler to not write any instructions that utilise absolute addresses. Instead, only commands that utilise relative addresses are used. For instance, the command to "jump ahead (or backward) by n bytes" is almost always present (as opposed to an instruction that gives a specific address to jump to). No matter where the shared library is located inside the virtual address space, this instruction operates as intended. The challenge may be resolved by staying away from absolute addresses. The term "position-independent code" refers to code that only utilises relative offsets.

File Maps

Shared libraries are really a specific kind of memory-mapped files, a more general feature. A process may make a system call to map a file onto a piece of its virtual address space, which is the goal here. In the majority of implementations, no pages are brought in at the time of mapping; instead, pages are brought in on demand, one page at a time, with the disc file serving as the backup store, as they are touched. The changed pages are written back to the file on disc when the process ends or manually umaps the file. An alternate I/O approach is offered by mapped files. It is possible to access the file in memory as a large character array rather than doing reads and writes. This paradigm may be more practical in some circumstances for programmers. Shared memory may be used to facilitate communication between programmes that concurrently map onto the same file. When the second process reads from the portion of its virtual advertisement spaced mapped onto the file, writes made by one process to the shared memory are instantly apparent. This method therefore offers a high-band- width conduit between processes and is widely employed as such (even to the extent of mapping a scratch file). Now it should be evident that shared libraries may utilise this approach if memory-mapped files are accessible.

Cleaning Procedure

Paging performs best when there is a large number of available free page frames that can be used when page errors happen. Before a new page can be brought in if every page frame is full and updated as well, an old page must first be written to disc. Paging systems often contain a background process called the paging daemon that sleeps most of the time but is regularly woken to check on the condition of memory in order to guarantee a bountiful supply of free page frames. If there aren't enough free page frames, it starts choosing which pages to evict using a page replacement method.

These pages are written to disc if they have changed after being loaded. In either case, the page's previous contents are stored in memory. One of the evicted pages may be recovered by removing it from the pool of available page frames in the event that it is required once more before its frame has been overwritten. Performance is improved by keeping a supply of page frames on hand as opposed to consuming all available memory and then frantically searching for a frame when it is required.

At the absolute least, the paging daemon makes sure that every free frame is clean, preventing the need for hasty disc writing when it is necessary. Using a two-handed clock is one method to put this cleaning strategy into practise. The paging daemon is in charge of controlling the front hand. It advances the front hand and writes the dirty page back to disc when it points to one. It is only advanced when it leads to a clear page. To change pages, the back hand is employed as in the conventional clock method. Only now, thanks to the paging daemon's efforts, is there a higher chance that the back hand will land on a clean page.

Interface for Virtual Memory

Our whole explanation up to this point has been predicated on the idea that processes and programmers only perceive a large virtual address space on a machine with a small(er) physical memory. It is true for many systems, but in certain sophisticated systems, programmers have considerable influence over the memory map and may make unconventional use of it to improve programme behaviour. We'll take a quick look at a couple of them in this section. Giving programmers control over their memory map is one way to enable memory sharing across two or more processes sometimes in artful ways. It could be conceivable for one process to tell another process the name of a memory area so that the other process can also map it in if programmers can assign regions of their memory names. High bandwidth sharing is made feasible when two (or more) processes share the same pages; one process writes to the shared memory while another reads from it. De Bruijn gives a complex illustration of such a communication route (2011). Another way to construct a high-performance message-passing system is using page sharing. Data are often transferred from one address space to another at a significant expense when messages are passed. Messages may be transmitted across processes if they have control over their page maps by having the transmitting process unmap the page(s) holding the message and the receiving process map them in. Here, just the page titles need to be copied—not the whole content. Distributed shared memory is yet another cutting-edge memory management method (Feeley et al., 1995; Li, 1986; Li and Hudak, 1989; and Zekauskas et al., 1994). The goal is to enable the network-based sharing of a set of pages by several processes, maybe as a single shared linear address space, but not necessary. A page fault occurs when a process refers to a page that isn't presently mapped in. The page fault handler then finds the machine holding the page and sends it a message requesting it to unmap the page and deliver it over the network, depending on whether it is in the kernel or user space. The faulting instruction is restarted when the page loads after being mapped in. We'll investigate shared memory that is distributed.

Issues with Implementation

The main theoretical techniques for second chance vs ageing, local versus global page allocation, and demand paging versus preparation must be chosen by virtual memory system implementers. But they must also be aware of a number of challenges with actual implementation. We'll look at a few of the typical issues in this part, along with potential remedies.

Operating System Connection to Pager

The operating system has paging-related tasks to execute four times: at process creation, during process execution, during page faults, and during process termination. Now that we have a quick look at each of them, we can see what needs to be done. When a new process is formed in a paging system, the operating system needs to calculate how big the programme and data will be (initially) and generate a page table. The page table has to be initialised and memory space provided for it. When a process is switched out, the page table need not be present, but it must be in memory while the process is active. Moreover, space must be allotted in the swap region of the disc so that a page may have a somewhere to go when it is switched out. Also, programme text and data must be initialised in the swap region so that the pages may be brought in when the new process begins to experience page faults. Some systems immediately page the programme text from the executable file, reducing startup time and storage space. The process table must also include information about the page table and swap space on disc. The MMU must be reset for the new process when one is scheduled for execution, and the TLB must be flushed to remove any remnants of the previously running process. The page table for the new process must be updated, often by copying it or a reference to it to some hardware register (s). To initially lower the amount of page faults, it is optional to bring part or all of the process's pages into memory (e.g., it is certain that the page pointed to by the programme counter will be needed). The operating system must scan hardware registers once a page fault occurs to identify which virtual address was at fault. This data must be used to determine which page is required and find that page on the disc. The new page must then be inserted into an open page frame, maybe displacing an existing page. The required page must then be read into the page frame. The programme counter must then be backed up in order to point to the malfunctioning instruction and allow that instruction to resume execution. The operating system must release the page table, the pages, and the disc space that the pages take up when they are stored on disc when a process terminates. The pages in memory and on storage that are shared with other processes can only be freed once the last process that was utilising them has ended.

