

Operating System

DECAP560

Edited by
Ajay Kumar Bansal



L OVELY
P ROFESSIONAL
U NIVERSITY



Operating System

**Edited By:
Ajay Kumar Bansal**

CONTENT

Unit 1:	Introduction to Operating System	1
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 2:	Operating System Services	17
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 3:	Process	42
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 4:	Process Management	63
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 5:	Inter-Process Communication	79
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 6:	CPU Scheduling	93
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 7:	Scheduling Algorithms	115
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 8:	Process Synchronization	129
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 9:	Deadlocks	146
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 10:	Memory Management - I	158
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 11:	Memory Management - II	167
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 12:	Memory Management - III	169
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 13:	File Management	188
	<i>Dr. Amit Sharma, Lovely Professional University</i>	
Unit 14:	Disk Management	206
	<i>Dr. Amit Sharma, Lovely Professional University</i>	

Unit 01: Introduction to Operating System

CONTENTS

Objectives

Introduction

- 1.1 Operating System: Meaning
- 1.2 Operating System Definitions
- 1.3 Operations and Functions of OS
- 1.4 Types of Operating System
- 1.5 Operating System: Examples
- 1.6 Components of Operating System:
- 1.7 Abstract View of System Components
- 1.8 Viewpoints of OS
- 1.9 Evolution of OS

Summary

Keywords

Self Assessment

Answers Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the basic organization of computer systems.
- analyzing the major components of operating systems.
- illustrate the viewpoints of operating system.
- understand the evolution of operating system
- illustrate the many types of computing environments
- analyze the several open-source operating systems
- understand the various functions of operating systems
- Describe operations and functions of operating system
- Explain various types of operating system

Introduction

A program that acts as an intermediary between a user of a computer and the computer hardware. Execute user programs and make solving user problems easier. Make the computer system convenient to use. Use the computer hardware in an efficient manner.

Operating System is a software, which makes a computer to work. It is the software that enables all the programs we use. The OS organizes and controls the hardware. OS acts as an interface between the application programs and the machine hardware.



Examples: Windows, Linux, Unix and Mac OS, etc.

1.1 Operating System: Meaning

An operating system (sometimes abbreviated as “OS”) is the program that, after being initially loaded into the computer by a boot program, manages all the other programs in a computer. The other programs are called applications or application programs. The application programs make use of the operating system by making requests for services through a defined Application Program Interface (API). In addition, users can interact directly with the operating system through a user interface such as a command language or a Graphical User Interface (GUI).

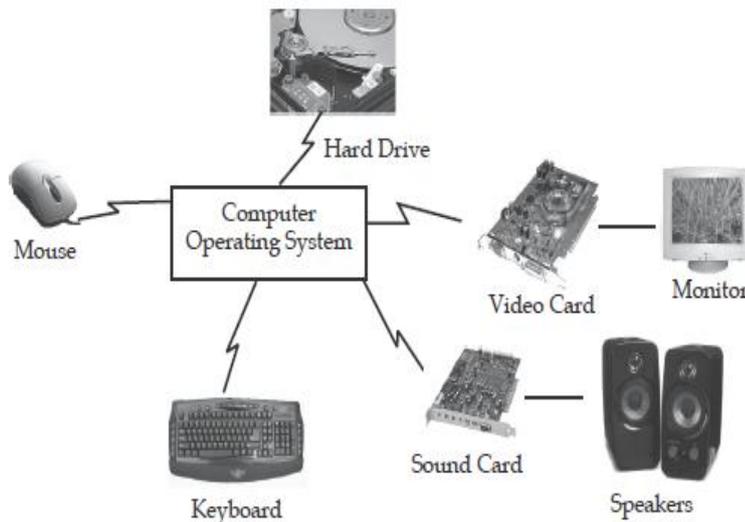


Figure: Operating System Interface

1.2 Operating System Definitions

Operating system is a set of components that manages the hardware and the software resources and connects them all. Once a computer is switched on, the operating system must be loaded into the computer system. OS is a resource allocator that manages all resources. It decides between conflicting requests for efficient and fair resource use. OS is a control program that controls execution of programs to prevent errors and improper use of the computer. Resource allocator- manages and allocates resources.

- Control program- controls the execution of user programs and operations of I/O devices.
- Kernel- the one program always running (all else being application programs).

An operating system performs basic tasks such as:

- controlling and allocating memory
- prioritizing system requests
- controlling input and output devices
- facilitating networking
- managing file systems

1.3 Operations and Functions of OS

The main operations and functions of an operating system are as follows:

1. Process Management
2. Memory Management
3. Secondary Storage Management

4. I/O Management
5. File Management
6. Protection
7. Networking Management
8. Command Interpretation.

1. Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general. The operating system is responsible for the following activities in connection with processes management:

- a) The creation and deletion of both user and system processes
- b) The suspension and resumption of processes.
- c) The provision of mechanisms for process synchronization
- d) The provision of mechanisms for deadlock handling.

2. Memory Management

Memory is the most expensive part in the computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory. There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

- a) Keep track of which parts of memory are currently being used and by whom.
- b) Decide which processes are to be loaded into memory when memory space becomes available.
- c) Allocate and de-allocate memory space as needed.

3. Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modern computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing.

Hence the proper management of disk storage is of central importance to a computer system. There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus, tapes are more suited for storing infrequently used files, where speed is not a primary concern.

The operating system is responsible for the following activities in connection with disk management:

- a) Free space management
- b) Storage allocation
- c) Disk scheduling.

4. I/O Management

Operating System

One of the purposes of an operating system is to hide the peculiarities or specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The operating system is responsible for the following activities in connection to I/O management:

- a) A buffer caching system
- b) To activate a general device driver code
- c) To run the driver software for specific hardware devices as and when required.

5. File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms: magnetic tape, disk, and drum are the most common forms. Each of these devices has its own characteristics and physical organisation. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices. A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, or alphanumeric. Files may be free form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept. The operating system implements the abstract concept of the file by managing mass storage device, such as tapes and disks. Also, files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed. The operating system is responsible for the following activities in connection to the file management:

- a) The creation and deletion of files.
- b) The creation and deletion of directory.
- c) The support of primitives for manipulating files and directories.
- d) The mapping of files onto disk storage.
- e) Backup of files on stable (non-volatile) storage.
- f) Protection and security of the files.

6. Protection

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorization from the operating system.

Example: Memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement. Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

7. Networking Management

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high-speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general-purpose computer systems. The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies and the problems of connection and security. A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

8. Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system. Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed. This program is variously called (1) the control card interpreter, (2) the command line interpreter, (3) the shell (in Unix), and so on. Its function is quite simple: get the next command statement and execute it. The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

1.4 Types of Operating System

Modern computer operating systems may be classified into three groups, which are distinguished by the nature of interaction that takes place between the computer user and his or her program during its processing. The three groups are called batch, time-sharing and real-time operating systems.

1. Batch Processing Operating System

In a batch processing operating system environment user submit jobs to a central place where these jobs are collected into a batch, and subsequently placed on an input queue at the computer where they will be run. In this case, the user has no interaction with the job during its processing, and the computer's response time is the turnaround time the time from submission of the job until execution is complete, and the results are ready for return to the person who submitted the job.

2. Time Sharing

Another mode for delivering computing services is provided by time sharing operating systems. In this environment a computer provides computing services to several or many users concurrently on-line. Here, the various users are sharing the central processor, the memory, and other resources of the computer system in a manner facilitated, controlled, and monitored by the operating system. The user, in this environment, has nearly full interaction with the program during its execution, and the computer's response time may be expected to be no more than a few seconds.

3. Real-time Operating System (RTOS)

The third class is the real time operating systems, which are designed to service those applications where response time is of the essence in order to prevent error, misrepresentation or even disaster. Examples of real time operating systems are those which handle airlines reservations, machine tool control, and monitoring of a nuclear power station. The systems, in this case, are designed to be interrupted by external signals that require the immediate attention of the computer system. These real time operating systems are used to control machinery, scientific instruments and industrial systems. An RTOS typically has very little user-interface capability, and no end-user utilities. A very important part of an RTOS is managing the resources of the computer so that a particular operation executes in precisely the same amount of time every time it occurs. In a complex machine, having a part move more quickly just because system resources are available may be just as catastrophic as having it not move at all because the system is busy. A number of other definitions are important to gain an understanding of operating systems:

4. Multiprogramming Operating System

A multiprogramming operating system is a system that allows more than one active user program (or part of user program) to be stored in main memory simultaneously. Thus, it is evident that a time-sharing system is a multiprogramming system, but note that a multiprogramming system is not necessarily a time-sharing system. A batch or real time operating system could, and indeed usually does, have more than one active user program simultaneously in main storage. Another important, and all too similar, term is "multiprocessing".

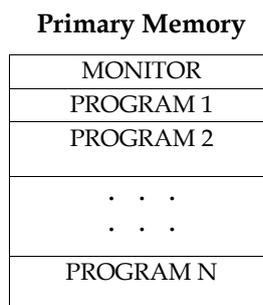


Figure 2.2: Memory Layout in Multiprogramming Environment

Buffering and Spooling improve system performance by overlapping the input, output and computation of a single job, but both of them have their limitations. A single user cannot always keep CPU or I/O devices busy at all times. Multiprogramming offers a more efficient approach to increase system performance. In order to increase the resource utilization, systems supporting multiprogramming approach allow more than one job (program) to reside in the memory to utilize CPU time at any moment. More number of programs competing for system resources better will mean better resource utilization. The idea is implemented as follows. The main memory of a system contains more than one program (Figure 2.2). The operating system picks one of the programs and starts executing. During execution of program 1 it needs some I/O operation to complete in a sequential execution environment (Figure 2.3a). The CPU would then sit idle whereas in a multiprogramming system, (Figure 2.3b) the operating system will simply switch over to the next program (program 2).

When that program needs to wait for some I/O operation, it switches over to program 3 and soon. If there is no other new program left in the main memory, the CPU will pass its control back to the previous programs.

Multiprogramming has traditionally been employed to increase the resources utilization of a computer system and to support multiple simultaneously interactive users (terminals).

5. Multiprocessing System

A multiprocessing system is a computer hardware configuration that includes more than one independent processing unit. The term multiprocessing is generally used to refer to large computer hardware complexes found in major scientific or commercial applications. A multiprocessor system is simply a computer that has >1 & not ≤ 1 CPU on its motherboard. If the operating system is built to take advantage of this, it can run different processes (or different threads belonging to the same process) on different CPUs.

Today's operating systems strive to make the most efficient use of a computer's resources. Most of this efficiency is gained by sharing the machine's resources among several tasks (multi-processing). Such "large-grain" resource sharing is enabled by operating systems without any additional information from the applications or processes. All these processes can potentially execute concurrently, with the CPU (or CPUs) multiplexed among them. Newer operating systems provide mechanisms that enable applications to control and share machine resources at a finer grain-, that is, at the threads level. Just as multiprocessing operating systems can perform more than one task concurrently by running more than a single process, a process can perform more than one task by running more than a single thread.

6. Networking Operating System

A networked computing system is a collection of physical interconnected computers. The operating system of each of the interconnected computers must contain, in addition to its own stand-alone functionality, provisions for handling communication. These additions do not change the essential structure of the operating systems.

7. Distributed Operating System

A distributed computing system consists of a number of computers that are connected and managed so that they automatically share the job processing load among the constituent computers, or separate the job load as appropriate particularly configured processors. Such a system requires an operating system which, in addition to the typical stand-alone functionality, provides coordination of the operations and information flow among the component computers. The networked and distributed computing environments and their respective operating

systems are designed with more complex functional capabilities. In a network operating system, the users are aware of the existence of multiple computers, and can log in to remote machines and copy files from one machine to another. Each machine runs its own local operating system and has its own user (or users). A distributed operating system, in contrast, is one that appears to its users as a traditional uni-processor system, even though it is actually composed of multiple processors. In a true distributed system, users should not be aware of where their programs are being run or where their files are located; that should all be handled automatically and efficiently by the operating system.

True distributed operating systems require more than just adding a little code to a uni-processor operating system, because distributed and centralized systems differ in critical ways. Distributed systems, for example, often allow program to run on several processors at the same time, thus requiring more complex processor scheduling algorithms in order to optimize the amount of parallelism achieved.

8. *Operating Systems for Embedded Devices*

As embedded systems (PDAs, cellphones, point-of-sale devices, VCR's, industrial robot control, or even your toaster) become more complex hardware-wise with every generation, and more features are put into them day-by-day, applications they run require more and more to run on actual operating system code in order to keep the development time reasonable. Some of the popular OS are:

1. *Nexus's Conic*: an embedded operating system for ARM processors.
2. *Sun's Java OS*: a standalone virtual machine not running on top of any other OS; mainly targeted at embedded systems.
3. *Palm Computing's Palm OS*: Currently the leader OS for PDAs, has many applications and supporting companies.
4. Microsoft's Windows CE and Windows NT Embedded OS.

9. *Single Processor System*

In theory, every computer system may be programmed in its machine language, with no system software support. Programming of the "bare-machines" was customary for early computer systems. A slightly more advanced version of this mode of operating is common for the simple evaluation boards that are sometimes used in introductory microprocessor design and interfacing courses. Programs for the bare machine can be developed by manually translating sequences of instructions into binary or some other code whose base is usually an integer power of 2. Instructions and data are then fed into the computer by means of console switches, or perhaps through a hexadecimal keyboard. Programs are started by loading the program counter with the address of the first instruction. Results of execution are obtained by examining the contents of the relevant registers and memory locations. Input/output devices, if any, must be controlled by the executing program directly, say, by reading and writing the related I/O ports. Evidently, programming of the bare machine results in low productivity of both users and hardware. The long and tedious process of program and data entry practically precludes execution of all but very short programs in such an environment.

The next significant evolutionary step in computer system usage came about with the advent of input/output devices, such as punched cards and paper tape, and of language translators. Programs, now coded in a programming language, are translated into executable form by a computer program, such as compiler or an interpreter. Another program, called the loader, automates the process of loading executable programs into memory. The user places a program and its input data on an input device, and the loader transfers information from that input device into memory. After transferring control to the loaded program by manual or automatic means, execution of the program commences. The executing program reads its input from the designated input device and may produce some output on an output device, such as a printer or display screen. Once in memory, the program may be rerun with different set of input data. The mechanics of development and preparation of programs in such environments are quite slow and cumbersome due to serial execution of programs and numerous manual operations involved in the process. In a typical sequence, the editor program is loaded to prepare the source code of the user program. The next step is to load and execute the language translator and to provide it with the source code of the user program. When serial input devices, such as card readers, are used, multiple-pass language translators may require the source code to be repositioned for reading during each pass. If syntax errors are detected, the whole process must be repeated from the

beginning. Eventually, the object code produced from the syntactically correct sourcecode is loaded and executed. If run-time errors are detected, the state of the machine can be examined and modified by means of console switches, or with the assistance of a program called a debugger. The mode of operation described here was initially used in late fifties, but it was also common in low-end microcomputers of early eighties with cassettes as I/O devices. In addition to language translators, system software includes the loader and possibly editor and debugger programs. Most of them use input/output devices and thus must contain some code to exercise those devices. Since many user programs also use input/output devices, the logical refinement is to provide a collection of standard I/O routines for the use of all programs. In the described system, I/O routines and the loader program represent a rudimentary form of an operating system. Although quite crude, it still provides an environment for execution of programs far beyond what is available on the bare machine. Language translators, editors, and debuggers are system programs that rely on the services of, but are not generally regarded as part of, the operating system. Although a definite improvement over the bare machine approach, this mode of operation is obviously not very efficient. Running of the computer system may require frequent manual loading of programs and data. This results in low utilization of system resources. User productivity, especially in multiuser environments, is low as users await their turn at the machine. Even with such tools as editors and debuggers, program development is very slow and is ridden with manual program and data loading.

10. Parallel Processing System

Parallel operating systems are primarily concerned with managing the resources of parallel machines. This task faces many challenges: application programmers demand all the performance possible, many hardware configurations exist and change very rapidly, yet the operating system must increasingly be compatible with the mainstream versions used in personal computers and workstations due both to user pressure and to the limited resources available for developing new versions of these systems. There are several components in an operating system that can be parallelized. Most operating systems do not approach all of them and do not support parallel applications directly. Rather, parallelism is frequently exploited by some additional software layer such as a distributed file system, distributed shared memory support or libraries and services that support particular parallel programming languages while the operating system manages concurrent task execution. The convergence in parallel computer architectures has been accompanied by a reduction in the diversity of operating systems running on them. The current situation is that most commercially available machines run a flavor of the UNIX OS (Digital UNIX, IBM AIX, HP UX, Sun Solaris, Linux). Others run a UNIX based microkernel with reduced functionality to optimize the use of the CPU, such as Cray Research's UNICOS. Finally, a number of shared memory MIMD machines run Microsoft Windows NT (soon to be superseded by the high-end variant of Windows 2000). There are a number of core aspects to the characterization of a parallel computer operating system: general features such as the degrees of coordination, coupling and transparency; and more particular aspects such as the type of process management, inter-process communication, parallelism and synchronization and the programming model.

11. Multitasking

In computing, multitasking is a method where multiple tasks, also known as processes, share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by scheduling which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called a context switch. When context switches occur frequently enough the illusion of parallelism is achieved. Even on computers with more than one CPU (called multiprocessor machines), multitasking allows many more tasks to be run than there are CPUs. In the early ages of the computers, they were considered advanced card machines and therefore the jobs they performed were like: "find all females in this bunch of cards (or records)". Therefore, utilization was high since one delivered a job to the computing department, which prepared and executed the job on the computer, delivering the final result to you. The advances in electronic engineering increased the processing power several times, now leaving input/output devices (card readers, line printers) far behind. This means that the CPU had to wait for the data it required to perform a given task. Soon, engineers thought: "what if we could both prepare, process and output data at the same time" and multitasking was born. Now one could read data for the next job while executing the current job and outputting the results of a previously job, thereby increasing the utilization of the very expensive computer. Cheap terminals allowed the users themselves to input data to the computer and to execute jobs (having the department do it

often took days) and see results immediately on the screen, which introduced what was called interactive tasks. They required a console to be updated when a key was pressed on the keyboard (again a task with slow input). Same thing happens today, where your computer actually does no work most of the time - it just waits for your input. Therefore, using multitasking where several tasks run on the same computer improves performance. Multitasking is the process of letting the operating system perform multiple tasks at what seems to the user simultaneously. In SMP (Symmetric Multi-Processor systems) this is the case, since there are several CPUs to execute programs on - in systems with only a single CPU this is done by switching execution very rapidly between each program, thus giving the impression of simultaneous execution. This process is also known as task switching or timesharing. Practically all modern OS has this ability. Multitasking is, on single-processor machines, implemented by letting the running process own the CPU for a while (a time slice) and when required gets replaced with another process, which then owns the CPU. The two most common methods for sharing the CPU time is either cooperative multitasking or preemptive multitasking.

- a) **Cooperative Multitasking:** The simplest form of multitasking is cooperative multitasking. It lets the programs decide when they wish to let other tasks run. This method is not good since it lets some process monopolize the CPU and never let other processes run. This way a program may be reluctant to give away processing power in the fear of another process hogging all CPU-time. Early versions of the MacOS (until MacOS 8) and versions of Windows earlier than Win95/WinNT used cooperative multitasking (Win95 when running old apps).
- b) **Preemptive Multitasking:** Preemptive multitasking moves the control of the CPU to the OS, letting each process run for a given amount of time (a time slice) and then switching to another task. This method prevents one process from taking complete control of the system and thereby making it seem as if it is crashed. This method is most common today, implemented by among others OS/2, Win95/98, WinNT, Unix, Linux, BeOS, QNX, OS9 and most mainframe OS. The assignment of CPU time is taken care of by the scheduler.

1.5 Operating System: Examples

Disk Operating System (DOS)

DOS (Disk Operating System) was the first widely-installed operating system for personal computers. It is a master control program that is automatically run when you start your personal computer (PC). DOS stays in the computer all the time letting you run a program and manage files. It is a single-user operating system from Microsoft for the PC. It was the first OS for the PC and is the underlying control program for Windows 3.1, 95, 98 and ME. Windows NT, 2000 and XP emulate DOS in order to support existing DOS applications.

UNIX

UNIX operating systems are used in widely-sold workstation products from Sun Microsystems, Silicon Graphics, IBM, and a number of other companies. The UNIX environment and the client/server program model were important elements in the development of the Internet and the reshaping of computing as centered in networks rather than in individual computers. Linux, a UNIX derivative available in both "free software" and commercial versions, is increasing in popularity as an alternative to proprietary operating systems. UNIX is written in C. Both UNIX and C were developed by AT&T and freely distributed to government and academic institutions, causing it to be ported to a wider variety of machine families than any other operating system. As a result, UNIX became synonymous with "open systems".

UNIX is made up of the kernel, file system and shell (command line interface). The major shells are the Bourne shell (original), C shell and Korn shell. The UNIX vocabulary is exhaustive with more than 600 commands that manipulate data and text in every way conceivable. Many commands are cryptic, but just as Windows hid the DOS prompt, the Motif GUI presents a friendlier image to UNIX users. Even with its many versions, UNIX is widely used in mission critical applications for client/server and transaction processing systems. The UNIX versions that are widely used are Sun's Solaris, Digital's UNIX, HP's HP-UX, IBM's AIX and SCO's UnixWare. A large number of IBM mainframes also run UNIX applications, because the UNIX interfaces were

Operating System

added to MVS and OS/390, which have obtained UNIX branding. Linux, another variant of UNIX, is also gaining enormous popularity.

Windows

Windows is a personal computer operating system from Microsoft that, together with some commonly used business applications such as Microsoft Word and Excel, has become a de facto "standard" for individual users in most corporations as well as in most homes. Windows contains built-in networking, which allows users to share files and applications with each other if their PC's are connected to a network. In large enterprises, Windows clients are often connected to a network of UNIX and NetWare servers. The server versions of Windows NT and 2000 are gaining market share, providing a Windows-only solution for both the client and server. Windows is supported by Microsoft, the largest software company in the world, as well as the Windows industry at large, which includes tens of thousands of software developers. This networking support is the reason why Windows became successful in the first place. However, Windows 95, 98, ME, NT, 2000 and XP are complicated operating environments. Certain combinations of hardware and software running together can cause problems, and troubleshooting can be daunting. Each new version of Windows has interface changes that constantly confuse users and keep support people busy, and installing Windows applications is problematic too. Microsoft has worked hard to make Windows 2000 and Windows XP more resilient to installation of problems and crashes in general.

Macintosh

The Macintosh (often called "the Mac"), introduced in 1984 by Apple Computer, was the first widely-sold personal computer with a Graphical User Interface (GUI). The Mac was designed to provide users with a natural, intuitively understandable, and, in general, "user-friendly" computer interface. This includes the mouse, the use of icons or small visual images to represent objects or actions, the point-and-click and click-and-drag actions, and a number of window operation ideas. Microsoft was successful in adapting user interface concepts first made popular by the Mac in its first Windows operating system. The primary disadvantage of the Mac is that there are fewer Mac applications on the market than for Windows. However, all the fundamental applications are available, and the Macintosh is a perfectly useful machine for almost everybody. Data compatibility between Windows and Mac is an issue, although it is often overblown and readily solved. The Macintosh has its own operating system, Mac OS which, in its latest version is called Mac OS X. Originally built on Motorola's 68000 series microprocessors, Mac versions today are powered by the PowerPC microprocessor, which was developed jointly by Apple, Motorola, and IBM. While Mac users represent only about 5% of the total numbers of personal computer users, Macs are highly popular and almost a cultural necessity among graphic designers and online visual artists and the companies they work for. *Task* DOS is a character-based operating system what about Windows operating system.

1.6 Components of Operating System:

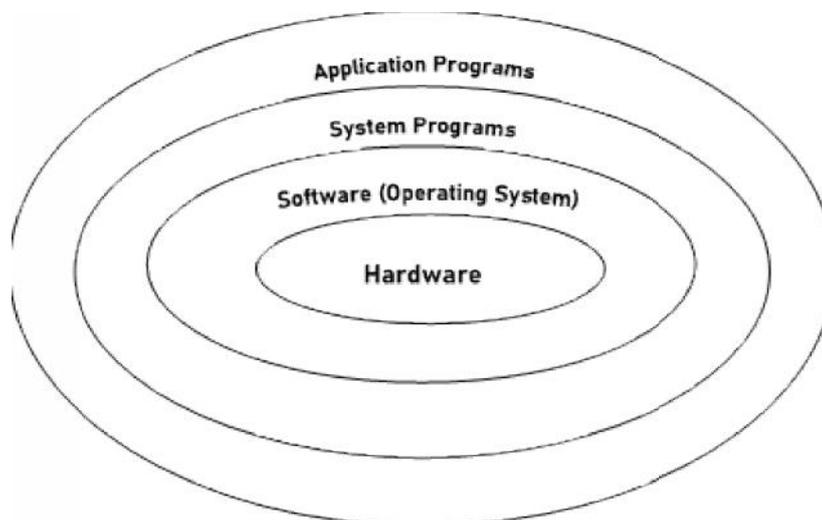


Figure: Components of Operating System

Modern general-purpose computers, including personal computers and mainframes, have an operating system to run other programs, such as application software. Examples of operating systems for personal computers include Microsoft Windows, Mac OS (and Darwin), Unix, and Linux. The lowest level of any operating system is its kernel. This is the first layer of software loaded into memory when a system boots or starts up. The kernel provides access to various common core services to all other system and application programs.

These services include, but are not limited to disk access, memory management, task scheduling, and access to other hardware devices. As the kernel, an operating system is often distributed with tools for programs to display and manage a graphical user interface (although Windows and the Macintosh have these tools built into the operating system), as well as utility programs for tasks such as managing files and configuring the operating system. They are also often distributed with application software that does not relate directly to the operating system's core function, but which the operating system distributor finds advantageous to supply with the operating system. The delineation between the operating system and application software is not precise and is occasionally subject to controversy. From commercial or legal points of view, the delineation can depend on the contexts of the interests involved. For example, one of the key questions in the United States v. Microsoft antitrust trial was whether Microsoft's web browser was part of its operating system or whether it was a separable piece of application software. Like the term "operating system" itself, the question of what exactly should form the "kernel" is subject to some controversy, with debates over whether things like file systems should be included in the kernel. Various camps advocate microkernels, monolithic kernels, and so on. Operating systems are used on most, but not all, computer systems. The simplest computers, including the smallest embedded systems and many of the first computers, did not have operating systems. Instead, they relied on the application programs to manage the minimal hardware themselves, perhaps with the aid of libraries developed for the purpose. Commercially supplied operating systems are present on virtually all modern devices described as computers, from personal computers to mainframes, as well as mobile computers such as PDAs and mobile phones.

The structure of OS consists of 4 layers

1. Hardware
2. Software (Operating System)
3. System Programs
4. Application Programs

1. **Hardware:** The hardware—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system.
2. **Software (Operating System):** The operating system controls and coordinates the use of the hardware among the various application programs for the various users.
3. **System Programs:** This layer consists of compilers, Assemblers, linker etc.
4. **Application Programs:** The application programs—such as word processors, spreadsheets, compilers, and web browsers—define the ways in which these resources are used to solve users' computing problems.

We can also view a computer system as consisting of hardware, software, and data. The operating system provides the means for proper use of these resources in the operation of the computer system. An operating system is similar to a government. Like a government, it performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

1.7 Abstract View of System Components

Computer system can be divided into four components:

- 1) Hardware: - provides basic computing resources like the CPU, memory, I/O devices etc.

- 2) Operating system: - controls and coordinates use of hardware among various applications and users.
- 3) Application programs: - define the ways in which the system resources are used to solve the computing problems of the users. For example: Word processors, compilers, web browsers, database systems and video games.
- 4) Users: - People, machines, other computers

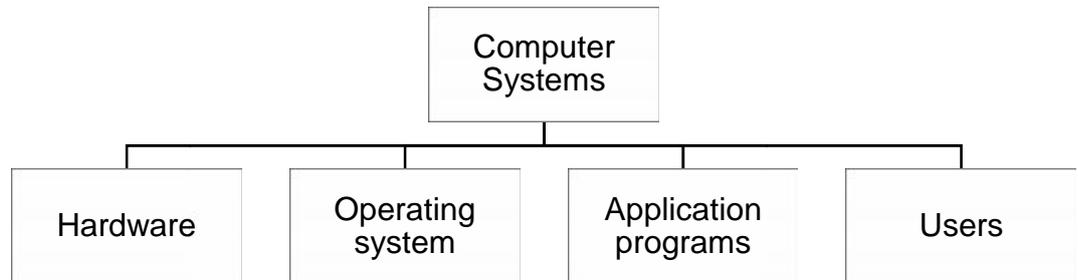


Figure: Abstract View of System Components

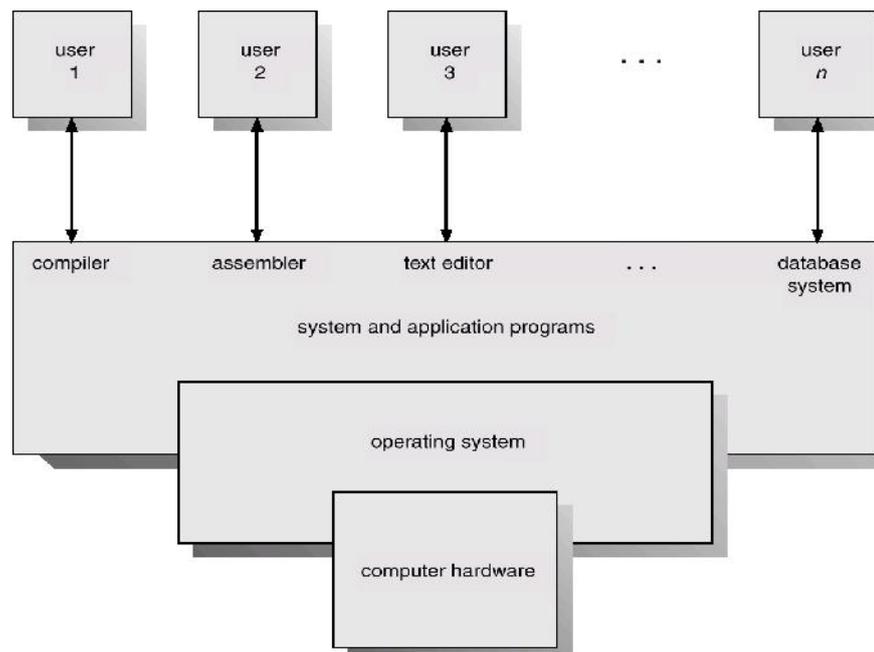


Figure: Operating System Concepts

1.8 Viewpoints of OS

1. User View
2. System View

1. **User View** -The user's view of the computer varies according to the interface being used. Most computer users sit in front of a PC, consisting of a monitor, keyboard, mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case, the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization— how various hardware and software resources are shared. Performance is, of course, important to the user; but rather than resource utilization, such systems are optimized for a single-user experience. In other cases, a user sits at a terminal connected to a mainframe or minicomputer. Other users can connect to the same computer via different terminals. These

users share resources and may exchange information. The operating system, in such cases, is designed to maximize resource utilization – to assure that all available CPU time, memory, and I/O are used efficiently, and that no individual user takes more than her fair share. In still other cases, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers – file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.

2. **System View** - From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a resource allocator. The operating system acts as the manager of these resources. A computer system has many resources that may be required to solve a problem, like the CPU time, memory space, file-storage space, I/O devices, and so on. A slightly different view of an operating system emphasizes the need to control the various I/O devices and user programs. An operating system is a control program. A control program manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

1.9 Evolution of OS

The evolution of operating systems went through seven major phases.

- Six of them significantly changed the ways in which users accessed computers through the open shop, batch processing, multiprogramming, timesharing, personal computing, and distributed systems.
- In the seventh phase the foundations of concurrent programming were developed and demonstrated in model operating systems.

Table: Phases during the evolution of operating system

Major Phases	Technical Innovations	Operating Systems
Open Shop	The idea of OS	IBM 701 open shop (1954)
Batch Processing	Tape batching, First-in, first-out scheduling	BKS system (1961)
Multi-Programming	Processor multiplexing, Indivisible operations, Demand paging, Input/output spooling, Priority scheduling, Remote job entry	Atlas supervisor (1961), Exec II system (1966)
Timesharing	Simultaneous user interaction, On-line file systems	Multics file system (1965), Unix (1974)
Concurrent Programming	Hierarchical systems, Extensible kernels, Parallel programming concepts, Secure parallel languages	RC 4000 system (1969), 13 Venus system (1972), 14 Boss 2 system (1975).
Personal Computing	Graphic user interfaces	OS 6 (1972) Pilot system (1980)
Distributed Systems	Remote servers	WFS file server (1979) Unix United RPC (1982) 24 Amoeba system (1990)

Summary

Operating systems may be classified by both how many tasks they can perform “simultaneously” and by how many users can be using the system “simultaneously”. That is: single-user or multi-user and single-task or multi-tasking. A multi-user system must clearly be multi-tasking. A possible solution to the external fragmentation problem is to permit the logical address space of a process to be noncontiguous, thus allowing a process to be allocated physical memory wherever the latter is available. Physical memory is broken into fixed-sized blocks called frames. Logical memory is also

broken into blocks of the same size called pages. Memory protection in a paged environment is accomplished by protection bit that are associated with each frame. Segmentation is a memory-management scheme that supports this user view of memory. Segmentation may then cause external fragmentation, when all blocks of free memory are too small to accommodate a segment.

Keywords

- **Clustered System:** A clustered system is a group of loosely coupled computers that work together closely so that in many respects they can be viewed as though they are a single computer.
- **Distributed System:** A distributed system is a computer system in which the resources reside in separate units connected by a network, but which presents to the user a uniform computing environment.
- **Real-time Operating System:** A Real-time Operating System (RTOS) is a multitasking operating system intended for real-time applications. Such applications include embedded systems (programmable thermostats, household appliance controllers, mobile telephones), industrial robots, spacecraft, industrial control and scientific research equipment.

Self Assessment

1. A is a program in execution.
 2. is a large array of words or bytes, each with its own address.
 3. A is a collection of related information defined by its creator.
 4. A provides the user with access to the various resources the system maintains.
 5. An RTOS typically has very little user-interface capability, and no
 6. A cannot always keep CPU or I/O devices busy at all times.
 7. A multiprocessing system is a computer hardware configuration that includes more than
 8. independent processing unit.
 9. A system is a collection of physical interconnected computers.
 10. A system task, such as, is also a process.
 11. The process is achieved through a sequence of reads or writes of specific memory address.
12. Which is not the function of the operating system?
 - A. Memory management
 - B. Disk management
 - C. Application management
 - D. Virus protection
 13. Which of the following is a goal of the operating System?
 - A. Execute user programs and make solving user problems easier.
 - B. Make the computer system convenient to use.
 - C. Use the computer hardware in an efficient manner.
 - D. All of the given choices
 14. Which of the following statement is false with respect to the operating systems?
 - A. Operating System is a software, which makes a computer to work.
 - B. It is the software that enables all the programs we use.

Operating System

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 02: Operating System Services

CONTENTS

Objectives

Introduction

2.1 Operating System Functions

2.2 System Calls

2.3 Types of System Call

2.4 Standard C Library Example

2.5 Operating System Structure

2.6 Monolithic Systems

2.7 Client-server Model

2.8 Exokernel

2.9 Layered Structure

2.10 Virtual Machine

Keywords

Summary

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- learn the various Operating System Services.
- understand the command-Interpreter system in detail.
- analyze the various Operating System Functions.
- understand the Operating System Functions.
- understand the mechanism and use of OS services
- learn the types of system calls and the methods to invoke them
- Understand the relationship between the process, system calls and the operating system
- Learn the parameter and system call passing in OS.

Introduction

In computing, a **system call** is the mechanism used by an application program to request service from the operating system based on the monolithic kernel or to system servers on operating systems based on the microkernel-structure. Timings of requested service have to be strictly predictable for application in real time systems – those are most advanced and secure. So far, the only thing we have done was to use well defined kernel mechanisms to register /proc files and device handlers. This is fine if you want to do something the kernel programmers thought you had wanted, such as write a device driver. But what if you want to do something unusual, to change the behavior of the system in some way? Then, you are mostly on your own.

Operating System

Every general-purpose computer must have an operating system to run other programs. Operating systems perform basic tasks, such as recognizing input from the keyboard, sending output to the display screen, keeping track of files and directories on the disk, and controlling peripheral devices such as disk drives and printers. For large systems, the operating system has even greater responsibilities and powers. It is like a traffic cop – it makes sure that different programs and users running at the same time do not interfere with each other. The operating system is also responsible for security, ensuring that unauthorized users do not access the system.

System calls provide an interface between the process and the operating system. System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do. It is because of the critical nature of operations that the operating system itself does them every time they are needed.



Example: For I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

The fact that improper use of the system can easily cause a system crash, thus the operating system is introduced; it executes at the highest level of order and allows the applications to request for a service – a system call – which is implemented through hooking interrupt(s). A system call is the mechanism used by an application program to request service from the operating system. There are different types of system call.

2.1 Operating System Functions

The main functions of operating systems are:

1. Process Management
2. Memory Management
3. Secondary Storage Management
4. I/O Management
5. File Management
6. Protection
7. Networking Management
8. Command Interpretation.

Process Management

The CPU executes a large number of programs. While its main concern is the execution of user programs, the CPU is also needed for other system activities. These activities are called processes. A process is a program in execution. Typically, a batch job is a process. A time-shared user program is a process. A system task, such as spooling, is also a process. For now, a process may be considered as a job or a time-shared program, but the concept is actually more general. The operating system is responsible for the following activities in connection with process management:

1. The creation and deletion of both user and system processes
2. The suspension and resumption of processes.
3. The provision of mechanisms for process synchronization
4. The provision of mechanisms for deadlock handling.

Memory Management

Memory is the most expensive part in the computer system. Memory is a large array of words or bytes, each with its own address. Interaction is achieved through a sequence of reads or writes of specific memory address. The CPU fetches from and stores in memory.

There are various algorithms that depend on the particular situation to manage the memory. Selection of a memory management scheme for a specific system depends upon many factors, but especially upon the hardware design of the system. Each algorithm requires its own hardware support.

The operating system is responsible for the following activities in connection with memory management.

1. Keep track of which parts of memory are currently being used and by whom.
2. Decide which processes are to be loaded into memory when memory space becomes available.
3. Allocate and deallocate memory space as needed.

Secondary Storage Management

The main purpose of a computer system is to execute programs. These programs, together with the data they access, must be in main memory during execution. Since the main memory is too small to permanently accommodate all data and program, the computer system must provide secondary storage to backup main memory. Most modern computer systems use disks as the primary on-line storage of information, of both programs and data. Most programs, like compilers, assemblers, sort routines, editors, formatters, and so on, are stored on the disk until loaded into memory, and then use the disk as both the source and destination of their processing. Hence the proper management of disk storage is of central importance to a computer system. There are few alternatives. Magnetic tape systems are generally too slow. In addition, they are limited to sequential access. Thus, tapes are more suited for storing infrequently used files, where speed is not a primary concern. The operating system is responsible for the following activities in connection with disk management:

1. Free space management
2. Storage allocation
3. Disk scheduling.

I/O Management

One of the purposes of an operating system is to hide the peculiarities or specific hardware devices from the user. For example, in UNIX, the peculiarities of I/O devices are hidden from the bulk of the operating system itself by the I/O system. The operating system is responsible for the following activities in connection to I/O management:

1. A buffer caching system
2. To activate a general device driver code
3. To run the driver software for specific hardware devices as and when required.

File Management

File management is one of the most visible services of an operating system. Computers can store information in several different physical forms: magnetic tape, disk, and drum are the most common forms. Each of these devices has its own characteristics and physical organisation. For convenient use of the computer system, the operating system provides a uniform logical view of information storage. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped, by the operating system, onto physical devices.

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic or alphanumeric. Files may be free-form, such as text files, or may be rigidly formatted. In general, a file is a sequence of bits, bytes, lines or records whose meaning is defined by its creator and user. It is a very general concept.

The operating system implements the abstract concept of the file by managing mass storage device, such as tapes and disks. Also, files are normally organized into directories to ease their use. Finally, when multiple users have access to files, it may be desirable to control by whom and in what ways files may be accessed.

The operating system is responsible for the following activities in connection to the file management:

1. The creation and deletion of files.
2. The creation and deletion of directory.

Operating System

3. The support of primitives for manipulating files and directories.
4. The mapping of files onto disk storage.
5. Backup of files on stable (non volatile) storage.
6. Protection and security of the files.

Protection

The various processes in an operating system must be protected from each other's activities. For that purpose, various mechanisms which can be used to ensure that the files, memory segment, CPU and other resources can be operated on only by those processes that have gained proper authorization from the operating system.



Example: Memory addressing hardware ensures that a process can only execute within its own address space. The timer ensures that no process can gain control of the CPU without relinquishing it. Finally, no process is allowed to do its own I/O, to protect the integrity of the various peripheral devices. Protection refers to a mechanism for controlling the access of programs, processes, or users to the resources defined by a computer controls to be imposed, together with some means of enforcement.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a subsystem that is malfunctioning. An unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user.

Networking

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with each other through various communication lines, such as high-speed buses or telephone lines. Distributed systems vary in size and function. They may involve microprocessors, workstations, minicomputers, and large general-purpose computer systems. The processors in the system are connected through a communication network, which can be configured in the number of different ways. The network may be fully or partially connected. The communication network design must consider routing and connection strategies and the problems of connection and security. A distributed system provides the user with access to the various resources the system maintains. Access to a shared resource allows computation speed-up, data availability, and reliability.

Command Interpretation

One of the most important components of an operating system is its command interpreter. The command interpreter is the primary interface between the user and the rest of the system. Many commands are given to the operating system by control statements. When a new job is started in a batch system or when a user logs-in to a time-shared system, a program which reads and interprets control statements is automatically executed.

This program is variously called

- (1) the control card interpreter,
- (2) the command line interpreter,
- (3) the shell (in Unix), and so on.

Its function is quite simple: get the next command statement, and execute it. The command statements themselves deal with process management, I/O handling, secondary storage management, main memory management, file system access, protection, and networking.

The Figure 2.1 depicts the role of the operating system in coordinating all the functions.

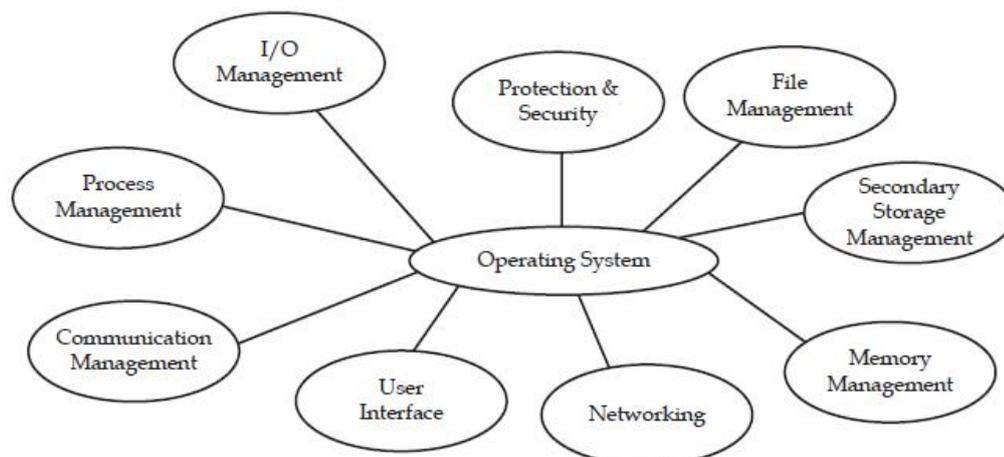


Figure 2.1: Functions Coordinated by the Operating System

2.2 System Calls

System calls provide an interface between a running program and operating system. System calls are generally available as assembly language instructions. Several higher-level languages such as C also allow to make system calls directly. In UNIX operating system the system call interface layer contains entry point in kernel code. All system resources are managed by the kernel. Any request from user or application that involves access to any system resource must be handled by kernel code. The user process must not be given open access to kernel code for security reason. Many openings into kernel code called system calls are provided to user so that the user processes can invoke the execution of kernel code. System calls allow processes and users to manipulate system resources.

In general, a process is not supposed to be able to access the kernel. It can't access kernel memory and it can't call kernel functions. The hardware of the CPU enforces this (that is the reason why it is called 'protected mode'). System calls are an exception to this general rule. What happens is that the process fills the registers with the appropriate values and then calls a special instruction which jumps to a previously defined location in the kernel (of course, that location is readable by user processes, it is not writable by them)? Under Intel CPUs, this is done by means of `int0x80`. The hardware knows that once you jump to this location, you are no longer running in restricted user mode, but as the operating system kernel and, therefore, you are allowed to do whatever you want.

The location in the kernel a process can jump to is called system call. The procedure at that location checks the system call number, which tells the kernel what service the process requested. Then, it looks at the table of system calls (`sys_call_table`) to see the address of the kernel function to call. Then it calls the function, and after it returns, does a few system checks and then return back to the process (or to a different process, if the process time ran out). If you want to read this code, it is at the source file `arch/$<$architecture>/kernel/entry`. So, after the line `ENTRY(systemically)`. So, if we want to change the way a certain system call works, what we need to do is to write our own function to implement it (usually by adding a bit of our own code, and then calling the original function) and then change the pointer at `sys_call_table` to point to our function. Because we might be removed later and we don't want to leave the system in an unstable state, it is important for `cleanup_module` to restore the table to its original state.

The source code here is an example of such a kernel module. We want to 'spy' on a certain user, and to `printk()` a message whenever that user opens a file. Towards this end, we replace the system call to open a file with our own function, called `our_sys_open`. This function checks the `uid` (user's id) of the current process, and if it is equal to the `uid` we spy on, it calls `printk()` to display the name of the file to be opened. Then, either way, it calls the original `open()` function with the same parameters, to actually open the file.

The `init_module` function replaces the appropriate location in `sys_call_table` and keeps the original pointer in a variable. The `cleanup_module` function uses that variable to restore everything back to normal. This approach is dangerous, because of the possibility of two kernel modules changing the same system call. Imagine we have two kernel modules, A and B. A's open system call will be A open and B's will be `B_open`. Now, when A is inserted into the kernel, the system call

is replaced with `A_open`, which will call the original `sys_open` when it is done. Next, `B` is inserted into the kernel, which replaces the system call with `B_open`, which will call what it thinks is the original system call, `A_open`, when it is done.

Now, if `B` is removed first, everything will be well it will simply restore the system call to `A_open`, which calls the original. However, if `A` is removed and then `B` is removed, the system will crash. `A`'s removal will restore the system call to the original, `sys_open`, cutting `B` out of the loop. Then, when `B` is removed, it will restore the system call to what it thinks is the original, `A_open`, which is no longer in memory. At first glance, it appears we could solve this particular problem by checking if the system call is equal to our open function and if so not changing it at all (so that `B` won't change the system call when it is removed), but that will cause an even worse problem. When `A` is removed, it sees that the system call was changed to `B_open` so that it is no longer pointing to `A_open`, so it won't restore it to `sys_open` before it is removed from memory. Unfortunately, `B_open` will still try to call `A_open` which is no longer there, so that even without removing `B` the system would crash.

There are two ways to prevent this problem. The first is to restore the call to the original value, `sys_open`. Unfortunately, `sys_open` is not part of the kernel system table in `/proc/kSYMS`, so we can't access it. The other solution is to use the reference count to prevent root from rmmod'ing the module once it is loaded. This is good for production modules, but bad for an educational sample which is why I didn't do it here.

it is no longer pointing to `A_open`, so it won't restore it to `sys_open` before it is removed from memory. Unfortunately, `B_open` will still try to call `A_open` which is no longer there, so that even without removing `B` the system would crash.

There are two ways to prevent this problem. The first is to restore the call to the original value, `sys_open`. Unfortunately, `sys_open` is not part of the kernel system table in `/proc/kSYMS`, so we can't access it. The other solution is to use the reference count to prevent root from rmmod'ing the module once it is loaded. This is good for production modules, but bad for an educational sample which is why I didn't do it here.

The Library as an Intermediary

Generally, systems provide a library that sits between normal programs and the operating system, usually an implementation of the C library (`libc`), such as `glibc`. This library exists between the OS and the application, and increases portability. On exokernel based systems, the library is especially important as an intermediary. On exokernels, libraries shield user applications from the very low-level kernel API, and provide abstractions and resource management.

Examples and Tools

On Unix, Unix-like and other POSIX-compatible Operating Systems, popular system calls are `open`, `read`, `write`, `close`, `wait`, `exec`, `fork`, `exit`, and `kill`. Many of today's operating systems have hundreds of system calls. For example, Linux has 319 different system calls. Similarly, FreeBSD has almost 330. Tools such as `Strace` and `Truss` allow a process to execute from start and report all system calls the process invokes, or can attach to an already running process and intercept any system call made by said process if the operation does not violate the permissions of the user. This special ability of the program is usually also implemented with a system call, e.g. the GNU's `Strace` is implemented with `ptrace()`.

Typical Implementations

Implementing system calls requires a control transfer which involves some sort of architecture specific feature. A typical way to implement this is to use a software interrupt or trap. Interrupts transfer control to the OS so software simply needs to set up some register with the system call number they want and execute the software interrupt. For many RISC processors this is the only feasible implementation, but CISC architectures such as x86 support additional techniques. One example is `SYSCALL/SYSENTER`, `SYSRET/SYSEXIT` (the two mechanisms were independently created by AMD and Intel, respectively, but in essence do the same thing). These are "fast" control transfer instructions that are designed to quickly transfer control to the OS for a system call without the overhead of an interrupt. Linux 2.5 began using this on the x86, where available; formerly it used the `INT` instruction, where the system call number was placed in the `EAX` register before interrupt `0x80` was executed. An older x86 mechanism is called a call gate and is a way for a program to literally call a kernel function directly using a safe control transfer mechanism the OS sets up in advance. This approach has been unpopular, presumably due to the requirement

of a far call which uses x86 memory segmentation and the resulting lack of portability it causes, and existence of the faster instructions mentioned above. For IA64 architecture, EPC (Enter Privileged Mode) instruction is used. The first eight system call arguments are passed in registers, and the rest are passed on the stack. A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable.

2.3 Types of System Call

System calls can be roughly grouped into five major categories:

- 1) Process Control – this involves system calls like end, abort, load, execute, create process, terminate process, get process attributes, set process attributes, wait for time, wait event, signal event, allocate and free memory
- 2) File Management – this involves system calls like create file, delete file, open, close, read, write, reposition, get file attributes and set file attributes
- 3) Device Management – this involves system calls like, request device, release device, read, write, reposition, get device attributes, set device attributes, logically attach or detach devices
- 4) Information Maintenance – this involves system calls like get time or date, set time or date, get system data, set system data, get process, file, or device attributes, set process, file, or device attributes
- 5) Communication – this involves system calls like create, delete communication connection send, receive messages, transfer status information, attach or detach remote devices

There are three general methods that are used to pass information (parameters) between a running program and the operating system.

1. One method is to store parameters in registers.
2. Another is to store parameters in a table in memory and pass the address of table.
3. The third method is to push parameters on stack and allow operating system to pop the parameters off the stack.

System Calls for Process Management

These types of system calls are used to control the processes. Some examples are end, abort, load, execute, create process, terminate process etc.



Example: The `exit()` system call ends a process and returns a value to its parent

In UNIX every process has an alarm clock stored in its system-data segment. When the alarm goes off, signal `SIGALRM` is sent to the calling process. A child inherits its parent's alarm clock value, but the actual clock isn't shared. The alarm clock remains set across an exec.

System Calls for Signaling

A signal is a limited form of inter-process communication used in UNIX, UNIX-like, and other POSIX-compliant operating systems. Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred. The number of signals available is system dependent. When a signal is sent to a process, the operating system interrupts the process' normal flow of execution. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise the default signal handler is executed.

Programs can respond to signals three different ways. These are:

1. **Ignore the signal:** This means that the program will never be informed of the signal no matter how many times it occurs.

Operating System

A signal can be set to its default state, which means that the process will be ended when it receives that signal.

2. **Catch the signal:** When the signal occurs, the system will transfer control to a previously defined subroutine where it can respond to the signal as is appropriate for the program.

System Calls for File Management

The file structure related system calls available in some operating system like UNIX let you create, open, and close files, read and write files, randomly access files, alias and remove files, get information about files, check the accessibility of files, change protections, owner, and group of files, and control devices. These operations either use a character string that defines the absolute or relative path name of a file, or a small integer called a file descriptor that identifies the I/O channel. When doing I/O, a process specifies the file descriptor for an I/O channel, a buffer to be filled or emptied, and the maximum size of data to be transferred. An I/O channel may allow input, output, or both. Furthermore, each channel has a read/write pointer. Each I/O operation starts where the last operation finished and advances the pointer by the number of bytes transferred. A process can access a channel's data randomly by changing the read/write pointer. These types of system calls are used to manage files.



Example: Create file, delete file, open, close, read, write etc.

System Calls for Directory Management

You may need the same sets of operations as for file management for directories also. If you have a directory structure for organizing files in the file system. In addition, for either files or directories, you need to be able to determine the values of various attributes, and perhaps to reset them if necessary. File attributes include the file name, a file type, protection codes, accounting information, and so on. At least two system calls, get file attribute and set file attribute, are required for this function. Some operating systems provide many more calls.

System Calls for Protection

Improper use of the system can easily cause a system crash. Therefore, some level of control is required; the design of the microprocessor architecture on basically all modern systems (except embedded systems) offers several levels of control - the (low privilege) level of which normal applications execute limits the address space of the program to not be able to access nor modify other running applications nor the operating system itself (called "protected mode" on x86), it also prevents the application from using any system devices (i.e. the frame buffer, network devices - any I/O mapped device). But obviously any normal application needs this ability, thus the operating system is introduced, it executes at the highest level of order and allows the applications to request for a service - a system call - which is implemented through hooking interrupt(s). If allowed the system enters a higher privileged state, executes a specific set of instructions which the interrupting program has no direct control over, then returns control to the former flow of execution. This concept also serves as a way to implement security. With the development of separate operating modes with varying levels of privilege, a mechanism was needed for transferring control safely from lesser privileged modes to higher privileged modes. Less privileged code could not simply transfer control to more privileged code at any arbitrary point and with any arbitrary processor state. To allow it to do so could allow it to break security. For instance, the less privileged code could cause the higher privileged code to execute in the wrong order, or provide it with a bad stack.

System Calls for Time Management

Many operating systems provide a time profile of a program. It indicates the amount of time that the program executes at a particular location or set of locations. A time profile requires either a tracing facility or regular timer interrupts. At every occurrence of the timer interrupt, the value of the program counter is recorded. With sufficiently frequent timer interrupts, a statistical picture of the time spent on various parts of the program can be obtained.

System Calls for Device Management

A program, as it is running, may need additional resources to proceed. Additional resources may be more memory, tape drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user program; otherwise, the program will have to wait until sufficient resources are available. These types of system calls are used to manage devices.

Example: Request device, release device, read, write, get device attributes etc.

A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. A system call is a way for programs to **interact with the operating system**. A computer program makes a system call when it makes a request to the operating system's kernel. System call provides the services of the operating system to the user programs via Application Program Interface (API). It provides an interface between a process and operating system to allow user-level processes to request services of the operating system. System calls are the only entry points into the kernel system. All programs needing resources must use system calls.

Mechanism to Use OS Services

The OS services act as an interface between a process and Operating system Example: System call sequence to copy the contents of one file to another file

Methods to Invoke System Call

System calls are invoked directly from a high-level language program. For example: C / C++ program can invoke a Unix system call.

Invoke a call to a special routine that makes a system call. In MS Windows, system calls are part of Win32 API (Application Programming Interface) which is available for use by all compilers written for MS Windows

Process - System Call - OS Relationship

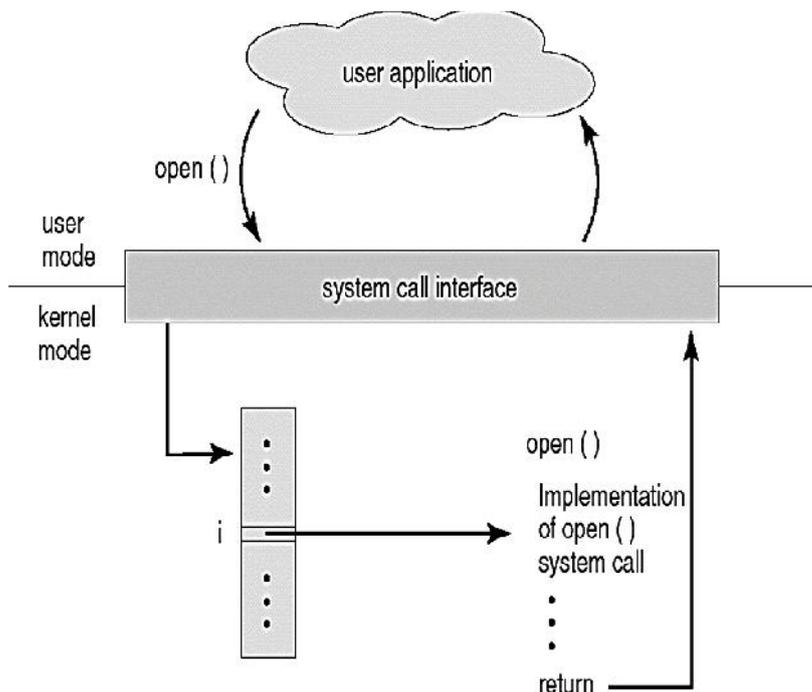


Figure 2.2: Relationship between process, system call and Operating system.

2.4 Standard C Library Example

The C program invoking printf() library call, which calls write() system call

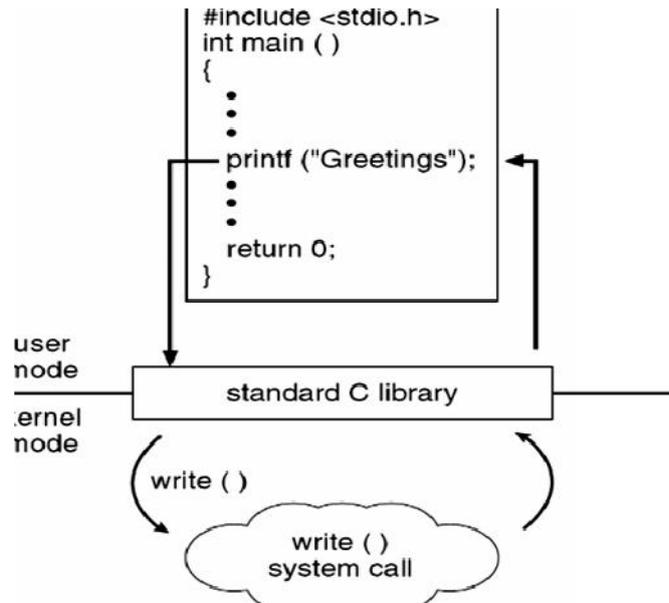
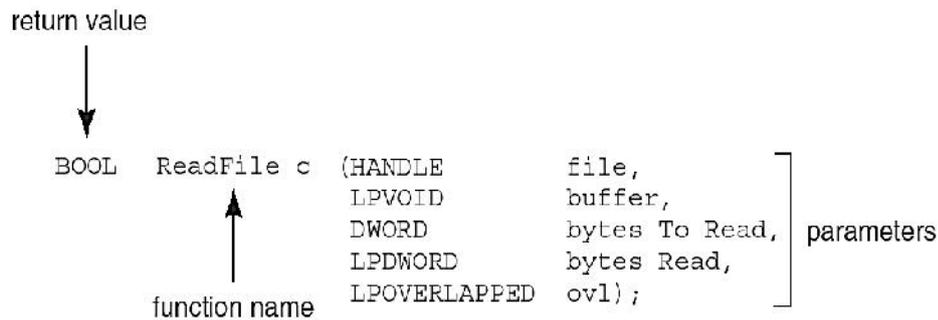


Figure 2.3: C program invoking printf() library call, which calls write() system call



Example of Standard API

Consider the ReadFile() function in the Win32 API—a function for reading from a file



Example of Standard API

A description of the parameters passed to ReadFile()

- HANDLE file – the file to be read
- LPVOID buffer – a buffer where the data will be read into and written from
- DWORD bytesToRead – the number of bytes to be read into the buffer
- LPDWORD bytesRead – the number of bytes read during the last read
- LPOVERLAPPED ovl – indicates if overlapped I/O is being used

System Call Implementation

Typically, a number associated with each system call. System-call interface maintains a table indexed according to these numbers. The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values. The caller need not to know nothing about how the system call is implemented. Just needs to obey API and understand what OS will do as a result call. Most details of OS interface hidden from programmer by API. It is managed by run-time support library (set of functions built into libraries included with compiler).

System Call Parameter Passing

Information required to invoke system call includes the system call name and other parameters depending on the task to perform

Methods used to pass parameters to the OS include:

1. Using registers (Simplest) - it should be noted that this method is not useful when more parameters than registers
2. Stored in a block, or table, in memory, and address of block passed as a parameter in a register. This approach taken by Linux and Solaris
3. Parameters pushed onto the stack by the program and popped off the stack by the operating system
4. Block and stack methods do not limit the number or length of parameters being passed

Parameter Passing via Table

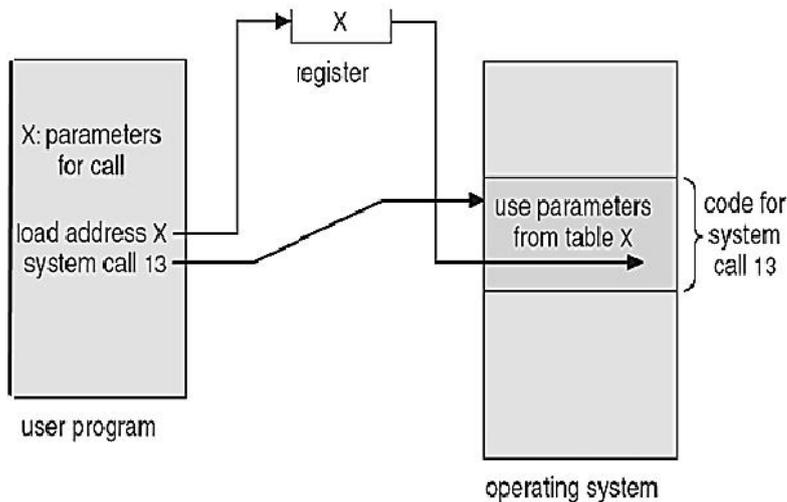


Figure 2.4: Parameter passing via table.

