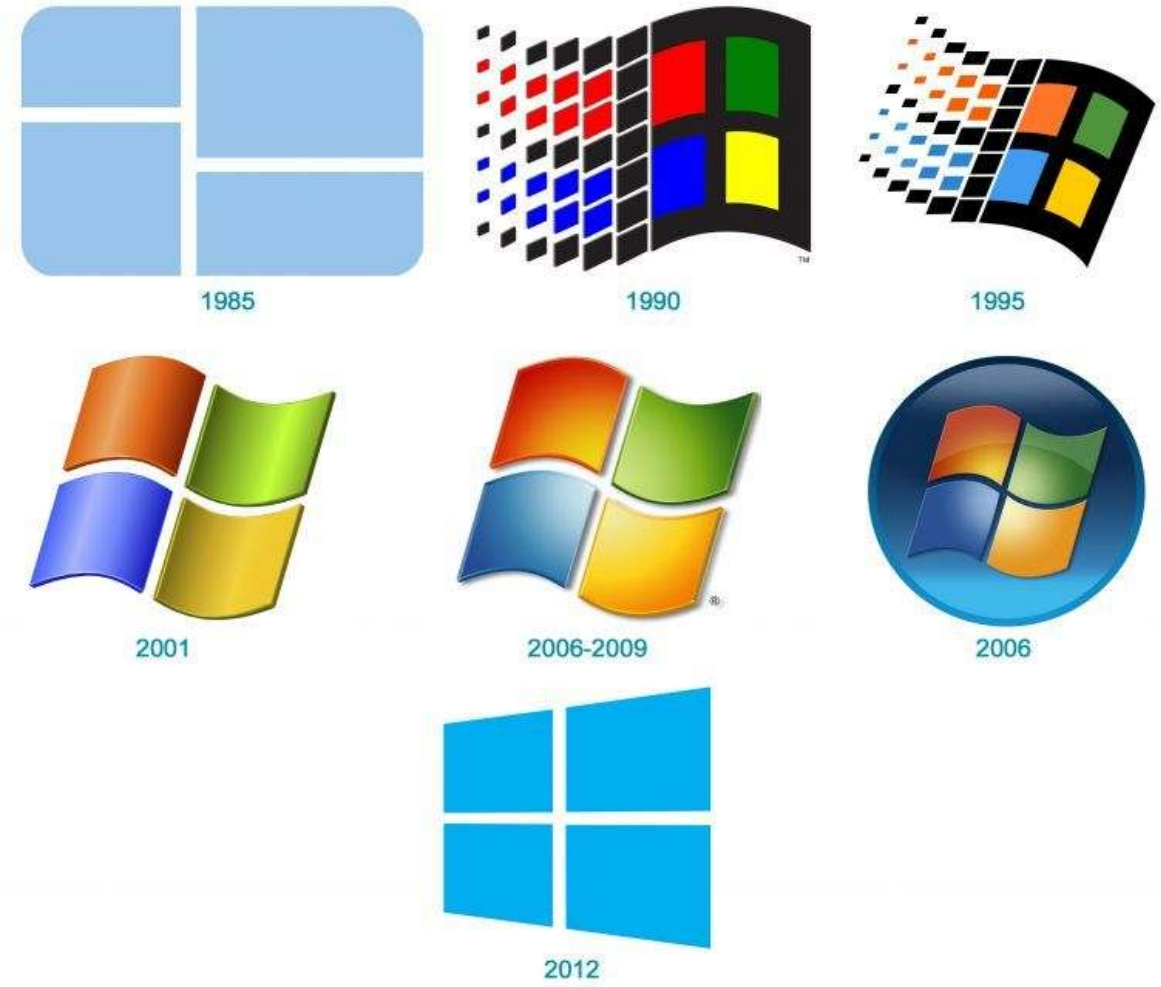
Operating System

CMP-3611



Prepared By:

Muhammad Shahid Azeem

Lecturer CS/IT
MCS, M.Phil (CS)

Dedication

All glories praises and gratitude to Almighty Allah Pak, who blessed us with a super, unequalled processor! Brain…

I dedicate all of my efforts to my students who gave me an urge and inspiration to work more.
I also dedicate this piece of effort to my family members who always support me to move on. Especially, my father (Ch. Ali Muhammad) who pushed me ever and makes me to stand at this position today.

Muhammad Shahid Azeem

Author

Course Outline:

1. Introduction: Over view of: Operating Systems, Operating-System Structure, Operating-System Operations, Process management, Memory Management, Storage Management, Protection and Security, Protection and Security, Distributed Systems, Special-Purpose Systems, Computing Environments. [TB: Ch1]
2. Operating-System Structures: Operating-System Services, Operating-System Structure, User Operating-System Interface, Virtual Machines, System Calls, Operating-System Generation, Types of System Calls, System Boot, System Programs. [TB:Ch2].
3. Processes: Process Concept, Process Scheduling, Operations on Processes, Interprocess Communication, Communication in Client- Server Systems. Threads: Multithreading Models, Thread Libraries, Threading Issues. [TB: Ch. 3, 4]
4. CPU Scheduling: Basic Concepts, Scheduling Criteria, Scheduling Algorithms, Multiple-Processor Scheduling, Thread Scheduling, Algorithm Evaluation. [TB: Ch. 5]
5. Process Synchronization: Background, Monitors, The Critical-Section Problem, Peterson's Solution, Synchronization Hardware, Semaphores, Classic Problems of Synchronization. [TB: Ch. 6]
6. Deadlocks: System Model, Deadlock Characterization, Methods for Handling Deadlocks, Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, Recovery from Deadlock. [TB: Ch. 7]
7. Main Memory: Swapping, Contiguous Memory Allocation, Paging, Structure of the Page Table, Segmentation, Example: The Intel Pentium. [TB: Ch.8]
8. Virtual Memory: Allocating Kernel Memory, Demand Paging, Copy-on-Write, Page Replacement, Allocation of Frames, Thrashing. [TB: Ch. 9]
9. File-System Implementation: File-System Structure, Log-Structured File Systems, File-System Implementation, Directory Implementation, Allocation Methods, Free-Space Management, Efficiency and Performance, Recovery. [TB: Ch. 11]
10. I/O Systems: STREAMS, Hardware, Performance, Application I/O Interface, Kernel I/O Subsystem, Transforming I/O Requests to Bibliographical Notes, Hardware Operations. [TB: Ch. 13]
11. Security: The Security Problem, Computer-Security, Program Threats, Classifications, System and Network Threats, Cryptography as a Security Tool, User Authentication, Implementing Security Defenses, Firewalling to Protect Systems and Networks. [TB: Ch. 15]
12. Case studies: Linux, Windows Operating Systems

This page is left blank intentionally

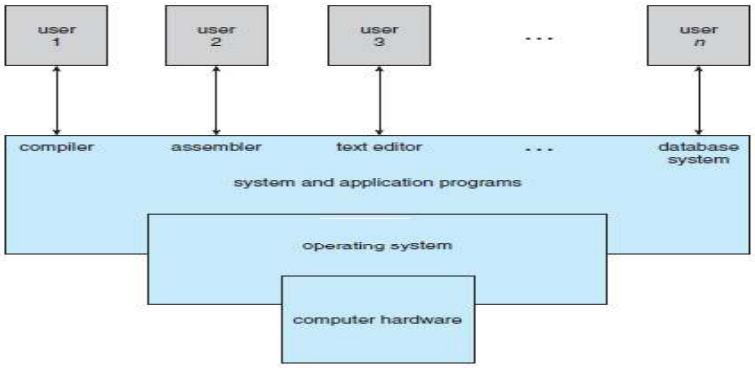
Chapter 01
INTRODUCTION

1.1. Operating System:

An **operating system** is a program that manages a computer's hardware. It also provides a basis for application programs and acts as an intermediary between the computer user and the computer hardware. Various types of OS are there which vary in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for mobile computers provide an environment in which a user can easily interface with the computer to execute programs. Thus, some operating systems are designed to be convenient, others to be efficient, or combination of both. As, an operating system is large and complex, it must be created piece by piece. Each of these pieces should be a well-delineated portion of the system, with carefully defined inputs, outputs, and functions.

The hardware, the central processing unit (CPU), the memory, and the input/output (I/O) devices—provide the basic computing resources for the system. 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. The operating system controls the hardware and coordinates its use among the various application programs for the various users. A computer system consists 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.

Operating System is a mean of interaction between user and computer. It provides interface to monopolized computer resources. As per computer's point of view, the Operating System is a set of programs that manage hardware resources allocation. A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/O devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As, resource allocation is especially important where many users access the same mainframe or minicomputer.



1.1.1 Definition

"Operating system is the software responsible for controlling the allocation and usage of hardware resources such as memory, central processing unit (CPU) time, disk space, and peripheral devices. The operating system is the foundation on which applications, such as word processing and spreadsheet programs, are built. (*Microsoft*)"

1.1.2 Responsibilities of Operating System are

1. Communicating with the computer user: receiving commands and carrying them out or rejecting them with an error message.

2. Managing allocation of memory, of processor time, and of other resources for various tasks.

3. Collecting input from the keyboard, mouse, and other input devices, and providing this data to the currently running program.

4. Conveying program output to the screen, printer, or other output device.

5. Accessing data from secondary storage.

6. Writing data to secondary storage.

1.2. Operating-System Structure

An operating system provides the environment within which programs are executed. We can use different types of operating systems on single processor and multi process systems. A brief description different type of operating system is given below:-

1.2.1. Single user operating system

Single user operating system allows a single user to access the computer resources at a specific time. This type of operating system is mostly used on personal computers like desktop, laptop and palmtop computers. On single user operating system, the CPU remains idle during an I/O operation. So the CPU utilization is reduced. The working of single user operating system is shown in figure 2.5.

Single user operating system is further divided into two classes:-

1. Single user Single tasking operating system
2. Single user multitasking operating system

Single user Single tasking operating system

The single user single tasking operating system allows a single user to execute one program at a time. MS.DOC is an example of this kind of operating system.

Single user multitasking operating system

In multitasking, more than one program can be executed at a time on a single processer. Single user multitasking operating system allows a single user to execute multiple programs at the same time. The Windows and Mac. OS are examples of single user multitasking operating system. For example, in MS Windows we can load multiple programs like MS-EXCEL, MS word and MS Access along with listening music in the background.

1.2.2. Batch operating system

In early computer systems, the user did not interact directly with the computers system. The data and programs were first prepared on the input media such as punched cards or punched tape. The data and programs prepared on the punched tape or punched cards were

referred to as jobs. These jobs were submitted to the computers operator at specific offices. The computer operator would arrange the jobs into proper sequence known as batches and run the batches through the computer. The batch operating system was used to manage and control the jobs. The working of a single batch operating system is shown in figure 2.6.

The simple batch operating system transfers the jobs to the process one by one. When a job is completed, the control is transferred to next job, for example, if first job is about to print a document on a printer and second job is to execute a program for creating and editing text document. In this case, on the completion of first job, the second job can be started.

The first batch operating system was developed in the mid-1950s by General Motors for IBM 701 computers. This system was revised and then implemented on the IBM 704 computers. By the early 1960s, a number of vendors had developed batch processing systems for their computers but the most popular batch operating system was "IBSYS" of IBM. This operating system was developed for the IBM 7090/7094 computers.

1.2.3. Multi-Programmed Batch System

In multiprogramming environment, multiple programs of different users can be executed simultaneously. The multiple jobs that are to be executed must be kept in main memory and the operating system must execute them in parallel fashion i.e. CPU switches between the jobs so fast that the jobs seem to be executing simultaneously. Before starting execution, operating system will decide which job to run first, which one to second and so on. This decision is called CPU Scheduling and will be discussed in next chapters.

In multi-programmed batch system, the operating system keeps multiple jobs in main memory at a time, since; in general, main memory is limited space and may not accommodate all the jobs to be executed. So the jobs that enter the system to be executed are kept initially on the disk in the job pool. In other words, we can say that a job pool consists of all the jobs residing on the disk waiting allocation of main memory. When the operating system selects a job from a job pool, it loads that job into memory for execution.

Normally, the jobs in main memory are kept in smaller size than the jobs in job pool. The jobs in job pool wait for allocation of main memory. If several jobs are ready to be brought into memory, and there is not enough room for all the them, then the system needs memory management through operating system. Similarly, if many jobs are ready to run at the same time, the system must schedule the turns of these jobs according to some scheduling algorithm.

Processer picks and begins to execute one of the jobs in main memory. During execution, some jobs may need to wait for certain tasks (such as I/O operation.) In a simple batch system or non-multi-programmed system, the processor would sit idle. However, in multi-programmed system, the CPU switches to second job and begins to execute it. Similarly, when second job needs to wait, the processor is switched to third job, and so on. Operating system also checks the status of previous jobs, whether they are completed or not.

The multi-programmed system takes comparatively less time to complete the jobs than the simple batch system. Please note that the multi-programmed system do not allow interaction between the processes when they are running on the computers.

Multiprogramming increases the CPU's utilization. Multi-programmed system provides an environment in which various computer resources are utilized effectively. The CPU always remains busy to run one of the jobs until and jobs complete their execution.

In multi-programmed system, the hardware must have the facilities to support multiprogramming.

1.2.4. Timesharing Operating System

Timesharing system is a multiprogramming, multiprocessing and interactive system. It allows multiple users to share the computer at the same time. This system executes multiple jobs of users by switching among them. Timesharing is used when multiple users are connected to a single computer in a communication network. Each user accesses the computer with its own terminal.

Like multiprogramming, timesharing operating system also user the CPU scheduling. However in timesharing system, each user is assigned a small time unit known as time slice. The job of a user is tried to execute within its time slice. If the job's execution completes within the given time slice, it is removed from the job queue. However, in case the job needs more time for its execution then it is moved to ready state. During execution if job needs I/O operation during its time slice, then it is moved to Blocked state for completion of I/O operation. After completion of I/O operation, jobs are moved to Ready State from Blocked State. From Ready State, the job is again dispatched to CPU for remaining execution. This process continues in a cycle. Thus at a regular time intervals, some users may logout from the system, while new users may login into the system.

The processor switches between the jobs so rapidly that each user feels that the entire computer system is dedicated to his job only. In timesharing system, the users can interact with their jobs, while these are running.

In timesharing system (like multiprogramming system), multiple jobs are also simultaneously loaded in main memory. The main memory may not accommodate all these jobs at the same time. In this case, the jobs are kept on the disk in the job pool. The jobs in job pool await allocation of main memory. If several jobs are ready to be brought into memory, and if there is not enough room for all of them, then the system must require memory management. Similarly, if many jobs are ready to run at the same time, the system must schedule these jobs accordingly. The time-sharing systems must also provide a file system management to manage the input and output data of the multiple users.

Timesharing system (and multiprogramming system) also creates challenges for the operating system. If there are multiple jobs in memory, then they must be protected from interfering with each other such that there should not be undesirable change in any of the jobs.

One of the first timesharing operating system was the Compatible Time-Sharing System (CTSS).This operating system was first developed for the IBM 709 in 1961 and later transferred to IBM 7094.Nowadays examples of important timesharing operating systems are UNIX, Linux, Windows NT Server and Windows 2000 Server.

1.2.5. Real-Time Operating System

The real-time operating system is designed to support executions of programs within strict time limits. The real-time system requires the correct results within the specified time period. The correctness of the program's result is totally dependent on the completion of the job within the specified period. Many real-time systems are embedded in specialized devices, such as microwave ovens, washing machines, digital cameras, cellular telephones.

For example, human brain works on the principle of real time operating system. If any hot object touches our finger, the brain directs the finger to quickly move it away from the hot object, thus the removal of finger completes within the required time. Real-time operating systems are used in medical imaging system, airline reservation system, telecommunication system; military weapons control systems, and many other laboratory experiments etc.

1.2.5.1. Types of Real-Time Systems

There are two types of real time systems.

i.Hard real-time systems
ii.Soft real-time systems

i. Hard Real-Time Systems

A hard real-time system is one that must meet its deadline. The tasks must be completed within their deadline, otherwise real-time system will fail or breakdown. The examples of hard real-time systems are as follows:

- Military weapons control systems like missile systems.
- Flight management system like airplanes operations.
- Health control systems used in ICU in hospitals.

ii. Soft Real-Time Systems

In soft real-time system, the tasks do have associated deadlines. But this system is not as restrictive as hard real-time system. The soft real-time system does not guarantee that a task or job will be completed within a specified time period. This system re-schedules and completes the job if it has passed its deadline.

In real-time system, the operating system must assign the priority to the real time tasks over other tasks and they must retain their priority until they complete.

Today, many operating systems are soft real-time. The Linux is also included in these operating systems. Soft real-time systems are used in multimedia environments. For example, if a DVD player cannot process a frame of video, still we can continue watching the video.

1.2.6. Distributed Computer System

A computer system, in which different computing resources like processors, printers, storage area, data etc. are physically distributed in an organization through a networked system is called distributed computer system, the organization may spread over different places, cities and even countries around the world. The resources of the distributed system are networked to provide access to its users in the system. Distributed system depends on networking for their functionality. The network may be LAN or WAN. Different networks use different protocols. The most commonly used network protocol is TCP/IP, which is supported by most of the operating system such as Windows and UNIX.

The ultimate goal of distributed computer system is to maximize performance by connecting users and IT resources in a cost-effective, transparent and reliable manner. It also ensures fault tolerance and enables resource accessibility in the event that one of the components fails.

In distributed system two types of operating systems are used.

i.Network operating system.
ii.Distributed operating system.

1.2.6.1. Network Operating System

A network is a collection of one or more servers, workstations and printers etc. For users to access these resources. The server provides the network services or applications, such as file storage and management. Each computer or node has its own private operating system. With a network operating system, the resources on each machine on the network are managed by that machine's operating system. The network operating system is simply an addition of local operating system that allows application machines to interact with server machines.

1.2.6.2. Distributed Operating System

In a distributed operating system, the users access remote resources in the same way they access local resources. The requests of users are carried out independently on more than one location. The computers (or nodes) communicate with each other under the control of distributed operating system. With a distributed operating system, the operating systems on

all the machines work together to manage the collective network resources. A single collective distributed operating system manages the network resources. Provided by each node of distributed system.

1.2.7. Handheld Systems

A portable computer that is small enough to be held in one's hand is called handheld computer. Cellular telephones, pocket-PCs personal digital assistants (PDAs) are examples of handheld system. They have the same features as general-purpose digital computers. These systems use special-purpose embedded operating systems. The memory of these devices normally ranges between 512 KB to 128 MB. The operating system and application must manage the memory efficiently.

1.2.8. Embedded Operating Systems

Specialized electronic devices are controlled by special built-in operating systems, called embedded operating systems. This type of operating system is permanently stored into the ROM chips and that's why it is known as embedded operating system. Embedded operating systems are mainly used in mobile devices (or handheld computers) such as mobile phones, microwave ovens and TV sets etc. Examples of embedded operating systems are **palm OS and windows CE.**

1.3. Operating-System Operations:

An **interrupt** is a signal generated by a hardware device (usually an I/O device) to get CPU's attention. Interrupt transfers control to the **interrupt service routine** (ISR), generally through the **interrupt vector table**, which contains the addresses of all the service routines. The interrupt service routine executes; on completion the CPU resumes the interrupted computation. Interrupt architecture must save the address of the interrupted instruction. Incoming interrupts are disabled while another interrupt is being processed to prevent a *lost interrupt*. An operating system is an interrupt driven software.

A **trap** (or an *exception)* is a software-generated interrupt caused either by an error (division by zero or invalid memory access) or by a user request for an operating system service.

A **signal** is an event generated to get attention of a process. An example of a signal is the event that is generated when you run a program and then press <Ctrl-C>. The signal generated in this case is called SIGINT (Interrupt signal). Three actions are possible on a signal:

1. Kernel-defined default action—which usually results in process termination and, in some cases, generation of a 'core' file that can be used the programmer/user to know the state of the process at the time of its termination.
2. Process can intercept the signal and ignore it.
3. Process can intercept the signal and take a programmer-defined action.

We will discuss signals in detail in some of the subsequent lectures.

1.3.1. Hardware Protection

Multi-programming put several programs in memory at the same time; while this increased system utilization it also increased problems. With sharing, many processes could be adversely affected by a bug in one program. One erroneous program could also modify the program or data of another program or even the resident part of the operating system. A file may overwrite another file or folder on disk. A process may get the CPU and

never relinquish it. So the issues of hardware protection are: I/O protection, memory protection, and CPU protection. We will discuss them one by one, but first we talk about the dual-mode operation of a CPU.

a) Dual Mode Operation

To ensure proper operation, we must protect the operating system and all other programs and their data from any malfunctioning program. Protection is needed for any shared resources. Instruction set of a modern CPU has two kinds of instructions, privileged instructions and non-privileged instructions. Privileged instructions can be used to perform hardware operations that a normal user process should not be able to perform, such as communicating with I/O devices. If a user process tries to execute a privileged instruction, a trap should be generated and process should be terminated prematurely. At the same time, a piece of operating system code should be allowed to execute privileged instructions. In order for the CPU to be able to differentiate between a user process and an operating system code, we need two separate modes of operation: user mode and monitor mode (also called supervisor mode, system mode, or privileged mode). A bit, called the **mode bit**, is added to the hardware of the computer to indicate the current mode: **monitor mode** (0) or **user mode** (1). With the mode bit we are able to distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

The concept of privileged instructions also provides us with the means for the user to interact with the operating system by asking it to perform some designated tasks that only the operating system should do. A user process can request the operating system to perform such tasks for it by executing a **system call**. Whenever a system call is made or an interrupt, trap, or signal is generated, CPU mode is switched to system mode before the relevant kernel code executes. The CPU mode is switched back to user mode before the control is transferred back to the user process.

b) I/O Protection

A user process may disrupt the normal operation of the system by issuing illegal I/O instructions, by accessing memory locations within the operating system itself, or by refusing to relinquish the CPU. We can use various mechanisms to ensure that such disruptions cannot take place in the system. To prevent users from performing illegal I/O, we define all I/O instructions to be privileged instructions. Thus users cannot issue I/O instructions directly; they must do it through the operating system. For I/O protection to be complete, we must be sure that a user program can never gain control of the computer in monitor mode. If it could, I/O protection could be compromised. Consider a computer executing in user mode. It will switch to monitor mode whenever an interrupt or trap occurs, jumping to the address determined from the interrupt from the interrupt vector. If a user program, as part of its execution, stores a new address in the interrupt vector, this new address could overwrite the previous address with an address in the user program. Then, when a corresponding trap or interrupt occurred, the hardware would switch to monitor mode and transfer control through the modified interrupt vector table to a user program, causing it to gain control of the computer in monitor mode. Hence we need all I/O instructions and instructions for changing the contents of the system space in memory to be protected. A user

process could request a privileged operation by executing a system call such as read (for reading a file).

1.3.2. Process Management:

A Program in execution called Process. For example a word processing software in running is a process. A system task, like sending output to the O/P device is also a process. A process needs certain resources—including CPU time, memory, files, and I/O devices—to accomplish its task. These resources are either given to the process when it is created or allocated to it while it is running. In addition to the various physical and logical resources that a process obtains when it is created, various initialization data (input) may be passed along. For example, consider a process whose function is to display the status of a file on the screen of a terminal. The process will be given the name of the file as an input and will execute the appropriate instructions and system calls to obtain and display the desired information on the terminal. When the process terminates, the operating system will reclaim any reusable resources. A program by itself is not a process. A program is a passive entity, like the contents of a file stored on disk, whereas a process is an active entity.

A process is the unit of work in a system. A system consists of a collection of processes, some of which are operating-system processes (those that execute system code) and the rest of which are user processes (those that execute user code). All these processes can potentially execute concurrently by multiplexing on a single CPU.

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

- Scheduling processes and threads on the CPUs
- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication.

1.3.3. Memory Management

The main memory is central to the operation of a modern computer system. Main memory is a large array of bytes, ranging in size from hundreds of thousands to billions. Each byte has its own address.

Main memory is a repository of quickly accessible data shared by the CPU and I/O devices. The central processor reads instructions from main memory during the instruction-fetch cycle and both reads and writes data from main memory during the data-fetch cycle (on a von Neumann architecture). The main memory is generally the only large storage device that the CPU is able to address and access directly. For example, for the CPU to process data from disk, those data must first be transferred to main memory by CPU-generated I/O calls. In the same way, instructions must be in memory for the CPU to execute them.

For a program to be executed, it must be mapped to absolute addresses and loaded into memory. As the program executes, it accesses program instructions and data from memory by generating these absolute addresses. Eventually, the program terminates, its memory space is declared available, and the next program can be loaded and executed. To improve both the utilization of the CPU and the speed of the computer's response to its users,

general-purpose computers must keep several programs in memory, creating a need for memory management. Many different memory management schemes are used. These schemes reflect various approaches, and the effectiveness of any given algorithm depends on the situation. In selecting a memory-management scheme for a specific system, we must take into account many factors—especially 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:

- Keeping track of free memory space
- Keeping track of which parts of memory are currently being used and by whom
- Deciding which processes are to be loaded into memory when memory space becomes available
- Deciding how much memory is to be allocated to a process
- Allocating and de-allocating memory space as needed
- Insuring that a process is not overwritten on top of another

1.3.4. Storage Management:

To make the computer system convenient for users, 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. The operating system maps files onto physical media and accesses these files via the storage devices.

1.3.4.1. Secondary Storage Management:

The main purpose of a computer system is to execute programs. The programs, along with the data they access, must be in the main memory or primary storage during their execution. Since main memory is too small to accommodate all data and programs, and because the data it holds are lost when the power is lost, the computer system must provide secondary storage to backup main memory. Most programs are stored on a disk until loaded into the memory and then use disk as both the source and destination of their processing. Like all other resources in a computer system, proper management of disk storage is important.