Handling page errors

We can now go into great depth about what occurs on a page fault.

The following is the order of the events: The hardware saves the programme counter on the stack via trapping to the kernel. On most devices, certain CPU registers hold some data on the status of the currently executed instruction. To prevent the operating system from erasing the general registers and other volatile data, an assembly-code process is initiated to store it. This technique launches the operating system. An operating system page fault is identified, and an attempt is made to identify which virtual page is required. The system checks to determine whether the virtual address that produced the fault is genuine and that the protection is compatible with the access once the virtual address that caused the fault is known. If not, a signal is delivered or the procedure is terminated. The system checks to determine whether a page frame is free if the advertisement is legitimate and no protection fault has occurred. If no frames are available, the victim is chosen using the page re- location method. When a context transition occurs, the faulting process is suspended and another one is allowed to continue until the disc transfer has finished if the page frame chosen is dirty. The frame is in any case designated as busy to prevent unauthorised usage of it. The operating system searches up the disc address where the required page is located and schedules a disc operation to bring it in as soon as the page frame is clear (either immediately after it is written to disc or later). The faulting process is still halted while the page loads, and if another user process is available, it is started. The page tables are changed to reflect the location of the page when the disc interrupt signals that it has arrived, and the frame is then flagged as being in the usual condition. The programme counter is reset to point to the malfunctioning instruction, which is then backed up to the condition it was in before it started. The operating system resumes the (assembly-language) routine that called the faulting process once it has been scheduled. In order to continue running, this function reloads the registers and other state information and switches back to user space.

Backup Instructions

When a programme refers to a page that is not in memory, the instruction that is responsible for the error is interrupted, and an operating system trap is set off. The operating system must resume the instruction triggering the trap after it has obtained the necessary page. It's simpler to say than to accomplish. Consider a CPU having instructions that have two locations, such as the Motorola 680x0, which is often used in embedded systems, to understand the nature of this issue at its worst. The operating system must ascertain the location of the instruction's initial byte in order to resume it. Depending on which operand failed and how the CPU's microcode was implemented at the moment of the trap, the programme counter's value will change.

The instruction word and two operand offsets are three memory accesses made by an instruction commencing at address 1000. The programme counter at the time of the error may have been 1000, 1002, or 1004 depending on which of these three memory visits resulted in the page fault. The operating system often struggles to identify clearly where an instruction started. If the programme counter is at 1002 at the time of the error, the operating system is unable to determine whether the word in 1002 is an operand address or a memory address connected to an instruction at 1000.

Even though this issue is bad, it might have been worse. Certain 680x0 addressing modes include auto incrementing, which implies that as a result of carrying out the instruction, one (or more) registers are increased. It is also possible for instructions in auto increment mode to fail. The operating system must decrease the register in software before continuing the instruction if the increment is performed before the memory reference, depending on the specifics of the microcode. It's also possible that the auto increment was performed after the memory reference, in which case the operating system cannot reverse it since it was not performed at the time of the trap. There is also auto decrement mode, which has a similar issue. Depending on the instruction and the CPU type, the precise information on whether auto increments and auto decrements were performed before the relevant memory accesses may vary. Thankfully, the CPU designers on certain computers offer a workaround, often in the shape of a secret internal register into which the programme counter is duplicated right before each instruction is carried out. Moreover, these machines could feature a second register that indicates which registers have previously undergone automatic incrementing or decrementing as well as by how much. With this knowledge, the operating system may obstinately reverse all the consequences of the malfunctioning command in order to restart. If this data is not accessible, the operating system must go through a number of hoops to determine what went wrong and how to fix it. It's as though the operating system authors were left to handle the issue after the hardware designers gave up trying to find a solution.

CHAPTER 15

DISC MANAGEMENT

Vaibhav Goutham Suresh, Assistant Professor,
Department of General Management,
CMS Business School, Jain Deemed to-be University, Bangalore, Karnataka, India.
Email Id: - vaibhavsuresh@cms.ac.in

I/O and virtual memory have nuanced interactions. Take into account a process that has just used a system function to read from a file or device into a buffer that is located inside its address space. The process is put on hold while waiting for the I/O to finish, while another one is given permission to run. A page fault occurs for this other process. A small but not zero possibility exists that the page holding the I/O buffer will be selected to be deleted from memory if the paging mechanism is global. Removing an I/O device will result in some of the data being written over the newly loaded page and part of the data being placed in the buffer where it belongs if a DMA transfer is already being performed to that page. To prevent their removal, pages in memory that are actively performing I/O may be locked. Pinning a page in memory is another term for locking it. Doing all I/O to kernel buffers and afterwards copying the data to user pages is an alternative option.

Supporting Shop

We looked at how a page is chosen for removal in our discussion on page replacement algorithms. About where it is placed on the disc when it is paged out, not much has been stated. Let me now outline some of the challenges relating to disc management. Having a dedicated swap partition on the disc, or better yet, having it on a different disc from the file system (to balance the I/O load), is the simplest technique for allocating page space on the disc. This is how most UNIX systems operate. Due to the absence of a typical file system on this partition, there is no cost associated with converting file offsets to block addresses. Instead, all block numbers are related to the partition's beginning. This swap partition is empty when the system boots and is stored in memory as a single entry that specifies its size and origin. The simplest plan involves reserving a portion of the partition area the size of the first process when it is begun and reducing the remaining space by that amount. Their disc space is released after they're done. Each process has a disc address for the space on the swap partition where its image is stored, known as the swap area. The process table contains these data. Just add the offset of the page inside the virtual address space to the start of the swap area to get the address to write a page to. Nevertheless, the swap area has to be setup before a process can begin. To make it possible to bring it in as required, one method is to transfer the complete process picture to the swap area. The option is to load the full process in memory and let it be paged out as required. This straightforward paradigm, however, has a flaw: once they begin, processes may grow in size. The programme text is typically constant, although the data area and stack may both expand on occasion. So, it may be preferable to set aside distinct swap areas for the text, data, and stack and let each of these regions to include several disc chunks. The other extreme is to allocate no space in advance, deallocate it when a page is switched back in, then allot disc space for each page when it is swapped out. Hence, programmes running in memory save up swap space. The drawback is that in order to maintain track of each page on disc, a disc address must be kept in memory. In other words, there has to be a table for each process that indicates the location of each page on the disc.