Types of System Calls

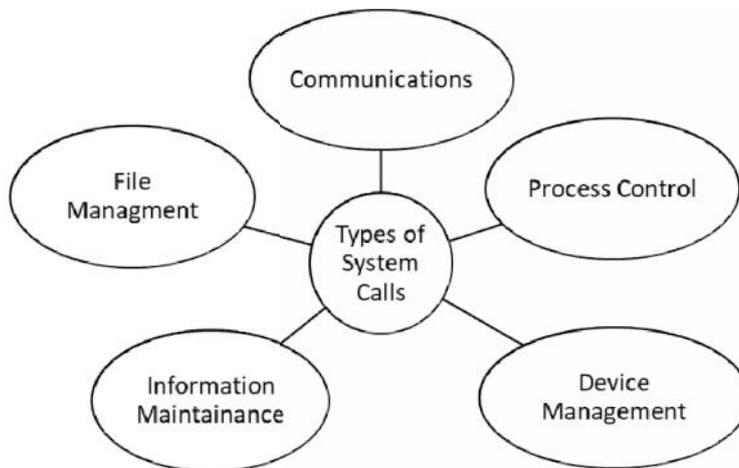


Figure 2.5: Types of System Calls

1. Process control

These system calls deal with processes such as process creation, process termination etc. Examples of process control system calls are: end, abort, create, terminate, allocate and free memory.

2. File Management

These system calls are responsible for file manipulation such as creating a file, reading a file, writing into a file etc. Examples of file management system calls are: create, open, close, delete, read file etc.

3. Device Management

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

4. Information Maintenance

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

5. Communications

These system calls are useful for inter-process communication. They also deal with creating and deleting a communication connection. Some of the examples of all the above types of system calls in Windows and Unix are given in the following table:

Table 2.1: Types of System Calls in Windows and Linux

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

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

`wait()`, `exec()`, `fork()`, `exit()` and `kill()`

wait()

In some systems, a process may wait for another process to complete its execution. This happens when a parent process creates a child process and the execution of the parent process is suspended until the child process executes. The suspending of the parent process occurs with a `wait()` system call. When the child process completes execution, the control is returned back to the parent process.

exec()

This system call runs an executable file in the context of an already running process. It replaces the previous executable file. This is known as an overlay. The original process identifier remains since a new process is not created but data, heap, stack etc. of the process are replaced by the new process.

fork()

Processes use the `fork()` system call to create processes that are a copy of themselves. This is one of the major methods of process creation in operating systems. When a parent process creates a child process and the execution of the parent process is suspended until the child process executes. When the child process completes execution, the control is returned back to the parent process.

exit()

The `exit()` system call is used by a program to terminate its execution. In a multithreaded environment, this means that the thread execution is complete. The operating system reclaims resources that were used by the process after the `exit()` system call.

kill()

The `kill()` system call is used by the operating system to send a termination signal to a process that urges the process to exit. However, `kill` system call does not necessarily mean killing the process and can have various meanings.

System Programs

Another aspect of a modern system is the collection of system programs. In the logical computer hierarchy the lowest level is hardware. Next is the operating system, then the system programs, and finally the application programs. System programs provide a convenient environment for program development and execution. Some of them are simply user interfaces to system calls; others are considerably more complex.

They can be divided into these categories:

1. **File management:** These programs create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.
2. **Status information:** Some programs simply ask the system for the date, time, amount of available memory or disk space, number of users, or similar status information. That information is then formatted, and is printed to the terminal or other output device or file.
3. **File modification:** Several text editors may be available to create and modify the content of files stored on disk or tape.
4. **Programming-language support:** Compilers, assemblers, and interpreters for common programming languages (such as C, C++, Java, Visual Basic, and PERL) are often provided to the user with the operating system. Some of these programs are now priced and provided separately.
5. **Program loading and execution:** Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide absolute loaders, relocatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed also.
6. **Communications:** These programs provide the mechanism for creating virtual connections among processes, users, and different computer systems. They allow users to send messages to one another's screens, to browse web pages, to send electronic-mail messages, to login remotely, or to transfer files from one machine to another.

Most operating systems are supplied with programs that solve common problems, or perform common operations. Such programs include web browsers, word processors and text formatters, spreadsheets, database systems, compiler compilers, plotting and statistical-analysis packages, and games. These programs are known as system utilities or application programs.

Perhaps the most important system program for an operating system is the command interpreter, the main function of which is to get and execute the next user-specified command. Many of the commands given at this level manipulate files: create, delete, list, print, copy, execute, and so on. These commands can be implemented in two general ways. In one approach, the command interpreter itself contains the code to execute the command.

Example: A command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code. An alternative approach—used by UNIX, among other operating systems—implements most commands by system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed. Thus, the UNIX command to delete a file.

2.5 Operating System Structure

The operating system structure is a container for a collection of structures for interacting with the operating system's file system, directory paths, processes, and I/O subsystem. The types and functions provided by the operating system substructures are meant to present a model for handling these resources that is largely independent of the operating system. There are different types of structure as described in Figure 3.1.

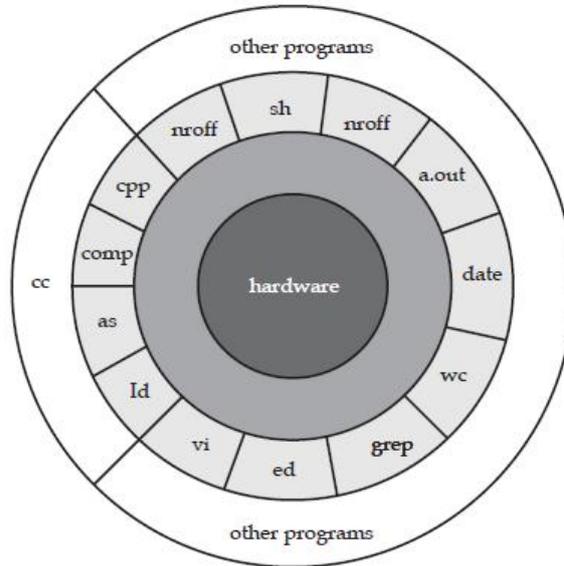


Figure 2.6: UNIX Operating System

It's very common to find pictures like Figure 3.2 below that describe the basic structure of an operating system.

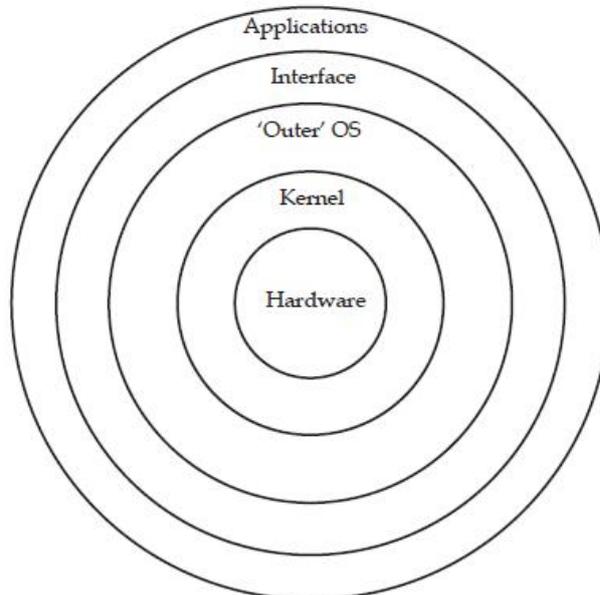


Figure 2.7: Operating System Structure Concepts

You might find that some versions of this have different numbers of rings. What does each part represent?

1. **Hardware:** The hardware is, obviously, the physical hardware and not particularly interesting to us in this module.
2. **Kernel:** The kernel of an operating system is the bottom-most layer of software present on a machine and the only one with direct access to the hardware. The code in the kernel is the most

'trusted' in the system - and all requests to do anything significant must go via the kernel. It provides the most key facilities and functions of the system.

3. **Outer OS:** Surrounding the kernel are other parts of the operating system. These perform less critical functions - for example, the graphics system which is ultimately responsible for what you see on the screen.

4. **Interface:** The interface provides a mechanism for you to interact with the computer.

5. **Applications:** There are what do the actual work - they can be complex (for example Office) or simple (for example the ls command commonly found on UNIX and Linux systems that lists files in a directory (or folder)).

2.6 Monolithic Systems

This approach is well known as "The Big Mess". The operating system is written as a collection of procedures, each of which can call any of the other ones whenever it needs to. When this technique is used, each procedure in the system has a well-defined interface in terms of parameters and results, and each one is free to call any other one, if the latter provides some useful computation that the former needs.

For constructing the actual object program of the operating system when this approach is used, one compiles all the individual procedures, or files containing the procedures, and then binds them all together into a single object file with the linker. In terms of information hiding, there is essentially none - every procedure is visible to every other one i.e. opposed to a structure containing modules or packages, in which much of the information is local to module, and only officially designated entry points can be called from outside the module.

However, even in Monolithic systems, it is possible to have at least a little structure. The services like system calls provided by the operating system are requested by putting the parameters in well-defined places, such as in registers or on the stack, and then executing a special trap instruction known as a kernel call or supervisor call.

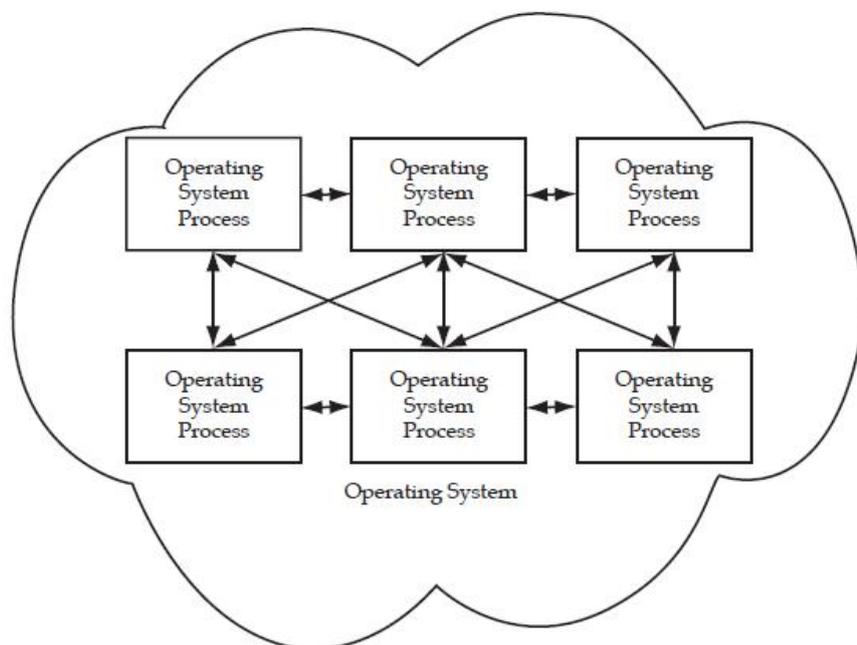


Figure 2.8: Monolithic Systems

2.7 Client-server Model

A trend in modern operating systems is to take this idea of moving code up into higher layers even further, and remove as much as possible from the operating system, leaving a minimal kernel. The usual approach is to implement most of the operating system functions in user processes. To request a service, such as reading a block of a file, a user process (presently known as the client process) sends the request to a server process, which then does the work and sends back the answer.

In Client-server Model, all the kernel does is handle the communication between clients and servers. By splitting the operating system up into parts, each of which only handles one fact of the system, such as file service, process service, terminal service, or memory service, each part becomes small and manageable; furthermore, because all the servers run as user-mode processes, and not in kernel mode, they do not have direct access to the hardware. As a consequence, if a bug in the file server is triggered, the file service may crash, but this will not usually bring the whole machine down.

Another advantage of the client-server model is its adaptability to use in distributed system. If a client communicates with a server by sending it messages, the client need not know whether the message is handled locally in its own machine, or whether it was sent across a network to a server on a remote machine. As far as the client is concerned, the same thing happens in both cases: a request was sent and a reply came back.

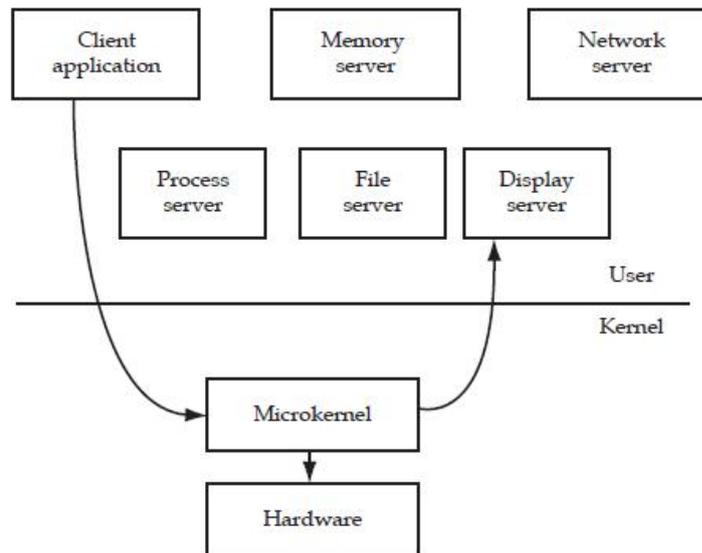


Figure 2.9: Client-Server Model

2.8 Exokernel

Exokernel is an operating system kernel developed by the MIT Parallel and Distributed Operating Systems group, and also a class of similar operating systems. The idea behind exokernel is to force as few abstractions as possible on developers, enabling them to make as many decisions as possible about hardware abstractions. Applications may request specific memory addresses, disk blocks, etc. The kernel only ensures that the requested resource is free, and the application is allowed to access it. This low-level hardware access allows the programmer to implement custom abstractions, and omit unnecessary ones, most commonly to improve a program's performance. It also allows programmers to choose what level of abstraction they want, high, or low.

Exokernels can be seen as an application of the end-to-end principle to operating systems, in that they do not force an application program to layer its abstractions on top of other abstractions that were designed with different requirements in mind.

Example: In the MIT Exokernel project, the Cheetah web server stores preformatted

Internet Protocol packets on the disk, the kernel provides safe access to the disk by preventing unauthorized reading and writing, but how the disk is abstracted is up to the application or the libraries the application uses.

Operating systems define the interface between applications and physical resources. Unfortunately, this interface can significantly limit the performance and implementation freedom of applications. Traditionally, operating systems hide information about machine resources behind high-level abstractions such as processes, files, address spaces and inter-process communication. These abstractions define a virtual machine on which applications execute; their implementation cannot be replaced or modified by untrusted applications.

Hardcoding the implementations of these abstractions is inappropriate for three main reasons:

1. It denies applications the advantages of domain-specific optimizations,

2. It discourages changes to the implementations of existing abstractions, and
3. It restricts the flexibility of application builders, since new abstractions can only be added by awkward emulation on top of existing ones (if they can be added at all).

These problems can be solved through application level resource management in which traditional operating system abstractions, such as Virtual Memory (VM) and Inter-process Communication (IPC), are implemented entirely at application level by untrusted software. In this architecture, a minimal kernel-called an exokernel-securely multiplexes available hardware resources. Library operating systems, working above the exokernel interface, implement higher-level abstractions. Application writers select libraries or implement their own. New implementations of library operating systems are incorporated by simply relinking application executables. Applications can benefit greatly from having more control over how machine resources are used to implement higher-level abstractions. The high cost of general-purpose virtual memory primitives reduce the performance of persistent stores, garbage collectors, and distributed shared memory systems.

Application-level control over file caching can reduce application-running time considerably. Application-specific virtual memory policies can increase application performance. The inappropriate file-system implementation decisions can have a dramatic impact on the performance of databases. The exceptions can be made an order of magnitude faster by deferring signal handling to applications.

To provide applications control over machine resources, an exokernel defines a low-level interface. The exokernel architecture is founded on and motivated by a single, simple, and old observation that the lower the level of a primitive, the more efficiently it can be implemented, and the more latitude it grants to implementors of higher-level abstractions.

To provide an interface that is as low-level as possible (ideally, just the hardware interface), an exokernel designer has a single overriding goal of separating protection from management. For instance, an exokernel should protect frame buffers without understanding windowing systems and disks without understanding file systems.

One approach is to give each application its own virtual machine. Virtual machines can have severe performance penalties. Therefore, an exokernel uses a different approach - it exports hardware resources rather than emulating them, which allows an efficient and simple implementation. An exokernel employs three techniques to export resources securely:

1. By using secure bindings, applications can securely bind to machine resources and handle events.
2. By using visible re-source revocation, applications participate in a resource revocation protocol.
3. By using an abort protocol, an exokernel can break secure bindings of uncooperative applications by force.

The advantages of exokernel systems among others are:

1. Exokernels can be made efficient due to the limited number of simple primitives they must provide
2. Low-level secure multiplexing of hardware resources can be provided with low overhead
3. Traditional abstractions, such as VM and IPC, can be implemented efficiently at application level, where they can be easily extended, specialized, or replaced
4. Applications can create special-purpose implementations of abstractions, tailored to their functionality and performance needs.

Finally, many of the hardware resources in microkernel systems, such as the network, screen, and disk, are encapsulated in heavyweight servers that cannot be bypassed or tailored to application specific needs. These heavyweight servers can be viewed as fixed kernel subsystems that run in the user-space.

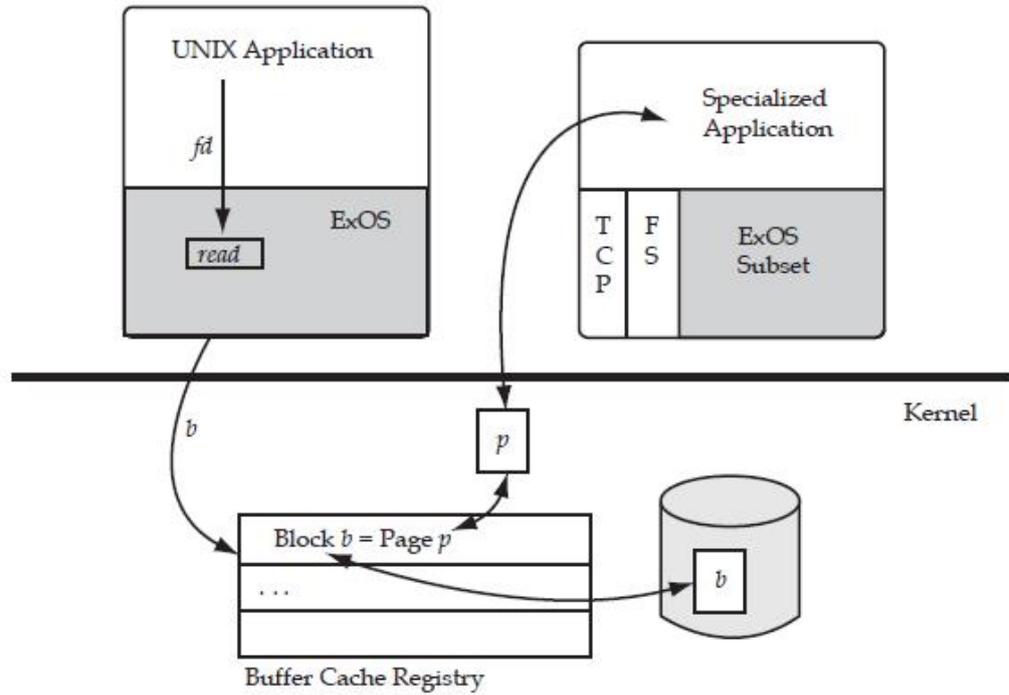


Figure 2.10: Exokernel

2.9 Layered Structure

A generalization of the approach as shown in the Figure 3.6 for organizing the operating systems as a hierarchy of layers, each one constructed upon the one below it.

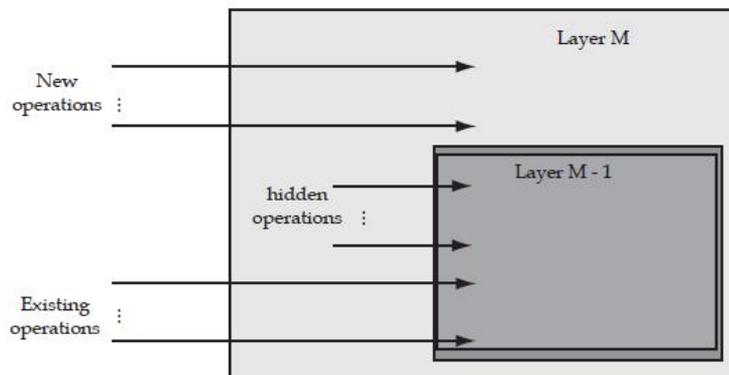


Figure 2.11: An Operating System Layer

The system has 6 layers. Layer 0 dealt with allocation of the processor, switching between processes when interrupts occurred or timers expired. Above layer 0, the system consisted of sequential processes, each of which could be programmed without having to worry about the fact that multiple processes were running on a single processor. In other words, layer 0 provided the basic multiprogramming of the CPU.

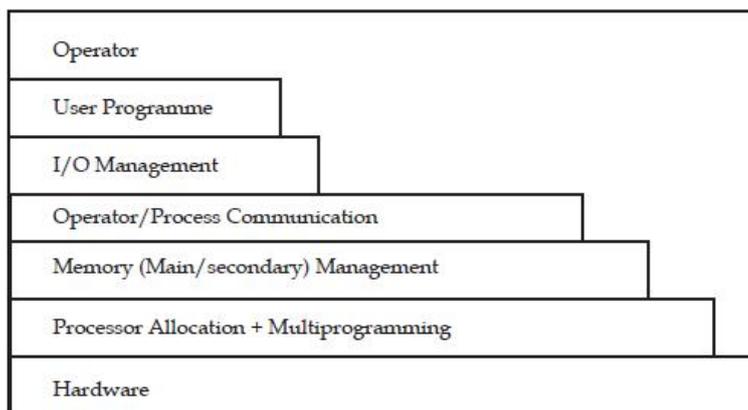


Figure 2.12: Layered System

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

Layer 2 handled communication between each process and the operator console. Above this layer each process effectively had its own operator console.

Layer 3 took care of managing the I/O devices and buffering the information streams to and from them. Above layer 3 each process could deal with abstract I/O devices with nice properties, instead of real devices with many peculiarities.

Layer 4 was where the user programs were found. They did not have to worry about process, memory, console, or I/O management.

The system operator process was located in layer 5.

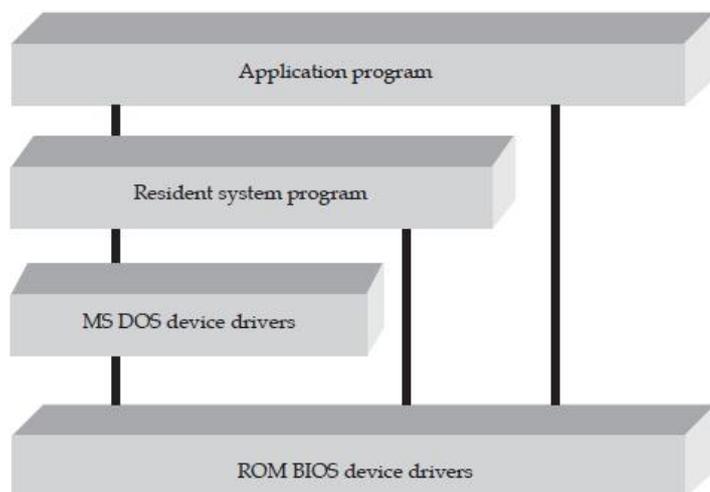


Figure 2.13: MS - DOS Layer Structure

In MS-DOS, the interfaces and levels of functionality are not well separated. For instance, application programs are able to access the basic I/O routines to write directly to the display and disk drives. Such freedom leaves MS-DOS vulnerable to errant (or malicious) programs, causing entire system crashes when user programs fail. Of course, MS-DOS was also limited by the hardware of its era. Because the Intel 8088 for which it was written provides no dual mode and no hardware protection, the designers of MS-DOS had no choice but to leave the base hardware accessible.

Operating System

The main advantage of the layered approach is modularity. The layers are selected such that each uses functions (operations) and services of only lower level layers. This approach simplifies debugging and system verification. The first layer can be debugged without any concern for the rest of the system, because, by definition, it uses only the basic hardware (which is assumed correct) to implement its functions. Once the first layer is debugged, its correct functioning can be assumed while the second layer is worked on, and so on. If an error is found during the

debugging of a particular layer, we know that the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system is simplified when the system is broken down into layers.

Each layer is implemented using only those operations provided by lower level layers. A layer does not need to know how these operations are implemented; it needs to know only what these operations do. Hence, each layer hides the existence of certain data structures, operations, and hardware from higher-level layers.

The layer approach to design was first used in the operating system at the Technische Hogeschool Eindhoven. The system was defined in six layers. The bottom layer was the hardware. The next layer implemented CPU scheduling. The next layer implemented memory management; the memory-management scheme was virtual memory. Layer 3 contained device driver for the operator's console. Because it and I/O buffering (level 4) were placed above memory management, the device buffers could be placed in virtual memory. The I/O buffering was also above the operator's console, so that I/O error conditions could be output to the operator's console. This approach can be used in many ways. For example, the Venus system was also designed using a layered approach. The lower layers (0 to 4), dealing with CPU scheduling and memory management, were then put into microcode. This decision provided the advantages of additional speed of execution and a clearly defined interface between the microcoded layers and the higher layers.

The major difficulty with the layered approach involves the appropriate definition of the various layers. Because a layer can use only those layers that are at a lower level, careful planning is necessary.

Example: The device driver for the backing store (disk space used by virtual-memory algorithms) must be at a level lower than that of the memory-management routines, because memory management requires the ability to use the backing store. Other requirements may not be so obvious. The backing-store driver would normally be above the CPU scheduler, because the driver may need to wait for I/O and the CPU can be rescheduled during this time. However, on a large system, the CPU scheduler may have more information about all the active processes than can fit in memory. Therefore, this information may need to be swapped in and out of memory, requiring the backing-store driver routine to be below the CPU scheduler.

A final problem with layered implementations is that they tend to be less efficient than other types. For instance, for a user program to execute an I/O operation, it executes a system call which is trapped to the I/O layer, which calls the memory-management layer, through to the CPU scheduling layer, and finally to the hardware. At each layer, the parameters may be modified, data may need to be passed, and so on. Each layer adds overhead to the system call and the net result is a system call that takes longer than one does on a non-layered system. These limitations have caused a small backlash against layering in recent years. Fewer layers with more functionality are being designed, providing most of the advantages of modularized code while avoiding the difficult problems of layer definition and interaction. For instance, OS/2 is a descendant of MS-DOS that adds multitasking and dual-mode operation, as well as other new features.

Because of this added complexity and the more powerful hardware for which OS/2 was designed, the system was implemented in a more layered fashion. Contrast the MS-DOS structure to that of the OS/2. It should be clear that, from both the system-design and implementation standpoints, OS/2 has the advantage. For instance, direct user access to low-level facilities is not allowed, providing the operating system with more control over the hardware and more knowledge of which resources each user program is using. As a further example, consider the history of Windows NT. The first release had a very layer-oriented organization. However, this version suffered low performance compared to that of Windows 95. Windows NT 4.0 redressed some of these performance issues by moving layers from user space to kernel space and more closely integrating them.

2.10 Virtual Machine

A virtual machine is a type of computer application used to create a virtual environment, which is referred to as virtualization. Virtualization allows the user to see the infrastructure of a network through a process of aggregation. Virtualization may also be used to run multiple operating systems at the same time. Through the help of a virtual machine, the user can operate software located on the computer platform.

There are different types of virtual machines. Most commonly, the term is used to refer to hardware virtual machine software, also known as a hypervisor or virtual machine monitor. This type of virtual machine software makes it possible to perform multiple identical executions on one computer. In turn, each of these executions runs an operating system. This allows multiple applications to be run on different operating systems, even those they were not originally intended for.

Virtual machine can also refer to application virtual machine software. With this software, the application is isolated from the computer being used. This software is intended to be used on a number of computer platforms. This makes it unnecessary to create separate versions of the same software for different operating systems and computers. Java Virtual Machine is a very well-known example of an application virtual machine.

A virtual machine can also be a virtual environment, which is also known as a virtual private server. A virtual environment is used for running programs at the user level. Therefore, it is used solely for applications and not for drivers or operating system kernels.

A virtual machine may also be a group of computers that work together to create a more powerful machine. In this type of virtual machine, the software makes it possible for one environment to be formed throughout several computers. This makes it appear to the end user as if he or she is using a single computer, when there are actually numerous computers at work.

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

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

Conceptually, a computer system is made up of layers. The hardware is the lowest level in all such systems. The kernel running at the next level uses the hardware instructions to create a set of system calls for use by outer layers. The systems programs above the kernel are therefore able to use either system calls or hardware instructions, and in some ways these programs do not differentiate between these two. Thus, although they are accessed differently, they both provide functionality that the program can use to create even more advanced functions. System programs, in turn, treat the hardware and the system calls as though they both are at the same level.

Some systems carry this scheme even a step further by allowing the system programs to be called easily by the application programs. As before, although the system programs are at a level higher than that of the other routines, the application programs may view everything under them in the hierarchy as though the latter were part of the machine itself. This layered approach is taken to its logical conclusion in the concept of a virtual machine. The VM operating system for IBM systems is the best example of the virtual-machine concept, because IBM pioneered the work in this area. By using CPU scheduling and virtual-memory techniques, an operating system can create the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.

Of course, normally, the process has additional features, such as system calls and a file system, which are not provided by the bare hardware. The virtual-machine approach, on the other hand, does not provide any additional function, but rather provides an interface that is identical to the underlying bare hardware. Each process is provided with a (virtual) copy of the underlying computer.

The resources of the physical computer are shared to create the virtual machines. CPU scheduling can be used to share the CPU and to create the appearance that users have their own processor.

Spooling and a file system can provide virtual card readers and virtual line printers. A normal user timesharing terminal provides the function of the virtual machine operator's console.

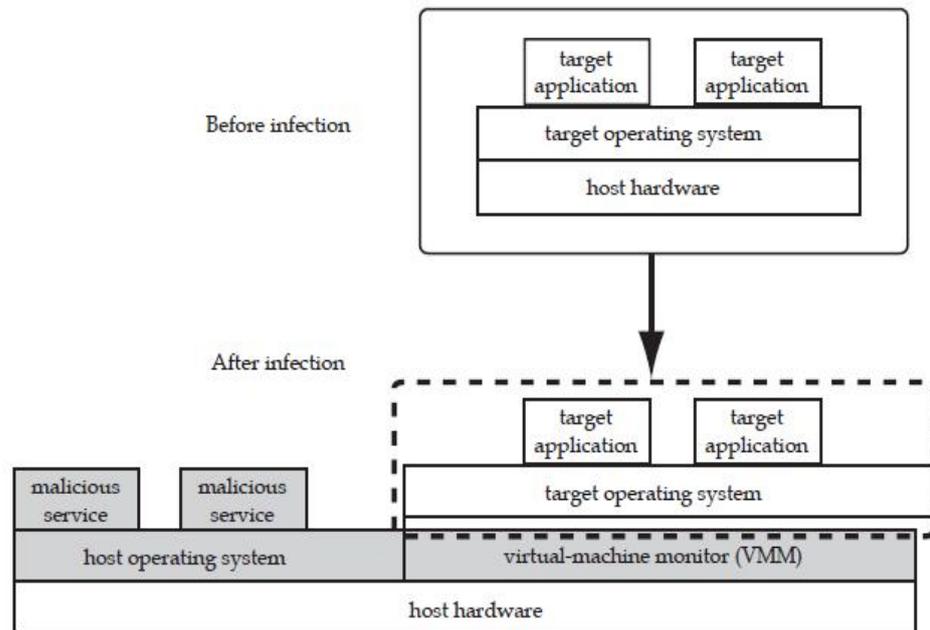


Figure 2.14: Virtual Machine

A major difficulty with the virtual-machine approach involves disk systems. Suppose that the physical machine has three disk drives but wants to support seven virtual machines. Clearly, it cannot allocate a disk drive to each virtual machine. Remember that the virtual-machine software itself will need substantial disk space to provide virtual memory and spooling. The solution is to provide virtual disks, which are identical in all respects except size; these are termed minidisks in IBM's VM operating system. The system implements each minidisk by allocating as many tracks as the minidisk needs on the physical disks. Obviously, the sum of the sizes of all minidisks must be less than the actual amount of physical disk space available.

Users thus are given their own virtual machine. They can then run any of the operating systems or software packages that are available on the underlying machine. For the IBM VM system, a user normally runs CMS, a single-user interactive operating system. The virtual-machine software is concerned with multiprogramming multiple virtual machines onto a physical machine, but does not need to consider any user-support software. This arrangement may provide a useful partitioning of the problem of designing a multiuser interactive system into two smaller pieces.

Summary

The operating system provides an environment by hiding the details of underlying hardware where the user can conveniently run programs. All the user sees is that the I/O has been performed without any details. The output of a program may need to be written into new files or input taken from some files. It involves secondary storage management. The user does not have to worry about secondary storage management. There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes.

Keywords

Asymmetric Multiprocessing: Asymmetric hardware systems commonly dedicated individual processors to specific tasks.

Computer Server System: Computer-server systems provide an interface to which clients can send requests to perform an action, in response to which they execute the action and send back results to the client.

Operating System: An operating system (OS) is a software program that manages the hardware and software resources of a computer.

Peer-to-Peer System: Peer-to-peer (P2P) computing or networking is a distributed application architecture that partitions tasks or workloads between peers.

Real Time Operating System (RTOS): Real-time operating systems are used to control machinery, scientific instruments and industrial systems such as embedded systems.

SIMD Multiprocessing: In a single instruction stream, multiple data stream computer one processor handles a stream of instructions, each one of which can perform calculations in parallel on multiple data locations.

Symmetric Multiprocessing: SMP involves a multiprocessor computer architecture where two or more identical processors can connect to a single shared main memory.

System Calls: System call is the mechanism used by an application program to request service from the operating system.

Self Assessment

1. A/an is a software program that manages the hardware and software resources of a computer.
2. Operating systems can be explored from two viewpoints: the and the system.
3. A manages the execution of user programs.
4. A is the mechanism used by an application program to request service from the operating system.
5. A is a program in execution.
 - A. process
 - B. file
 - C. system
 - D. None of the choices
6. The operating system is responsible for which of the following activities in connections with memory management?
 - I) Keep track of the parts of memory currently being used and by whom.
 - II) Decide which processes to load when memory space becomes available.
 - III) Allocate and deallocate memory space as needed.
 - IV) Support of primitives for manipulating files and directories.
 - A. I, II and IV
 - B. I, II and III
 - C. II, III and IV
 - D. I, III and IV
7. A is a collection of related information defined by its creator.
 - A. File
 - B. Storage
 - C. Record
 - D. File

8. The operating system manages
 - A. Memory
 - B. Processes
 - C. Disks and I/O devices
 - D. All of the above

9. The key components of the I/O system do not consist of:
 - A. A buffer-caching system.
 - B. A general device-driver interface.
 - C. Drivers for specific hardware devices.
 - D. File backup on stable (nonvolatile) storage media.

10. The operating system is not responsible for which of the following activities in connection with disk management:
 - A. Free space management
 - B. Buffer caching system
 - C. Storage allocation
 - D. Disk scheduling

11. Which of the following statements is not true?
 - A. A distributed system is a collection of processors that share memory or a clock.
 - B. Each processor has its own local memory.
 - C. The processors in the system are connected through a communication network.
 - D. Communication takes place using a protocol.

12. is a large array of words or bytes, each with its own address.
 - A. Storage
 - B. Memory
 - C. File
 - D. Record

13. A provides the user with access to the various resources the system maintains.

14. What is a shell?
 - A. It is a hardware component
 - B. It is a command interpreter
 - C. It is a part in compiler
 - D. It is a tool in CPU scheduling

15. Multiprogramming systems
 - A. Are easier to develop than single programming systems
 - B. Execute each job faster
 - C. Execute more jobs in the same time
 - D. Are used only on large main frame computers

16. Which of the following operating systems is better for implementing a Client-Server network?

- A. MS DOS
- B. Windows 95
- C. Windows 98
- D. Windows 2000

17. Which of the following is not an example of the multi-programming operating system?

- A. Atlas supervisor
- B. Multics Operating System
- C. Exec II system
- D. All of the given choices

Answers for Self Assessment

- | | | | | |
|---------------------|---------|------------------------|----------------|-------|
| 1. operating system | 2. user | 3. control program | 4. system call | 5. A |
| 6. B | 7. A | 8. D | 9. D | 10. B |
| 11. A | 12. B | 13. distributed system | 14. B | 15. C |
| 16. D | 17. B | | | |

Review Questions

1. Discuss the Simple Operating System Structure. Describe the layered approach.
2. What are the services that the operating system provides to both the users and to the programs?
3. What is system calls in OS? Explain in detail with its types.
4. Explain operating system functions and services in detail.
5. "With the help of a virtual machine, the user can operate software located on the computer platform". Do you agree with the statement? Give reasons to support your answer.



Further Readings

Operating Systems, by Harvey M. Deitel , Paul J. Deitel, David R. Choffnes.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull



Web Links

wiley.com/coolege.silberschatz

Unit 03: Process

CONTENTS

Objectives

Introduction

- 3.1 Process Concepts
- 3.2 Processes Creation
- 3.3 Process State Transitions
- 3.4 Process Termination
- 3.5 Inter-Process Communication
- 3.6 Process Communication in Client-Server Environment
- 3.7 Concept of Thread
- 3.8 Multi-threading
- 3.9 Multi-tasking vs. Multi-threading
- 3.10 Threading Issues
- 3.11 Processes vs. Threads

Summary

Keywords

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Explain process concepts
- Define PCB
- Describe operation on processes
- Explain inter-process communication
- Describe concept of thread

Introduction

Earlier a computer was used to be fasten the jobs pertaining to computation diligently and incessantly for a single person. Soon it was realized that the computer was far more powerful than just carrying out a single man's single job. Such was the speed of operation that the CPU would sit idle for most of the time awaiting user input. The CPU was certainly capable of carrying out many jobs simultaneously. It could also support many users simultaneously. But, the operating systems then available were not capable of this support. The operating systems facilitating a single-user support at a time was felt inadequate. Then a mechanism was developed which would prevent the wastage of CPU cycles. Hence multi-tasking systems were developed. In a multi-tasking system, a job or task is submitted as what is known as a process. Multi-tasking operating systems could handle multiple processes on a single processor. Process is a unit of program execution that enables the systems to implement multi-tasking behavior. Most of the operating systems today have multi-processing capabilities. This unit is dedicated to process and

process related issues. In this unit, present and discuss the mechanisms that support or enforce more structured forms of inter-process communications. Subsequent sections are devoted to messages, an extremely versatile and popular mechanism in both centralized and distributed systems, and to facilitate inter-process communication and synchronization.

3.1 Process Concepts

An operating system manages each hardware resource attached with the computer by representing it as an abstraction. An abstraction hides the unwanted details from the users and programmers allowing them to have a view of the resources in the form, which is convenient to them. A process is an abstract model of a sequential program in execution. The operating system can schedule a process as a unit of work. The term "process" was first used by the designers of the MULTICS in 1960's. Since then, the term "process" is used somewhat interchangeably with 'task' or 'job'. The process has been given many definitions as mentioned below:

1. A program in Execution.
2. An asynchronous activity.
3. The 'animated spirit' of a procedure in execution.
4. The entity to which processors are assigned.
5. The 'dispatchable' unit.

Though there is no universally agreed upon definition, but the definition "Program in Execution" is the one that is most frequently used. And this is a concept you will use in the present study of operating systems. Now the question is - what is the relation between process and program. It is same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process. Well, to be very precise.

Process is not the same as program. A process is more than a program code. A process is an 'active' entity as oppose to program which is considered to be a 'passive' entity. As you all know that a program is an algorithm expressed with the help of a programming language. A program is a passive entity sitting on some secondary storage device.

A process is an activity of executing a program. Basically, it is a program under execution. Every process needs certain resources to complete its task.

Process, on the other hand, includes:

1. Current value of Program Counter (PC)
2. Contents of the processor's registers
3. Value of the variables
4. The process-stack (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
5. A data section that contains global variables.
6. A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multi-programming).

A process includes, besides instructions to be executed, the temporary data such as subroutine parameters, return addresses and variables (stored on the stack), data section having global variables (if any), program counter value, register values and other associated resources. Although two processes may be associated with the same program, yet they are treated as two separate processes having their respective set of resources.

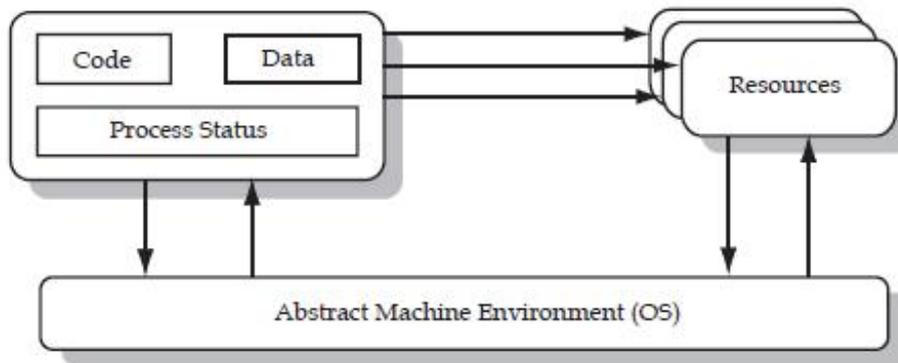


Figure: A Schematic Representation of a Process

Operation on Processes

Modern operating systems, such as UNIX, execute processes concurrently. Although there is a single Central Processor (CPU), which executes the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing a single process a few hundred microseconds of CPU time before replacing it with another process.)

Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system. The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU: the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPU's registers, and other data), and removes the process from the use of the CPU. The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this. The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

Process Control Block (PCB)

It is a data structure (a table) that holds information about a process. Whenever a process is created (initialized, installed), the operating system creates a corresponding process control block to serve as its run-time description during the lifetime of the process. When the process terminates, its PCB is released to the pool of free cells from which new PCBs are drawn.

Process States

During the lifespan of a process, its execution status may be in one of four states (associated with each state is usually a queue on which the process resides):

- **Executing:** The process is currently running and has control of a CPU.
- **Waiting:** The process is currently able to run, but must wait until a CPU becomes available.
- **Blocked:** The process is currently waiting on I/O, either for input to arrive or output to be sent.
- **Suspended:** The process is currently able to run, but for some reason the OS has not placed the process on the ready queue.
- **Ready:** The process is in memory, will execute given CPU time.

- Terminated:** The process has finished execution. These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems more finely delineate process states.

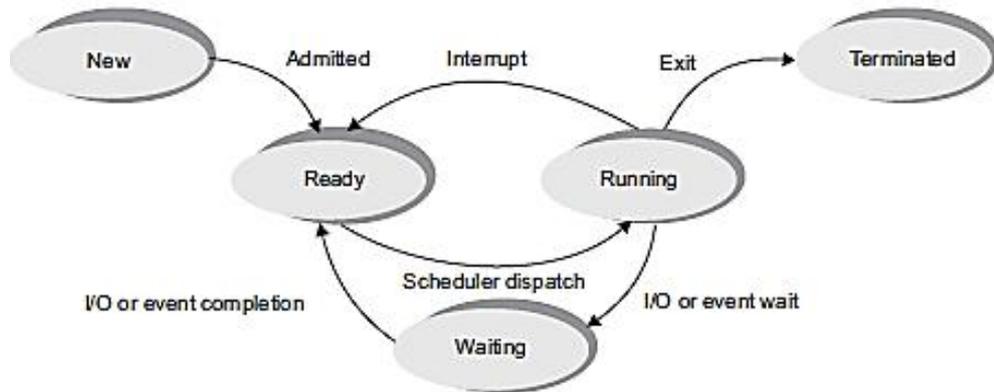


Figure: Process States

3.2 Processes Creation

The creation of a process requires the following steps. The order in which they are carried out is not necessarily the same in all cases.

- Name:** The name of the program which is to run as the new process must be known.
- Process ID and Process Control Block:** The system creates a new process control block, or locates an unused block in an array. This block is used to follow the execution of the program through its course, keeping track of its resources and priority. Each process control block is labeled by its PID or process identifier.
- Locate the program to be executed on disk and allocate memory for the code segment in RAM.
- Load the program into the code segment and initialize the registers of the PCB with the start address of the program and appropriate starting values for resources.
- Priority:** A priority must be computed for the process, using a default for the type of process and any value which the user specified as a 'nice' value.
- Schedule the process for execution.

Process Hierarchy: Children and Parent Processes

In a democratic system anyone can choose to start a new process, but it is never users which create processes but other processes! That is because anyone using the system must already be running a shell or command interpreter in order to be able to talk to the system, and the command interpreter is itself a process. When a user creates a process using the command interpreter, the new process becomes a child of the command interpreter. Similarly, the command interpreter process becomes the parent for the child. Processes therefore form a hierarchy.

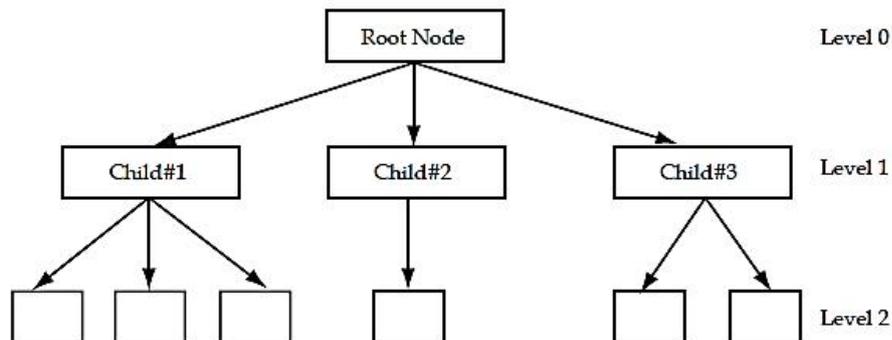


Figure: Process Hierarchies

The processes are linked by a tree structure. If a parent is signaled or killed, usually all its children receive the same signal or are destroyed with the parent. This doesn't have to be the case – it is possible to detach children from their parents – but in many cases it is useful for processes to be linked in this way.

When a child is created it may do one of two things.

- Duplicate the parent process.
- Load a completely new program.

Similarly, the parent may do one of two things.

- Continue executing along-side its children.
- Wait for some or all of its children to finish before proceeding.

The specific attributes of the child process that differ from the parent process are:

1. The child process has its own unique process ID.
2. The parent process ID of the child process is the process ID of its parent process.
3. The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes.
4. The elapsed processor times for the child process are set to zero.
5. The child doesn't inherit file locks set by the parent process.
6. The child doesn't inherit alarms set by the parent process.
7. The set of pending signals for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process.)

3.3 Process State Transitions

Blocking: It occurs when process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.

This state transition is: *Block:* Running? Block.

Time-Run-Out: It occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

This state transition is: *Time-Run-Out:* Running? Ready.

Dispatch: It occurs when all other processes have had their share and it is time for the first process to run again

This state transition is: *Dispatch:* Ready? Running.

Wakeup: It occurs when the external event for which a process was waiting (such as arrival of input) happens.

This state transition is: *Wakeup:* Blocked? Ready.

Admitted: It occurs when the process is created.

This state transition is: *Admitted:* New? Ready.

Exit: It occurs when the process has finished execution.

This state transition is: *Exit:* Running? Terminated.

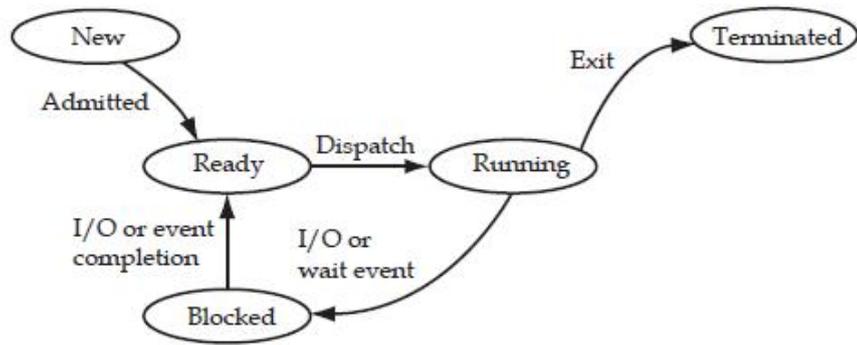


Figure (a):

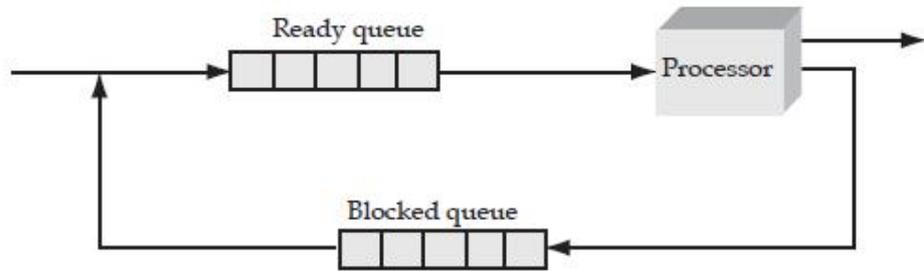


Figure (b): Process State Transitions

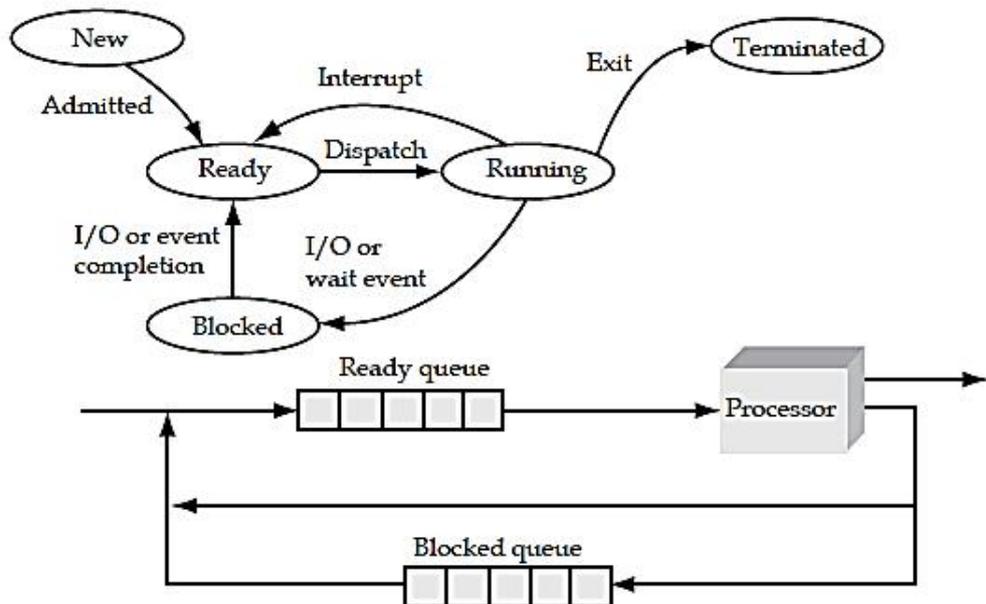


Figure (C): Process State Transitions

3.4 Process Termination

Processes terminate in one of two ways:

1. Normal Termination occurs by a return from main or when requested by an explicit call to exit.
2. Abnormal Termination occurs as the default action of a signal or when requested by abort.
3. On receiving a signal, a process looks for a signal-handling function. Failure to find a signal-handling function forces the process to call exit, and therefore to terminate.

4. A parent may terminate the execution of one of its children for a variety of reasons, such as these:
 - a) The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
 - b) The task assigned to the child is no longer required.
 - c) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

Cooperating Processes

The Concurrent processes executing in the operating system allows for the processes to cooperate (both mutually or destructively) with other processes. Processes are cooperating if they can affect each other. The simplest example of how this can happen is where two processes are using the same file. One process may be writing to a file, while another process is reading from the file; so, what is being read may be affected by what is being written. Processes cooperate by sharing data. Cooperation is important for several reasons:

- **Information Sharing**

Several processes may need to access the same data (such as stored in a file.)

- **Computation Speedup**

A task can often be run faster if it is broken into subtasks and distributed among different processes. For example, the matrix multiplication code you saw in class. This depends upon the processes sharing data. (Of course, real speedup also required having multiple CPUs that can be shared as well.) For another example, consider a web server which may be serving many clients. Each client can have their own process or thread helping them. This allows the server to use the operating system to distribute the computer's resources, including CPU time, among the many clients.

- **Modularity**

It may be easier to organize a complex task into separate subtasks, and then have different processes or threads running each subtask. *Example:* A single server process dedicated to a single client may have multiple threads running – each performing a different task for the client.

- **Convenience**

An individual user can run several programs at the same time, to perform some task. *Example:* A network browser is open, while the user has a remote terminal program running (such as telnet), and a word processing program editing data. Cooperation between processes requires mechanisms that allow processes to communicate data between each other and synchronize their actions so they do not harmfully interfere with each other. The purpose of this note is to consider ways that processes can communicate data with each other, called Inter-process Communication (IPC).

Note Another note will discuss process synchronization, and in particular, the most important means of synchronizing activity, the use of semaphores.

3.5 Inter-Process Communication

Inter-process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. It involves sending information from one process to another. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and Remote Procedure Calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated. Two processes might want to co-operate in performing a particular task. For example, a process might want to print a document in response to a user request, so it starts another process to handle the printing and sends a message to it to start printing. Once the process handling the printing request finishes, it sends a message back to the original process, which reads the message and uses this to pop up a dialog box informing the user that the document has been printed.

There are other ways in which processes can communicate with each other, such as using a shared memory space.

Table: Inter Process Communication

Method	Provided by (Operating systems or other environments)
File	All operating systems.
Signal	Most operating systems; some systems, such as Windows, only implement signals in the C run-time library and do not actually provide support for their use as an IPC technique.
Socket	Most operating systems.
Pipe	All POSIX systems.
Named pipe	All POSIX systems.
Semaphore	All POSIX systems.
Shared memory	All POSIX systems.
Message passing (shared nothing)	Used in MPI paradigm, Java RMI, CORBA and others.
Memory-mapped file	All POSIX systems; may carry race condition risk if a temporary file is used. Windows also supports this technique but the APIs used are platform specific.
Message queue	Most operating systems.
Mailbox	Some operating systems.

3.6 Process Communication in Client-Server Environment

Basically, the Client/Server environment is architected to split an application's processing across multiple processors to gain the maximum benefit at the least cost while minimizing the network traffic between machines. The key phase is to split the application processing. In a Client/Server mode each processor works independently but in cooperation with other processors. Each is relying on the other to perform an independent activity to complete the application process. A good example of this would be the Mid-Range computer, normally called a File Server, which is responsible for holding the customer master file while the Client, normally the Personal Computer, is responsible for requesting an update to a specific customer. Once the Client is authenticated, the File Server is notified that the Client needs Mr. Smith's record for an update.

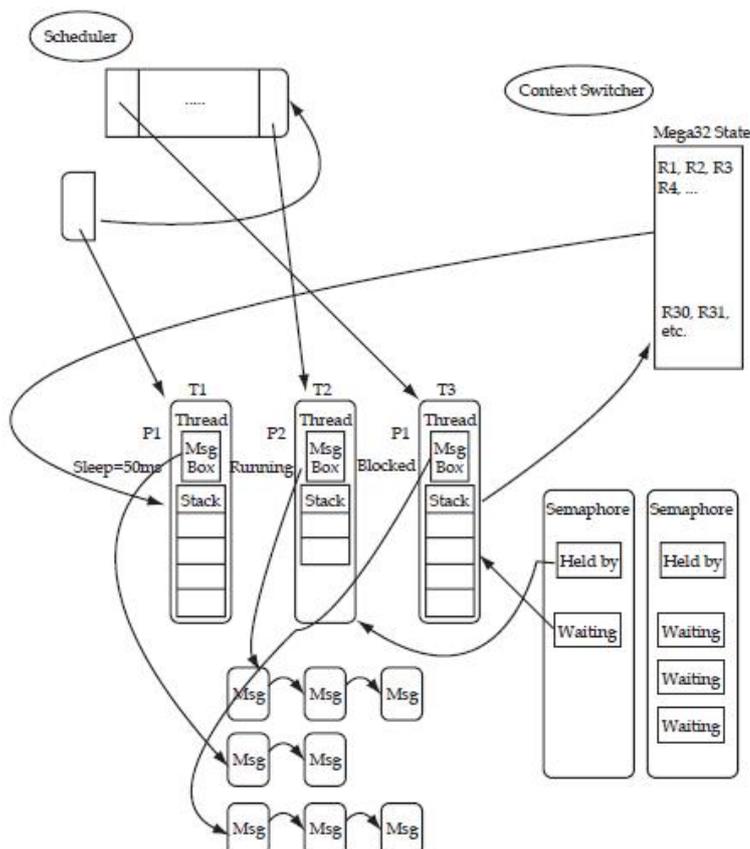
The File Server is responsible for obtaining Mr. Smith's record and passing it to the Client for the actual modification. The Client performs the changes and then passes the changed record back to the File Server which in turn updates the master file. As in this scenario, each processor has a distinct and independent responsibility to complete the update process. The key is to perform this cooperative task while minimizing the dialog or traffic between the machines over the network. Networks have a limited capacity to carry data and if overloaded the application's response time would increase. To accomplish this goal, static processes such as edits, and menus are usually designed to reside on the Client. Update and reporting processes usually are designed to reside on the File Server. In this way, the network traffic to complete the transaction process is minimized. In addition, this design minimizes the processing cost as the Personal Computer usually is the least expensive processor, the File Server being the next expensive, and finally the Main Frame the most expensive.

There are many Client/Server Models. First, one could install all of the application's object programs on the personal computer. Secondly, one could install the static object program routines such as edits and menus on the personal computer and the business logic object programs on the file server. Thirdly, one could install all the object programs on the file server. As another option, one could install all the object programs on the mainframe. Which model you choose depends on your application design.

3.7 Concept of Thread

Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respects, threads are popular way to improve application through parallelism.

The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack.



An operating system that has thread facility, the basic unit of CPU utilization is a thread. A thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one another like process as a result threads share with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Multitasking and multiprogramming, the two techniques that intend to use the computing resources optimally have been dealt with in the previous unit at length. In this unit you will learn about yet another technique that has caused remarkable improvement on the utilization of resources - thread. A thread is a finer abstraction of a process. Recall that a process is defined by the resources it uses and by the location at which it is executing in the memory. There are many instances, however, in which it would be useful for resources to be shared and accessed concurrently. This concept is so useful that several new operating systems are providing mechanism to support it through a thread facility.

Thread Structure

A thread, sometimes called a lightweight process (LWP), is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack. It shares with peer threads its code section, data section, and operating-system resources such as open files and signals, collectively known as a task. A traditional or heavyweight process is equal to a task with one thread. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and the creation of threads inexpensive, compared with context switches among heavyweight processes. Although a thread context switch still requires a register set switch, no memory-management-related work need be done. Like any parallel processing environment, multithreading a process may introduce concurrency control problems that require the use of critical sections or locks.

Also, some systems implement user-level threads in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system, and to cause an interrupt to the kernel. Switching between user-level threads can be done independently of the operating system and, therefore, very quickly. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of how a server can handle many requests efficiently. User-level threads do have disadvantages, however. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

You can grasp the functionality of threads by comparing multiple-thread control with multiple-process control. With multiple processes, each process operates independently of the others; each process has its own program counter, stack register, and address space. This type of organization is useful when the jobs performed by the processes are unrelated. Multiple processes can perform the same task as well. For instance, multiple processes can provide data to remote machines in a network file system implementation. However, it is more efficient to have one process containing multiple threads serve the same purpose. In the multiple process implementation, each process executes the same code but has its own memory and file resources. One multi-threaded process uses fewer resources than multiple redundant processes, including memory, open files and CPU scheduling, for example, as Solaris evolves, network daemons are being rewritten as kernel threads to increase greatly the performance of those network server functions. Threads operate, in many respects, in the same manner as processes. Threads can be in one of several states: ready, blocked, running, or terminated. A thread within a process executes sequentially, and each thread has its own stack and program counter. Threads can create child threads, and can block waiting for system calls to complete; if one thread is blocked, another can run. However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks. This structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with

multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.

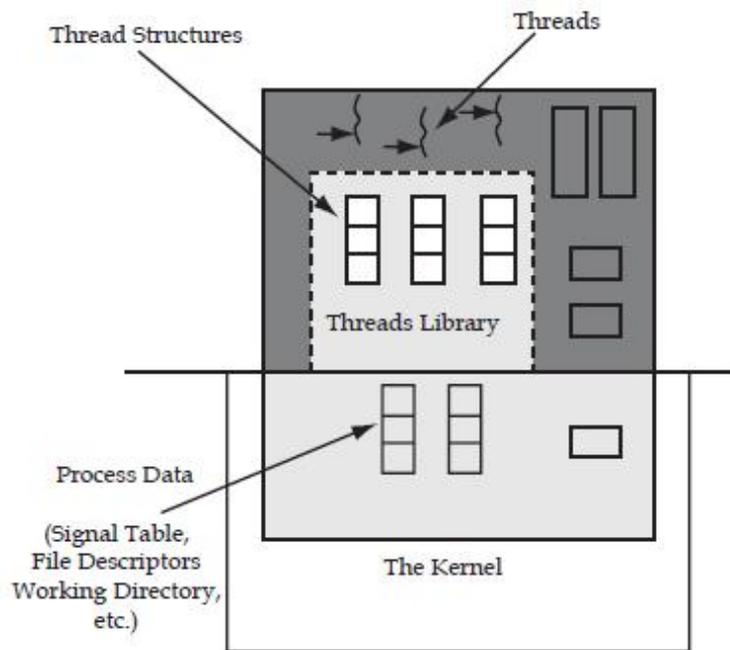


Figure: Thread Structure

Let us return to our example of the blocked file-server process in the single-process model. In this scenario, no other server process can execute until the first process is unblocked. By contrast, in the case of a task that contains multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run. In this application, the cooperation of multiple threads that are part of the same job confers the advantages of higher throughput and improved performance. Other applications, such as the producer-consumer problem, require sharing a common buffer and so also benefit from this feature of thread utilization. The producer and consumer could be threads in a task. Little overhead is needed to switch between them, and, on a multiprocessor system, they could execute in parallel on two processors for maximum efficiency.

User Level and Kernel Level Threads

The abstraction presented by a group of lightweight processes is that of multiple threads of control associated with several shared resources. There are many alternatives regarding threads. Threads can be supported by the kernel (as in the Mach and OS/2 operating systems). In this case, a set of system calls similar to those for processes is provided. Alternatively, they can be supported above the kernel, via a set of library calls at the user level (as is done in Project Andrew from CMU).

To implement parallel and concurrent mechanisms you need to use specific primitives of our operating system. These must have context switching capabilities, which can be implemented in two ways, using kernel level threads or using user level threads.

If I use kernel level threads, the operating system will have a descriptor for each thread belonging to a process and it will schedule all the threads. This method is commonly called one to one. Each user thread corresponds to a kernel thread.