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

- Free-space management
- Storage allocation and de-allocation
- Disk scheduling

1.3.5. File management:

File management is one of the most visible components of an operating system. Computers can store information on several different types of physical media. I.e. Magnetic disk, optical disk, and magnetic tape

Data is arranged in form of file on these storage media. A file is a collection of related information defined by its creator. The operating system implements the abstract concept of a file by managing mass-storage media and files are normally organized into

directories to make them easier to use. When multiple users have access to files, it is desirable to control which user may access a file and how that user may access it (for example, read, write, append).

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

- Creating and deleting files
- Creating and deleting directories to organize files
- Supporting primitives for manipulating files and directories
- Mapping files onto secondary storage
- Backing up files on stable (nonvolatile) storage media.

1.3.6. Protection and Security

If a computer system has multiple users and allows the concurrent execution of multiple processes, then access to data must be regulated. For that purpose mechanisms ensure that files, memory segments, CPU, and other resources can be operated on by only those processes that have gained proper authorization from the operating system. For example, memory-addressing hardware ensures that a process can execute only within its own address space. The timer ensures that no process can gain control of the CPU without eventually relinquishing control.

1.3.6.1. Protection:

Protection is any mechanism for controlling the access of processes or users to the resources defined by a computer system. This mechanism must provide means to specify the controls to be imposed and to enforce the controls. Protection can improve reliability by detecting dormant errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by another subsystem that is malfunctioning. Furthermore, an unprotected resource cannot defend against use (or misuse) by an unauthorized user. A protection-oriented system provides a means to distinguish.

1.3.6.2. Security:

Security is a mechanism to defend a system from external and internal attacks. Such as viruses and worms, denial-of-service attacks, identity theft, and theft of service. Prevention of some of these attacks is considered an operating-system function on some systems, while other systems leave it to policy or additional software.

1.4. Computing Environments

1.4.1. Traditional Computing

Today, identification of traditional computing is very difficult due to emerging advancement in computation technology. Just a few years ago, office computation environment consisted of PCs connected to a network, with servers providing file and print services. Remote access was awkward, and portability was achieved by use of laptop computers. Terminals attached to mainframes were prevalent at many companies as well, with even fewer remote access and portability options.

The current trend is toward providing more ways to access these computing environments. Web technologies and increasing WAN bandwidth are stretching the boundaries of traditional computing. Companies establish **portals**, which provide Web

accessibility to their internal servers. **Network computers** (or **thin clients**)—which are essentially terminals that understand web-based computing—are used in place of traditional workstations where more security or easier maintenance is desired. Mobile computers can synchronize with PCs to allow very portable use of company information. Mobile computers can also connect to **wireless networks** and cellular data networks to use the company's Web portal (as well as the myriad other Web resources). At home, most users once had a single computer with a slow modem connection to the office, the Internet, or both. Today, network-connection speeds once available only at great cost are relatively inexpensive in many places, giving home users more access to more data. These fast data connections are allowing home computers to serve up Web pages and to run networks that include printers, client PCs, and servers. Many homes use **firewalls** to protect their networks from security breaches.

In the latter half of the 20th century, computing resources were relatively limited. For a period of time, systems were either batch or interactive. Batch systems processed jobs in bulk, with predetermined input from files or other data sources. Interactive systems waited for input from users. To optimize the use of the computing resources, multiple users shared time on these systems. Time-sharing systems used a timer and scheduling algorithms to cycle processes rapidly through the CPU, giving each user a share of the resources. Today, traditional time-sharing systems are uncommon. The same scheduling technique is still in use on desktop computers, laptops, servers, and even mobile computers, but frequently all the processes are owned by the same user (or a single user and the operating system). User processes, and system processes that provide services to the user, are managed so that each frequently gets a slice of computer time. Consider the windows created while a user is working on a PC, for example, and the fact that they may be performing different tasks at the same time. Even a web browser can be composed of multiple processes, one for each website currently being visited, with time sharing applied to each web browser process.

1.4.2. Mobile Computing

Mobile computing refers to computing on handheld smart phones and tablet computers. These devices are portable and lightweight. Historically, compared with desktop and laptop computers, mobile systems gave up screen size, memory capacity, and overall functionality in return for handheld mobile access to services such as e-mail and web browsing. Over the past few years, however, features on mobile devices have become so rich that the distinction in functionality between, a laptop and a tablet computer may be difficult to determine. In fact, some features of a modern mobile device allow it to provide functionality that is either unavailable or impractical on a desktop or laptop computer.

Today, mobile systems are used not only for e-mail and web browsing but also for playing music and video, reading digital books, taking photos, and recording high-definition video. Accordingly, tremendous growth continues in the wide range of applications that run on such devices. Many developers are now designing applications that take advantage of the unique features of mobile devices, such as global positioning system (GPS) chips, accelerometers, and gyroscopes. An embedded GPS chip allows a mobile device to use satellites to determine its precise location on earth. That functionality is especially useful in

designing applications that provide navigation—for example, telling users which way to walk or drive or perhaps directing them to nearby services, such as restaurants. An accelerometer allows a mobile device to detect its orientation with respect to the ground and to detect certain other forces, such as tilting and shaking. In several computer games that employ accelerometers, players interface with the system not by using a mouse or a keyboard but rather by tilting, rotating, and shaking the mobile device! Perhaps more a practical use of these features is found in ***augmented-reality*** applications, which overlay information on a display of the current environment. It is difficult to imagine how equivalent applications could be developed on traditional laptop or desktop computer systems.

To provide access to on-line services, mobile devices typically use either IEEE standard 802.11 wireless or cellular data networks. The memory capacity and processing speed of mobile devices, however, are more limited than those of PCs. whereas a smart-phone or tablet may have 64 GB in storage; it is not uncommon to find 1 TB in storage on a desktop computer. Similarly, because power consumption is such a concern, mobile devices often use processors that are smaller, are slower, and offer fewer processing cores than processors found on traditional desktop and laptop computers. Two operating systems currently dominate mobile computing: **Apple iOS** and **Google Android**. iOS was designed to run on Apple iPhone and iPad mobile devices. Android powers smart phones and tablet computers available from many manufacturers.

1.4.3. Distributed Systems

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide users with access to the various resources that the system maintains. Access to a shared resource increases computation speed, functionality, data availability, and reliability.

Some operating systems generalize network access as a form of file access, with the details of networking contained in the network interface's device driver. Others make users specifically invoke network functions. Generally, systems contain a mix of the two modes—for example FTP and NFS. The protocols that create a distributed system can greatly affect that system's utility and popularity.

A **network**, in the simplest terms, is a communication path between two or more systems. Distributed systems depend on networking for their functionality. Networks vary by the protocols used, the distances between nodes, and the transport media. **TCP/IP** is the most common network protocol, and it provides the fundamental architecture of the Internet. Most operating systems support TCP/IP, including all general-purpose ones. Some systems support proprietary protocols to suit their needs. To an operating system, a network protocol simply needs an interface device—a network adapter, for example—with a device driver to manage it, as well as software to handle data.

Networks are characterized based on the distances between their nodes. A **local-area network (LAN)** connects computers within a room, a building, or a campus. A **wide-area network (WAN)** usually links buildings, cities, or countries. A global company may have a WAN to connect its offices worldwide, for example. These networks may run one protocol or several protocols. The continuing advent of new technologies brings about new forms of networks.

For example, a **metropolitan-area network (MAN)** could link buildings within a city. BlueTooth and 802.11 devices use wireless technology to communicate over a distance of several feet, in essence creating a **personal-area network (PAN)** between a phone and a headset or a smart-phone and a desktop computer. The media to carry networks are equally varied. They include copper wires, fiber strands, and wireless transmissions between satellites, microwave dishes, and radios. When computing devices are connected to cellular phones, they create a network. Even very short-range infrared communication can be used for networking. At a rudimentary level, whenever computers communicate, they use or create a network. These networks also vary in their performance and reliability.

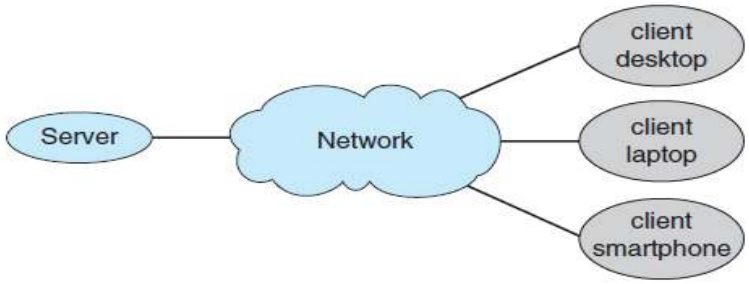
Some operating systems have taken the concept of networks and distributed systems further than the notion of providing network connectivity. A **network operating system** is an operating system that provides features such as file sharing across the network, along with a communication scheme that allows different processes on different computers to exchange messages. A computer running a network operating system acts autonomously from all other computers on the network, although it is aware of the network and is able to communicate with other networked computers. A distributed operating system provides a less autonomous environment. The different computers communicate closely enough to provide the illusion that only a single operating system controls the network.

1.4.4. Client–Server Computing

As PCs have become faster, more powerful, and cheaper, designers have shifted away from centralized system architecture. Terminals connected to centralized systems are now being supplanted by PCs and mobile devices. Correspondingly, user-interface functionality once handled directly by centralized systems is increasingly being handled by PCs, quite often through a web interface. As a result, many of today's systems act as **server systems** to satisfy requests generated by **client systems**. This form of specialized distributed system, called a **client–server** system.

Server systems can be broadly categorized as compute servers and file servers:

- The **compute-server system** provides an interface to which a client can send a request to perform an action (for example, read data). In response, the server executes the action and sends the results to the client. A server running a database that responds to client requests for data is an example of such a system.
- The **file-server system** provides a file-system interface where clients can create, update, read, and delete files. An example of such a system is a web server that delivers files to clients running web browsers.



1.4.5. Peer-to-Peer Computing

Another structure for a distributed system is the peer-to-peer (P2P) system model. In this model, clients and servers are not distinguished from one another. Instead, all nodes within the system are considered peers, and each may act as either a client or a server, depending on whether it is requesting or providing a service. Peer-to-peer systems offer an advantage over traditional client-server systems. In a client-server system, the server is a bottleneck; but in a peer-to-peer system, services can be provided by several nodes distributed throughout the network.

To participate in a peer-to-peer system, a node must first join the network of peers. Once a node has joined the network, it can begin providing services to—and requesting services from—other nodes in the network. Determining what services are available is accomplished in one of two general ways:

- When a node joins a network, it registers its service with a centralized lookup service on the network. Any node desiring a specific service first contacts this centralized lookup service to determine which node provides the service. The remainder of the communication takes place between the client and the service provider.
- An alternative scheme uses no centralized lookup service. Instead, a peer acting as a client must discover what node provides a desired service by broadcasting a request for the service to all other nodes in the network. The node (or nodes) providing that service responds to the peer making the request. To support this approach, a ***discovery protocol*** must be provided that allows peers to discover services provided by other peers in the network.

Peer-to-peer networks gained widespread popularity in the late 1990s with several file-sharing services, such as Napster and Gnutella that enabled peers to exchange files with one another. The Napster system used an approach similar: a centralized server maintained an index of all files stored on peer nodes in the Napster network, and the actual exchange of files took place between the peer nodes. The Gnutella system used a technique similar to the second type: a client broadcasted file requests to other nodes in the system, and nodes that could service the request responded directly to the client. The future of exchanging files remains uncertain because peer-to-peer networks can be used to exchange copyrighted materials anonymously, and there are laws governing the distribution of copyrighted material. Notably, Napster ran into legal trouble for copyright infringement and its services were shut down in 2001. Skype is another example of peer-to-peer computing. It allows clients to make voice calls and video calls and to send text messages over the Internet using a technology known as **voice over IP (VoIP)**. Skype uses a hybrid peer to- peer approach. It includes a centralized login server, but it also incorporates decentralized peers and allows two peers to communicate.

1.4.6. Virtualization

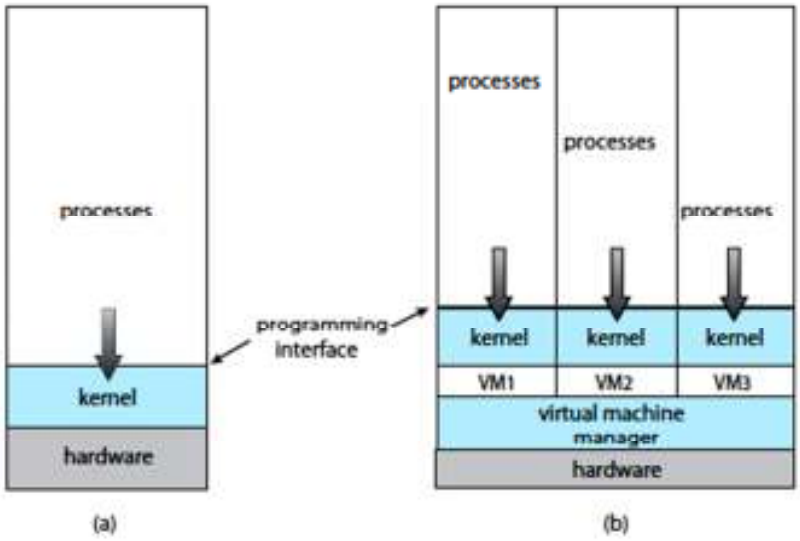
Virtualization is a technology that allows operating systems to run as applications within other operating systems. At first blush, there seems to be no reason for such functionality. But the virtualization industry is vast and growing, which is a testament to its utility and importance.

Virtualization is one member of a class of software that also includes emulation. **Emulation** is used when the source CPU type is different from the target CPU type. For example, when Apple switched from the IBM Power CPU to the Intel x86 CPU for its

desktop and laptop computers, it included an emulation facility called "Rosetta," which allowed applications compiled for the IBM CPU to run on the Intel CPU. That same concept can be extended to allow an entire operating system written for one platform to run on another. Emulation comes at a heavy price, however. Every machine-level instruction that runs natively on the source system must be translated to the equivalent function on the target system, frequently resulting in several target instructions. If the source and target CPUs have similar performance levels, the emulated code can run much slower than the native code.

A common example of emulation occurs when a computer language is not compiled to native code but instead is either executed in its high-level form or translated to an intermediate form. This is known as **interpretation**. Some languages, such as BASIC, can be either compiled or interpreted. Java, in contrast, is always interpreted. Interpretation is a form of emulation in that the high-level language code is translated to native CPU instructions, emulating not another CPU but a theoretical virtual machine on which that language could run natively. Thus, we can run Java programs on "Java virtual machines," but technically those virtual machines are Java emulators. With **virtualization**, in contrast, an operating system that is natively compiled for a particular CPU architecture runs within another operating system also native to that CPU. Virtualization first came about on IBM mainframes as a method for multiple users to run tasks concurrently. Running multiple virtual machines allowed (and still allows) many users to run tasks on a system designed for a single user. Later, in response to problems with running multiple Microsoft Windows XP applications on the Intel x86 CPU, VMware created a new virtualization technology in the form of an application that ran on XP.

That application ran one or more **guest** copies of Windows or other native x86 operating systems, each running its own applications.



VMWare

Windows was the **host** operating system, and the VMware application was the virtual machine manager VMM. The VMM runs the guest operating systems, manages their resource use, and protects each guest from the others. Even though modern operating systems are fully capable of running multiple applications reliably, the use of virtualization continues

to grow. On laptops and desktops, a VMM allows the user to install multiple operating systems for exploration or to run applications written for operating systems other than the native host. For example, an Apple laptop running Mac OSX on the x86 CPU can run a Windows guest to allow execution of Windows applications. Companies writing software for multiple operating systems can use virtualization to run all of those operating systems on a single physical server for development, testing, and debugging. Within data centers, virtualization has become a common method of executing and managing computing environments. VMMs like VMware, ESX, and Citrix XenServer no longer run on host operating systems but rather ***are*** the hosts.

1.4.7. Cloud Computing

Cloud computing is a type of computing that delivers computing, storage, and even applications as a service across a network. In some ways, it's a logical extension of virtualization, because it uses virtualization as a base for its functionality. For example, the Amazon Elastic Compute Cloud **(EC2)** facility has thousands of servers, millions of virtual machines, and petabytes of storage available for use by anyone on the Internet. Users pay per month based on how much of those resources they use.

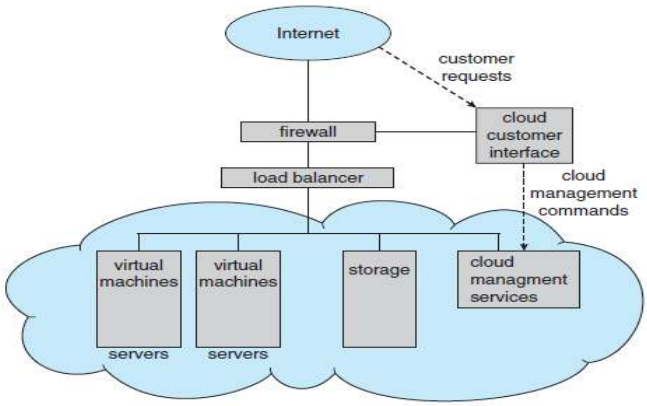
There are actually many types of cloud computing, including the following:

- **Public cloud**—a cloud available via the Internet to anyone willing to pay for the services
- **Private cloud**—a cloud run by a company for that company's own use
- **Hybrid cloud**—a cloud that includes both public and private cloud components
- Software as a service **(SaaS)**—one or more applications (such as word processors or spreadsheets) available via the Internet
- Platform as a service **(PaaS)**—a software stack ready for application use via the Internet (for example, a database server)
- Infrastructure as a service **(IaaS)**—servers or storage available over the Internet (for example, storage available for making backup copies of production data)

These cloud-computing types are not discrete, as a cloud computing environment may provide a combination of several types. For example, an organization may provide both SaaS and IaaS as a publicly available service.

Certainly, there are traditional operating systems within many of the types of cloud infrastructure. Beyond those are the VMMs that manage the virtual machines in which the user processes run. At a higher level, the VMMs themselves are managed by cloud management tools, such as Vware vCloud Director and the open-source Eucalyptus toolset. These tools manage the resources within a given cloud and provide interfaces to the cloud components, making a good argument for considering them a new type of operating system.

Following Figure illustrates a public cloud providing IaaS. Notice that both the cloud services and the cloud user interface are protected by a firewall.



Cloud computing.

1.4.8. Real-Time Embedded Systems

Embedded computers are the most prevalent form of computers in existence. These devices are found everywhere, from car engines and manufacturing robots to DVDs and microwave ovens. They tend to have very specific tasks. The systems they run on are usually primitive, and so the operating systems provide limited features. Usually, they have little or no user interface, preferring to spend their time monitoring and managing hardware devices, such as automobile engines and robotic arms.

These embedded systems vary considerable. Some are general-purpose computers, running standard operating systems—such as Linux—with special-purpose applications to implement the functionality. Others are hardware devices with a special-purpose embedded operating system providing just the functionality desired. Yet others are hardware devices with application specific integrated circuits (**ASICs**) that perform their tasks without an operating system.

The use of embedded systems continues to expand. The power of these devices, both as standalone units and as elements of networks and the web, is sure to increase as well. Even now, entire houses can be computerized, so that a central computer—either a general-purpose computer or an embedded system—can control heating and lighting, alarm systems, and even coffee makers. Web access can enable a home owner to tell the house to heat up before she arrives home. Someday, the refrigerator can notify the grocery store when it notices the milk is gone.

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems. A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into

the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints.

Chapter 02

Operating System Structures

2.1. Operating-System Services

An operating system provides an environment for the execution of programs. It provides certain services to programs and to the users of those programs. The specific services provided, differ from one operating system to another, but we can identify common classes. These operating system services are provided for the convenience of the programmer, to make the programming task easier.

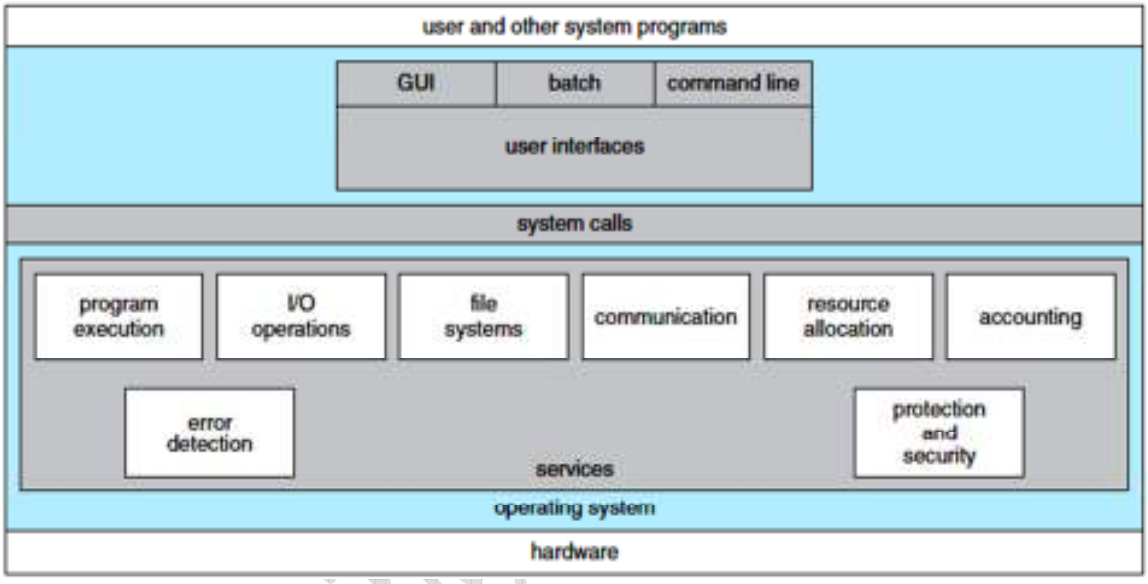


Figure 2.1 A view of operating system services.

1. **User interface.**

Almost all operating systems have a **user interface (UI)**. There are various forms of the interface. One is a **command-line interface (CLI)**, in command Line Interface the user have to type text command and use keyboard for typing commands in a specific format with specific options). Another is a **batch interface**, in which commands and directives to control those commands are entered into files, and those files are executed. Most commonly, a **graphical user interface (GUI)** is used. Here, the interface is a window system with a pointing device to direct I/O, choose from menus, and make selections and a keyboard to enter text.

2. **Program execution.**

The system must be able to load a program into memory and to run that program. The program must be able to end its execution, either normally or abnormally (indicating error).

3. Input/ Output operations.

A running program may require Input Output I/O operation form a file or an I/O device. For specific devices, special functions may be desired. For efficiency and protection, the operating system must provide a means to do I/O.

4. File-system manipulation:

During execution, programs need to read and write files and directories. They also need to create and delete files, search for a given file, and list file information. Operating system provides a verity of file manipulation services.

5. Communications:

Sometime one process needs to exchange information with another process. Such communication may occur between processes that are executing on the same computer or between processes that are executing on different computer systems tied together by a computer network. Communications may be implemented via **shared memory**, in which two or more processes read and write to a shared section of memory, or **message passing**, in which packets of information in predefined formats are moved between processes by the operating system.

6. Error detection:

The OS constantly needs to be aware of possible errors. Error may occur in the CPU and memory hardware, in I/O devices and in the user program. For each type of error, the OS should take appropriate action to ensure correct and consistent computing.

In order to assist the efficient operation of the system itself, the system provides the following functions:

7. Resource allocation:

When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. There are various routines to schedule jobs, allocate plotters, modems and other peripheral devices.

8. Accounting:

We want to keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting or simply for accumulating usage statistics.

9. Protection and Security:

The owners of information stored in a multi user computer system may want to control use of that information. When several disjointed processes execute concurrently it should not be possible for one process to interfere with the others or with the operating system itself. Protection involves ensuring that all access to system resources is controlled.

2.2. User and Operating-System Interface

There are several ways for users to interface with the operating system. There are two fundamental approaches.

- Command-Line Interface
- Graphical User Interface

1.2.1 Command Line Interface

Command Line Interface, also called Command Interpreter allows the user to type the command directly by using keyboard. Some operating systems include the command interpreter in the kernel. Others, such as Windows and UNIX, treat the command interpreter as a special program that is running when a job is initiated or when a user first logs on. Some operating system provides many command line interpreters to chose one from them, these interpreters are known as **shells**. For example, on UNIX and Linux systems, a user may choose among several different shells, including the ***Bourne shell***, ***C shell***, ***Bourne-Again shell***, ***Korn shell***, and others. Most shells provide similar functionality, Figure 2.2 shows the Bourne shell command interpreter being used on Solaris 10.

The main function of the command interpreter is to get and execute the user-specified command. Many of the commands are to manipulate files: create, delete, list, print, copy, execute, and so on. The MS-DOS and UNIX shells operate in this way.

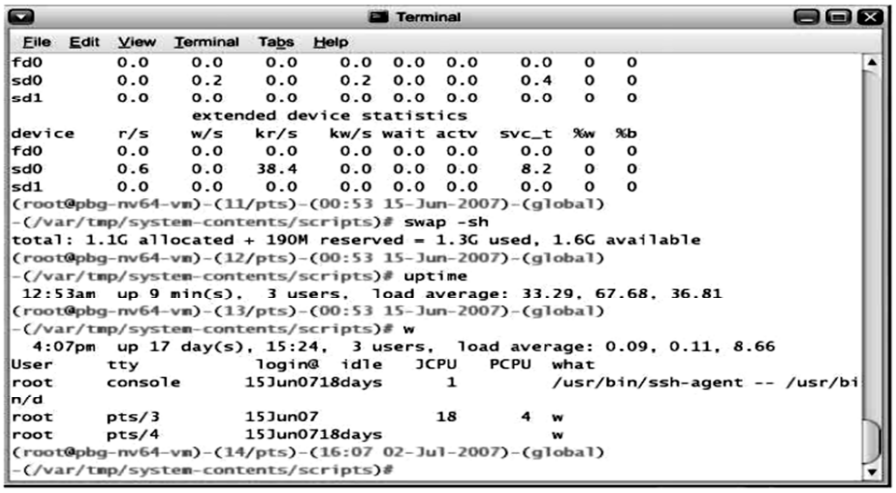


Figure 2.2 The Bourne shell command interpreter in Solrais 10.