Separation of Policy and Mechanism

By separating policy from mechanism, one may manage the complexity of any system. By having the majority of the memory manager operate as a user-level process, this idea may be applied to memory management. The explanation that follows is based on the first time such a distinction was made, which was in Mach.

The MMU handler, which is machine-dependent code and has to be rebuilt for each new platform the operating system is ported to, contains all the specifics of how the MMU functions. The majority of the paging mechanism is contained in the machine-independent code known as the page-fault handler. The external pager, which operates as a user process, heavily influences the policy. The external pager is informed when a process starts up so that it may build up the process' page map and, if necessary, assign the required backup store on the disc. The external pager is once again informed as the process continues since it could map additional items into its address space. After the operation has begun, a page fault might occur. The fault handler determines which virtual page is required and notifies the external pager of the issue in a message. The required page is then copied to a section of the external pager's address space after being read from the disc. The problem handler is then informed of the page's location. The fault handler then requests the MMU handler to insert the page into the user's address space at the appropriate location after uncapping it from the external pager's address space. The user process may then be started again. Where the page replacement method is placed in this implementation is up for debate. The external pager would be the clearest place for it, however there are several drawbacks to this strategy. The main one is that not all of the pages' R and M bits are accessible to the external pager. Many paging algorithms make use of these bits. So, either a method is required to transmit this information to the external pager, or the page replacement technique has to be included in the kernel. In the latter scenario, the fault handler notifies the external pager of the page it has chosen for eviction and supplies the data, either by mapping it into the address space of the external pager or putting it in a message. The external pager stores the information on disc in any case. This implementation's key benefits are more modular code and more flexibility. The biggest drawback is the additional cost of repeatedly crossing the user-kernel barrier and the expense of sending multiple messages among the system's components. The topic is now quite divisive, but as computers get faster and faster and software becomes more sophisticated, most implementers will likely be willing to trade some speed for more dependable software in the long term.

Segmentation

As the virtual addresses range from 0 to some maximum address, one address after another, the virtual memory that has been explained thus far is one-dimensional. Having two or more distinct virtual address spaces may be significantly better than having just one for various issues. While the compilation process continues, a compiler, for instance, builds up a number of tables, maybe including The source text being kept for the printed listing (on batch systems), The variable names and characteristics are shown in the symbol table the list of all utilized floating-point and integer constants, The stack utilized by the compiler for procedure calls, the parse tree, which contains the syntactic analysis of the programme.

As compilation moves on, each of the first four tables continuously expands. The last one changes size during compilation in unforeseen ways. These five tables would need to be given contiguous blocks of virtual address space in a one-dimensional memory. Think about what happens if a programme has a typical quantity of everything else but a lot more variables than usual. There may not be much room in the symbol table's portion of the address space, but there could be enough in the other tables. In the same manner that virtual memory takes the bother out of structuring the programme into overlays, a method is required to release the programmer from having to handle the growing and contracting tables.

The provision of the computer with several entirely distinct address spaces, or "segments," is an easy and broadly applicable method. A linear list of addresses, beginning at 0 and increasing in value up to some limit, makes up each segment. Each segment's length may range from zero to the highest address permitted. The lengths of individual parts may and often do vary. Also, segment lengths might alter while an execution continues. As something is put onto a stack segment, its length may grow, and when something is taken off of a stack segment, its length may shrink. Several segments may expand or contract independently of one another since each segment has its own address space. There is nothing else in a segment's address space that a stack may collide with if it requires extra address space to expand. Of course a segment may fill up, but because segments are often rather big, this seldom happens. The software has to provide a two-part address, a segment number, and an address inside the segment in order to define an address in this segmented or two-dimensional memory.

Thus, it is important to stress that a segment is a logical object, one that the programmer is aware of and employs. A segment may include a method, an array, a stack, or a group of scalar variables, although it often does not have a combination of several kinds. Along with making it easier to manage expanding or contracting data structures, a segmented memory provides further benefits. The linking of processes that have been compiled independently is much simplified if each procedure takes up a distinct segment with address 0 as its beginning address. A procedure call to the procedure in segment n will utilise the two-part address (n, 0) to address word 0 after all the procedures that make up a programme have been built and linked together (the entry point).

No other procedures need to be updated if the procedure in segment n is afterwards modified and recompiled (because no beginning addresses have been adjusted), even if the new version is bigger than the previous one. The methods are crammed closely next to one another in a one-dimensional memory, with no ad- dress space in between. As a result, altering the size of one operation might change the beginning address of every other (unrelated) procedure in the segment. As a result, any processes that make use of any of the shifted procedures must be updated to include their new beginning addresses. This technique might be expensive if a software has hundreds of processes. Segmentation also permits exchanging methods or data across many operations. The shared library is a prevalent illustration. Large graphics libraries are often built into almost every application on contemporary workstations that operate sophisticated window systems. The graphics library doesn't have to be in the address space of every process in a segmented system since it may be placed in a segment and shared by many processes. Pure paging systems can also support shared libraries, although doing so is more difficult. Some systems do this by, in essence, imitating segmentation. Different segments may have various types of security since each segment creates a logical item that programmers are aware of, such as a method or an array. It is possible to designate a procedure segment as execute-only, which prevents efforts to read from or put data into it. Attempts to jump to a floating-point array will fail since it can only be declared as read/write rather than execute.