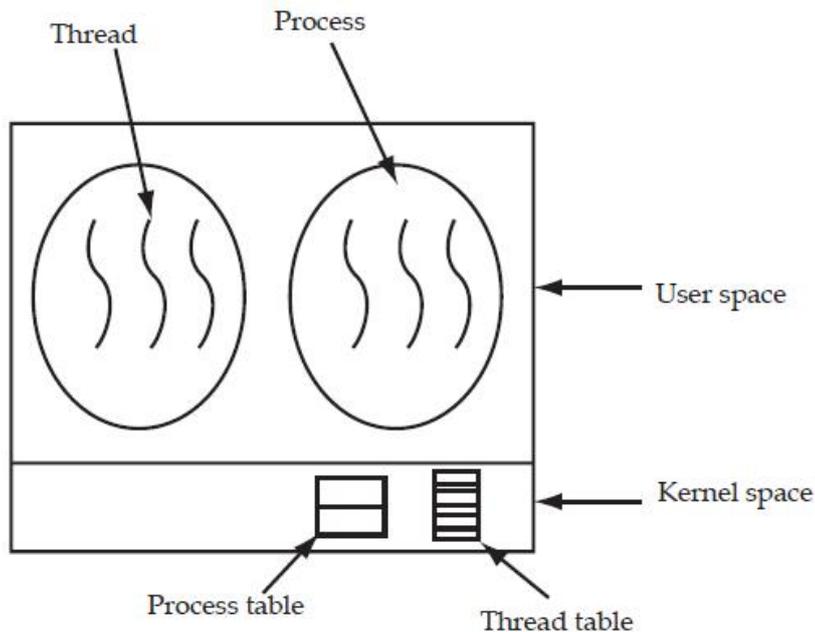


Figure: Diagram of Kernel Level Threads

There are two major advantages around this kind of thread. The first one concerns switching aspects; when a thread finishes its instruction or is blocked, another thread can be executed. The second one is the ability of the kernel to dispatch threads of one process on several processors. These characteristics are quite interesting for multi-processor architectures. However, thread switching is done by the kernel, which decreases performances.

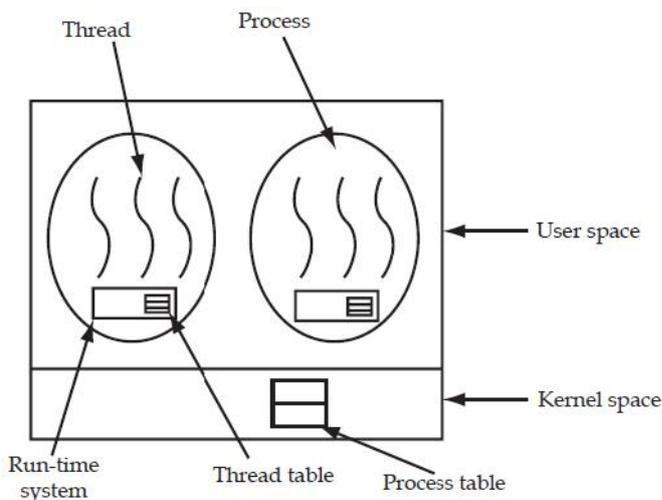


Figure: Diagram of User Threads

User level threads are implemented inside a specialized library that provides primitives to handle them. All information about threads is stored and managed inside the process address space. This is called many to one, because one kernel thread is associated to several user threads. It has some advantages: The first is that is independent of the system, thus, it runs faster than context switching at kernel level. The second comes from the scheduler that can be chosen by the user in order to manage a better thread execution. Nevertheless, if a thread of a process is jammed, all other threads of the same process are jammed too. Another disadvantage is the impossibility to execute two threads of the same process on two processors. So, user level thread is not interesting in multi-processor architectures.

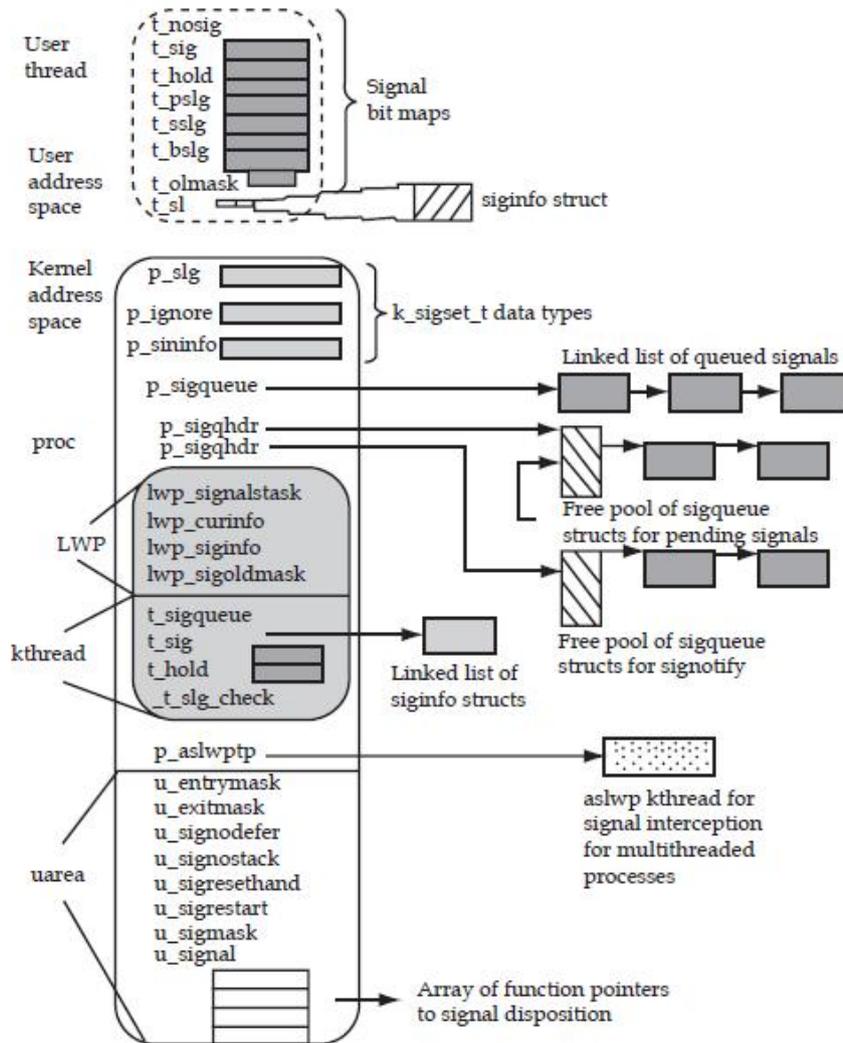


Figure: Detailed diagrammatic representation of User Level Threads

Why should an operating system support one version or the other? User-level threads do not involve the kernel, and therefore are faster to switch among than kernel-supported threads. However, any calls to the operating system can cause the entire process to wait, because the kernel schedules only processes (having no knowledge of threads), and a process which is waiting gets no CPU time. Scheduling can also be unfair. Consider two processes, one with 1 thread (process a) and the other with 100 threads (process b). Each process generally receives the same number of time slices, so the thread in process a runs 100 times as fast as a thread in process b. On systems with kernel-supported threads, switching among the threads is more time-consuming because the kernel (via an interrupt) must do the switch. Each thread may be scheduled independently, however, so process b could receive 100 times the CPU time that process a receives. Additionally, process b could have 100 system calls in operation concurrently, accomplishing far more than the same process would on a system with only user-level thread support.

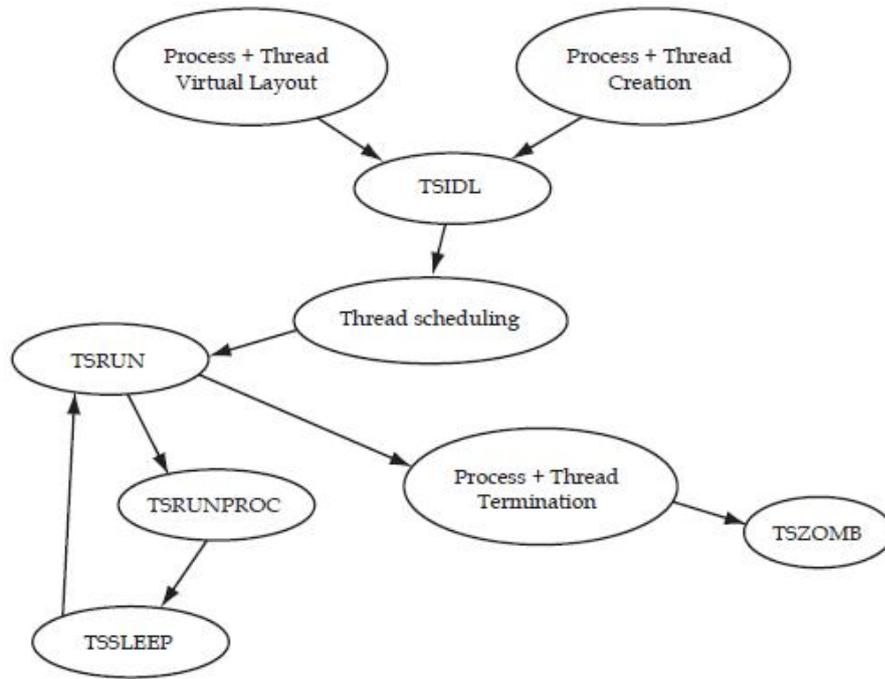


Figure: Detailed diagram of a kernel thread

Because of the compromises involved in each of these two approaches to threading, some systems use a hybrid approach in which both user-level and kernel-supported threads are implemented. Solaris 2 is such a system. A diagrammatic approach of hybrid thread is mentioned in Figure

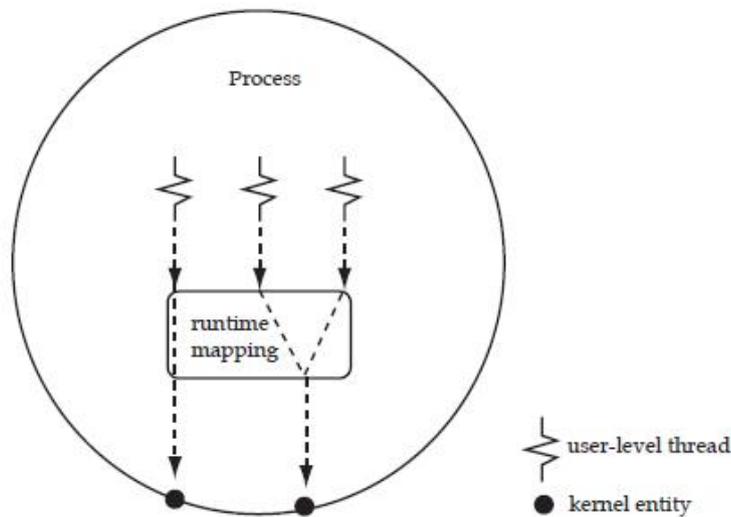


Figure: Hybrid Thread

3.8 Multi-threading

When the computers were first invented, they were capable of executing one program at a time. Thus, once one program was completely executed, they then picked the second one to execute and so on. With time, the concept of timesharing was developed whereby each program was given a specific amount of processor time and when its time got over the second program standing in queue was called upon (this is called Multi-tasking, and you would learn more about it soon).

Each running program (called the process) had its own memory space, its own stack, heap and its own set of variables. One process could spawn another process, but once that occurred the two behaved independent of each other. Then the next big thing happened. The programs wanted

to do more than one thing at the same time (this is called Multi-threading, and you would learn what it is soon). A browser, for example, might want to download one file in one window, while it is trying to upload another and print some other file. This ability of a program to do multiple things simultaneously is implemented through threads (detailed description on threads follows soon).

3.9 Multi-tasking vs. Multi-threading

Multi-tasking is the ability of an operating system to execute more than one program simultaneously. Though I say so but in reality, no two programs on a single processor machine can be executed at the same time. The CPU switches from one program to the next so quickly that appears as if all of the programs are executing at the same time. Multi-threading is the ability of an operating system to execute the different parts of the program, called threads, simultaneously. The program has to be designed well so that the different threads do not interfere with each other. This concept helps to create scalable applications because you can add threads as and when needed. Individual programs are all isolated from each other in terms of their memory and data, but individual threads are not as they all share the same memory and data variables. Hence, implementing multi-tasking is relatively easier in an operating system than implementing multithreading.

Thread Libraries

The threads library allows concurrent programming in Objective *Caml*. It provides multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space. Threads communicate by in-place modification of shared data structures, or by sending and receiving data on communication channels. The threads library is implemented by time-sharing on a single processor. It will not take advantage of multi-processor machines. Using this library will therefore never make programs run faster. However, many programs are easier to write when structured as several communicating processes.

Two implementations of the thread's library are available, depending on the capabilities of the operating system:

1. **System threads:** This implementation builds on the OS-provided threads facilities: POSIX 1003.1c threads for Unix, and Win32 threads for Windows. When available, system threadssupport both bytecode and native-code programs.
2. **VM-level threads:** This implementation performs time-sharing and context switching at thelevel of the OCaml virtual machine (bytecode interpreter). It is available on Unix systems,and supports only bytecode programs. It cannot be used with native-code programs.

Programs that use system threads must be linked as follows:

ocamlc -thread other options unix.cmathreads.cma other files

ocamlopt -thread other options unix.cmxathreads.cmxa other files

3.10 Threading Issues

The threading issues are:

1. System calls form and exec is discussed here. In a multithreaded program environment, form and exec system calls is changed. Unix system have two version of form system calls. One call duplicates all threads and another that duplicates only the thread that invoke the form system call whether to use one or two version of form system call totally depends upon the application. Duplicating all threads in unnecessary if exec is called immediately after form system call.

2. Thread cancellation is a process of thread terminate before its completion of task. *Example:* In multiple thread environment thread concurrently searching through a database. If any thread return the result, the remaining thread might be cancelled.

3. Thread cancellation is of two types:

- a) *Asynchronous cancellation*: One thread immediately terminates the target thread.
- b) *Deferred cancellation*: The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not.

3.11 Processes vs. Threads

As we mentioned earlier that in many respects' threads operate in the same way as that of processes. Let us point out some of the similarities and differences.

Similarities

1. Like processes threads share CPU and only one thread active (running) at a time.
2. Like processes, threads within a process, threads within a process execute sequentially.
3. Like processes, thread can create children.
4. And like process, if one thread is blocked, another thread can run.

Differences

1. Unlike processes, threads are not independent of one another.
2. Unlike processes, all threads can access every address in the task.
3. Processes might or might not assist one another because processes may originate from different users, but threads are design to assist one other.

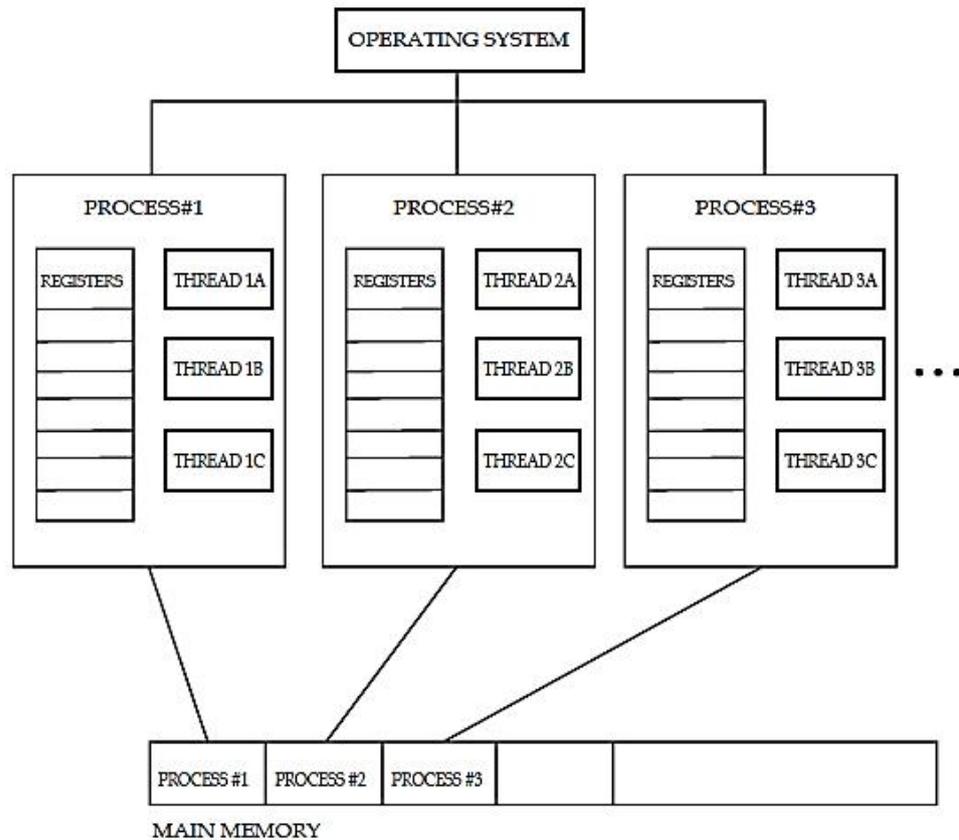


Figure: Diagram on Process with Multi-Thread

Benefits of Threads

Following are some reasons why threads are used in designing operating systems:

1. A process with multiple threads makes a great server for example printer server.
2. Because threads can share common data, they do not need to use inter-process communication.
3. Because of the very nature, threads can take advantage of multi-processors.
4. Threads need a stack and storage for registers therefore, threads are cheap to create.
5. Threads do not need new address space, global data, program code or operating system resources.

Summary

- Process management is an operating system's way of dealing with running multiple processes at once.
- A multi-tasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multi-threading or other similar technology).
- Processes are often called tasks in embedded operating systems. Process is the entity to which processors are assigned. The rapid switching back and forth of CPU among processes is called multi-programming.
- A thread is a single sequence stream within in a process. A process can have five states like created, ready, running, blocked and terminated.
- A process control block or PCB is a data structure (a table) that holds information about a process.
- Time-Run-Out occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- Dispatch occurs when all other processes have had their share and it is time for the first process to run again. Wakeup occurs when the external event for which a process was waiting (such as arrival of input) happens. Admitted occurs when the process is created.
- Exit occurs when the process has finished execution.

Keywords

Admitted: It is a process state transition which occurs when the process is created.

Blocking: It is a process state transition which occurs when process discovers that it cannot continue.

Dispatch: It is a process state transition which occurs when all other processes have had their share and it is time for the first process to run again.

Exit: It is a process state transition which occurs when the process has finished execution.

Multiprogramming: The rapid switching back and forth of CPU among processes is called multiprogramming.

Process control block (PCB): It is a data structure (a table) that holds information about a process.

Process management: It is an operating system's way of dealing with running multiple processes at once.

Process: It is the entity to which processors are assigned.

Thread: A thread is a single sequence stream within in a process.

Time-Run-Out: It is a process state transition which occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

Wakeup: It is a process state transition which occurs when the external event for which a process was waiting (such as arrival of input) happens.

Self Assessment

1. Interrupt driven processes will normally run at a very priority.
2. Processes are often called in embedded operating systems.
3. The term "process" was first used by the designers of the in
4. In new state, the process awaits admission to the state.
5. The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or
6. is a set of techniques for the exchange of data among two or more threads in one or more processes.
7. are a way for a program to fork itself into two or more simultaneously running tasks.
8. is the ability of an operating system to execute more than one programs simultaneously.
9. The threads library is implemented by time-sharing on a
10. A process includes PC, registers, and

11. A process does not include which of the following:
 - A. program counter
 - B. stack
 - C. data section
 - D. queue

12. Which of the following is true regarding programs and processes?
 - A. Program is passive entity stored on disk
 - B. Process is active.
 - C. Program becomes process when executable file loaded into memory
 - D. All of the above

13. Which of the following are not the states of a process?
 - A. New, running, waiting, ready and terminated
 - B. New, running, waiting and ready
 - C. New, running, waiting, terminating and terminated
 - D. New, executing, waiting, ready and terminated

14. Which of the following is not true regarding process creation?
 - A. Process creation means the construction of a new process for the execution.
 - B. Process creation might be performed by system, user or old process itself.
 - C. A process cannot create a new process itself while executing.
 - D. The system creates several background processes on starting the computer.

15. Which of the following statements is not true?
- A. Dispatching is done by operating system when the resources are free or the process has higher priority than the ongoing process.
 - B. Scheduling means the operating system puts the process from running state to the ready state.
 - C. Block mode is basically a mode where process waits for input-output.
 - D. All of the given choices
16. The state of a process after it encounters an I/O instruction is
- A. Ready
 - B. Blocked/Waiting
 - C. Idle
 - D. Running
17. Identify the incorrect statement with respect to the Blocking Process State
- A. When a process invokes an input-output system call, it blocks the process
 - B. When a process invokes an input-output system call, it puts the operating system in the block mode.
 - C. Block mode is basically a mode where process waits for input-output.
 - D. All of the Above
18. In operating system each process has its own
- A. Address space and global variables
 - B. Open files
 - C. Pending alarms signals and signal handlers
 - D. All of the mentioned
19. A process can be terminated due to
- A. Normal exit
 - B. Fatal error
 - C. Killed by another process
 - D. All of the mentioned
20. What is the ready state of a process?
- A. When process is scheduled to run after some execution
 - B. When process is unable to run until some task has been completed
 - C. When process is using the CPU
 - D. None of the mentioned
21. What is the inter process communication?
- A. communication between two processes
 - B. communication within the process
 - C. communication between two threads of same process
 - D. none of the mentioned

Operating System

22. A process stack does not contain.
- Function parameters
 - Local variables
 - Return addresses
 - PID of child process
23. A parent process may terminate the execution of one of its children for which of the following reasons?
- The child has exceeded its usage of some of the resources that it has been allocated.
 - The task assigned to the child is no longer required.
 - The parent is exiting, and the operating system does not allow a child to continue
 - All of these
24. In new state, the process awaits admission to the state.
- Running
 - Ready
 - Waiting
 - Blocked
25. The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or
- Process Management
 - Process State
 - Process Control Block
 - None of these

Answers for Self Assessment

- | | | | | |
|--------------------------------------|------------|--------------------|---------------------|---------------------------------|
| 1. high priority | 2. tasks | 3. MULTICS, 1960's | 4. ready | 5. Process Control Block (PCB). |
| 6. Inter-process Communication (IPC) | 7. Threads | 8. Multitasking | 9. single processor | 10. variables |
| 11. D | 12. D | 13. A | 14. C | 15. B |
| 16. B | 17. D | 18. D | 19. D | 20. A |
| 21. A | 22. D | 23. D | 24. B | 25. C |

Review Questions

- Do you think a single user system requires process communication? Support your answer with logic.

2. Suppose a user program faced an error during memory access. What will it do then? Will it be informed to the OS? Explain.
3. What resources are used when a thread created? How do they differ from those when a process is created?
4. What are the different process states? What is the state of the processor, when a process is waiting for some event to occur?
5. Write a brief description on process state transition.
6. What is PCB? What is the function of PCB?
7. How a process is created?
8. What is process hierarchy?
9. How a process terminated?
10. What is cooperating process? Explain it with example.
11. Why inter-process communication required?



Further Readings

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Published by Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., *Operating Systems*, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, *Operating System Concepts*, John Wiley & Sons, Seventh Edition.

Stalling, W., *Operating Systems*, 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 04: Process Management

CONTENTS

Objectives

Introduction

- 4.1 Process Concepts
- 4.2 Operation on Processes
- 4.3 Process Control Block (PCB)
- 4.4 Processes Creation
- 4.5 Process Hierarchy: Children and Parent Processes
- 4.6 Process State Transitions
- 4.7 Process Termination
- 4.8 Scheduling Queues

Self Assessment

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Explain process concepts
- Define PCB
- Describe operation on processes
- Explain inter-process communication
- Describe concept of thread

Introduction

Earlier a computer was used to be fasten the jobs pertaining to computation diligently and incessantly for a single person. Soon it was realized that the computer was far more powerful than just carrying out a single man's single job. Such was the speed of operation that the CPU would sit idle for most of the time awaiting user input. The CPU was certainly capable of carrying out many jobs simultaneously. It could also support many users simultaneously. But, the operating systems then available were not capable of this support. The operating systems facilitating a single-user support at a time was felt inadequate. Then a mechanism was developed which would prevent the wastage of CPU cycles. Hence multi-tasking systems were developed. In a multi-tasking system, a job or task is submitted as what is known as a process. Multi-tasking operating systems could handle multiple processes on a single processor. Process is a unit of program execution that enables the systems to implement multi-tasking behavior. Most of the operating systems today have multi-processing capabilities. This unit is dedicated to process and process related issues. In this unit, present and discuss the mechanisms that support or enforce more structured forms of inter-process communications. Subsequent sections are devoted to messages, an extremely versatile and popular

mechanism in both centralized and distributed systems, and to facilitate inter-process communication and synchronization.

4.1 Process Concepts

An operating system manages each hardware resource attached with the computer by representing it as an abstraction. An abstraction hides the unwanted details from the users and programmers allowing them to have a view of the resources in the form, which is convenient to them. A process is an abstract model of a sequential program in execution. The operating system can schedule a process as a unit of work. The term “process” was first used by the designers of the MULTICS in 1960’s. Since then, the term “process” is used somewhat interchangeably with ‘task’ or ‘job’. The process has been given many definitions as mentioned below:

1. A program in Execution.
2. An asynchronous activity.
3. The ‘animated spirit’ of a procedure in execution.
4. The entity to which processors are assigned.
5. The ‘dispatchable’ unit.

Though there is no universally agreed upon definition, but the definition “Program in Execution” is the one that is most frequently used. And this is a concept you will use in the present study of operating systems. Now the question is - what is the relation between process and program. It is same beast with different name or when this beast is sleeping (not executing) it is called program and when it is executing becomes process. Well, to be very precise.

Process is not the same as program. A process is more than a program code. A process is an ‘active’ entity as oppose to program which is considered to be a ‘passive’ entity. As you all know that a program is an algorithm expressed with the help of a programming language. A program is a passive entity sitting on some secondary storage device.

A process is an activity of executing a program. Basically, it is a program under execution. Every process needs certain resources to complete its task.

Process, on the other hand, includes:

1. Current value of Program Counter (PC)
2. Contents of the processor’s registers
3. Value of the variables
4. The process-stack (SP) which typically contains temporary data such as subroutine parameter, return address, and temporary variables.
5. A data section that contains global variables.
6. A process is the unit of work in a system.

In Process model, all software on the computer is organized into a number of sequential processes. A process includes PC, registers, and variables. Conceptually, each process has its own virtual CPU. In reality, the CPU switches back and forth among processes. (The rapid switching back and forth is called multi-programming).

A process includes, besides instructions to be executed, the temporary data such as subroutine parameters, return addresses and variables (stored on the stack), data section having global variables (if any), program counter value, register values and other associated resources. Although two processes may be associated with the same program, yet they are treated as two separate processes having their respective set of resources.

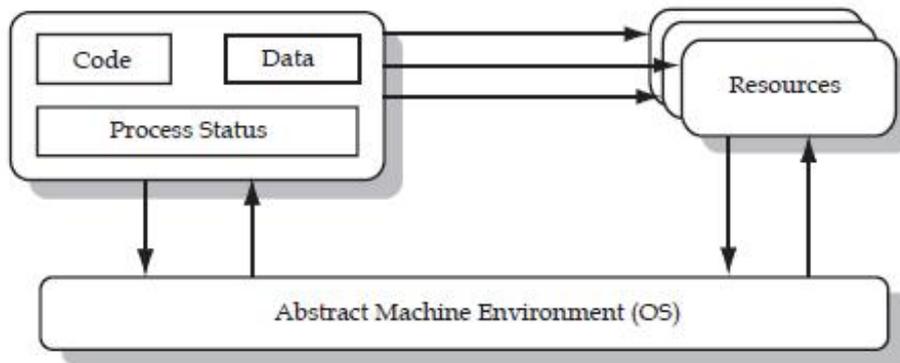


Figure: A Schematic Representation of a Process

4.2 Operation on Processes

Modern operating systems, such as UNIX, execute processes concurrently. Although there is a single Central Processor (CPU), which execute the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing a single process a few hundred microseconds of CPU time before replacing it with another process.)

Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system. The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU: the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPU's registers, and other data), and removes the process from the use of the CPU. The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this. The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination.

4.3 Process Control Block (PCB)

It is a data structure (a table) that holds information about a process. Whenever a process is created (initialized, installed), the operating system creates a corresponding process control block to serve as its run-time description during the lifetime of the process. When the process terminates, its PCB is released to the pool of free cells from which new PCBs are drawn.

Process States

During the lifespan of a process, its execution status may be in one of four states (associated with each state is usually a queue on which the process resides):

- **Executing:** The process is currently running and has control of a CPU.
- **Waiting:** The process is currently able to run, but must wait until a CPU becomes available.
- **Blocked:** The process is currently waiting on I/O, either for input to arrive or output to be sent.
- **Suspended:** The process is currently able to run, but for some reason the OS has not placed the process on the ready queue.
- **Ready:** The process is in memory, will execute given CPU time.

- **Terminated:** The process has finished execution. These state names are arbitrary, and they vary across operating systems. The states that they represent are found on all systems, however. Certain operating systems more finely delineate process states.

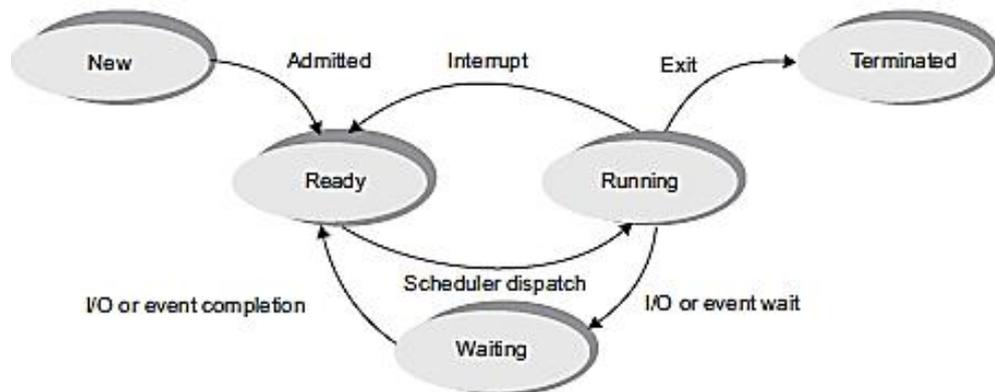


Figure: Process States

4.4 Processes Creation

The creation of a process requires the following steps. The order in which they are carried out is not necessarily the same in all cases.

1. **Name:** The name of the program which is to run as the new process must be known.
2. **Process ID and Process Control Block:** The system creates a new process control block, or locates an unused block in an array. This block is used to follow the execution of the program through its course, keeping track of its resources and priority. Each process control block is labeled by its PID or process identifier.
3. Locate the program to be executed on disk and allocate memory for the code segment in RAM.
4. Load the program into the code segment and initialize the registers of the PCB with the start address of the program and appropriate starting values for resources.
5. **Priority:** A priority must be computed for the process, using a default for the type of process and any value which the user specified as a 'nice' value.
6. Schedule the process for execution.

4.5 Process Hierarchy: Children and Parent Processes

In a democratic system anyone can choose to start a new process, but it is never users which create processes but other processes! That is because anyone using the system must already be running a shell or command interpreter in order to be able to talk to the system, and the command interpreter is itself a process. When a user creates a process using the command interpreter, the new process becomes a child of the command interpreter. Similarly, the command interpreter process becomes the parent for the child. Processes therefore form a hierarchy.

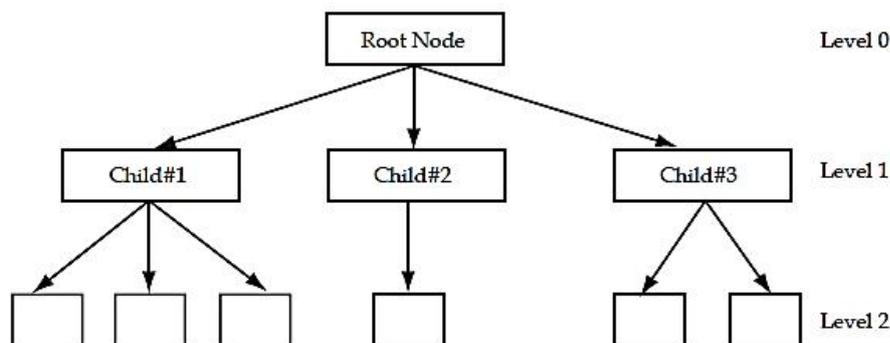


Figure: Process Hierarchies

The processes are linked by a tree structure. If a parent is signaled or killed, usually all its children receive the same signal or are destroyed with the parent. This doesn't have to be the case – it is possible to detach children from their parents – but in many cases it is useful for processes to be linked in this way.

When a child is created it may do one of two things.

- Duplicate the parent process.
- Load a completely new program.

Similarly, the parent may do one of two things.

- Continue executing along-side its children.
- Wait for some or all of its children to finish before proceeding.

The specific attributes of the child process that differ from the parent process are:

1. The child process has its own unique process ID.
2. The parent process ID of the child process is the process ID of its parent process.
3. The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, the file position associated with each descriptor is shared by both processes.
4. The elapsed processor times for the child process are set to zero.
5. The child doesn't inherit file locks set by the parent process.
6. The child doesn't inherit alarms set by the parent process.
7. The set of pending signals for the child process is cleared. (The child process inherits its mask of blocked signals and signal actions from the parent process.)

4.6 Process State Transitions

Blocking: It occurs when process discovers that it cannot continue. If running process initiates an I/O operation before its allotted time expires, the running process voluntarily relinquishes the CPU.

This state transition is: *Block:* Running? Block.

Time-Run-Out: It occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.

This state transition is: *Time-Run-Out:* Running? Ready.

Dispatch: It occurs when all other processes have had their share and it is time for the first process to run again

This state transition is: *Dispatch:* Ready? Running.

Wakeup: It occurs when the external event for which a process was waiting (such as arrival of input) happens.

This state transition is: *Wakeup:* Blocked? Ready.

Admitted: It occurs when the process is created.

This state transition is: *Admitted:* New? Ready.

Exit: It occurs when the process has finished execution.

This state transition is: *Exit:* Running? Terminated.

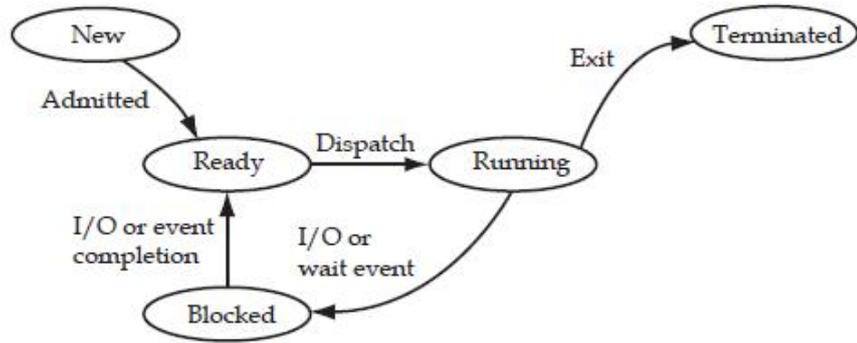


Figure (a):

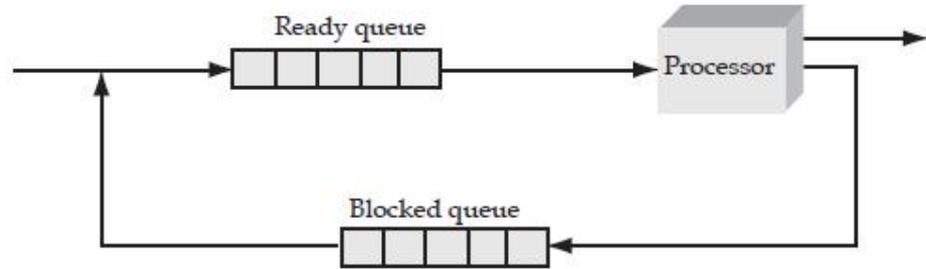


Figure (b): Process State Transitions

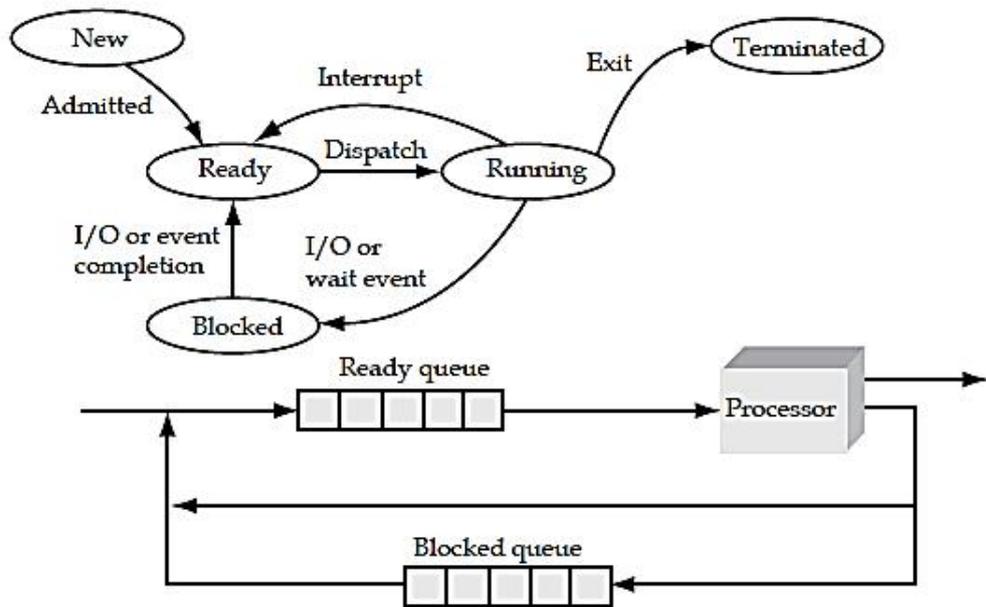


Figure (C): Process State Transitions

4.7 Process Termination

Processes terminate in one of two ways:

1. Normal Termination occurs by a return from main or when requested by an explicit call to exit.
2. Abnormal Termination occurs as the default action of a signal or when requested by abort.
3. On receiving a signal, a process looks for a signal-handling function. Failure to find a signal-handling function forces the process to call exit, and therefore to terminate.

4. A parent may terminate the execution of one of its children for a variety of reasons, such as these:

- a) The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- b) The task assigned to the child is no longer required.
- c) The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as cascading termination, is normally initiated by the operating system.

PCB (Process Control Blocks)

The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or a Process Control Block (PCB). Whenever a process is created (initialized, installed), the operating system creates a corresponding process control block to serve as its run-time description during the lifetime of the process. When the process terminates, its PCB is released to the pool of free cells from which new PCBs are drawn. The dormant state is distinguished from other states because a dormant process has no PCB. A process becomes known to the O.S. and thus eligible to compete for system resources only when it has an active PCB associate with it.

Information stored in a PCB typically includes some or all of the following:

1. Process name (ID)
2. Priority
3. State (ready, running, suspended)
4. Hardware state.
5. Scheduling information and usage statistics
6. Memory management information (registers, tables)
7. I/O Status (allocated devices, pending operations)
8. File management information
9. Accounting information.

Once constructed for a newly created process, the PCB is filled with the programmer defined attributes found in the process template or specified as the parameters of the CREATE-PROCESS operating system call. Whenever a process is suspended, the contents of the processor registers are usually saved on the stack, and the pointer to the related stack frame is stored in the PCB. In this way, the hardware state can be restored when the process is scheduled to run again. A process control block or PCB is a data structure (a table) that holds information about a process.

Every process or program that runs needs a PCB. When a user requests to run a particular program, the operating system constructs a process control block for that program.

Typical information that is stored in a process control block is:

- The location the process in memory
- The priority of the process
- A unique process identification number (called PID)
- The current process state (ready, running, blocked)
- Associated data for the process.

The PCB is a certain store that allows the operating systems to locate key information about a process. Thus, the PCB is the data structure that defines a process to the operating systems.

Process state
Process number
Parent process number
Program counter
Register
Memory limits
List of open files

Figure 2: Process Control Block

Each process is represented in the operating system by a process control block (PCB), also called a task control block. A PCB is shown in Figure 2.2. It contains many pieces of information associated with a specific process, including these:

Process state: The state may be new, ready, running, waiting, halted, and so on.

Program counter: The counter indicates the address of the next instruction to be executed for this process.

CPU registers: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure.3).

CPU-scheduling information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. **Memory-management information:** This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

Memory-management information: This information may include such information as the value of the base and limit registers, the page tables, or the segment tables, depending on the memory system used by the operating system.

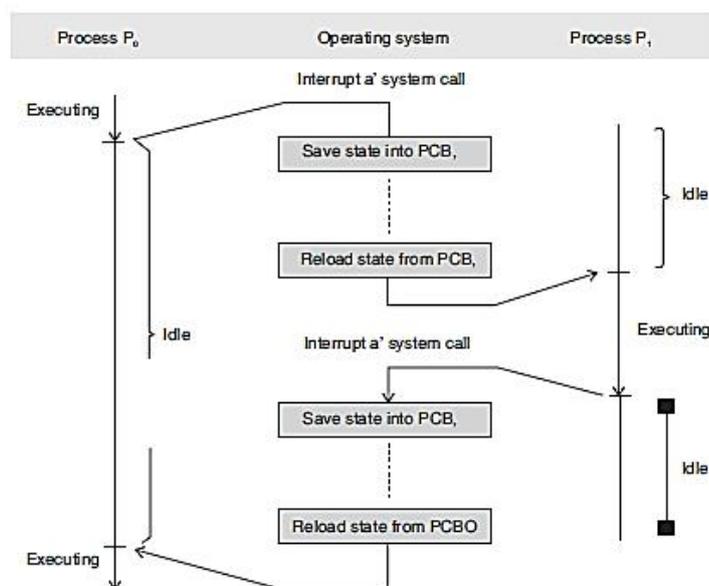


Figure 3: CPU Switching from process to process

Accounting information: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

Status information: The information includes the list of I/O devices allocated to this process, a list of open files, and so on. The PCB simply serves as the repository for any information that may vary from process to process.



Did you know?

Threads: The process model discussed so far has implied that a process is a program that performs a single thread of execution. For example, if a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time.

Process Scheduling

The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization. The objective of time-sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. A uniprocessor system can have only one running process.

If more processes exist, the rest must wait until the CPU is free and can be rescheduled.

4.8 Scheduling Queues

As processes enter the system, they are put into a job queue. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. We extend each PCB to include a pointer field that points to the next PCB in the ready queue.

The operating system also has other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of a I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since the system has many processes, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue (Figure 4).

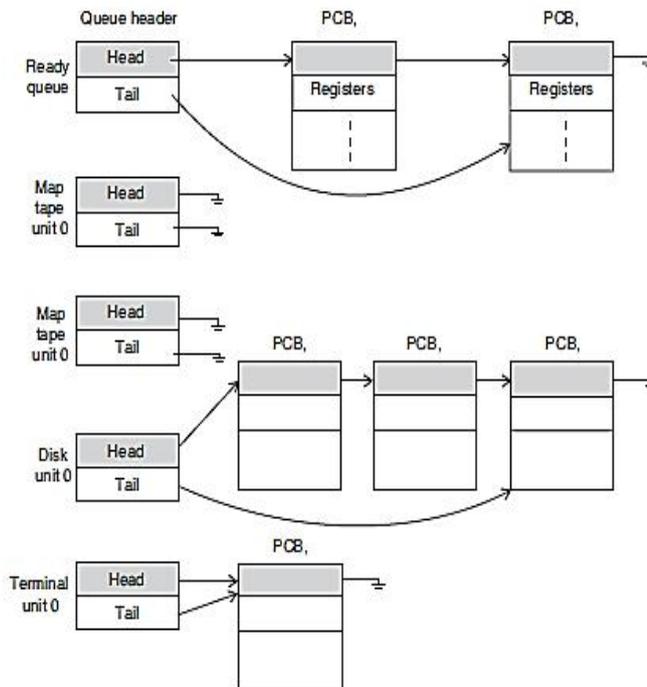


Figure: The Ready queue as well as the various I/O Device Queues

A common representation of process scheduling is a queueing diagram, such as that in Figure 5. Each rectangular box represents a queue. Two types of queues are present – the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

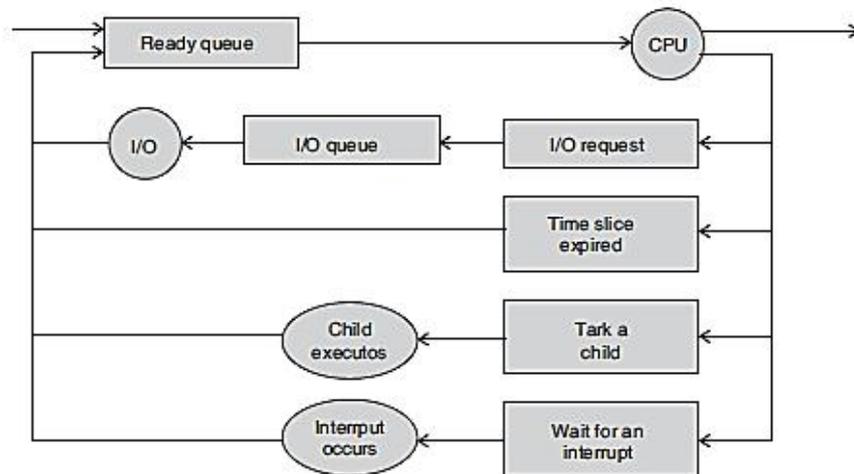


Figure 5: Queuing Diagram Representation of Process Scheduling

A new process is initially put in the ready queue. It waits in the ready queue until it is selected for execution (or dispatched). Once the process is assigned to the CPU and is executing, one of several events could occur:

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

In the first two cases, the process eventually switches from the waiting state to the ready state, and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Schedulers

A process migrates between the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate scheduler.

In a batch system, often more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution. The long-term scheduler, or job scheduler, selects processes from this pool and loads them into memory for execution. The short-term scheduler, or CPU scheduler, selects from among the processes that are ready to execute, and allocates the CPU to one of them. The primary distinction between these two schedulers is the frequency of their execution. The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short-term scheduler executes at least once every 100 milliseconds. Because of the brief time between executions, the short-term scheduler must be fast. If it takes 10 milliseconds to decide to execute a process for 100 milliseconds, then $10/(100 + 10) = 9$ percent of the CPU is being used (or wasted) simply for scheduling the work.

The long-term scheduler, on the other hand, executes much less frequently. There may be minutes between the creation of new processes in the system. The long-term scheduler controls the degree of multiprogramming – the number of processes in memory. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of

processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to select a process for execution.

The long-term scheduler must make a careful selection. In general, most processes can be described as either I/O bound or CPU bound. A I/O-bound process spends more of its time doing I/O than it spends doing computations. A **CPU-bound** process, on the other hand, generates I/O requests infrequently, using more of its time doing computation than an I/O bound process uses. The long-term scheduler should select a good process mix of I/O-bound and CPU-bound processes. If all processes are I/O bound, the ready queue will almost always be empty, and the short-term scheduler will have little to do. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, timesharing systems such as UNIX often have no long-term scheduler, but simply put every new process in memory for the short-term scheduler. The stability of these systems depends either on a physical limitation (such as the number of available terminals) or on the self-adjusting nature of human users. If the performance declines to unacceptable levels, some users will simply quit. Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This medium-term scheduler, diagrammed in Figure 6, removes processes from memory (and from active contention for the CPU), and thus reduces the degree of multiprogramming. At some later time, the process can be reintroduced into memory and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix, or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up.

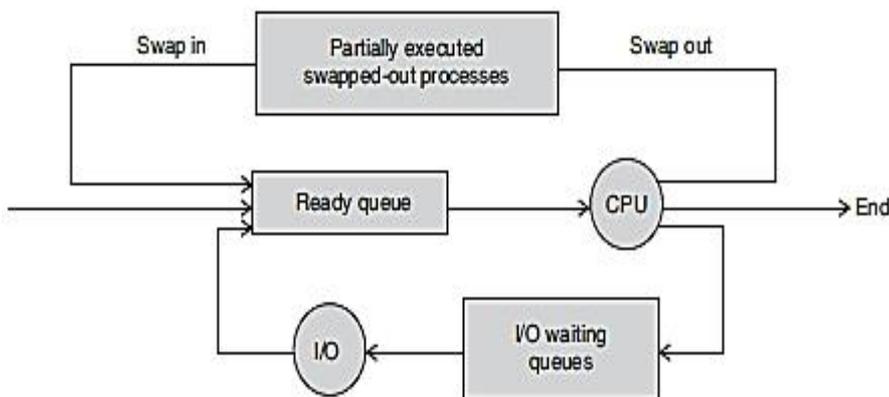


Figure: Addition of Medium-term Scheduling to the Queuing Diagram

Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information. When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run. Context-switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instructions (such as a single instruction to load or store all registers). Typical speeds range from 1 to 1000 microseconds. Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers. A context switch simply includes changing the pointer to the current register set. Of course, if active processes exceed register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the more work must be done during a context switch. Advanced memory-management techniques may require extra data to be switched with each context. For instance, the address space of the current process must be preserved as the space of the next task is prepared for

use. How the address space is preserved, and what amount of work is needed to preserve it, depend on the memory-management method of the operating system..

Summary

- A process is a sequential program in execution. A process migrates between the various scheduling queues throughout its lifetime.
- The operating system must select, for scheduling purposes, processes from these queues in some fashion.
- The selection process is carried out by the appropriate scheduler.
- Switching the CPU to another process requires saving the state of the old process and loading the saved state for the new process. This task is known as a context switch. The context of a process is represented in the PCB of a process; it includes the value of the CPU registers, the process state and memory-management information.
- A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.

Keywords

Buffering: A buffer is a temporary storage location for data while the data is being transferred.

Context Switch: A context switch (also sometimes referred to as a process switch or a task switch) is the switching of the CPU (central processing unit) from one process or thread to another.

Cooperating Processes: Processes can cooperate with each other to accomplish a single task.

Cooperating processes can:

- Improve performance by overlapping activities or performing work in parallel.
- Enable an application to achieve a better program structure as a set of cooperating processes, where each is smaller than a single monolithic program.

CPU Registers: The central processing unit (CPU) contains a number of memory locations which are individually addressable and reserved for specific purpose. These memory locations are called registers.

'Inter-process Communication' (IPC): In computing, 'Inter-process communication' (IPC) is a set of techniques for the exchange of data among multiple threads in one or more processes.

Message-Passing System: Message passing in computer science is a form of communication used in parallel computing, object-oriented programming, and inter-process communication.

Process Control Block (PCB): The PCB is a certain store that allows the operating systems to locate key information about a process.

Process Counter: Program instructions uniquely identified by their program counters (PCs) provide a convenient and accurate means of recording the context of program execution and PC-based prediction techniques have been widely used for performance optimizations at the architectural level.

Process Management: The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server, etc. Each of these activities is encapsulated in a process.

Process Scheduling: The problem of determining when processors should be assigned and to which processes is called processor scheduling or CPU scheduling.

Process State: The process state consists of everything necessary to resume the process execution if it is somehow put aside temporarily.

Synchronization: In computer science, especially parallel computing, synchronization means the coordination of simultaneous threads or processes to complete a task in order to get correct runtime order and avoid unexpected race conditions.

Thread: A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams.

Self Assessment

1. Any process that shares data with other processes is a process.
2. Message sent by a process can be of either fixed or size.
3. A is associated with more than two processes.
4. A owned by the operating system is independent.
5. The Mach kernel supports the creation and destruction of multiple task.
 - A. True
 - B. False
6. Window 2000 uses two types of message passing techniques over a port.
 - A. True
 - B. False
7. The Information associated with each process in a Process Control Block does not include which of the following?
 - A. Process State
 - B. CPU Scheduling Information
 - C. Accounting Information
 - D. Program-management information
8. Which of the following is not a type of queue maintained in the Process Scheduling Queues?
 - A. Job Queue
 - B. Ready Queue
 - C. Waiting Queue
 - D. Device Queue
9. In a tightly coupled system or parallel systems, the processor shares _____?
 - A. Memory and clock
 - B. All of the choices
 - C. Only memory is shared, not the clock
 - D. Only clock is shared, memory is not shared
10. In operating system each process has its own

- A. Address space and global variables
 - B. Open files
 - C. Pending alarms signals and signal handlers
 - D. All of the mentioned
11. To access the services of operating system, the interface is provided by which of the following?
- A. System calls
 - B. (b)API
 - C. (c) library
 - D. (d) assembly instructions
12. The medium-term scheduler can be added if_____?
- A. The degree of multiple programming needs to increase
 - B. Process is removed from memory and brought back to the disk
 - C. The degree of multiple programming needs to decrease
 - D. All of the above
13. The iOS provides_____?
- A. Multiple background processes running in the memory with limits
 - B. Single foreground process controlled via user interface
 - C. Multiple background processes running in the memory but not on the display
 - D. All of the given choices
14. Which of the following is not true regarding Android?
- A. Background process uses a service to perform tasks
 - B. Service can keep running even if background process is suspended
 - C. Service has no user interface
 - D. It requires larger memory use due to the user interface.
15. Which of the following is not true regarding context-switching?
- A. Context of a process is represented in the Process Control Block
 - B. Context-switch time is an overhead for the system
 - C. Context switching time is independent of the hardware support
 - D. None of the choices
16. What is operating system?
- A. collection of programs that manages hardware resources
 - B. system service provider to the application programs
 - C. link to interface the hardware and application programs
 - D. all of the mentioned

Answers for Self Assessment

1. Cooperating 2. Variable 3. Link 4. mailbox 5. A
6. B 7. D 8. C 9. B 10. D
11. A 12. C 13. D 14. D 15. C
16. D

Review Questions

1. What is a process?
2. What about process states?
3. What is a process control block?
4. How do processes inter-communicate?
5. How do processes synchronize their activity?
6. How do processes protect critical data (Critical sections)?
7. Consider the inter-process-communication scheme where mailboxes are used:
 - a) Suppose a process P wants to wait for two messages, one from mailbox A and one from mailbox B. What sequence of **send** and **receive** should it execute?
 - b) What sequence of **send** and **receive** should P execute if P wants to wait for one message from mailbox A or from mailbox B (or from both)?
8. What are the benefits and the detriments of each of the following? Consider both the systems and the programmers' levels.
 - a) Direct and indirect communication
 - b) Symmetric and asymmetric communication
 - c) Automatic and explicit buffering
 - d) Send by copy and send by reference **Notes**
 - e) Fixed-sized and variable-sized messages
9. Describe the actions taken by a kernel to switch context between processes.
10. Write a socket-based Fortune Teller server. Your program should create a server that listens to a specified port. When a client receives a connection, the server should respond with a random fortune chosen from its database of fortunes.
11. Describe the actions used in Buffering in the processes.
12. Describe about the process scheduling in the operating system.
13. How do processes inter-process communication?
14. What are the benefits and the detriments of Cooperating process.
15. Describe the Process States in operating system.

**Further Readings**

Operating Systems, by Harvey M. Deitel , Paul J. Deitel, David R. Choffnes.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull



Web Links

wiley.com/coolege.silberschatz

Unit 05: Inter-Process Communication

CONTENTS

Objectives

- 5.1 Cooperating Processes
- 5.2 Inter-Process Communication
- 5.3 Process Communication in Client-Server Environment
- 5.4 Concept of Thread
- 5.5 User Level and Kernel Level Threads
- 5.6 Multi-Threading
- 5.7 Thread Libraries
- 5.8 Threading Issues
- 5.9 Processes vs. Threads
- 5.10 Benefits of Threads

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the Inter-process communication
- understand the concept of cooperating processes
- learn the various communication models
- learn the producer consumer problem
- learn the various types of inter-process communications
- learn the multithreading models
- explore the various threading issues

Introduction

Earlier a computer was used to be fasten the jobs pertaining to computation diligently and incessantly for a single person. Soon it was realized that the computer was far more powerful than just carrying out a single man's single job. Such was the speed of operation that the CPU would sit idle for most of the time awaiting user input. The CPU was certainly capable of carrying out many jobs simultaneously. It could also support many users simultaneously. But, the operating systems then available were not capable of this support. The operating systems facilitating a single-user support at a time was felt inadequate. Then a mechanism was developed which would prevent the wastage of CPU cycles. Hence multi-tasking systems were developed. In a multi-tasking system, a job or task is submitted as what is known as a process. Multi-tasking operating systems could handle multiple processes on a single processor. Process is a unit of program execution that enables the systems to implement multi-tasking behavior. Most of the operating systems today have multi-processing capabilities. This unit is dedicated to process and process related issues. In this unit,

present and discuss the mechanisms that support or enforce more structured forms of inter-process communications. Subsequent sections are devoted to messages, an extremely versatile and popular mechanism in both centralized and distributed systems, and to facilitate inter-process communication and synchronization.

5.1 Cooperating Processes

The Concurrent processes executing in the operating system allows for the processes to cooperate (both mutually or destructively) with other processes. Processes are cooperating if they can affect each other. The simplest example of how this can happen is where two processes are using the same file. One process may be writing to a file, while another process is reading from the file; so, what is being read may be affected by what is being written. Processes cooperate by sharing data. Cooperation is important for several reasons:

- **Information Sharing**

Several processes may need to access the same data (such as stored in a file).

- **Computation Speedup**

A task can often be run faster if it is broken into subtasks and distributed among different processes. For example, the matrix multiplication code you saw in class. This depends upon the processes sharing data. (Of course, real speedup also required having multiple CPUs that can be shared as well.) For another example, consider a web server which may be serving many clients. Each client can have their own process or thread helping them. This allows the server to use the operating system to distribute the computer's resources, including CPU time, among the many clients.

- **Modularity**

It may be easier to organize a complex task into separate subtasks, and then have different processes or threads running each subtask. *Example:* A single server process dedicated to a single client may have multiple threads running – each performing a different task for the client.

- **Convenience**

An individual user can run several programs at the same time, to perform some task. *Example:* A network browser is open, while the user has a remote terminal program running (such as telnet), and a word processing program editing data. Cooperation between processes requires mechanisms that allow processes to communicate data between each other and synchronize their actions so they do not harmfully interfere with each other. The purpose of this note is to consider ways that processes can communicate data with each other, called Inter-process Communication (IPC).



Note: Another note will discuss process synchronization, and in particular, the most important means of synchronizing activity, the use of semaphores.

5.2 Inter-Process Communication

Inter-process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes. It involves sending information from one process to another. Processes may be running on one or more computers connected by a network. IPC techniques are divided into methods for message passing, synchronization, shared memory, and Remote Procedure Calls (RPC). The method of IPC used may vary based on the bandwidth and latency of communication between the threads, and the type of data being communicated. Two processes might want to co-operate in performing a particular task. For example, a process might want to print to document in response to a user request, so it starts another process to handle the printing and sends a message to it to start printing. Once the process handling the printing request finishes, it sends a message back to the original process, which reads the message and uses this to pop up a dialog box informing the user that the document has been printed.

There are other ways in which processes can communicate with each other, such as using a shared memory space.

Table 5.1: Inter Process Communication

Method	Provided by (Operating systems or other environments)
File	All operating systems.
Signal	Most operating systems; some systems, such as Windows, only implement signals in the C run-time library and do not actually provide support for their use as an IPC technique.
Socket	Most operating systems.
Pipe	All POSIX systems.
Named pipe	All POSIX systems.
Semaphore	All POSIX systems.
Shared memory	All POSIX systems.
Message passing (shared nothing)	Used in MPI paradigm, Java RMI, CORBA and others.
Memory-mapped file	All POSIX systems; may carry race condition risk if a temporary file is used. Windows also supports this technique but the APIs used are platform specific.
Message queue	Most operating systems.
Mailbox	Some operating systems.

5.3 Process Communication in Client-Server Environment

Basically, the Client/Server environment is architected to split an application's processing across multiple processor to gain the maximum benefit at the least cost while minimizing the network traffic between machines. The key phase is to split the application processing. In a Client/Server mode each processing works independently but in cooperation with other processors. Each is relying on the other to perform an independent activity to complete the application process. A good example of this would be the Mid-Range computer, normally called a File Server, which is responsible for holding the customer master file while the Client, normally the Personal Computer, is responsible for requesting an update to a specific customer. Once the Client is authenticated, the File Server is notified that the Client needs Mr. Smith's record for an update.

The File Server is responsible for obtaining Mr. Smith's record and passing it to the Client for the actual modification. The Client performs the changes and then passes the changed record back to the File Server which in turn updates the master file. As in this scenario, each processor has a distinct and independent responsibility to complete the update process. The key is to perform this cooperative task while minimizing the dialog or traffic between the machines over the network. Networks have a limited capacity to carry data and if overloaded the application's response time would increase. To accomplish this goal, static processes such as edits, and menus are usually designed to reside on the Client. Update and reporting processes usually are designed to reside on the File Server. In this way, the network traffic to complete the transaction process is minimized. In addition, this design minimizes the processing cost as the Personal Computer usually is the least expensive processor, the File Server being the next expensive, and finally the Main Frame the most expensive.

There are many Client/Server Models. First, one could install all of the application's object programs on the personal computer. Secondly, one could install the static object program routines such as edits and menus on the personal computer and the business logic object programs on the file server. Thirdly, one could install all the object programs on the file server. As another option,

one could install all the object programs on the mainframe. Which model you choose depends on your application design.

5.4 Concept of Thread

Threads are a way for a program to fork (or split) itself into two or more simultaneously (or pseudo-simultaneously) running tasks. A thread is a single sequence stream within in a process. Because threads have some of the properties of processes, they are sometimes called lightweight processes. In a process, threads allow multiple executions of streams. In many respects, threads are popular way to improve application through parallelism.

The CPU switches rapidly back and forth among the threads giving illusion that the threads are running in parallel. Like a traditional process i.e., process with one thread, a thread can be in any of several states (Running, Blocked, Ready or Terminated). Each thread has its own stack. Since thread will generally call different procedures and thus a different execution history. This is why thread needs its own stack.

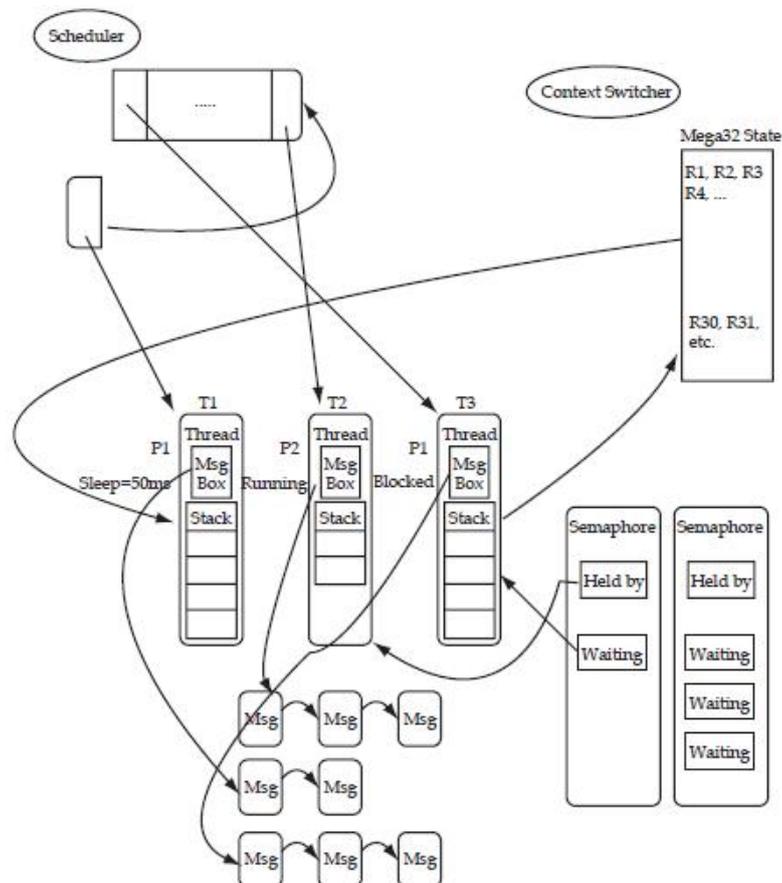


Figure 5.1: Threading Process

An operating system that has thread facility, the basic unit of CPU utilization is a thread. As shown in Figure 5.1, a thread has or consists of a program counter (PC), a register set, and a stack space. Threads are not independent of one other like process as a result threads shares with other threads their code section, data section, OS resources also known as task, such as open files and signals.