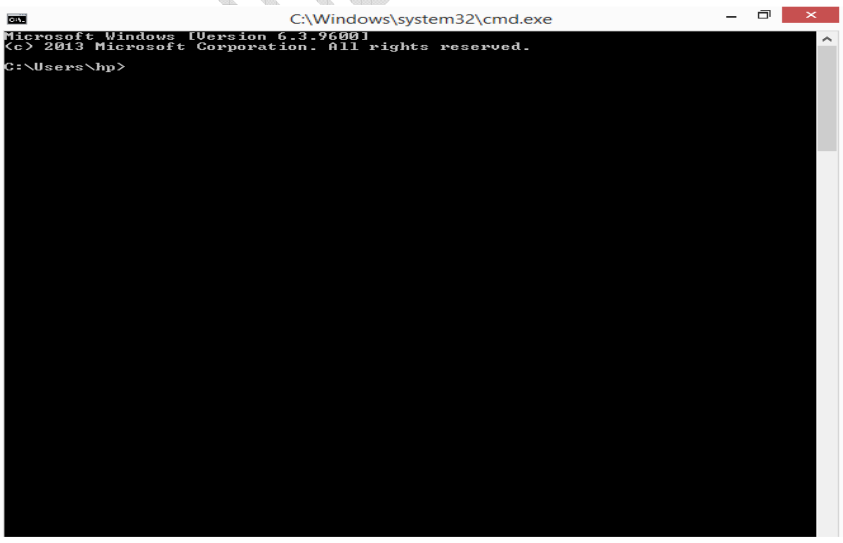


Figure 2.3 Command Line shell of MS DOS

1.2.2 Graphical User Interfaces

The second approach to access system is user friendly Graphical User Interface, or GUI. In GUI users use a mouse-based window and-menu system characterized

by a **desktop** metaphor. The user moves the mouse to position its pointer on images, or **icons**, on the screen that represent programs, files, directories, and system functions. Depending on the mouse pointer's location, clicking a button on the mouse can invoke a program, select a file or directory or pull down a menu that contains commands.

Graphical user interfaces first appeared due in part to research taking place in the early 1970s at Xerox PARC research facility. The first GUI appeared on the Xerox Alto computer in 1973. However, graphical interfaces became more widespread with the advent of Apple Macintosh computers in the 1980s. The user interface for the Macintosh operating system (Mac OS) has made various changes over the years, the most significant being the adoption of the ***Aqua*** interface that appeared with Mac OS X. Microsoft's first version of Windows—Version 1.0—was based on the addition of a GUI interface to the MS-DOS operating system. Later versions of Windows have made cosmetic changes in the appearance of the GUI along with several enhancements in its functionality.

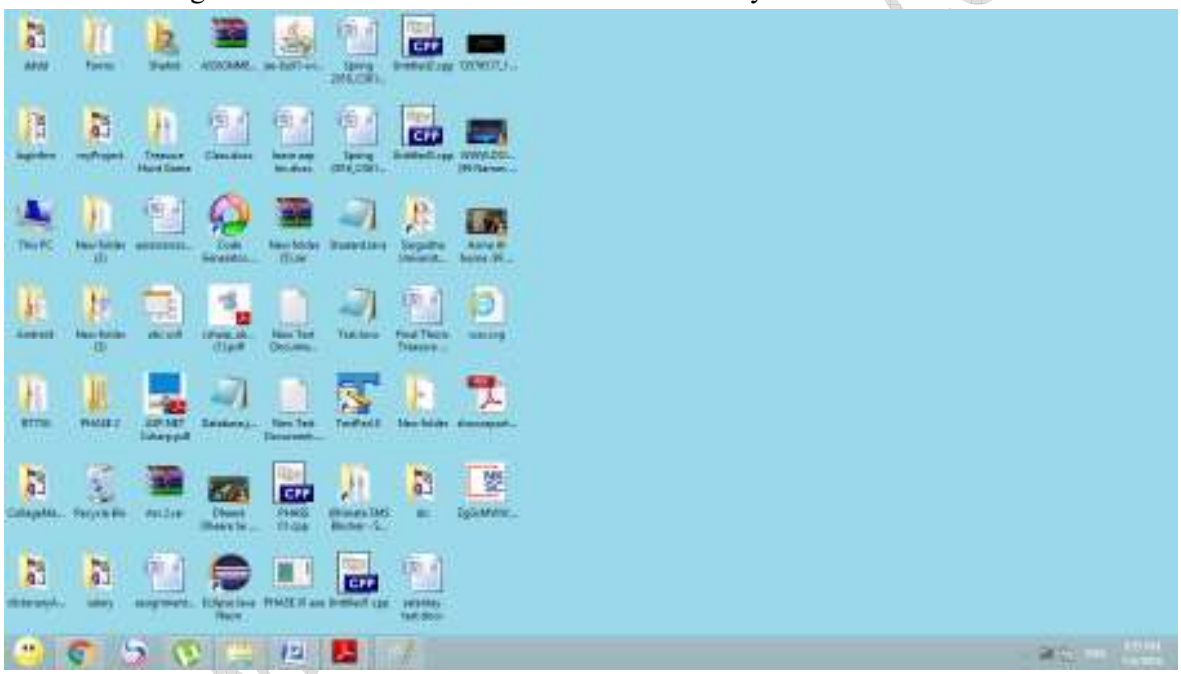


Figure 2.4 Windows 8.1 GUI (Desktop)

1.2.3 Choice of Interface

The choice of whether to use a command-line or GUI interface is mostly one of personal preference. **System administrators** who manage computers and **power users** who have deep knowledge of a system frequently use the command-line interface. It is more efficient for them, giving faster access to the activities they need to perform. Command line interfaces usually make repetitive tasks easier,

In contrast, most Windows users are happy to use the Windows GUI environment and almost never use the MS-DOS shell interface. The various changes undergone by the Macintosh operating systems provide a nice study in contrast. Historically, Mac OS has not provided a command-line interface, always requiring its users to interface with the operating system using its GUI.

The user interface can vary from system to system and even from user to user within a system. It typically is substantially removed from the actual system structure. The design of a useful and friendly user interface is therefore not a direct function of the operating system.

2.3. Virtual Machines

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 call for use by outer layers. The system 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. System programs in turn treat the hardware and the system calls as though they were both at the same level. In some systems the application programs can call the system programs. The application programs 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** (VM). The VM operating system for IBM systems is the best example of VM concept.

By using CPU scheduling and virtual memory techniques an operating system can create the illusion that a process has its own memory with its own (virtual) memory. The 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 physical computer shares resources to create the virtual machines. Figure 2.5 illustrates the concepts of virtual machines by a diagram.

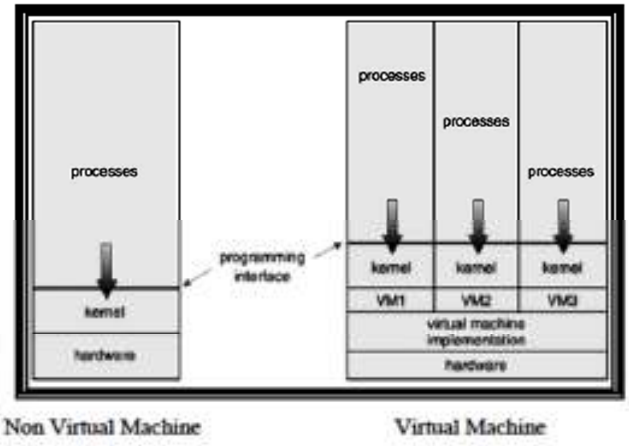
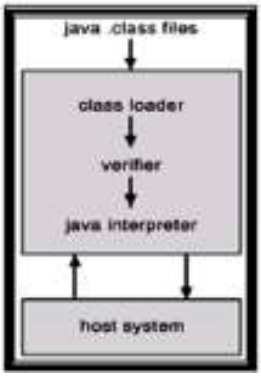


Figure 2.5 Illustration of virtual and non-virtual machines

Although the virtual machine concept is useful it is difficult to implement. There are two primary advantages to using virtual machines: first by completely protecting system resources the virtual machine provides a robust level of security. Second the virtual machine allows system development to be done without disrupting normal system operation.



Java Virtual Machine (**JVM**) loads, verifies, and executes programs that have been translated into Java Bytecode, as shown in Figure 4.4. **VMWare** can be run on a Windows platform to create a virtual

machine on which you can install an operating of
.

Figure2.6: Java Virtual Machine

your choice, such as Linux. We have shown a couple of snapshots of VMWare on a Windows platform in the lecture slides. **Virtual PC** software works in a similar fashion.

2.4. System Calls

System calls provide the interface between a process and the OS. These calls are generally available as assembly language instructions. The system call interface layer contains entry point in the kernel code; because all system resources are managed by the kernel any user or application request that involves access to any system resource must be handled by the kernel code, but user process must not be given open access to the kernel code for security reasons. So that user processes can invoke the execution of kernel code, several openings into the kernel code, also called *system calls*, are provided. System calls allow processes and users to manipulate system resources such as files and processes.

Typically, application developers design programs according to an **application programming interface (API)**. The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect.

The Functions of API invoke the actual system calls on behalf of the application programmer. For example, the Windows function CreateProcess() actually invokes the NTCreateProcess() system call in the Windows kernel.

For most programming languages, the run-time support system (a set of functions built into libraries included with a compiler) provides a **system call interface** that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. Typically, a number is associated with each system call, and the system-call interface maintains a table indexed according to these numbers. The system call interface returns the status of the system call and any return values. The caller need know nothing about how the system call is implemented or what it does during execution. Rather, the caller need only obey the API and understand what the operating system will do as a result of the execution of that system call. Thus, most of the details of the operating-system interface are hidden from the programmer by the API and are managed by the run-time support library.

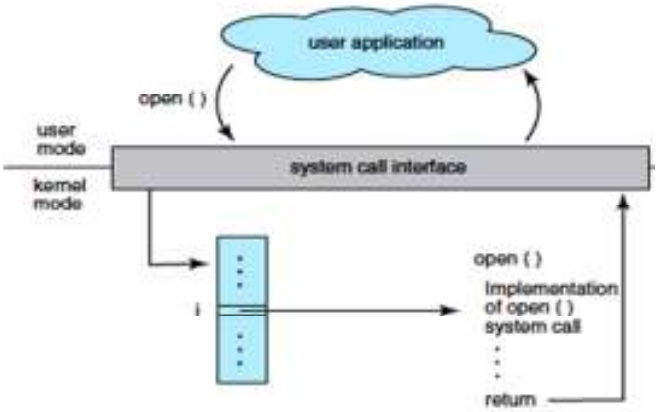


Figure 2.7 The handling of a user application invoking the open () system call.

Types of System Calls

System calls can be categorized into the following groups:

1. Process control
2. File manipulation
3. Device manipulation
4. Information maintenance
5. Communications
6. Protection.

2.4.1 Process Control

A running program needs to be able to halt its execution either normally (end()) or abnormally (abort()). A process or job executing one program may want to load() and execute() another program. To create a new job or process to be multi-programmed. Often, there is a system call specifically for this purpose (create process () or submit job ()).

If we create a new job or process, or perhaps even a set of jobs or processes, we should be able to control its execution. This control requires the ability to determine and reset the attributes of a job or process, including the job's priority, its maximum allowable execution time, and so on (get process attributes () and set process attributes ())To terminate a job or process that is created (terminate process()) if it is incorrect or is no longer needed.

To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. Depending on the way the command was issued, the shell then either waits for the process to finish or runs the process "in the background." In the latter case, the shell immediately requests another command. When a process is running in the background, it cannot receive input directly from the keyboard, because the shell is using this resource. I/O is therefore done through files or through a GUI interface. Meanwhile, the user is free to ask the shell to run other programs, to monitor the progress of the running process, to change that program's priority, and so on. When the process is done, it executes an exit() system call to terminate, returning to the invoking process a status code of 0 or a nonzero error code. This status or error code is then available to the shell or other programs.

2.4.2 File Management

There are several common system calls dealing with files. i.e. create() and delete() files. Either system call requires the name of the file and perhaps some of the file's attributes. Once the file is created, we need to open () it and to use it. We may also read(), write(), or reposition() (rewind or skip to the end of the file, for example).

Finally, we need to close () the file, indicating that we are no longer using it. We may need these same sets of operations for directories if we have a directory structure for organizing files in the file system. In addition, for either files or directories, we need to be able to determine the values of various attributes and perhaps to reset them if necessary. File attributes include the file name, file type, protection codes, accounting information, and so on. At least two system calls, getfile*attributes()* and setfile*attributes(),* are required for this function. Some operating systems provide many more calls, such as calls for file move() and copy().

2.4.3 Device Management

A process may need several resources to execute—main memory, disk drives, access to files, and so on. If the resources are available, they can be granted, and control can be returned to the user process. Otherwise, the process will have to wait until sufficient resources are available.

The various resources controlled by the operating system can be thought of as devices. Some of these devices are physical devices (for example, disk drives), while others can be thought of as abstract or virtual devices (for example, files).A system with multiple users may first request() a device. After completing the task release() system call is used to release the resource. Once the device has been requested (and allocated to us), process can read(), write(), and (possibly) reposition() the device.

2.4.4 Information Maintenance

Many system calls exist simply for the purpose of transferring information between the user program and the operating system. For example, most systems have a system call to return the current time() and date(). Other system calls may return information about the system, such as the number of current users, the version number of the operating system, the amount of free memory or disk space, and so on.

Another set of system calls is helpful in debugging a program. Many systems provide system calls to dump() memory. This provision is useful for debugging. A program trace lists each system call as it is executed. Even microprocessors provide a CPU mode known as single step, in which a trap is executed by the CPU after every instruction. The trap is usually caught by a debugger. Generally, calls are also used to reset the process information (get process attributes() and set process attributes()).

2.4.5 Communication

There are two common models of inter process communication: the message passing model and the shared-memory model. In the message-passing model, the communicating processes exchange messages with one another to transfer information. Messages can be exchanged between the processes either directly or indirectly through a common mailbox. Before communication can take place, a connection must be opened. The name of the other communicator must be known, be it another process on the same system or a process on

another computer connected by a communications network. Each computer in a network has a host name by which it is commonly known. A host also has a network identifier, such as an IP address. Similarly, each process has a process name, and this name is translated into an identifier by which the operating system can refer to the process. The get hostid() and get processid() system calls do this translation. The identifiers are then passed to the general purpose open() and close() calls provided by the file system or to specific open connection() and close connection() system calls, depending on the system's model of communication. The recipient process usually must give its permission for communication to take place with an accept connection() call.

The source of the communication, known as the client, and the receiver, known as a server, then exchange messages by using read message() and write message() system calls. The close connection () call terminates the communication.

In the shared-memory model, processes use shared memory create() and shared memory attach() system calls to create and gain access to regions of memory owned by other processes.

2.4.6 Protection

Protection provides a mechanism for controlling access to the resources provided by a computer system. Historically, protection was a concern only on multi-programmed computer systems with several users. However, with the advent of networking and the Internet, all computer systems, from servers to mobile handheld devices, must be concerned with protection. Typically, system calls providing protection include set permission() and get permission(), which manipulate the permission settings of resources such as files and disks. The allowuser() and denyuser() system calls specify whether particular users can—or cannot—be allowed access to certain resources.

Type	System Calls
Process control	• 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
File management	• create file, delete file • open, close • read, write, reposition • get file attributes, set file attributes
Device management	• request device, release device • read, write, reposition • get device attributes, set device attributes • logically attach or detach devices
Information maintenance	• 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

Communications	• create, delete communication connection • send, receive messages • transfer status information • attach or detach remote devices

2.5. System Programs

System programs, also known as **system utilities**, 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. Others are more complex, providing detailed performance, logging, and debugging information. Typically, these programs format and print the output to the terminal or other output devices or files or display it in a window of the GUI. Some systems also support a **registry**, which is used to store and retrieve configuration information.
3. **File modification**. Several text editors may be available to create and modify the content of files stored on disk or other storage devices. There may also be special commands to search contents of files or perform transformations of the text.
4. **Programming-language support**. Compilers, assemblers, debuggers, and interpreters for common programming languages (such as C, C++, Java, and PERL) are often provided with the operating system or available as a separate download.
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, re-locatable loaders, linkage editors, and overlay loaders. Debugging systems for either higher-level languages or machine language are needed as well.
6. **Communications**. These programs provide the mechanism for creating virtual connections among processes, users, and computer systems. They allow users to send messages to one another's screens, to browse Web pages, to send e-mail messages, to log in remotely, or to transfer files from one machine to another.
7. **Background services**. All general-purpose systems have methods for launching certain system-program processes at boot time. Some of these processes terminate after completing their tasks, while others continue to run until the system is halted. Constantly running system-program processes are known as **services**, **subsystems**, or daemons.

2.6. Operating-System Structure

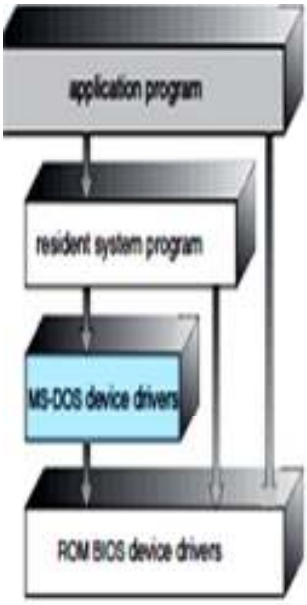
A large and complex modern operating system must be engineered carefully if it is to function properly and be modified easily. A common approach is to partition the task into small components, or modules, rather than have one monolithic system. Each of these modules should be a well-defined portion of the system, with carefully defined inputs, outputs, and functions.

2.6.1 Simple/ Monolithic Structure

Many operating systems do not have well-defined structures. Such systems started as small, simple, and limited systems and then grew beyond their original scope. MS-DOS is an example of such a system. It was written to provide the most functionality in the least space, so it was not carefully divided into modules. Figure 2.8 shows its 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 routinesto 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 butto leave the base hardware accessible.

Another example of limited structuring is the original UNIX operating system. Like MS-DOS, UNIX initially was limited by hardware functionality. It consists of two separable parts: the kernel and the system programs.



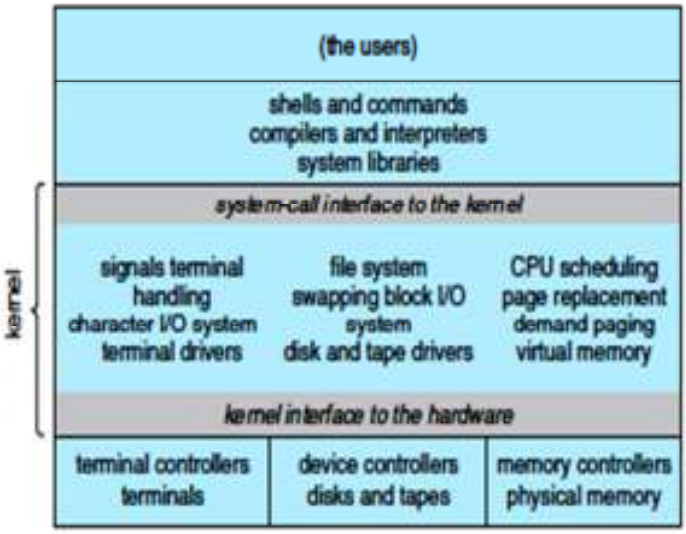


Figure 2.8 MS-DOS layer structure. **Figure 2.9**Traditional UNIX system structure

The kernel is further separated into a series of interfaces and device drivers, which have been added and expanded over the years as UNIX has evolved. We can view the traditional UNIX operating system as being layered to some extent, as shown in Figure 2.9. Everything below the system-call interface and above the physical hardware is the kernel. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Taken in sum, that is an enormous amount of functionality to be combined into one level. This monolithic structure was difficult to implement and maintain. It had a distinct performance advantage, however: there is very little overhead in the system call interface or in communication within the kernel.

2.6.2 Layered Approach

With proper hardware support, operating systems can be broken into pieces that are smaller and more appropriate than those allowed by the original MS-DOS and UNIX systems. The operating system can then retain much greater control over the computer and

over the applications that make use of that computer. Implementers have more freedom in changing the inner workings of the system and in creating modular operating systems. Under a top down approach, the overall functionality and features are determined and are separated into components. Information hiding is also important, because it leaves programmers free to implement the low-level routines as they see fit, provided that the external interface of the routine stays unchanged and that the routine itself performs the advertised task.

A system can be made modular in many ways. One method is the layered approach, in which the operating system is broken into a number of layers (levels). The bottom layer (layer 0) is the hardware; the highest (layer N) is the user interface. This layering structure is depicted in Figure 2.10.

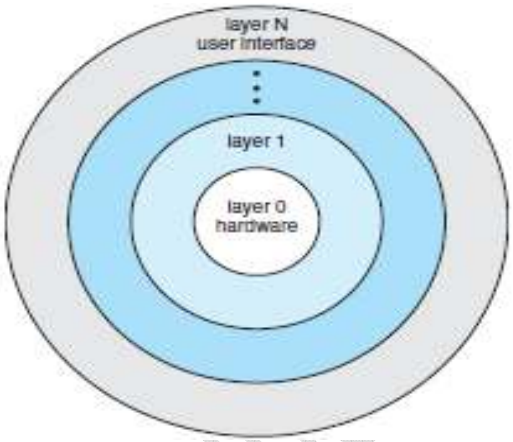


Figure 2.10 A layered operating system.

An operating-system layer is an implementation of an abstract object made up of data and the operations that can manipulate those data. A typical operating-system layer—say, layer M—consists of data structures and a set of routines that can be invoked by higher-level layers. Layer M, in turn, can invoke operations on lower-level layers. The main advantage of the layered approach is simplicity of construction and debugging. The layers are selected so 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 debugged, and so on. If an error is found during the debugging of a particular layer, the error must be on that layer, because the layers below it are already debugged. Thus, the design and implementation of the system are simplified. Each layer is implemented only with 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.

Problems with Layered Approach

The major difficulty with the layered approach involves appropriately defining the various layers. Because a layer can use only lower-level layers, careful planning is necessary. For example, the device driver for the backing store (disk space used by virtual-memory algorithms) must be at a lower level than 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, when a user program executes an I/O operation, it executes a system call that is trapped to the I/O layer, which calls the memory-management layer, which in turn calls the CPU-scheduling layer, which is then passed 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. The net result is a system call that takes longer than does one 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 problems of layer definition and interaction.

2.6.3 Micro kernels

This method structures the operating system by removing all non-essential components from the kernel and implementing as system and user level programs. The result is a smaller kernel. Micro kernels typically provide minimum process and memory management in addition to a communication facility. The main function of the micro kernel is to provide a communication facility between the client program and the various services that are also running in the user space.

The benefits of the micro kernel approach include the ease of extending the OS. All new services are added to user space and consequently do not require modification of the kernel. When the kernel does have to be modified, the changes tend to be fewer because the micro kernel is a smaller kernel. The resulting OS is easier to port from one hard ware design to another. It also provides more security and reliability since most services are running as user rather than kernel processes. Mach, MacOS X Server, QNX, OS/2, and Windows NT are examples of microkernel based operating systems. various types of services can be run on top of the Windows NT microkernel, thereby allowing applications developed for different platforms to run under Windows NT.

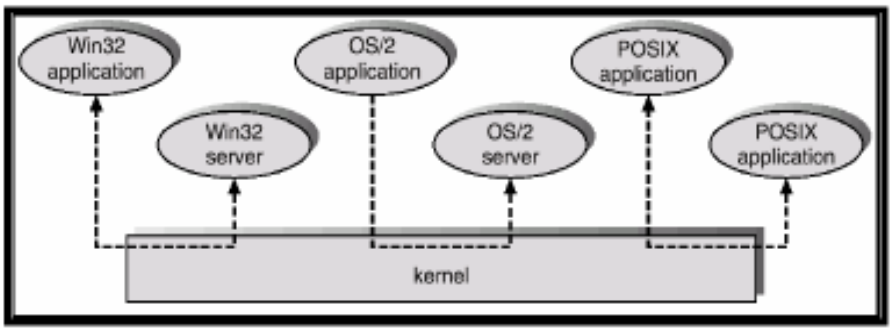


Figure 2.11 Windows NT client-server structure

2.7. System Boot

The procedure of starting a computer by loading the kernel is known as **booting** the system. On most computer systems, a small piece of code known as the **bootstrap program** or **bootstrap loader** locates the kernel, loads it into main memory, and starts its execution. Some computer systems, such as PCs, use a two-step process in which a simple bootstrap loader fetches a more complex boot program from disk, which in turn loads the kernel. When a CPU receives a reset event—for instance, when it is powered up or rebooted—the instruction register is loaded with a predefined memory location, and execution starts there. At that location is the initial bootstrap program. This program is in the form of **read-only memory (ROM)**, because the RAM is in an unknown state at system startup. ROM is convenient because it needs no initialization and cannot easily be infected by a computer virus.

The bootstrap program can perform a variety of tasks. Usually, one task is to run diagnostics to determine the state of the machine. If the diagnostics pass, the program can continue with the booting steps. It can also initialize all aspects of the system, from CPU registers to device controllers and the contents of main memory. Sooner or later, it starts the operating system.