Paging-based segmentation: MULTICS

It could be difficult or perhaps impossible to retain the whole segment in main memory if it is huge. As a result, it is suggested to paginate them so that only the pages of a segment that are truly required must be included. Paged segments are supported by a number of important systems. The first one, MULTICS, will be discussed in this section. The Intel x86 up to the x86-64 will be covered in the next one, which is a more recent one. One of the most important operating systems ever, MULTICS had a significant impact on a variety of subjects, including UNIX, the x86 memory architecture, TLBs, and cloud computing. It was developed at M.I.T. as a research project and launched in 1969. After 31 years of operation, the last MULTICS system was shut down in 2000. Hardly no other operating systems have survived so long more-or-less unchanged. Although other operating systems with the same name have existed for a similar amount of time, just the name and the fact that Windows 8 was created by Microsoft are shared by Windows 1.0. Most importantly, the concepts introduced in MULTICS remain relevant and helpful now just as they did when the initial article was published in 1965 (Corbato and Vysotsky, 1965). For this reason, we will now take a brief look at the virtual memory architecture, which is the most novel feature of MULTICS. The Honeywell 6000 computers and its offspring supported MULTICS, which gave each application access to a virtual memory of up to 218 segments, each up to 65,536 (36-bit) words long. The MULTICS designers decided to achieve this by treating each segment as a virtual memory and page it, combining the benefits of segmentation and paging (constant page size, no need to store the whole segment in memory when just a portion of it is being utilised) (ease of programming, modularity, protection, sharing). There was a segment table for each MULTICS programme, with one description for each segment. The segment table was itself a segment and was paginated since there may be over a quarter of a million items in it. Whether a segment was in main memory or not was indicated by the segment descriptor. The segment was deemed to be in memory and its page table was in memory if any portion of the segment was in memory.

Paging-based segmentation using the Intel x86

The virtual memory system of the x86 up to the x86-64 shared several characteristics with MULTICS, notably the inclusion of both segmentation and paging. The x86 features 16K separate segments, each storing up to 1 billion 32-bit words, as opposed to MULTICS' 256K independent segments, each of which could carry up to 64K 36-bit words. Even though there are fewer segments, the greater segment size is considerably more significant since many applications need large segments but few require more than 1000 segments. Segmentation is no longer supported as of x86-64, with the exception of legacy mode. While certain relics of the previous segmentation techniques are still present in x86-native 64's mode, they no longer fulfil the same function and do not provide genuine segmentation. This is mainly for compatibility reasons. Nonetheless, the x86-32 still has everything, and it is this Processor that we will talk about in this part. Two tables, the LDT (Local Descriptor Table) and the GDT, make up the x86 virtual memory's core (Global Descriptor Table). There is a single GDT that is shared by all of the applications on the computer, but each programme has its own LDT. The GDT covers system segments, including the operating system, while the LDT describes segments local to each application, including its code, data, stack, and so forth. An x86 software must first load a segment selector for that segment into one of the machine's six segment registers before it may access that segment. The CS register and DS register respectively store the selectors for the code and data segments during execution. The significance of the other segment registers is lower. A 16-bit number represents each selection.

The segment's locality or globality may be determined by one of the selector bits (i.e., wheth- er it is in the LDT or GDT). Each of these tables is limited to carrying 8K segment descriptors because to the 13 additional bits that identify the LDT or GDT entry number. The two more bits, which have to do with protection, will be discussed later. Descriptor 0 is prohibited. To signal that a segment register is not presently accessible, it may be securely put into the register. If employed, it creates a trap. The matching descriptor is acquired from the LDT or GDT and placed in microprogram registers so it may be accessible quickly at the time a selector is put into a segment register. A descriptor is made up of 8 bytes, as shown in Fig. 3-39, and contains information such the segment's base address and size. The selector's format was well crafted to make finding the description simple. Based on selector bit 2, the LDT or GDT is first chosen. The 3 low-order bits are then set to 0 once the selector has been transferred to an internal scratch register. Lastly, to provide a direct reference to the descriptor, the address of the LDT or GDT table is appended to it. Selector 72, for instance, relates to GDT entry 9, which is stored at location GDT + 72. Let's now go through the process of turning a (selector, offset) pair into a physical address. The whole description for that selection may be found in the microprogram's internal registers as soon as it understands which segment register is being utilised. A trap happens if the segment doesn't exist (selector 0) or is presently paged out.

The hardware then checks the offset against the segment's end using the Limit field, and if it is, a trap is also set off. As just 20 bits are available, a different approach is used to represent the segment size that logically should be represented by a 32-bit field in the descriptor. The Limit field contains the precise segment size, up to 1 MB, if the Gbit (Granularity) field is set to 0. If it is 1, the segment size is given in pages rather than bytes in the Limit field. 20 bits are sufficient for segments up to 232 bytes on a 4 KB page.