Multitasking and multiprogramming, the two techniques that intend to use the computing resources optimally have been dealt with in the previous unit at length. In this unit you will learn about yet another technique that has caused remarkable improvement on the utilization of resources - thread. A thread is a finer abstraction of a process. Recall that a process is defined by the resources it uses and by the location at which it is executing in the memory. There are many instances, however, in which it would be useful for resources to be shared and accessed concurrently. This concept is so useful that several new operating systems are providing mechanism to support it through a thread facility.

Thread Structure

A thread, sometimes called a lightweight process (LWP), is a basic unit of resource utilization, and consists of a program counter, a register set, and a stack. It shares with peer threads its code section, data section, and operating-system resources such as open files and signals, collectively known as a task. A traditional or heavyweight process is equal to a task with one thread. A task does nothing if no threads are in it, and a thread must be in exactly one task. The extensive sharing makes CPU switching among peer threads and the creation of threads inexpensive, compared with context switches among heavyweight processes. Although a thread context switch still requires a register set switch, no memory-management-related work need be done. Like any parallel processing environment, multithreading a process may introduce concurrency control problems that require the use of critical sections or locks.

Also, some systems implement user-level threads in user-level libraries, rather than via system calls, so thread switching does not need to call the operating system, and to cause an interrupt to the kernel. Switching between user-level threads can be done independently of the operating system and, therefore, very quickly. Thus, blocking a thread and switching to another thread is a reasonable solution to the problem of how a server can handle many requests efficiently. User-level threads do have disadvantages, however. For instance, if the kernel is single-threaded, then any user-level thread executing a system call will cause the entire task to wait until the system call returns.

You can grasp the functionality of threads by comparing multiple-thread control with multiple-process control. With multiple processes, each process operates independently of the others; each process has its own program counter, stack register, and address space. This type of organization is useful when the jobs performed by the processes are unrelated. Multiple processes can perform the same task as well. For instance, multiple processes can provide data to remote machines in a network file system implementation. However, it is more efficient to have one process containing multiple threads serve the same purpose. In the multiple process implementation, each process executes the same code but has its own memory and file resources. One multi-threaded process uses fewer resources than multiple redundant processes, including memory, open files and CPU scheduling, for example, as Solaris evolves, network daemons are being rewritten as kernel threads to increase greatly the performance of those network server functions. Threads operate, in many respects, in the same manner as processes. Threads can be in one of several states: ready, blocked, running, or terminated. As shown in figure 5.2, a thread within a process executes sequentially, and each thread has its own stack and program counter. Threads can create child threads, and can block waiting for system calls to complete; if one thread is blocked, another can run. However, unlike processes, threads are not independent of one another. Because all threads can access every address in the task, a thread can read or write over any other thread's stacks. This structure does not provide protection between threads. Such protection, however, should not be necessary. Whereas processes may originate from different users, and may be hostile to one another, only a single user can own an individual task with multiple threads. The threads, in this case, probably would be designed to assist one another, and therefore would not require mutual protection.

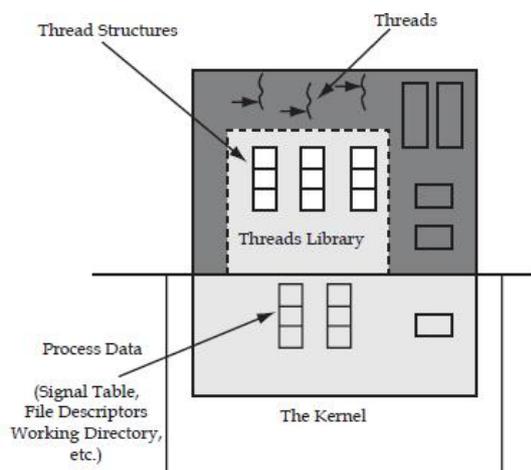


Figure 5.2: Thread Structure

Let us return to our example of the blocked file-server process in the single-process model. In this scenario, no other server process can execute until the first process is unblocked. By contrast, in the

case of a task that contains multiple threads, while one server thread is blocked and waiting, a second thread in the same task could run. In this application, the cooperation of multiple threads that are part of the same job confers the advantages of higher throughput and improved performance. Other applications, such as the producer-consumer problem, require sharing a common buffer and so also benefit from this feature of thread utilization. The producer and consumer could be threads in a task. Little overhead is needed to switch between them, and, on a multiprocessor system, they could execute in parallel on two processors for maximum efficiency.

5.5 User Level and Kernel Level Threads

The abstraction presented by a group of lightweight processes is that of multiple threads of control associated with several shared resources. There are many alternatives regarding threads. Threads can be supported by the kernel (as in the Mach and OS/2 operating systems). In this case, a set of system calls similar to those for processes is provided. Alternatively, they can be supported above the kernel, via a set of library calls at the user level (as is done in Project Andrew from CMU).

To implement parallel and concurrent mechanisms you need to use specific primitives of our operating system. These must have context switching capabilities, which can be implemented in two ways, using kernel level threads or using user level threads.

If I use kernel level threads, the operating system will have a descriptor for each thread belonging to a process and it will schedule all the threads. This method is commonly called one to one. Each user thread corresponds to a kernel thread.

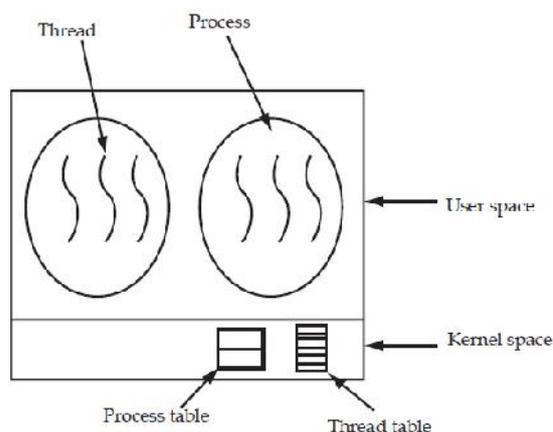


Figure 5.3: Diagram of Kernel Level Threads

There are two major advantages around this kind of thread. The first one concerns switching aspects; when a thread finishes its instruction or is blocked, another thread can be executed. The second one is the ability of the kernel to dispatch threads of one process on several processors. These characteristics are quite interesting for multi-processor architectures. However, thread switching is done by the kernel, which decreases performances.

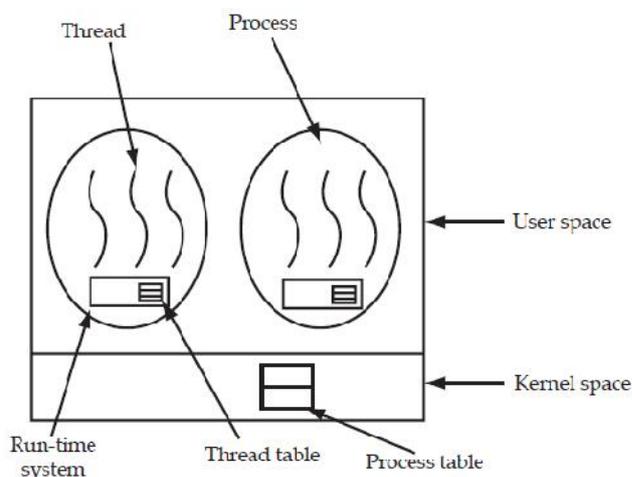


Figure 5.4: Diagram of User Threads

User level threads are implemented inside a specialized library that provides primitives to handle them. All information about threads is stored and managed inside the process address space. Refer to figure 5.4. This is called many to one, because one kernel thread is associated to several user threads. It has some advantages: The first is that is independent of the system, thus, it runs faster than context switching at kernel level. The second comes from the scheduler that can be chosen by the user in order to manage a better thread execution. Nevertheless, if a thread of a process is jammed, all other threads of the same process are jammed too. Another disadvantage is the impossibility to execute two threads of the same process on two processors. So, user level thread is not interesting in multi-processor architectures as in figure 5.5.

Why should an operating system support one version or the other? User-level threads do not involve the kernel, and therefore are faster to switch among than kernel-supported threads. However, any calls to the operating system can cause the entire process to wait, because the kernel schedules only processes (having no knowledge of threads), and a process which is waiting gets no CPU time.

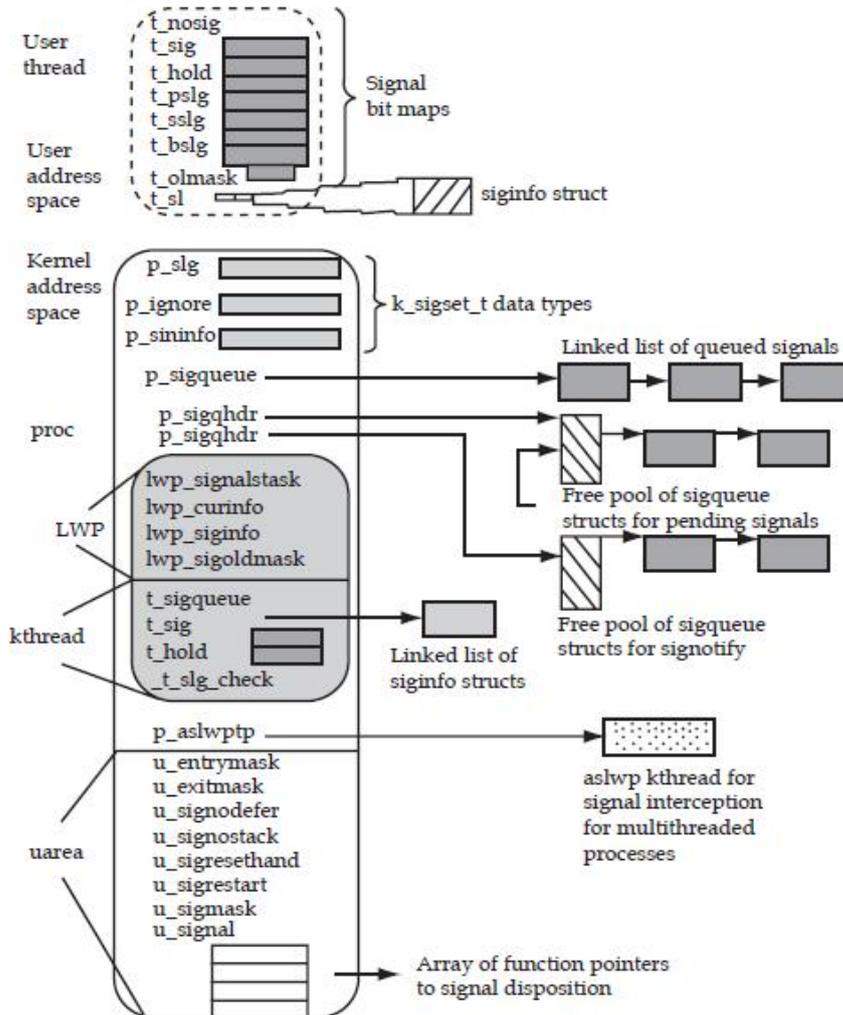


Figure 5.5: Detailed diagrammatic representation of User Level Threads

Scheduling can also be unfair. Consider two processes, one with 1 thread (process a) and the other with 100 threads (process b). Each process generally receives the same number of time slices, so the thread in process a runs 100 times as fast as a thread in process b. On systems with kernel-supported threads, switching among the threads is more time-consuming because the kernel (via an interrupt) must do the switch. Each thread may be scheduled independently, however, so process b could receive 100 times the CPU time that process it receives. Additionally, process b could have 100 system calls in operation concurrently, accomplishing far more than the same process would on a system with only user-level thread support.

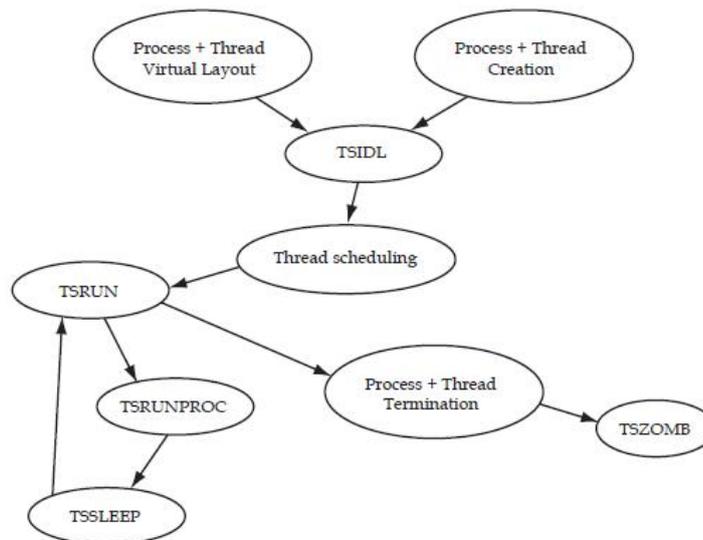


Figure 5.6: Detailed diagram of a kernel thread

Because of the compromises involved in each of these two approaches to threading, some systems use a hybrid approach in which both user-level and kernel-supported threads are implemented. Solaris 2 is such a system. A diagrammatic approach of hybrid thread is mentioned in Figure

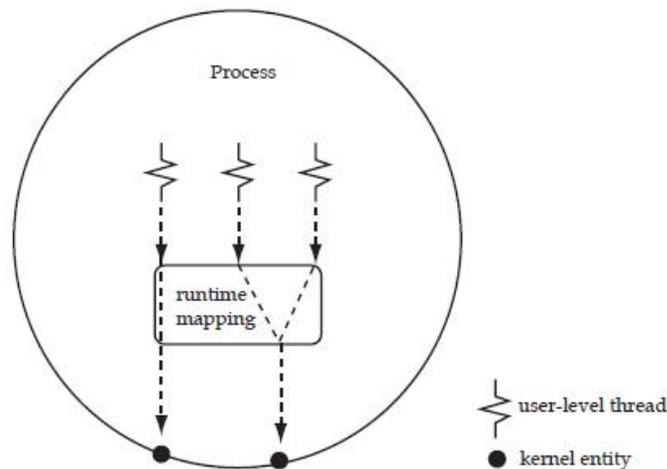


Figure 5.7: Hybrid Thread

5.6 Multi-Threading

When the computers were first invented, they were capable of executing one program at a time. Thus, once one program was completely executed, they then picked the second one to execute and so on. With time, the concept of timesharing was developed whereby each program was given a specific amount of processor time and when its time got over the second program standing in queue was called upon (this is called Multi-tasking, and you would learn more about it soon).

Each running program (called the process) had its own memory space, its own stack, heap and its own set of variables. One process could spawn another process, but once that occurred the two behaved independent of each other. Then the next big thing happened. The programs wanted to do more than one thing at the same time (this is called Multi-threading, and you would learn what it is soon). A browser, for example, might want to download one file in one window, while it is trying to upload another and print some other file. This ability of a program to do multiple things simultaneously is implemented through threads (detailed description on threads follows soon).

Multi-Tasking vs. Multi-Threading

Multi-tasking is the ability of an operating system to execute more than one program simultaneously. Though I say so but in reality, no two programs on a single processor machine can be executed at the same time. The CPU switches from one program to the next so quickly that

appears as if all of the programs are executing at the same time. Multi-threading is the ability of an operating system to execute the different parts of the program, called threads, simultaneously. The program has to be designed well so that the different threads do not interfere with each other. This concept helps to create scalable applications because you can add threads as and when needed. Individual programs are all isolated from each other in terms of their memory and data, but individual threads are not as they all share the same memory and data variables. Hence, implementing multi-tasking is relatively easier in an operating system than implementing multithreading.

5.7 Thread Libraries

The threads library allows concurrent programming in Objective *Caml*. It provides multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space. Threads communicate by in-place modification of shared data structures, or by sending and receiving data on communication channels. The threads library is implemented by time-sharing on a single processor. It will not take advantage of multi-processor machines. Using this library will therefore never make programs run faster. However, many programs are easier to write when structured as several communicating processes.

Two implementations of the thread's library are available, depending on the capabilities of the operating system:

1. **System threads:** This implementation builds on the OS-provided threads facilities: POSIX 1003.1c threads for Unix, and Win32 threads for Windows. When available, system threads support both bytecode and native-code programs.
2. **VM-level threads:** This implementation performs time-sharing and context switching at the level of the OCaml virtual machine (bytecode interpreter). It is available on Unix systems, and supports only bytecode programs. It cannot be used with native-code programs.

Programs that use system threads must be linked as follows:

ocamlc -thread other options unix.cmathreads.cma other files

ocamlopt -thread other options unix.cmxathreads.cmxa other files

5.8 Threading Issues

The threading issues are:

1. System calls `fork` and `exec` is discussed here. In a multithreaded program environment, `fork` and `exec` system calls is changed. Unix system have two version of `fork` system calls. One call duplicates all threads and another that duplicates only the thread that invoke the `fork` system call whether to use one or two version of `fork` system call totally depends upon the application. Duplicating all threads in unnecessary if `exec` is called immediately after `fork` system call.
2. Thread cancellation is a process of thread terminate before its completion of task. *Example:* In multiple thread environment thread concurrently searching through a database. If any thread return the result, the remaining thread might be cancelled.
3. Thread cancellation is of two types:
 - a) *Asynchronous cancellation:* One thread immediately terminates the target thread.
 - b) *Deferred cancellation:* The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

With deferred cancellation, one thread indicates that a target thread is to be cancelled, but cancellation occurs only after the target thread has checked a flag to determine if it should be cancelled or not.

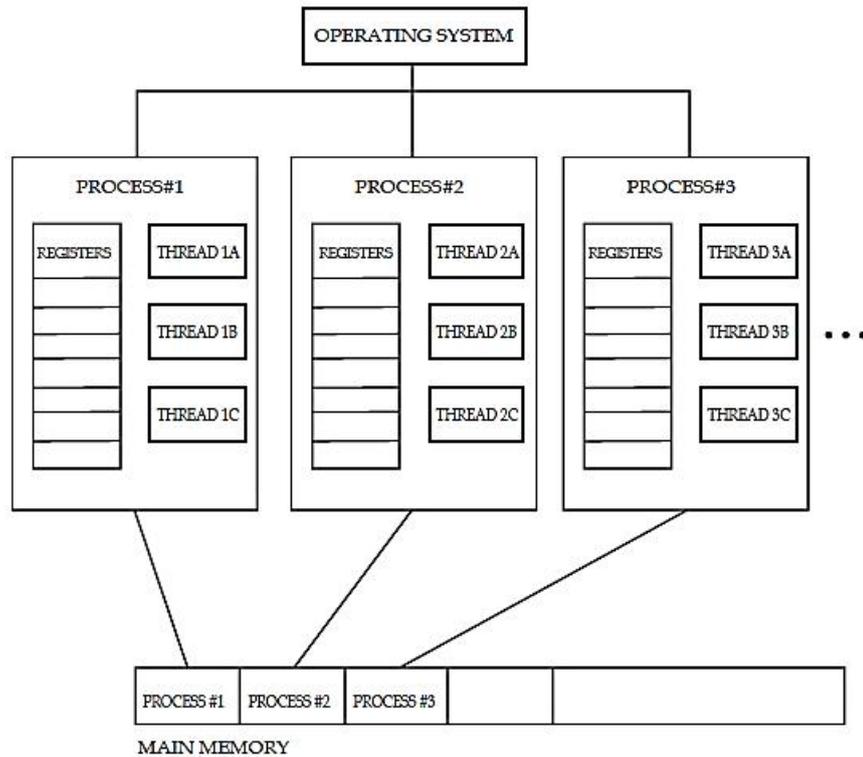


Figure 5.8: Diagram on Process with Multi-Thread

5.9 Processes vs. Threads

As we mentioned earlier that in many respects' threads operate in the same way as that of processes. Let us point out some of the similarities and differences.

Similarities

1. Like processes threads share CPU and only one thread active (running) at a time.
2. Like processes, threads within a process, threads within a process execute sequentially.
3. Like processes, thread can create children.
4. And like process, if one thread is blocked, another thread can run.

Differences

1. Unlike processes, threads are not independent of one another.
2. Unlike processes, all threads can access every address in the task.
3. Processes might or might not assist one another because processes may originate from different users, but threads are design to assist one other.

5.10 Benefits of Threads

Following are some reasons why threads are used in designing operating systems:

1. A process with multiple threads makes a great server for example printer server.
2. Because threads can share common data, they do not need to use inter-process communication.
3. Because of the very nature, threads can take advantage of multi-processors.
4. Threads need a stack and storage for registers therefore, threads are cheap to create.
5. Threads do not need new address space, global data, program code or operating system resources.

Summary

- Process management is an operating system's way of dealing with running multiple processes at once.
- A multi-tasking operating system may just switch between processes to give the appearance of many processes executing concurrently or simultaneously, though in fact only one process can be executing at any one time on a single-core CPU (unless using multi-threading or other similar technology).
- Processes are often called tasks in embedded operating systems. Process is the entity to which processors are assigned. The rapid switching back and forth of CPU among processes is called multi-programming.
- A thread is a single sequence stream within in a process. A process can have five states like created, ready, running, blocked and terminated.
- A process control block or PCB is a data structure (a table) that holds information about a process.
- Time-Run-Out occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- Dispatch occurs when all other processes have had their share and it is time for the first process to run again. Wakeup occurs when the external event for which a process was waiting (such as arrival of input) happens. Admitted occurs when the process is created.
- Exit occurs when the process has finished execution.

Keywords

- **Admitted:** It is a process state transition which occurs when the process is created.
- **Blocking:** It is a process state transition which occurs when process discovers that it cannot continue.
- **Dispatch:** It is a process state transition which occurs when all other processes have had their share and it is time for the first process to run again.
- **Exit:** It is a process state transition which occurs when the process has finished execution.
- **Multiprogramming:** The rapid switching back and forth of CPU among processes is called multiprogramming.
- **Process control block (PCB):** It is a data structure (a table) that holds information about a process.
- **Process management:** It is an operating system's way of dealing with running multiple processes at once.
- **Process:** It is the entity to which processors are assigned.
- **Thread:** A thread is a single sequence stream within in a process.
- **Time-Run-Out:** It is a process state transition which occurs when the scheduler decides that the running process has run long enough and it is time to let another process have CPU time.
- **Wakeup:** It is a process state transition which occurs when the external event for which a process was waiting (such as arrival of input) happens.

SelfAssessment

1. Interrupt driven processes will normally run at a very priority.
2. Processes are often called in embedded operating systems.

3. The term "process" was first used by the designers of the in
4. In new state, the process awaits admission to the state.
5. The operating system groups all information that it needs about a particular process into a data structure called a *process descriptor* or
6. is a set of techniques for the exchange of data among two or more threads in one or more processes.
7. are a way for a program to fork itself into two or more simultaneously running tasks.
8. is the ability of an operating system to execute more than one program simultaneously.
9. The threads library is implemented by time-sharing on a
10. A process includes PC, registers, and

11. A process does not include which of the following:
 - A. program counter
 - B. stack
 - C. data section
 - D. queue

12. Which of the following is not related to the physical implementation of communication link?
 - A. Shared memory
 - B. Automatic or explicit buffering
 - C. Hardware bus
 - D. Network

13. Which of the following is a shortcut for sending an INT signal using the keyboard?
 - A. Ctrl-Z
 - B. Ctrl-C
 - C. Ctrl-K
 - D. Ctrl-I

14. A single threaded process is the one in which?
 - A. One command is processed at a time
 - B. Multiple parts of the program are allowed to executed at the same time
 - C. There are lightweight processes available within the process
 - D. None of the given choices

15. Which of the following is not an advantage of the Cooperating Processes?

- A. Information Privacy
- B. Communication speed-up
- C. Modularity
- D. Convenience

Answers for Self Assessment

1. high priority 2. tasks 3. MULTICS, 1960's 4. ready 5. Process Control Block (PCB)
6. Inter-process Communication (IPC) 7. Threads 8. Multitasking 9. single processor 10. variables
11. D 12. B 13. B 14. A 15. A

Review Questions

1. Do you think a single user system requires process communication? Support your answer with logic.
2. Suppose a user program faced an error during memory access. What will it do then? Will it be informed to the OS? Explain.
3. What resources are used when a thread created? How do they differ from those when a process is created?
4. What are the different process states? What is the state of the processor, when a process is waiting for some event to occur?
5. Write a brief description on process state transition.
6. What is PCB? What is the function of PCB?
7. How a process is created?
8. What is process hierarchy?
9. How a process terminated?
10. What is cooperating process? Explain it with example. Also explain why inter-process communication is required?



Further Readings

- Andrew M. Lister, Fundamentals of Operating Systems, Wiley.
- Andrew S. Tanenbaum and Albert S. Woodhull, Systems Design and Implementation, Published by Prentice Hall.
- Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.
- Colin Ritchie, Operating Systems, BPB Publications.
- Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.
- I.A. Dhotre, Operating System, Technical Publications.
- Milankovic, Operating System, Tata MacGraw Hill, New Delhi.
- Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh

Edition.

Stalling, W., "*Operating Systems*", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 06: CPU Scheduling

CONTENTS

Objectives

Introduction

6.1 CPU Scheduling

6.2 CPU Scheduling Basic Criteria

6.3 Scheduling Algorithms

6.4 Operating Systems and Scheduling Types

6.5 Types of Scheduling

6.6 Multiple Processor Scheduling

6.7 Thread Scheduling

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Describe CPU scheduling
- Explain CPU scheduling basic criteria
- Know scheduling algorithms
- Describe types of scheduling
- Explain multiple processor scheduling
- Define thread scheduling

Introduction

CPU scheduling is the basics of multiprogramming. By switching the CPU among several processes, the operating systems can make the computer more productive. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.

6.1 CPU Scheduling

The objective of multiprogramming is to have some process running at all times to maximize CPU utilization. The objective of time-sharing system is to switch the CPU among processes so frequently that users can interact with each program while it is running. For a uni-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled. As processes enter the system, they are put into a job queue. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list as shown in figure 6.1. A ready-queue header will contain pointers to the first and last PCBs in the list. Each PCB has a pointer field that points to the next process in

the ready queue. There are also other queues in the system. When a process is allocated the CPU, it executes for awhile and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. In the case of an I/O request, such a request may be to a dedicated tape drive, or to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

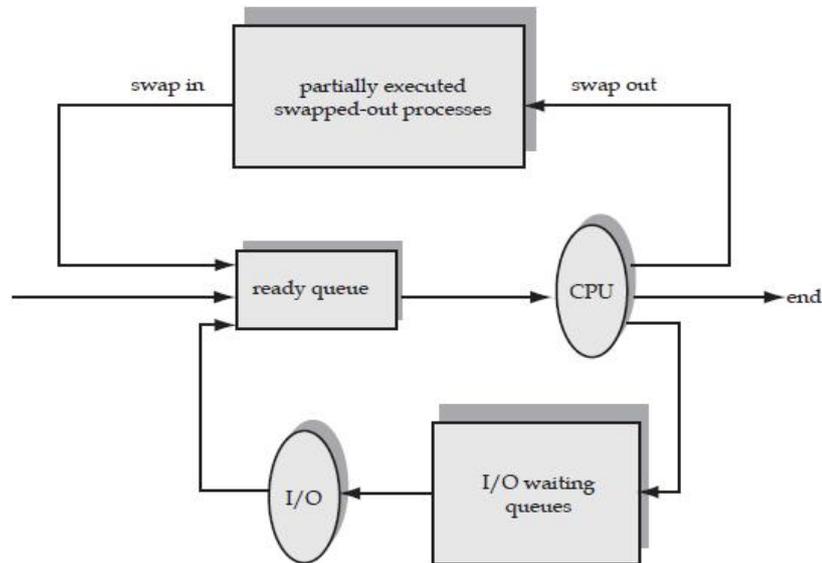


Figure 6.1: CPU Scheduling

Scheduling Mechanisms

A multiprogramming operating system allows more than one process to be loaded into the executable memory at a time and for the loaded process to share the CPU using time-multiplexing. Part of the reason for using multiprogramming is that the operating system itself is implemented as one or more processes, so there must be a way for the operating system and application processes to share the CPU. Another main reason is the need for processes to perform I/O operations in the normal course of computation. Since I/O operations ordinarily require orders of magnitude more time to complete than do CPU instructions, multiprogramming systems allocate the CPU to another process whenever a process invokes an I/O operation.

Goals for Scheduling

Make sure your scheduling strategy is good enough with the following criteria:

1. **Utilization/Efficiency:** keep the CPU busy 100% of the time with useful work
2. **Throughput:** maximize the number of jobs processed per hour.
3. **Turnaround time:** from the time of submission to the time of completion, minimize the time batch users must wait for output
4. **Waiting time:** Sum of times spent in ready queue - Minimize this
5. **Response Time:** time from submission till the first response is produced, minimize response time for interactive users
6. **Fairness:** make sure each process gets a fair share of the CPU

Context Switching

Typically, there are several tasks to perform in a computer system. So, if one task requires some I/O operation, you want to initiate the I/O operation and go on to the next task. You will come back to it later. This act of switching from one process to another is called a "Context Switch". When you return back to a process, you should resume where you left off. For all practical purposes, this process should never know there was a switch, and it should look like this was the only process in the system.

To implement this, on a context switch, you have to

1. Save the context of the current process
2. Select the next process to run
3. Restore the context of this new process.

Non-preemptive vs. Preemptive Scheduling

• Non-preemptive

Non-preemptive algorithms are designed so that once a process enters the running state (is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor).

`context_switch()` is called only when the process terminates or blocks.

• Preemptive

Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the process on the processor should be removed and returned to the ready list until it is once again the highest priority process in the system.

`context_switch()` is called even when the process is running usually done via a timer interrupt.

6.2 CPU Scheduling Basic Criteria

CPU scheduling is the basics of multiprogramming. By switching the CPU among several processes, the operating systems can make the computer more productive. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. On systems with 1 processor, only one process may run at a time; any other processes must wait until CPU is free to be rescheduled. In multiprogramming, a process executes until it must wait (either interrupted, or doing IO), at which point, the CPU is assigned to another process, which again, executes until it must wait, at which point another process gets the CPU, and so on. Processes generally execute a CPU burst, followed by an IO burst, followed by the CPU burst, followed by the CPU burst, etc. This cycle is central to all processes. Every process must have CPU bursts, and every process must do some IO. The operating system maintains what is known as a ready-queue. Processes on this queue are ready to be executed. Whenever a currently executing process needs to wait (does IO, is interrupted, etc.) the operating system picks a process from the ready queue and assigns the CPU to that process. The cycle then continues. There are many scheduling algorithms, ranging in complexity and robustness: First-come, First-serve scheduling, Shortest Job First scheduling, Round-Robin scheduling, etc. A major task of an operating system is to manage a collection of processes. In some cases, a single process may consist of a set of individual threads. In both situations, a system with a single CPU or a multi-processor system with fewer CPUs than processes has to divide CPU time among the different processes/threads that are competing to use it. This process is called CPU scheduling. There are many scheduling algorithms and various criteria to judge their performance. Different algorithms may favor different types of processes. Some criteria are as follows:

1. **CPU utilization:** CPU must be as busy as possible in performing different tasks. CPU utilization is more important in real-time system and multi-programmed systems.
2. **Throughput:** The number of processes executed in a specified time period is called throughput. The throughput increases for short processes. It decreases if the size of processes is huge.
3. **Turnaround Time:** The amount of time that is needed to execute a process is called turnaround time. It is the actual job time plus the waiting time.
4. **Waiting Time:** The amount of time the process has waited is called waiting time. It is the turnaround time minus actual job time.
5. **Response Time:** The amount of time between a request is Submitted and the first response is produced is called response time.

6.3 Scheduling Algorithms

Most Operating Systems today use very similar CPU time scheduling algorithms, all based on the same basic ideas, but with Operating System-specific adaptations and extensions. What follows is a description of those rough basic ideas. What should be remarked is that this algorithm is not the best algorithm that you can imagine, but it is, proven mathematically and by experience in the early days of OS programming (sixties and seventies), the algorithm that is the closest to the 'best' algorithm. Perhaps when computers get more powerful someday, then we might implement the ideal CPU time scheduler.

Another remark is that this algorithm is designed for general-purpose computers. Special-purpose Operating Systems or systems, and some real-time systems will use a very different algorithm. CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher. A CPU scheduling algorithm should try to maximize the following:

1. CPU utilization
2. Throughput

A CPU scheduling algorithm should try to minimize the following:

1. Turnaround time
2. Waiting time
3. Response time

Different algorithms are used for CPU scheduling.

First-Come, First-Served (FCFS)

This is a Non-Preemptive scheduling algorithm. FCFS strategy assigns priority to processes in the order in which they request the processor. The process that requests the CPU first is allocated the CPU first. When a process comes in, add its PCB to the tail of ready queue. When running process terminates, dequeue the process (PCB) at head of ready queue and run it. Figure 6.2 depicts the FCFS scheduling.

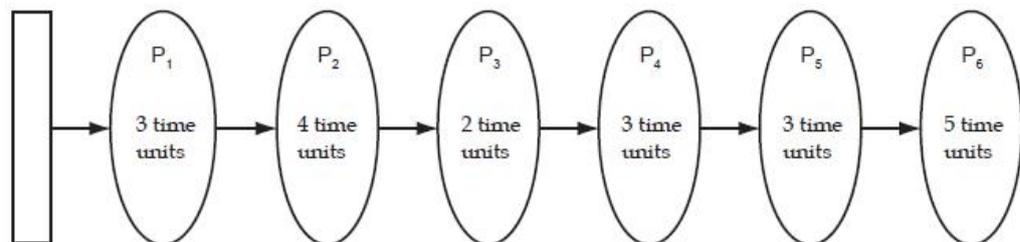


Figure 6.2: FCFS Scheduling

Advantage

Very simple

Disadvantages

1. Long average and worst-case waiting times
2. Poor dynamic behavior (convoy effect - short process behind long process as in figure 6.3)

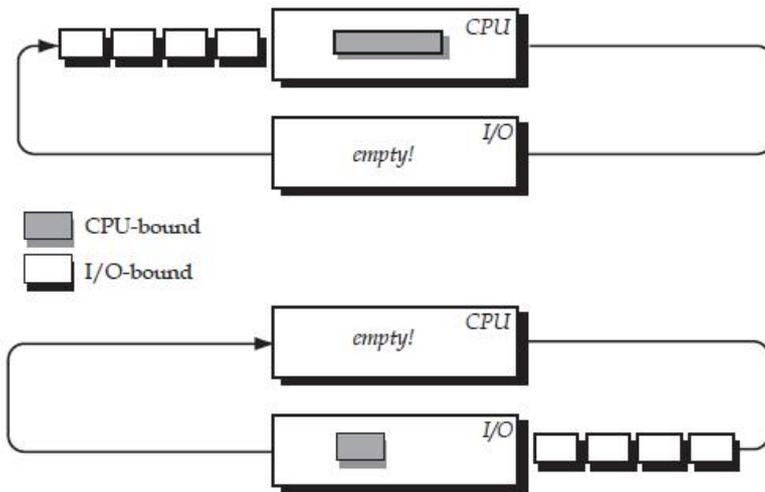


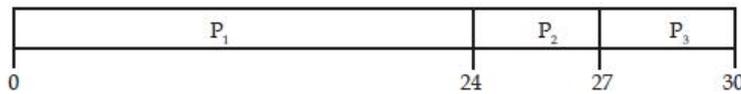
Figure 6.3: Convey Effects



Example:

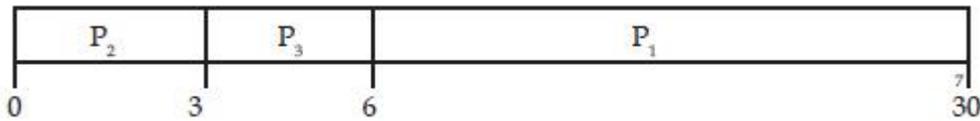
Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

Suppose that the processes arrive in the order: P₁, P₂, P₃. The Gantt chart for the schedule is:



Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
 Average waiting time: $(0 + 24 + 27)/3 = 17$

Suppose that the processes arrive in the order P₂, P₃, P₁. The Gantt chart for the schedule is:



Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
 Average waiting time: $(6 + 0 + 3)/3 = 3$

Shortest-Job-First (SJF)

The SJF algorithm takes processes that use the shortest CPU time first. Mathematically seen, and corresponding to the experience, this is the ideal scheduling algorithm. I won't give details in here about the performance. It's all to do about overhead and response time, actually: the system will be fast when the scheduler doesn't take much of the CPU time itself, and when interactive processes react quickly (get a fast response). This system would be very good.

The overhead caused by the algorithm itself is huge, however. The scheduler would have to implement some way to time the CPU usage of processes and predict it, or the user should tell the scheduler how long a job (this is really a word that comes from very early computer design, when Batch Job Operating Systems were used would take. So, it is impossible to implement this algorithm without hurting performance very much.

Advantage

Minimizes average waiting times.

Disadvantages

1. How to determine length of next CPU burst?

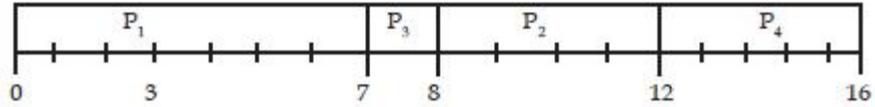
2. Starvation of jobs with long CPU bursts.



Examples

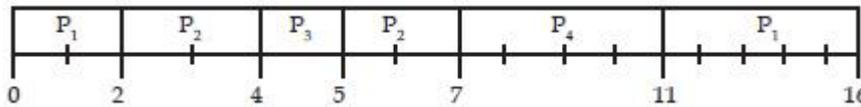
Process	Arrival Time	Burst Time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4

1. SJF (non-preemptive)



Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

2. SRT (preemptive SJF)



Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

Shortest Remaining Time (SRT)

Shortest remaining time is a method of CPU scheduling that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added, though this threat can be minimal when process times follow a heavy-tailed distribution. Like shortest job first scheduling, shortest remaining time scheduling is rarely used outside of specialized environments because it requires accurate estimations of the runtime of all processes that are waiting to execute.

Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities to compute the priority of a process. Figure 6.4 depicts the priority scheduling.



Example: Time limits, memory requirements, no. of open files, ratio of average I/O burst time to average CPU burst time etc. external priorities are set by criteria that are external to the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring work and other often political factors.

Priority scheduling can be preemptive or non-preemptive. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue. A major problem with priority

scheduling algorithms is indefinite blocking or starvation. This can be solved by a technique called aging wherein I gradually increase the priority of a long waiting process.

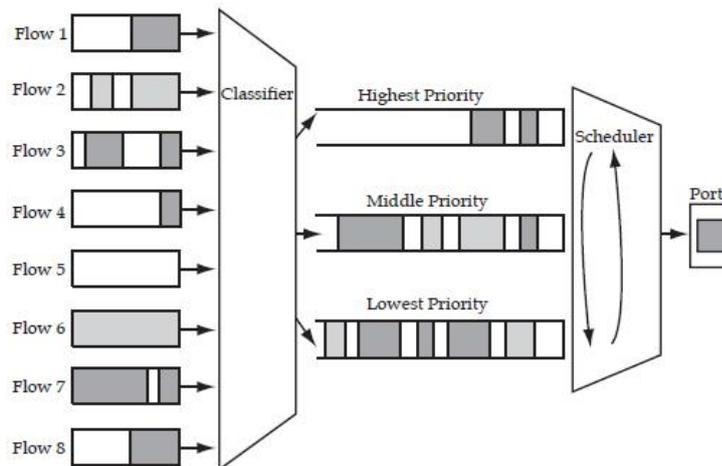


Figure 6.4: Priority Scheduling

Advantages and Disadvantages

The main advantage is the important jobs can be finished earlier as much as possible. The

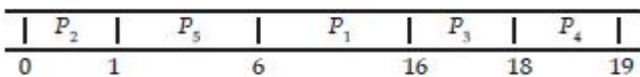


Examples

Process	Burst Time	Arrival	Priority
P ₁	10	0	3
P ₂	1	0	1
P ₃	2	0	4
P ₄	1	0	5
P ₅	5	0	2

main disadvantage is the lower priority jobs will starve.

Gantt Chart



Average waiting time: $(0+1+6+16+18)/5 = 8.2$

Round-Robin (RR)

Round-robin scheduling is really the easiest way of scheduling. All processes form a circular array and the scheduler gives control to each process at a time. It is off course very easy to implement and causes almost no overhead, when compared to all other algorithms. But response time is very low for the processes that need it. Of course, it is not the algorithm I want, but it can be used eventually. This algorithm is not the best at all for general-purpose Operating Systems, but it is useful for batch processing Operating Systems, in which all jobs have the same priority, and in which response time is of minor or no importance. And this priority leads us to the next way of scheduling.

Advantages

Simple, low overhead, works for interactive systems

Disadvantages

1. If quantum is too small, too much time wasted in context switching
2. If too large (i.e., longer than mean CPU burst), approaches FCFS

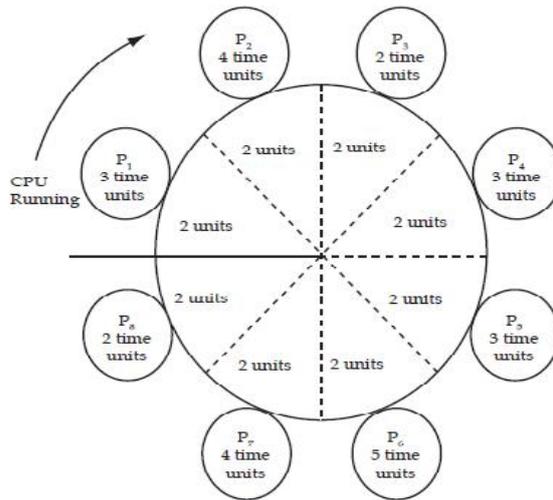


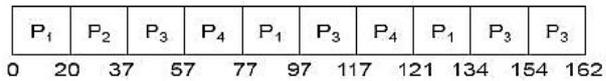
Figure 6.5: Round Robin Scheduling



Example of RR with Time Quantum = 20

Process	Burst Time
P ₁	53
P ₂	17
P ₃	68
P ₄	24

The Gantt chart is:



Typically, higher average turnaround than SJF, but better response

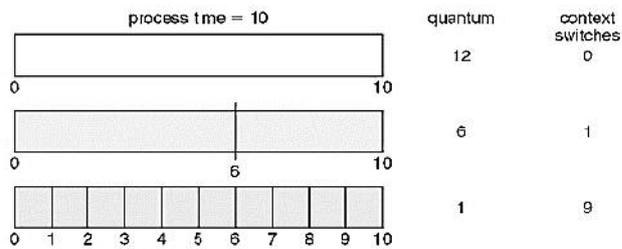


Figure 6.6: Time Quantum and Context Switch Time

Turnaround Time Varies with the Time Quantum

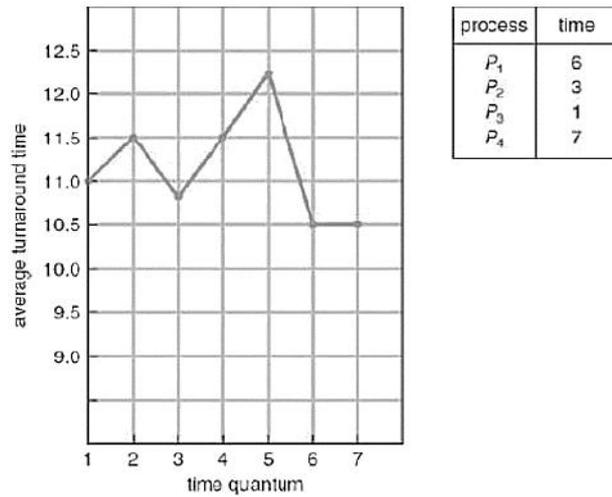


Figure 6.7: Variation of Turnaround Time with the Time Quantum



Example: Assume you have the following jobs to execute with one processor, with the jobs arriving in the order listed here:

I	T(pi)	Arrival Time
0	80	0
1	20	10
2	10	10
3	20	80
4	50	85

1. Suppose a system uses RR scheduling with a quantum of 15. Create a Gantt chart illustrating the execution of these processes?
2. What is the turnaround time for process p₃?
3. What is the average wait time for the processes?

Solution:

1. As the Round-Robin Scheduling follows a circular queue implementation, the Gantt chart is as follows:

P0	P1	P2	P0	P1	P2	P0	P1	P3	p4	P0	P3	P4	P0	P4	
0	15	30	45	60	75	85	100	110	125	140	155	160	175	190	205

2. The turnaround time for process P3 is =160-80= 80 sec.
3. Average waiting time:
 Waiting time for process p0 = 0 sec.
 Waiting time for process p1 = 5 sec.
 Waiting time for process p2 = 20 sec.
 Waiting time for process p3 = 30 sec.
 Waiting time for process p4 = 40 sec.

Therefore, the average waiting time is (0+5+20+30+40)/5 = 22 sec.

Multilevel Feedback Queue Scheduling

In this CPU schedule a process is allowed to move between queues. If a process uses too much CPU time, it will be moved to a lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queues. Similarly, a process that waits too long in a lower priority queue may be moved to a higher priority queue as shown in figure 6.8.

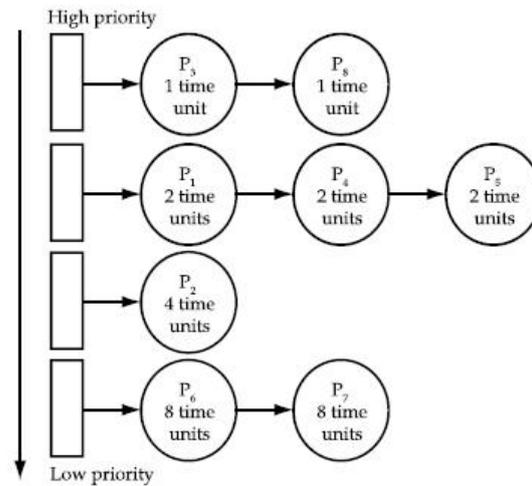


Figure 6.8: Multi-level Feedback Queue Scheduling



Example: Multilevel Feedback Queue(as shown in figure 6.9)

Three Queues

1. Q_0 -RR with time quantum 8 milliseconds
2. Q_1 -RR time quantum 16 milliseconds
3. Q_2 -FCFS

Scheduling

1. A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
2. At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

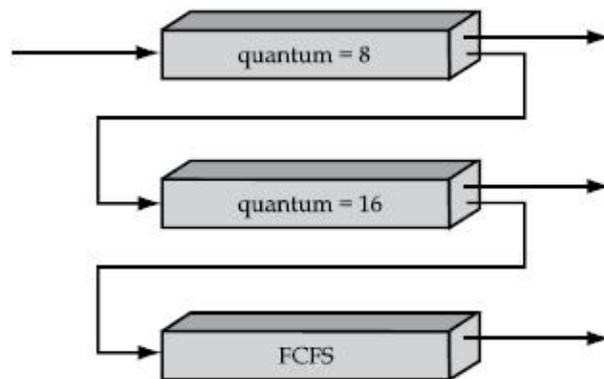


Figure 6.9: Scheduling Queues

Real-Time Scheduling

Real-time systems design is an increasingly important topic in systems research communities as well as the software industries. Real-time applications and their requirements can be found in almost every area of operating systems and networking research. An incomplete list of such domains includes distributed systems, embedded systems, network protocol processing, aircraft design, spacecraft design..., and the list goes on. One of the most important responsibilities of a real-time system is to schedule tasks according to their deadlines in order to guarantee that all real-time activities achieve the required service level. Many scheduling algorithms exist for a variety of task models, but fundamental to many of these are the earliest deadline first (EDF) and rate-monotonic (RM) scheduling policies. A schedule for a set of tasks is said to be feasible if a proof

exists that every task instance in the set will complete processing by its associated deadline. Also, a task set is schedulable if there exists a feasible schedule for the set. The utilization associated with a given task schedule and resource (i.e. CPU) is the fraction of time that the resource is allocated over the time that the scheduler is active. Figure 6.10 shows real time scheduling.

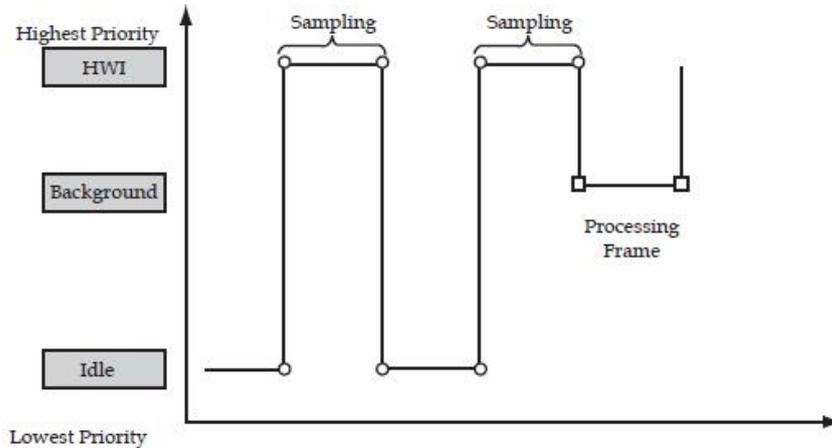


Figure 6.10: Real Time Scheduling

Earliest Deadline First

The EDF scheduling algorithm is a preemptive and dynamic priority scheduler that executes tasks in order of the time remaining before their corresponding deadlines. Tasks with the least time remaining before their deadline are executed before tasks with more remaining time. At each invocation of the scheduler, the remaining time is calculated for each waiting task, and the task with the least remaining time is dispatched. If a task set is schedulable, the EDF algorithm results in a schedule that achieves optimal resource utilization. However, EDF is shown to be unpredictable if the required utilization exceeds 100%, known as an overload condition. EDF is useful for scheduling periodic tasks, since the dynamic priorities of tasks do not depend on the determinism of request periods.

Rate Monotonic

Under the static-priority rate monotonic scheduling algorithm, tasks are ordered according to the value of their request period, T . Tasks with shorter request periods are assigned higher priority than those with longer periods. Liu and Layland proved that a feasible schedule is found under rate monotonic scheduling if the total requested utilization is less than or equal to \ln , which is approximately 69.3%. RM scheduling is optimal with respect to maximum utilization over all static-priority schedulers. However, this scheduling policy only supports tasks that fit the periodic task model, since priorities depend upon request periods. Because the request times of a periodic task are not always predictable, these tasks are not supported by the RM algorithm, but are instead typically scheduled using a dynamic priority scheduler such as EDF.

Characteristics of Scheduling Algorithms

	FCFS	Round Robin	SJF	SRT	HRRN	Feedback
Selection Function	max[w]	constant	Min[s]	min[s-e]	max[(w+s)/s]	see text
Decision Mode	Non-preemptive	preemptive	Non-preemptive	preemptive	Non-preemptive	Preemptive at time quantum
Throughput	N/A	low for small quantum	high	High	High	N/A
Response Time	May be high	good for short processes	good for short processes	Good	Good	N/A
Overhead	minimal	low	Can be high	Can be high	Can be high	Can be high
Starvation	No	No	Possible	Possible	No	Possible

w = time spent in the system so far, waiting and executing

e = time spent in execution so far.

s = total service time required by the process, including e.

6.4 Operating Systems and Scheduling Types

- 1) **Solaris 2** uses priority-based process scheduling.
- 2) **Windows 2000** uses a priority-based preemptive scheduling algorithm.
- 3) **Linux** provides two separate process-scheduling algorithms: one is designed for timesharing processes for fair preemptive scheduling among multiple processes; the other designed for real-time tasks:
 - a) For processes in the time-sharing class Linux uses a prioritized credit-based algorithm
 - b) Real-time scheduling: Linux implements two real-time scheduling classes namely FCFS (First come first serve) and RR (Round Robin)

6.5 Types of Scheduling

In many multitasking systems the processor scheduling subsystem operates on three levels, differentiated by the time scale at which they perform their operations. In this sense differentiate among:

1. **Long term scheduling:** which determines which programs are admitted to the system for execution and when, and which ones should be exited.
2. **Medium term scheduling:** which determines when processes are to be suspended and resumed;
3. **Short term scheduling (or dispatching):** which determines which of the ready processes can have CPU resources, and for how long.

Long-term Scheduling

Long term scheduling obviously controls the degree of multiprogramming in multitasking systems, following certain policies to decide whether the system can honor a new job submission or, if more than one job is submitted, which of them should be selected. The need for some form of compromise between degree of multiprogramming and throughput seems evident, especially when one considers interactive systems. The higher the number of processes, in fact, the smaller the time each of them may control CPU for, if a fair share of responsiveness is to be given to all processes. Moreover, you have already seen that a too high number of processes causes waste of CPU time for system housekeeping chores (trashing in virtual memory systems is a particularly nasty example of this). However, the number of active processes should be high enough to keep the CPU busy servicing the payload (i.e. the user processes) as much as possible, by ensuring that - on average - there always be a sufficient number of processes not waiting for I/O.

Simple policies for long term scheduling are:

1. **Simple First Come First Served (FCFS):** It's essentially a FIFO scheme. All job requests (e.g. a submission of a batch program, or a user trying to log in in a time-shared system) are honored up to a fixed system load limit, further requests being refused tout court, or enqueued for later processing.
2. **Priority schemes:** Note that in the context of long-term scheduling "priority" has a different meaning than in dispatching: here it affects the choice of a program to be entered the system as a process, there the choice of which ready process should be executed. Long term scheduling is performed when a new process is created. It is shown in the figure below. If the number of ready processes in the ready queue becomes very high, then there is an overhead on the operating system (i.e., processor) for maintaining long lists, context switching and dispatching increases. Therefore, allow only limited number of processes in to the ready queue. The "long-term scheduler" manages this. Long-term scheduler determines which programs are admitted into the system for processing. Once when admit a process or job, it becomes

process and is added to the queue for the short-term scheduler. In some systems, a newly created process begins in a swapped-out condition, in which case it is added to a queue for the medium-term scheduler scheduling manage queues to minimize queuing delay and to optimize performance.

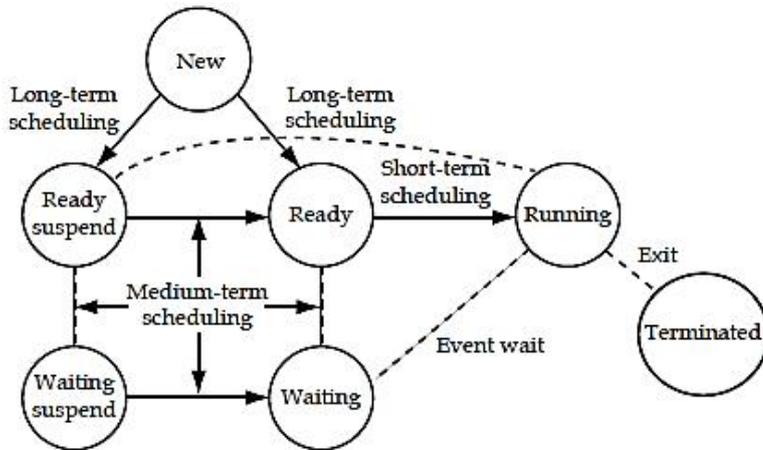


Figure 6.11: Long term Scheduling

The long-term scheduler limits the number of processes to allow for processing by taking the decision to add one or more new jobs, based on FCFS (First-Come, first-serve) basis or priority or execution time or Input/output requirements. Long-term scheduler executes relatively infrequently.

Medium Term Scheduling

Medium term scheduling is essentially concerned with memory management; hence it's very often designed as a part of the memory management subsystem of an OS. Its efficient interaction with the short-term scheduler is essential for system performances, especially in virtual memory systems. This is the reason why in paged system the pager process is usually run at a very high (dispatching) priority level.

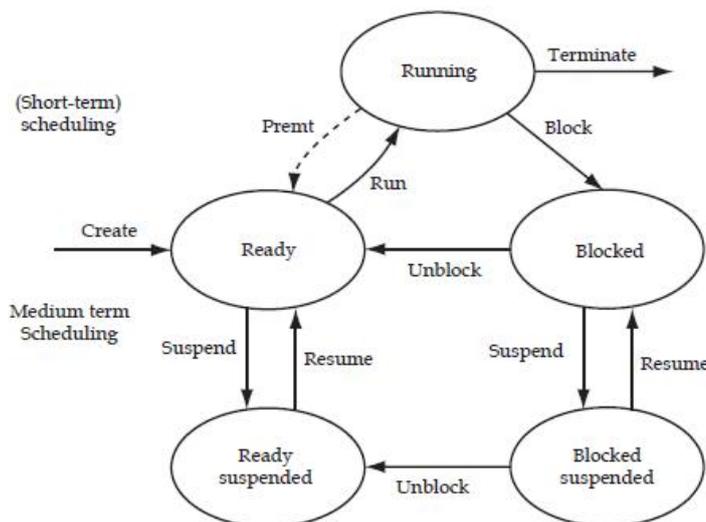


Figure 6.12: Medium term Scheduling

Unblock is done by another task (a.k.a. wakeup, release, V) Block is a.k.a. sleep, request, P) In addition to the short-term scheduling we have discussed, we add medium-term scheduling in which decisions are made at a coarser time scale. Recall my favorite diagram, shown again on the right. Medium term scheduling determines the transitions from the top triangle to the bottom line. We suspend (swap out) some process if memory is over-committed, dropping the (ready or blocked) process down. We also need resume transitions to return a process to the top triangle.

Criteria for choosing a victim to suspend include:

- How long since previously suspended.
- How much CPU time used recently.
- How much memory does it use.
- External priority (pay more, get swapped out less).

Short-term Scheduling

Short term scheduling concerns with the allocation of CPU time to processes in order to meet some pre-defined system performance objectives. The definition of these objectives (scheduling policy) is an overall system design issue, and determines the "character" of the operating system from the user's (i.e. the buyer's) point of view, giving rise to the traditional distinctions among "multi-purpose, time shared", "batch production", "real-time" systems, and so on. From a user's point of view, the performance criteria base on:

- 1) **Response time:** The interval of time from the moment a service is requested until the response begins to be received. In time-shared, interactive systems this is a better measure of responsiveness from a user's point of view than turnaround time, since processes may begin to produce output early in their execution.
- 2) **Turnaround time:** The interval between the submission of a process and the completion of its execution, including the actual running time, plus the time spent sleeping before being dispatched or while waiting to access various resources. This is the appropriate responsiveness measure for batch production, as well as for time-shared systems that maintain multiple batch queues, sharing CPU time among them.
- 3) **Meeting deadlines:** The ability of the OS to meet pre-defined deadlines for job completion. It makes sense only when the minimal execution time of an application can be accurately predicted.
- 4) **Predictability:** The ability of the system to ensure that a given task is executed within a certain time interval, and/or to ensure that a certain constant response time is granted within a strict tolerance, no matter what the machine load is.

When the overall system performance is considered, additional scheduling criteria must be considered:

- 1) **Throughput:** The rate of completion of processes (processes completed per unit time). This is a "raw" measure of how much work is performed, since it depends on the execution length of processes, but it's obviously affected by the scheduling policy.
- 2) **User processor utilization:** Time (percentage of unit time) during which the CPU is running user processes. This is a measure of how well the system can serve the payload and keep at minimum time spent in housekeeping chores.
- 3) **Overall processor utilization:** Time percentage during which the CPU is busy. It's a significant criterion for expensive hardware, that must be kept busy as much as possible in order to be justify its cost (e.g. supercomputers for numerical calculus applications).
- 4) **Resource utilization balance:** It extends the idea of processor utilization to consider all system resources. A good scheduler should try to keep all the hardware resources in use at any time.

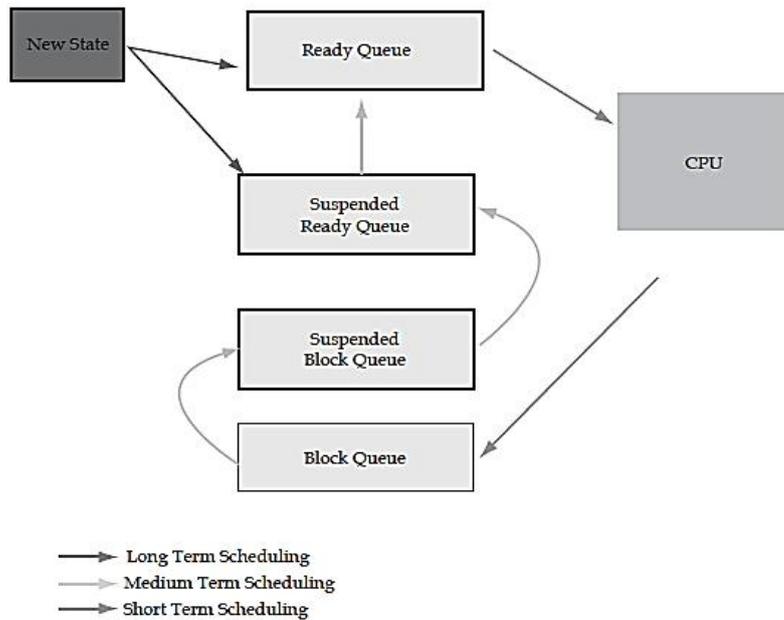


Figure 6.13: Medium term Scheduling

The design of the short-term scheduler is one of the critical areas in the overall system design, because of the immediate effects on system performance from the user's point of view. It's usually one of the trickiest as well: since most processor architectures support their own task switching facilities, the implementation of the process switch mechanism is generally machine-dependent. The result is that the actual process switch software is usually written in the assembly language of a particular machine, whether the operating system is meant to be portable across different machines or not.

6.6 Multiple Processor Scheduling

The development of appropriate scheduling schemes for multiprocessor systems is problematic. Not only are uni-processor algorithms not directly applicable but some of the apparently correct methods are counter intuitive. The scheduling problem for multiprocessor systems can be generally stated as "How can you execute a set of tasks T on a set of processors P subject to some set of optimizing criteria C ?" The most common goal of scheduling is to minimize the expected runtime of a task set. Examples of other scheduling criteria include minimizing the cost, minimizing communication delay, giving priority to certain users' processes, or needs for specialized hardware devices. The scheduling policy for a multiprocessor system usually embodies a mixture of several of these criteria. Issues in Multiprocessor Scheduling Solutions to the scheduling problem come in two general forms: algorithms and scheduling systems. Algorithms concentrate on policy while scheduling systems provide mechanism to implement the algorithms. Some scheduling systems run outside the operating system kernel, while others are part of a tightly-integrated distributed or parallel operating system.

Distributed systems communicate via message-passing, while parallel systems use shared memory. A task is the unit of computation in computing systems, and a job consists of one or more cooperating tasks. Global scheduling involves assigning a task to a particular processor within the system.

Local scheduling determines which of the set of available tasks at a processor runs next on that processor. Task migration can change the global mapping by moving a task to a new processor.