Some systems—such as cellular phones, tablets, and game consoles—store the entire operating system in ROM. Storing the operating system in ROM is suitable for small operating systems, simple supporting hardware, and rugged operation. A problem with this approach is that changing the bootstrap code requires changing the ROM hardware chips. Some systems resolve this problem by using **erasable programmable read-only memory (EPROM)**, which is read only except when explicitly given a command to become writable.

For large operating systems (including most general-purpose operating systems like Windows, Mac OS X, and UNIX) or for systems that change frequently, the bootstrap loader is stored in ROM, and the operating system is on disk. In this case, the bootstrap runs diagnostics and has a bit of code that can read a single block at a fixed location (say block zero) from disk into memory and execute the code from that **boot block**. The program stored in the boot block may be sophisticated enough to load the entire operating system into memory and begin its execution. More typically, it is simple code (as it fits in a single disk block) and knows only the address on disk and length of the remainder of the bootstrap program. **GRUB** is an example of an open-source bootstrap program for Linux systems. All of the disk-bound bootstrap, and the operating system itself, can be easily changed by writing new versions to disk. A disk that has a boot partition is called a **boot disk** or **system disk**.

Now that the full bootstrap program has been loaded, it can traverse the file system to find the operating system kernel, load it into memory, and start its execution. It is only at this point that the system is said to be **running**.

Chapter 03

Process

3.1. Process:

A process can be thought of as a program in execution. A process will need certain resources – such as CPU time, memory, files, and I/O devices – to accompany its task. These resources are allocated to the process either when it is created or while it is executing.

A process is the unit of work in most systems. Such a system consists of a collection of processes: operating system processes execute system code and user processes execute user code. All these processes may execute concurrently. Although traditionally a process contained only a single thread of control as it ran, most modern operating systems now support processes that have multiple threads. A batch system executes jobs (background processes), whereas a time-shared system has user programs, or tasks. Even on a single user system, a user may be able to run several programs at one time: a word processor, web browser etc.

A process is more than program code, which is sometimes known as the text section. It also includes the current activity, as represented by the value of the program counter and the contents of the processor's register. In addition, a process generally includes the process stack, which contains temporary data (such as method parameters, the process stack, which contains temporary data), and a data section, which contains global variables.

A program by itself is not a process: a program is a passive entity, such as contents of a file stored on disk, whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. Although two processes may be associated with the same program, they are considered two separate sequences of execution. E.g. several users may be running different instances of the mail program, of which the text sections are equivalent but the data sections vary. Processes may be of two types:

- **IO bound processes:** spend more time doing IO than computations, have many short CPU bursts. Word processors and text editors are good examples of such processes.
- **CPU bound processes:** spend more time doing computations, few very long CPU bursts.

3.1.1 Process State

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

- **New**. The process is being created.
- **Running**. Instructions are being executed.
- **Waiting**. The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**. The process is waiting to be assigned to a processor.
- **Terminated**. The process has finished execution.

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

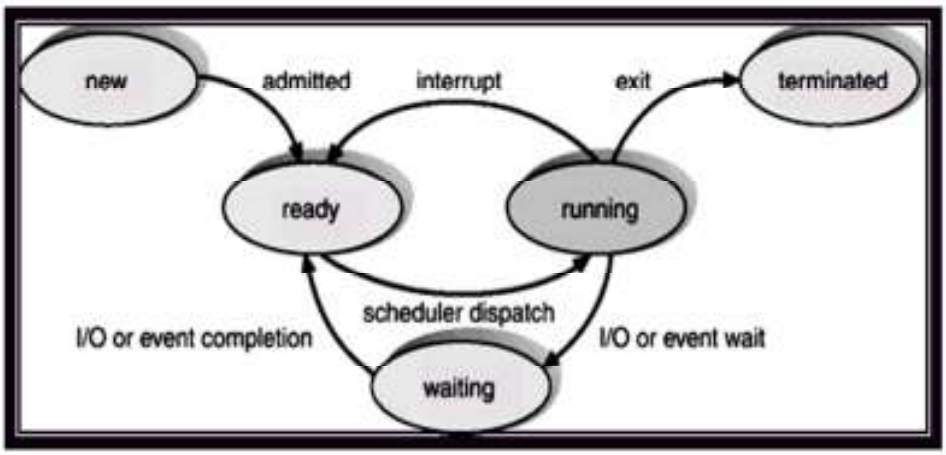


Figure 3.1 Process state diagram

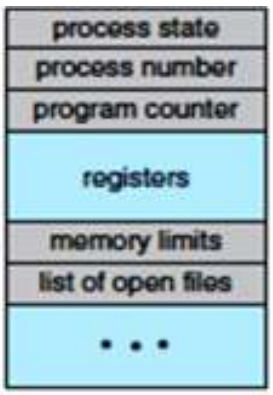
3.1.2 Process Control Block

Each process is represented in the operating system by a **process control block** (PCB) – also called a task control block, as shown in Figure 3.2. A PCB contains many pieces of information associated with a specific process, including these:

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

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

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



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 such 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.

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

Figure 3.2 PCB

I/O status information: The information includes the list of I/O devices allocated to the process, a list of open files, and so on.

3.2. Process Scheduling

The objective of multiprogramming is to have some process running all the time 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 uni-processor system can have only one running process at a given time. If more processes exist, the rest must wait until the CPU is free and can be rescheduled. Switching the CPU from one process to another requires saving of the context of the current process and loading the state of the new process, as shown in Figure 3.3 This is called **context switching**.

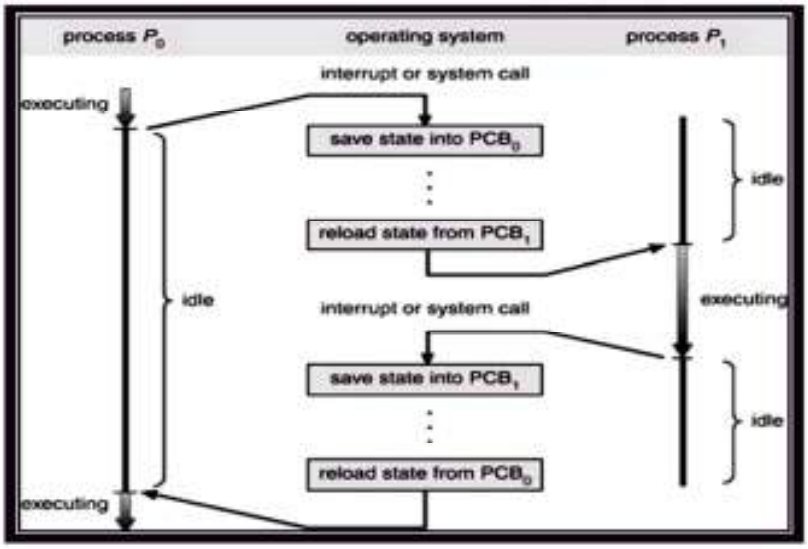


Figure 3.3 Context switching

3.2.1. Scheduling Queues

A modern computer system maintains many scheduling queues. Here is a brief description of some of these queues:

Job Queue:

As processes enter the system, they are put into a job queue. This queue consists of all processes in the system.

Ready Queue:

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. Each PCB is extended to include a pointer field that points to the next PCB in the ready queue.

Device Queue:

When a process is allocated the CPU, it executes for a while, and eventually quits, is interrupted or waits for a particular event, such as completion of an I/O request. In the case of an I/O request, the device may be busy with the I/O request of some other process, hence the list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

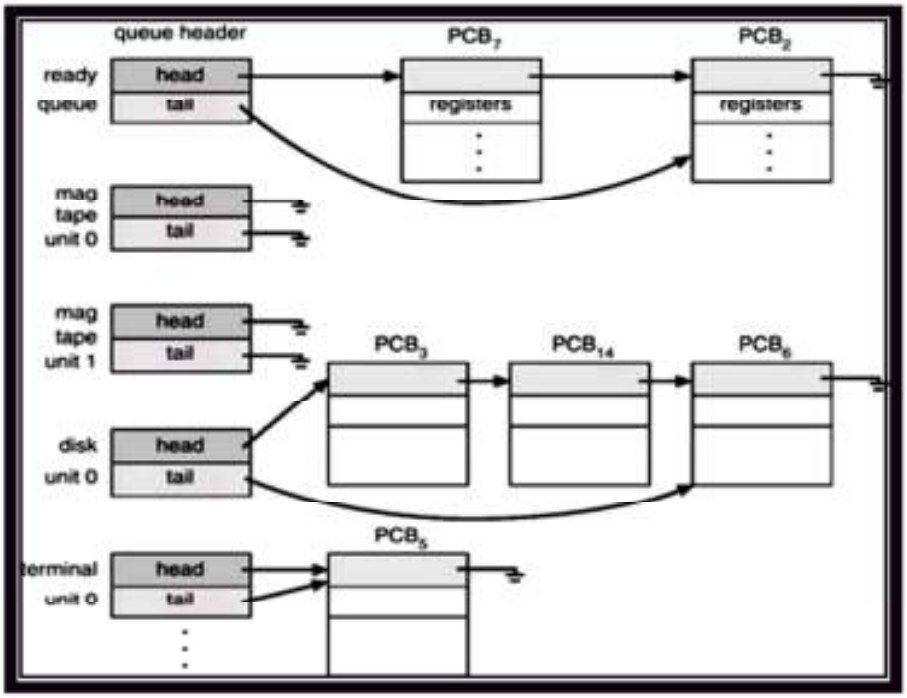


Figure 3.4 Scheduling queue

Such as that in Figure 3.5. 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.

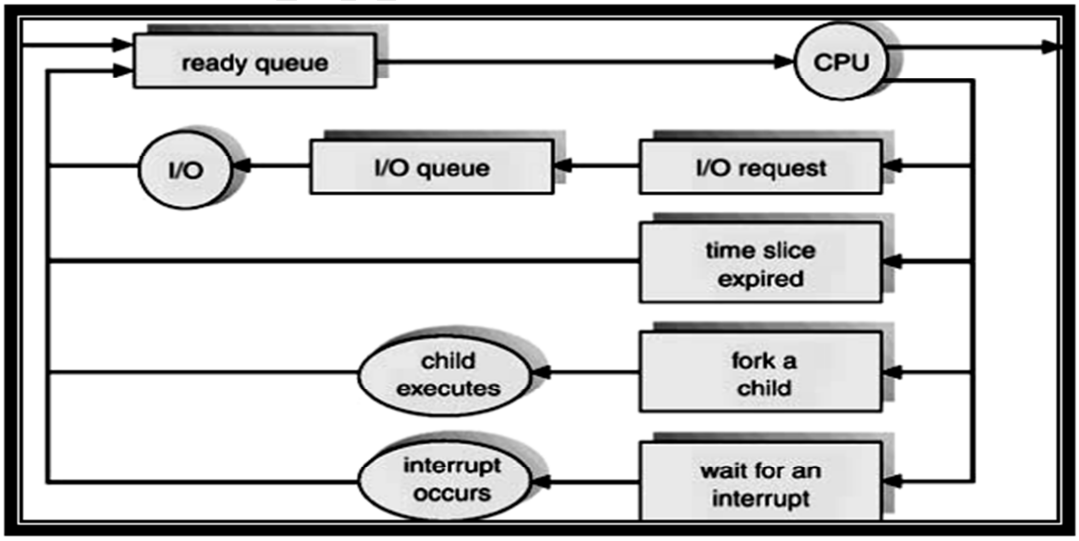


Figure 3.5 Queuing diagram of a computer system

A new process is initially put in the ready queue. It waits there until it is selected for execution, or **dispatched**. Once the process is allocated 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 child process and wait for the child's 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 de-allocated.

3.2.2. Schedulers

When multiple processes are to be executed, the operating system must schedule the processes for execution. The part of operating system, which schedules the processes (or responsible for scheduling), is called the scheduler. The scheduler selects a process from the ready queue and allocates the CPU to that process. The ready queue can be implemented as a FIFO queue, a priority queue, an unordered linked list etc. However, all the processes in the ready queue are linked up waiting for a chance to run on the CPU.

Types of Schedulers

Several types of schedulers may be used in operating systems. There are three fundamentals types of schedulers.

I. Long-Term Scheduler
II. Short-Term Scheduler
III. Medium-Term Scheduler

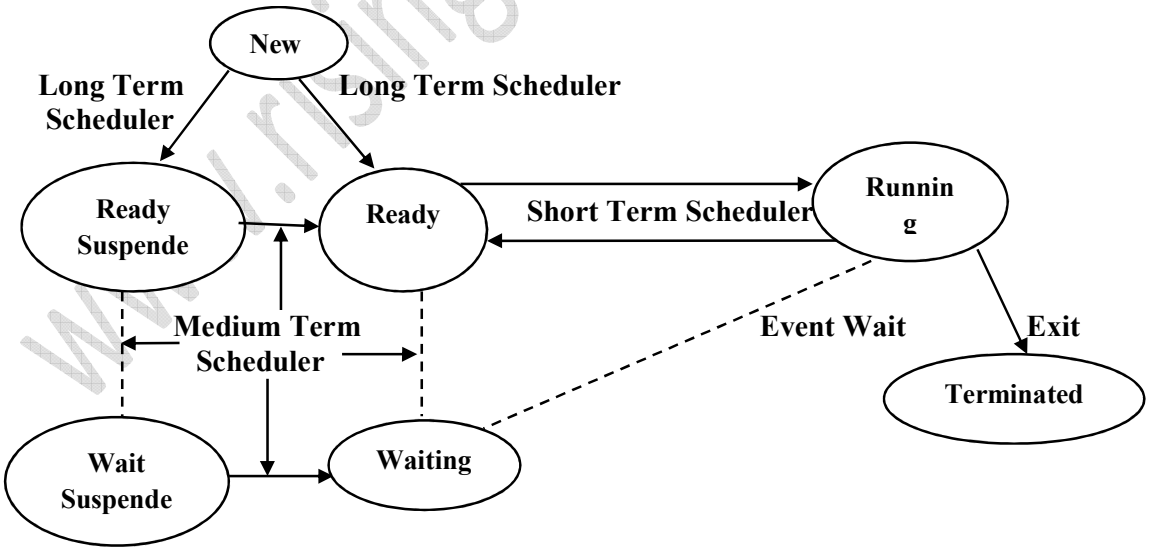


Figure 3.6 Working of different types of CPU Schedulers

Long-Term Scheduler

Long-term scheduler is also known as Job Scheduler. It selects jobs or user programs from job pool on the disk and loads them into main memory. Once a job or user program is

loaded into memory, it becomes the process and is added to the ready queue. The long-term scheduler controls the number of processes in memory. It means that it controls the degree of multiprogramming.

If the degree of programming is stable, then the average rate of processes creation must be equal to the average departure rate of processes leaving the system. When a process leaves the system, the long-term scheduler performs its function. It decides which processes should be selected for execution. For this purpose, the long-term scheduler may have to spend a long time for making decision.

The long-term scheduler must select a good mix of I/O bound and CPU bound jobs. The reason why the long-term scheduler must select a good mix of I/O bound and CPU bound jobs is that if the processes are I/O bound, the ready queue will be mostly empty and the short-term scheduler will have little work. On the other hand, if the processes are mostly CPU bound, then the devices will go unused and the system will be unbalanced.

Short-Term Scheduler

The short-term scheduler retrieves a process from the ready queue and allocates CPU to that process. The process state is changed from ready to running. If an interrupt or time-out occurs, the scheduler places the running process back into the ready queue and marks it as ready. Typically, the short-term scheduler gives the control of CPU for a short period of time. A process may be executed for only a few milliseconds, and then the next process is selected from ready queue for execution.

The primary distinction between the two schedulers is the frequency of 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 % of the CPU is being used for scheduling only. The long-term scheduler, on the other hand executes much less frequently. There may be minutes between the creations 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 department rate of processes leaving 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.

Medium-Term Scheduler

The medium-term scheduling is also known as “Swapping”. In this scheduling swapper is used to exchange (or swap) processes between main memory and secondary storage (disk). Typically, multiprocessing systems use swapping technique. Sometimes, a process that is in job queue on disk is too long and cannot fit in memory, and then swapper removes another process (or multi process) from ready queue to make room for it. Later, the same process is re-loaded into memory, which is added to the ready queue and its execution is continued where it was left off. If a process is waiting for completion for a short term I/O operation, then it is not swapped out. The area on the disk where swapped out processes are stored is called the **swap space**.

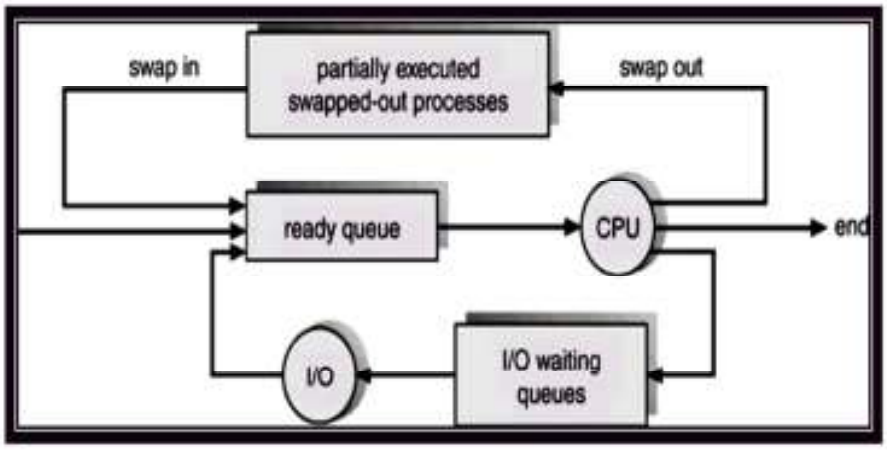


Figure 3.7 Computer system queues, servers, and swapping

3.2.3. Context Switch

Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine. Such operations happen frequently on general-purpose systems. When an interrupt occurs, the system needs to save the current **context** of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it. The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state, and memory-management information. Generically, we perform a **state save** of the current state of the CPU, be it in kernel or user mode, and then a **state restore** to resume operations.

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as **context switch**. 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. Switching 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). A typical speed is a few milliseconds.

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 here simply requires changing the pointer to the current register set. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory, as before. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. Advanced memory-management techniques may require that extra data 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.

3.3. Operations on Processes

The processes in the system execute concurrently and they must be created and deleted dynamically thus the operating system must provide the mechanism for the creation and deletion of processes.

3.3.1. Process Creation

A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a parent process while the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Figure 3.7 shows partially the process tree in a UNIX/Linux system.

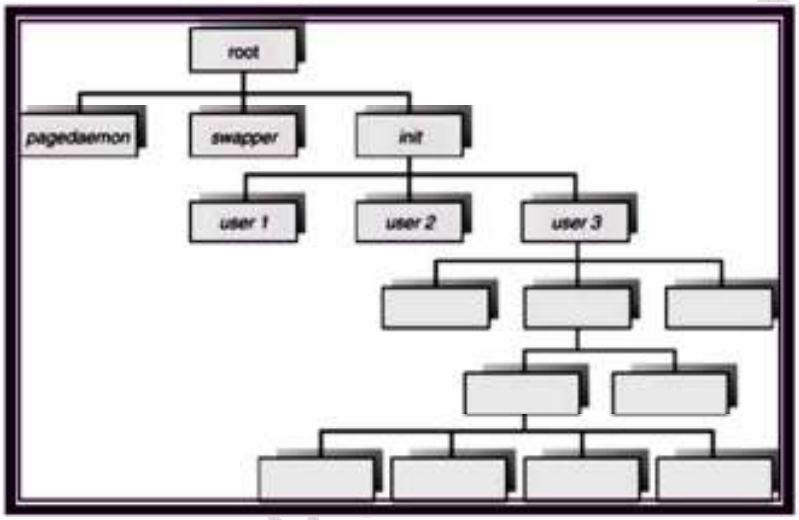


Figure 3.8 Process tree in UNIX/Linux

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX its process identifier identifies a process, which is a unique integer. A new process is created by the fork system call. The new process consists of a copy

of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the fork call, with one difference, the return code for the fork system call is zero for the child process, while the process identifier of the child is returned to the parent process.

Typically the execlp() system call is used after a fork system call by one of the two processes to replace the process' memory space with a new program. The execlp() system call loads a binary file in memory –destroying the memory image of the program containing the execlp() system call.—and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a wait system call to move itself off the ready queue until the termination of the child. The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the exit system call.

The fork() system call

When the fork system call is executed, a new process is created. The original process is called the parent process whereas the process is called the child process. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. On success, both processes continue execution at the instruction after the fork call, with one difference, the return code for the fork system call is zero for the child process, while the process identifier of the child is returned to the parent process. On failure, a -1 will be returned in the parent's context, no child process will be created, and an error number will be set appropriately.

The synopsis of the fork system call is as follows:

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main()
{
pid t pid;
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occurred */
fprintf(stderr, "Fork Failed");
return 1;
}
else if (pid == 0) { /* child process */
execlp("/bin/ls","ls",NULL);
}
else { /* parent process */
/* parent will wait for the child to complete */
wait(NULL);
printf("Child Complete");
}
return 0;
}
```

Figure 3.9 Creating a separate process using the UNIX fork() system call.

Figure 3.8 shows sample code, showing the use of the fork() system call and Figure 3.9 shows the semantics of the fork system call. As shown in Figure 6.3, fork()creates an exact memory image of the parent process and returns 0 to the child process and the process ID of the child process to the parent process.

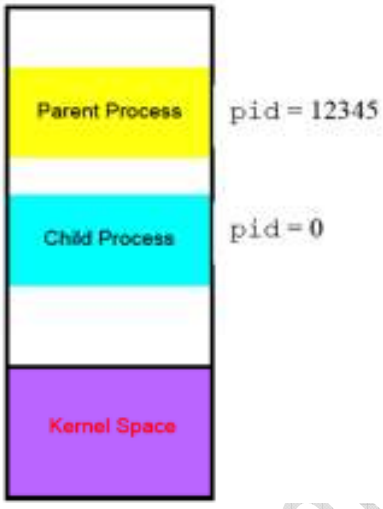


Figure 3.10 Semantics of the fork system call

After the fork() system call the parent and the child share the following:

- Environment
- Open file descriptor table
- Signal handling settings
- Nice value
- Current working directory
- Root directory
- File mode creation mask (umask)

The following things are different in the parent and the child:

- Different process ID (PID)
- Different parent process ID (PPID)
- Child has its own copy of parent's file descriptors

The fork() system may fail due to a number of reasons. One reason maybe that the maximum number of processes allowed to execute under one user has exceeded, another could be that the maximum number of processes allowed on the system has exceeded. Yet another reason could be that there is not enough swap space.

The wait() system call

The wait system call suspends the calling process until one of the immediate children terminate, or until a child that is being traced stops because it has hit an event of interest. The wait will return prematurely if a signal is received. If all child processes stopped or terminated prior to the call on wait, return is immediate. If the call is successful, the process ID of a child is returned. If the parent terminates however all its children have assigned as their new parent, the init process. Thus the children still have a parent to collect their status and execution statistics.

The execlp() system call

Typically, the execlp() system call is used after a fork() system call by one of the two processes to replace the process' memory space with a new program. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful exec because the calling process image is overlaid by the new process image. In this manner, the two processes are able to communicate and then go their separate ways.

3.3.2. Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call. At that point, the process may return data to its parent process (via the wait system call). All the resources of the process including physical and virtual memory, open the files and I/O buffers – are de allocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another via an appropriate system call (such as abort). Usually only the parent of the process that is to be terminated can invoke this system call. Therefore parents need to know the identities of its children, and thus when one process creates another process, the identity of the newly created process is passed to the parent. A parent may terminate the execution of one of its children for a variety of reasons, such as:

- 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.
- 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 if its parent terminates. On such a system, 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.

Considering an example from UNIX, we can terminate a process by using the exit system call; its parent process may wait for the termination of a child process by using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates however all its children have assigned as their new parent, the init process. Thus the children still have a parent to collect their status and execution statistics.

3.4. Cooperative Processes

Processes executing concurrently in the operating system may be either independent processes or cooperating processes. A process is ***independent*** if it cannot affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is ***cooperating*** if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation:

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

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

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

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

3.5. Inter process Communication

Cooperating processes require an **interprocess communication (IPC)** mechanism that will allow them to exchange data and information. There are two fundamental models of inter process communication methods:

1. **Shared memory**
2. **Message passing**.

In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

3.5.1. Shared-Memory Systems

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

Two types of buffers can be used. The **unbounded buffer** places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items. The **bounded buffer** assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

3.5.2. Message Passing System

The function of a message system is to allow processes to communicate without the need to resort to the shared data. Messages sent by a process may be of either fixed or variable size. If processes P and Q want to communicate, a communication link must exist between them and they must send messages to and receive messages from each other through this link. Here are several methods for logically implementing a link and the send and receive options:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed size or variable size messages

We now look at the different types of message systems used for IPC.

Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or receiver of the communication. The send and receive primitives are defined as:

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

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

Unlike this symmetric addressing scheme, a variant of this scheme employs asymmetric addressing, in which the recipient is not required to name the sender.

- Send(P, message) – send a message to process P
- Receive (id, message) – receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

Indirect Communication

With indirect communication, messages can be sent to and received from mailboxes. Here, two processes can communicate only if they share a mailbox. The send and receive primitives are defined as:

- Send(A, message) – send a message to mailbox A.
- Receive(A, message) – receive a message from mailbox A.

A communication link in this scheme has the following properties:

- A link is established between a pair of processes only if both members have a shared mailbox.
- A link is associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Synchronization

Communication between processes takes place by calls to send and receive primitives (i.e., functions). Message passing may be either blocking or non-blocking also called as synchronous and asynchronous.

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

Buffering

Whether the communication is direct or indirect, messages exchanged by the processes reside in a temporary queue. This queue can be implemented in three ways:

- **Zero Capacity:** The queue has maximum length zero, thus the link cannot have any messages waiting in it. In this case the sender must block until the message has been received.
- **Bounded Capacity:** This queue has finite length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender resumes operation. If the queue is full, the sender blocks until space is available.
- **Unbounded Capacity:** The queue has infinite length; thus the sender never blocks.

3.6 Threads:

A thread is a basic unit of CPU utilization; it comprises a thread ID, a program counter, a register set, and a stack. It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals. A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time. Figure 3.10 illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.

3.6.1 Motivation

Most software applications that run on modern computers are multithreaded. An application typically is implemented as a separate process with several threads of control. A web browser might have one thread display images or text while another thread retrieves data from the network, for example. A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and a third thread for performing spelling and grammar checking in the background. Applications can also be designed to leverage processing capabilities on multicore systems. Such applications can perform several CPU-intensive tasks in parallel across the multiple computing cores.

In certain situations, a single application may be required to perform several similar tasks. For example, a web server accepts client requests for web pages, images, sound, and so forth. A busy web server may have several (perhaps thousands of) clients concurrently accessing it. If the web server ran as a traditional single-threaded process, it would be able to service only one client at a time, and a client might have to wait a very long time for its request to be serviced.

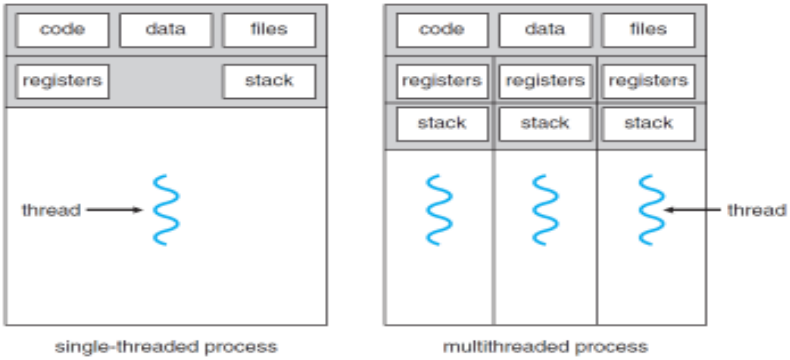


Figure 3.11 Single-threaded and multithreaded processes.

One solution is to have the server run as a single process that accepts requests. When the server receives a request, it creates a separate process to service that request. In fact, this process-creation method was in common use before threads became popular. Process creation is time consuming and resource intensive, however. If the new process will perform the same tasks as the existing process, why incur all that overhead? It is generally more efficient to use one process that contains multiple threads. If the web-server process is multithreaded, the server will create a separate thread that listens for client requests. When a request is made, rather than creating another process, the server creates a new thread to service the request and resume listening for additional requests. This is illustrated in Figure 3.11.

Threads also play a vital role in **remote procedure call** (RPC) systems. RPCs allow interprocess communication by providing a communication mechanism similar to ordinary function or procedure calls. Typically, RPC servers are multithreaded. When a server receives a message, it services the message using a separate thread. This allows the server to service several concurrent requests.

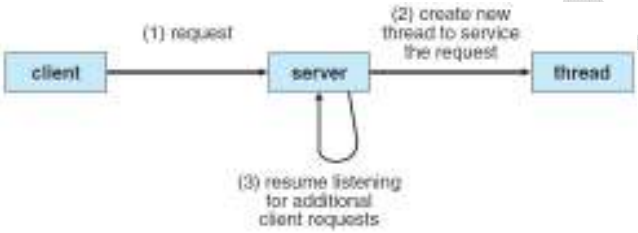


Figure 3.12 Multithreaded server architecture.

Finally, most operating-system kernels are now multithreaded. Several threads operate in the kernel, and each thread performs a specific task, such as managing devices, managing memory, or interrupt handling. For example, Solaris has a set of threads in the kernel specifically for interrupt handling;

Linux uses a kernel thread for managing the amount of free memory in the system.

3.6.2 Benefits

The benefits of multithreaded programming can be broken down into four major categories:

1. **Responsiveness.**

Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user. This quality is especially useful in designing user interfaces. For instance, consider what happens when a user clicks a button that results in the performance of a time-consuming operation. A single-threaded application would be unresponsive to the user until the operation had completed. In contrast, if the time-consuming operation is performed in a separate thread, the application remains responsive to the user.

2. **Resource sharing.**

Processes can only share resources through techniques such as shared memory and message passing. Such techniques must be explicitly arranged by the programmer. However, threads share the memory and the resources of the process to which they belong by default.

The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

3. Economy.

Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is significantly more time consuming to create and manage processes than threads. In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

Figure 3.13 concurrent executions on a single-core system.

4. Scalability.

The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

3.6.3 User and Kernel Threads

Support for threads may be provided at either user level for *user threads* or by kernel for *kernel threads.*

User threads are supported above kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Since the kernel is unaware of user-level threads, all thread creation and scheduling is done in the user space without the need for kernel intervention, and therefore are fast to create and manage. If the kernel is single threaded, then any user level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application. User thread libraries include POSIX Pthreads , Solaris 2 UI-threads, and Mach Cthreads.

Kernel threads are supported directly by the operating system. The kernel performs the scheduling, creation, and management in kernel space; the kernel level threads are hence slower to create and manage, compared to user level threads. However since the kernel is managing threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Windows NT, Windows 2000, Solaris, BeOS and Tru64 UNIX support kernel threads.

3.6.4 Multithreading Models

There are various models for mapping user-level threads to kernel-level threads. We

1. **The Many- To-One Model**
2. **The One-To-One Model**
3. **The Many-To-Many Model**

1 Many-to-One Model

The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire

process will block if a thread makes a blocking system call. Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multi-core systems.

Green threads—a thread library available for Solaris systems and adopted in early versions of Java—used the many-to-one model. However, very few systems continue to use the model because of its inability to take advantage of multiple processing cores.

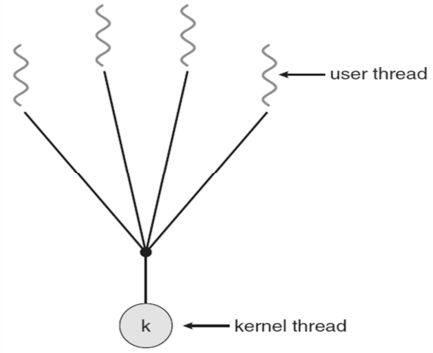


Figure 3.14 Many-to-one model.

2 One-to-One Model

The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread. Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system. Linux, along with the family of Windows operating systems, implement the one-to-one model.

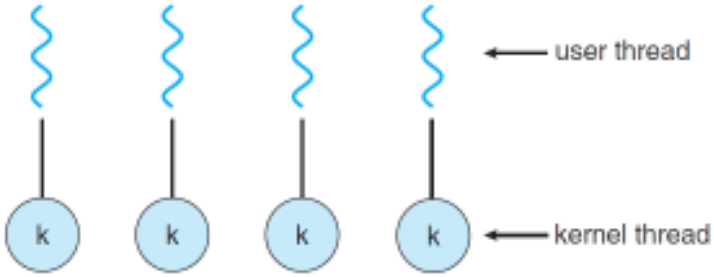


Figure 3.15 One-to-one model.

3 Many-to-Many Model

The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).

Let's consider the effect of this design on concurrency. Whereas the many-to-one model allows the developer to create as many user threads as she wishes, it does not result in true concurrency, because the kernel can schedule only one thread at a time. The one-to-one

model allows greater concurrency, but the developer has to be careful not to create too many threads within an application. When a thread performs a blocking system call, the kernel can schedule another thread for execution.

One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the **two-level model**. The Solaris operating system supported the two-level model in versions older than Solaris 9. However, beginning with Solaris 9, this system uses the one-to-one model.

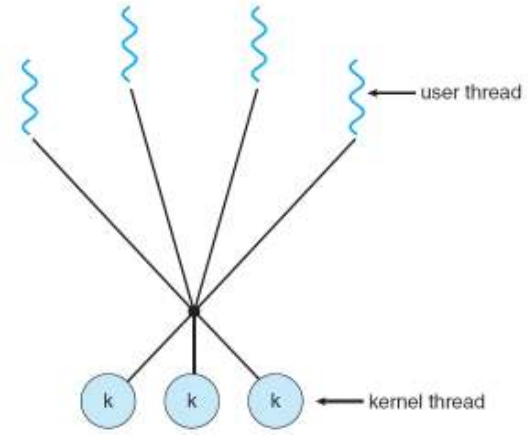


Figure 3.16 Many-to-many model

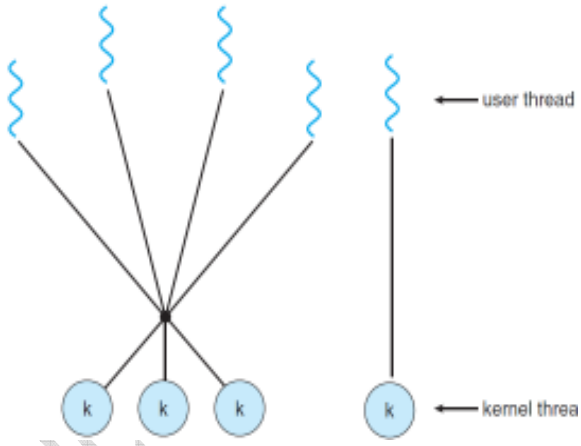


Figure 3.17 Two-level model.

3.7 Threading Issues

There are some issues which are necessary to consider in designing multithreaded programs.

3.7.1 The fork() and exec() System Calls

fork() system call is used to create a separate, duplicate process. The semantics of the fork() and exec() system calls change in a multithreaded program.

If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call. If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads. Which of the two versions of fork() to use depends on the application. If exec() is called immediately after forking, then duplicating all threads is unnecessary, as the program specified in the parameters to exec() will replace the process. In this instance, duplicating only the calling thread is appropriate.

If, however, the separate process does not call exec() after forking, the separate process should duplicate all threads.

3.7.2 Signal Handling

A **signal** is used in UNIX systems to notify a process that a particular event has occurred. A signal may be received either synchronously or asynchronously, depending on the source of and the reason for the event being signaled. All signals, whether synchronous or asynchronous, follow the same pattern:

1. A signal is generated by the occurrence of a particular event.
2. The signal is delivered to a process.
3. Once delivered, the signal must be handled.

Examples of synchronous signal include illegal memory access and division by 0. If a running program performs either of these actions, a signal is generated. Synchronous signals are delivered to the same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

When a signal is generated by an event external to a running process, that process receives the signal asynchronously. Examples of such signals include terminating a process with specific keystrokes (such as **<control><C>**) and having a timer expires. Typically, an asynchronous signal is sent to another process.

A signal may be ***handled*** by one of two possible handlers:

1. A default signal handler
2. A user-defined signal handler

Every signal has a **default signal handler** that the kernel runs when handling that signal. This default action can be overridden by a **user-defined signal handler** that is called to handle the signal. Signals are handled in different ways. Some signals (such as changing the size of a window) are simply ignored; others (such as an illegal memory access) are handled by terminating the program.

Handling signals in single-threaded programs is straightforward: signals are always delivered to a process. However, delivering signals is more complicated in multithreaded programs, where a process may have several threads. Where, then, should a signal be delivered?

In general, the following options exist:

1. Deliver the signal to the thread to which the signal applies.
2. Deliver the signal to every thread in the process.
3. Deliver the signal to certain threads in the process.
4. Assign a specific thread to receive all signals for the process.

The method for delivering a signal depends on the type of signal generated. For example, synchronous signals need to be delivered to the thread causing the signal and not to other threads in the process. However, the situation with asynchronous signals is not as clear. Some asynchronous signals—such as a signal that terminates a process (<control><C>, for example)—should be sent to all threads.

The standard UNIX function for delivering a signal is

kill(pid t pid, int signal)

This function specifies the process (pid) to which a particular signal (signal) is to be delivered. Most multithreaded versions of UNIX allow a thread to specify which signals it will accept and which it will block. Therefore, in some cases, an asynchronous signal may be delivered only to those threads that are not blocking it. However, because signals need to be handled only once, a signal is typically delivered only to the first thread found that is not blocking it. POSIX Pthreads provides the following function, which allows a signal to be delivered to a specified thread (tid):

Pthread_kill(pthread t tid, int signal)

Although Windows does not explicitly provide support for signals, it allows us to emulate them using **asynchronous procedure calls (APCs)**. The APC facility enables a user thread to specify a function that is to be called when the user thread receives notification of a particular event. As indicated by its name, an APC is roughly equivalent to an asynchronous signal in UNIX. However, whereas UNIX must contend with how to deal with signals in a multithreaded environment, the APC facility is more straightforward, since an APC is delivered to a particular thread rather than a process.

3.7.3 Thread Cancellation

Thread cancellation involves terminating a thread before it has completed. For example, if multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled. Another situation might occur when a user presses a button on a web browser that stops a web page from loading any further. Often, a web page loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the **target thread**. Cancellation of a target thread may occur in two different scenarios:

1. Asynchronous cancellation. One thread immediately terminates the target thread.

2. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

The difficulty with cancellation occurs in situations where resources have been allocated to a canceled thread or where a thread is canceled while in the midst of updating data it is sharing with other threads. This becomes especially troublesome with asynchronous cancellation. Often, the operating system will reclaim system resources from a canceled thread but will not reclaim all resources. Therefore, canceling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation, in contrast, one thread indicates that a target thread is to be canceled, but cancellation occurs only after the target thread has checked a flag to determine whether or not it should be canceled. The thread can perform this check at a point at which it can be canceled safely.

3.7.4 Thread-Local Storage

Threads belonging to a process share the data of the process. Indeed, this data sharing provides one of the benefits of multithreaded programming. However, in some circumstances, each thread might need its own copy of certain data. We will call such data **thread-local storage** (or **TLS**.) For example, in a transaction-processing system, we might service each transaction in a separate thread. Furthermore, each transaction might be assigned a unique identifier. To associate each thread with its unique identifier, we could use thread-local storage.

It is easy to confuse TLS with local variables. However, local variables are visible only during a single function invocation, whereas TLS data are

visible across function invocations. In some ways, TLS is similar to static data. The difference is that TLS data are unique to each thread. Most thread libraries—including Windows and Pthreads—provide some form of support for thread-local storage; Java provides support as well.

3.7.5 Scheduler Activations

A final issue to be considered with multithreaded programs concerns communication between the kernel and the thread library, which may be required by the many-to-many and two-level models discussed in Section 4.3.3. Such coordination allows the number of kernel threads to be dynamically adjusted to help ensure the best performance.

Many systems implementing either the many-to-many or the two-level model place an intermediate data structure between the user and kernel threads.

This data structure typically known as a **lightweight process**, or **LWP**. To the user-thread library, the LWP appears to be a virtual processor on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is

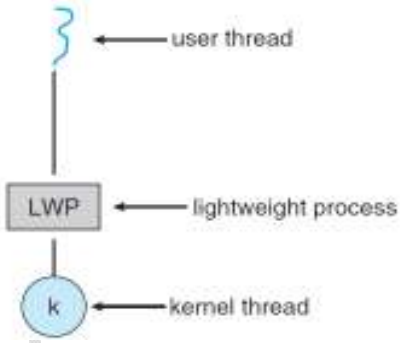


Figure 3.18: Lightweight process (LWP).

kernel threads that the operating system schedules to run on physical processors. If a kernel thread blocks the LWP blocks as well. Up the chain, the user-level thread attached to the LWP also blocks. An application may require any number of LWPs to run efficiently. Consider a CPU-bound application running on a single processor. In this scenario, only one thread can run at a time, so one LWP is sufficient. An application that is I/O-intensive may require multiple LWPs to execute, however. Typically, an LWP is required for each concurrent blocking system call. Suppose, for example, that five different file-read requests occur simultaneously. Five LWPs are needed, because all could be waiting for I/O completion in the kernel. If a process has only four LWPs, then the fifth request must wait for one of the LWPs to return from the kernel.

One scheme for communication between the user-thread library and the kernel is known as **scheduler activation**. It works as follows: The kernel provides an application with

a set of virtual processors (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as an **upcall**. Upcalls are handled by the thread library with an **upcall handler**, and upcall handlers must run on a virtual processor.

One event that triggers an upcall occurs when an application thread is about to block. In this scenario, the kernel makes an upcall to the application informing it that a thread is about to block and identifying the specific thread. The kernel then allocates a new virtual processor to the application. The application runs an upcall handler on this new virtual processor, which saves the state of the blocking thread and relinquishes the virtual processor on which the blocking thread is running. The upcall handler then schedules another thread that is eligible to run on the new virtual processor. When the event that the blocking thread was waiting for occurs, the kernel makes another upcall to the thread library informing it that the previously blocked thread is now eligible to run. The upcall handler for this event also requires a virtual processor, and the kernel may allocate a new virtual processor or preempt one of the user threads and run the upcall handler on its virtual processor. After marking the unblocked thread as eligible to run, the application schedules an eligible thread to run on an available virtual processor.

Summary of Threads:

A thread is a flow of control within a process. A multithreaded process contains several different flows of control within the same address space. The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and scalability factors, such as more efficient use of multiple processing cores.

User-level threads are threads that are visible to the programmer and are unknown to the kernel. The operating-system kernel supports and manages kernel-level threads. In general, user-level threads are faster to create and manage than are kernel threads, because no intervention from the kernel is required.

Three different types of models relate user and kernel threads. The many-to- one model maps many user threads to a single kernel thread. The one-to-one model maps each user thread to a corresponding kernel thread. The many-to-many model multiplexes many user threads to a smaller or equal number of kernel threads.

Most modern operating systems provide kernel support for threads. These include Windows, Mac OS X, Linux, and Solaris. Thread libraries provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: POSIX Pthreads, Windows threads, and Java threads.

In addition to explicitly creating threads using the API provided by a library, we can use implicit threading, in which the creation and management of threading is transferred to compilers and run-time libraries. Strategies for implicit threading include thread pools, OpenMP, and Grand Central Dispatch.

Multithreaded programs introduce many challenges for programmers, including the semantics of the fork() and exec() system calls. Other issues include signal handling, thread cancellation, thread-local storage, and scheduler activations.

Chapter 04

CPU Scheduling

4.1 Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uni-processor system, only one process may run at a time; any other processes much wait until the CPU is free and can be rescheduled. In multiprogramming, a process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. Multiprogramming entails productive usage of this time. When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process. Almost all computer resources are scheduled before use.

4.1.1 Life of a Process

As shown in Figure 4.1, process execution consists of a cycle of CPU execution and I/O wait. Processes alternates between these two states. Process execution begins with a CPU burst. An I/O burst follows that, and so on. Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst. An I/O bound program would typically have many very short CPU bursts. A CPU bound program might have a few very long CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm. Figure 4.2 shows results on an empirical study regarding the CPU bursts of processes. The study shows that most of the processes have short CPU bursts of 2-3 milliseconds.

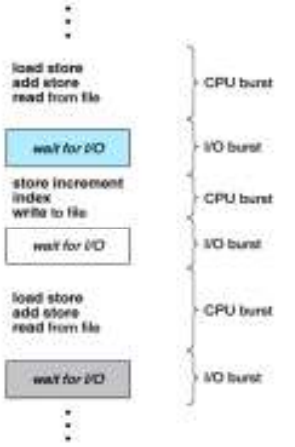


Figure 4.1 Alternating sequence of CPU and I/O bursts.

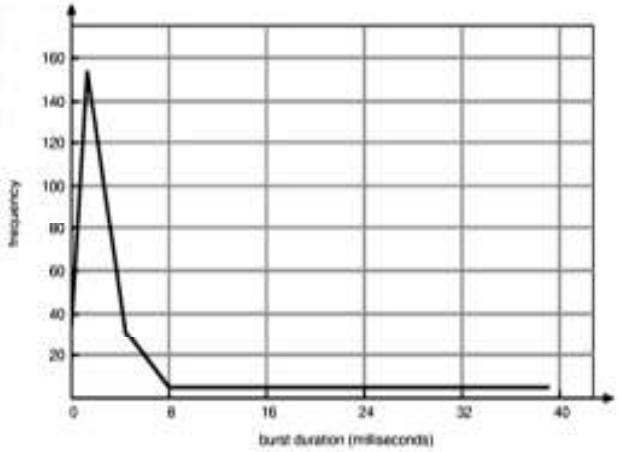


Figure 4.2 Histogram of CPU-burst Times

4.2 CPU–I/O Burst Cycle

The success of CPU scheduling depends on an observed property of processes: process execution consists of a **cycle** of CPU execution and I/O wait. Processes alternate between these two states. Process execution begins with a **CPU burst**. That is followed by an **I/O burst**, which is followed by another CPU burst, then another I/O burst, and so on. Eventually, the final CPU burst ends with a system request to terminate execution. The

durations of CPU bursts have been measured extensively. Although they vary greatly from process to process and from computer to computer, they tend to have a frequency curve similar to that shown in Figure 4.2. The curve is generally characterized as exponential or hyper exponential, with a large number of short CPU bursts and a small number of long CPU bursts.

An I/O-bound program typically has many short CPU bursts. A CPU-bound program might have a few long CPU bursts. This distribution can be important in the selection of an appropriate CPU-scheduling algorithm.

4.3 CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the **short-term scheduler**, or CPU scheduler. The scheduler selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list. Conceptually, however, all the processes in the ready queue are lined up waiting for a chance to run on the CPU. The records in the queues are generally process control blocks (PCBs) of the processes.

4.4 Preemptive and Non-Preemptive Scheduling

CPU scheduling can take place under the following circumstances:

1. When a process switches from the running state to the waiting state (for example, an I/O request is being completed)
2. When a process switches from the running state to the ready state (for example when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In 1 and 4, there is no choice in terms of scheduling; a new process must be selected for execution. There is a choice in case of 2 and 3. When scheduling takes place only under 1 and 4, we say, scheduling is **non-preemptive**; otherwise the scheduling scheme is **preemptive**. Under non-preemptive scheduling once the CPU has been allocated to a process the process keeps the CPU until either it switches to the waiting state, finishes its CPU burst, or terminates. This scheduling method does not require any special hardware needed for preemptive scheduling.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data.

One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms are needed to coordinate access to shared data.

4.5 Dispatcher