The x86 then adds the 32-bit Base field in the descriptor to the offset to create what is known as a linear address, assuming that the segment is in memory and that the offset is within range. Since the Base field in the 286 only has 24 bits, it is divided into three parts and distributed across the descriptor. Each segment may, in fact, begin at any location inside the 32-bit linear address space thanks to the Base field. If paging is deactivated (by a bit in a global control register), the linear address is interpreted as the physical address and transmitted to the memory for the read or write. We now have a pure segmentation system with no paging, and each segment's base address is provided in its descriptor. The fact that segments cannot be stopped from overlapping is likely due to the difficulty and length of time required to confirm that they are all separate. The linear address is understood as a virtual address and mapped into the physical address using page tables, just as in our prior instances, if paging is enabled. A two-level mapping is utilised to lower the page table size for short segments since the only significant challenge is that a segment may have 1 million pages with a 4-KB page and a 32-bit virtual address. The page directory for each running application has 1024 32-bit entries. It is situated in a location that a global register points to. Each item in this directory directs the user to a page table with 1024 32-bit entries as well. Page frames are indicated by the page table entries. A single page table manages 4 gigabytes of memory since each page table includes entries for 1024 4-KB page frames. A page directory for a segment less than 4M bytes will contain a single item that serves as a reference to the segment's only page table. Instead of the million pages required by a one-level page table, the overhead for short segments is reduced to only two pages in this manner. The x86, like MULTICS, features a tiny TLB that directly translates the most recently used Dir-Page combinations onto the actual address of the page frame to prevent making repetitive visits to memory. The procedure of actually carrying out and updating the TLB only occurs when the current combination is not already contained in the TLB. Performance is decent as long as TLB misses are few. It is also important to note that this approach is feasible if certain applications are satisfied with a single, paged, 32-bit address space and do not need segmentation. The same selector, whose descriptor has Base 0 and Limit set to the maximum, may be used to configure all of the segment registers. In this case, just one address space will be utilised, which is equivalent to standard paging, and the instruction offset will be the linear address. In actuality, this is how all x86 operating systems now in use function. The Intel MMU architecture's full potential was only used by OS/2. In light of this, why would Intel discontinue support for a form of the MULTICS memory model, which it had maintained for over three decades? Even though it was highly efficient since it did away with system calls in favour of lightning-fast procedure calls to the appropriate location inside a protected operating system segment, the fundamental reason why neither UNIX nor Windows ever implemented it is probably the case. Since it would compromise portability to other platforms, neither UNIX or Windows system developer wanted to modify their memory model to one that was x86-specific. Intel was weary of wasting chip space to support the function since the software wasn't utilising it, therefore it was taken out of the 64-bit CPUs. All in all, the x86 creators deserve praise. The resultant architecture is remarkably straightforward and tidy considering the competing objectives of achieving pure paging, pure segmentation, and paged segments while still being compatible with the 286 and performing all of this effectively.

File Systems

Information must be stored and retrieved by all computer programmes. A process can only store a certain amount of data in its own address space while it is operating. The virtual address space's size, however, places a limit on the storage capacity. This size is sufficient for certain uses, but it is considerably too tiny for others, including airline bookings, banking, or business record keeping. Keeping data in a process' address space has the additional drawback of erasing it once the process has ended. Information must be kept on file for days, weeks, or even years in many applications (like databases, for example). It is unacceptable for it to disappear after the procedure that uses it finishes. Moreover, it must persist even after a computer crash ends the operation. A third issue is the frequent need for many processes to access (parts of) the information simultaneously. Only that process may access an online telephone directory if it is kept within the address space of a single process. Making the information independent of any one procedure is the solution to this issue. As a result, there are three crucial conditions that must be met for long-term data storage: There must be a very significant quantity of storage space available. The data must remain accessible after the procedure that used it has ended. The data must be accessible by several processes at once. For this kind of long-term storage, magnetic discs have long been used. Solid-state drives have grown in popularity recently since they don't have any moving components that may malfunction. They also provide quick random access. While they have significantly lower performance and are normally used for backups, tapes and optical discs have also been widely utilised. Nonetheless, for the purposes of this discussion, it is sufficient to conceive of a disc as a linear succession of fixed-size blocks enabling two operations: Write block k, then read block k. While there are more in reality, the long-term storage issue may theoretically be resolved with these two actions. Nevertheless, these actions are exceedingly cumbersome, particularly on big systems that are utilised by several programmes and sometimes multiple users (e.g., on a server). a couple of the questions are:

We can solve this issue with a new abstraction: the file, much as we saw how the operating system abstracted away the idea of the processor to create the abstraction of a process and how it abstracted away the concept of physical memory to provide processes with (virtual) address spaces. The three most crucial ideas in relation to operating systems are the abstractions of processes (and threads), address spaces, and files. You are well on your way to mastering operating systems if you can really grasp these three ideas from beginning to finish. Processes produce files, which are logical units of information. Thousands or even millions of them, each separate from the others, are often present on a disc. While they are used to mimic the disc rather than the RAM, if you think of each file as a kind of address space, you are not too far off. In case new ones are required, processes may also read already existing files. File-stored data must be durable, or unaffected by the beginning and ending of processes. Only when a file's owner expressly removes it should it vanish. While the most frequent operations are those for reading and writing files, there are many more, some of which we shall look at below. The operating system is in charge of managing files. One of the main areas of discussion in operating system design is how components are organised, named, accessible, utilised, protected, implemented, and controlled. The file system is the collective name for that portion of the operating system that deals with files. From the user's perspective, a file system's appearance—that is, what comprises a file, how files are named and secured, what activities are permitted on files, etc.—is its most crucial feature. There is no need to know the specifics of how many sectors make up a logical disc block or whether linked lists or bitmaps are used to manage free storage, even if these details are crucial to the file system's designers. The first two, correspondingly, deal with the user interface for files and directories. The file system's implementation and management are then covered in depth. Lastly, we provide some actual file system examples.

Files

In the pages that follow, we'll examine files from the perspective of the user, including how they're used and what attributes they have.

Naming files

An abstraction mechanism is a file. It offers a mechanism to put data to the disc and retrieve it later. This has to be done in a manner that hides from the user the specifics of how and where the data is stored and how the discs really function.

The way the objects being handled are named is perhaps the most crucial aspect of any abstraction mechanism, thus we will begin our investigation of file systems with this topic. A process gives a file a name when it creates it. The file survives the process' termination and may still be accessed by other processes using its name. Systems have somewhat different restrictions for file naming, although all modern operating systems accept strings of one to eight letters as valid file names. Hence, viable file names are Andrea, Bruce, and Kathy. Names like 2, urgent!, and Fig.2-14 are often acceptable as well since numerals and special characters are frequently allowed as well. Names may be up to 255 characters long in certain file systems.