If you have several jobs, each composed of many tasks, you can either assign several processors to a single job, or you can assign several tasks to a single processor. The former is known as space sharing, and the latter is called time sharing. Global scheduling is often used to perform load sharing. Load sharing allows busy processors to off-load some of their work to less busy processors. Load balancing is a special case of load sharing, in which the goal is to keep the load even across all processors. Sender-initiated load sharing occurs when busy processors try to find idle processors to off-load some work. Receiver-initiated load sharing occurs when idle processors seek busy processors. It is now accepted wisdom that full load balancing is generally not worth doing, as the

small gain in execution time over simpler load sharing is more than offset by the effort expended in maintaining the balanced load as shown in figure 6.14.

As the system runs, new tasks arrive while old tasks complete execution (or are served). If the arrival rate is greater than the service rate then the system is said to be unstable. If tasks are served as least as fast as they arrive, the system is said to be stable. If the arrival rate is just slightly less than the service rate for a system, an unstable scheduling policy can push the system into instability. A stable policy will never make a stable system unstable.

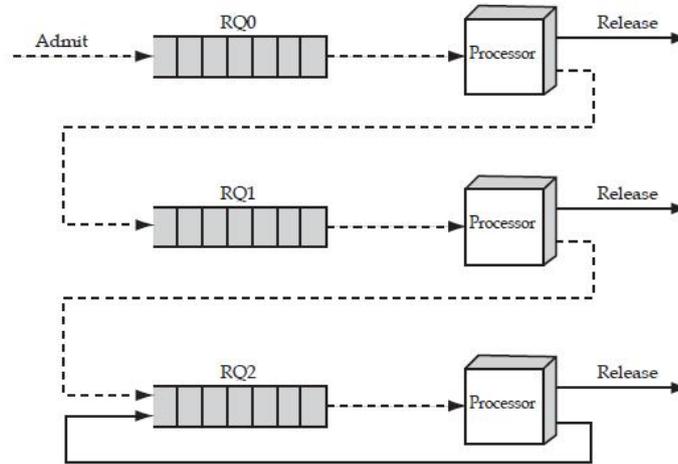


Figure 6.14: Multi-processor Queue

6.7 Thread Scheduling

The main approaches of threading scheduling are:

- 1) Load sharing
- 2) Gang scheduling
- 3) Dedicated processor assignment
- 4) Dynamic scheduling

1. Load Sharing

Processes are not assigned to a particular processor. A global queue of ready threads is maintained and each processor, when idle select a thread from the queue. There are three versions of load sharing are these are:

- 1) First come first served
- 2) Smallest number of threads first
- 3) Preemptive smallest number of threads first
- 1) **First come first served:** when a job arrives each of its threads is placed consecutively at the end of the shared queue. When a processor becomes idle it picks the next ready thread, which it executes until completion or blocking.
- 2) **Smallest number of threads first:** The shared ready queue is organized as a priority queue with **highest** priority given to threads from jobs with the smallest number of unscheduled threads. Jobs of equal priority are ordered according to which job arrives first.
- 3) **Preemptive smallest number of threads first:** Highest is given to jobs with the smallest

Advantages

Advantages of load sharing are:

- 1) The load is distributed evenly across the processors assuring that no processor is idle while work is available to do.
- 2) No centralized scheduler is required

3) The global queue can be organized and accessed by using any of the schemes.

Disadvantages

Disadvantages of load sharing are:

- 1) The central queue copies a region of memory that must be accessed in a manner that enforces mutual exclusion.
- 2) Preempted threads are unlikely to resume execution on the same processor.
- 3) If all threads are treated as a common pool of threads it is unlikely that all the threads of a program will gain access to processors at the same time.

2. Gang Scheduling

- 1) If closely related processes execute in parallel, synchronization blocking may be reduced.
- 2) Set of related threads of scheduled to run on a set of processors.
- 3) Gang scheduling has three parts.
 - (a) Groups of related threads are scheduled as a unit, a gang
 - (b) All members of a gang run simultaneously on different timeshared CPUs.
 - (c) All gang members start and end their time slices together.
- 4) The trick that makes gang scheduling work is that all CPU are scheduled synchronously. This means that time is divided into discrete quanta.
- 5) An example of how gang scheduling works is given in the Table 5.1. Here you have a multiprocessor with six CPU being used by five processes, A through E, with a total of 24 ready threads.

Table 6.1: Gang Scheduling

		CPU					
		0	1	2	3	4	5
Time slot	0	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	1	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	2	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	3	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆
	4	A ₀	A ₁	A ₂	A ₃	A ₄	A ₅
	5	B ₀	B ₁	B ₂	C ₀	C ₁	C ₂
	6	D ₀	D ₁	D ₂	D ₃	D ₄	E ₀
	7	E ₁	E ₂	E ₃	E ₄	E ₅	E ₆

- a) During time slot 0, threads A₀ through A₅ are scheduled and run
- b) During time slot 1, threads B₀, B₁, B₂, C₀, C₁, C₂ are scheduled and run
- c) During time slot 2, D's five threads and E₀ get to run
- d) The remaining six threads belonging to process E run in the time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.
- e) Gang scheduling is useful for applications where performance severely degrades when any part of the application is not running.

3. Dedicated Processor Assignment

- 1) When application is scheduled its threads are assigned to a processor.
- 2) Some processor may be idle and no multiprogramming of processors.
- 3) Provides implicit scheduling defined by assignment of threads to processors. For the duration of program execution, each program is allocated a set of processors equal in number to the number of threads in the program. Processors are chosen from the available pool.

4. Dynamic Scheduling

- 1) Number of threads in a process are altered dynamically by the application.
- 2) Operating system and the application are involved in making scheduling decisions. The OS is responsible for partitioning the processors among the jobs.
- 3) Operating system adjusts load to improve the use:
 - a) Assign idle processors.
 - b) New arrivals may be assigned to a processor that is used by a job currently using more than one processor.
 - c) Hold request until processor is available
 - d) New arrivals will be given a processor before existing running applications.

Summary

- The processes in the system can execute concurrently, and they must be created and deleted dynamically. Thus, the operating system must provide a mechanism (or facility) for process creation and termination. Processes can be terminated in one of two ways i.e. Normal Termination and Abnormal Termination.
- When more than one processes are executing concurrently in the operating system, then they are allowed to cooperate (both mutually and destructively) with each other.
- Those processes are known as cooperating process. Inter-Process Communication (IPC) is a set of techniques for the exchange of data among two or more threads in one or more processes.
- When two or more concurrent processes are reading or writing some shared data and the final result depends on who runs precisely when, are called race conditions.
- Critical Section is a part of the program where the shared memory is accessed. Mutual Exclusion is a way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semapho initialize'.
- Message passing is a form of inter process communication used in concurrent computing, where the communication is made by the sending of messages to recipients.

Keywords

CPU scheduling: It is the basic of multiprogramming where the task of selecting a waiting process from the ready queue and allocating the CPU to it.

CPU utilization: It is an important criterion in real-time system and multi-programmed systems where the CPU must be as busy as possible in performing different tasks.

Response Time: The amount of time between a request is Submitted and the first response is produced is called response time.

Throughput: The number of processes executed in a specified time period is called throughput.

Turnaround Time: The amount of time that is needed to execute a process is called turnaround time. It is the actual job time plus the waiting time.

Waiting Time: The amount of time the process has waited is called waiting time. It is the turnaround time minus actual job time.

Self Assessment

- 1) A header will contain pointers to the first and last PCBs in the list.
 - 2) scheduling is the basics of multiprogramming.
 - 3) A major task of an operating system is to manage a collection of
 - 4) The CPU is to the selected process by the dispatcher.
 - 5) is a method of CPU scheduling that is a preemptive version of shortest job next scheduling.
 - 6) A scheduling algorithm will simply put the new process at the head of the ready queue.
 - 7) scheduling is essentially concerned with memory management.
 - 8) The most common goal of scheduling is to of a task set.
 - 9) scheduling involves assigning a task to a particular processor within the system.
 - 10) scheduling is really the easiest way of scheduling.
- 11) is a kind of program without any kind of user interaction.
 - A. User Program
 - B. Job
 - C. Process
 - D. System Program
 - 12) The goal of CPU scheduling is to maximize the.....
 - A. Turnaround time
 - B. Waiting time
 - C. Throughput
 - D. Response time
 - 13) The time required by the process to complete the execution is called?
 - A. Burst Time
 - B. Arrival Time
 - C. Finish Time
 - D. None of these
 - 14) The condition when a number of programs which can be present in the memory at the same time is called?
 - A. Multi-tasking
 - B. Multi-programming
 - C. Time sharing
 - D. Real time

- 15) In a soft real-time system, a process is
- required to move between the queues using the concept of aging
 - required to complete a critical task within a guaranteed amount of time
 - the scheduling technique which requires that critical processes receive priority over less fortunate ones.
 - A technique that does not make use of priority scheduling

Answers for Self Assessment

1. ready-queue 2. CPU 3. Processes 4. allocated 5. Shortest remaining time
6. non preemptive priority 7. Medium term 8. minimize the expected runtime 9. Global 10. Round-robin
11. B 12. C 13. A 14. B 15. C

Review Questions

- Suppose that a scheduling algorithm favors those processes that have used the least processor time in the recent past. Why will this algorithm favor I/O-bound programs and yet not permanently starve CPU-bound programs?
- Assume you have the following

i	T(pi)
0	80
1	20
2	10
3	20
4	50

- Suppose a system uses FCFS scheduling. Create a Gantt chart illustrating the execution of these processes?
 - What is the turnaround time for process p_3 ?
 - What is the average wait time for the processes?
- Suppose a new process in a system arrives at an average of six processes per minute and each such process requires an average of 8 seconds of service time. Estimate the fraction of time the CPU is busy in a system with a single processor.
 - A CPU scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many possible different schedules are there? Give a formula in terms of n .
 - Many CPU-scheduling algorithms are parameterized. For example, the RR algorithm requires a parameter to indicate the time slice. Multilevel feedback queues require parameters to define the number of queues, the scheduling algorithms for each queue, the criteria used to move processes between queues, and so on.

Unit 06: CPU Scheduling

These algorithms are a really sets of algorithms (for example, the set of RR algorithms for all time slices, and so on). One set of algorithms may include another (for example, the FCFS algorithm is the RR algorithm with an infinite time quantum). What (if any) relation holds between the following pairs of sets of algorithms?

- a) Priority and SJF
 - b) Multilevel Feedback Queues and FCFS
 - c) Priority and FCFS
 - d) RR and SJF
- 6) Distinguish between long term and short-term scheduling.
 - 7) Consider the following set of processes, with the length of the CPU burst given in milliseconds.

Process	Burst Time	Priority
P ₁	10	3
P ₂	1	1
P ₃	2	3
P ₄	1	4
P ₅	5	2

The processes are assumed to have arrived in the order P₁, P₂, P₃, P₄, P₅ all at time 0.

- a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1).
 - b) What is the turnaround time of each process for each of the scheduling algorithms in part a?
 - c) What is the waiting time of each process for each of the scheduling algorithms in part a?
- 8) Consider the following set of processes, with the length of the CPU burst and arrival time given in milliseconds.

Process	Burst Time	Priority
P ₁	8	0
P ₂	4	0.4
P ₃	1	1

- a) Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, (the algorithm can look into the future and wait for a shorter process that will arrive).
 - b) What is the turnaround time of each process for each of the scheduling algorithms in part a?
 - c) What is the waiting time of each process for each of the scheduling algorithms in part a?
- 9) Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:
 - a) First Come First Served
 - b) Round Robin
 - c) Multilevel Feedback Queues

10. Write short notes on:

- a) Waiting time
- b) Response time
- c) Throughput



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Colin Ritchie, Operating Systems, BPB Publications.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 07: Scheduling Algorithms

CONTENTS

Objectives

Introduction

- 7.1 Scheduling Algorithms
- 7.2 First-Come, First-Served (FCFS)
- 7.3 Shortest Remaining Time (SRT)
- 7.4 Priority Scheduling
- 7.5 Round-Robin (RR)
- 7.6 Multilevel Feedback Queue Scheduling
- 7.7 Real-Time Scheduling
- 7.8 Earliest Deadline First
- 7.9 Rate Monotonic
- 7.10 Operating Systems and Scheduling Types

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Describe CPU scheduling
- Explain CPU scheduling basic criteria
- Know scheduling algorithms
- Describe types of scheduling
- Explain multiple processor scheduling
- Define thread scheduling

Introduction

CPU scheduling is the basics of multiprogramming. By switching the CPU among several processes, the operating systems can make the computer more productive. The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization.

7.1 Scheduling Algorithms

Most Operating Systems today use very similar CPU time scheduling algorithms, all based on the same basic ideas, but with Operating System-specific adaptations and extensions. What follows is a description of those rough basic ideas.

What should be remarked is that this algorithm is not the best algorithm that you can imagine, but it is, proven mathematically and by experience in the early days of OS programming (sixties and seventies), the algorithm that is the closest to the 'best' algorithm. Perhaps when computers get more powerful someday then we might implement the ideal CPU time scheduler.

Another remark is that this algorithm is designed for general-purpose computers. Special-purpose Operating Systems or systems, and some real-time systems will use a very different algorithm.

CPU scheduling is the task of selecting a waiting process from the ready queue and allocating the CPU to it. The CPU is allocated to the selected process by the dispatcher.

A CPU scheduling algorithm should try to maximize the following:

1. CPU utilization
2. Throughput

A CPU scheduling algorithm should try to minimize the following:

1. Turnaround time
2. Waiting time
3. Response time

Different algorithms are used for CPU scheduling.

7.2 First-Come, First-Served (FCFS)

This is a non-Preemptive scheduling algorithm. FCFS strategy assigns priority to processes in the order in which they request the processor. The process that requests the CPU first is allocated the CPU first. When a process comes in, add its PCB to the tail of ready queue. When running process terminates, dequeue the process (PCB) at head of ready queue and run it.

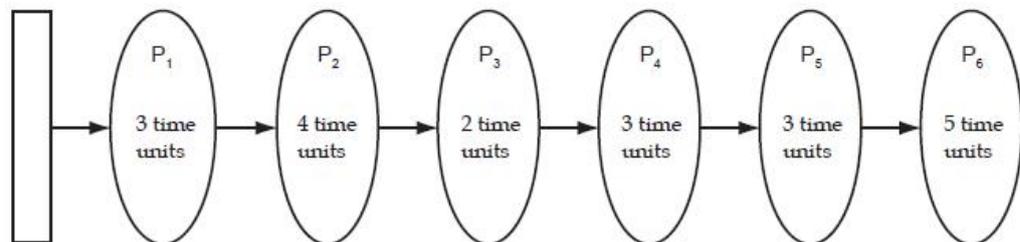


Figure: FCFS Scheduling

Advantage

- Very simple

Disadvantages

1. Long average and worst-case waiting times
2. Poor dynamic behavior (convoy effect - short process behind long process)

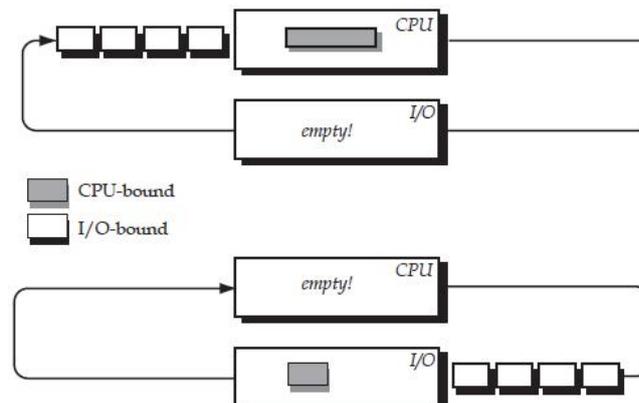


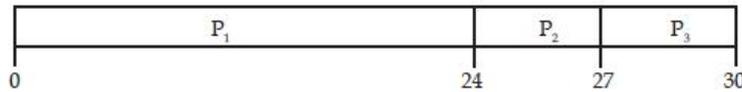
Figure: Convey Effects



Example:

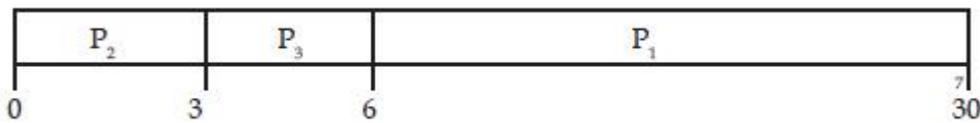
Process	Burst Time
P ₁	24
P ₂	3
P ₃	3

Suppose that the processes arrive in the order: P₁, P₂, P₃. The Gantt chart for the schedule is:



Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
 Average waiting time: $(0 + 24 + 27)/3 = 17$

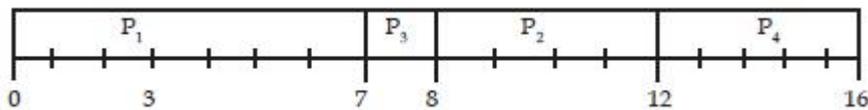
Suppose that the processes arrive in the order P₂, P₃, P₁. The Gantt chart for the schedule is:



Example:

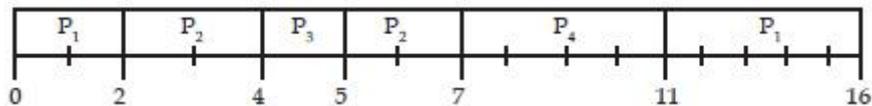
Process	Arrival Time	Burst Time
P ₁	0.0	7
P ₂	2.0	4
P ₃	4.0	1
P ₄	5.0	4

1. SJF (non-preemptive)



Average waiting time = $(0 + 6 + 3 + 7)/4 = 4$

2. SRT (preemptive SJB)



Average waiting time = $(9 + 1 + 0 + 2)/4 = 3$

7.3 Shortest Remaining Time (SRT)

Shortest remaining time is a method of CPU scheduling that is a preemptive version of shortest job next scheduling. In this scheduling algorithm, the process with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only takes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute. However, it has the potential for process starvation for processes which will require a long time to complete if short processes are continually added, though this threat can be minimal when process times follow a heavy-tailed distribution. Like shortest job first scheduling, shortest

remaining time scheduling is rarely used outside of specialized environments because it requires accurate estimations of the runtime of all processes that are waiting to execute.

7.4 Priority Scheduling

A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Priority can be defined either internally or externally. Internally defined priorities use some measurable quantities to compute the priority of a process.

Example: Time limits, memory requirements, no. of open files, ratio of average I/O burst time to average CPU burst time etc. external priorities are set by criteria that are external to the OS, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring work and other often political factors.

Priority scheduling can be preemptive or non-preemptive. A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-preemptive priority scheduling algorithm will simply put the new process at the head of the ready queue. A major problem with priority scheduling algorithms is indefinite blocking or starvation. This can be solved by a technique called aging wherein I gradually increase the priority of a long waiting process.

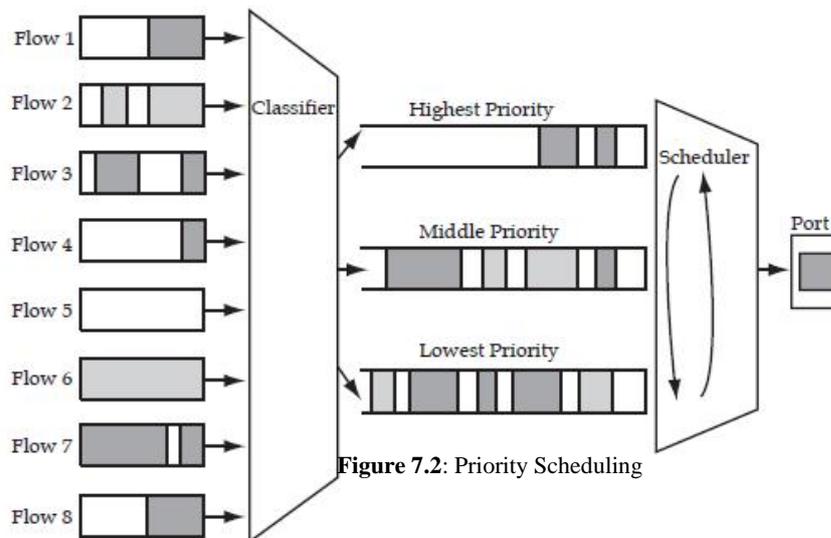


Figure 7.2: Priority Scheduling

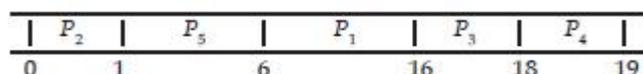
Advantages and Disadvantages

The main advantage is the important jobs can be finished earlier as much as possible. The main disadvantage is the lower priority jobs will starve.

Example:

Process	Burst Time	Arrival	Priority
P ₁	10	0	3
P ₂	1	0	1
P ₃	2	0	4
P ₄	1	0	5
P ₅	5	0	2

Gantt Chart



Average waiting time: $(0+1+6+16+18)/5 = 8.2$

7.5 Round-Robin (RR)

Round-robin scheduling is really the easiest way of scheduling. All processes form a circular array and the scheduler gives control to each process at a time. It is of course very easy to implement and causes almost no overhead, when compared to all other algorithms. But response time is very low for the processes that need it. Of course, it is not the algorithm I want, but it can be used eventually. This algorithm is not the best at all for general-purpose Operating Systems, but it is useful for batch processing Operating Systems, in which all jobs have the same priority, and in which response time is of minor or no importance. And this priority leads us to the next way of scheduling.

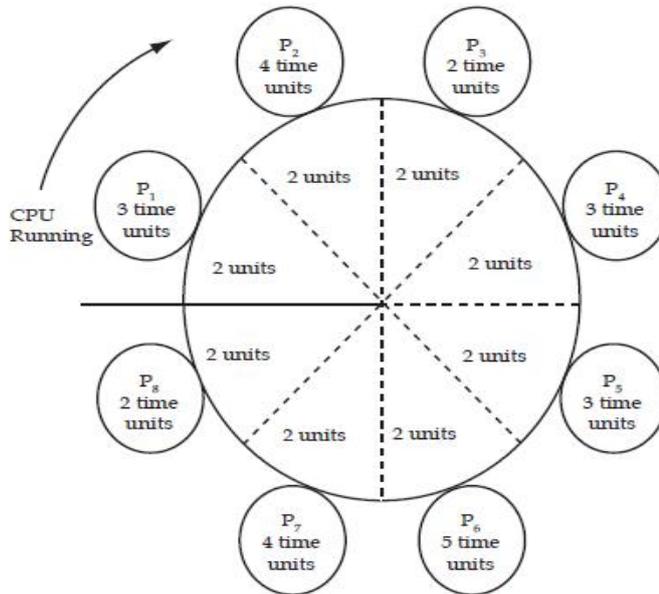


Figure: Round Robin Scheduling

Advantages

Simple, low overhead, works for interactive systems

Disadvantages

1. If quantum is too small, too much time wasted in context switching
2. If too large (i.e., longer than mean CPU burst), approaches FCFS



Example of RR with Time Quantum = 20

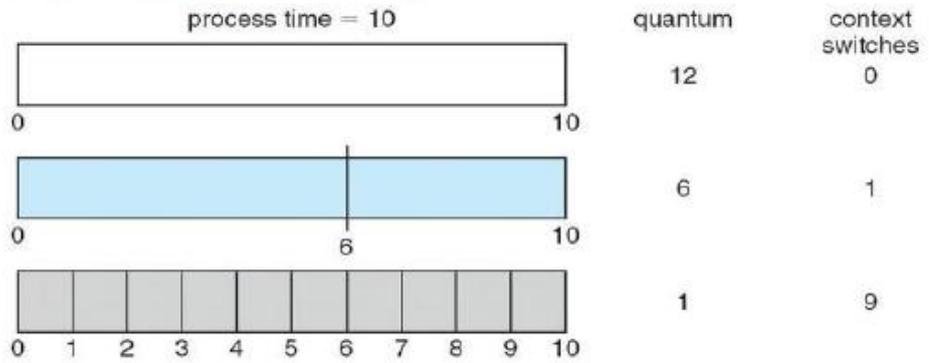
Process	Burst Time
P ₁	53
P ₂	17
P ₃	68
P ₄	24

The Gantt chart is:

P ₁	P ₂	P ₃	P ₄	P ₁	P ₃	P ₄	P ₁	P ₃	P ₃	
0	20	37	57	77	97	117	121	134	154	162

Typically, higher average turnaround than SJF, but better response

Time Quantum and Context Switch Time



Turnaround Time Varies with the Time Quantum

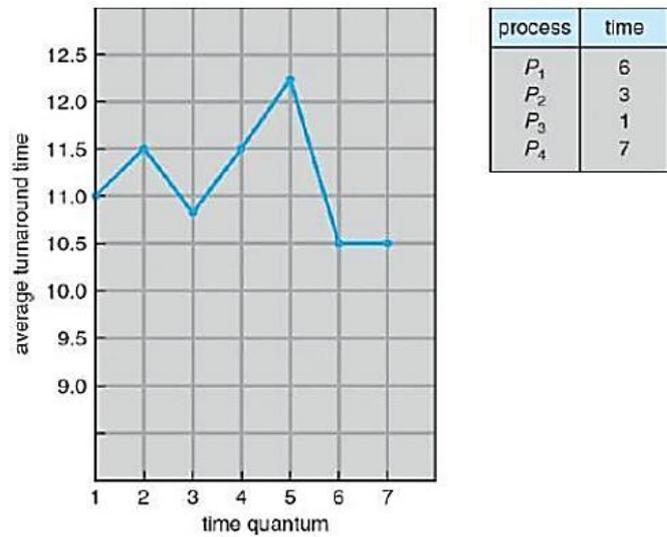


Figure: Variation of Turnaround Time with the Time Quantum



Example: Assume you have the following jobs to execute with one processor, with the jobs arriving in the order listed here:

i	T(pi)	Arrival Time
0	80	0
1	20	10
2	10	10
3	20	80
4	50	85

1. Suppose a system uses RR scheduling with a quantum of 15. Create a Gantt chart illustrating the execution of these processes?
2. What is the turnaround time for process p_3 ?
3. What is the average wait time for the processes?

Solution:

- As the Round-Robin Scheduling follows a circular queue implementation, the Gantt chart is as follows:

P0	P1	P2	P0	P1	P2	P0	P1	P3	p4	P0	P3	P4	P0	P4	
0	15	30	45	60	75	85	100	110	125	140	155	160	175	190	205

- The turnaround time for process P3 is =160-80
= 80 sec.

- Average waiting time:

Waiting time for process p0 = 0 sec.

Waiting time for process p1 = 5 sec.

Waiting time for process p2 = 20 sec.

Waiting time for process p3 = 30 sec.

Waiting time for process p4 = 40 sec.

Therefore, the average waiting time is $(0+5+20+30+40)/5 = 22$ sec.

7.6 Multilevel Feedback Queue Scheduling

In this CPU schedule a process is allowed to move between queues. If a process uses too much CPU time, it will be moved to a lower priority queue. This scheme leaves I/O bound and interactive processes in the higher priority queues. Similarly, a process that waits too long in a lower priority queue may be moved to a higher priority queue.

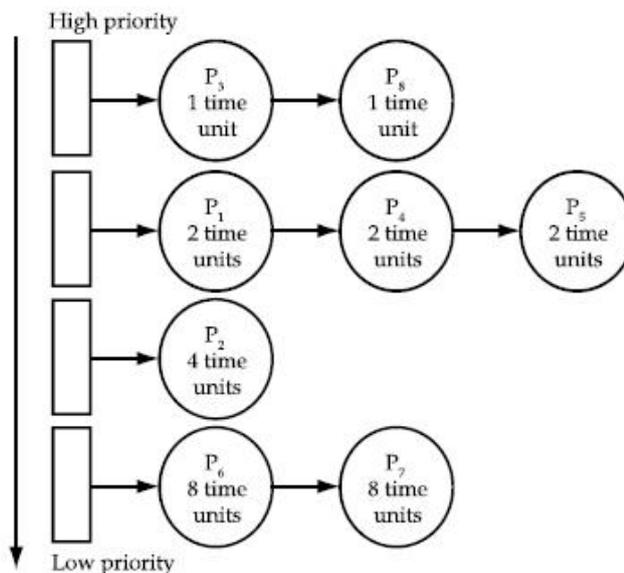


Figure 5.6: Multi-level Feedback Queue Scheduling



Example: Multilevel Feedback Queue

Three Queues

- Q₀-RR with time quantum 8 milliseconds
- Q₁-RR time quantum 16 milliseconds
- Q₂-FCFS

Scheduling

1. A new job enters queue Q_0 which is served FCFS. When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
2. At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2 .

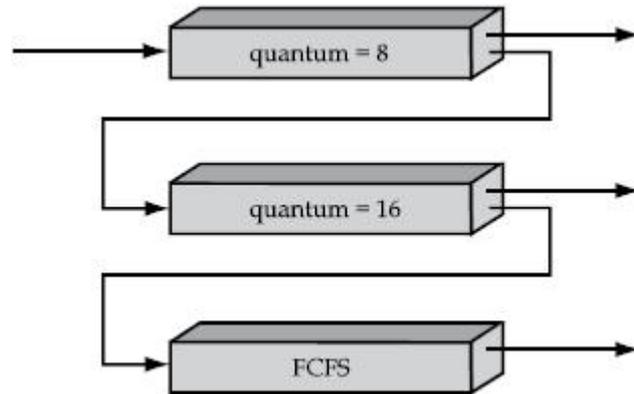


Figure: Scheduling Queues

7.7 Real-Time Scheduling

Real-time systems design is an increasingly important topic in systems research communities as well as the software industries. Real-time applications and their requirements can be found in almost every area of operating systems and networking research. An incomplete list of such domains includes distributed systems, embedded systems, network protocol processing, aircraft design, spacecraft design..., and the list goes on.

One of the most important responsibilities of a real-time system is to schedule tasks according to their deadlines in order to guarantee that all real-time activities achieve the required service level. Many scheduling algorithms exist for a variety of task models, but fundamental to many of these are the earliest deadline first (EDF) and rate-monotonic (RM) scheduling policies. A schedule for a set of tasks is said to be feasible if a proof exists that every task instance in the set will complete processing by its associated deadline. Also, a task set is schedulable if there exists a feasible schedule for the set. The utilization associated with a given task schedule and resource (i.e. CPU) is the fraction of time that the resource is allocated over the time that the scheduler is active.

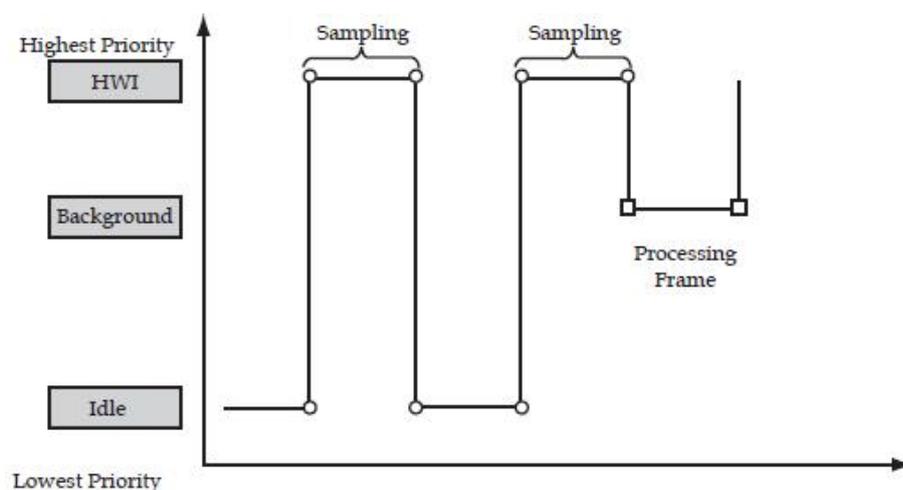


Figure: Real Time Scheduling

7.8 Earliest Deadline First

The EDF scheduling algorithm is a preemptive and dynamic priority scheduler that executes tasks in order of the time remaining before their corresponding deadlines. Tasks with the least time remaining before their deadline are executed before tasks with more remaining time. At each invocation of the scheduler, the remaining time is calculated for each waiting task, and the task with the least remaining time is dispatched. If a task set is schedulable, the EDF algorithm results in a schedule that achieves optimal resource utilization. However, EDF is shown to be unpredictable if the required utilization exceeds 100%, known as an overload condition. EDF is useful for scheduling periodic tasks, since the dynamic priorities of tasks do not depend on the determinism of request periods.

7.9 Rate Monotonic

Under the static-priority rate monotonic scheduling algorithm, tasks are ordered according to the value of their request period, T . Tasks with shorter request periods are assigned higher priority than those with longer periods. Liu and Layland proved that a feasible schedule is found under rate monotonic scheduling if the total requested utilization is less than or equal to \ln , which is approximately 69.3%. RM scheduling is optimal with respect to maximum utilization over all static-priority schedulers. However, this scheduling policy only supports tasks that fit the periodic task model, since priorities depend upon request periods. Because the request times of a periodic tasks are not always predictable, these tasks are not supported by the RM algorithm, but are instead typically scheduled using a dynamic priority scheduler such as EDF.

Characteristics of Scheduling Algorithms

	FCFS	Round Robin	SJF	SRT	HRRN	Feedback
Selection Function	$\max[w]$	Constant	$\text{Min}[s]$	$\text{min}[s-e]$	$\max[(w+s)/s]$	see text
Decision Mode	Non-preemptive	Preemptive	Non-preemptive	preemptive	Non-preemptive	Preemptive at time quantum
Throughput	N/A	low for small quantum	High	High	High	N/A
Response Time	May be high	good for short processes	good for short processes	Good	Good	N/A
Overhead	Minimal	Low	Can be high	Can be high	Can be high	Can be high
Effect on Processes						
Starvation	No	No	Possible	Possible	No	Possible

w = time spent in the system so far, waiting and executing

e = time spent in execution so far.

s = total service time required by the process, including e .

7.10 Operating Systems and Scheduling Types

- 1) **Solaris 2** uses priority-based process scheduling.
- 2) **Windows 2000** uses a priority-based preemptive scheduling algorithm.

- 3) **Linux** provides two separate process-scheduling algorithms: one is designed for timesharing processes for fair preemptive scheduling among multiple processes; the other designed for real-time tasks:
- For processes in the time-sharing class Linux uses a prioritized credit-based algorithm
 - Real-time scheduling: Linux implements two real-time scheduling classes namely FCFS (First come first serve) and RR (Round Robin)

Summary

- Thread is a single sequence stream within in a process. In this method, the kernel knows about and manages the threads.
- Context Switch IIT-VIS has added support for using threads internally in IDL to accelerate specific numerical computations on multi-processor systems.
- The concept of multi-threading involves an operating system that is multi-thread capable allowing programs to split tasks between multiple execution threads.

Keywords

Process Management: Process management is a series of techniques, skills, tools, and methods used to control and manage a business process within a large system or organization.

Threads: A thread is a single sequence stream within in a process.

Process: A process is an instance of a computer program that is being executed. It contains the program code and its current activity. Depending on the operating system (OS).

Kernel: The kernel is the central component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level.

Context Switch: A context switch is the computing process of storing and restoring state (context) of a CPU so that execution can be resumed from the same point at a later time.

Multitasking: Multitasking is the ability of an operating system to execute more than one program simultaneously.

The Cost of Context Switching: Context switching represents a substantial *cost* to the system in terms of CPU time and can, in fact, be the most-costly operation on an operating system.

BIOS: The BIOS software is built into the PC, and is the first code run by a PC when powered on ('boot firmware'). The primary function of the BIOS is to load and start an operating system.

CPU Scheduling: CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.

Scheduling Algorithm: A scheduling algorithm is the method by which threads, processes or dataflows are given access to system resources (e.g. processor time, communications bandwidth).

Self Assessment

- A thread
 - is a lightweight process where the context switching is low.
 - is a lightweight process where the context switching is high.
 - is used to speed up paging.
 - none of the above.
- Process is

- A. program in High level language kept on disk
 - B. contents of main memory
 - C. a program in execution
 - D. a job in secondary memory
3. Fork is
- A. the dispatching of a task
 - B. the creation of a new job
 - C. the creation of a new process
 - D. increasing the priority of a task
4. The components that process data are located in the:
- A. input devices
 - B. output devices
 - C. system unit
 - D. storage component
5. System software is the set of programs that enables your computer's hardware devices and software to work together.
- A. management
 - B. processing
 - C. utility
 - D. application
6. is a method of CPU scheduling that is a preemptive version of shortest job next scheduling.
- A. Shortest remaining time
 - B. Round robin
 - C. Multilevel queue
 - D. First come first serve
7. A scheduling algorithm will simply put the new process at the head of the ready queue.
- A. preemptive
 - B. non-preemptive priority
 - C. preemptive priority
 - D. non-preemptive and non-priority
8. scheduling is essentially concerned with memory management.
- A. Medium term
 - B. Short term
 - C. Long term
 - D. End term

-
5. Which of the following components of program state are shared across threads in a multithreaded process?
 - (a) Register values
 - (b) Heap memory
 - (c) Global variables
 - (d) Stack memory
 6. Can a multi-threaded solution using multiple user-level threads achieve better performance on a multi-processor system than on a single-processor system?
 7. Consider a multi-processor system and a multi-threaded program written using the many-to-many threading model. Let the number of user-level threads in the program be more than the number of processors in the system. Discuss the performance implications of the following scenarios.
 - a) The number of kernel threads allocated to the program is less than the number of processors.
 - b) The number of kernel threads allocated to the program is equal to the number of processors.
 - c) The number of kernel threads allocated to the program is greater than the number of processors but less than the number of user level threads.
 8. Write a multi-threaded Java, Pthreads, or Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.
 9. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?
 10. Discuss how the following pairs of scheduling criteria conflict in certain settings.
 - a) CPU utilization and response time
 - b) Average turnaround time and maximum waiting time
 - c) I/O device utilization and CPU utilization
 11. Which of the following scheduling algorithms could result in starvation?
 - a) First-come, first-served
 - b) Shortest job first
 - c) Round robin
 12. Consider a system implementing multilevel queue scheduling. What strategy can a computer user employ to maximize the amount of CPU time allocated to the user's process?
 13. Explain the differences in the degree to which the following scheduling algorithms discriminate in favor of short processes:
 - a) FCFS
 - b) RR
 - c) Multilevel feedback queues
 14. Using the Windows XP scheduling algorithm, what is the numeric priority of a thread for the following scenarios?
 - a) A thread in the REALTIME PRIORITY CLASS with a relative priority of HIGHEST.
 - b) A thread in the NORMAL PRIORITY CLASS with a relative priority of NORMAL.
 - c) A thread in the HIGH PRIORITY CLASS with a relative priority of ABOVE NORMAL.
 15. Consider the scheduling algorithm in the Solaris operating system for time sharing threads:
 - a) What is the time quantum (in milliseconds) for a thread with priority 10? With priority 55?
 - b) Assume a thread with priority 35 has used its entire time quantum without blocking. What new priority will the scheduler assign this thread?
 - c) Assume a thread with priority 35 blocks for I/O before its time quantum has expired. What new priority will the scheduler assign this thread?



Further Readings

Introduction to Operating Design and Implementation, by Michael Kifer, Scott A. Smolka.

Operating Systems, by Andrew Tanebaum, Albert S. Woodhull.



Web Links

wiley.com/coolege.silberschatz

www.en.wikipedia.org

www.web-source.net

Unit 08: Process Synchronization

CONTENTS

Objectives

Introduction

8.1 Synchronization Process

8.2 Critical Section Problem

8.3 Semaphores

8.4 Types of Semaphore

8.5 What is Mutex?

8.6 Monitors

8.7 Schedule

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- learn the concept of process synchronization
- understand the problems that arose due to multiple process executions
- explore the various categorizations of processes based on synchronization
- understand the critical section problem
- learn the solution to the critical section problem
- identify the conditions that should be satisfied to solve the critical section problem
- understand the race condition
- understand the critical section problem
- identify the need and use of semaphores
- learn the difference between semaphores and mutex
- learn the basic concept of serializability
- understand the different types of schedules
- learn the benefits of serializable schedule

Introduction

Modern operating systems, such as Unix, execute processes concurrently. Although there is a single Central Processor (CPU), which execute the instructions of only one program at a time, the operating system rapidly switches the processor between different processes (usually allowing a single process a few hundred microseconds of CPU time before replacing it with another process.) Some of these resources (such as memory) are simultaneously shared by all processes. Such resources are being used in parallel between all running processes on the system. Other

resources must be used by one process at a time, so must be carefully managed so that all processes get access to the resource. Such resources are being used in concurrently between all running processes on the system. The most important example of a shared resource is the CPU, although most of the I/O devices are also shared. For many of these shared resources the operating system distributes the time a process requires of the resource to ensure reasonable access for all processes. Consider the CPU, the operating system has a clock which sets an alarm every few hundred microseconds. At this time the operating system stops the CPU, saves all the relevant information that is needed to re-start the CPU exactly where it last left off (this will include saving the current instruction being executed, the state of the memory in the CPU's registers, and other data), and removes the process from the use of the CPU. The operating system then selects another process to run, returns the state of the CPU to what it was when it last ran this new process, and starts the CPU again. Let's take a moment to see how the operating system manages this. In this unit, we shall discuss about the deadlock. A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does. It is often seen in a paradox like 'the chicken or the egg'. This situation may be likened to two people who are drawing diagrams, with only one pencil and one ruler between them. If one person takes the pencil and the other takes the ruler, a deadlock occurs when the person with the pencil needs the ruler and the person with the ruler needs the pencil, before he can give up the ruler. Both requests can't be satisfied, so a deadlock occurs.

8.1 Synchronization Process

Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time. Process Synchronization is the task of coordinating the execution of processes in a way that no two processes can have access to the same shared data and resources. Process Scheduling is an OS task that schedules processes of different states like ready, waiting, and running. It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes. In order to synchronize the processes, there are various synchronization mechanisms. Process scheduling allows OS to allocate a time interval of CPU execution for each process. Another important reason for using a process scheduling system is that it keeps the CPU busy all the time. This allows you to get the minimum response time for programs. It is specially needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or data at the same time. Concurrent access to shared data may result in data inconsistency. So, the change made by one process not necessarily reflected when other processes accessed the same shared data. To avoid this type of inconsistency of data, the processes need to be synchronized with each other. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes. It is the task phenomenon of coordinating the execution of processes in such a way that no two processes can have access to the same shared data and resources. It is a procedure that is involved in order to preserve the appropriate order of execution of cooperative processes. In order to synchronize the processes, there are various synchronization mechanisms. Process Synchronization is mainly needed in a multi-process system when multiple processes are running together, and more than one processes try to gain access to the same shared resource or any data at the same time. Process Synchronization was introduced to handle problems that arose while multiple process executions.

Process is categorized into two types on the basis of synchronization and these are given below:

- 1) Independent Process
 - 2) Cooperative Process
- 1) Independent Process - Two processes are said to be independent if the execution of one process does not affect the execution of another process.
 - 2) Cooperative Process - Two processes are said to be cooperative if the execution of one process affects the execution of another process. These processes need to be synchronized so that the order of execution can be guaranteed.

How Process Synchronization Works?

Process synchronization refers to the idea that multiple processes are to join up or handshake at a certain point, so as to reach an agreement or commit to a certain sequence of action. Synchronization involves the orderly sharing of system resources by processes.

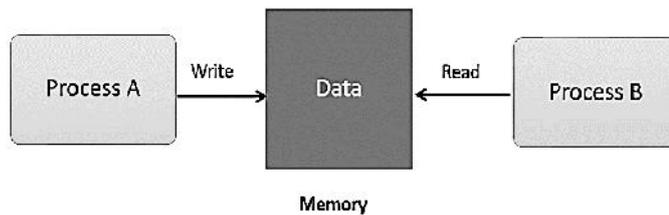


Figure: 8.1 Working of Process Synchronization

As shown in figure 8.1, the process A changing the data in a memory location while another process B is trying to read the data from the same memory location. There is a high probability that data read by the second process will be erroneous. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.

To illustrate the process synchronization, consider the railway intersection scenario in Figure 8.2, we can think of this intersection as a system resource that is shared by two processes: the car process and the train process. If only one process is active, then no resource conflict exists. But what happens when both processes are active and they both arrive at the intersection simultaneously? In this case, the shared resource becomes a problem. They cannot both use the resource at the same time or a collision will occur. Similarly, processes sharing resources on a computer must be properly managed in order to avoid "collisions." This problem can be overcome by making use of a signal as shown in Figure 8.3.

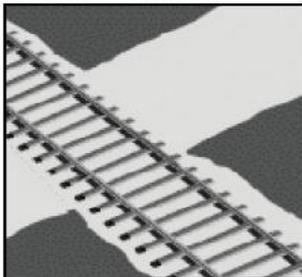


Figure: 8.2 Railway Intersection

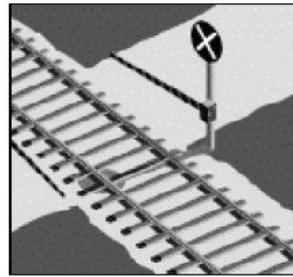


Figure 8.3: Railway-road Intersection with signal

Consider a machine with a single printer running a time-sharing operation system. If a process needs to print its results, it must request that the operating system give it access to the printer's device driver. At this point, the operating system must decide whether to grant this request, depending upon whether the printer is already being used by another process. If it is not, the operating system should grant the request and allow the process to continue; otherwise, the operating system should deny the request and perhaps classify the process as a waiting process until the printer becomes available. Indeed, if two processes were given simultaneous access to the machine's printer, the results would be worthless to both. Now that the problem of synchronization is properly stated, consider the following related definitions:

1. **Critical Resource:** A resource shared with constraints on its use (e.g. memory, files, printers, etc.)
2. **Critical Section:** Code that accesses a critical resource
3. **Mutual Exclusion:** At most one process may be executing a Critical Section with respect to a particular critical resource simultaneously

In the example given above, the printer is the critical resource. Let's suppose that the processes which are sharing this resource are called process A and process B. The critical sections of process A and process B are the sections of the code which issue the print command. In order to ensure that both processes do not attempt to use the printer at the same, they must be granted

mutually exclusive access to the printer driver. The idea of mutual exclusion with our railroad intersection by adding a semaphore to the picture.

Semaphores are used in software systems in much the same way as they are in railway systems. Corresponding to the section of track that can contain only one train at a time is a sequence of instructions that can be executed by only one process at a time. Such a sequence of instructions is called a critical section.

8.2 Critical Section Problem

The critical section is a code segment where the shared variables can be accessed. The key to preventing trouble involving shared storage is find some way to prohibit more than one process from reading and writing the shared data simultaneously. That part of the program where the shared memory is accessed is called the Critical Section.

To understand critical section problem let us assume that there are n processes all competing to use some shared data. Each process has a code segment, called *critical section*, in which the shared data is accessed. An atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections. A diagram that demonstrates the critical section is as follows:

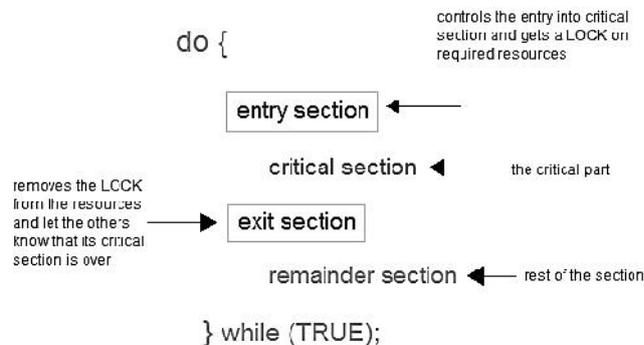


Figure: 8.4 The Critical Section

In the above diagram, the entry section handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that the critical section is free.

The Critical - Section Problem

The problem in the critical section is to ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section. Let us consider an example as in Figure 8.5 to understand the race condition.

To avoid race conditions and flawed results, one must identify codes in Critical Sections in each thread as can be seen in figure 8.4. The characteristic properties of the code that form a Critical Section are:

1. Codes that reference one or more variables in a "read-update-write" fashion while any of those variables is possibly being altered by another thread.
2. Codes that alter one or more variables that are possibly being referenced in "read-update-write" fashion by another thread.
3. Codes use a data structure while any part of it is possibly being altered by another thread.
4. Codes alter any part of a data structure while it is possibly in use by another thread.

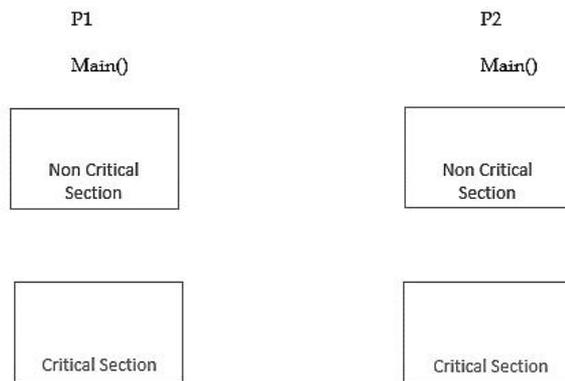


Figure 8.5 Critical Section Problem

Here, the important point is that when one process is executing shared modifiable data in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive in time.

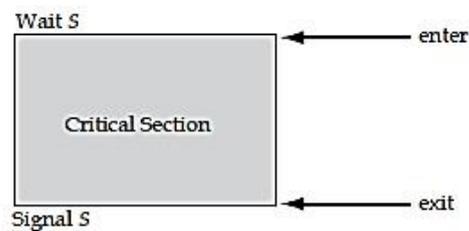


Figure 8.6: Different sections in the Critical Section

Solution of the Critical Section Problem

The critical section problem can be avoided by introducing the entry and the exit section in the program code as shown in figure 8.7.

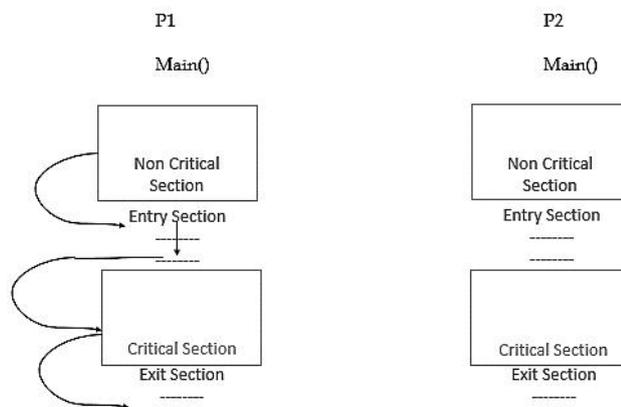


Figure 8.7: Solution to the critical section problem

The critical section problem needs a solution to SYNCHRONIZE the different processes. For synchronization the critical section problem must satisfy the following conditions –

- Mutual Exclusion
- Progress
- Bounded Waiting
- No assumption related to hardware or speed

It should be noted that Mutual Exclusion and Progress are Primary rules whereas bounded waiting and No assumption related to hardware or speed are Secondary Rules respectively.

A way of making sure that if one process is using a shared modifiable data, the other processes will be excluded from doing the same thing. Formally, while one process executes the shared variable, all other processes desiring to do so at the same time moment should be kept waiting; when that process has finished executing the shared variable, one of the processes waiting; while that process has finished executing the shared variable, one of the processes waiting to do so should be allowed to proceed. In this fashion, each process executing the shared data (variables) excludes all others from doing so simultaneously. This is called Mutual Exclusion.



Notes: Mutual exclusion needs to be enforced only when processes access shared modifiable data - when processes are performing operations that do not conflict with one another they should be allowed to proceed concurrently.

Solution: 1 Mutual Exclusion Conditions

If you could arrange matters such that no two processes were ever in their critical sections simultaneously, you could avoid race conditions. You need four conditions to hold to have a good solution for the critical section problem (mutual exclusion).

1. No two processes may at the same moment inside their critical sections.
2. No assumptions are made about relative speeds of processes or number of CPUs.
3. No process outside its critical section should block other processes.
4. No process should wait arbitrary long to enter its critical section.

If process P_i is executing in its critical section, then no other processes can be executing in their critical sections.

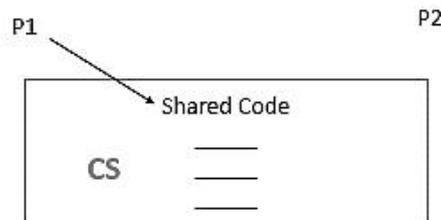


Figure: 8.8 Mutual Exclusion

Proposals for Achieving Mutual Exclusion

The mutual exclusion problem is to devise a pre-protocol (or entry protocol) and a post-protocol (or exist protocol) to keep two or more threads from being in their critical sections at the same time.

Problem: When one process is updating shared modifiable data in its critical section, no other process should be allowed to enter in its critical section.

Proposal 1: Disabling Interrupts (Hardware Solution)

Each process disables all interrupts just after entering in its critical section and re-enables all interrupts just before leaving critical section. With interrupts turned off the CPU could not be switched to other process. Hence, no other process will enter its critical and mutual exclusion achieved.

Conclusion

Disabling interrupts is sometimes a useful technique within the kernel of an operating system, but it is not appropriate as a general mutual exclusion mechanism for user process. The reason is that it is unwise to give user process the power to turn off interrupts.

Proposal 2: Lock Variable (Software Solution)

In this solution, you consider a single, shared, (lock) variable, initially 0. When a process wants to enter in its critical section, it first tests the lock. If lock is 0, the process first sets it to 1 and then enters the critical section. If the lock is already 1, the process just waits until (lock) variable becomes 0. Thus, a 0 means that no process is in its critical section, and 1 means hold your horses - some process is in its critical section.

Conclusion

The flaw in this proposal can be best explained by example. Suppose process A sees that the lock is 0. Before it can set the lock to 1 another process B is scheduled, runs, and sets the lock to 1. When the process A runs again, it will also set the lock to 1, and two processes will be in their critical section simultaneously.

Proposal 3: Strict Alteration

In this proposed solution, the integer variable 'turn' keeps track of whose turn is to enter the critical section. Initially, process A inspects turn, finds it to be 0, and enters in its critical section. Process B also finds it to be 0 and sits in a loop continually testing 'turn' to see when it becomes 1. Continuously testing a variable waiting for some value to appear is called the Busy-Waiting.

Conclusion

Taking turns is not a good idea when one of the processes is much slower than the other. Suppose process 0 finishes its critical section quickly, so both processes are now in their noncritical section. This situation violates above mentioned condition 3.

Using Systems calls 'sleep' and 'wakeup'

Basically, what above mentioned solution do is this: when a process wants to enter in its critical section, it checks to see if then entry is allowed. If it is not, the process goes into tight loop and waits (i.e., start busy waiting) until it is allowed to enter. This approach waste CPU-time. Now look at some inter-process communication primitives is the pair of sleep-wakeup.

Sleep

It is a system call that causes the caller to block, that is, be suspended until some other process wakes it up.

Wakeup

It is a system call that wakes up the process. Both 'sleep' and 'wakeup' system calls have one parameter that represents a memory address used to match up 'sleeps' and 'wakeups'.

Bounded Buffer Producers and Consumers

The bounded buffer producers and consumers assumes that there is a fixed buffer size i.e., a finite numbers of slots are available.

Statement

To suspend the producers when the buffer is full, to suspend the consumers when the buffer is empty, and to make sure that only one process at a time manipulates a buffer so there are no race conditions or lost updates.

As an example, how, sleep-wakeup system calls are used, consider the producer-consumer problem also known as bounded buffer problem.

Two processes share a common, fixed-size (bounded) buffer. The producer puts information into the buffer and the consumer takes information out.

Trouble arises when:

1. The producer wants to put a new data in the buffer, but buffer is already full.

Solution: Producer goes to sleep and to be awakened when the consumer has removed data.

2. The consumer wants to remove data the buffer but buffer is already empty.

Solution: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.

Conclusion: This approach also leads to same race conditions you have seen in earlier approaches. Race condition can occur due to the fact that access to 'count' is unconstrained. The essence of the problem is that a wakeup call, sent to a process that is not sleeping, is lost.

Solution 2: Progress

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder sections can

participate in the decision on which will enter the critical section next, and this selection cannot be postponed indefinitely.

Solution 3: Bounded Waiting

A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Solution 4: No assumption related to Hardware or Speed



For example issues related to processor speed are manageable without having assumptions regarding the hardware or the speed of the processor.

8.3 Semaphores

To understand the concept of semaphores, let us first understand the concept of the 'Race Condition'. A race condition is a situation that may occur inside a critical section. This happens when the result of multiple thread execution in critical section differs according to the order in which the threads execute. Race conditions in critical sections can be avoided if the critical section is treated as an atomic instruction. Also, proper thread synchronization using locks or atomic variables can prevent race conditions. The critical section in a code segment where the shared variables can be accessed. The critical section in a code segment where the shared variables can be accessed. Atomic action is required in a critical section i.e. only one process can execute in its critical section at a time. All the other processes have to wait to execute in their critical sections as shown in figure 8.9. The critical section is given as follows:

```
do{
    Entry Section
    Critical Section
    Exit Section
    Remainder Section
} while (TRUE);
```

In the code, the entry sections handles the entry into the critical section. It acquires the resources needed for execution by the process. The exit section handles the exit from the critical section. It releases the resources and also informs the other processes that critical section is free. A semaphore is a signaling mechanism and a thread that is waiting on a semaphore can be signaled by another thread. This is different than a mutex as the mutex can be signaled only by the thread that called the wait function. A semaphore uses two atomic operations, wait and signal for process synchronization. The wait operation decrements the value of its argument S, if it is positive. If S is negative or zero, then no operation is performed. Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and a thread that is waiting on a semaphore can be signaled by another thread.

E.W. Dijkstra (1965) abstracted the key notion of mutual exclusion in his concepts of semaphores.

Definition

A semaphore is a protected variable whose value can be accessed and altered only by the operations P and V and initialization operation called 'Semaphorinitilize'.

Binary Semaphores can assume only the value 0 or the value 1 counting semaphores also called general semaphores can assume only nonnegative values.

The P (or wait or sleep or down) operation on semaphores S, written as P(S) or wait (S), operates as follows:

```
P(S): IF S > 0
    THEN S := S - 1
    ELSE (wait on S)
```

The V (or signal or wakeup or up) operation on semaphore S, written as V(S) or signal (S), operates as follows:

```
V(S): IF (one or more processes are waiting on S)
      THEN (let one of these processes proceed)
      ELSE S: = S + 1
```

Operations P and V are done as single, indivisible, atomic action. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until operation has completed. Mutual exclusion on the semaphore, S, is enforced within P(S) and V(S). If several processes attempt a P(S) simultaneously, only process will be allowed to proceed. The other processes will be kept waiting, but the implementation of P and V guarantees that processes will not suffer indefinite postponement. Semaphores solve the lost-wakeup problem. Semaphore was proposed by Dijkstra in 1965 which is a very significant technique to manage concurrent processes by using a simple integer value, which is known as a semaphore. It uses two atomic operations,

- 1) wait, and
- 2) signal for the process synchronization.

A semaphore either allows or disallows access to the resource, which depends on how it is set up.

Semaphore Limitations

A few limitations of semaphores are listed below:

- One of the biggest limitations of semaphore is priority inversion.
- Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely. One of the biggest limitations of semaphore is priority inversion.
- The operating system has to keep track of all calls to wait and to signal the semaphore.

Semaphore Implementation Problems

The main problem with semaphores is that they require busy waiting,

- if a process is in the critical section, then other processes trying to enter critical section will be waiting until the critical section is not occupied by any process.
- Whenever any process waits then it continuously checks for semaphore value (look at this line while (s==0); in P operation) and waste CPU cycle.
- There is also a chance of "spinlock" as the processes keep on spins while waiting for the lock.

Characteristics of Semaphore

The main characteristics of the semaphores are:

- It is a mechanism that can be used to provide synchronization of tasks.
- It is a low-level synchronization mechanism.
- It will always hold a non-negative integer value.
- It can be implemented using test operations and interrupts, which should be executed using file descriptors.

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

The signal operation increments the value of its argument S.

```
signal(S){
```

```

        S++;
    }

```

8.4 Types of Semaphore

Semaphores are of two types:

- 1) **Binary Semaphore**
 - 2) **Counting Semaphore**
- 1) **Binary Semaphore** - This is also known as mutex lock. It can have only two values - 0 and 1. Its value is initialized to 1. It is used to implement the solution of critical section problems with multiple processes.
 - 2) **Counting Semaphore** - Its value can range over an unrestricted domain. It is used to control access to a resource that has multiple instances. Now let us see how it does so.

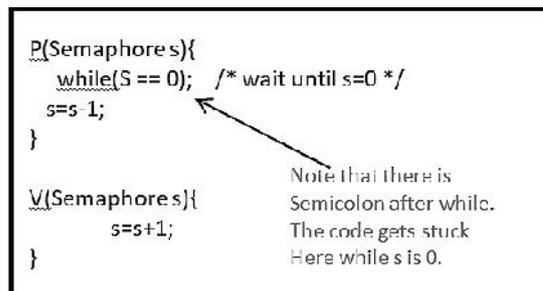


Figure 8.9: Semaphore variable values

First, look at two operations that can be used to access and change the value of the semaphore variable. Let us understand some point regarding P and V operation. P operation is also called wait, sleep, or down operation, and V operation is also called signal, wake-up, or up operation. Both operations are atomic and semaphore(s) is always initialized to one. Here atomic means that variable on which read, modify and update happens at the same time/moment with no pre-emption i.e. in-between read, modify and update no other operation is performed that may change the variable. A critical section is surrounded by both operations to implement process synchronization. As can be seen in the image, the critical section of Process P is in between P and V operation.

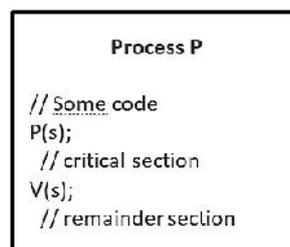


Figure 8.10: The P and V operations of Semaphores

As can be seen in figure 8.10, the critical section of Process P is in between P and V operation. Now, let us see how it implements mutual exclusion. Let there be two processes P1 and P2 and a semaphore s is initialized as 1. Now if suppose P1 enters in its critical section then the value of semaphore s becomes 0. Now if P2 wants to enter its critical section then it will wait until $s > 0$, this can only happen when P1 finishes its critical section and calls V operation on semaphore s.

This way mutual exclusion is achieved.

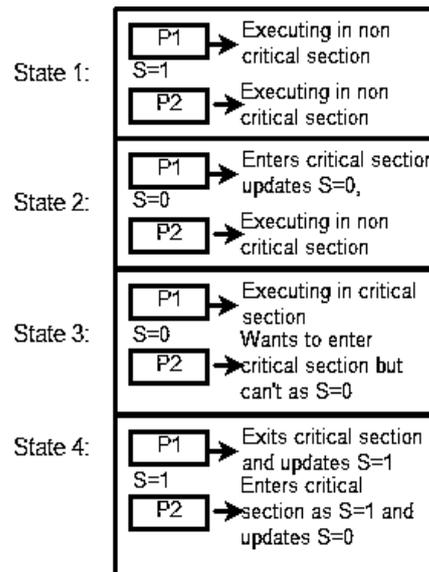


Figure 8.11: Binary Semaphore

Look at the figure 8.11: for details of Binary semaphore. This binary semaphore which can take only two values 0 and 1 and ensure mutual exclusion. There is one other type of semaphore called counting semaphore which can take values greater than one. Now suppose there is a resource whose number of instances is 4. Now we initialize $S = 4$ and the rest is the same as for binary semaphore. Whenever the process wants that resource it calls P or waits for function and when it is done it calls V or signal function. If the value of S becomes zero then a process has to wait until S becomes positive. For example, Suppose there are 4 processes P1, P2, P3, P4, and they all call wait operation on S (initialized with 4). If another process P5 wants the resource then it should wait until one of the four processes calls the signal function and the value of semaphore becomes positive.