The **dispatcher** is a kernel module that takes control of the CPU from the current process and gives it to the process selected by the short-term scheduler. This function involves:

- Switching the context (i.e., saving the context of the current process and restoring the context of the newly selected process)
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**

4.6 Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. Which characteristics are used for comparison can make a substantial difference in which algorithm is judged to be best. The criteria include the following:

CPU utilization: We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent
(for a heavily loaded system).

Throughput: If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called **throughput**. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

Turnaround time: From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time: The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. It affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time: In an interactive system, turnaround time may not be the best criterion. Often, a process can produce some output fairly early and can continue computing new results while previous results are being output to the user. Thus, another measure is the time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device.

4.7 Scheduling Algorithms

CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms. In this section, we describe several of them.

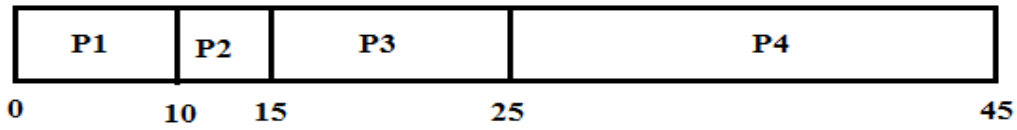
4.7.1. First-Come, First-Served Scheduling:

By far the simplest CPU-scheduling algorithm is the **first-come, first-served (FCFS)** scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long. Consider the following set of processes that arrive at time 0,

Process	Burst Time(milliseconds)
P1	10
P2	5
P3	10
P4	20

If the processes arrive in the order *P*1, *P*2, *P*3, and are served in FCFS order,

Gantt chart:



Turnaround Time:

Turnaround time is the total time span a process sped in the system. It can be calculated by subtracting arrival time from termination time.

Process	Finish Time – Arrival time	Waiting Time
P1	10-0	10
P2	15-0	15
P3	25-0	55
P4	45-0	45

The average turnaround time = (10+15+25+45)/4 = 23.57 milliseconds.

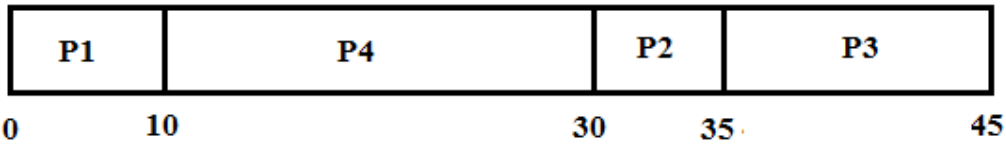
Waiting Time:

Waiting Time of each process can be calculated by subtracting finishing time of the process from its arrival time in this way.

Process	Start Time – Arrival time	Waiting Time
P1	0-0	0
P2	10-0	10
P3	15-0	15
P4	25-0	25

The average waiting time = (0+10+15+25)/4 = 12.5 milliseconds.

If the processes arrive in the order *P*1, *P*4, *P*2, P3 however, the results will be as shown in the following Gantt chart:



Process	Start Time – Arrival time	Waiting Time
P1	0-0	0
P2	10-0	10
P3	30-0	30
P4	35-0	35

The average waiting time = (0+10+30+35)/4 = 18.75 milliseconds.

In FCFS, if the processes with large burst time are moved to upper order, it will increase the average waiting time greatly.

In addition, consider the performance of FCFS scheduling in a dynamic situation. Assume we have one CPU bound process and many I/O-bound processes. As the processes flow around the system, the following scenario may result. The CPU-bound process will get and hold the CPU. During this time, all the other processes will finish their I/O and will move into the ready queue, waiting for the CPU. While the processes wait in the ready queue, the I/O devices are idle. Eventually, the CPU-bound process finishes its CPU burst and moves to an I/O device. All the I/O-bound processes, which have short CPU bursts, execute quickly and move, back to the I/O queues. At this point, the CPU sits idle. The CPU-bound process will then move back to the ready queue and be allocated the CPU. Again, all the I/O processes end up waiting in the ready queue until the CPU-bound process is done. There is a **convoy effect** as all the other processes wait for the one big process to get off the CPU. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

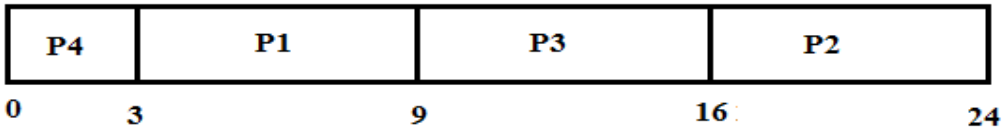
FCFS scheduling algorithm is **non-preemptive.** Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O. The FCFS algorithm is thus particularly troublesome for time-sharing systems, where it is important that each user get a share of the CPU at regular intervals. It would be disastrous to allow one process to keep the CPU for an extended period.

4.7.2. Shortest-Job-First Scheduling

A different approach to CPU scheduling is the shortest-job-first (SJF) scheduling algorithm. This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie. Note that a more appropriate term for this scheduling method would be the shortest-next-CPU-burst algorithm, because scheduling depends on the length of the next CPU burst of a process, rather than its total length. As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time(milliseconds)
P1	6
P2	8
P3	7
P4	3

Gantt chart:



Turnaround Time:

Process	Finish Time – Arrival time	Waiting Time
P1	9-0	9
P2	24-0	24
P3	16-0	16
P4	3-0	3

The average turnaround time = (9+24+16+3)/4 = 13 milliseconds.

Waiting Time:

Process	Start Time – Arrival time	Waiting Time
P1	3-0	3
P2	16-0	16
P3	9-0	9
P4	0-0	0

The Average Waiting Time = (3 + 16 + 9 + 0)/4 = 7 milliseconds.

By comparison, with FCFS scheduling scheme, the average waiting time would be 10.25 milliseconds.

The SJF scheduling algorithm is optimal; it gives the minimum average waiting time for a given set of processes. Moving a short process before long one decrease the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.

The real difficulty with the SJF algorithm is knowing the length of the next CPU request. For long-term (job) scheduling in a batch system, we can use the process time limit that a user specifies when he submits the job. In this situation, users are motivated to estimate the process time limit accurately, since a lower value may mean faster response but too low a value will cause a time-limit-exceeded error and require resubmission. SJF scheduling is used frequently in long-term scheduling.

Although the SJF algorithm is optimal, it cannot be implemented at the level of short-term CPU scheduling. With short-term scheduling, there is no way to know the length of the next CPU burst. One approach to this problem is to try to approximate SJF scheduling. We may not know the length of the next CPU burst, but we may be able to predict its value. It is expected that next CPU burst will be similar in length to the previous ones. By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.

4.7.3. Shortest-Remaining-Time-First:

The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The

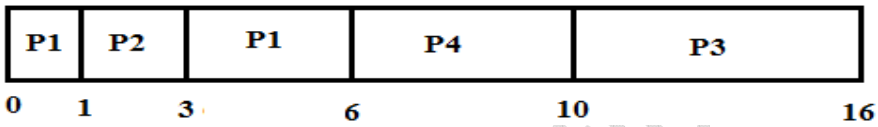
next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process, whereas a non-preemptive SJF algorithm will allow the currently running process to finish its CPU burst.

Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first** scheduling.

For Example:

Process	Arrival Time	Burst Time(milliseconds)
P1	0	4
P2	1	2
P3	2	6
P4	3	4

Gantt chart:



First of All we have to calculate turnaround time. Turnaround time can be calculated by subtracting arrival time from finish time of the process.

Turnaround Time:

Process	Finish Time – Arrival time	Waiting Time
P1	6-0	6
P2	3-1	2
P3	16-2	14
P4	10-3	7

The average turnaround time = (6+2+14+7)/4 = 7.25 milliseconds.

Waiting time:

Waiting time can be calculated by subtracting total execution time from its turnaround time.

Process	Turnaround Time – Burst	Waiting Time
P1	6-4	2
P2	2-2	0
P3	14-6	8
P4	7-4	3

Average waiting Time = (2+0+8+3)/4 = 3.25 millisecond

4.7.4 Priority Scheduling

In such scheduling, a priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority, and vice versa.

Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest

priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. Here, we assume that low numbers represent high priority.

As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, · · ·, P5, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

Gantt chart:

Waiting Time:

Process	Start Time – Arrival time	Waiting Time
P1	6-0	6
P2	0-0	0
P3	16-0	16
P4	18-0	18
P5	1-0	1

Average Waiting Time = (6+0+16+18+1)/5 = 8.2 milliseconds

Priorities can be defined either internally or externally. Internally defined priorities use some measurable quantity or quantities to compute the priority of a process. For example, time limits, memory requirements, the number of open files, and the ratio of average I/O burst to average CPU burst have been used in computing priorities. External priorities are set by criteria outside the operating system, such as the importance of the process, the type and amount of funds being paid for computer use, the department sponsoring the work, and other, often political, factors.

Priority scheduling can be either **preemptive** or **non-preemptive**. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. 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**. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. (Rumor has it that

when they shut down the IBM 7094 at MIT in 1973, they found a low-priority process that had been submitted in 1967 and had not yet been run.)

A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed. In fact, it would take no more than 32 hours for a priority-127 process to age to a priority-0 process.

4.7.5 Round-Robin Scheduling

The **round-robin (RR)** scheduling algorithm is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes. A small unit of time, called a **time quantum** or **time slice**, is defined. A time quantum is generally from 10 to 100 milliseconds in length. The ready queue is treated as a circular queue.

The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum. To implement RR scheduling, we again treat the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.

One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue. If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.

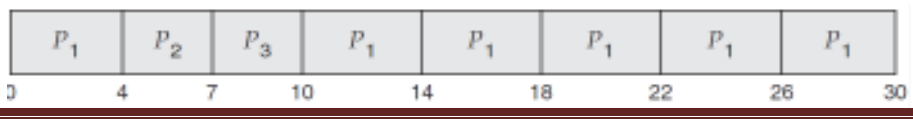
The average waiting time under the RR policy is often long. Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P1	24
P2	3
P3	3

If we use a **time quantum = 4 milliseconds**,

Then process P_1 gets the first 4milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P_2. Process P_2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P_3. Once each process has received 1 time quantum, the CPU is returned to process P_1 for an additional time quantum. The resulting RR schedule is as follows:

Gantt chart:



Turnaround Time:

Process	Finish Time – Burst	Turnaround Time
P1	30-24	6
P2	7-3	4
P3	10-3	7

Waiting Time:

Waiting time can be calculated by subtracting arrival time from turnaround time

Process	Turnaround Time – Arrival time	Waiting Time
P1	6-0	6
P2	4-0	4
P3	7-0	7

Average Waiting Time = (6+4+7) /3=5.67

In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. The RR scheduling algorithm is thus preemptive. If there are n processes in the ready queue and the time quantum is q, then each process gets $1/n$ of the CPU time in chunks of at most q time units.

Each process must wait no longer than $(n - 1) \times q$ time units until its next time quantum. For example, with five processes and a time quantum of 20 milliseconds, each process will get up to 20 milliseconds every 100 milliseconds.

The performance of the RR algorithm depends heavily on the size of the time quantum. At one extreme, if the time quantum is extremely large, the RR policy **is** the same as the FCFS policy. In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches. Assume, for example, that we have only one process of 10 time units. If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead. If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch. If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process accordingly (Figure 4.3).

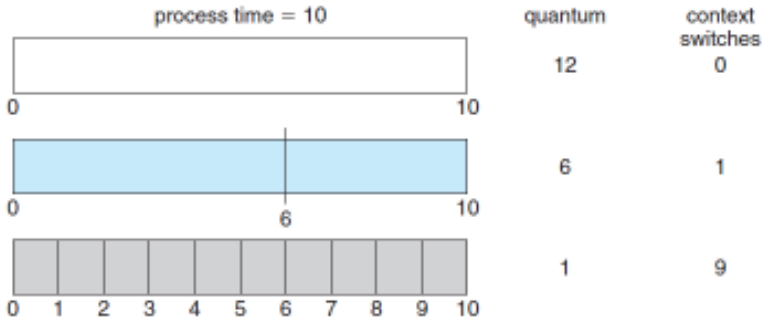
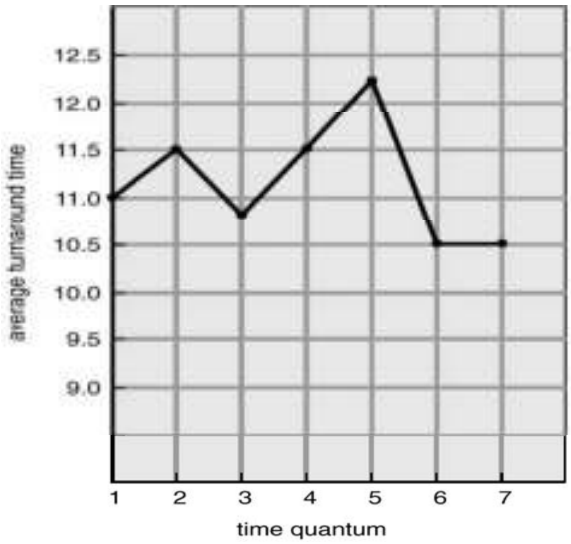


Figure 4.3 How a smaller time quantum increases context switches.



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Figure 4.4 Turnaround time versus quantum size

Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching. In practice, most modern systems have time quanta ranging from 10 to 100 milliseconds. The time required for a context switch is typically less than 10 microseconds; thus, the context-switch time is a small fraction of the time quantum.

Turnaround time also depends on the size of the time quantum. The average turnaround time of a set of processes does not necessarily improve as the time-quantum size increases. In general, the average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum. For example, given three processes of 10 time units each and a quantum of 1 time unit, the average turnaround time is 29. If the time quantum is 10, however, the average turnaround time drops to 20. If context-switch time is added in, the average turnaround time increases even more for a smaller time quantum, since more context switches are required.

Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy. A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

4.7.5 Multilevel Queue Scheduling

Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups. For example, a common division is made between **foreground** (or **interactive**) processes and **background** (or **batch) processes**.

These two types of processes have different response time requirements and so might have different scheduling needs. In addition, foreground processes may have priority over background processes.

A **multilevel queue-scheduling algorithm** partitions the ready queue into several separate queues, as shown in Figure 16.5. Each queue has its own priority and scheduling algorithm. Processes are permanently assigned to one queue, generally based o some property of the process, such as memory size, process priority or process type. In addition, there must

be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling i.e., serve all from foreground then from background. Another possibility is to time slice between queues. Each queue gets a certain portion of the CPU time, which it can then schedule among the various processes in its queue, e.g., 80% to foreground in RR and 20% to background in FCFS. Scheduling across queues prevents starvation of processes in lower-priority queues.

Let's look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:

1. System processes
2. Interactive processes
3. Interactive editing processes
4. Batch processes
5. Student processes

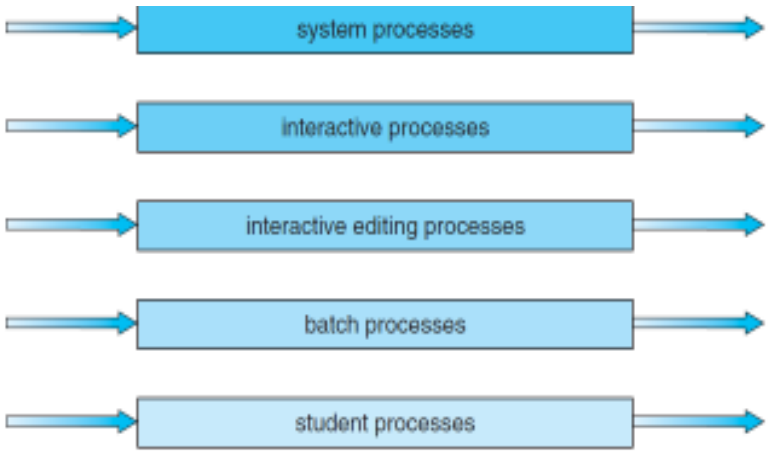


Figure 4.5 Multilevel queue scheduling.

Each queue has absolute priority over lower-priority queues. No process in the batch queue, for example, could run unless the queues for system processes, interactive processes, and interactive editing processes were all empty. If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes. For instance, in the foreground–background queue example, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, while the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

4.7.6 Multilevel Feedback Queue Scheduling

Multilevel feedback queue scheduling allows a process to move between queues. The idea is to separate processes with different CPU burst characteristics. 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 o a higher priority queue. This form of aging prevents starvation.

In general, a multi-level feedback queue scheduler is defined by the following parameters:

- Number of queues
- Scheduling algorithm for each queue
- Method used to determine when to upgrade a process to higher priority queue
- Method used to determine when to demote a process

- Method used to determine which queue a process enters when it needs service

Figure 4.6 shows an example multilevel feedback queue scheduling system with the ready queue partitioned into three queues. In this system, processes with next CPU bursts less than or equal to 8 time units are processed with the shortest possible wait times, followed by processes with CPU bursts greater than 8 but no greater than 16 time units. Processes with CPU greater than 16 time units wait for the longest time.

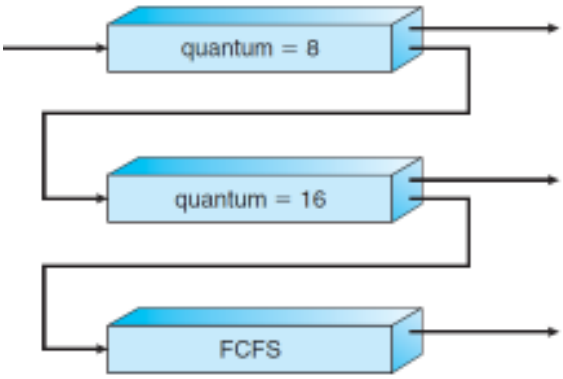


Figure 4.6 Multilevel Feedback Queues Scheduling

4.8 Thread Scheduling

Threads can be either user-level or kernel-level threads. If an operating systems supports threads, then it has to schedule kernel-level threads instead of processes, that are being scheduled by the operating system. User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP). There are some scheduling issues involving user-level and kernel-level threads.

4.8.1 Contention Scope

One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process contention scope (PCS)**, since competition for the CPU takes place among threads belonging to the same process. To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope (SCS)**.

Competition for the CPU with SCS scheduling takes place among all threads in the system. Systems using the one-to-one model, such as Windows, Linux, and Solaris, schedule threads using only SCS.

Typically, PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread. It is important to note that PCS will typically preempt the thread currently running in favor of a higher-priority thread; however, there is no guarantee of time slicing among threads of equal priority.

4.9 Multiple-Processor Scheduling

We focused on the problems of scheduling the CPU in a system with a single processor. If multiple CPUs are available, **load sharing** becomes possible—but scheduling problems become correspondingly more complex. Many possibilities have been tried; and as we saw with single processor

CPU scheduling, there is no one best solution.

There are several concerns in multiprocessor scheduling. We concentrate on systems in which the processors are identical—homogeneous —in terms of their functionality. Any available processor can be used to run any process in the queue. Even with homogeneous multiprocessors, there are sometimes limitations on scheduling. Consider a system with an I/O device attached to a private bus of one processor. Processes that wish to use that device must be scheduled to run on that processor.

4.9.1 Approaches to Multiple-Processor Scheduling

One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server. The other processors execute only user code.

This **asymmetric multiprocessing** is simple because only one processor accesses the system data structures, reducing the need for data sharing. A second approach uses **symmetric multiprocessing (SMP)**, where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes. Regardless, scheduling proceeds by having the scheduler for each processor examine the ready queue and select a process to execute. If we have multiple processors trying to access and update a common data structure, the scheduler must be programmed carefully. We must ensure that two separate processors do not choose to schedule the same process and that processes are not lost from the queue. Virtually all modern operating systems support SMP, including Windows, Linux, and Mac OS X.

4.9.2 Processor Affinity

Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most SMP systems try to avoid migration of processes from one processor to another and instead attempt to keep a process running on the same processor. This is known as **processor affinity**—that is, a process has an affinity for the processor on which it is currently running. Processor affinity takes several forms. When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—it is called **soft affinity**.

The operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors. In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it may run. Many systems provide both soft and hard affinity. For

example, Linux implements soft affinity, but it also provides the sched_setaffinity() system call, which supports hard affinity.

The main-memory architecture of a system can affect processor affinity issues. Figure 4.7 illustrates an architecture featuring non-uniform memory access (NUMA), in which a CPU has faster access to some parts of main memory than to other parts. Typically, this occurs in systems containing combined CPU and memory boards. The CPUs on a board can access the memory on that board faster than they can access memory on other boards in the system.

If the operating system's CPU scheduler and memory-placement algorithms work together, then a process that is assigned affinity to a particular CPU can be allocated memory on the board where that CPU resides. This example also shows that operating systems are frequently not as cleanly defined and implemented as described in operating-system textbooks. Rather, the "solid lines" between sections of an operating system are frequently only "dotted lines," with algorithms creating connections in ways aimed at optimizing performance and reliability.

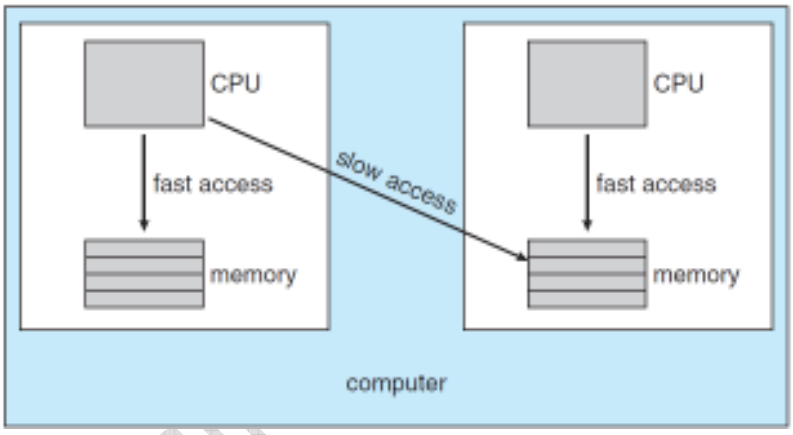


Figure 4.7 NUMA and CPU scheduling.

4.9.3 Load Balancing

On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor. Otherwise, one or more processors may sit idle while other processors have high workloads, along with lists of processes awaiting the CPU. **Load balancing** attempts to keep the workload evenly distributed across all processors in an SMP system. It is important to note that load balancing is typically necessary only on systems where each processor has its own private queue of eligible processes to execute. On systems with a common run queue, load balancing is often unnecessary, because once a processor becomes idle, it immediately extracts a runnable process from the common run queue. In most contemporary operating systems supporting SMP, each processor does have a private queue of eligible processes. There are two general approaches to load balancing: **push migration** and **pull migration**. With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors. Pull migration occurs when an idle processor pulls a waiting task from a busy processor. Push and pull migration need not be mutually exclusive and are in fact often

implemented in parallel on load-balancing systems. For example, the Linux scheduler and the ULE scheduler available for FreeBSD systems implement both techniques.

Load balancing often counteracts the benefits of processor affinity. That is, the benefit of keeping a process running on the same processor is that the process can take advantage of its data being in that processor's cache memory. Either pulling or pushing a process from one processor to another removes this benefit. As is often the case in systems engineering, there is no absolute rule concerning what policy is best. Thus, in some systems, an idle processor always pulls a process from a non-idle processor. In other systems, processes are moved only if the imbalance exceeds a certain threshold.

4.9.4 Multicore Processors

SMP systems have allowed several threads to run concurrently by providing multiple physical processors. However, a recent practice in computer hardware has been to place multiple processor cores on the same physical chip, resulting in a **multicore processor**. Each core maintains its architectural state and thus appears to the operating system to be a separate physical processor.

SMP systems that use multicore processors are faster and consume less power than systems in which each processor has its own physical chip. Multicore processors may complicate scheduling issues. Researchers have discovered that when a processor accesses memory, it spends a significant amount of time waiting for the data to become available. This situation, known as a **memory stall**, may occur for various reasons, such as a cache miss (accessing data that are not in cache memory).

4.10Algorithm Evaluation

To select an algorithm, many factors must be considered, defining their relative importance, such as:

- Maximum CPU utilization under the constraint that maximum response time is 1 second.
- Maximize throughput such that turnaround time is (on average) linearly proportional to total execution time.

Once the selection criteria have been defined, we want to evaluate the algorithms under consideration. Scheduling algorithms can be evaluated by using the following techniques:

4.10.1 Analytic Evaluation

A scheduling algorithm and some system workload are used to produce a formula or number, which gives the performance of the algorithm for that workload. Analytic evaluation falls under two categories: Deterministic modeling, Queuing Models

4.10.1.1 Deterministic modeling:

Deterministic modeling is a type of analytic evaluation. This method takes a particular predetermined workload and defines the performance of each algorithm for workload in terms of numbers for parameters such as average wait time, average turnaround time, and average response time. Gantt charts are used to show executions of processes. We have been using this technique to explain the working of an algorithm as well as to evaluate the performance of an algorithm with a given workload. Deterministic modeling is simple and

fast. It gives exact numbers, allowing the algorithms to be compared. However it requires exact numbers for input and its answers apply to only those cases.

For example, assume that we have the workload shown below. All five processes arrive at time 0, in the order given, with the length of the CPU burst given in milliseconds:

Process	Burst
P1	10
P2	29
P3	3
P4	7
P5	12

Consider the FCFS, SJF, and RR (quantum = 10 milliseconds) scheduling algorithms for this set of processes. Which algorithm would give the minimum average waiting time?

FCFS Scheduling

The Average Waiting Time = (0 + 10 + 39 + 42 + 49)/5 = 28 milliseconds.

Non-preemptive SJF scheduling

The Average Waiting Time = (10 + 32 + 0 + 3 + 20)/5 = 13 milliseconds.

Round Robin Scheduling:

The average waiting time = (0 + 32 + 20 + 23 + 40)/5 = 23 milliseconds.

The main uses of deterministic modeling are in describing scheduling algorithms and providing examples. In cases where we are running the same program over and over again and can measure the program's processing requirements exactly, we may be able to use deterministic modeling to select a scheduling algorithm. Furthermore, over a set of examples, deterministic modeling may indicate trends that can then be analyzed and proved separately. For example, it can be shown that, for the environment described (all processes and their times available at time 0), the SJF policy will always result in the minimum waiting time.

4.10.1.2 Queuing Models

The computer system can be defined as a network of servers. Each server has a queue of waiting processes. The CPU is a server with its ready queue, as are I/O systems with their device queues. Knowing the arrival and service rates of processes for various servers, we can compute utilization, average queue length, average wait time, and so on. This kind of study is called **queuing-network analysis**. If n is the average queue length, W is the average waiting time in the queue, and let λ is the average arrival rate for new processes in the queue, then

$$n = \lambda * W$$

This formula is called the **Little's formula**, which is the basis of **queuing theory**, a branch of mathematics used to analyze systems involving queues and servers. At the moment, the classes of algorithms and distributions that can be handled by queuing analysis are fairly limited. The mathematics involved is complicated and distributions can be difficult to work with. It is also generally necessary to make a number of independent assumptions that may not be accurate. Thus so that they will be able to compute an answer, queuing models are often an approximation of real systems.

As a result, the accuracy of the computed results may be questionable. The table 4.1 shows the average waiting times and average queue lengths for the various scheduling

algorithms for a pre-determined system workload, computed by using Little's formula. The average job arrival rate is 0.5 jobs per unit time.

Table 4.1 Average Wait Time and Average Queue Length Computed With Little's Equation

Algorithm	Average Wait Time $W = t_w$	Average Queue Length (n)
FCFS	4.6	2.3
SRTF	3.2	1.6
RR (q=1)	)7.0	3.5
RR (q=4)	6.0	3.0

4.10.2 Simulations

Simulations involve programming a model of the computer system, in order to get a more accurate evaluation of the scheduling algorithms. Software data structures represent the major components of the system. The simulator has a variable representing a clock; as this variable's value is increased, the simulator modifies the system state to reflect the activities of the devices, the processes and the scheduler. As the simulation executes, statistics that indicate algorithm performance are gathered and printed. Figure 4.8 shows the schematic for a simulation system used to evaluate the various scheduling algorithms.

Some of the major issues with simulation are:

- Expensive: hours of programming and execution time are required
- Simulations may be erroneous because of the assumptions about distributions used for arrival and service rates may not reflect a real environment

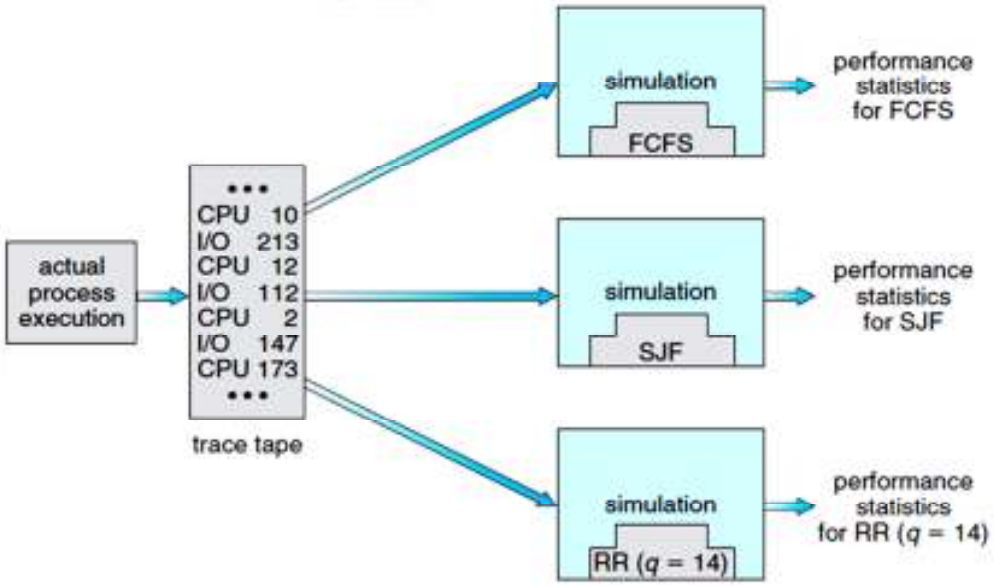


Figure 4.8 Evaluation of CPU schedulers by simulation.

4.10.3 Implementation

Even a simulation is of limited accuracy. The only completely accurate way to evaluate a scheduling algorithm is to code it, put it in the operating system and see how it

works. This approach puts the actual algorithm in the real system for evaluation under real operating conditions. The Open Source software licensing has made it possible for us to test various algorithms by implementing them in the Linux kernel and measuring their true performance.

The major difficulty is the cost of this approach. The expense is incurred in coding the algorithm and modifying the operating system to support it, as well as its required data structures. The other difficulty with any algorithm evaluation is that the environment in which the algorithm works will change.

Chapter 05

Process Synchronization

A **cooperating process** is one that can affect or be affected by other processes executing in the system. Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through files or messages. Share of logical address space is achieved through the use of threads.

Producer consumer Problem:

To illustrate the concept of communicating processes, let us consider the producer-consumer problem. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. To allow a producer and consumer to run concurrently, we must have available a buffer of items that can be filled by a producer and emptied by a consumer. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced. The bounded buffer problem assumes a fixed buffer size, and the consumer must wait if the buffer is empty and the producer must wait if the buffer is full, whereas the unbounded buffer places no practical limit on the size of the buffer. Figure 5.1 shows the problem in a diagram. This buffer may be provided by inter process communication or with the use of shared memory.

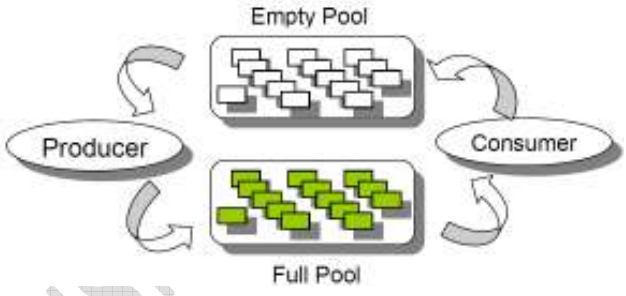


Figure 5.1 The producer-consumer problem

Figure 5.2 shows the shared buffer and other variables used by the producer and consumer processes.