Some file systems differentiate between letters in upper- and lowercase, whereas others do not. The old MS-DOS comes within the second group, whereas UNIX falls into the first. (As an aside, MS-DOS is by no means out of date; despite being old, embedded systems still heavily use it.) As a result, Maria, Maria, and MARIA may all exist as three separate files on a UNIX system. All of these names in MS-DOS pertain to the same file. Here, a digression on file systems is probably appropriate. Windows 95 and Windows 98 both made use of the FAT-16 file system from MS-

DOS, and as a result, they share many of its characteristics, including the way file names are created. FAT-16 was given several extensions by Windows 98, which eventually led to FAT-32, albeit the two are quite similar. In addition, the FAT file systems, which are now essentially defunct, are still supported by Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7, and Windows 8. Nevertheless, these more recent operating systems also contain a native file system (NTFS) that is far more complex and has many characteristics (such as file names in Unicode). ReFS (or Resilient File System) is a second file system for Windows 8 that is intended for the server edition of the operating system. Unless otherwise stated, when we refer to the MS-DOS or FAT file systems, we mean the Windows-compatible FAT-16 and FAT-32 file systems. thoroughly explain Windows 8, we will talk about the FAT file systems. A new FAT-like file system called exFAT, a Microsoft extension to FAT-32 that is designed for flash devices and big file systems, is also available. The only contemporary Microsoft file system that OS X can read and write to is Exfat. Many operating systems allow for file names that have two sections and are separated by a period, such as prog.c The phrase that comes after the period is referred to as the file extension and often contains information about the file. In MS-DOS, for ex- ample, file names are 1 to 8 characters, with an optional extension of 1 to 3 charac- ters. On UNIX, the size of the extension, if any, is up to the user, and a file may even have two or more extensions, as in homepage.html.zip, where .html denotes a Web page in HTML and .zip indicates that the file (homepage.html) has been compressed using the zip software.

File extensions are only conventions in some systems (like all variants of UNIX), and the operating system makes no effort to enforce them. While a file with the name file.txt may be a text file, the name serves more as a reminder to the owner than as actual information for the computer. A C compiler, on the other hand, could actually demand that the files it is to build end in.c and might refuse to do so if they don't. The operating system, though, is unconcerned. This sort of convention is very helpful when a software can handle a variety of file types. For instance, the C compiler may be given a list of many files to link and build, some of which are in C and some of which are in assembly language. The extension is thus crucial for the compiler to identify whether files are C files, assembly files, and other files. Windows, on the other hand, is aware of the extensions and gives them a purpose. Users (or processes) may define which software "owns" each extension when registering it with the operating system. Double clicking a file name launches the application associated with that file extension with the file as a parameter. For instance, when file.docx is double-clicked, Microsoft Word opens with file.docx as the first file to edit.

File Organization

The most flexibility is achieved by having the operating system treat files as nothing more than byte sequences. User programmes are free to name their files anyway they choose and to put anything they like in them. While it does not assist, the operating system also does not obstruct. The latter may be crucial for people who wish to do novel things. This file model is used by every UNIX variant, including Linux and OS X, as well as Windows. According to this paradigm, a file is a collection of records with defined lengths and some internal organisation. The notion that a read action flips one record around and a write operation overwrites or appends one record is fundamental to the concept of a file being a series of records. , on the wall of the the the the the the eye and the eye and the eye, in the the, in the and and the to the of the the ejpurling the king the the king, I heing the king in the king in the king the the the The 132-character records files for the line printer were likewise supported by these systems (which in those days were big chain printers having 132 columns). While the remaining 52 characters may, of course, be spaces, programmes accepted input in units of 80 characters and wrote it in units of 132 characters. This type was formerly a typical one on mainframe computers in the period of 80-column punched cards and 132-character line printer paper, but it is no longer the predominant file system for any modern general-purpose system. In this system, a file is made up of a tree of records, not all of which must have the same length, and each of which has a key field at a set location. To enable quick searching for a certain key, the tree is arranged according to the key field. The primary goal in this situation is to get the record that contains a certain key, not the "next" record, but doing so is still an option. Without caring about the record's precise location in the file, one may instruct the system to return the record whose key is pony, for example. Additionally, the operating system, rather than the user, chooses where to put new entries as they are added to the file. This form of file, which is used on certain big mainframe computers for business data processing, is obviously quite different from the unstructured byte streams used in UNIX and Windows.

File Formats

Many file formats are supported by many operating systems. For instance, Windows and UNIX (again, including OS X) both contain standard files and folders. User information is included in regular files. Directories are system files that keep the file system's organisation in place. Here, we'll examine directories. Input/output character special files are used to simulate serial I/O devices like terminals, printers, and networks. Modeling discs is done using block special files. We'll be mostly interested in common files.

Binary or ASCII files typically make up regular files. Lines of text comprise ASCII files. In certain systems, a carriage return character is used to end each line. The line feed character is used in others. Some systems (like Windows) combine the two. It's not necessary for lines to have the same length. The main benefit of ASCII files is that they can be changed with any text editor and viewed and printed in their original form. Moreover, connecting the output of one programme to the input of another is simple when several applications utilise ASCII files for input and output, as with shell pipelines.

The fact that certain files are binary simply denotes that they are not ASCII files. The printer's listing of them produces a confusing list filled with random garbage. They often have some internal structure that programmes that utilise them are aware of. We see a straightforward executable binary file that was extracted from an early UNIX version. While a file is only a series of bytes in theory, the operating system will only execute a file if it has the right format. The header, text, data, relocation bits, and symbol table are among its five parts. The header's first character—the "magic number"—designates the file as an executable file (to pre- vent the accidental execution of a file not in this format). The sizes of the individual file fragments, the location at which execution begins, and a few flag bits follow. The program's text and data are shown after the header. The relocation bits are used to load them into memory and move them. For debugging, the symbol table is used. The archive is our second example of a binary file and it again comes from UNIX. It is made up of a number of compiled but unlinked library routines (modules). Each one has a header at the front that includes the name, creation date, owner, protection code, and size of the file. The headers for the module are stuffed with binary numbers, much as the executable file is. These would print out as nonsense if copied to a printer. Every operating system has to be able to identify its own executable file as one of the file types, while some can recognise more. The outdated TOPS-20 system (for the DECsystem 20) even checked the creation time of each file that was about to be run. It then located the source file and checked to see whether it had been changed since the binary was created. If it had, the source was immediately recompiled. In UNIX words, the build programme had been incorporated into the shell. It was able to identify which binary software was taken from which source thanks to the mandatory file extensions. Strongly typed files create issues if a user does an action that the system designers did not anticipate. Think about, for instance, a system where programme output files have the.dat extension (data files). The output file will be of type.dat if a user creates a programme formatter that reads a.c file (C programme), alters it (for example, by converting it to a standard indentation layout), and then outputs the modified file as output. The system will reject the user's attempt to send this to the C compiler for compilation because it contains the incorrect extension. The system will reject any attempts to transfer file.dat to file.c as being invalid (to protect the user against mistakes). Although this type of "user friendliness" may benefit beginners, it frustrates seasoned users who must put forth a lot of effort to get around the operating system's interpretation of what is and isn't acceptable.