Semaphore Limitations:

1. One of the biggest limitations of semaphore is priority inversion.
2. Deadlock, suppose a process is trying to wake up another process which is not in a sleep state. Therefore, a deadlock may block indefinitely.
3. The operating system has to keep track of all calls to wait and to signal the semaphore.

Problem in Implementing Semaphores:

The main problem with semaphores is that they require busy waiting, If a process is in the critical section, then other processes trying to enter critical section will be waiting until the critical section is not occupied by any process. Whenever any process waits then it continuously checks for semaphore value (look at this line while ($s==0$); in P operation) and waste CPU cycle. There is also a chance of "spinlock" as the processes keep on spins while waiting for the lock.

Implementing Counting Semaphores:

In the counting semaphore implementation, whenever the process waits, it is added to a waiting queue of processes associated with that semaphore. This is done through system call `block()` on that process. When a process is completed it calls the signal function and one process in the queue is resumed. It uses `wakeup()` system call.

8.5 What is Mutex?

A mutex is a binary variable whose purpose is to provide locking mechanism. It is used to provide mutual exclusion to a section of code, means only one process can work on a particular code section at a time. Mutex is locking mechanism in OS. There is misconception that binary semaphore is same as mutex variable but both are different in the sense that the binary semaphore apart from providing locking mechanism also provides two atomic operation signal and wait, means after releasing resource semaphore will provide signaling mechanism for the processes who are waiting for the resource.

Difference between Mutex & Semaphore

1. Mutex is used for thread but semaphore is used for process.
2. Mutex works in user space but semaphore work in kernel space.
3. Mutex is locking mechanism ownership method but semaphore is signaling mechanism but not ownership.
4. Thread to thread mutex is used but for process to process locking mechanism, semaphore is used.

Producer-Consumer Problem using Semaphores

The Solution to producer-consumer problem uses three semaphores, namely, full, empty and mutex. The semaphore 'full' is used for counting the number of slots in the buffer that are full. The 'empty' for counting the number of slots that are empty and semaphore 'mutex' to make sure that the producer and consumer do not access modifiable shared section of the buffer simultaneously.

Initialization

1. Set full buffer slots to 0.
i.e., semaphore Full = 0.
2. Set empty buffer slots to N.
i.e., semaphore empty = N.
3. For control access to critical section set mutex to 1.
i.e., semaphore mutex = 1.

```

Producer ( )
WHILE (true)
produce-Item ( );
P (empty);
P (mutex);
enter-Item ( )
V (mutex)
V (full);
Consumer ( )
WHILE (true)
P (full)
P (mutex);
remove-Item ( );
V (mutex);
V (empty);
consume-Item (Item)

```

A semaphore is hardware or a software tag variable whose value indicates the status of a common resource. Its purpose is to lock the resource being used. A process which needs the resource will check the semaphore for determining the status of the resource followed by the decision for proceeding. In multitasking operating systems, the activities are synchronized by using these semaphore techniques.

Semaphore is a mechanism to resolve resources conflicts by tallying resource seekers what is the state of sought resources, achieving a mutual exclusive access to resources. Often semaphore operates as a type of mutual exclusive counters (such as mutexes) where it holds a number of access keys to the resources. Process that seeks the resources must obtain one of those access keys, one of semaphores, before it proceeds further to utilize the resource. If there is no more such a key available to the process, it has to wait for the current resource user to release the key.

A semaphore could have the value 0, indicating that no wakeups were saved, or some positive values if one or more wakeups were pending.

A semaphore s is an integer variable that apart from initialization, is accessed only through two standard atomic operations, wait and signal. These operations were originally, termed p (for wait to test) and v (for signal to increment).

The classical definition of wait in pseudocode is:

```
wait(s)
{
  while(s <= 0)
  ;// no-op
  s--;
}
```

The classical definition of signal in pseudocode is:

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

Modification to the integer value of semaphore in wait and signal operations must be executed individually. That is, when one process modifies the semaphore value no other process can simultaneously modify that same semaphore value.

8.6 Monitors

A monitor is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization. It allows only one process to be active within the monitor at a time. One simple implementation is shown below.

```
monitor monitor_name
{
  // shared variable declarations
  procedure P1 (...) { ... }
  ...
  procedure Pn (...) { ... }
  Initialization code ( ...) { ... }
  ...
}
```

8.7 Schedule

A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.

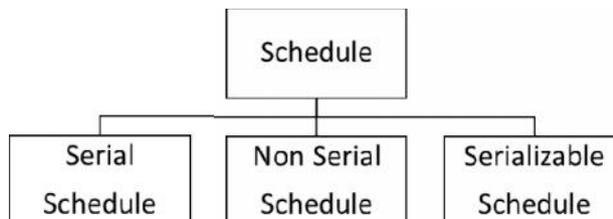


Figure: Different Types of Schedules

- 1) **Serial Schedule** - The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

Example:

Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

- Execute all the operations of T1 which was followed by all the operations of T2.
- Execute all the operations of T2 which was followed by all the operations of T1.

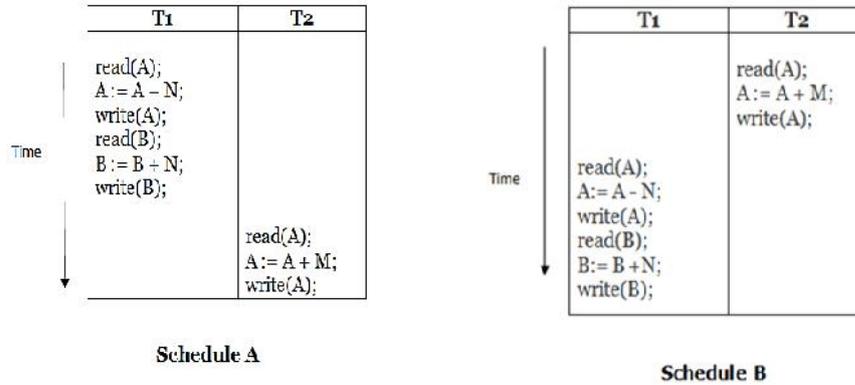


Figure: Serial Schedule Types

In the given figure, Schedule A shows the serial schedule where T1 followed by T2.

- 2) Non-serial Schedule - If interleaving of operations is allowed, then there will be non-serial schedule. It contains many possible orders in which the system can execute the individual operations of the transactions.

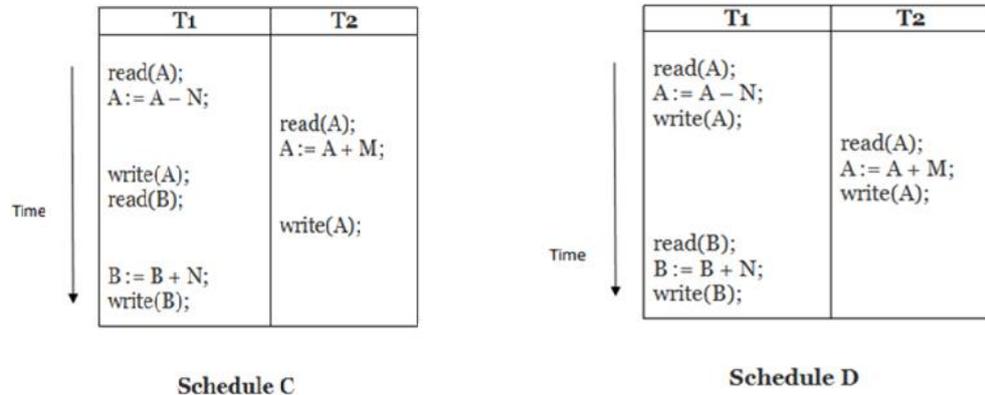


Figure:

In the given figure, Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

- 3) Serializable schedule - The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another. It identifies which schedules are correct when executions of the transaction have interleaving of their operations. A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

Summary

- Race condition is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.
- It may arise in multi-process environment, especially when communicating between separate processes or threads of execution.

- Mutual exclusion means that only one of the processes is allowed to execute its critical section at a time. Mutex, semaphores and monitors are some of the process synchronization tools. Mutex is a software tool used in concurrency control. It is short form of mutual exclusion.
- A mutex is a program element that allows multiple program processes to share the same resource but not simultaneously. Semaphore is a software concurrency control tool. It bears analogy to old Roman system of message transmission using flags. It enforces synchronization among communicating processes and does not require busy waiting.
- In counting semaphore, the integer value can range over an unrestricted domain. In binary semaphore the integer value can range only between 0 and 1.
- A monitor is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization. It allows only one process to be active within the monitor at a time.
- Bounded Buffer Problem, readers and writer's problem, sleeping barber problem, and dining philosopher problem are some of the classical synchronization problems taken from real life situations.

Keywords

Monitor: It is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization.

Mutex: It is a program element that allows multiple program processes to share the same resource but not simultaneously.

Mutex: It is a software tool used in concurrency control. It is short form of mutual exclusion.

Mutual exclusion: It means that only one of the processes is allowed to execute its critical section at a time.

Race condition: It is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.

Semaphore: It is a software concurrency control tool.

Self Assessment

1. Process synchronization facilitates in getting the maximum
2. are used in software systems in much the same way as they are in railway systems.
3. Part of the program where the shared memory is accessed is called the
4. A is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization?
5. To avoid the race condition, the number of processes that may be simultaneously inside their critical section is
6. Algorithms that avoid mutual exclusion are called synchronization algorithms.
7. abstracted the key notion of mutual exclusion in his concepts of semaphores.
8. "No preemption" condition also known as
9. processes share a common, fixed-size (bounded) buffer.
10. Binary Semaphores can assume only the value 0 or the value
11. Process Synchronization coordinates the execution of processes such that no two processes can have access to the same shared data and resources.

- A. True
- B. False

12. Maintaining data consistency requires a mechanism to ensure the orderly execution of cooperating processes.

- A. True
- B. False

13. A critical region is a piece of code which only one process executes at a time

- A. True
- B. False

14. When a process is executing in the critical section, then only one additional process can execute in the critical section.

- A. True
- B. False

15. The critical section is a code segment where the shared variables can be accessed.

- A. True
- B. False

16. Race conditions in critical sections can be avoided if the critical section is treated as non-atomic instruction

- A. True
- B. False

17. The schedules which have interleaving of operations are called serial schedules.

- A. True
- B. False

Answers for Self Assessment

- | | | | | |
|-----------------|-------------------------|---------------------|------------|--------|
| 1. Throughput | 2. Semaphores | 3. Critical Section | 4. Monitor | 5. one |
| 6. non-blocking | 7. E.W. Dijkstra (1965) | 8. lockout | 9. Two | 10. 1 |
| 11. A | 12. A | 13. A | 14. B | 15. A |
| 16. B | 17. B | | | |

Review Questions

1. What is the meaning of the term busy waiting? What other kinds of waiting are there in an operating system? Can busy waiting be avoided altogether? Explain your answer.
2. What do you mean by mutual exclusion conditions? Explain
3. Show that, if the wait and signal operations are not executed atomically, then mutual exclusion may be violated.
4. Demonstrate that monitors, conditional critical regions, and semaphores are all equivalent, insofar as the same types of synchronization problems can be implemented with them. 6.10 Write a bounded-buffer monitor in which the buffers (portions) are embedded within the monitor itself.
5. Consider a system consisting of processes P_1, P_2, P_3 , each of which has a unique priority number. Write a monitor that allocates three identical line printers to these processes, using the priority numbers for deciding the order of allocation.
6. A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint. The sum of all unique numbers associated with all the processes currently accessing the file must be less than n . Write a monitor to coordinate access to the file.



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Colin Ritchie, Operating Systems, BPB Publications.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 09: Deadlocks

CONTENTS

Objectives

Introduction

9.1 Deadlock

9.2 Deadlock Characterization

9.3 Deadlock Avoidance

9.4 Deadlock Detection and Recovery

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the system model
- learn the meaning, and causes of deadlocks
- Identify the various characteristics of deadlocks
- learn the methods of handling deadlocks
- understand the basics of resource allocation graph
- learn the various deadlock prevention techniques.
- understanding how these conditions can be eliminated to prevent deadlocks from occurring.
- learn the deadlock avoidance techniques
- understand the safe and the unsafe deadlock states
- explore the various detection algorithms
- learn the techniques to recover from deadlocks

Introduction

Deadlock is a situation that occurs in OS when any process enters a waiting state because another waiting process is holding the demanded resource. Deadlock is a common problem in multi-processing where several processes share a specific type of mutually exclusive resource known as a soft lock or software. It's important to prevent a deadlock before it can occur. A resource can be released only voluntarily by the process holding it after that process has finished its task. It is better to avoid a deadlock instead of taking an action after the Deadlock has occurred. It needs additional information, like how resources should be used. Deadlock avoidance is the simplest and most useful model that each process declares the maximum number of resources of each type that it may need.

9.1 Deadlock

Deadlock occurs when you have a set of processes [not necessarily all the processes in the system], each holding some resources, each requesting some resources, and none of them is able to obtain what it needs, i.e. to make progress. Those processes are deadlocked because all the processes are waiting. None of them will ever cause any of the events that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, I assume that processes have only a single thread and that there are no interrupts possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by, say, an alarm, and then causing events that release other processes in the set.

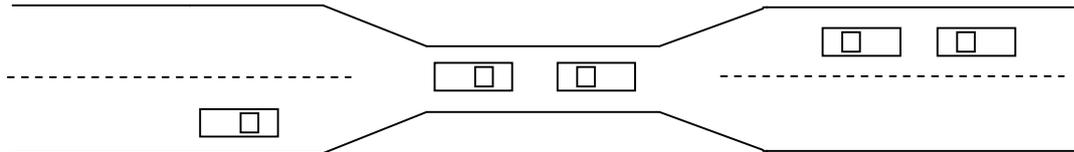


Figure: Bridge Crossing Example

It can be seen in Figure that traffic flows only in one direction. Each section of a bridge can be viewed as a resource. If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback). Several cars may have to be backed up if a deadlock occurs. In this case starvation is possible.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by another deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software.

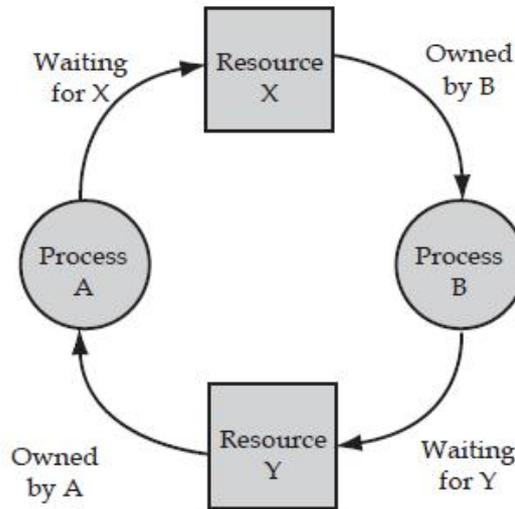


Figure 9.1: Processes are in Deadlock Situation

Understanding the System Model

A system consists of a number of resources (like R1, R2, . . . , Rm.), CPU cycles, memory space and I/O devices. Each resource type Ri has Wi instances. Each process utilizes a resource as a request, use and a release.

A process in operating systems uses different resources and uses resources in the following way.

- 1) Requests a resource
- 2) Use the resource
- 3) Releases the resource

The Deadlock Problem

A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.

For example a system has 2 tape drives. P_1 and P_2 each hold one tape drive and each needs another one. Semaphores A and B , initialized to 1

P_0	P_1
$wait(A);$	$wait(B)$
$wait(B);$	$wait(A)$

9.2 Deadlock Characterization

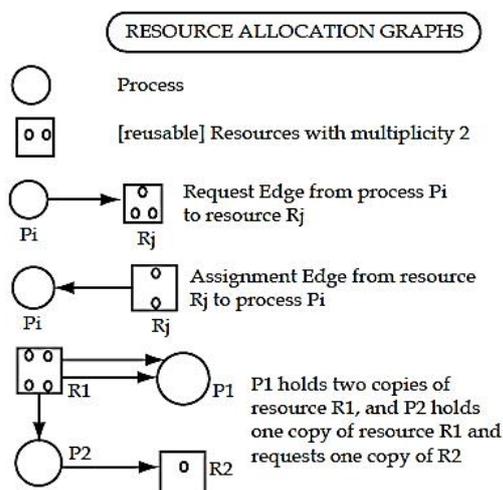
Necessary Conditions

Deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Resources are used in mutual exclusion.
2. Resources are acquired piecemeal (i.e. not all the resources that are needed to complete an activity are obtained at the same time in a single indivisible action).
3. Resources are not preempted (i.e. a process does not take away resources being held by another process).
4. Resources are not spontaneously given up by a process until it has satisfied all its outstanding requests for resources (i.e. a process, being that it cannot obtain some needed resource it does not kindly give up the resources that it is currently holding).

Resource Allocation Graphs

Resource Allocation Graphs (RAGs) are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system.



State transitions can be represented as transitions between the corresponding resource allocation graphs. Here are the rules for state transitions:

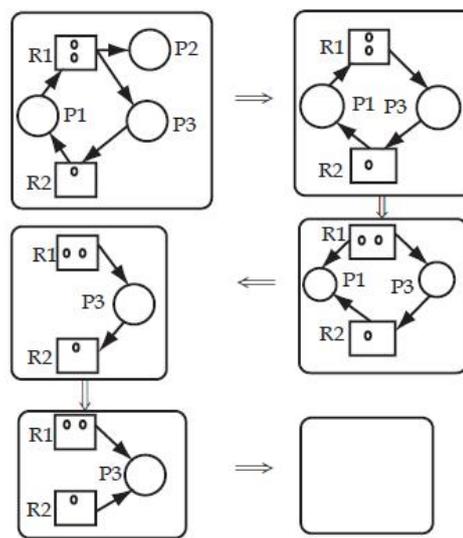
1. **Request:** If process P_i has no outstanding request, it can request simultaneously any number (up to multiplicity) of resources R_1, R_2, \dots, R_m . The request is represented by adding appropriate request edges to the RAG of the current state.
2. **Acquisition:** If process P_i has outstanding requests and they can all be simultaneously satisfied, then the request edges of these requests are replaced by assignment edges in the RAG of the current state.

3. **Release:** If process P_i has no outstanding request then it can release any of the resources it is holding, and remove the corresponding assignment edges from the RAG of the current state.

Here are some important propositions about deadlocks and resource allocation graphs:

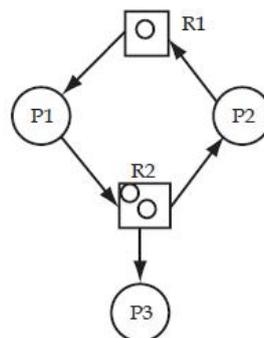
1. If a RAG of a state of a system is fully reducible (i.e. it can be reduced to a graph without any edges using ACQUISITION and RELEASE operations) then that state is not a deadlock state.
2. If a state is not a deadlock state then its RAG is fully reducible [this holds only if you are dealing with reusable resources; it is false if you have consumable resources]
3. A cycle in the RAG of a state is a necessary condition for that being a deadlock state
4. A cycle in the RAG of a state is a sufficient condition for that being a deadlock state only in the case of reusable resources with multiplicity one.

 Example: Here is an example of reduction of a RAG:



Reduction of a RAG

And here is a deadlock-free system with a loop.



RAG with Loop but no Deadlock

Deadlocks can be prevented by ensuring that at least one of the following four conditions occur:

1. **Mutual exclusion:** Removing the mutual exclusion condition means that no process may have exclusive access to a resource. This proves impossible for resources that cannot be spooled, and even with spooled resources deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

2. **Hold and wait:** The “hold and wait” conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations); this advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to release all their resources before requesting all the resources they will need. This too is often impractical. (Such algorithms, such as serializing tokens, are known as the all-or-none algorithms.)
3. **No preemption:** A “no preemption” (lockout) condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm.



Notes: Preemption of a “locked out” resource generally implies a rollback, and is to be avoided, since it is very costly in overhead.

Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control.

4. **Circular wait:** The circular wait condition: Algorithms that avoid circular waits include “disable interrupts during critical sections”, and “use a hierarchy to determine a partial ordering of resources” (where no obvious hierarchy exists, even the memory address of resources has been used to determine ordering) and Dijkstra’s solution.

9.3 Deadlock Avoidance

Deadlock Avoidance, assuming that you are in a safe state (i.e. a state from which there is a sequence of allocations and releases of resources that allows all processes to terminate) and you are requested certain resources, simulates the allocation of those resources and determines if the resultant state is safe. If it is safe the request is satisfied, otherwise it is delayed until it becomes safe.

The Banker’s Algorithm is used to determine if a request can be satisfied. It uses requires knowledge of who are the competing transactions and what are their resource needs. Deadlock avoidance is essentially not used in distributed systems.

9.4 Deadlock Detection and Recovery

Often neither deadlock avoidance nor deadlock prevention may be used. Instead deadlock detection and recovery are used by employing an algorithm that tracks resource allocation and process states, and rolls back and restarts one or more of the processes in order to remove the deadlock. Detecting a deadlock that has already occurred is easily possible since the resources that each process has locked and/or currently requested are known to the resource scheduler or OS. Detecting the possibility of a deadlock before it occurs is much more difficult and is, in fact, generally undecidable, because the halting problem can be rephrased as a deadlock scenario. However, in specific environments, using specific means of locking resources, deadlock detection may be decidable. In the general case, it is not possible to distinguish between algorithms that are merely waiting for a very unlikely set of circumstances to occur and algorithms that will never finish because of deadlock.

Ignore Deadlock

In the Ostrich Algorithm it is hoped that deadlock doesn’t happen. In general, this is a reasonable strategy. Deadlock is unlikely to occur very often; a system can run for years without deadlock occurring. If the operating system has a deadlock prevention or detection system in place, this will have a negative impact on performance (slow the system down) because whenever a processor thread requests a resource, the system will have to check whether granting this request could cause a potential deadlock situation. If deadlock does occur, it may be necessary to bring the

Operating System

system down, or at least manually kill a number of processes, but even that is not an extreme solution in most situations.

The Banker's Algorithm for Detecting/Preventing Deadlocks*Banker's Algorithm for Single Resource*

This is modeled on the way a small-town banker might deal with customers' lines of credit. In the course of conducting business, our banker would naturally observe that customers rarely draw their credit lines to their limits. This, of course, suggests the idea of extending more credit than the amount the banker actually has in her coffers. Suppose we start with the following situation

Customer	Credit Used	Credit Line
Andy	0	6
Barb	0	5
Marv	0	4
Sue	0	7
Funds Available	10	
Max Commitment		22

Our banker has 10 credits to lend, but a possible liability of 22. Her job is to keep enough in reserve so that ultimately each customer can be satisfied over time: That is, that each customer will be able to access his full credit line, just not all at the same time. Suppose, after a while, the bank's credit line book shows.

Customer	Credit Used	Credit Line
Sue	1	6
Barb	1	5
Marv	2	4
Sue	4	7
Funds Available	2	
Max Commitment		22

Eight credits have been allocated to the various customers; two remain. The question then is: Does a way exist such that each customer can be satisfied? Can each be allowed their maximum credit line in some sequence? We presume that, once a customer has been allocated up to his limit, the banker can delay the others until that customer repays his loan, at which point the credits become available to the remaining customers. If we arrive at a state where no customer can get his maximum because not enough credits remain, then a deadlock could occur, because the first customer to ask to draw his credit to its maximum would be denied, and all would have to wait.

To determine whether such a sequence exists, the banker finds the customer closest to his limit: If the remaining credits will get him to that limit, the banker then assumes that that loan is repaid, and proceeds to the customer next closest to his limit, and so on. If all can be granted a full credit, the condition is safe.

In this case, Marv is closest to his limit: assume his loan is repaid. This frees up 4 credits. After Marv, Barb is closest to her limit (actually, she's tied with Sue, but it makes no difference) and 3 of the 4 freed from Marv could be used to award her maximum. Assume her loan is repaid; we have now freed 6 credits. Sue is next, and her situation is identical to Barb's, so assume her loan is repaid. We have freed enough credits (6) to grant Andy his limit; thus, this state

safe. Suppose, however, that the banker proceeded to award Barb one more credit after the credit book arrived at the state immediately above:

Customer	Credit Used	Credit Line
Andy	1	6
Barb	2	5
Marv	2	4
Sue	4	7
Funds Available	1	
Max Commitment		22

Now it's easy to see that the remaining credit could do no good toward getting anyone to their maximum.

So, to recap, the banker's algorithm looks at each request as it occurs, and tests if granting it will lead to a safe state. If not, the request is delayed. To test for a safe state, the banker checks to see if enough resources will remain after granting the request to satisfy the customer closest to his maximum. If so, that loan is assumed repaid, and the next customer checked, and so on. If all loans can be repaid, then the request leads to a safe state, and can be granted. In this case, we see that if Barb is awarded another credit, Marv, who is closest to his maximum, cannot be awarded enough credits, hence Barb's request can't be granted — it will lead to an unsafe state.

Banker's Algorithm for Multiple Resources

Suppose, for example, we have the following situation, where the first table represents resources assigned, and the second resources still required by five processes, A, B, C, D, and E.

Resources Assigned

Processes	Tapes	Plotters	Printers	Toasters
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0
Total Existing	6	3	4	2
Total Claimed by Processes	5	3	2	2
Remaining Unclaimed	1	0	2	0

Resources Still Needed

Processes	Tapes	Plotters	Printers	Toasters
A	1	1	0	0
B	0	1	1	2

Operating System

C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

The vectors E, P and A represent Existing, Possessed and Available resources respectively:

$$E = (6, 3, 4, 2)$$

$$P = (5, 3, 2, 2)$$

$$A = (1, 0, 2, 0)$$

Notice that

$$A = E - P$$

Now, to state the algorithm more formally, but in essentially the same way as the example with Andy, Barb, Marv and Sue:

1. Look for a row whose unmet needs don't exceed what's available, that is, a row where $P \leq A$; if no such row exists, we are deadlocked because no process can acquire the resources it needs to run to completion. If there's more than one such row, just pick one.
2. Assume that the process chosen in 1 acquires all the resources it needs and runs to completion, thereby releasing its resources. Mark that process as virtually terminated and add its resources to A.
3. Repeat 1 and 2 until all processes are either virtually terminated (safe state), or a deadlock is detected (unsafe state).

Going thru this algorithm with the foregoing data, we see that process D's requirements are smaller than A, so we virtually terminate D and add its resources back into the available pool:

$$E = (6, 3, 4, 2)$$

$$P = (5, 3, 2, 2) - (1, 1, 0, 1) = (4, 2, 2, 1)$$

$$A = (1, 0, 2, 0) + (1, 1, 0, 1) = (2, 1, 2, 1)$$

Now, A's requirements are less than A, so do the same thing with A:

$$P = (4, 2, 2, 1) - (3, 0, 1, 1) = (1, 2, 1, 0)$$

$$A = (2, 1, 2, 1) + (3, 0, 1, 1) = (5, 1, 3, 2)$$

At this point, we see that there are no remaining processes that can't be satisfied from available resources, so the illustrated state is safe.

Summary

- Race condition is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.
- It may arise in multi-process environment, especially when communicating between separate processes or threads of execution.
- Mutual exclusion means that only one of the processes is allowed to execute its critical section at a time. Mutex, semaphores and monitors are some of the process synchronization tools. Mutex is a software tool used in concurrency control. It is short form of mutual exclusion.
- A mutex is a program element that allows multiple program processes to share the same resource but not simultaneously. Semaphore is a software concurrency control tool. It bears analogy to old Roman system of message transmission using flags. It

enforces synchronization among communicating processes and does not require busy waiting.

- In counting semaphore, the integer value can range over an unrestricted domain. In binary semaphore the integer value can range only between 0 and 1.
- A monitor is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization. It allows only one process to be active within the monitor at a time.
- Bounded Buffer Problem, readers and writer's problem, sleeping barber problem, and dining philosopher problem are some of the classical synchronization problems taken from real life situations.
- Deadlock is a state which occurs when there are two or more processes which are in an indefinite waiting condition waiting for an event which in turn is waiting for some resources held by some waiting processes. In other words, a deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.
- The main question here is "How to deal with deadlocks?". There can be three ways by which deadlocks can be dealt with:
 - Using some appropriate protocols that can avoid or prevent deadlocks from occurring. This is only possible by ensuring that the system never enters an unsafe deadlocked stage.
 - Second approach could be letting the system enter a deadlock state and then trying to detect and recover it.
 - Third approach could be ignoring the deadlock condition altogether and simply pretending as if nothing has happened in the system. This does not sound to be a wise approach. However, it is interesting to note that this approach is being used in operating systems like UNIX and Windows.

So, when does a deadlock occur? There are four necessary conditions that should hold simultaneously for a deadlock to occur. These four conditions are mutual exclusion, hold and wait, no preemption and circular wait. In simple words we can say that to prevent deadlocks, at least one of the necessary conditions should never hold.

- Resource Allocation Graphs (RAGs) are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system. There are several ways to address the problem of deadlock in an operating system
- Prevent, Avoid, Detection and recovery and Ignore.

Keywords

Deadlock: A deadlock is a situation wherein two or more competing actions are waiting for the other to finish, and thus neither ever does.

Monitor: It is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization.

Mutex: It is a program element that allows multiple program processes to share the same resource but not simultaneously.

Mutex: It is a software tool used in concurrency control. It is short form of mutual exclusion.

Mutual exclusion: It means that only one of the processes is allowed to execute its critical section at a time.

Race condition: It is a flaw in a system of processes whereby the output of the process is unexpectedly and critically dependent on the sequence of other processes.

Operating System

Resource Allocation Graphs (RAGs): Those are directed labeled graphs used to represent, from the point of view of deadlocks, the current state of a system.

Semaphore: It is a software concurrency control tool.

Self Assessment

1. involves the orderly sharing of system resources by processes.
2. are used in software systems in much the same way as they are in railway systems.
3. Part of the program where the shared memory is accessed is called the
4. A is a software synchronization tool with high-level of abstraction that provides a convenient and effective mechanism for process synchronization?
5. Resource Allocation Graphs (RAGs) are labeled graphs.
6. Algorithms that avoid mutual exclusion are called synchronization algorithms.
7. abstracted the key notion of mutual exclusion in his concepts of semaphores.
8. "No preemption" condition also known as
9. processes share a common, fixed-size (bounded) buffer.
10. A process can be said to be in the deadlock state, if it was waiting for an event that will never occur.
 - A. True
 - B. False
11. A system has 3 processes sharing 4 resources. If each process needs a maximum of 2 units, then deadlocks may occur
 - A. True
 - B. False
12. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.
 - A. True
 - B. False
13. Algorithms that avoid mutual exclusion are called synchronization algorithms.
 - A. Blocking
 - B. non-blocking
 - C. restricting
 - D. non-restricting
14. Which of the following helps us in identifying a deadlock condition?
 - A. Starvation graph
 - B. Resource allocation graph
 - C. Inversion graph
 - D. None of the given choices

15. A process can be said to be in the state, if it was waiting for an event that will never occur.
- safe
 - unsafe
 - Starvation
 - Deadlock

Answers for Self Assessment

1. Synchronization 2. Semaphores 3. Critical Section 4. Monitor 5. directed
6. non-blocking 7. E.W. Dijkstra (1965) 8. lockout 9. Two 10. A
11. B 12. A 13. B 14. B 15. D

Review Questions

- What is a safe state? What is its use in deadlock avoidance?
- Describe briefly any one method of deadlock prevention.
- What is concurrency? Explain with example deadlock and starvation.
- Explain the different deadlock strategies.
- Can a process be allowed to request multiple resources simultaneously in a system where deadlock is avoided? Discuss why or why not.
- How deadlock situation is avoided and prevented so that no systems are locked by deadlock?
- Consider the following resource allocation situation:

Process P = {P1, P2, P3, P4, P5}

Resources R = {R1, R2, R3}

Allocation E = {P1@R1, P1@R2, P2@R2, P3@R2, P4@R3, P5@R2, R2@P4, R3@P1}

Resource instances n(R1) =3, n(R2) =4, n(R3) =1

Draw the precedence graph. Determine whether there is a deadlock in the above situation.
- Explain process synchronization process.
- What do you mean by mutual exclusion conditions? Explain
- Write short note on semaphore.



Further Readings

- Andrew M. Lister, Fundamentals of Operating Systems, Wiley.
- Andrew S. Tanenbaum and Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.
- Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.
- Colin Ritchie, Operating Systems, BPB Publications.
- Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

Operating System

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata McGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 10: Memory Management - I

CONTENTS

Objectives

Introduction

10.1 Memory Management

10.2 Multistep Processing of a User Program

10.3 Logical and Physical Address Space

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the significance of main memory and input queue
- analyze the multistep processing of a user program
- learn the concept of logical and physical address space
- describe swapping
- explain segmentation with paging
- know virtual memory
- describe demand paging

Introduction

Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly. When the computer is in normal operation, its memory usually contains the main parts of the operating system and some or all of the application programs and related data that are being used. Memory is often used as a shorter synonym for random access memory (RAM). This kind of memory is located on one or more microchips that are physically close to the microprocessor in the computer. Most desktop and notebook computers sold today include at least 16 megabytes of RAM, and are upgradeable to include more. The more RAM you have, the less frequently the computer has to access instructions and data from the more slowly accessed hard disk form of storage.

Memory is sometimes distinguished from storage, or the physical medium that holds the much larger amounts of data that won't fit into RAM and may not be immediately needed there. Storage devices include hard disks, floppy disks, CD-ROM, and tape backup systems. The terms auxiliary storage, auxiliary memory, and secondary memory have also been used for this kind of data repository. Additional kinds of integrated and quickly accessible memory are read-only memory (ROM), programmable ROM (PROM), erasable programmable ROM (EPROM). These are used to keep special programs and data, such as the basic input/output system, that need to be in the computer all the time.

The memory is a resource that needs to be managed carefully. Most computers have a memory hierarchy, with a small amount of very fast, expensive, volatile cache memory, some number of megabytes of medium-speed, medium-price, volatile main memory (RAM), and hundreds of thousands

of megabytes of slow, cheap, non-volatile disk storage. It is the job of the operating system to coordinate how these memories are used.

10.1 Memory Management

In addition to the responsibility of managing processes, the operating system must efficiently manage the primary memory of the computer. The part of the operating system which handles this responsibility is called the memory manager. Since every process must have some amount of primary memory in order to execute, the performance of the memory manager is crucial to the performance of the entire system. The memory manager is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory. Managing the sharing of primary memory and minimizing memory access time are the basic goals of the memory manager.

Some basic goals of the main memory are:

- Bringing the program in the main memory and placed within a process for it to be run.
- *Input queue* - collection of processes on the disk that are waiting to be brought into memory to run the program.
- User programs go through several steps before being run.

When an operating system manages the computer's memory, there are two broad tasks to be accomplished:

1. Each process must have enough memory in which to execute, and it can neither run into the memory space of another process nor be run into by another process.
2. The different types of memory in the system must be used properly so that each process can run most effectively.

The first task requires the operating system to set up memory boundaries for types of software and for individual applications.



Example: Let's look at an imaginary small system with 1 megabyte (1,000 kilobytes) of RAM. During the boot process, the operating system of our imaginary computer is designed to go to the top of available memory and then "back up" far enough to meet the needs of the operating system itself. Let's say that the operating system needs 300 kilobytes to run. Now, the operating system goes to the bottom of the pool of RAM and starts building up with the various driver software required to control the hardware subsystems of the computer. In our imaginary computer, the drivers take up 200 kilobytes. So, after getting the operating system completely loaded, there are 500 kilobytes remaining for application processes. When applications begin to be loaded into memory, they are loaded in block sizes determined by the operating system. If the block size is 2 kilobytes, then every process that is loaded will be given a chunk of memory that is a multiple of 2 kilobytes in size. Applications will be loaded in these fixed block sizes, with the blocks starting and ending on boundaries established by words of 4 or 8 bytes. These blocks and boundaries help to ensure that applications won't be loaded on top of one another's space by a poorly calculated bit or two. With that ensured, the larger question is what to do when the 500-kilobyte application space is filled. In most computers, it's possible to add memory beyond the original capacity. For example, you might expand RAM from 1 to 2 megabytes. This works fine, but tends to be relatively expensive. It also ignores a fundamental fact of computing - most of the information that an application stores in memory is not being used at any given moment. A processor can only access memory one location at a time, so the vast majority of RAM is unused at any moment. Since disk space is cheap compared to RAM, then moving information in RAM to hard disk can greatly expand RAM space at no cost. This technique is called virtual memory management. Disk storage is only one of the memory types that must be managed by the operating system, and is the slowest. Ranked in order of speed, the types of memory in a computer system are:

1. **High-speed cache:** This is fast, relatively small amounts of memory that are available to the CPU through the fastest connections. Cache controllers predict which pieces of data the CPU will need next and pull it from main memory into high-speed cache to speed up system performance.
2. **Main memory:** This is the RAM that you see measured in megabytes when you buy a computer.

3. **Secondary memory:** This is most often some sort of rotating magnetic storage that keeps applications and data available to be used, and serves as virtual RAM under the control of the operating system.

The operating system must balance the needs of the various processes with the availability of the different types of memory, moving data in blocks (called pages) between available memory as the schedule of processes dictates.

Systems for managing memory can be divided into two categories: the system of moving processes back and forth between main memory and disk during execution (known as swapping and paging) and the process that does not do so (that is, no swapping and ping).

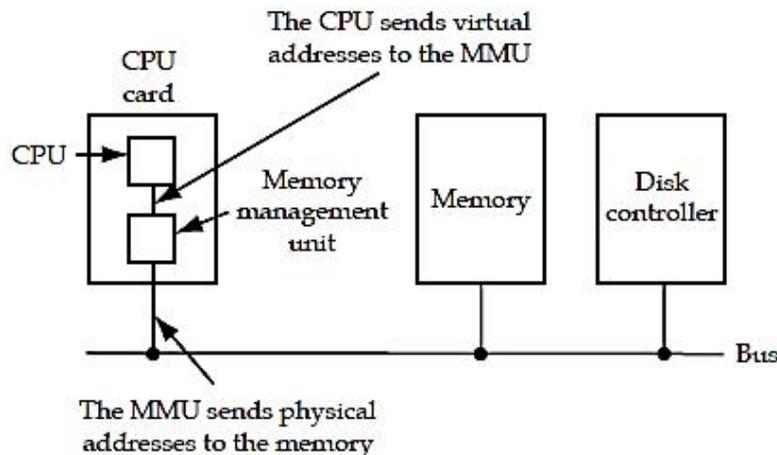


Figure 10.1: Memory Management

10.2 Multistep Processing of a User Program

We can define a program as a sequence of instructions written by the user for instructing the computer to solve a problem. For doing so the program goes through a sequence of steps before it actually gets executed. The program resides on the disc in the form of binary executable file. For running the program, it has to be brought from the disc to the main memory.

It is important or not that CPU you can only access main memory and registers directly during a program execution. The sequence of steps that a program goes through are as follows:

As the program to be executed is brought from the disk to the main memory, it is placed within the context of a process, (which is basically a program under execution) where it becomes available for execution on the CPU. During the execution, it accesses data and instructions from the memory and once the execution is completed, the process is terminated and the memory is reclaimed for use by another process.

The addresses in the source code are generally symbolic (like the variable count). The compiler binds these addresses to relocatable addresses and these are bonded by the linker or loader to the absolute addresses (Binding - mapping from one address space to another).

The binding of instructions and data to the memory addresses can be done at any of the following steps of program execution:

Compile Time: If we initially know where the process is residing in the memory at the time of compilation, then absolute code can be generated. If the starting location changes, then the code has to be recompiled.

Load Time: If the memory address where the process resides is not known at the compile time, then the compiler must generate relocatable code (does not have a static memory address for running). If the starting address changes, then the program must be reloaded to incorporate this value.

Execution Time: If the process can be moved from one segment to another during its execution time, then binding must be delayed till execution time. Special hardware is required for this type of binding

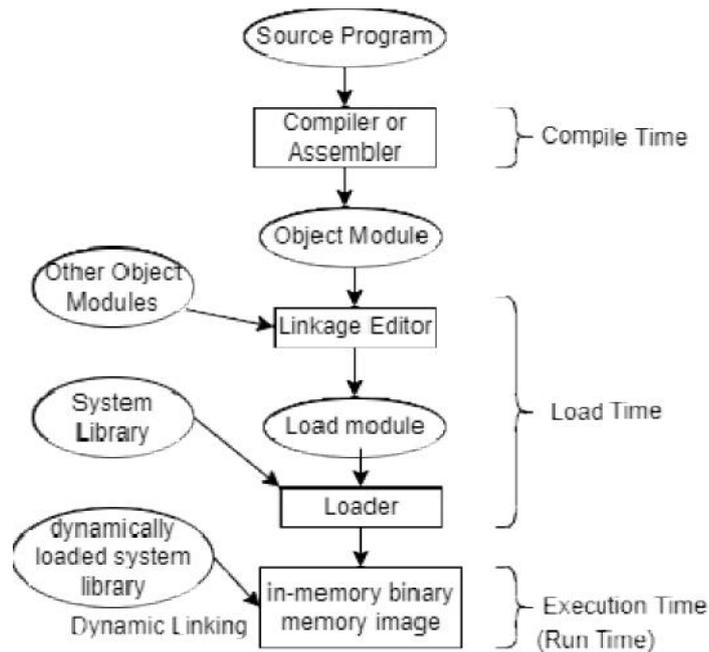
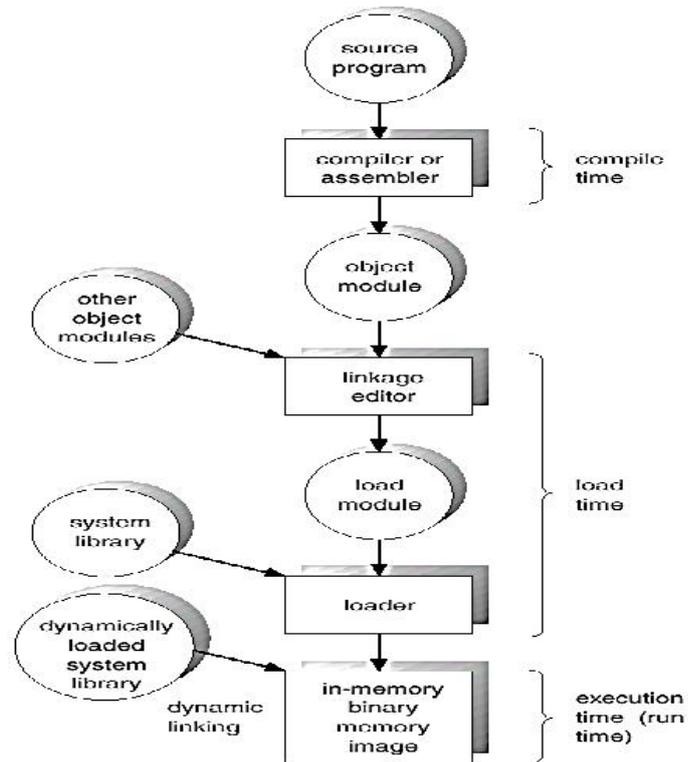
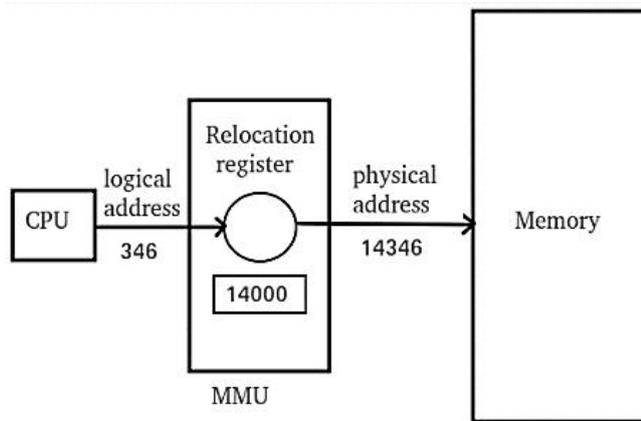


Figure - Multistep Processing of a User Program

10.3 Logical and Physical Address Space

A memory address identifies a physical location in computer memory, somewhat similar to a street address in a town. The address points to the location where data is stored, just like your address points to where you live. In the analogy of a person's address, the address space would be an area of locations, such as a neighborhood, town, city, or country. Two addresses may be numerically the same but refer to different locations, if they belong to different address spaces. This is similar to your address being, say, "32, Main Street", while another person may reside in "32, Main Street" in a different town from yours. Many programmers prefer to use a flat memory model, in which there

is no distinction between code space, data space, and virtual memory – in other words, numerically identical pointers refer to exactly the same byte of RAM in all three address spaces.



Logical Address is generated by CPU while a program is running. The logical address is virtual address as it does not exist physically, therefore, it is also known as Virtual Address. This address is used as a reference to access the physical memory location by CPU.

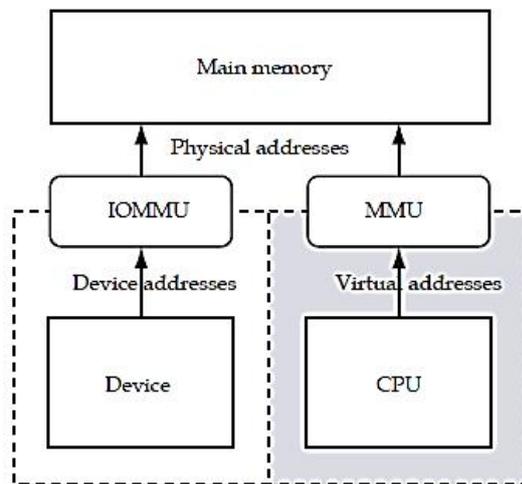


Figure 10.2: Memory Management Unit

Physical Address

A physical address, also real address, or binary address, is the memory address, that is electronically (in the form of binary number) presented on the computer address bus circuitry in order to enable the data bus to access a particular storage cell of main memory.

Logical Address

Logical address is the address at which a memory location appears to reside from the perspective of an executing application program. This may be different from the physical address due to the operation of a Memory Management Unit (MMU) between the CPU and the memory bus.

Physical memory may be mapped to different logical addresses for various purposes.

Example: The same physical memory may appear at two logical addresses and if accessed by the program at one address, data will pass through the processor cache whereas if it is accessed at the other address, it will bypass the cache.

In a system supporting virtual memory, there may actually not be any physical memory mapped to a logical address until an access is attempted. The access triggers special functions of the operating system which reprogram the MMU to map the address to some physical memory, perhaps writing the old contents of that memory to disk and reading back from disk what the memory should contain at the new logical address. In this case, the logical address may be referred to as a virtual address.

Logical vs. Physical Address Space

An address generated by the CPU is commonly referred to as a logical address, whereas an address seen by the memory unit – that is, the one loaded into the memory-address register of the memory – is commonly referred to as a physical address.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, you usually refer to the logical address as a virtual address. The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-address space. Thus, in the execution-time address-binding scheme, the logical- and physical-address spaces differ.

The logical address is referred to as virtual address in this case. At the run time, a hardware device called Memory Management Unit (MMU) does the mapping from virtual address to physical address.

Dynamic Loading is used for more efficient memory utilization. The advantage of Dynamic Loading is that a particular routine is loaded only when it is required. Dynamically Linked Libraries (DLLs) are the system libraries that are linked to the program when the programs are run. This is the overall brief summary of the multistep processing of a user program, right from the source code stage till the execution. The attached diagram depicts the multistep processing of the program precisely.

Summary

- The part of the operating system that manages the memory hierarchy is the memory manager.
- It keeps track of parts of memory that are in use and those that are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.

Keywords

- **Logical Address:** An address generated by the CPU is commonly referred to as a logical address.
- **Memory Management Unit (MMU):** It is a computer hardware component responsible for handling accesses to memory requested by the CPU.
- **Memory Manager:** The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- **Memory:** It is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- **Physical Address:** A physical address, or binary address, is the memory address, that is electronically presented on the computer address bus circuitry.

Self Assessment

1. The principal of locality of reference justifies the use of _____

2. _____ is an address generated by the CPU during execution
3. In address binding, the logical address undergoes translation by the _____
4. The process of bringing the program from the secondary memory to the main memory is known as _____
5. _____ loads the entire program into main memory before start of the program execution
6. In _____ linking load time might be reduced if the shared library code is already present in memory.
7. In the Physical Address Space, the Physical Address identifies a _____ of required data in a memory.
8. Static loading loads the program into the main memory when required.
- A. True
B. False
9. Dynamic loading leads to slow program execution.
- A. True
B. False
10. In Physical Address Space, the user directly deals with the physical address but can access by its corresponding logical address.
- A. True
B. False
11. In the concept of locality of reference, the page reference being made by a process _____.
- A. will always be to the page used in the previous page reference.
B. is likely to be, to one of the pages used in the last few page references.
C. will always be to one of the pages existing in memory.
D. will always lead to a Page fault.
12. The load time might be reduced in case of dynamic linking if _____
- A. The shared library code is already present in memory.
B. If we recompile the program
C. If we relink the program
D. If we recompile and relink again
13. Primary memory stores _____
- A. Data alone
B. Programs alone
C. Results alone
D. All of these
14. Memory is made up of _____
- A. Set of wires

- B. Set of circuits
- C. Large number of cells
- D. All of these

15. The principal of locality of reference justifies the use of

- A. re-enterable
- B. non-reusable
- C. virtual memory
- D. cache memory

Answers for Self Assessment

- | | | | | |
|--------------------|-------------------------|-----------------------------------|------------|----------------------|
| 1. cache
memory | 2. Logical
address | 3. address
translation
unit | 4. Loading | 5. Static
loading |
| 6. Dynamic | 7. physical
location | 8. B | 9. A | 10. B |
| 11. B | 12. A | 13. D | 14. D | 15. D |

Review Questions

1. Write a short description on:
 - (a) Binding of Instructions and Data to Memory
 - (b) Memory-Management Unit
 - (c) CPU utilization
 - (d) Memory Relocation
2. What is high-speed cache?
3. What is overlaying? Explain it.
4. Consider a logical address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.
5. How many bits are there in the logical address?
6. How many bits are there in the physical address?
7. Why are segmentation and paging sometimes combined into one scheme?
8. Describe a mechanism by which one segment could belong to the address space of two different processes.



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

Colin Ritchie, Operating Systems, BPB Publications.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 11: Memory Management - II

CONTENTS

Objectives

Introduction

11.1 Overlays

11.2 Swapping

11.3 Contiguous Memory Allocation

11.4 Paging

11.5 Segmentation

11.6 Segmentation with Paging

11.7 Virtual Memory

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Define memory management
- Describe swapping
- Explain segmentation with paging
- Know virtual memory
- Describe demand paging

Introduction

Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly. Operating systems at times have to run programs bigger in size than the physical memory. To deal with this it generally divides the program into modules such that not all modules need to be in the memory at the same time. For this we need to have a fixed partition size. Which in itself has a problem because with this approach the process has to be limited by the maximum partition size. To deal with this problem the concept of overlays is used. Video release whenever a process is running it will not use the complete program at the same time, but it will use only some part of it. So whenever a part is required, it will be loaded in the memory and whenever that part is done they will simply be unloaded or pulled back from the memory. This process of bringing the portion of the program in the main memory when required and removing it from the main memory once it has executed is called swapping. Swapping involves two tasks called swapping in and swapping out. Swapping-In is the task of placing the pages or blocks of data from the hard disk to the main memory. Swapping out is the task of removing pages or blocks of data from main memory to the hard disk.

11.1 Overlays

Overlay is a technique to run a program that is bigger than the size of the physical memory by keeping only those instructions and data that are needed at any given time. Divide the program into modules in such a way that not all modules need to be in the memory at the same time. The main problem in Fixed partitioning is the size of a process has to be limited by the maximum size of the partition. Overlay is a solution for such problems. The concept of overlays is that whenever a process is running it will not use the complete program at the same time, it will use only some part of it. Then overlays concept says that whatever part you required, you load it on and once the part is done, then you just unload it, means just pull it back and get the new part you required and run it. Formally, "The process of transferring a block of program code or other data into internal memory, replacing what is already stored". Sometimes it happens that compare to the size of the biggest partition, the size of the program will be even more, then, in that case, you should go with overlays.

11.2 Swapping

Any operating system has a fixed amount of physical memory available. Usually, application need more than the physical memory installed on your system, for that purpose the operating system uses a swap mechanism: instead of storing data in physical memory, it uses a disk file. Swapping is the act of moving processes between memory and a backing store. This is done to free up available memory. Swapping is necessary when there are more processes than available memory. At the coarsest level, swapping is done a process at a time.

To move a program from fast-access memory to a slow-access memory is known as "swap out", and the reverse operation is known as "swap in". The term often refers specifically to the use of a hard disk (or a swap file) as virtual memory or "swap space". When a program is to be executed, possibly as determined by a scheduler, it is swapped into core for processing; when it can no longer continue executing for some reason, or the scheduler decides its time slice has expired, it is swapped out again.

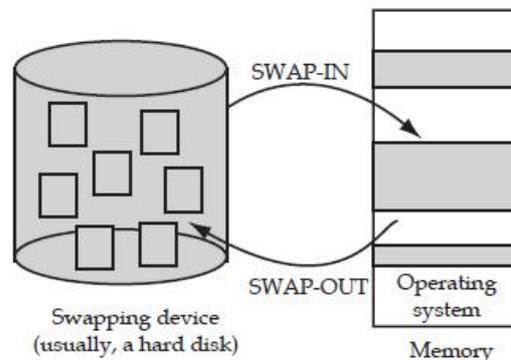


Figure 11.1: Memory Swapping

11.3 Contiguous Memory Allocation

The real challenge of efficiently managing memory is seen in the case of a system which has multiple processes running at the same time. Since primary memory can be space-multiplexed, the memory manager can allocate a portion of primary memory to each process for its own use. However, the memory manager must keep track of which processes are running in which memory locations, and it must also determine how to allocate and deallocate available memory when new processes are created and when old processes complete execution. While various different strategies are used to allocate space to processes competing for memory, three of the most popular are Best fit, Worst fit, and First fit. Each of these strategies is described below:

1. **Best fit:** The allocator places a process in the smallest block of unallocated memory in which it will fit. For example, suppose a process requests 12KB of memory and the memory manager currently has a list of unallocated blocks of 6KB, 14KB, 19KB, 11KB, and 13KB blocks. The best-fit strategy will allocate 12KB of the 13KB block to the process.

2. **Worst fit:** The memory manager places a process in the largest block of unallocated memory available. The idea is that this placement will create the largest hold after the allocations, thus increasing the possibility that, compared to best fit, another process can use the remaining space. Using the same example as above, worst fit will allocate 12KB of the 19KB block to the process, leaving a 7KB block for future use.

3. **First fit:** There may be many holes in the memory, so the operating system, to reduce the amount of time it spends analyzing the available spaces, begins at the start of primary memory and allocates memory from the first hole it encounters large enough to satisfy the request. Using the same example as above, first fit will allocate 12KB of the 14KB block to the process.

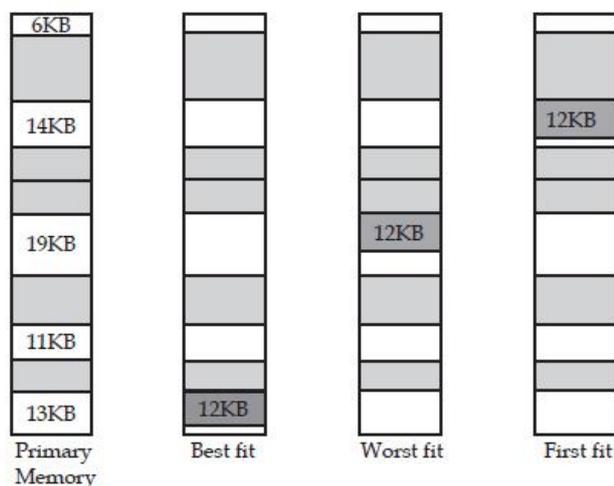


Figure 11.2: Best Fit, Worst Fit and First Fit Memory Allocation Method

Notice in the above figure that the Best fit and First fit strategies both leave a tiny segment of memory unallocated just beyond the new process. Since the amount of memory is small, it is not likely that any new processes can be loaded here. This condition of splitting primary memory into segments as the memory is allocated and deallocated is known as fragmentation. The Worst fit strategy attempts to reduce the problem of fragmentation by allocating the largest fragments to new processes. Thus, a larger amount of space will be left as seen in the Figure 11.2.

Buddy System

Memory management, especially memory allocation to processes, is a fundamental issue in operating systems. A fixed partitioning scheme limits the number of active processes and may use space inefficiently if there is a poor match between available partition sizes and process sizes. A dynamic partitioning scheme is more complex to maintain and includes the overhead of compaction. An interesting compromise is the buddy system.

In a buddy system, the entire memory space available for allocation is initially treated as a single block whose size is a power of 2. When the first request is made, if its size is greater than half of the initial block then the entire block is allocated. Otherwise, the block is split in two equal companion buddies. If the size of the request is greater than half of one of the buddies, then allocate one to it. Otherwise, one of the buddies is split in half again. This method continues until the smallest block greater than or equal to the size of the request is found and allocated to it. In this method, when a process terminates the buddy block that was allocated to it is freed.

Whenever possible, an unallocated buddy is merged with a companion buddy in order to form a larger free block. Two blocks are said to be companion buddies if they resulted from the split of the same direct parent block. The following Figure 11.3 illustrates the buddy system at work, considering a 1024k (1-megabyte) initial block and the process requests as shown at the left of the table.

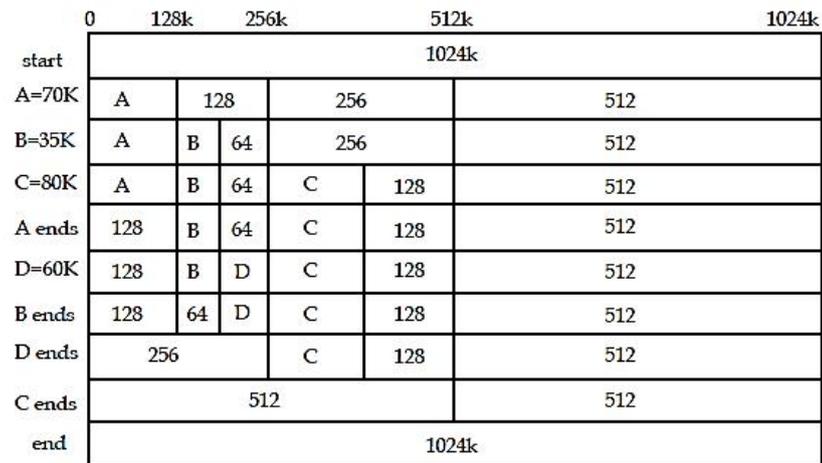


Figure 11.3: Diagram of Buddy System

Notice that, whenever there is a request that corresponds to a block of sizes, your program should select the block of that size that was most recently declared free. Furthermore, when a block is split in two, the left-one (lower addresses) should be selected before the right-one. You can assume that the list of requests is such that all requests can always be served. In other words, you can make the following assumptions: no process will request more than the available memory; processes are uniquely identified while active; and no request for process termination is issued before its corresponding request for memory allocation.

It is preferable when dealing with large amounts of memory to use physically contiguous pages in memory both for cache related and memory access latency reasons. Unfortunately, due to external fragmentation problems with the buddy allocator, this is not always possible.

11.4 Paging

It is a technique for increasing the memory space available by moving infrequently-used parts of a program's working memory from RAM to a secondary storage medium, usually hard disk. The unit of transfer is called a page.

A memory management unit (MMU) monitors accesses to memory and splits each address into a page number (the most significant bits) and an offset within that page (the lower bits). It then looks up the page number in its page table. The page may be marked as paged in or paged out. If it is paged in then the memory access can proceed after translating the virtual address to a physical address. If the requested page is paged out then space must be made for it by paging out some other page, i.e. copying it to disk. The requested page is then located on the area of the disk allocated for "swap space" and is read back into RAM. The page table is updated to indicate that the page is paged in and its physical address recorded.

The MMU also records whether a page has been modified since it was last paged in. If it has not been modified then there is no need to copy it back to disk and the space can be reused immediately.

Paging allows the total memory requirements of all running tasks (possibly just one) to exceed the amount of physical memory, whereas swapping simply allows multiple processes to run concurrently, so long as each process on its own fits within physical memory.

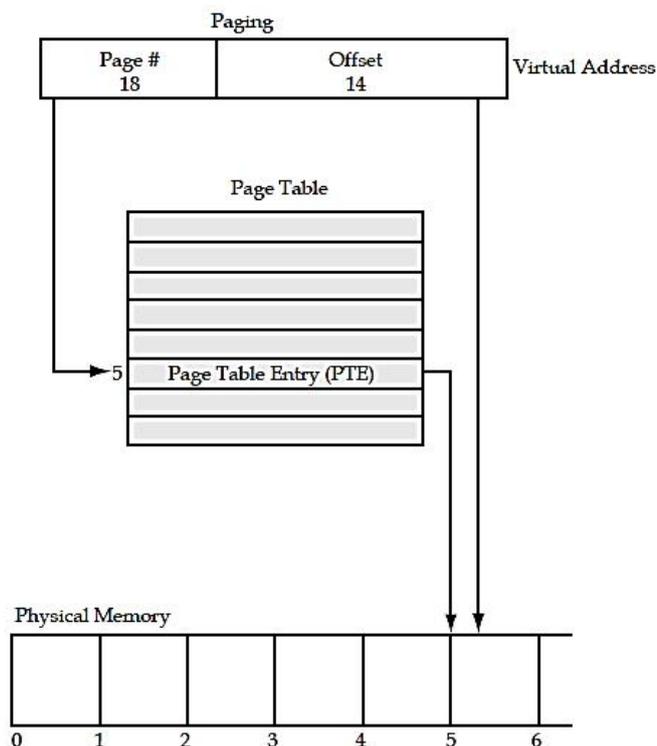


Figure 11.4: Paging

On operating systems, such as Windows NT, Windows 2000 or UNIX, the memory is logically divided in pages. When the system needs a certain portion of memory which is currently in the swap (this is called a page fault) it will load all the corresponding pages into RAM. When a page is not accessed for a long time, it is saved back to disk and discarded.

In a virtual memory system, it is common to map between virtual addresses and physical addresses by means of a data structure called a page table. A page table is the data structure used by a virtual memory system in a computer operating system to store the mapping between virtual addresses and physical addresses. Virtual addresses are those unique to the accessing process. Physical addresses are those unique to the CPU, i.e., RAM. The page number of an address is usually found from the most significant bits of the address; the remaining bits yield the offset of the location within the page. The page table is normally indexed by page number and contains information on whether the page is currently in main memory, and where it is in main memory or on disk.

Conventional page tables are sized to the virtual address space and store the entire virtual address space description of each process. Because of the need to keep the virtual-to-physical translation time low, a conventional page table is structured as a fixed, multi-level hierarchy, and can be very inefficient at representing a sparse virtual address space, unless the allocated pages are carefully aligned to the page table hierarchy.