```
#define BUFFER_SIZE 10

typedef struct

{

…

} item;

item buffer[BUFFER_SIZE];

int in=0;

int out=0;
```

Figure 5.2 Shared buffer and variables used by the producer and consumer processes

The shared buffer is implemented as a circular array with two logical pointers: in and out. The 'in' variable points to the next free position in the buffer; 'out' points to the first full position in the buffer. The buffer is empty when in==out, the buffer is full when ((in+1)%BUFFER_SIZE)==out. The code structures for the producer and consumer processes are shown in Figure 5.3.

Producer Process

```
while(1) {
/*Produce an item in nextProduced*/
while(((in+1)%BUFFER_SIZE)==out); /*do nothing*/
buffer[in]=nextProduced;
in=(in+1)%BUFFER_SIZE;
}
```

Consumer Process

```
while(1) {
while(in == out); //do nothing
nextConsumed=buffer[out];
out=(out+1)%BUFFER_SIZE;
/*Consume the item in nextConsumed*/
}
```

Figure 5.3 Code structures for the producer and consumer processes

5.1 Process Synchronization

Concurrent processes or threads often need access to shared data and shared resources. If there is no controlled access to shared data, it is often possible to obtain an inconsistent state of this data. Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes, and hence various process synchronization methods are used. In the producer-consumer problem the version only allows one item less than the buffer size to be stored, to provide a solution for the buffer to use its entire capacity of N items is not simple. The producer and consumer share data structure 'buffer' and use other variables shown below:

```
#define BUFFER_SIZE 10
typedef struct
{
...
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

The code for the producer process is:

```
while(1)
{
/*Produce an item in nextProduced*/
while(counter == BUFFER_SIZE); /*do nothing*/
buffer[in]=nextProduced;
in=(in+1)%BUFFER_SIZE;
counter++;
}
```

The code for the consumer process is:

```
while(1)
{
while(counter==0); //do nothing
nextConsumed=buffer[out];
out=(out+1)%BUFFER_SIZE;
counter--;
/*Consume the item in nextConsumed*/
}
```

Both producer and consumer routines may not execute properly if executed concurrently. Suppose that the value of the counter is 5, and that both the producer and the consumer execute the statement counter++ and counter- - concurrently. Following the execution of these statements the value of the counter may be 4,5,or 6! The only correct result of these statements should be counter= =5, which is generated if the consumer and the

producer execute separately. Suppose counter++ is implemented in machine code as the following instructions:

MOV R1, counter
INC R1
MOV counter, R1

Whereas counter- - maybe implemented as:

MOV R2, counter
DEC R2
MOV counter, R2

If both the producer and consumer attempt to update the buffer concurrently, the machine language statements may get interleaved. Interleaving depends upon how the producer and consumer processes are scheduled. Assume counter is initially 5. One interleaving of statements is:

producer: MOV R1, counter (R1 = 5)
INC R1 (R1 = 6)
consumer: MOV R2, counter (R2 = 5)
DEC R2 (R2 = 4)
producer: MOV counter, R1 (counter = 6)
consumer: MOV counter, R2 (counter = 4)

The value of count will be 4, where the correct result should be 5. The value of count could also be 6 if producer executes MOV counter, R1 at the end. The reason for this state is that we allowed both processes to manipulate the variable counter concurrently.

Race Condition:

A situation where several processes access and manipulate the same data concurrently and the outcome of the manipulation depends on the particular order in which the access takes place, is called a **race condition**. To guard against such race conditions, we require synchronization of processes. Concurrent transactions in a bank or in an airline reservation (or travel agent) office are a couple of other examples that illustrates the critical section problem. We show interleaving of two bank transactions, a deposit and a withdrawal. Here are the details of the transactions:

- Current balance = Rs. 50,000
- Check deposited = Rs. 10,000
- ATM withdrawn = Rs. 5,000

The codes for deposit and withdrawal are shown in Figure 5.4

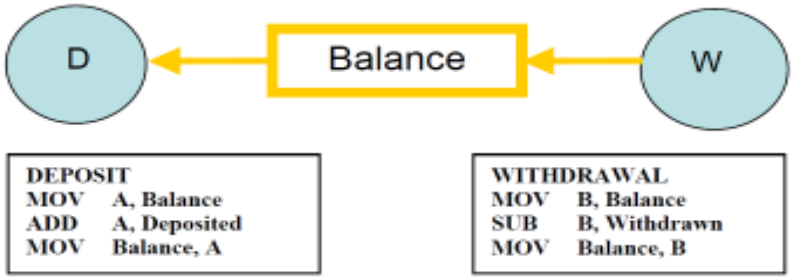


Figure 5.4 Bank transactions—deposit and withdrawal

Here is what may happen if the two transactions are allowed to execute concurrently, i.e., the transactions are allowed to interleave. Note that in this case the final balance will be Rs. 45,000, i.e., a loss of Rs. 5,000. If MOV Balance, A executes at the end, the result will be a gain of Rs. 5,000. In both cases, the final result is wrong.

Check Deposit:
MOV A, Balance // A = 50,000
ADD A, Deposited // A = 60,000
ATM Withdrawal:
MOV B, Balance // B = 50,000
SUB B, Withdrawn // B = 45,000
Check Deposit:
MOV Balance, A // Balance = 60,000
ATM Withdrawal:
MOV Balance, B // Balance = 45,000

The Critical Section Problem

Critical Section:

A piece of code in a cooperating process in which the process may updates shared data (variable, file, database, etc.).

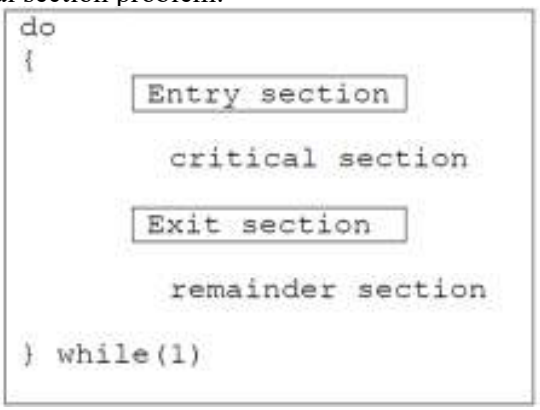
Critical Section Problem:

When a process executes code that manipulates shared data (or resource), the process is in its critical section (for that shared data). The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple processors). So each process must first request permission to enter its critical section. The section of code implementing this request is called the **entry section**. The remaining code is the **remainder section**. The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors).

There can be three kinds of solution to the critical section problem:

- **Software based solutions**
- **Hardware based solutions**
- **Operating system based solution**

Regardless of the type of solution, the structure of the solution should be as follows. The Entry and Exist sections comprise solution for the problem.



Software Based Solution to the Critical Section Problem

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

1. Mutual Exclusion

If process Pi is executing in its critical section, then no other process can be execute in their critical section.

2. Progress

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

3. Bounded Waiting

There exists a bound 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.

Assumptions

While formulating a solution, we must keep the following assumptions in mind:

- Assume that each process executes at a nonzero speed
- No assumption can be made regarding the relative speeds of the N processes.

Two general approaches are used to handle critical sections in operating systems:

1. **Preemptive kernels**
2. **Non-preemptive kernels**.

1. preemptive kernels

A preemptive kernel allows a process to be preempted while it is running in kernel mode.

2. **Non-preemptive kernels**.

A non-preemptive kernel does not allow a process running in kernel mode to be preempted; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

A non-preemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

Preemptive kernels carefully designed to ensure that shared kernel data are free from race conditions. Preemptive kernels are especially difficult to design for SMP architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.

Advantages of a preemptive kernel: A preemptive kernel may be **more responsive**, since there is less risk that a kernel-mode process will run for an **arbitrarily long period** before releasing the processor to waiting processes. Furthermore, a preemptive kernel is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

Peterson's Solution

Next, we illustrate a classic software-based solution to the critical-section problem known as Peterson's solution.

Because of the way modern computer architectures perform basic machine-language instructions, such as load and store, there are no guarantees that Peterson's solution will work correctly on such architectures. However, we present the solution because it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

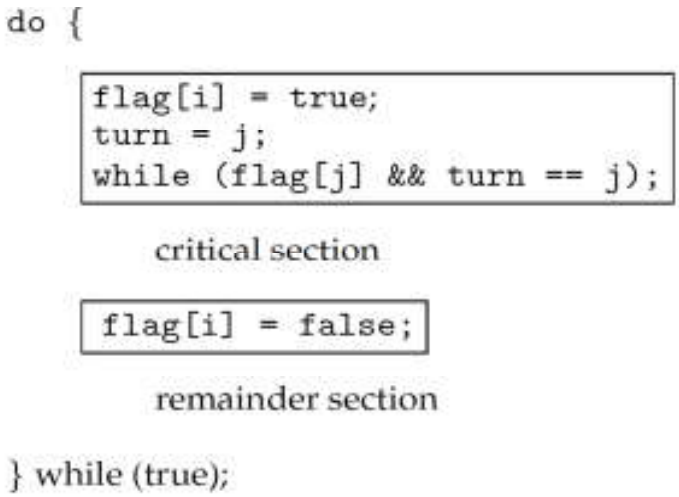


Figure 5.5 The structure of process *Pi* in Peterson's solution.

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered *P*0 and *P*1. For convenience, when presenting P_i, we use P_j to denote the other process; that is, j equals 1 − i.

Peterson's solution requires the two processes to share two data items:

int turn;

boolean flag[2];

The variable turn indicates whose turn it is to enter its critical section. That is, if turn == i, then process P_i is allowed to execute in its critical section. The flag array is used to indicate if a process is ready to enter its critical section.

For example, if flag[i] is true, this value indicates that P_i is ready to enter its critical section. With an explanation of these data structures complete, we are now ready to describe the algorithm shown in Figure 5.5.

To enter the critical section, process P_i first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so. If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last; the other will occur but will be overwritten immediately.

The eventual value of turn determines which of the two processes is allowed to enter its critical section first.

We now prove that this solution is correct. We need to show that:

1. Mutual exclusion is preserved.

2. The progress requirement is satisfied.

3. The bounded-waiting requirement is met.

To prove property 1, we note that each P_i enters its critical section only if either flag[j] == false or turn == i. Also note that, if both processes can be executing in their critical sections at the same time, then flag[0] == flag[1] == true. These two observations imply that *P*0 and *P*1 could not have successfully executed their while statements at about the

same time, since the—say, P_j—must have successfully executed the while statement, whereas P_i had to execute at least one additional statement ("turn == j"). However, at that time, flag[j] == true and turn == j, and this condition will persist as long as P_j is in its critical section; as a result, mutual exclusion is preserved.

To prove properties 2 and 3, we note that a process P_i can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j] == true and turn == j; this loop is the only one possible. If P_j is not ready to enter the critical section, then flag[j] == false, and P_i can enter its critical section. If P_j has set flag[j] to true and is also executing in its while statement, then either turn == i or turn == j. If turn == i, then *Pi* will enter the critical section. If turn == j, then P_j will enter the critical section. However, once P_j exits its critical section, it will reset flag[j] to false, allowing P_i to enter its critical section. If P_j resets flag[j] to true, it must also set turn to i. Thus, since *Pi* does not change the value of the variable turn while executing the while statement, P_i will enter the critical section (progress) after at most one entry by P_j (bounded waiting).

Hardware Solutions for the Critical Section Problem

There are some simple hardware (CPU) instructions that can be used to provide synchronization between processes and are available on many systems.

The critical section problem can be solved simply in a uni- processor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately this solution is not feasible in a multiprocessing environment, as disabling interrupts can be time consuming as the message is passed to all processors. This message passing delays entry into each critical section, and system efficiency decreases. Normally, access to a memory location excludes other accesses to that same location. Designers have proposed machine instructions that perform two operations atomically (indivisibly) on the same memory location (e.g., reading and writing). The execution of such an instruction is also mutually exclusive (even on Multiprocessors). They can be used to provide mutual exclusion but other mechanisms are needed to satisfy the other two requirements of a good solution to the critical section problem.

We can use these special instructions to solve the critical section problem. These instructions are TestAndSet (also known as TestAndSetLock; TSL) and Swap. The semantics of the TestAndSet instruction are as follows:

```
boolean TestAndSet(Boolean &target)
{
boolean rv=target;
target=true;
return rv;
}
```

The semantics simply say that the instruction saves the current value of 'target', set it to true, and returns the saved value. The important characteristic is that this

instruction is executed atomically. Thus if two TestAndSet instructions are executed simultaneously, they will be executed sequentially in some arbitrary order.

If the machine supports TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false. The structure of process Pi becomes:

```
Do
{
    while (TestAndSet(lock)) ;
Critical section
    lock=false;
Remainder section
} while(1);
```

The above TSL-based solution is no good because even though mutual exclusion and progress are satisfied, bounded waiting is not. The semantics of the Swap instruction, another atomic instruction, are, as expected, as follows:

```
boolean Swap(boolean &a, boolean &b)
{
boolean temp=a;
a=b;
b=temp;
}
```

If the machine supports the Swap instruction, mutual exclusion can be implemented as follows. A global Boolean variable lock is declared and is initialized to false. In addition each process also has a local Boolean variable key. The structure of process P_i is:

```
Do
{
    key=true;
    while(key == true)
    Swap(lock,key);
Critical section
    lock=false;
Remainder section
} while(1);
```

Just like the TSL-based solution shown in this section, the above Swap-based solution is not good because even though mutual exclusion and progress are satisfied, bounded waiting is not. In the next lecture, we will discuss a good solution for the critical section problem by using the hardware instructions.

Hardware Solutions

We discussed two possible solutions but realized that whereas both solutions satisfied the mutual exclusion and bounded waiting conditions, neither satisfied the progress condition. We now describe a solution that satisfies all three requirements of a solution to the critical section problem.

Algorithm

In this algorithm, we combine the ideas of the first two algorithms. The common data structures used by a cooperating process are:

boolean waiting[n];
boolean lock;

The structure of process Pi is:

```
Do
{
 waiting[i] = true;
 key = true;
 while (waiting[i] && key)
 key = TestAndSet(lock);
 waiting[i] = false;
Critical section
 j = (i+1) % n;
 while ((j!=i) && !waiting[j])
 j = (j+1)% n;
 if (j == i)
 lock = false;
 else
 waiting[j] = false;
Remainder section
} while(1);
```

These data structures are initialized to false. To prove that the mutual exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i]= = false or key = = false. The value of key can become false only if TestAndSet is executed. The first process to execute the TestAndSet instruction will find key= =false; all others must wait. The variable waiting[i] can only become false if another process leaves its critical section; only one waiting[i] is set to false, maintaining the mutual exclusion requirement.

To prove the progress requirement is met, we note that the arguments presented for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false or sets waiting[j] to false. Both allow a process that is waiting to enter its critical section to proceed. To prove that the bounded waiting requirement is met, we note that, when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+1, i+2, …, n-1, 0, 1, …, i-1). It designates the first process it sees that is in its entry section with waiting[j] =true as the next one to enter its critical section. Any process waiting to do so will enter its critical section within n-1 turns.

Semaphores

Hardware solutions to synchronization problems are not easy to generalize to more complex problems. To overcome this difficulty we can use a synchronization tool called

a semaphore. A **semaphore** S is an integer variable that, apart from initialization is accessible only through two standard atomic operations: wait and signal. These operations were originally termed P (for wait) and V (for signal). The classical definitions of wait and signal are:

```
wait(S) {
while(S<=0)
;// no op
S--;
}
```

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

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process is updating the value of a semaphore, other processes cannot simultaneously modify that same semaphore value. In addition, in the case of the wait(S), the testing of the integer value of S (S<=0) and its possible modification (S--) must also be executed without interruption.

We can use semaphores to deal with the n-process critical section problem. The n processes share a semaphore, **mutex** (standing for mutual exclusion) initialized to 1. Each process Pi is organized as follows:

```
do
{
    wait(mutex);

Critical section
    signal(mutex);

Remainder section
} while(1);
```

As was the case with the hardware-based solutions, this is not a good solution because even though it satisfies mutual exclusion and progress, it does not satisfy bounded wait.

In a uni-processor environment, to ensure atomic execution, while executing wait and signal, interrupts can be disabled. In case of a multi-processor environment, to ensure atomic execution is one can lock the data bus, or use a soft solution such as the Bakery algorithm.

The main disadvantage of the semaphore discussed in the previous section is that it requires **busy waiting**. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is

shared among many processes. Busy waiting wastes CPU cycles that some other process may be able to use productively. This type of semaphore is also called a **spinlock** (because the process spins while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

This is, spinlocks are useful when they are expected to be held for short times. The definition of semaphore should be modified to eliminate busy waiting.

This, when locks are expected to be held for short times, spinlocks are useful. To overcome the need for busy waiting, we can modify the definition of semaphore and the wait and signal operations on it. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU scheduling algorithm.) Such an implementation of a semaphore is as follows:

```
typedef struct {
int value;
struct process *L;
} semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore; it is added to the list of processes. A signal operation removes one process from the list of the waiting processes and awakens that process. The wait operation can be defined as:

```
void wait(semaphore S) {
S.value--;
if(S.value < 0) {
add this process to S.L;
block();
}
}
```

The signal semaphore operation can be defined as

```
void signal wait(semaphore S) {
S.value++;
if(S.value <= 0) {
remove a process P from S.L;
wakeup(P);
}
}
```

The block operation suspends the process that invokes it. The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls. The negative value of S. value indicates the number of processes waiting for the semaphore. A pointer in the PCB needed to maintain a queue of processes waiting for a semaphore. As mentioned before, the busy-waiting version is better when critical sections are small and queue-waiting version is better for long critical sections (when waiting is for longer periods of time).

Process Synchronization

You can use semaphores to synchronize cooperating processes. Consider, for example, that you want to execute statement B in Pj only after statement A has been executed in Pi. You can solve this problem by using a semaphore S initialized to 0 and structuring the codes for Pi and Pj as follows:

Pi	**Pj**
...	...
A;	wait(S);
signal(S);	B;
...	...

Pj will not be able to execute statement B until Pi has executed its statements A and signal(S). Here is another synchronization problem that can be solved easily using semaphores. We want to ensure that statement S1 in P1 executes only after statement S2 in P2 has executed, and statement S2 in P2 should execute only after statement S3 in P3 has executed. One possible semaphore-based solution uses two semaphores, A and B. Here is the solution.

semaphore A=0, B=0;

P1	**P2**	**P3**
...	...	...
wait(A);	wait(B);	S3;
S1;	S2;	signal(B);
signal(A);		
...	...	...

Problems with Semaphores

Here are some key points about the use of semaphores:

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinating processes.
- The wait(S) and signal(S) operations are scattered among several processes.

Hence, it is difficult to understand their effects.

- Usage of semaphores must be correct in all the processes.
- One bad (or malicious) process can fail the entire system of cooperating processes.

Incorrect use of semaphores can cause serious problems. We now discuss a few of

these problems.

Deadlocks and Starvation

A set of processes are said to be in a deadlock state if every process is waiting for an event that can be caused only by another process in the set. Here are a couple of examples of deadlocks in our daily lives.

- Traffic deadlocks
- One-way bridge-crossing

Starvation is infinite blocking caused due to unavailability of resources. Here is an example of a deadlock.

P0	**P1**
wait(S);	wait(Q);
wait(Q);	wait(S);
...	...
signal(S);	signal(Q);
signal(Q);	signal(S);
...	...

P0 and P1 need to get two semaphores, S and Q, before executing their critical sections. The following code structures can cause a deadlock involving P0 and P1. In this example, P0 grabs semaphore S and P1 obtains semaphore Q. Then, P0 waits for Q and P1 waits for S. P0 waits for P1 to execute signal(Q) and P1 waits for P0 to execute signal(S).

Kinds of Semaphores:

There are two kinds of semaphores:

1. **Counting semaphore**
2. **Binary semaphore**

Counting semaphore whose integer value can range over an unrestricted integer domain.
Binary semaphore whose integer value cannot be > 1; can be simpler to implement.

Let S be a counting semaphore. To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
int C;
```

Initially S1=1, S2=0, and the value of integer C is set to the initial value of the counting semaphore S. The wait operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C--;
if(C < 0) {
```

```
signal(S1);
wait(S2);
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
wait(S1);
C++;
if(C <= 0)
signal(S2);
else
signal(S1);
```

Classic Problems of Synchronization:

The three classic problems of synchronization are:

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining Philosophers Problem

Bounded Buffer Problem

The bounded-buffer problem, which was introduced in a previous lecture, is commonly used to illustrate the power of synchronization primitives. The solution assumes that the pool consists of n buffers, each capable of holding one item.

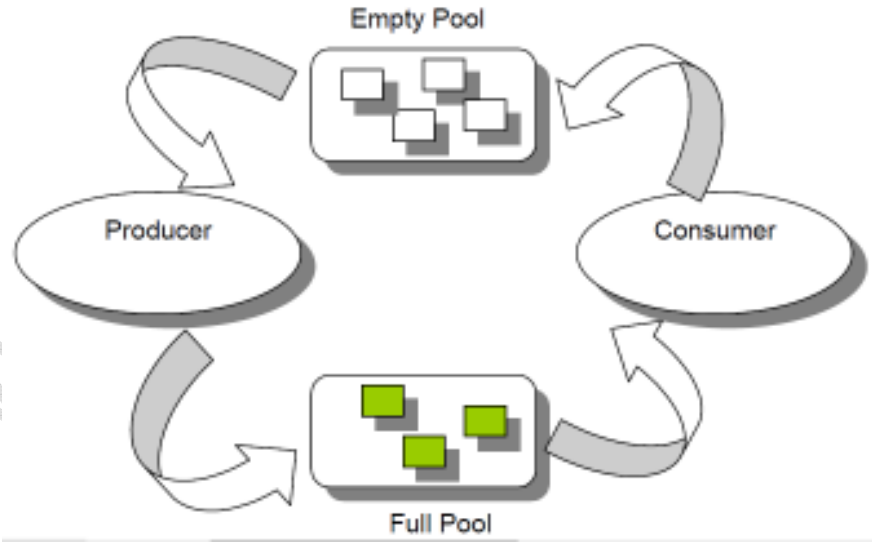


Figure 5.6 the Structure of Bounded Buffer Problem

The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.

The code for the producer is as follows:

```
do {
...
produce an item in nextp
...
wait(empty);
wait(mutex);
...
add nextp to buffer
...
signal(mutex);
signal(full);
} while(1);
```

And that for the consumer is as follows:

```
do {
wait(full);
wait(mutex);
...
remove an item from
buffer to nextc
...
signal(mutex);
signal(empty);
...
consume the item in nextc
...
} while(1);
```

Note the symmetry between the producer and the consumer process. This code can be interpreted as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

Readers Writers Problem

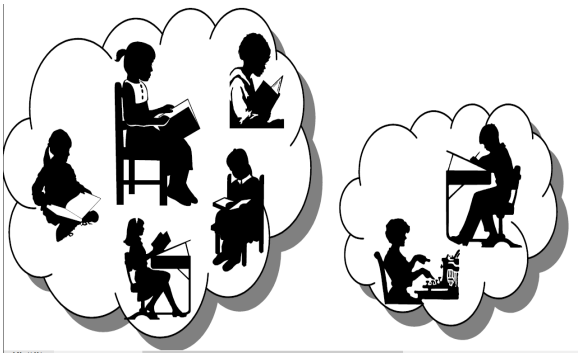


Figure 5.6 : Readers Writers Problem

A data object (such as a file or a record) is to be shared among several concurrent processes. Some of these processes, called **readers**, may want only to read the content of the shared object whereas others, called **writers**, may want to update (that is to read and write) the shared object. Obviously, if two readers access the data simultaneously, no adverse effects will result. However, if a writer and some other process (whether a writer or some readers) access the shared object simultaneously, chaos may ensue. To ensure these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to the readers writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers problem has several variations, all involving priorities. The simplest one, referred to as the **first readers-writers problem**, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The **second readers-writers problem** requires that once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we discuss a solution to the first readers writers problem. In the solution to the first readers-writers problem, processes share the following data structures.