File Access

Sequential access was the sole kind of file access offered by early operating systems. These systems allowed processes to read every byte or record in a file sequentially, beginning at the beginning, but not to hop around or read them out of sequence. Yet, sequential files may be rewound and viewed as often as required. When the store medium was magnetic tape rather than disc, sequential files were practical. When discs were used to store files, it was feasible to access records by key rather than by position or to read the bytes or records of a file out of order. Random-access files are those whose records or bytes may be read in any order. They are necessary for several applications. Several applications, including database systems, depend on random access files. The reservation software must be able to obtain the record for that flight without first reading the data for thousands of other flights if an airline customer phones to book a seat on a specific trip. There are two ways to indicate where to begin reading. Every read operation in the first one specifies the location in the file to begin reading at. In the second, the current location may be set via a particular procedure called seek.

Feature Files

Each file contains its data and a name. All operating systems also correlate additional data with each file, such as the size of the file and the date and time it was last edited. These additional elements will be referred to as the file's attributes. Some refer to them as metadata. The list of properties differs greatly across systems. All of them are present in some systems, although no contemporary system contains all of them. The file's protection is indicated by the first four properties, which also indicate who may and cannot access the file. There are several alternative systems, some of which we shall examine later. In certain systems, the password must be one of the properties if the user has to enter one to access a file. The flags are brief fields or bits that enable or govern a certain characteristic. For instance, lists of all the files do not include hidden files. A piece called the archive flag maintains track of how recently the file has been backed up. When a file is modified, the operating system sets it and the backup application clears it. The backup application can determine which files need backup in this manner. A file may be designated for automatic deletion at the termination of the process that produced it using the temporary flag. The record-length, key-position, and key-length fields are only included in files whose records may be searched up using a key. They provide the details needed to locate the keys. The different timings record the creation, most recent access, and most recent modification of the file. They provide a range of functions. For instance, a source file that was changed after the corresponding object file was created has to be recompiled. The required information may be found in these areas.

The current size indicates the file's current size. In order to allow the operating system to reserve the maximum amount of storage in advance, several vintage mainframe operating systems required the maximum size to be provided when the file was created. Thankfully, modern workstation and personal computer operating systems are intelligent enough to function without this capability (Table 15.1).

Table 15.1: Some possible file attributes.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Operation of Files

Information may be saved in files so they can be subsequently accessed. Various systems provide various retrieval and storing activities.

Create: The file is initially empty. The call's objectives are to set some of the properties and to signal the arrival of the file.

Delete: To free up disc space, a file must be erased when it is no longer required. For this reason, a system call is always available.

Open: A process must open a file before utilising it. The open call's goal is to provide the system access to main memory so it may quickly retrieve the characteristics and list of disc locations on subsequent calls.

Close: The attributes and disc advertisements are no longer required once all accesses have been completed, thus the file should be closed to release internal table space. Several systems promote this by limiting the amount of open files that a process may have. As a disc stores data in blocks, closing a file compels writing of its final block, even if that block isn't yet completely filled.

Read: The file is read for data. Typically, the current location is where the bytes originate. The caller must give a buffer for the data as well as the number of data that are required.

Write. The file is updated with new data, often at the present position. The size of the file increases if the current location is at the file's end. Existing data are overwritten and permanently lost if the location is in the middle of the file.

Append: A limited variant of write is this call. It can only append data to the file's end. Append is seldom present in systems that give a small number of system calls, but it is often present in systems that provide a variety of methods to accomplish the same task.

Seek: A mechanism is required to define where to take the data for random-access files. One such method is the system call seek, which moves the file pointer to a certain location in the file. Data may be read from or written to that place once this call has finished.

Acquire qualities: To complete their tasks, processes often need to read file characteristics. For instance, it's typical practise to handle software development projects with a large number of source files using the UNIX make application. When make is called, it looks at all of the source and object files' modification dates and determines how many compilations are necessary to bring everything up to date. To complete its work, it must look at the properties, specifically, the modification times.

Set characteristics. When the file has been produced, some of the properties may be modified by the user. This system call enables that. The information about protection mode is a prime example. The majority of flags also fit under this category.

Rename. It regularly occurs for a user to need to update a file's name. This system call enables that. The file may typically be transferred to a new file with a new name, and the old file can subsequently be removed, thus it is not always necessarily essential.

Directories

File systems often feature directories or folders, which are files in and of themselves, to keep track of files. The structure, characteristics, and activities that may be carried out on directories will all be covered in this section.

Systems for Single-Level Directories

Having a single directory that contains all the files is the most basic kind of directory system. While it is the sole one, it is sometimes referred to as the root directory, but this does not really relevant. This approach was widespread on early personal computers, in part because there was only one user. It's interesting to note that despite being used by several people at once, the CDC 6600, the world's first supercomputer, also had a single directory for all files. Without a doubt, the choice was taken to make the programme design simple. Four files are included in this di- rectory. This method's benefits include its simplicity and capacity for speedy file discovery—after all, there is only one place to search. Simple embedded gadgets like digital cameras and certain portable audio players still utilise it sometimes.