11.5 Segmentation

It is very common for the size of program modules to change dynamically. For instance, the programmer may have no knowledge of the size of a growing data structure. If a single address space is used, as in the paging form of virtual memory, once the memory is allocated for modules they cannot vary in size. This restriction results in either wastage or shortage of memory. To avoid the above problem, some computer systems are provided with many independent address spaces. Each of these address spaces is called a segment. The address of each segment begins with 0 and segments may be compiled separately. In addition, segments may be protected individually or shared between processes. However, segmentation is not transparent to the programmer like paging. The programmer is involved in establishing and maintaining the segments.

Segmentation is one of the most common ways to achieve memory protection like paging. An instruction operand that refers to a memory location includes a value that identifies a segment and an offset within that segment. A segment has a set of permissions, and a length, associated with it. If the currently running process is allowed by the permissions to make the type of reference to memory that it is attempting to make, and the offset within the segment is within the range

specified by the length of the segment, the reference is permitted; otherwise, a hardware exception is delivered.

In addition to the set of permissions and length, a segment also has associated with its information indicating where the segment is located in memory. It may also have a flag indicating whether the segment is present in main memory or not; if the segment is not present in main memory, an exception is delivered, and the operating system will read the segment into memory from secondary storage. The information indicating where the segment is located in memory might be the address of the first location in the segment, or might be the address of a page table for the segment. In the first case, if a reference to a location within a segment is made, the offset within the segment will be added to address of the first location in the segment to give the address in memory of the referred-to item; in the second case, the offset of the segment is translated to a memory address using the page table. A memory management unit (MMU) is responsible for translating a segment and offset within that segment into a memory address, and for performing checks to make sure the translation can be done and that the reference to that segment and offset is permitted.

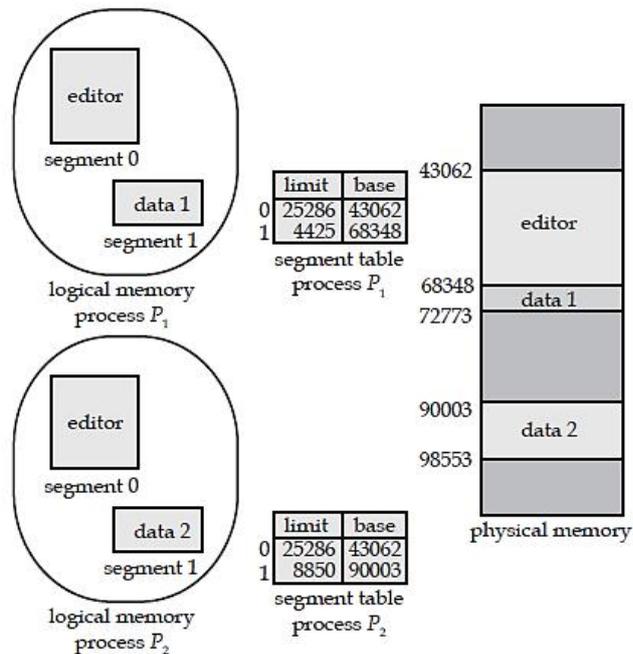


Figure 11.5: Sharing of Segments in a Segmented Memory System

11.6 Segmentation with Paging

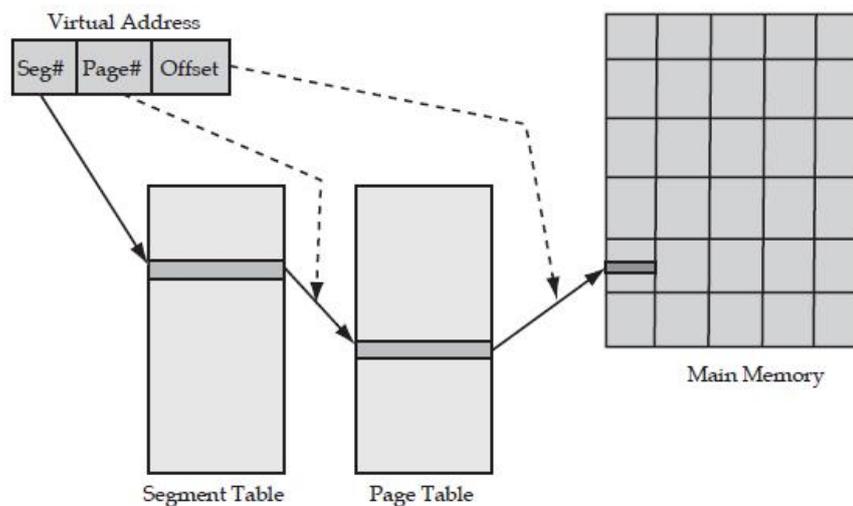


Figure 11.6: Diagram of Segmentation with Paging

Segments can be of different lengths, so it is harder to find a place for a segment in memory than a page. With segmented virtual memory, you get the benefits of virtual memory but you still have to do dynamic storage allocation of physical memory. In order to avoid this, it is possible

to combine segmentation and paging into a two-level virtual memory system. Each segment descriptor points to page table for that segment. This give some of the advantages of paging (easy placement) with some of the advantages of segments (logical division of the program).

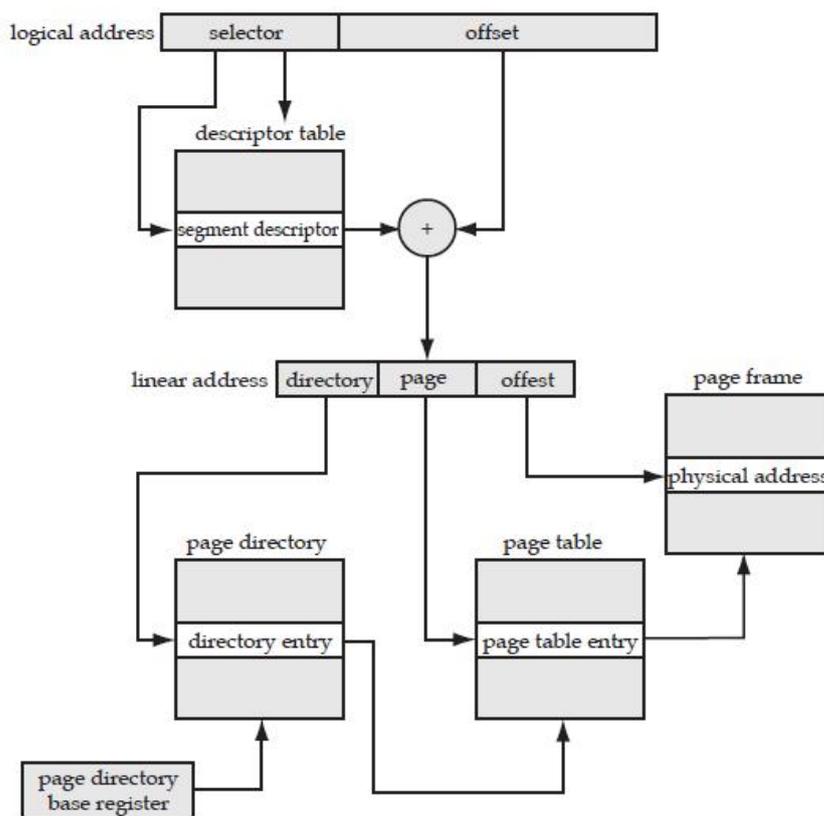


Figure 11.7: Intel 80386 Address Translation

Some operating systems allow for the combination of segmentation with paging. If the size of a segment exceeds the size of main memory, the segment may be divided into equal size pages. The virtual address consists of three parts: (1) segment number (2) the page within the segment and (3) the offset within the page. The segment number is used to find the segment descriptor and the address within the segment is used to find the page frame and the offset within that page.

11.7 Virtual Memory

Many of us use computers on a daily basis. Although you use it for many different purposes in many different ways, you share one common reason of using them; to make our job more efficient and easier. However, there are times when computers cannot run as fast as you want it to or just cannot handle certain processes effectively, due to the shortage of system resources. When the limitations of system resources become a major barrier to achieving your maximum productivity, you often consider the apparent ways of upgrading the system, such as switching to a faster CPU, adding more physical memory (RAM), installing utility programs, and so on. As a computer user, you want to make the most of the resources available; the process of preparing plans to coordinate the total system to operate in the most efficient manner. This is called a system optimization. When it comes to system optimization, there is one great invention of modern computing called virtual memory. It is an imaginary memory area supported by some operating system (for example, Windows but not DOS) in conjunction with the hardware. You can think of virtual memory as an alternate set of memory addresses. Programs use these virtual addresses rather than real addresses to store instructions and data. When the program is actually executed, the virtual addresses are converted into real memory addresses. The purpose of virtual memory is to enlarge the address space, the set of addresses a program can utilize.

Example: Virtual memory might contain twice as many addresses as main memory. A program using all of virtual memory, therefore, would not be able to fit in main memory all at once. Nevertheless, the computer could execute such a program by copying into main memory those portions of the program needed at any given point during execution.

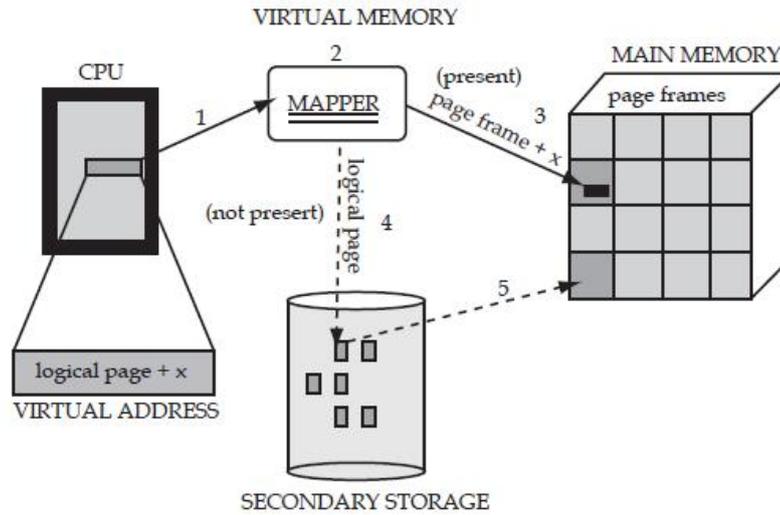


Figure 11.8: Virtual Memory

To facilitate copying virtual memory into real memory, the operating system divides virtual memory into pages, each of which contains a fixed number of addresses. Each page is stored on a disk until it is needed. When the page is needed, the operating system copies it from disk to main memory, translating the virtual addresses into real addresses.

The process of translating virtual addresses into real addresses is called mapping. The copying of virtual pages from disk to main memory is known as paging or swapping.

Some physical memory is used to keep a list of references to the most recently accessed information on an I/O (input/output) device, such as the hard disk. The optimization it provides, is that it is faster to read the information from physical memory, than use the relevant I/O channel to get that information. This is called caching. It is implemented inside the OS.

Demand Paging

As there is much less physical memory than virtual memory the operating system must be careful that it does not use the physical memory inefficiently. One way to save physical memory is to only load virtual pages that are currently being used by the executing program. For example, a database program may be run to query a database. In this case not the entire database needs to be loaded into memory, just those data records that are being examined. Also, if the database query is a search query then it does not make sense to load the code from the database program that deals with adding new records. This technique of only loading virtual pages into memory as they are accessed is known as demand paging.

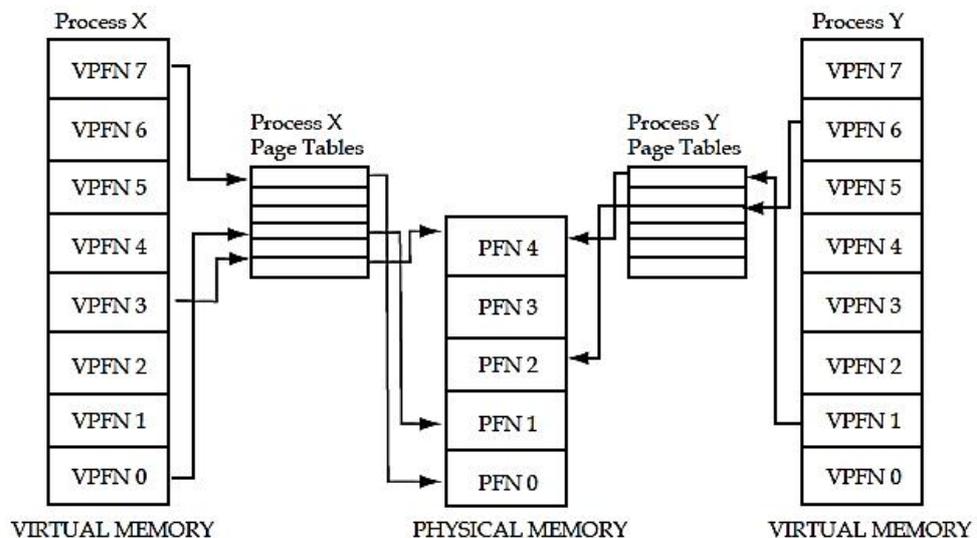


Figure 11.9: Abstract Model of Virtual to Physical Address Mapping

Unit 11: Memory Management - II

When a process attempts to access a virtual address that is not currently in memory the CPU cannot find a page table entry for the virtual page referenced. For example, in Figure 7.11 there is no entry in Process X's page table for virtual PFN 2 and so if Process X attempts to read from an address within virtual PFN 2 the CPU cannot translate the address into a physical one. At this point the CPU cannot cope and needs the operating system to fix things up. It notifies the operating system that a page fault has occurred and the operating system makes the process wait whilst it fixes things up. The CPU must bring the appropriate page into memory from the image on disk. Disk access takes a long time, relatively speaking, and so the process must wait quite a while until the page has been fetched. If there are other processes that could run then the operating system will select one of them to run. The fetched page is written into a free physical page frame and an entry for the virtual PFN is added to the processes page table. The process is then restarted at the point where the memory fault occurred. This time the virtual memory access is made, the CPU can make the address translation and so the process continues to run. This is known as demand paging and occurs when the system is busy but also when an image is first loaded into memory. This mechanism means that a process can execute an image that only partially resides in physical memory at any one time.

Summary

- The part of the operating system that manages the memory hierarchy is the memory manager.
- It keeps track of parts of memory that are in use and those that are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- Overlaying means replacement of a block of stored instructions or data with another. Overlay Manager is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.
- An address generated by the CPU is commonly referred to as a logical address and an address seen by the memory unit – that is, the one loaded into the memory-address register of the memory – is commonly referred to as a physical address.
- Memory Management Unit (MMU) is a computer hardware component responsible for handling accesses to memory requested by the CPU. It is also known as Paged Memory Management Unit (PMMU).

Keywords

- **Logical Address:** An address generated by the CPU is commonly referred to as a logical address.
- **Memory Management Unit (MMU):** It is a computer hardware component responsible for handling accesses to memory requested by the CPU.
- **Memory Manager:** The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- **Memory:** It is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.

- **Overlay Manager:** It is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.
- **Overlaying:** It means replacement of a block of stored instructions or data with another.
- **Paged Memory Management Unit (PMMU):** Same as MMU.
- **Physical Address:** An address seen by the memory unit-that is, the one loaded into the memory address register of the memory-is commonly referred to as a physical address.

Self Assessment

1. In the concept of _____, whenever a process is running, it will not use the complete program at the same time
2. The place in memory where an overlay is loaded is called a _____ region.
3. The method assumes dividing a program into self-contained object code blocks called _____
4. In _____ memory allocation method the memory manager places a process in the largest block of unallocated memory available.
5. Belady's Optimal Algorithm is also known as _____
6. The Best-Fit approach in the Dynamic Storage-Allocation Problem allocate the smallest hole that is big enough.
 - A. True
 - B. False
7. The main problem in Fixed partitioning is that the size of process is independent of the maximum size of the partition.
 - A. True
 - B. False
8. In the concept of overlays, once a part of the program is done with the execution, it is pulled back and a new required part is run.
 - A. True
 - B. False
9. Purpose of the swapping is to access the data present in the hard disk and bring it to RAM.
 - A. True
 - B. False
10. Program counter (PC) contains
 - A. Address of an instruction to be fetched
 - B. Instructions most recently fetched
 - C. Data to be written into memory
 - D. Data to be read from memory
11. To move a program from fast-access memory to a slow-access memory is known as _____

- A. Swap-in
 B. Swap-out
 C. Memory reallocation
 D. None of the given choices
12. The process of translating virtual addresses into real addresses is called
- A. Mapping
 B. Loading
 C. Linking
 D. None of the above
13. happens when a hard drive has to move its heads over the swap area many times due to the high number of page faults.
- A. Thrashing
 B. Spooling
 C. Mapping
 D. None of the given choices
14. The contiguous memory allocation is the one in which_____
- A. every process is contained in a single contiguous section of memory
 B. all processes are contained in a single contiguous section of memory
 C. the memory space is contiguous
 D. none of the mentioned
15. Which of the following is not a partition allocation method used in contiguous memory allocations?
- A. Best Fit
 B. Average Fit
 C. Worst Fit
 D. First Fit

Answers for Self Assessment

1. overlays 2. destination 3. Overlays 4. worst fit 5. perfect prediction
6. A 7. B 8. A 9. A 10. A
11. B 12. A 13. A 14. A 15. B

Review Questions

1. Write a short description on:
- a) Binding of Instructions and Data to Memory
 b) Memory-Management Unit
 c) CPU utilization
 d) Memory Relocation

Operating System

2. What is high-speed cache?
3. What is overlaying? Explain it.
4. Consider a logical address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.
5. How many bits are there in the logical address?
6. How many bits are there in the physical address?
7. Why are segmentation and paging sometimes combined into one scheme?
8. Describe a mechanism by which one segment could belong to the address space of two different processes.
9. Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?
10. Why is it that, on a system with paging, a process cannot access memory that it does not own? How could the operating system allow access to other memory? Why should it or should it not?
11. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
12. What is virtual memory? Explain the working of virtual memory.
13. Describe the dynamic page replacement method.



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

Colin Ritchie, Operating Systems, BPB Publications.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 12: Memory Management - III

CONTENTS

Objectives

Introduction

12.1 Page Replacement

12.2 Page Allocation Algorithm

12.3 Thrashing

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- Define memory management
- Describe swapping
- Explain segmentation with paging
- Know virtual memory
- Describe demand paging

Introduction

Paging is a technique in which the main memory of computer system is organized in the form of equal sized blocks called pages. The address of occupied pages of physical memory are stored inside the Page tables. Paging enables the operating system to obtain data from the physical memory location without specifying lengthy memory address in the instruction. In this technique, the virtual address is used to map the physical address of the data. The length of virtual address is specified in the instruction and is smaller than physical address of the data. It consists of two different numbers; first number is the address of page called virtual page in the page table and the second number is the offset value of the actual data in the page. In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in. A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory. Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

12.1 Page Replacement

When the number of available real memory frames on the free list becomes low, a page stealer is invoked. A page stealer moves through the Page Frame Table (PFT), looking for pages to steal. The PFT includes flags to signal which pages have been referenced and which have been modified. If the page stealer encounters a page that has been referenced, it does not steal that page, but instead,

resets the reference flag for that page. The next time the clock hand (page stealer) passes that page and the reference bit is still off, that page is stolen. A page that was not referenced in the first pass is immediately stolen.

The modify flag indicates that the data on that page has been changed since it was brought into memory. When a page is to be stolen, if the modify flag is set, a page out call is made before stealing the page. Pages that are part of working segments are written to paging space; persistent segments are written to disk.

All paging algorithms function on three basic policies: a fetch policy, a replacement policy, and a placement policy. In the case of static paging, describes the process with a shortcut: the page that has been removed is always replaced by the incoming page; this means that the placement policy is always fixed. Since you are also assuming demand paging, the fetch policy is also a constant; the page fetched is that which has been requested by a page fault. This leaves only the examination of replacement methods.

Static Page Replacement Algorithms

Optimal Replacement Theory

In a best-case scenario, the only pages replaced are those that will either never be needed again, or have the longest number of page requests before they are referenced. This "perfect" scenario is usually used only as a benchmark by which other algorithms can be judged, and is referred to as either Belady's Optimal Algorithm or Perfect Prediction (PP). Such a feat cannot be accomplished without full prior knowledge of the reference stream, or a record of past behavior that is incredibly consistent. Although usually a pipe dream for system designers, suggests it can be seen in very rare cases, such as large weather prediction programs that carry out the same operations on consistently sized data.

In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.



Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Example

7	0	1	2	0	3	0	4	2	3	0	3	2	3
			2	2	2	2	2	2	2	2	2	2	2
		1	1	1	1	1	4	4	4	4	4	4	4
	0	0	0	0	0	0	0	0	0	0	0	0	0
7	7	7	7	7	3	3	3	3	3	3	3	3	3
Miss	Miss	Miss	Miss	Hit	Miss	Hit	Miss	Hit	Hit	Hit	Hit	Hit	Hit

Total Page Fault=6

- Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots (so, 4 Page faults) 0 is already there (so, 0 Page faults)
- When 3 came it will take the place of 7 because it is not used for the longest duration of time in the future. (So, 1 Page fault)
- 0 is already there (so, 0 Page faults)
- 4 will take place of 1 (so, 1 Page Fault).

Now for the further page reference string there are 0 Page faults because they are already available in the memory. Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Random Replacement

On the flip-side of complete optimization is the most basic approach to page replacement: simply choosing the victim, or page to be removed, at random. Each page frame involved has an equal chance of being chosen, without taking into consideration the reference stream or locality principles. Due to its random nature, the behavior of this algorithm is quite obviously, random and unreliable. With most reference streams this method produces an unacceptable number of page faults, as well as victim pages being thrashed unnecessarily. A better performance can almost always be achieved by employing a different algorithm. Most systems stopped experimenting with this method as early as the 1960's.

First-In, First-Out (FIFO)

First-in, first-out is as easy to implement as Random Replacement, and although its performance is equally unreliable or worse, its behavior does follow a predictable pattern. Rather than choosing a victim page at random, the oldest page (or first-in) is the first to be removed.

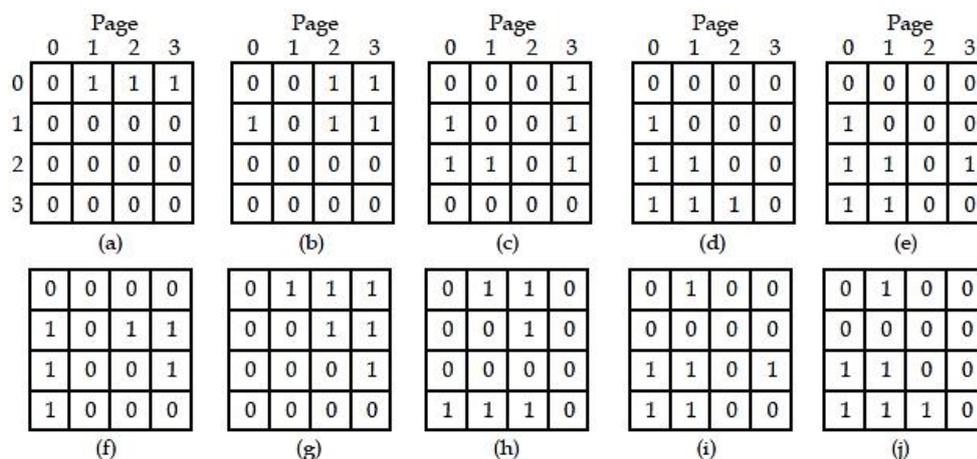


Figure 12.1: LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3

Conceptually compares FIFO to a limited size queue, with items being added to the queue at the tail. When the queue fills (all of the physical memory has been allocated), the first page to enter is pushed out of head of the queue. Similar to Random Replacement, FIFO blatantly ignores trends, and although it produces less page faults, still does not take advantage of locality trends unless by coincidence as pages move along the queue. A modification to FIFO that makes its operation much more useful is First-In Not-Used First-Out (FINUFO). The only modification here is that a single bit is used to identify whether or not a page has been referenced during its time in the FIFO queue. This utility, or referenced bit, is then used to determine if a page is identified as a victim. If, since it has been fetched, the page has been referenced at least once, its bit becomes set. When a page must be swapped out, the first to enter the queue whose bit has not been set is removed; if every active page has been referenced, a likely occurrence taking locality into consideration, all of the bits are reset. In a worst-case scenario this could cause minor and temporary thrashing, but is generally very effective given its low cost.



Consider page reference string 1, 3, 0, 3, 5, 6 with 3-page frames. Find number of page faults.

Example

1	3	0	3	5	6	3
		0	0	0	0	3
	3	3	3	3	6	6
1	1	1	1	5	5	5
Miss	Miss	Miss	Hit	Miss	Miss	Miss

Total Page Faults = 6

Initially all slots are empty, so when 1, 3, 0 came they are allocated to the empty slots. (3 Page Faults).

when 3 comes, it is already in memory so (0 Page Faults).

Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e. 1. (So, 1 Page Fault).

6 comes, it is also not available in memory so it replaces the oldest page slot i.e. 3 (So, 1 Page Fault).

Finally, when 3 come it is not available so it replaces 0 (So, 1page fault)



Note

Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if we consider reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if we increase slots to 4, we get 10 page faults.

Least Frequently Used (LFU)

Often confused with LRU, Least Frequently Used (LFU) selects a page for replacement if it has not been used often in the past. Instead of using a single age as in the case of LRU, LFU defines a frequency of use associated with each page. This frequency is calculated throughout the reference stream, and its value can be calculated in a variety of ways. The most common frequency implementation begins at the beginning of the page reference stream, and continues to calculate the frequency over an ever-increasing interval. Although this is the most accurate representation of the actual frequency of use, it does have some serious drawbacks. Primarily, reactions to locality changes will be extremely slow. Assuming that a program either changes its set of active pages, or terminates and is replaced by a completely different program, the frequency count will cause pages in the new locality to be immediately replaced since their frequency is much less than the pages associated with the previous program. Since the context has changed, and the pages swapped out will most likely be needed again soon (due to the new program's principal of locality), a period of thrashing will likely occur. If the beginning of the reference stream is used, initialization code of a program can also have a profound influence. The pages associated with initial code can influence the page replacement policy long after the main body of the program has begun execution. One way to remedy this is to use a popular variant of LFU, which uses frequency counts of a page since it was last loaded rather than since the beginning of the page reference stream. Each time a page is loaded, its frequency counter is reset rather than being allowed to increase indefinitely throughout the execution of the program. Although this policy will for the most part prevent "old" pages from having a huge influence in the future of the stream, it will still tend to respond slowly to locality changes.

Dynamic Page Replacement Algorithms

All of the static page replacement algorithms considered have one thing in common: they assumed that each program is allocated a fixed amount of memory when it begins execution, and does not request further memory during its lifetime. Although static algorithms will work in this scenario, they are hardly optimized to handle the common occurrence of adjusting to page allocation changes. This can lead to problems when a program rapidly switches between needing relatively large and relatively small page sets or localities. Depending on the size of the memory requirements of a program, the number of page faults may increase or decrease rapidly; for Stack Algorithms, you know that as the memory size is decreased, the number of page faults will increase. Other static algorithms may become completely unpredictable. Generally speaking, any program can have its number of page faults statistically analyzed for a variety of memory allocations. At some point the rate of increase of the page faults (derivative of the curve) will peak; this point is sometimes referred to as the hysteresis point. If the memory allocated to the program is less than the hysteresis point, the program is likely to thrash its page replacement. Past the point, there is generally little noticeable change in the fault rate, making the hysteresis the target page allocation. Since a full analysis is rarely available to a virtual memory controller, and that program behavior is quite dynamic, finding the optimal page allocation can be incredibly difficult. A variety of methods must be employed to develop replacement algorithms that work hand-in-hand with the locality changes present in complex programs. Dynamic paging algorithms accomplish this by attempting to predict program memory requirements, while adjusting available pages based on reoccurring trends. This policy of controlling available pages is also referred to as "prefetch" paging, and is contrary to the idea of demand paging. Although localities (within the scope of a set of operations) may change,

states, it is likely that within the global locality (encompassing the smaller clusters), locality sets will be repeated.

12.2 Page Allocation Algorithm

How do you allocate the fixed amount of free memory among the various processes? If you have 93 free frames and two processes, how many frames does each process get? The simplest case of virtual memory is the single-user system. Consider a single-user system with 128 KB memory composed of pages of size 1 KB. Thus, there are 128 frames. The operating system may take 35 KB, leaving 93 frames for the user process. Under pure demand paging, all 93 frames would initially be put on the free-frame list. When a user process started execution, it would generate a sequence of page faults. The first 93 page faults would all get free frames from the free-frame list. When the free-frame list was exhausted, a page replacement algorithm would be used to select one of the 93 in-memory pages to be replaced with the ninety-fourth, and so on. When the process terminated, the 93 frames would once again be placed on the free-frame list. There are many variations on this simple strategy. You can require that the operating system allocate all its buffer and table space from the free-frame list. When this space is not in use by the operating system, it can be used to support user paging. You can try to keep three free frames reserved on the free-frame list at all times. Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected, which is then written to the disk as the user process continues to execute.

Other variants are also possible, but the basic strategy is clear. The user process is allocated any free frame.

12.3 Thrashing

Thrashing happens when a hard drive has to move its heads over the swap area many times due to the high number of page faults. This happens when memory accesses are causing page faults as the memory is not located in main memory. The thrashing happens as memory pages are swapped out to disk only to be paged in again soon afterwards. Instead of memory access happening mainly in main memory, access is mainly to disk causing the processes to become slow as disk access is required for many memory pages and thus thrashing.

The OS can reduce the effects of thrashing and improve performance by choosing a more suitable replacement strategy for pages. Having a replacement strategy that does not cause memory areas to be written to disk that have not modified since been retrieved reduces thrashing. Using replacement strategies that allow little used rarely accessed pages to remain in memory while the most required pages are swapped in and out.

Thrashing is a situation where large amounts of computer resources are used to do a minimal amount of work, with the system in a continual state of resource contention. Once started, thrashing is typically self-sustaining until something occurs to remove the original situation that led to the initial thrashing behavior. Usually thrashing refers to two or more processes accessing a shared resource repeatedly such that serious system performance degradation occurs because the system is spending a disproportionate amount of time just accessing the shared resource. Resource access time may generally be considered as wasted, since it does not contribute to the advancement of any process. This is often the case when a CPU can process more information than can be held in available RAM; consequently, the system spends more time preparing to execute instructions than actually executing them.

Concept of Thrashing

If the number of frames allocated to a low priority process is lower than the minimum number required by the computer architecture then in this case we must suspend the execution of this low priority process. After this we should page out all of its remaining pages and free all of its allocated frames. This provision introduces a swap in, swap-out level of intermediate CPU scheduling. Let us take an example of a process that does not have enough number of frames. If the process does not have the number of frames it needs to support pages in active use, it will quickly page fault. The only option remains here for process is to replace some active pages with the page that requires a frame. However, since all of its pages are in active use, it must replace a page that

will be needed again right away. Consequently, it quickly faults again and again that mean replacing pages that it must bring back in immediately. This high paging activity is called Thrashing. Or we can say that a process is Thrashing if it is spending more time in paging than executing. Thrashing results in severe performance problems.

Summary

- The part of the operating system that manages the memory hierarchy is the memory manager.
- It keeps track of parts of memory that are in use and those that are not in use, to allocate memory to processes when they need it and de-allocate it when they are done, and to manage swapping between main memory and disk when main memory is too small to hold all the processes.
- Memory is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- Overlaying means replacement of a block of stored instructions or data with another. Overlay Manager is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.
- An address generated by the CPU is commonly referred to as a logical address and an address seen by the memory unit - that is, the one loaded into the memory-address register of the memory - is commonly referred to as a physical address.
- Memory Management Unit (MMU) is a computer hardware component responsible for handling accesses to memory requested by the CPU. It is also known as Paged Memory Management Unit (PMMU).

Keywords

- **Logical Address:** An address generated by the CPU is commonly referred to as a logical address.
- **Memory Management Unit (MMU):** It is a computer hardware component responsible for handling accesses to memory requested by the CPU.
- **Memory Manager:** The memory manager is a part of operating system which is responsible for allocating primary memory to processes and for assisting the programmer in loading and storing the contents of the primary memory.
- **Memory:** It is the electronic holding place for instructions and data that the computer's microprocessor can reach quickly.
- **Overlay Manager:** It is part of the operating system, which loads the required overlay from external memory into its destination region in order to be used.
- **Overlaying:** It means replacement of a block of stored instructions or data with another.
- **Paged Memory Management Unit (PMMU):** Same as MMU.
- **Physical Address:** An address seen by the memory unit - that is, the one loaded into the memory address register of the memory - is commonly referred to as a physical address.

Self Assessment

1. The algorithm in which we split m frames among n processes, to give everyone an equal share, m/n frames is known as _____
2. Thrashing _____ the CPU utilization.
3. A program is generally composed of several different localities, which _____ overlap.
4. A page fault occurs when a page is not available in memory and cannot be accessed.
 - A. True
 - B. False
5. At least a minimum number of frames should be allocated to each process because a lesser number of frames for allocation, leads to a less number of page faults
 - A. True
 - B. False
6. Thrashing leads to a constant state of paging and page faults
 - A. True
 - B. False
7. When a process becomes inactive, its working set cannot migrate to disk.
 - A. True
 - B. False
8. Thrashing
 - A. reduces page I/O
 - B. decreases the degree of multiprogramming
 - C. implies excessive page I/O
 - D. improves the system performance
9. Dirty bit for a page in a page table
 - A. helps avoid unnecessary writes on a paging device
 - B. helps maintain LRU information
 - C. allows only read on a page
 - D. none of the above
10. The page replacement algorithm with the lowest fault rate is _____
 - A. Optimal page replacement algorithm
 - B. LRU replacement algorithm
 - C. FIFO
 - D. Counting based
11. Which of the following algorithms is commonly not used to allocate frames to a process?
 - A. Equal allocation algorithm
 - B. Proportional allocation algorithm
 - C. Final allocation algorithm
 - D. None of the given choices

12. With either equal or proportional algorithm, a high priority process is treated _____ a low priority process.
- greater than
 - same as
 - lesser than
 - none of the mentioned
13. The working set model is used in memory management to implement the concept of ____?
- Thrashing
 - segmentation
 - principle of locality
 - paging
14. in which of the following conditions can we say that a process is thrashing?
- If it spends a lot of time executing, rather than paging
 - If it spends a lot of time paging than executing
 - If it has no memory allocated to it
 - None of the mentioned
15. By locality we mean _____?
- a set of pages that are actively used together
 - a space in memory
 - an area near a set of processes
 - none of the mentioned

Answers for Self Assessment

- | | | | | |
|-------------------------------|--------------|--------|-------|-------|
| 1. equal allocation algorithm | 2. Decreases | 3. May | 4. A | 5. B |
| 6. A | 7. B | 8. C | 9. A | 10. A |
| 11. C | 12. B | 13. C | 14. B | 15. A |

Review Questions

- Write a short description on:
 - Binding of Instructions and Data to Memory
 - Memory-Management Unit
 - CPU utilization
 - Memory Relocation
- What is high-speed cache?
- What is overlaying? Explain it.

4. Consider a logical address space of eight pages of 1,024 words each, mapped onto a physical memory of 32 frames.
5. How many bits are there in the logical address?
6. How many bits are there in the physical address?
7. Why are segmentation and paging sometimes combined into one scheme?
8. Describe a mechanism by which one segment could belong to the address space of two different processes.
9. Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the first-fit, best-fit, and worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?
10. Why is it that, on a system with paging, a process cannot access memory that it does not own? How could the operating system allow access to other memory? Why should it or should it not?
11. What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing,
12. What can the system do to eliminate this problem?
12. What is virtual memory? Explain the working of virtual memory.
13. Describe the dynamic page replacement method.



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

Colin Ritchie, Operating Systems, BPB Publications.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 13: File Management

CONTENTS

Objectives

Introduction

- 13.1 File Systems
- 13.2 Types of File Systems
- 13.3 File Systems and Operating Systems
- 13.4 File Concept
- 13.5 Access Methods
- 13.6 Directory Structure
- 13.7 File System Mounting
- 13.8 File Sharing
- 13.9 Protection
- 13.10 File System Implementation
- 13.11 Allocation Methods
- 13.12 Free-space Management
- 13.13 Directory Implementation

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the goals and principles of protection in a modern computer system
- analyze capability and language-based protection system
- understand the different file systems
- learn the concept of file directories
- understand the different types of file directories
- understand the different file systems
- learn the concept of file directories
- understand the different types of file directories

Introduction

Another part of the operating system is the file manager. While the memory manager is responsible for the maintenance of primary memory, the file manager is responsible for the maintenance of secondary storage (e.g., hard disks).

Each file is a named collection of data stored in a device. The file manager implements this abstraction and provides directories for organizing files. It also provides a spectrum of commands to read and write the contents of a file, to set the file read/write position, to set and use the protection mechanism, to change the ownership, to list files in a directory, and to remove a file. The file manager provides a protection mechanism to allow machine users to administer how processes executing on behalf of different users can access the information in files. File protection is a fundamental property of files because it allows different people to store their information on a shared computer, with the confidence that the information can be kept confidential.

13.1 File Systems

A file system is a method for storing and organizing computer files and the data they contain to make it easy to find and access them. File systems may use a data storage device such as a hard disk or CD-ROM and involve maintaining the physical location of the files, they might provide access to data on a file server by acting as clients for a network protocol (e.g., NFS, SMB, or 9P clients), or they may be virtual and exist only as an access method for virtual data. More formally, a file system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data. File systems share much in common with database technology, but it is debatable whether a file system can be classified as a special-purpose database (DBMS).

13.2 Types of File Systems

File system types can be classified into disk file systems, network file systems and special purpose file systems.

1. Disk file systems:

A disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.

Example: Disk file systems include FAT, FAT32, NTFS, HFS and HFS+, ext2, ext3, ISO 9660, ODS-5, and UDF. Some disk file systems are journaling file systems or versioning file systems.

2. Flash file systems:

A flash file system is a file system designed for storing files on flash memory devices. These are becoming more prevalent as the number of mobile devices is increasing, and the capacity of flash memories catches up with hard drives.

While a block device layer can emulate a disk drive so that a disk file system can be used on a flash device, this is suboptimal for several reasons:

- a) *Erasing blocks:* Flash memory blocks have to be explicitly erased before they can be written to. The time taken to erase blocks can be significant, thus it is beneficial to erase unused blocks while the device is idle.
- b) *Random access:* Disk file systems are optimized to avoid disk seeks whenever possible, due to the high cost of seeking. Flash memory devices impose no seek latency.
- c) *Wear leveling:* Flash memory devices tend to wear out when a single block is repeatedly overwritten; flash file systems are designed to spread out writes evenly.

Log-structured file systems have all the desirable properties for a flash file system. Such file systems include JFFS2 and YAFFS.

3. Database file systems:

A new concept for file management is the concept of a database-based file system. Instead of, or in addition to, hierarchical structured management, files are identified by their characteristics, like type of file, topic, author, or similar metadata. Example: dbfs.

4. **Transactional file systems:**

Each disk operation may involve changes to a number of different files and disk structures. In many cases, these changes are related, meaning that it is important that they all be executed at the same time. Take for example a bank sending another bank some money electronically. The bank's computer will "send" the transfer instruction to the other bank and also update its own records to indicate the transfer has occurred. If for some reason the computer crashes before it has had a chance to update its own records, then on reset, there will be no record of the transfer but the bank will be missing some money.

Transaction processing introduces the guarantee that at any point while it is running, a transaction can either be finished completely or reverted completely (though not necessarily both at any given point). This means that if there is a crash or power failure, after recovery, the stored state will be consistent. (Either the money will be transferred or it will not be transferred, but it won't ever go missing "in transit".)

This type of file system is designed to be fault tolerant, but may incur additional overhead to do so. Journaling file systems are one technique used to introduce transaction-level consistency to file system structures.

5. **Network file systems:**

A network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.



Example: Network file systems include clients for the NFS, SMB protocols, and file-system-like clients for FTP and WebDAV.

6. **Special purpose file systems:** A special purpose file system is basically any file system that is not a disk file system or network file system. This includes systems where the files are arranged dynamically by software, intended for such purposes as communication between computer processes or temporary file space. Special purpose file systems are most commonly used by file-centric operating systems such as Unix. Examples include the `procfs (/proc)` file system used by some Unix variants, which grants access to information about processes and other operating system features.

Deep space science exploration craft, like Voyager I & II used digital tape based special file systems. Most modern space exploration craft like Cassini-Huygens used Real-time operating system file systems or RTOS influenced file systems. The Mars Rovers are one such example of an RTOS file system, important in this case because they are implemented in flash memory.

Task

Discuss NTFS type of file system. Also explain the various benefits of NTFS filesystem over FAT file.

13.3 **File Systems and Operating Systems**

Most operating systems provide a file system, as a file system is an integral part of any modern operating system. Early microcomputer operating systems' only real task was file management - a fact reflected in their names. Some early operating systems had a separate component for handling file systems which was called a disk operating system. On some microcomputers, the disk operating system was loaded separately from the rest of the operating system. On early operating systems, there was usually support for only one, native, unnamed file system; for example, CP/M supports only its own file system, which might be called "CP/M file system" if needed, but which didn't bear any official name at all. Because of this, there needs to be an interface provided by the operating system software between the user and the file system. This interface can be textual (such as provided by a command line interface, such as the Unix shell, or OpenVMS DCL) or graphical (such as provided by a graphical user interface, such as file browsers). If graphical, the metaphor of the folder, containing documents, other files, and nested folders is often used.

Flat file systems: In a flat file system, there are no subdirectories-everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc. While simple, this system rapidly becomes inefficient as the number of files grows, and makes it difficult for users to organize data into related groups.

Like many small systems before it, the original Apple Macintosh featured a flat file system, called Macintosh File System. Its version of Mac OS was unusual in that the file management software (Macintosh Finder) created the illusion of a partially hierarchical filing system on top of MFS. This structure meant that every file on a disk had to have a unique name, even if it appeared to be in a separate folder. MFS was quickly replaced with Hierarchical File System, which supported real directories.

13.4 File Concept

A file is a collection of letters, numbers and special characters: it may be a program, a database, a dissertation, a reading list, a simple letter etc. Sometimes you may import a file from elsewhere, for example from another computer. If you want to enter your own text or data, you will start by creating a file. Whether you copied a file from elsewhere or created your own, you will need to return to it later in order to edit its contents.

The most familiar file systems make use of an underlying data storage device that offers access to an array of fixed-size blocks, sometimes called sector, generally 512 bytes each. The file system software is responsible for organizing these sectors into files and directories, and keeping track of which sectors belong to which file and which are not being used. Most file systems address data in fixed-sized units called "clusters" or "blocks" which contain a certain number of disk sectors (usually 1-64). This is the smallest logical amount of disk space that can be allocated to hold a file.

However, file systems need not make use of a storage device at all. A file system can be used to organize and represent access to any data, whether it be stored or dynamically generated (e.g. from a network connection).

Whether the file system has an underlying storage device or not, file systems typically have directories which associate file names with files, usually by connecting the file name to an index into a file allocation table of some sort, such as the FAT in an MS-DOS file system, or an inode in a Unix-like file system. Directory structures may be flat, or allow hierarchies where directories may contain subdirectories. In some file systems, file names are structured, with special syntax for filename extensions and version numbers. In others, file names are simple strings, and per-file metadata is stored elsewhere.

Other bookkeeping information is typically associated with each file within a file system. The length of the data contained in a file may be stored as the number of blocks allocated for the file or as an exact byte count. The time that the file was last modified may be stored as the file's timestamp. Some file systems also store the file creation time, the time it was last accessed, and the time that the file's meta-data was changed.

Note Many early PC operating systems did not keep track of file times. Other information can include the file's device type (e.g., block, character, socket, subdirectory, etc.), its owner user-ID and group-ID, and its access permission settings (e.g., whether the file is read-only, executable, etc.).

The hierarchical file system was an early research interest of Dennis Ritchie of Unix fame; previous implementations were restricted to only a few levels, notably the IBM implementations, even of their early databases like IMS. After the success of Unix, Ritchie extended the file system concept to every object in his later operating system developments, such as Plan 9 and Inferno.

Traditional file systems offer facilities to create, move and delete both files and directories. They lack facilities to create additional links to a directory (hard links in Unix), rename parent links (".." in Unix-like OS), and create bidirectional links to files.

Traditional file systems also offer facilities to truncate, append to, create, move, delete and in-place modify files. They do not offer facilities to prepend to or truncate from the beginning of a file, let alone arbitrary insertion into or deletion from a file. The operations provided are highly asymmetric and lack the generality to be useful in unexpected contexts.

Example: Inter-process pipes in Unix have to be implemented outside of the file system because the pipes concept does not offer truncation from the beginning of files.

Secure access to basic file system operations can be based on a scheme of access control lists or capabilities. Research has shown access control lists to be difficult to secure properly, which is why research operating systems tend to use capabilities. Commercial file systems still use access control lists.

13.5 Access Methods

There are several ways that the information in the file can be accessed. Some systems provide only one access method for files. On other systems, many different access methods are supported.

Sequential Access

Information in the file is processed in order, one record after the other. This is by far the most common mode of access of files. For example, computer editors usually access files in this fashion. A read operation reads the next portion of the file and automatically advances the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the file and the file pointer. Similarly, a write appends to the end of the end of the file and advances to the end of the newly written material (the new end of file). Such a file can be reset to the beginning, and, on some systems, a program may be able to skip forward or backward n records, for some integer n . This scheme is known as sequential access to a file. Sequential access is based on a tape model of a file.

A sequential file may consist of either formatted or unformatted records. If the records are formatted, you can use formatted I/O statements to operate on them. If the records are unformatted, you must use unformatted I/O statements only. The last record of a sequential file is the end-of-file record.

Direct Access

Direct access is based on a disk model of a file. For direct access, the file is viewed as a numbered sequence of block or records. A direct-access file allows arbitrary blocks to be read or written. Thus, after block 18 has been read, block 57 could be next, and then block 3. There are no restrictions on the order of reading and writing for a direct access file. Direct access files are of great use for intermediate access to large amounts of information.

The file operations must be modified to include the block number as a parameter. Thus, you have "read n ", where n is the block number, rather than "read next", and "write n ", rather than "write next". An alternative approach is to retain "read next" and "write next" and to add an operation; "position file to n " where n is the block number. Then, to affect a "read n ", you would issue the commands "position to n " and then "read next".

Not all OS support both sequential and direct access for files. Some systems allow only sequential file access; others allow only direct access. Some systems require that a file be defined as sequential or direct when it is created; such a file can be accessed only in a manner consistent with its declaration.

Direct-access files support both formatted and unformatted record types. Both formatted and unformatted I/O work exactly as they do for sequential files.

Other Access Methods

Other access methods can be built on top of a direct-access method. These additional methods generally involve the construction of an index for a file. The index contains pointers to the various blocks. To find an entry in the file, the index is searched first and the pointer is then used to access the file directly to find the desired entry. With a large file, the index itself may become too large to be kept in memory. One solution is to create an index for the index file. The primary index file would contain pointers to secondary index files, which would point to the actual data items. For example, IBM's indexed sequential access method (ISAM) uses a small master index that points to disk blocks of a secondary index. The secondary index blocks point to the actual file blocks. The file is kept sorted on a defined key. To find a particular item, I first make a binary search of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally, this block

is searched sequentially. In this way, any record can be located from its key by at most direct access reads.

13.6 Directory Structure

The directories themselves are simply files indexing other files, which may in turn be directories if a hierarchical indexing scheme is used. In order to protect the integrity of the file system in spite of user or program error, all modifications to these particular directory files are commonly restricted to the file management system. The typical contents of a directory are:

- 1) file name (string uniquely identifying the file), type (e.g. text, binary data, executable, library), organization (for systems that support different organizations);
- 2) device (where the file is physically stored), size (in blocks), starting address on device (to be used by the device I/O subsystem to physically locate the file);
- 3) creator, owner, access information (who is allowed to access the file, and what they may do with it);
- 4) date of creation/of last modification;
- 5) locking information (for the system that provide file/record locking).

As far as organization, by far the most common scheme is the hierarchical one: a multi-level indexing scheme is used, in which a top-level directory indexes both files and other directories, which in turn index files and directories, and so on. Usually this scheme is represented in the form of a tree. The hierarchical architecture has distinct advantages over a simple, one-level indexing one:

the tree structure can be effectively used to reflect a logical organization of the data stored in the files; names can be reused (they must uniquely identify files within each directory, not across the whole file system); in a multi-user system, name conflicts between files owned by different users can be solved by assigning to each user a directory for her own files and sub-directories, the so-called user's "home" directory.

A complete indexing of a file is obtained by navigating the tree starting from the top-level, "root", directory, and walking along a path to the tree leaf corresponding to the file. A "pathname" is thus obtained, which uniquely identifies the file within the whole file system.

Example: The pathname for file "File-6" in Figure 8.1 is "Root-dir:Subdir-1:File-6", where a colon is used to separate tree nodes.

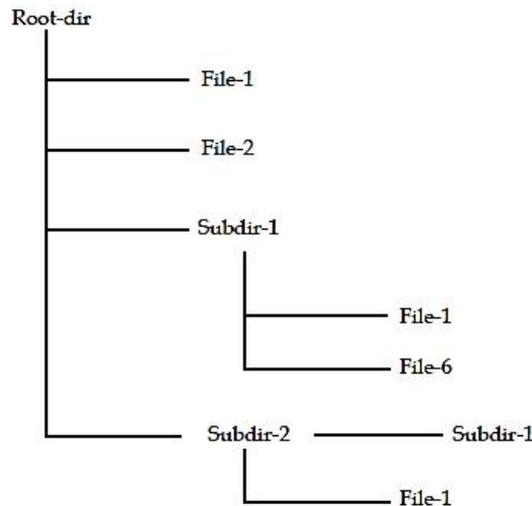


Figure 13.1: Tree Representation of a Hierarchical Directory Structure

A complete pathname is not the only way to identify a file in the directory tree structure: a "relative" pathname, starting from a parent directory is suited just as well, provided that the FMS already knows about that directory. This addressing method can be usefully exploited by making the FMS assign to all processes a "current working directory" (CWD) attribute, i.e. the complete

path name of a directory of interest, and defining a way for the process to identify files by just specifying a "relative" pathname starting from that directory. In the same example, if ":Rootdir:Subdir-1" is the CWD of a process, the above file might be identified simply as "File-6", using the convention that path names not starting with a colon are relative to the CWD. The advantage is twofold: the entire file system structure up to the CWD need not be known by a program (hence its data can be safely moved in other directories without having to rewrite the program), and file access time is decreased, since it's no longer necessary to navigate the whole tree in order to find the address of a file.

Single Level Directory

In single level directory all files are contained in the same directory. It is easy to support and understand. It has some limitations like:

1. Large number of files (naming).
2. Ability to support different users/topics (grouping)

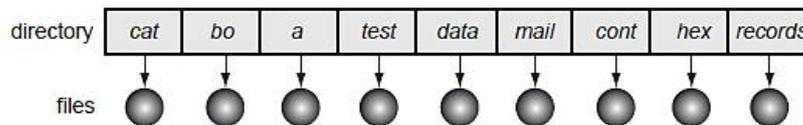


Figure 13.2: Single Level Directory

Two Level Directory

In two level directory structure one is master file directory and the other is user file directory. Here each user has their own user file directory. Each entry in the master file directory points to user file directory. Each user has rights to access their own directory but can't access other user's directory, if permission is not given by the owner of the second one.

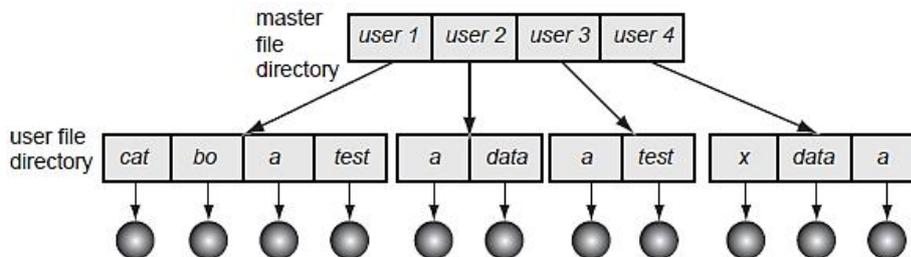


Figure 13.3: Two Level Directory

Three Level Directory

In three level directory the directory structure is a tree with arbitrary height. Here users may create their own subdirectories.

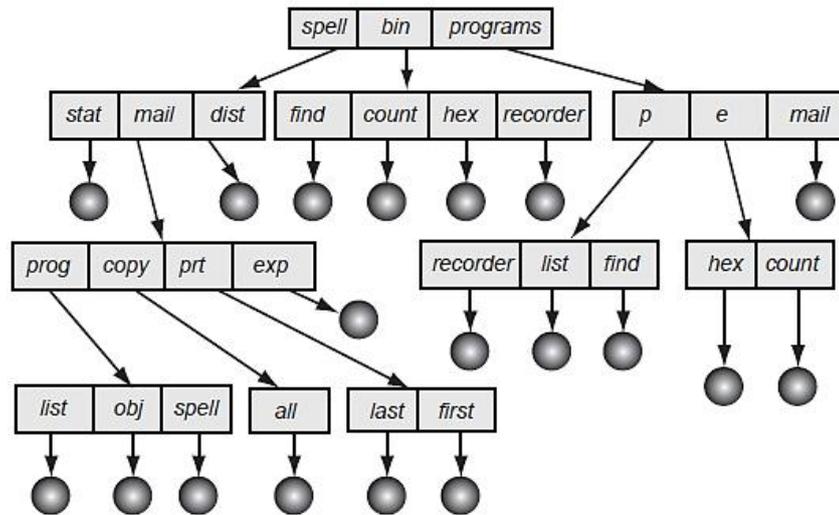


Figure 13.4: Three Level Directory

13.7 File System Mounting

The file system structure is the most basic level of organization in an operating system. Almost all of the ways an operating system interacts with its users, applications, and security model are dependent upon the way it organizes files on storage devices. Providing a common file system structure ensures users and programs are able to access and write files. File systems break files down into two logical categories:

1. Shareable vs. un-shareable files
2. Variable vs. static files

Shareable files are those that can be accessed locally and by remote hosts; un-shareable files are only available locally. Variable files, such as documents, can be changed at any time; static files, such as binaries, do not change without an action from the system administrator. The reason for looking at files in this manner is to help correlate the function of the file with the permissions assigned to the directories which hold them. The way in which the operating system and its users interact with a given file determines the directory in which it is placed, whether that directory is mounted with read-only or read/write permissions, and the level of access each user has to that file. The top level of this organization is crucial. Access to the underlying directories can be restricted or security problems could manifest themselves if, from the top level down, it does not adhere to a rigid structure. It is important to understand the difference between a file system and a directory. A file system is a section of hard disk that has been allocated to contain files. This section of hard disk is accessed by mounting the file system over a directory. After the file system is mounted, it looks just like any other directory to the end user.

However, because of the structural differences between the file systems and directories, the data within these entities can be managed separately. When the operating system is installed for the first time, it is loaded into a directory structure, as shown in the figure 13.5.

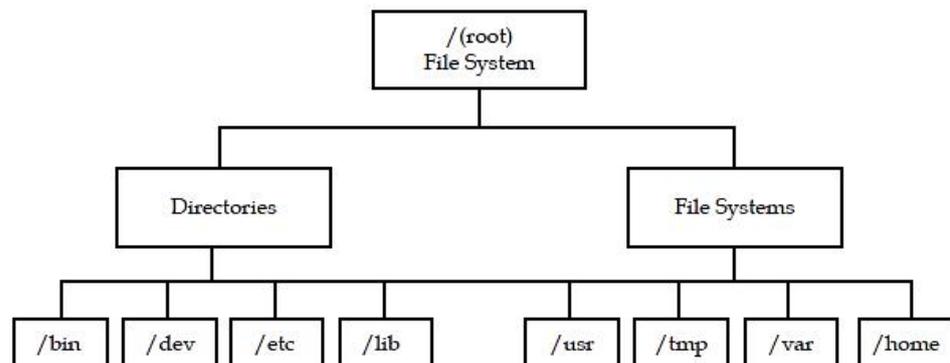


Figure 13.5: File System Tree.

This tree chart shows a directory structure with the / (root) file system at the top, branching downward to directories and file systems. Directories branch to /bin, /dev, /etc, and /lib. File systems branch to /usr, /tmp, /var, and /home.

The directories on the right (/usr, /tmp, /var, and /home) are all file systems so they have separate sections of the hard disk allocated for their use. These file systems are mounted automatically when the system is started, so the end user does not see the difference between these file systems and the directories listed on the left (/bin, /dev, /etc, and /lib).

On standalone machines, the following file systems reside on the associated devices by default:

/File System	/Device
/dev/hd1	/home
/dev/hd2	/usr
/dev/hd3	/tmp
/dev/hd4	/(root)
/dev/hd9var	/var
/proc	/proc
/dev/hd10opt	/opt

The file tree has the following characteristics:

- Files that can be shared by machines of the same hardware architecture are located in the /usr file system.
- Variable per-client files, for example, spool and mail files, are located in the /var file system.
- The / (root) file system contains files and directories critical for system operation. Example: It contains
 - A device directory (/dev)
 - Mount points where file systems can be mounted onto the root file system, for example, /mnt
- The /home file system is the mount point for users' home directories.
- For servers, the /export directory contains paging-space files, per-client (unshared) root file systems, dump, home, and /usr/share directories for diskless clients, as well as exported /usr directories.
- The /proc file system contains information about the state of processes and threads in the system.
- The /opt file system contains optional software, such as applications.

The following list provides information about the contents of some of the subdirectories of the / (root) file system.

/bin	Symbolic link to the /usr/bin directory.
/dev	Contains device nodes for special files for local devices. The /dev directory contains special files for tape drives, printers, disk partitions, and terminals.
/etc	Contains configuration files that vary for each machine. Examples include: <ol style="list-style-type: none"> /etc/hosts

	2. /etc/passwd
/export	Contains the directories and files on a server that are for remote clients.
/home	Serves as a mount point for a file system containing user home directories. The /home file system contains per-user files and directories. In a standalone machine, a separate local file system is mounted over the /home directory. In a network, a server might contain user files that should be accessible from several machines. In this case, the server's copy of the /home directory is remotely mounted onto a local /home file system.
/lib	Symbolic link to the /usr/lib directory, which contains architecture-independent libraries with names in the form lib*.a.
/sbin	Contains files needed to boot the machine and mount the /usrfile system. Most of the commands used during booting come from the boot image's RAM disk file system; therefore, very few commands reside in the /sbin directory.
/tmp	Serves as a mount point for a file system that contains system-generated temporary files.
/u	Symbolic link to the /home directory.
/usr	Serves as a mount point for a file system containing files that do not change and can be shared by machines (such as executable programs and ASCII documentation). Standalone machines mount a separate local file system over the /usr directory. Diskless and disk-poor machines mount a directory from a remote server over the /usrfile system.
/var	Serves as a mount point for files that vary on each machine. The /var file system is configured as a file system because the files that it contains tend to grow. For example, it is a symbolic link to the /usr/tmp directory, which contains temporary work files.

13.8 File Sharing

In today's world where the working is a multiuser environment a file is required to be shared among more than one user. There are several techniques and approaches to affect this operation. Simple approach is to copy the file at the user's local hard disk. This approach essentially creates different files, in therefore cannot be treated as file sharing.

A file can be shared in three different modes:

- 1) **Read only:** The user can only read or copy the file.
- 2) **Linked shared:** All the users can share the file and can make the changes but the changes are reflected in the order defined by the operating systems.
- 3) **Exclusive mode:** The file is acquired by one single user who can make the changes while others can only read or copy it.

Sharing can also be done through symbolic links, but there occur certain problems like concurrent updating problem, deletion problem. Updating cannot be done simultaneously by two users at a time, also one cannot delete a file if it is in use by another user. The solution for this problem is done by locking file techniques.

13.9 Protection

The data in the computer system should be protected and kept secure. A major concern is to protect data from both physical damage (reliability) and improper access (protection). There is a mechanism

in the computer system that a system program or manually it can take the backup or duplicate the files automatically. File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes. Also, the data can be lost due to bugs on system. Protection can be provided in many ways. For a small single-user system, you might provide protection by physically removing the floppy disks and locking them in a desk drawer or file cabinet. In a multi-user system, however, other mechanisms are needed.

13.10 File System Implementation

The most important issue in file storage is keeping track of which disk blocks go with which file. Different operating systems use different methods - contiguous allocation and linked list allocation are important to know. In the former, each file is stored as a contiguous block of data on the disk, in the latter, the file is kept as a linked list of disk blocks - the first word of each block is used as a pointer to the next one. UNIX uses i-nodes to keep track of which blocks belong to each file. An i-node is a table that lists the attributes and disk addresses of the file's blocks. The first few disk addresses are stored in the i-node itself, so for small files, all the necessary information is in the i-node itself which is fetched from disk to main memory when the file is

opened. For larger files, one of the addresses in the i-node is the address of a disk block called a single indirect block which contains additional disk addresses. If this is insufficient, another address called the double indirect block may contain the address of a block that contains a list of single indirect blocks. In order to create the illusion of files from the block-oriented disk drives, the OS must keep track of the location of the sectors containing the data of the file. This is accomplished by maintaining

a set of data structures both in memory and on disk that keep track of where data is allocated to each file, and the name to file mapping encoded in the directory structure.

The simplest allocation of files is a contiguous allocation of sectors to each file. A directory entry would contain the name of the file, its size in bytes and the first and last sector of the file. This results in a fast read of a given file and a compact representation, but also of sizable external

fragmentation which can require compaction to correct. The analog in memory management is the base/limit register system of memory allocation. As with memory management, you turn to more complex data structures and non-contiguous allocation to solve the problems. I can use a bitmap to record the allocated and unallocated sectors on the disk, and keep a list of sectors assigned to each file in its directory entry. This isn't often used, because it makes searching for free space difficult and replicates the allocation information in the file itself. (When it is used, the bitmap is kept both on disk and in memory). The other system that is used in file systems and in memory management is a linked list. A simple mechanism is to take one integer out of every file block and use that as the next sector following this one (similar to linking the holes in memory management). This is an improvement over bitmaps in efficiency of storage use, but has a significant drawback in that finding the proper sector for a random access is expensive. Finding the right sector containing a random sector is as expensive as reading to that point in the file. To solve this, we collect the sector pointers into a table (usually cached in main memory) separate from the files. Now the OS can follow the separate pointers to find the appropriate sector for a random access without reading each disk block. Furthermore, the conceptual disk blocks and the physical disk blocks now have the same size. This is essentially the FAT file system of MS-DOS. Another organization is one optimized for small files (which research has shown dominate the file system, in the sense that most files are small) while accommodating large ones. The system is called the index node or i-node system. An i-node contains the attributes of the file and pointers to its first few blocks.

The last 3 sector pointers are special. The first points to i-node structures that contain only pointers to sectors; this is an indirect block. The second to pointers to pointers to sectors (a double indirect node) and the third to pointers to pointers to pointers to sectors (triple indirect). This results in increasing access times for blocks later in the file. Large files will have longer access times to the end of the file. I-nodes specifically optimize for short files.

13.11 Allocation Methods

One main problem in file management is how to allocate space for files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are contiguous, linked, and indexed. Each method has its advantages and disadvantages.

Accordingly, some systems support all three (e.g. Data General's RDOS). More commonly, a system will use one particular method for all files.

Contiguous Allocation

The contiguous allocation method requires each file to occupy a set of contiguous address on the disk. Disk addresses define a linear ordering on the disk. Notice that, with this ordering, accessing block $b+1$ after block b normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguous allocated files is minimal, as is seek time when a seek is finally needed. Contiguous allocation of a file is defined by the disk address and the length of the first block. If the file is n blocks long, and starts at location b , then it occupies blocks $b, b+1, b+2, \dots, b+n-1$. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file. The difficulty with contiguous allocation is finding space for a new file. If the file to be created is n blocks long, then the OS must search for n free contiguous blocks. First-fit, best-fit, and worst-fit strategies are the most common strategies used to select a free hole from the set of available holes. Simulations have shown that both first-fit and best-fit are better than worst-fit in terms of both time and storage utilization. Neither first-fit nor best-fit is clearly best in terms of storage utilization, but first-fit is generally faster. These algorithms also suffer from external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but this space is not contiguous; storage is fragmented into a large number of small holes.

Another problem with contiguous allocation is determining how much disk space is needed for a file. When the file is created, the total amount of space it will need must be known and allocated. How does the creator (program or person) know the size of the file to be created? In some cases, this determination may be fairly simple (e.g. copying an existing file), but in general the size of an output file may be difficult to estimate.

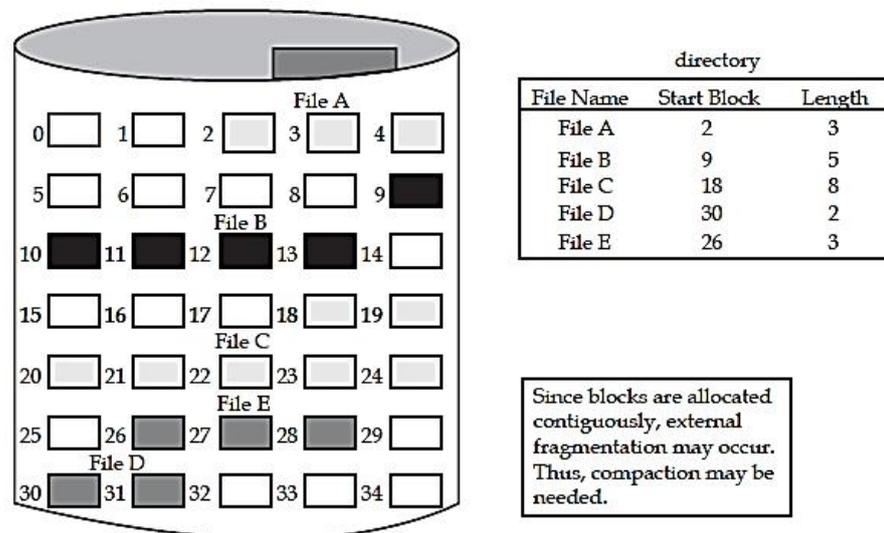


Figure 13.6: Diagram of Contiguous Allocation

Linked Allocation

The problems in contiguous allocation can be traced directly to the requirement that the spaces be allocated contiguously and that the files that need these spaces are of different sizes. These requirements can be avoided by using linked allocation. In linked allocation, each file is a linked list of disk blocks. The directory contains a pointer to the first and (optionally the last) block of the file. For example, a file of 5 blocks which starts at block 4, might continue at block 7, then

block 16, block 10, and finally block 27. Each block contains a pointer to the next block and the last block contains a NIL pointer. The value -1 may be used for NIL to differentiate it from block 0.

With linked allocation, each directory entry has a pointer to the first disk block of the file. This pointer is initialized to nil (the end-of-list pointer value) to signify an empty file. A write to a file removes the first free block and writes to that block. This new block is then linked to the end of the file. To read a file, the pointers are just followed from block to block.

There is no external fragmentation with linked allocation. Any free block can be used to satisfy a request. Notice also that there is no need to declare the size of a file when that file is created. A file can continue to grow as long as there are free blocks. Linked allocation, does have disadvantages, however. The major problem is that it is inefficient to support direct-access; it is effective only for sequential-access files. To find the *i*th block of a file, it must start at the beginning of that file and follow the pointers until the *i*th block is reached.

Another severe problem is reliability. A bug in OS or disk hardware failure might result in pointers being lost and damaged. The effect of which could be picking up a wrong pointer and linking it to a free block or into another file.

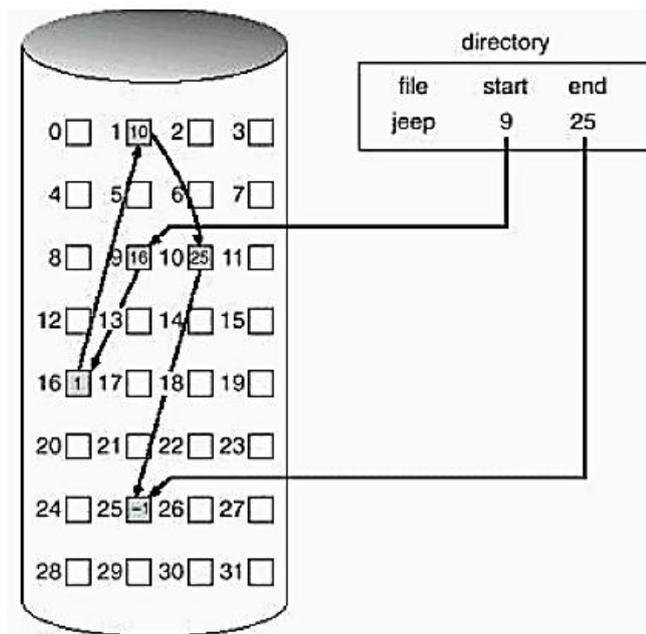


Figure 13.7: Diagram of Linked Allocation

Indexed Allocation

The indexed allocation method is the solution to the problem of both contiguous and linked allocation. This is done by bringing all the pointers together into one location called the index block. Of course, the index block will occupy some space and thus could be considered as an overhead of the method. In indexed allocation, each file has its own index block, which is an array of disk sector addresses. The *i*th entry in the index block points to the *i*th sector of the file. The directory contains the address of the index block of a file. To read the *i*th sector of the file, the pointer in the *i*th index block entry is read to find the desired sector. Indexed allocation supports direct access, without suffering from external fragmentation. Any free block anywhere on the disk may satisfy a request for more space.

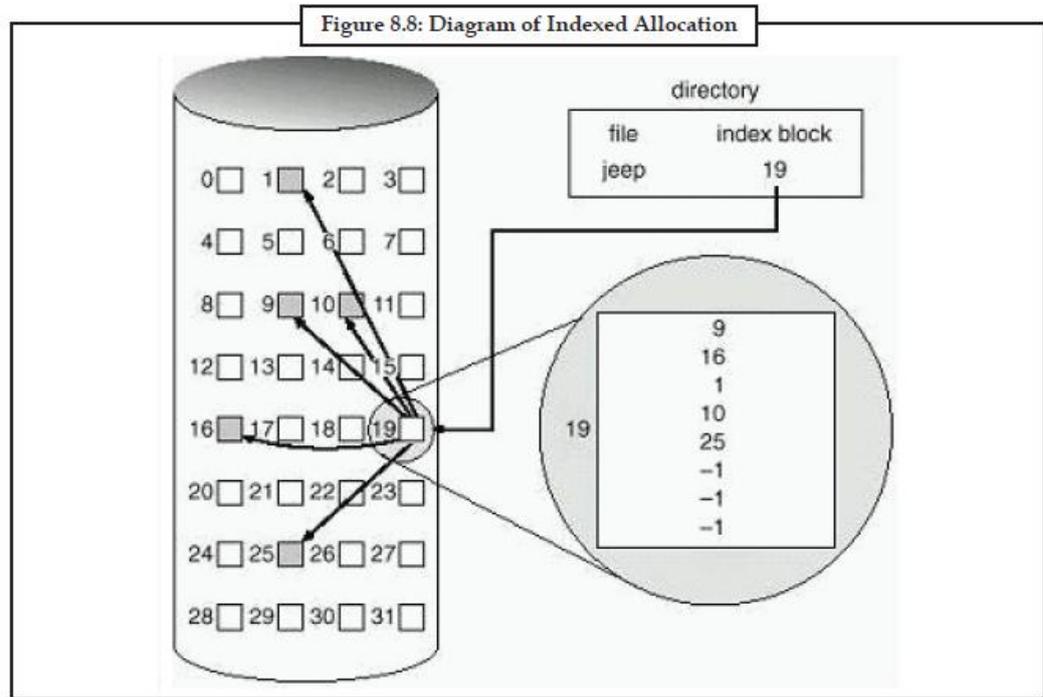


Figure 13.8:

13.12 Free-space Management

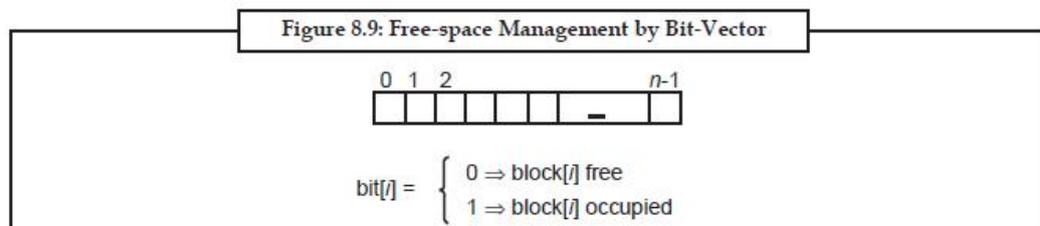
Since there is only a limited amount of disk space, it is necessary to reuse the space from deleted files for new files. To keep track of free disk space, the system maintains a free-space list. The free-space list records all disk blocks that are free (i.e., are not allocated to some file). To create a file, the free-space list has to be searched for the required amount of space, and allocate that space to a new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list.

Bit-Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by a 1 bit. If the block is free, the bit is 0; if the block is allocated, the bit is 1.

Example: Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks are allocated. The free-space bit map would be: 1100001100000011100111110001111...

The main advantage of this approach is that it is relatively simple and efficient to find n consecutive free blocks on the disk. Unfortunately, bit vectors are inefficient unless the entire vector is kept in memory for most accesses. Keeping it in main memory is possible for smaller disks such as on microcomputers, but not for larger ones.

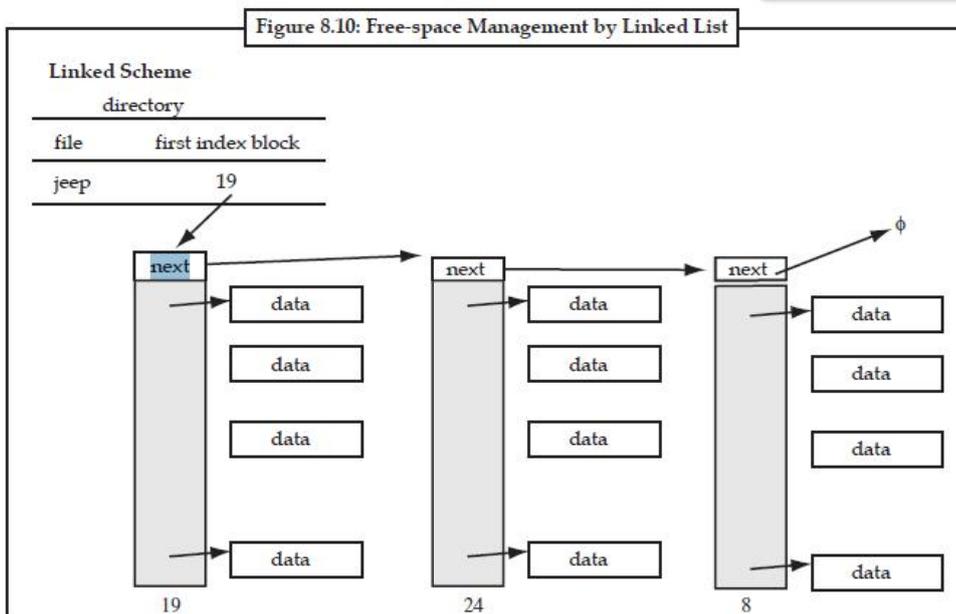


Linked List

Another approach is to link all the free disk blocks together, keeping a pointer to the first free block. This block contains a pointer to the next free disk block, and so on. In the previous example,

a pointer could be kept to block 2, as the first free block. Block 2 would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on. This scheme is not efficient; to traverse the list, each block must be read, which requires substantial I/O time.

Next



Grouping

A modification of the free-list approach is to store the addresses of n free blocks in the first free block. The first $n-1$ of these are actually free. The last one is the disk address of another block containing addresses of another n free blocks. The importance of this implementation is that addresses of a large number of free blocks can be found quickly.

Counting

Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when contiguous allocation is used. Thus, rather than keeping a list of free disk addresses, the address of the first free block is kept and the number n of free contiguous blocks that follow the first block. Each entry in the free-space list then consists of a disk address and a count. Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

13.13 Directory Implementation

Directories are generally simply files with a special interpretation. Some directory structures contain the name of a file, its attributes and a pointer³ either into its FAT list or to its i-node. This choice bears directly on the implementation of linking. If attributes are stored directly in the directory node, (hard) linking is difficult because changes to the file must be mirrored in all directories. If the directory entry simply points to a structure (like an i-node) that holds the attributes internally, only that structure needs to be updated. The simplest method is to use a linear list of file names with pointers to the data blocks. This requires a linear search to find a particular entry. Hash tables are also used by some operating systems - a linear list stores the directory entries but a hash function based on some computation from the file name returns a pointer to the file name in the list. Thus, directory search time is greatly reduced.

In UNIX, each entry in the directory contains just a file name and its i-node number. When a file is opened, the file system takes the file name and locates its disk blocks. The i-node is read into memory and kept there until the file is closed.

Summary

- File is a named collection of data stored in a device.
- File manager is an integral part of the operating system which is responsible for the maintenance of secondary storage.
- File system is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.
- Disk file system is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.
- Flash file system is a file system designed for storing files on flash memory devices. Network file system is a file system that acts as a client for a remote file access protocol, providing access to files on a server.
- Flat file system is a file system where is no subdirectories and everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc.
- Directory is simple file containing the indexing of other files, which may in turn be directories if a hierarchical indexing scheme is used.

Keywords

- **Directory:** It is simple file containing the indexing of other files, which may in turn be directories if a hierarchical indexing scheme is used.
- **Disk file system:** It is a file system designed for the storage of files on a data storage device, most commonly a disk drive, which might be directly or indirectly connected to the computer.
- **File manager:** It is an integral part of the operating system which is responsible for the maintenance of secondary storage.
- **File system:** It is a set of abstract data types that are implemented for the storage, hierarchical organization, manipulation, navigation, access, and retrieval of data.
- **File:** It is a named collection of data stored in a device.
- **Flash file system:** It is a file system designed for storing files on flash memory devices.
- **Flat file system:** It is a file system where no subdirectories are present and everything is stored at the same (root) level on the media, be it a hard disk, floppy disk, etc.
- **Network file system:** It is a file system that acts as a client for a remote file access protocol, providing access to files on a server.

Self Assessment

1. Shareable files are those that can be accessed and by
2. Three major methods of allocating disk space are, and.....
3. The difficulty with contiguous allocation is for a new file.
4. There is no external fragmentation with allocation.
5. FAT stands for
6. NTFS stands for

7. Direct access is based on a of a file.
8. files support both formatted and unformatted record types.
9. A system is a file system designed for storing files on flash memory devices.
10. A is a collection of letters, numbers and special characters.
11. Which of the following is an approach to restricting system access to authorized users.
- Role-based access control
 - Process-based access control
 - Job-based access control
 - None of the mentioned
12. Contiguous allocation of a file is defined by _____
- disk address of the first block & length
 - length & size of the block
 - size of the block
 - total size of the file
13. In which method, a file-length logical record exists that allows the program to read and write record rapidly in no particular order?
- Sequential access
 - Direct access
 - Indexed sequential access
 - None of the given choices
14. Which of the following techniques always reads or writes a large block of data, which contains several file records, from or to the I/O medium?
- Buffering of records
 - Blocking of records
 - Buffering and blocking of records
 - None of the above
15. A Domain may be defined as a _____
- set of all objects
 - collection of protection policies
 - set of access-rights
 - None of the mentioned

Answers for Self Assessment

- | | | | | |
|--------------------------------|--|----------------------|---------------|--------------------------------|
| 1. locally,
remote
hosts | 2. contiguous,
linked, and
indexed | 3. finding
space | 4. linked | 5. File
allocation
table |
| 6. NT File
system | 7. disk model | 8. direct-
access | 9. flash file | 10. file |

11. A 12. A 13. B 14. B 15. C

Review Questions

1. What is a directory? Can we consider a directory as a file? Explain your answer.
2. What is a flash file system? Give an example of it.
3. What are the differences between file system and file manager?
4. Write short notes on:
 - (a) Disk file system
 - (b) Flat file system
 - (c) Network file system
5. What are the differences between a file system and a directory?
6. What is logical damage of data? How can it be recovered?
7. Write a short note on free space management.
8. What is indexed allocation method? How differs does it from linked list allocation?
9. Compare and contrast between contiguous disk space allocation method and linked list allocation method.
10. What is disk scheduling? Describe different disk scheduling policies.



Further Readings

Andrew M. Lister, Fundamentals of Operating Systems, Wiley.

Andrew S. Tanenbaum And Albert S. Woodhull, Systems Design and Implementation, Prentice Hall.

Andrew S. Tanenbaum, Modern Operating System, Prentice Hall.

Colin Ritchie, Operating Systems, BPB Publications.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

I.A. Dhotre, Operating System, Technical Publications.

Milankovic, Operating System, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

Unit 14: Disk Management

CONTENTS

Objectives

Introduction

- 14.1 The Benefits of Secondary Storage
- 14.2 Disk Scheduling
- 14.3 Shortest Seek Time First (SSTF)
- 14.4 Selecting a Disk Scheduling Algorithm
- 14.5 Disk Management
- 14.6 Swap Space Management
- 14.7 Swap Space Parameters
- 14.8 Swap Space Management
- 14.9 RAID Structure

Summary

Keywords

Self Assessment

Answers for Self Assessment

Review Questions

Further Readings

Objectives

After studying this unit, you will be able to:

- understand the goals and principles of protection in a modern computer system
- analyze capability and language-based protection system
- understand the different file systems
- learn the concept of file directories
- understand the different types of file directories
- understand the different file systems
- learn the concept of file directories
- understand the different types of file directories

Introduction

Any non-volatile storage medium that is not directly accessible to the processor. Memory directly accessible to the processor includes main memory, cache and the CPU registers. Secondary storage includes hard drives, magnetic tape, CD-ROM, DVD drives, floppy disks, punch cards and paper tape.

Secondary storage devices are usually accessed via some kind of controller. This contains registers that can be directly accessed by the CPU like main memory ("memory mapped"). Reading and writing these registers can cause the device to perform actions like reading a block of data off a disk or rewinding a tape.

14.1 The Benefits of Secondary Storage

Picture, if you can, how many filing-cabinet drawers would be required to hold the millions of files of, say, tax records kept by the Internal Revenue Service or historical employee records kept by General Motors. The record storage rooms would have to be enormous. Computers, in contrast, permit storage on tape or disk in extremely compressed form. Storage capacity is unquestionably one of the most valuable assets of the computer.

Secondary storage, sometimes called auxiliary storage, is storage separate from the computer itself, where you can store software and data on a semi-permanent basis. Secondary storage is necessary because memory, or primary storage, can be used only temporarily. If you are sharing your computer, you must yield memory to someone else after your program runs; if you are not sharing your computer, your programs and data will disappear from memory when you turn off the computer. However, you probably want to store the data you have used or the information you have derived from processing; that is why secondary storage is needed. Furthermore, memory is limited in size, whereas secondary storage media can store as much data as necessary. Keep in mind the characteristics of the memory hierarchy that were described in the section on the CPU and memory.

Storage	Speed	Capacity	Relative Cost (\$)	Permanent
Registers	Fastest	Lowest	Highest	No
RAM	Very Fast	Low/Moderate	High	No
Floppy Disk	Very Slow	Low	Low	Yes
Hard Disk	Moderate	Very High	Very Low	Yes

The benefits of secondary storage can be summarized as follows:

Capacity: Organizations may store the equivalent of a roomful of data on sets of disks that take up less space than a breadbox. A simple diskette for a personal computer holds the equivalent of 500 printed pages, or one book. An optical disk can hold the equivalent of approximately 400 books.

Reliability: Data in secondary storage is basically safe, since secondary storage is physically reliable. Also, it is more difficult for unscrupulous people to tamper with data on disk than data stored on paper in a file cabinet.

Convenience: With the help of a computer, authorized people can locate and access data quickly.

Cost: Together the three previous benefits indicate significant savings in storage costs. It is less expensive to store data on tape or disk (the principal means of secondary storage) than to buy and house filing cabinets. Data that is reliable and safe is less expensive to maintain than data subject to errors. But the greatest savings can be found in the speed and convenience of filing and retrieving data.

These benefits apply to all the various secondary storage devices but, as you will see, some devices are better than others. We begin with a look at the various storage media, including those used for personal computers, and then consider what it takes to get data organized and processed.

Disk Structure

Disks provide the bulk of secondary storage for modern computer systems. Magnetic tape was used as an early secondary-storage medium, but the access time is much slower than for disks. Thus, tapes are currently used mainly for backup, for storage of infrequently used information, as a medium for transferring information from one system to another, and for storing quantities of data so large that they are impractical as disk systems. Modern disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to choose a different logical block size, such as 1,024 bytes. The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest

of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost. By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons. First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives. On media that use **constant linear velocity (CLV)**, the density of bits per track is uniform. The farther a track is from the center of the disk, the greater its length, so the more sectors it can hold. As we move from outer zones to inner zones, the number of sectors per track decreases. Tracks in the outermost zone typically hold 40 percent more sectors than do tracks in the innermost zone. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head. This method is used in CD-ROM and DVD-ROM drives. Alternatively, the disk rotation speed can stay constant, and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant. This method is used in hard disks and is known as **constant angular velocity (CAV)**. The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.



Did you know?

The number of sectors per track has been increasing as disk technology improves, and the outer zone of a disk usually has several hundred sectors per track. Similarly, the number of cylinders per disk has been increasing; large disks have tens of thousands of cylinders.

14.2 Disk Scheduling

In multiprogramming systems, many processes may be generating requests for reading and writing disk records. Because these processes often make requests faster than they can be serviced by the moving head disks, waiting queues are built up for each device. In order to stop unbounded increase in the queue length these pending requests must be examined and serviced in an efficient manner. Disk scheduling involves a careful examination of pending requests to determine the most efficient ways to service the waiting requests terms.

Latency Time: The time it takes for the data block to rotate from its current to just under the read-write head is called latency time.

Seek Time: The time it takes to position the read-write head on the top of the track where data block is stored.

Transfer Time: The time it takes to transfer a block of data from the disk to memory.

These times \gg CPU processing time.



Transfer rate of RL81 (VAX Disk) = 2.2 mb/sec

Data block size = 512 bytes.

Example

Total block transfer time (Latency + Seek + Transfer) = about 0.1 sec.

CPU will take about 9600 ns (0.0000096 sec) to read this block.



Required tracks: 98, 183, 37, 122, 14, 124, 14, 124, 65 and 67. Head starts at: 53

Example

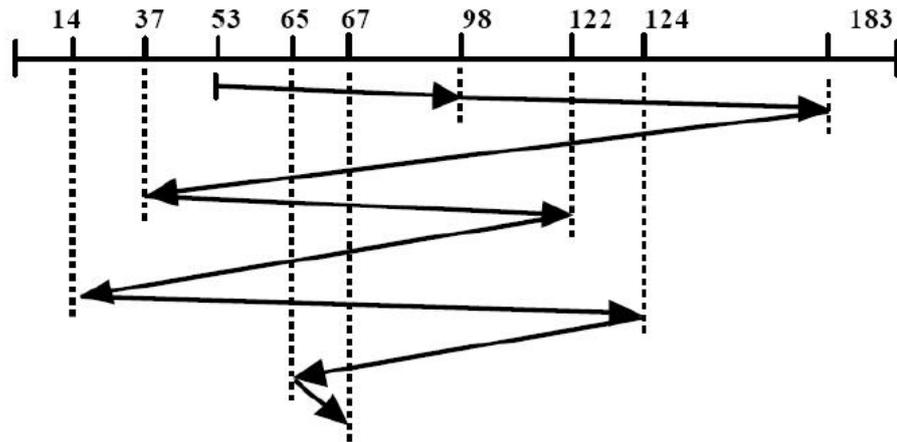


Figure: Total Tracks Covered: 53 to 98 = 45, 98 to 183 = 85, and so on = 640

Problems: Wild swing. Several close requests can be serviced together such as 37 and 14, 122 and 124 etc.

14.3 Shortest Seek Time First (SSTF)

Service all requests close to the current head position together, before moving the head far away to service another request. This policy is similar to shortest job first.

Example

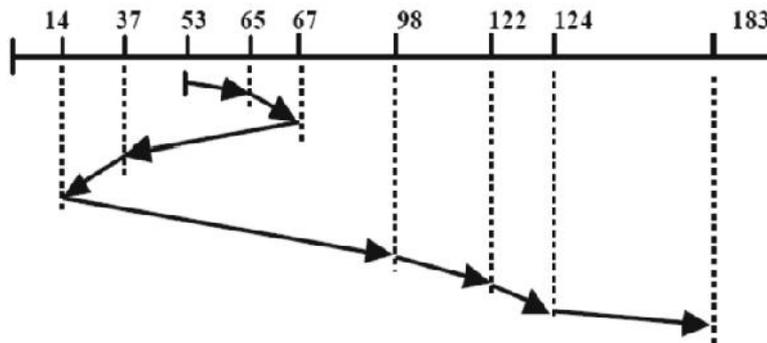


Figure: Total Number of Tracks Covered: 236

Problems: May cause starvation to some requests, since requests may arrive at anytime.

Scan

Head starts at one end and moves towards the other end, servicing the requests on its way. At the end the head movement direction is reversed and servicing continues. Example, head position at 53 movement towards zero, servicing 37, 14 and goes up to zero and then changes direction.

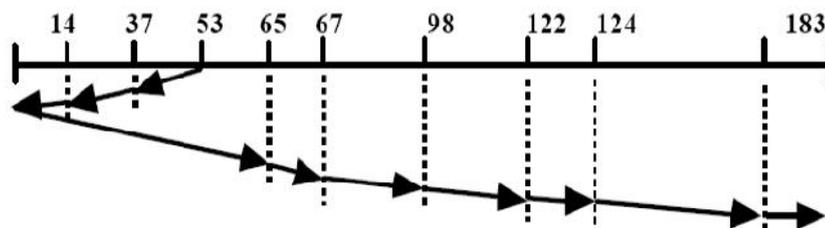


Figure: Scan

This algorithm is sometimes called “elevator” algorithm, since it resembles to the behavior of elevator.

Cscan

A variant of scan designed to provide a more uniform wait. Starts from one end and moves towards the other end servicing all requests on its way. When the head reaches to the other end, it

immediately returns to the beginning of the disk, without servicing any request on its return journey.

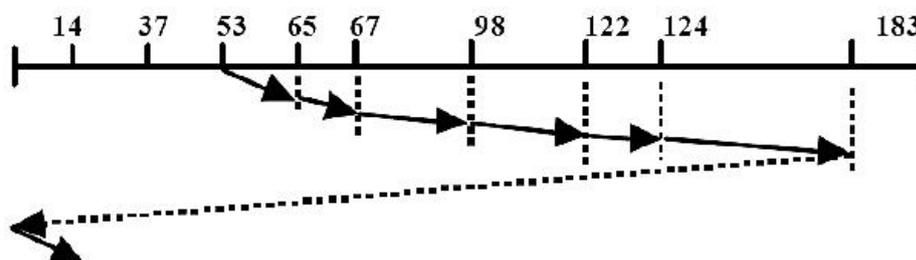


Figure: Cscan

14.4 Selecting a Disk Scheduling Algorithm

The performance of these algorithms depends heavily on the workload (number of requests). Under light load all algorithms perform the same. If the queue seldom has more than one outstanding request, then all algorithms are effectively the same. Their performance also depends upon the file organization and the type of generated requests. In a sequential processing and sequential file, the head movement will be minimum and therefore the seek time and latency time will be minimum so FCFS may perform better. A indexed sequential file, on the other hand, may include blocks that may be scattered all over the disk and a sequential processing with FCFS will be very slow. SSTF is quite common and scan and cscan are good for heavy load.

Scheduling Fixed-head Devices

Fixed head disk = DRUM. One head per track on the drum.

Seek time = 0.

Latency time < moving head disk.

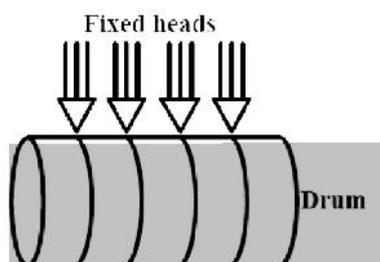


Figure: Scheduling Fixed-head Devices

Different algorithm is required for this device.

14.5 Disk Management

The hard disk is the secondary storage device that is used in the computer system. Usually the primary memory is used for the booting up of the computer. But a hard disk drive is necessary in the computer system since it needs to store the operating system that is used to store the information of the devices and the management of the user data. The management of the IO devices that is the Input Output devices, like the printer and the other peripherals like the keyboard and the etc.; all require the usage of the operating system. Hence the information of the all such devices and the management of the system are done by the operating system. The operating system works as an interpreter between the machine and the user. The operating system is a must for the proper functioning of the computer. The computer is a device that needs to be fed with the instructions that are to be carried out and executed. Hence there needs to be an interpreter who is

going to carry out the conversions from the high-level language of the user to the low-level language of the computer machine.

The hard disk drive as secondary memory is therefore needed for the purpose of installing the operating system. If there is no operating system then the question arises where to install the operating system. The operating system obviously cannot be installed in the primary memory however large that may be. The primary memory is also a volatile memory that cannot be used for the permanent storage of the system files of the operating system. The operating system requires the permanent file storage media like the hard disk.

Moreover, the hard disk management is an important part of maintaining the computer, since it requires an efficient management of the data or the user information. The information regarding the Master Boot Record is stored in the hard disk drive. This is the information that is required during the startup of the computer. The computer system needs this information for loading the operating system.

The file management and the resources management are also a part of the hard disk management. The hard disk management requires an efficient knowledge of the operating system and its resources and the methods of how these resources can be employed in order to achieve maximum benefit. The operating system contains the resources and the tools that are used to manage the files in the operating system. The partitioning and the installation of the operating system itself may be considered as the hard disk management.

The hard disk management also involves the formatting of the hard disk drive and to check the integrity of the file system. The data redundancy check can also be carried out for the consistency of the hard disk drive. The hard disk drive management is also important in the case of the network where there are many hard disk drives to be managed.

Managing a single hard disk in a single user operating system is quite easy in comparison with the management of the hard disk drives in a multi user operating system where there is more than one user. It is not that much easy since the users are also required to be managed.

Disk Formatting

A new magnetic disk is a blank slate: It is just platters of a magnetic recording material. Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting (or **physical formatting**). **Low-level formatting** fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size), and a trailer. The header and trailer contain information used by the disk controller, such as a sector number and an **error-correcting code (ECC)**. When the controller writes a sector of data during normal I/O, the ECC is updated with a value calculated from all the bytes in the data area. When the sector is read, the ECC is recalculated and is compared with the stored value. If the stored and calculated numbers are different, this mismatch indicates that the data area of the sector has become corrupted and that the disk sector may be bad. The ECC is an error-correcting code because it contains enough information that, if only a few bits of data have been corrupted, the controller can identify which bits have changed and can calculate what their correct values should be. The controller automatically does the ECC processing whenever a sector is read or written. Most hard disks are low-level formatted at the factory as a part of the manufacturing process. This formatting enables the manufacturer to test the disk and to initialize the mapping from logical block numbers to defect-free sectors on the disk. For many hard disks, when the disk controller is instructed to low-level format the disk, it can also be told how many bytes of data space to leave between the header and trailer of all sectors. It is usually possible to choose among a few sizes, such as 256, 512, and 1,024 bytes. Formatting a disk with a larger sector size means that fewer sectors can fit on each track, but that also means fewer headers and trailers are written on each track, and thus increases the space available for user data. Some operating systems can handle only a sector size of 512 bytes. To use a disk to hold files, the operating system still needs to record its own data structures on the disk. It does so in two steps. The first step is to **partition** the disk into one or more groups of cylinders. The operating system can treat each partition as though it were a separate disk.

For instance, one partition can hold a copy of the operating system's executable code, while another holds user files. After partitioning, the second step is **logical formatting** (or creation of a file system). In this step, the operating system stores the initial file-system data structures onto the disk.

These data structures may include maps of free and allocated space (a FAT or inodes) and an initial empty directory.

Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed raw I/O. For example, some database systems prefer raw I/O because it enables them to control the exact disk location where each database record is stored. Raw I/O bypasses all the file-system services, such as the buffer cache, file locking, prefetching, space allocation, file names, and directories. We can make certain applications more efficient by implementing their own special-purpose storage services on a raw partition, but most applications perform better when they use the regular file-system services.

Boot Block

For a computer to start running—for instance, when it is powered up or rebooted—it needs to have an initial program to run. This initial bootstrap program tends to be simple. It initializes all aspects of the system, from CPU registers to device controllers and the contents of main memory, and then starts the operating system. To do its job, the bootstrap program finds the operating system kernel on disk, loads that kernel into memory, and jumps to an initial address to begin the operating-system execution. For most computers, the bootstrap is stored in read-only memory (ROM). This location is convenient, because ROM needs no initialization and is at a fixed location that the processor can start executing when powered up or reset. And, since ROM is read only, it cannot be infected by a computer virus. The problem is that changing this bootstrap code requires changing the ROM hardware chips. For this reason, most systems store a tiny bootstrap loader program in the boot ROM, whose only job is to bring in a full bootstrap program from disk. The full bootstrap program can be changed easily: A new version is simply written onto the disk. The full bootstrap program is stored in a partition called the boot blocks, at a fixed location on the disk. A disk that has a boot partition is called a boot disk or system disk. The code in the boot ROM instructs the disk controller to read the boot blocks into memory (no device drivers are loaded at this point), and then starts executing that code. The full bootstrap program is more sophisticated than the bootstrap loader in the boot ROM; it is able to load the entire operating system from a non-fixed location on disk, and to start the operating system running. Even so, the full bootstrap code may be small. For example, MS-DOS uses one 512-byte block for its boot program. Be careful during the disk management because the disk is very important in a computer system.

Bad Blocks

Because disks have moving parts and small tolerances (recall that the disk head flies just above the disk surface), they are prone to failure. Sometimes the failure is complete, and the disk needs to be replaced, and its contents restored from backup media to the new disk. More frequently, one or more sectors become defective. Most disks even come from the factory with bad blocks. Depending on the disk and controller in use, these blocks are handled in a variety of ways. On simple disks, such as some disks with IDE controllers, bad blocks are handled manually. For instance, the MS-DOS format command does a logical format and, as a part of the process, scans the disk to find bad blocks. If format finds a bad block, it writes a special value into the corresponding FAT entry to tell the allocation routines not to use that block. If blocks go bad during normal operation, a special program (such as `chkdsk`) must be run manually to search for the bad blocks and to lock them away as before. Data that resided on the bad blocks usually are lost.

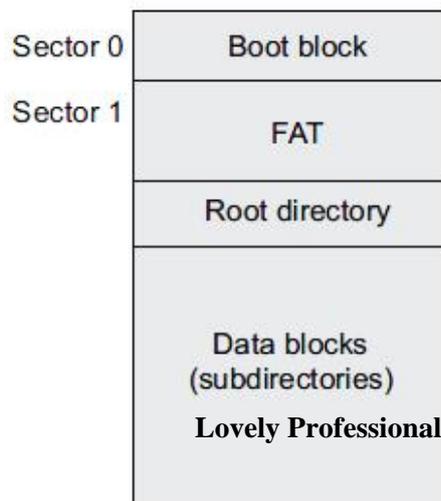


Figure 7.6: MS-DOS Disk Layout

More sophisticated disks, such as the SCSI disks used in high-end PCs and most workstations and servers, are smarter about bad-block recovery. The controller maintains a list of bad blocks on the disk. The list is initialized during the low-level format at the factory, and is updated over the life of the disk. Low level formatting also sets aside spare sectors not visible to the operating system. The controller can be told to replace each bad sector logically with one of the spare sectors. This scheme is known as sector sparing or forwarding. A typical bad-sector transaction might be as follows: The operating system tries to read logical block 87. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system. The next time that the system is rebooted, a special command is run to tell the SCSI controller to replace the bad sector with a spare. After that, whenever the system requests logical block 87, the request is translated into the replacement sector's address by the controller. Such a redirection by the controller could invalidate any optimization by the operating system's disk-scheduling algorithm! For this reason, most disks are formatted to provide a few spare sectors in each cylinder, and a spare cylinder as well. When a bad block is remapped, the controller uses a spare sector from the same cylinder, if possible. As an alternative to sector sparing, some controllers can be instructed to replace a bad block by sector slipping. Here is an example: Suppose that logical block 17 becomes defective, and the first available spare follows sector 202. Then, sector slipping would remap all the sectors from 17 to 202, moving them all down one spot. That is, sector 202 would be copied into the spare, then sector 201 into 202, and then 200 into 201, and so on, until sector 18 is copied into sector 19. Slipping the sectors in this way frees up the space of sector 18, so sector 17 can be mapped to it. The replacement of a bad block generally is not a totally automatic process because the data in the bad block are usually lost. Thus, whatever file was using that block must be repaired (for instance, by restoration from a backup tape), and that requires manual intervention. The operating system tries to read logical block 87. The controller calculates the ECC and finds that the sector is bad. It reports this finding to the operating system.

14.6 Swap Space Management

Swap space is an area on a high-speed storage device (almost always a disk drive), reserved for use by the virtual memory system for deactivation and paging processes. At least one swap for use by the virtual memory system for deactivation and paging processes. At least one swap device (primary swap) must be present on the system. During system startup, the location (disk block number) and size of each swap device is displayed in 512 KB blocks. The swapper reserves swap space at process creation time, but does not allocate swap space from the disk until pages need to go out to disk. Reserving swap at process creation protects the swapper from running out of swap space. You can add or remove swap as needed (that is, dynamically) while the system is running, without having to regenerate the kernel. HP-UX uses both physical and pseudo swap to enable efficient execution of programs.

Pseudo-Swap Space

System memory used for swap space is called pseudo-swap space. It allows users to execute processes in memory without allocating physical swap. Pseudo-swap is controlled by an operating-system parameter; by default, `swpmem` on is set to 1, enabling pseudo-swap. Typically, when the system executes a process, swap space is reserved for the entire process, in case it must be paged out. According to this model, to run one gigabyte of processes, the system would have to have one gigabyte of configured swap space. Although this protects the system from running out of swap space, disk space reserved for swap is under-utilized if minimal or no swapping occurs.

To avoid such waste of resources, HP-UX is configured to access up to three-quarters of system memory capacity as pseudo-swap. This means that system memory serves two functions: as process-execution space and as swap space. By using pseudo-swap space, a one-gigabyte memory system with one-gigabyte of swap can run up to 1.75 GB of processes. As before, if a process attempts to grow or be created beyond this extended threshold, it will fail. When using pseudo-swap for swap, the pages are locked; as the amount of pseudo-swap increases, the amount of lockable memory decreases. For factory-floor systems (such as controllers), which perform best when the entire application is resident in memory, pseudo-swap space can be used to enhance performance: you can either lock the application in memory or make sure the total number of

processes created does not exceed three-quarters of system memory. Pseudo-swap space is set to a maximum of three-quarters of system memory because the system can begin paging once three-quarters of system available memory has been used. The unused quarter of memory allows a buffer between the system and the swapper to give the system computational flexibility. When the number of processes created approaches capacity, the system might exhibit thrashing and a decrease in system response time. If necessary, you can disable pseudo-swap space by setting the tunable parameter `swpmem` in `/usr/conf/master.d/core-hpux` to zero. At the head of a doubly linked list of regions that have pseudo-swap allocated is a null terminated list called `pswaplist`.

Physical Swap Space

There are two kinds of physical swap space – device swap and file-system swap.

Device Swap Space

Device swap space resides in its own reserved area (an entire disk or logical volume of an LVM disk) and is faster than file-system swap because the system can write an entire request (256KB) to a device at once.

File-system Swap Space

File-system swap space is located on a mounted file system and can vary in size with the system's swapping activity. However, its throughput is slower than device swap, because free file-system blocks may not always be contiguous; therefore, separate read/write requests must be made for each file-system block. To optimize system performance, file-system swap space is allocated and de-allocated in `swchunk`-sized chunks. `Swchunk` is a configurable operating system parameter; its default is 2048 KB (2 MB). Once a it is released for file system use, unless it has been pre-allocated with `swapon`. If swapping to file-system swap space, each chunk of swap space is a file in the file system swap directory, and has a name constructed from the system name and the `swaptab` index (such as `becky.6` for `swaptab[6]` on a system named `becky`).

14.7 Swap Space Parameters

Several configurable parameters deal with swapping.

Table 14.1: Configurable Swap Space Parameters

Parameter	Purpose
<code>swchunk</code>	The number of <code>DEV_BSIZE</code> blocks in a unit of swap space, by default, 2 MB on all systems.
<code>maxswapchunks</code>	Maximum number of swap chunks allowed on a system.
<code>swpmem_on</code>	Parameter allowing creation of more processes than you have physical swap space for, by using pseudo-swap.

Swap Space Global Variables

When the kernel is initialized, `conf.c` includes `globals.h`, which contains numerous characteristics related to swap space, shown in the next table. The most important to swap space reservation are `swspc_cnt`, `swspc_max`, `swpmem_cnt`, `swpmem_max`, and `sys_mem`.

14.8 Swap Space Management

Swap space is an area on a high-speed storage device (almost always a disk drive), reserved for use by the virtual memory system for deactivation and paging processes. At least one swap device (primary swap) must be present on the system.

During system startup, the location (disk block number) and size of each swap device is displayed in 512-KB blocks. The swapper reserves swap space at process creation time, but do not allocate swap space from the disk until pages need to go out to disk. Reserving swap at process creation protects the swapper from running out of swap space. You can add or remove swap as needed (that is, dynamically) while the system is running, without having to regenerate the kernel.

Pseudo-Swap Space

When the system memory is used for swap space then it is called pseudo-swap space. It allows users to execute processes in memory without allocating physical swap. Pseudo-swap is controlled by an operating-system parameter.

Typically, when the system executes a process, swap space is reserved for the entire process, in case it must be paged out. According to this model, to run one gigabyte of processes, the system would have to have one gigabyte of configured swap space. Although this protects the system from running out of swap space, disk space reserved for swap is under-utilized if minimal or no swapping occurs. When using pseudo swap for swap, the pages are locked; as the amount of pseudo-swap increases, the amount of lockable memory decreases. Pseudo-swap space is set to a maximum of three-quarters of system memory because the system can begin paging once three-quarters of system available memory has been used. The unused quarter of memory allows a buffer between the system and the swapper to give the system computational flexibility. When the number of processes created approaches capacity, the system might exhibit thrashing and a decrease in system response time.

Physical Swap Space

There are two kinds of physical swap space: device swap and file-system swap.

Device Swap Space

Device swap space resides in its own reserved area (an entire disk or logical volume of an LVMdisk) and is faster than file-system swap because the system can write an entire request (256 KB) to a device at once.

File-system Swap Space

File-system swap space is located on a mounted file system and can vary in size with the system's swapping activity. However, its throughput is slower than device swap, because free file-system blocks may not always be contiguous; therefore, separate read/write requests must be made for each file-system block.

Three Rules of Swap Space Allocation

1. Start at the lowest priority swap device or file system. The lower the number, the higher priority; that is, space is taken from a system with a zero priority before it is taken from a system with a one priority.
2. If multiple devices have the same priority, swap space is allocated from the devices in around-robin fashion. Thus, to interleave swap requests between a number of devices, the devices should be assigned the same priority. Similarly, if multiple file systems have the same priority, requests for swap are interleaved between the file systems. In the figure, swap requests are initially interleaved between the two swap devices at priority 0.
3. If a device and a file system have the same swap priority, all the swap space from the device is allocated before any file-system swap space. Thus, the device at priority 1 will be filled before swap is allocated from the file system at priority 1.

14.9 RAID Structure

RAID stands for Redundant Array of Independent (or Inexpensive) Disks. It involves the configuration (setting up) of two or more drives in combination for fault tolerance and performance. RAID disk drives are used frequently on servers and are increasingly being found in home and office personal computers.

Disks have high failure rates and hence there is the risk of loss of data and lots of downtime for restoring and disk replacement. To improve disk usage many techniques have been implemented. One such technology is RAID (Redundant Array of Inexpensive Disks). Its organization is based on disk striping (or interleaving), which uses a group of disks as one storage unit. Disk striping is a way of increasing the disk transfer rate up to a factor of N , by splitting files across N different disks. Instead of saving all the data from a given file on one disk, it is split across many. Since the N heads can now search independently, the speed of transfer is, in principle, increased manifold. Logical disk data/blocks can be written on two or more separate physical disks which can further transfer their sub-blocks in parallel. The total transfer rate system is directly proportional to the number of disks. The larger the number of physical disks striped together, the larger the total transfer rate of the system. Hence, the overall performance and disk accessing speed is also enhanced. The enhanced version of this scheme is mirroring or shadowing. In this RAID organization a duplicate copy of each disk is kept. It is costly but a much faster and more reliable approach. The disadvantage with disk striping is that, if one of the N disks becomes damaged, then the data on all N disks is lost. Thus, striping needs to be combined with a reliable form of backup in order to be successful.

Another RAID scheme uses some disk space for holding parity blocks. Suppose, three or more disks are used, then one of the disks will act as a parity block, which contains corresponding bit positions in all blocks. In case some error occurs or the disk develops a problem all its data bits can be reconstructed. This technique is known as disk striping with parity or block interleaved parity, which increases speed. But writing or updating any data on a disk requires corresponding recalculations and changes in parity block. To overcome this the parity blocks can be distributed over all disks.

RAID is a method of creating one or more pools of data storage space from several hard drives. It can offer fault tolerance and higher throughput levels than a single hard drive or group of independent hard drives. You can build a RAID configuration with IDE (parallel ATA), SATA (Serial ATA) or SCSI hard disks or, in fact, even drives like the old 3.5" floppy disk drive!

The exact meaning of RAID has been much debated and much argued. The use of "Redundant" is, in itself, a contentious point. That several manufacturers have deviated from accepted RAID terminology, created new levels of disk arrangements, called them RAID, and christened them with a number has not helped. There are even some single disk RAID configurations! Double parity, RAID 1.5, Matrix RAID etc., are examples of proprietary RAID configurations.

Data can be distributed across a RAID "array" using either hardware, software or a combination of the two. Hardware RAID is usually achieved either on-board on some server class motherboards or via an add-on card, using an ISA/PCI slot.

Basic RAID levels are the building blocks of RAID. Compound RAID levels are built by using:

JBOD: JBOD is NOT RAID. JBOD stands for 'Just a Bunch of Disks'. This accurately describes the underlying physical structure that all RAID structures rely upon. When a hardware RAID controller is used, it normally defaults to JBOD configuration for attached disks. Some disk controller manufacturers incorrectly use the term JBOD to refer to a Concatenated array.

Concatenated Array: A Concatenated array is NOT RAID, although it is an array. It is a group of disks connected together, end-to-end, for the purpose of creating a larger logical disk. Although it is not RAID, it is included here as it is the result of early attempts to combine multiple disks into a single logical device. There is no redundancy with a Concatenated array. Any performance improvement over a single disk is achieved because the file-system uses multiple disks. This type of array is usually slower than a RAID-0 array of the same number of disks.

The good point of a Concatenated array is that different sized disks can be used in their entirety. The RAID arrays below require that the disks that make up the RAID array be the same size, or that the size of the smallest disk be used for all the disks. The individual disks in a concatenated array are organized as follows:

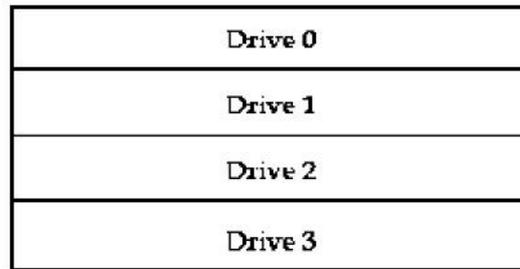


Figure: Concatenated Segments

RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using the idea of disk striping combined with “parity” bits (which we describe next) have been proposed. These schemes have different cost-performance tradeoffs and are classified into levels called **RAID levels**. We describe the various levels here; Figure, shows them pictorially (in the figure, *P* indicates error-correcting bits and *C* indicates a second copy of the data). In all cases depicted in the figure, four disks’ worth of data is stored, and the extra disks are used to store redundant information for failure recovery.

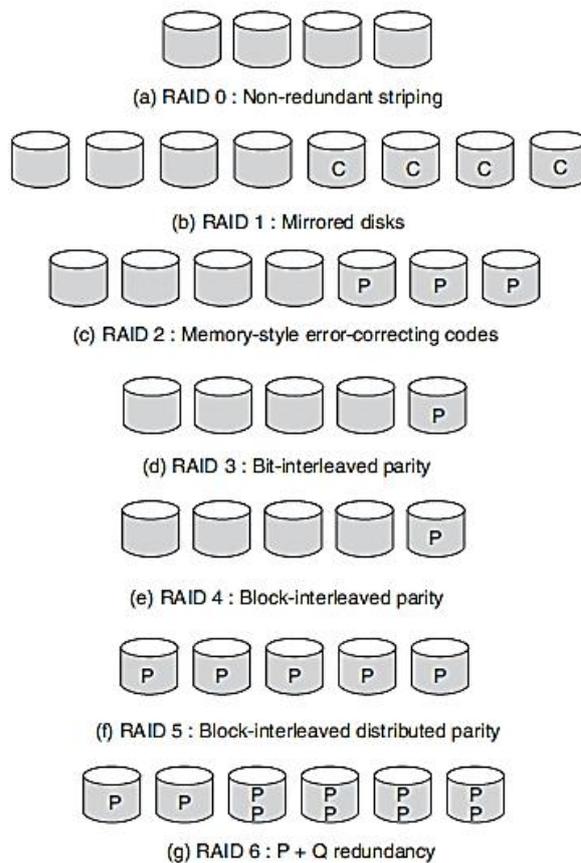


Figure: RAID Levels

RAID-0: In RAID Level 0 (also called striping), each segment is written to a different disk, until all drives in the array have been written to. The I/O performance of a RAID-0 array is significantly better than a single disk. This is true on small I/O requests, as several can be processed simultaneously, and for large requests, as multiple disk drives can become involved in the operation. Spindle-sync will improve the performance for large I/O requests.

This level of RAID is the only one with no redundancy. If one disk in the array fails, data is lost. The individual segments in a 4-wide RAID-0 array is organized as follows:

Drive 0	Drive 1	Drive 2	Drive 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23

Figure: RAID-0 Segments

RAID-1: In RAID Level 1 (also called mirroring), each disk is an exact duplicate of all other disks in the array. When a write is performed, it is sent to all disks in the array. When a read is performed, it is only sent to one disk. This is the least space efficient of the RAID levels. A RAID-1 array normally contains two disk drives. This will give adequate protection against drive failure. It is possible to use more drives in a RAID-1 array, but the overall reliability will not be significantly affected. RAID-1 arrays with multiple mirrors are often used to improve performance in situations where the data on the disks is being read from multiple programs or threads at the same time. By being able to read from the multiple mirrors at the same time, the data throughput is increased, thus improving performance. The most common use of RAID-1 with multiple mirrors is to improve performance of databases.

Spindle-sync will improve the performance of writes, but have virtually no effect on reads. The read performance for RAID-1 will be no worse than the read performance for a single drive. If the RAID controller is intelligent enough to send read requests to alternate disk drives, RAID-1 can significantly improve read performance.

RAID-2: RAID Level 2 is an intellectual curiosity, and has never been widely used. It is more space efficient than RAID-1, but less space efficient than other RAID levels. Instead of using a simple parity to validate the data (as in RAID-3, RAID-4 and RAID-5), it uses a much more complex algorithm, called a Hamming Code. A Hamming code is larger than a parity, so it takes up more disk space, but, with proper code design, is capable of recovering from multiple drives being lost. RAID-2 is the only simple RAID level that can retain data when multiple drives fail.

The primary problem with this RAID level is that the amount of CPU power required to generate the Hamming Code is much higher than is required to generate parity. A RAID-2 array has all the penalties of a RAID-4 array, with an even larger write performance penalty. The reason for the larger write performance penalty is that it is not usually possible to update the Hamming Code. In general, all data blocks in the stripe modified by the write, must be read in, and used to generate new Hamming Code data.

Also, on large writes, the CPU time to generate the Hamming Code is much higher than to generate Parity, thus possibly slowing down even large writes. The individual segments in a 4+2 RAID-2 array are organized as follows:

Drive 0	Drive 1	Drive 2	Drive 3	Drive 4	Drive 5
0	1	2	3	Code	Code
4	5	6	7	Code	Code
8	9	10	11	Code	Code
12	13	14	15	Code	Code
16	17	18	19	Code	Code
20	21	22	23	Code	Code

Figure: RAID-2 Segments

RAID-3: RAID Level 3 is defined as byte-wise (or bit-wise) striping with parity. Every I/O to the array will access all drives in the array, regardless of the type of access (read/write) or the size of the I/O request.

During a write, RAID-3 stores a portion of each block on each data disk. It also computes the parity for the data, and writes it to the parity drive. In some implementations, when the data is read back

in, the parity is also read, and compared to a newly computed parity, to ensure that there were no errors.

RAID-3 provides a similar level of reliability to RAID-4 and RAID-5, but offers much greater I/O bandwidth on small requests. In addition, there is no performance impact when writing. Unfortunately, it is not possible to have multiple operations being performed on the array at the same time, due to the fact that all drives are involved in every operation. As all drives are involved in every operation, the use of spindle-sync will significantly improve the performance of the array. Because a logical block is broken up into several physical blocks, the block size on the disk drive would have to be smaller than the block size of the array. Usually, this causes the disk drive to need to be formatted with a block size smaller than 512 bytes, which decreases the storage capacity of the disk drive slightly, due to the larger number of block headers on the drive. RAID-3 also has configuration limitations. The number of data drives in a RAID-3 configuration must be a power of two. The most common configurations have four or eight data drives. Some disk controllers claim to implement RAID-3, but have a segment size. The concept of segment size is not compatible with RAID-3. If an implementation claims to be RAID-3, and has a segment size, then it is probably RAID-4.

RAID-4: RAID Level 4 is defined as block wise striping with parity. The parity is always written to the same disk drive. This can create a great deal of contention for the parity drive during write operations. For reads, and large writes, RAID-4 performance will be similar to a RAID-0 array containing an equal number of data disks. For small writes, the performance will decrease considerably. To understand the cause for this, a one-block write will be used as an example.

1. A write request for one block is issued by a program.
2. The RAID software determines which disks contain the data, and parity, and which block they are in.
3. The disk controller reads the data block from disk.
4. The disk controller reads the corresponding parity block from disk.
5. The data block just read is XORed with the parity block just read.
6. The data block to be written is XORed with the parity block.
7. The data block and the updated parity block are both written to disk.

It can be seen from the above example that a one block write will result in two blocks being read from disk and two blocks being written to disk. If the data blocks to be read happen to be in a buffer in the RAID controller, the amount of data read from disk could drop to one, or even zero blocks, thus improving the write performance.

The individual segments in a 4+1 RAID-4 array are organized as follows:

Drive 0	Drive 1	Drive 2	Drive 3	Drive 4
0	1	2	3	Parity
4	5	6	7	Parity
8	9	10	11	Parity
12	13	14	15	Parity
16	17	18	19	Parity
20	21	22	23	Parity

Figure: RAID-4 Segments

RAID-5: RAID Level 5 is defined as block wise striping with parity. It differs from RAID-4, in that the parity data is not always written to the same disk drive. RAID-5 has all the performance issues and benefits that RAID-4 has, except as follows:

Since there is no dedicated parity drive, there is no single point where contention will be created. This will speed up multiple small writes. Multiple small reads are slightly faster. This is because data resides on all drives in the array. It is possible to get all drives involved in the read operation.

The above block layout is an example of Linux RAID-5 in left-asymmetric mode.

Summary

- Input is the signal or data received by the system and output is the signal or data sent from it.
- I/O devices are used by a person (or other system) to communicate with a computer.
- Keyboard is the one of the main input devices used on a computer, a PC's keyboard looks very similar to the keyboards of electric typewriters, with some additional keys.
- Mouse is an input device that allows an individual to control a mouse pointer in a graphical user interface (GUI).
- Scanner is a hardware input device that allows a user to take an image and/or text and convert it into a digital file, allowing the computer to read and/or display the scanned object.
- A microphone is a hardware peripheral that allows computer users to input audio into their computers.
- Web Cam is a camera connected to a computer or server that allows anyone connected to the internet to view still pictures or motion video of a user.
- Digital camera is a type of camera that stores the pictures or video it takes in electronic format instead of to film.
- A computer joystick allows an individual to easily navigate an object in a game such as navigating a plane in a flight simulator.
- Monitor is a video display screen and the hard shell that holds it. It is also called a video display terminal (VDT).
- Printer is an external hardware device responsible for taking computer data and generating a hard copy of that data.
- Sound card is a sound card is an expansion card or integrated circuit that provides a computer with the ability to produce sound that can be heard by the user. It is also known as a sound board or an audio card.
- Speaker is a hardware device connected to a computer's sound card that outputs sound generated by the card.

Keywords

- **Bit-level Striping:** Data striping consists of splitting the bits of each byte across multiple disks; such striping is called bit-level striping.
- **Constant Linear Velocity (CLV):** Constant linear velocity (CLV) is a qualifier for the rated speed of an optical disc drive and may also be applied to the writing speed of recordable discs.
- **Data Striping:** The distribution of a unit of data over two or more hard disks, enabling the data to be read more quickly, known as data striping.
- **Error Correcting Code (ECC):** Error correction code is a coding system that incorporates extra parity bits in order to detect errors.
- **Logical Blocks:** The logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes.
- **Logical Formatting:** Logical formatting is the process of placing a file system upon a hard disk drive partition of a hard disk so that an operating system can use available hard disk platter space to store and retrieve files.

- **Low-level Formatted:** The sector identification on a disk that the drive uses to locate sector for reading and writing is called low level formatted.

Self Assessment

1. An online backing storage system capable of storing larger quantities of data is
 - A. CPU
 - B. Memory
 - C. Mass storage
 - D. Secondary storage

2. Which is an item of storage medium in the form of circular plate?
 - A. Disk
 - B. CPU
 - C. Printer
 - D. ALU

3. Which of the following disk scheduling techniques has a drawback of starvation? **Notes**
 - A. SCAN
 - B. SST
 - C. FCFS
 - D. LIFO

4. The total time to prepare a disk drive mechanism for a block of data to be read from is its
 - A. latency
 - B. latency plus transmission time
 - C. latency plus seek time
 - D. latency plus seek time plus transmission time

5. Which among the following are the best tools for fixing errors on disks?
 - A. Fdisk
 - B. Scandisk
 - C. Chkdsk
 - D. Fixdsk

6. Which command can be used to create the disk's tracks and sectors?
 - A. Fdisk
 - B. Format
 - C. Chkdsk
 - D. Attrib

7. Which command is used to create root directory and FAT on disk?
 - A. Chkdsk

- B. Command.com
C. Format
D. Fat
8. is a technique of temporarily removing inactive programs from the memory of computer system.
A. Swapping
B. Spooling
C. Semaphore
D. Scheduler
9. _____ is the time taken in locating the disk arm to a specified track where the read/write request will be satisfied.
A. Latency time
B. Seek time
C. Disk response time
D. Disk access time
10. Which of the following algorithm performs better for systems that place a heavy load on the disk.
A. FCFS
B. SSTF
C. LOOK
D. SCAN
11. Low-level formatting fill the with a special data structure for each sector.
12. System memory used for swap-space is called space.
13. If multiple devices have the same priority, swap-space is allocated from the devices in a fashion.
14. The set of tracks that are at one arm position make up a _____
15. The time taken for the desired sector to rotate to the disk head is called ____

Answers forSelf Assessment

1. C 2. A 3. B 4. C 5. B
6. B 7. C 8. A 9. B 10. D
11. Disk 12. pseudo- 13. round- 14. Cylinders 15. rotational
swap robin latency

Review Questions

1. In what situations would using memory as a RAM disk be more useful than using it as a disk cache?

2. None of the disk-scheduling disciplines, except FCFS, are truly fair (starvation may occur).
 - a. Explain why this assertion is true.
 - b. Describe a way to modify algorithms such as SCAN to ensure fairness.
 - c. Explain why fairness is an important goal in a time-sharing system.
 - d. Give three or more examples of circumstances in which it is important that the operating system be unfair in serving I/O requests.
3. What is the function of a system disk controller?
4. Write short notes on:
 - a) Pseudo swap space
 - b) Device swap space
 - c) File system swap space
5. Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is 86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130. Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?
 - a) FCFS
 - b) SSTF
 - c) SCAN
 - d) LOOK
 - e) C-SCAN
 - f) C-LOOK
6. Is disk scheduling, other than FCFS scheduling, useful in a single-user environment? Explain your answer.
7. Compare the performance of C-SCAN and SCAN scheduling, assuming a uniform distribution of requests. Consider the average response time (the time between the arrival of a request and the completion of that request's service), the variation in response time, and the effective bandwidth. How does performance depend on the relative sizes of seek time and rotational latency?
8. Why is rotational latency usually not considered in disk scheduling? How would you modify SSTF, SCAN, and C-SCAN to include latency optimization?



Further Readings

Andrew M. Lister, *Fundamentals of Operating Systems*, Wiley.

Andrew S. Tanenbaum and Albert S. Woodhull, *Systems Design and Implementation*, Prentice Hall.

Andrew S. Tanenbaum, *Modern Operating System*, Prentice Hall.

Colin Ritchie, *Operating Systems*, BPB Publications.

Deitel H.M., "Operating Systems, 2nd Edition, Addison Wesley.

I.A. Dhotre, *Operating System*, Technical Publications.

Milankovic, *Operating System*, Tata MacGraw Hill, New Delhi.

Silberschatz, Gagne & Galvin, "Operating System Concepts", John Wiley & Sons, Seventh Edition.

Stalling, W., "Operating Systems", 2nd Edition, Prentice Hall.



Web Links

www.en.wikipedia.org

www.web-source.net

www.webopedia.com

LOVELY PROFESSIONAL UNIVERSITY

Jalandhar-Delhi G.T. Road (NH-1)

Phagwara, Punjab (India)-144411

For Enquiry: +91-1824-521360

Fax.: +91-1824-506111

Email: odl@lpu.co.in