Systems using Hierarchical Directories

If all files were in a single directory, it would be hard for current users to discover anything. The single level is acceptable for extremely basic specialised programmes (and was even used on the earliest personal computers). As a result, a method for grouping similar files is required. For instance, a professor may have a group of files that together make up a book he is writing, another group of files that contain student programmes submitted for another course, a third group of files that contain the code for an advanced compiler-writing system he is building, a fourth group of files that contain grant proposals, as well as other files for electronic mail, meeting minutes, papers he is writing, games, and so on. A key structuring tool for users to arrange their work is the ability to establish an unlimited number of subdirectories. Because of this, this is how almost all contemporary file systems are set up.

Route Names

Some method of naming files is required when the file system is set up as a directory tree. There are often two approaches employed. In the first way, each file is given an absolute path name consisting of the path from the root directory to the file. For instance, the path /usr/ast/mailbox indicates that the mailbox file is located in the ast subfolder, which is itself a subdirectory of usr in the root directory. Absolute path names are distinct and always begin in the root directory.

Whichever character is used, if the separator is the first character in the route name, the path is absolute. The relative path name is the other kind of name. This goes hand-in-hand with the idea of the working directory (also called the current directory). All path names that don't start at the root directory are relative to the working directory when a user designates one directory as the current working directory. For instance, the file whose absolute address is /usr/ast/mailbox may be referred to simply as mailbox if the current working directory is /usr/ast.

Certain applications need access to a particular file regardless of the working directory. They ought to always utilise absolute path names in such situation. For instance, a spelling checker may require to read from /usr/lib/dictionary in order to function properly. s,s,,s, and s the the s of d'essen the s the of the om No matter what the working directory is, the absolute path name will always function. Of course, an alternate strategy is for the spelling checker to send a system call to shift its working directory to /usr/lib and then use merely dictionary as the first argument to open if it requires a significant number of files from that location. It may utilise relative paths after explicitly altering the working directory since it is then certain of its location inside the directory tree. Each process has its own working directory, so when one changes it and then quits, no other processes are impacted and the file system is not changed in any way. In this method, a programme may change its working directory anytime it sees fit and do so with complete safety. Nevertheless, if a library method modifies the working directory and does not return to its original location after it is completed, the remainder of the programme may not function since its previous assumptions about its location may now be unexpectedly incorrect. The working directory is very seldom changed by library processes, and when it is, it is always changed back before proceeding.

Operations in Directories

System to system, the permitted system calls for managing directories differ considerably from those for managing files. We'll provide an example to give you an idea of what they are and how they operate (taken from UNIX).

Create: It creates a directory. Apart for dot and dotdot, which the system automatically placed there, it is empty (or in a few cases, by the mkdir program).

Delete: The deletion of a directory. Only a directory that is empty may be deleted. As these characters cannot be removed, a directory that solely contains dot and dotdot is regarded as empty.

Opendir: Readers may read directories. To list all the files in a directory, for instance, a listing application accesses the directory and reads out all the file names. Like opening and reading a file, a directory must first be opened before it can be read.

Closedir: To free up internal table space, a directory should be closed once it has been read.

Readdir: This method returns the next item in an open directory. Formerly, reading directories was feasible using the standard read system call, but that method had the drawback of requiring the programmer to understand and deal with the underlying structure of directories. In contrast, regardless of the potential directory structures being utilised, readdir always returns one record in a standard manner.

Rename: Directories are similar to files in many ways, including their ability to be renamed

Link. A method called linking enables a file to appear in several directories. An existing file and a path name are supplied in this system call, which then makes a link from the existing file to the name specified by the path. The same file might therefore show up in numerous directories. This kind of connection, which increases the counter in the file's i-node to count the directory entries that include the file, is frequently referred to as a hard link.

Unlink. There is a directory entry deletion. It is deleted from the file system if the file being unlinked is only present in one directory, which is the typical situation. When many directories are involved, just the path name is deleted. They are still there. The system function for removing files in UNIX that was previously explained is really called unlink.

The list above includes the most crucial calls, but there are also a few more, such as those used to manage a directory's protection data.

The symbolic link is a variation on the notion of connecting files. The file system follows the route and locates the name at the end when, for instance, opening the first file. Then, using the new name, it restarts the search procedure. Symbolic connections provide the benefit of being able to name files on distant machines and bridge disc boundaries. Yet, they are not as effective at implementation as hard linkages.

File-System Implementation

The user's perspective of the file system must now give way to the implementor's perspective. Users are worried about things like the directory tree's appearance, how files are named, and the activities that may be performed on them. The storage of files and directories, the management of disc space, and the effective and dependable operation of everything are all of importance to implementors. We will look at a few of these areas in the sections that follow to determine the problems and trade-offs.

System File Layout

On drives, file systems are kept. Most drives may be split into one or more partitions, each of which has an own file system. The Master Boot Record, or MBR, is located in Sector 0 of the disc and is what allows the computer to start up. The partition table is located at the end of the MBR. The beginning and ending addresses of each division are shown in this table. The table has one partition that is designated as active. The MBR is read and run by the BIOS when the machine boots. The boot block, which is the first block of the active partition, is read into by the MBR software, which then executes it. The boot block software starts the operating system in that partition. Every partition has a boot block at the beginning for consistency's sake, even if it doesn't have a bootable operating system on it. Moreover, it could do so in the future.

Questionnaire

1. Why is the operating system important?
2. What's the main purpose of an OS?
3. What are the benefits of a multiprocessor system?
4. What is RAID structure in OS?
5. What is GUI?
6. What is a Pipe and when it is used?
7. What are the different kinds of operations that are possible on semaphore?

Reference Books for further reading:

1. "Operating System Concepts" by Avi Silberschatz and Peter Galvin.
2. "Operating Systems: Internals and Design Principles" by William Stallings.
3. "Operating Systems: A Concept-Based Approach" by D M Dhamdhere.
4. "Operating System: A Design-oriented Approach" by Charles Crowley.
5. "Operating Systems: A Modern Perspective" by Gary J Nutt.
6. "Design of the Unix Operating Systems" by Maurice Bach.