```
semaphore mutex, wrt;
int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both the reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the reader processes update the readcount variable. The readcount variable keeps track of how many processes are currently reading the object. The wrt semaphore is used to ensure mutual exclusion for writers or a writer and readers. This semaphore is also used by the first and last readers to block entry of a writer into its critical section and to allow open access to the wrt semaphore, respectively. It is not used by readers who enter or exit, while at least one reader is in its critical sections.
The codes for reader and writer processes are shown below:

```
wait(mutex);
readcount++;
if(readcount == 1)
wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
```

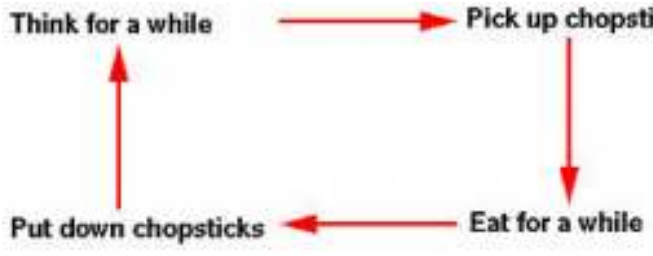
```
readcount--;
if(readcount == 0)
signal(wrt);
signal(mutex);
```

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n-1 readers are queued on mutex. Also observe that when a writer executes signal(wrt) we may resume the execution of either the waiting readers or a single waiting writer; the selection is made by the CPU scheduler.

Dining Philosophers Problem

Consider five philosophers who spend their lives thinking and eating, as shown in the following diagram.



The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks.

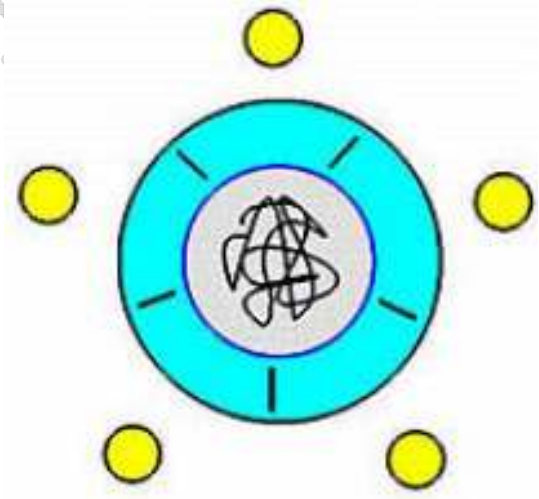


Figure 5.7: Dining Philosophers

When a philosopher thinks, she does not interact with her colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (the chopsticks that are between her and her left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of her neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing her chopsticks. When she is finished eating, she puts down both of her chopsticks and starts thinking again.

The dining philosophers problem is considered to be a classic synchronization problem because it is an example of a large class of concurrency control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tires to grab the chopstick by executing a wait operation on that semaphore; she releases her chopsticks by executing the signal operation on the appropriate semaphores. Thus the shared data are:

```
semaphore chopstick[5];
```

All the chopsticks are initialized to 1. The structure of philosopher i is as follows:

```
do {
wait(chopstick[i];
wait(chopstick[(i+1)%5]);
...
eat
...
signal(chopstick[i]);
signal(chopstick[(i+1)%5]);
...
think
...
}
```

Although this solution guarantees that no two neighbors are eating simultaneously, it nevertheless must be rejected because it has the possibility of creating a deadlock.

Suppose that all five gets hungry at the same time and pick up their left chopsticks as shown in the following figure. In this case, all chopsticks are locked and none of the philosophers can successfully lock her right chopstick. As a result, we have a circular waiting (i.e., every philosopher waits for his right chopstick that is currently being locked by his right neighbor), and hence a deadlock occurs.

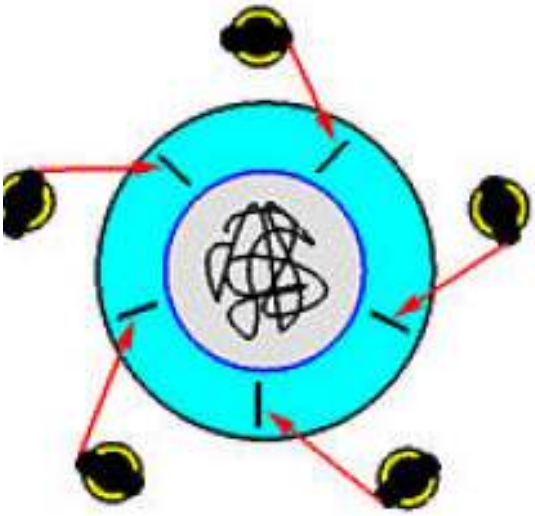


Figure 5.8: Dining Philosophers

Several possibilities that remedy the deadlock situation discussed in the last lecture are listed. Each results in a good solution for the problem.

- Allow at most four philosophers to be sitting simultaneously at the table.
- Allow a philosopher to pick up her chopsticks only if both chopsticks are available (to do this she must pick them up in a critical section)
- Use an asymmetric solution; that is, an odd philosopher picks up first her left chopstick, whereas an even philosopher picks up her right chopstick and then her left chopstick.

Removing the possibility of deadlock does not ensure that starvation does not occur. Imagine that two philosophers are fast thinkers and fast eaters. They think fast and get hungry fast. Then, they sit down in opposite chairs as shown below. Because they are so fast, it is possible that they can lock their chopsticks and eat. After finish eating and before their neighbors can lock the chopsticks and eat, they come back again and lock the chopsticks and eat. In this case, the other three philosophers, even though they have been sitting for a long time, they have no chance to eat. This is a starvation. Note that it is not a deadlock because there is no circular waiting, and everyone has a chance to eat!

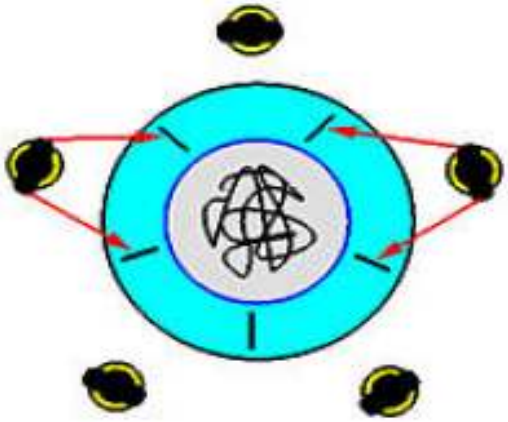


Figure 5.9: Dining Philosophers

High-level Synchronization Constructs

We discussed the problems of deadlock, starvation, and violation of mutual exclusion caused by the poor use of semaphores in lecture 23. We now discuss some high-level synchronization constructs that help solve some of these problems.

Critical regions

Although semaphores provide a convenient and effective mechanism for process synchronization, their incorrect usage can still result in timing errors that are difficult to detect, since these errors occur only if some particular execution takes place, and these sequences do not always happen.

To illustrate how, let us review the solution to the critical section problem using semaphores. All processes share a semaphore variable mutex, which is initialized to 1. Each process must execute wait(mutex) before entering the critical section and signal(mutex) afterward. If this sequence is not observed, two processes may be in their critical sections simultaneously.

To deal with the type of errors we outlined above and in lecture 23, a number of high level constructs have been introduced. In this section we describe one fundamental high level synchronization construct—the **critical region**. We assume that a process consists of some local data, and a sequential program that can operate on the data. Only the sequential program code that is encapsulated within the same process can access the local data. That is, one process cannot directly access the local data of another process.

Processes can however share global data. The critical region high-level synchronization construct requires that a variable v of type T, which is to be shared among many processes, be declared as:

```
v:shared T;
```

The variable v can be accessed only inside a region statement of the following form:

```
region v when B do S;
```

This construct means that, while statement S is being executed, no other process can access the variable v. The expression B is a Boolean expression that governs the access to the critical region. When a process tries to enter the critical-section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v. Thus if the two statements,

```
region v when(true) S1;
region v when(true) S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution "S1 followed by S2" or "S2 followed by S1". The critical region construct can be effectively used to solve several certain general synchronization problems. We now show use of the critical region construct to solve the bounded buffer problem. Here is the declaration of buffer:

```
struct buffer {
item pool[n];
int count,in,out;
};
```

The producer process inserts a new item (stored in nextp) into the shared buffer by executing

```
region buffer when(count < n) {
pool[in] = nextp;
in = (in+1)%n;
count++;
}
```

The consumer process removes an item from the shared buffer and puts it in next by Executing

```
region buffer when(count > 0) {
nextc = pool[out];
out = (out+1)%n;
count--;
}
```

Monitors

Another high-level synchronization construct is the monitor type. A **monitor** is characterized by local data and a set of programmer-defined operators that can be used to access this data; local data be accessed only through these operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. Normal scoping rules apply to parameters of a function and to its local variables. The syntax of the monitor is as follows:

```
monitor monitor_name
{
shared variable declarations
procedure body P1(..) { ...}
procedure body P1(..) { ...}
...
procedure body P1(..) { ...}
{
initialization code
}
}
```

The monitor construct ensures that only one process at a time can be active within the monitor. Consequently, the programmer does not need to code this synchronization construct explicitly. While one process is active within a monitor, other processes trying to access a monitor wait outside the monitor. The following diagram shows the big picture of a monitor.

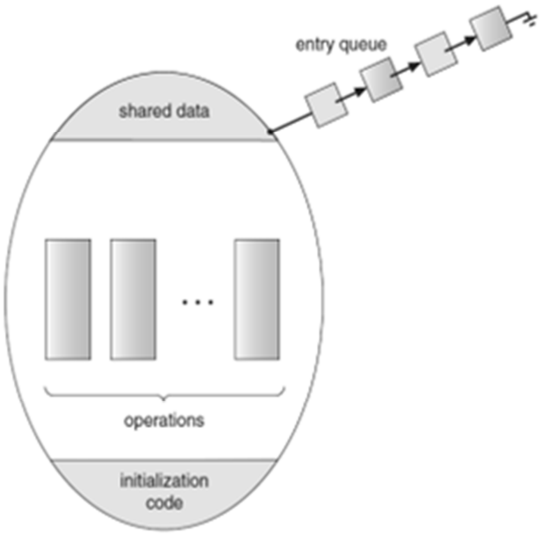


Figure 5.10 Schematic view of a monitor.

However, the monitor construct as defined so far is not powerful enough to model some synchronization schemes. For this purpose we need to define additional synchronization mechanisms. These mechanisms are provided by the **condition construct** (also called **condition variable**). A programmer who needs to write her own tailor made synchronization scheme can define one or more variables of type condition.

condition x,y;

The only operations that can be invoked on a condition variable are wait and signal. The operation

x.wait();

means that the process invoking this operation is suspended until another process invokes.

x.signal();

The x.signal() operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, the state of x is as though the operation were never executed. This is unlike the signal operation on a semaphore, where a signal operation always increments value of the semaphore by one. Monitors with condition variables can solve more synchronization problems that monitors alone. Still only one process can be active within a monitor but many processes may be waiting for a condition variable within a monitor, as shown in the following diagram.

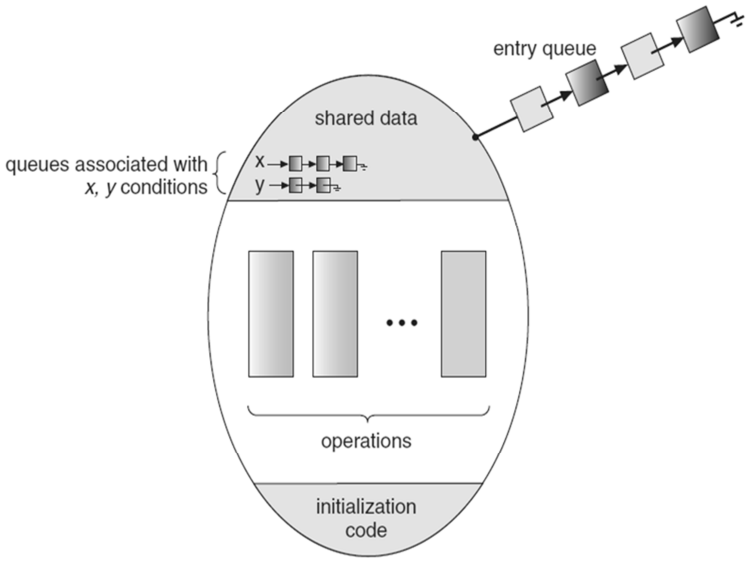


Figure 5.11: Monitor with condition variables.

There are reasonable arguments in favor of adopting either option. On the one hand, since *P* was already executing in the monitor, the ***signal-andcontinue*** method seems more reasonable. On the other, if we allow thread *P* to continue, then by the time *Q* is resumed, the logical condition for which *Q* was waiting may no longer hold. A compromise between these two choices was adopted in the language Concurrent Pascal. When thread *P* executes the signal operation, it immediately leaves the monitor. Hence, *Q* is immediately resumed.

Many programming languages have incorporated the idea of the monitor, including Java and C# (pronounced "C-sharp"). Other languages—such as Erlang—provide some type of concurrency support using a similar mechanism.

Chapter 06

Deadlocks

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes. This situation is called a **deadlock**.

An illustration of deadlock may be :

"When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone."

Here's an example:

- System has 2 tape drives.
- P1 and P2 each hold one tape drive and each needs another one.

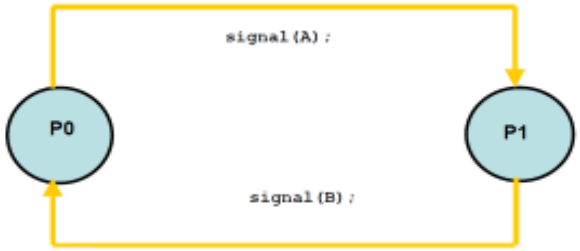


Figure 6.1: Deadlock Situation

Another deadlock situation can occur when the poor use of semaphores. Assume that two processes, P0 and P1, need to access two semaphores, A and B, before executing their critical sections. Semaphores are initialized to 1 each. The following code snippets show how a situation can arise where P0 holds semaphore A, P1 holds semaphore B, and both wait for the other semaphore—a typical deadlock situation as shown in the figure that follows the code.

P0	P1
wait (A);	wait(B);
wait (B);	wait(A);

In the first solution for the dining philosophers problem, if all philosophers become hungry at the same time, they will pick up the chopsticks on their right and wait for getting the chopsticks on their left. This causes a deadlock.

Yet another example of a deadlock situation can occur on a one-way bridge, as shown if figure 6.2. Traffic flows only in one direction, and 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. Starvation is possible.

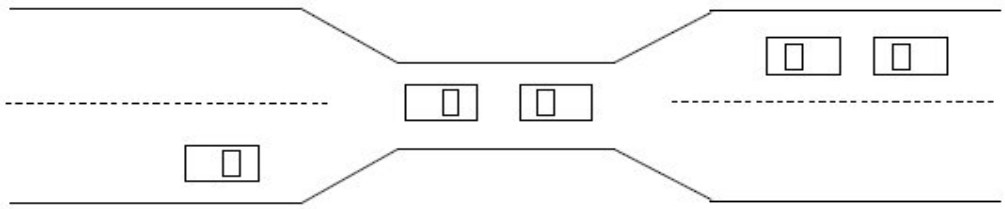


Figure 6.2: Traffic Block on a narrow bridge

6.1 System Model

A system consists of a finite number of resources to be distributed among a finite number of cooperating processes. The resources are partitioned into several types, each of which consists of some number of identical instances. Memory space, CPU cycles, disk drive, file are examples of resource types. A system with two identical tape drives is said to have two instances of the resource type disk drive.

If a process requests an instance of a resource type, the allocation of any instance of that type will satisfy the request. If it will not, then the instances are not identical and the resource type classes have not been defined properly.

A process must request a resource before using it, and must release the resource after using it. A process may request as many resources as it requires in order to carry out its designated task. Obviously, the number of resources requested may not exceed the total number of resources available in the system. Under the normal mode of operation, a process may utilize a resource in only the following sequence:

1. **Request:** The process requests a needed resource. If the request cannot be granted immediately, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can use the resource.
3. **Release:** The process releases the resource.

6.2 Deadlock Characterization

The following four conditions must hold simultaneously for a deadlock to occur:

1. **Mutual exclusion**: At least one resource must be held in a non-sharable mode; that is only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.
2. **Hold and wait**: A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. **No preemption**: Resources cannot be preempted. That is, after using it a process releases a resource only voluntarily.
4. **Circular wait**: A set $\{P_0, P_1 \ldots P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, and so on, P_{n-1} is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

6.3 Resource Allocation Graphs

Deadlocks can be described more precisely in terms of a directed graph called a system **resource allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices is portioned into two different types of nodes $P=\{P_0, P_1 \ldots P_n\}$, the set of the active processes in the system, and $R=\{R_0, R_1 \ldots R_n\}$, the set consisting of all resource types in the system. A directed edge from a process P_i to resource type R_j signifies that process P_i requested an instance of R_j and is waiting for that resource. A directed edge from R_j to P_i signifies that an instance of R_j has been allocated to P_i. We will use the following symbols in a resource allocation graph.

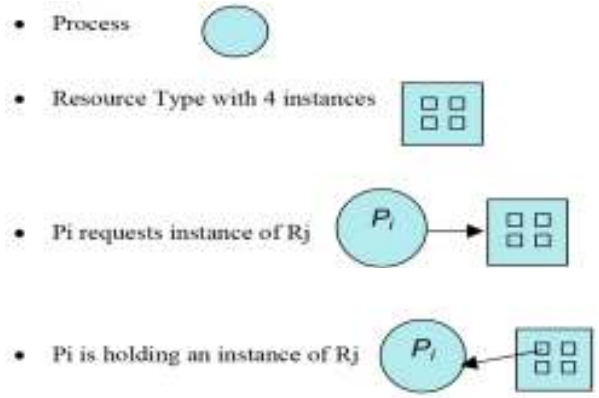


Figure 6.3: Symbols used in Resources Allocation Graph

The resource allocation graph shown below depicts the following situation:

- $P=\{P_1, P_2, P_3\}$
- $R=\{R_1, R_2, R_3\}$
- $E=\{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, P_3 \rightarrow R_3\}$

Resource Instances

- One instance of resource type R_1
- Two instances of resource type R_2
- One instance of resource type R_3
- Three instances of resource type R_4

Process States

- Process P_1 is holding an instance of resource R_2, and is waiting for an instance of resource R_1.
- Process P_2 is holding an instance of resource R_1 and R_2, and is waiting for an instance of resource R_3.
- Process P_3 is holding an instance of resource R_3.

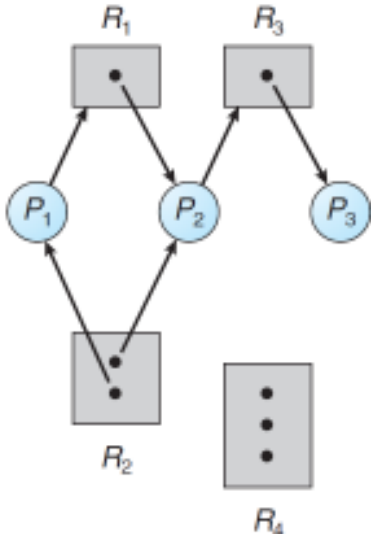


Figure 6.4 Resource-allocation graph.

Given the definition of a resource allocation graph, it can be shown that if the graph contains no cycles, then no process is deadlocked. If the graph contains cycles then:

- If only one instance per resource type, then a deadlock exists.
- If several instances per resource type, possibility of deadlock exists.

Here is a resource allocation graph with a deadlock. There are two cycles in this graph:
$\{P_1 \rightarrow R_1, R_1 \rightarrow P_2, P_2 \rightarrow R_3, R_3 \rightarrow P_3, P_3 \rightarrow R_2, R_2 \rightarrow P_1\}$ and
$\{P_2 \rightarrow R_3, R_3 \rightarrow P_3, P_3 \rightarrow R_2, R_2 \rightarrow P_2\}$

No process will release an already acquired resource and the three processes will remain in the deadlock state.

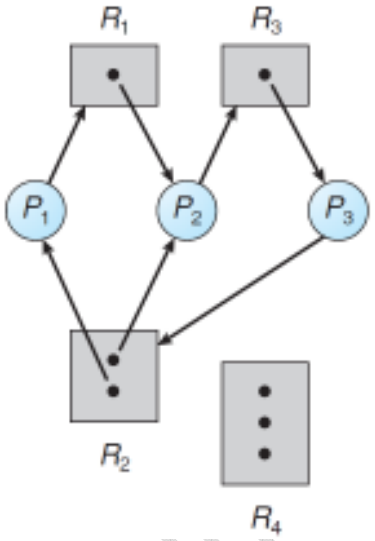


Figure 6.5 Resource-allocation graph with a deadlock.

The graph shown below has a cycle but there is no deadlock because processes P2 and P4 do not require further resources to complete their execution and will release the resources they are currently hold in finite time. These resources can then be allocated to P1 and P3 for them to resume their execution.

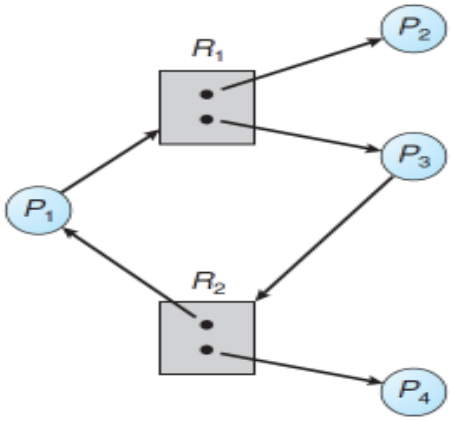


Figure 6.6: Resource-allocation graph with a cycle but no deadlock.

6.4 Deadlock Handling

We can deal with deadlocks in a number of ways:

- Ensure that the system will never enter a deadlock state.
- Allow the system to enter a deadlock state and then recover from deadlock.
- Ignore the problem and believe that deadlocks never occur in the system.

These three ways result in the following general methods of handling deadlocks:

1. **Deadlock prevention:**
   Deadlock Prevention a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how processes can request for resources.
2. **Deadlock Avoidance:**
   This method of handling deadlocks requires that processes give advance additional information concerning which resources they will request and use during their lifetimes. With this information, it may be decided whether a process should wait or not.
3. **Allowing Deadlocks and Recovering:**
   This method allows the system to enter a deadlocked state, detect it, and recover.

6.4.1 Deadlock Prevention

By ensuring that one of the four necessary conditions for a deadlock does not occur, we may prevent a deadlock.

1. **Mutual exclusion**

The mutual exclusion condition must hold for non-sharable resources, e.g., printer. Sharable resources do not require mutually exclusive access and thus cannot be involved in a deadlock, e.g., read-only files. Also, resources whose states can be saved and restored can be shared, such as a CPU. In general, we cannot prevent deadlocks by denying the mutual exclusion condition, as some resources are intrinsically non-sharable.

2. **Hold and Wait**

To ensure that the hold and wait condition does not occur in a system, we must guarantee that whenever a process requests a resource, it does not hold any other resources. One protocol that can be used requires each process to request and be allocated all its resources before it begins execution. We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls. An alternative protocol requires a process to request resources only when it has none. A process may request some resources and use them. But it must release these before requesting more resources.

Disadvantages: The two main disadvantages of these protocols are: firstly, resource utilization may be low, since many resources may be allocated but unused for a long time. Secondly, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

3. **No preemption**

To ensure that this condition does not hold we may use the protocol: if a process is holding some resources and requests another that cannot be allocated immediately to it, then all resources currently being held by the process are preempted. These resources are implicitly released, and added to the list of resources for which the process is waiting. The process will be restarted when it gets all its old, as well as the newly requested resources.

4. **Circular Wait**

One way to ensure that this condition never holds is to impose a total ordering of all resource types, and to require that each process requests resources in an increasing ordering of enumeration.

Let R={ R_1, R_2, R_3 }be resource types. We assign to each a unique integer, which allows us to compare two resources and to determine whether one precedes another in our ordering. For example, if the set of resource types R includes tape drivers, disk drives, and printers then the function F: R→N might be used to assign positive integers to these resources as follows:
F(tape drive) =1
F(disk drive) =5
F(printer)=12

Each process can request resources in an increasing order of enumeration. For example, a process wanting to use the tape and the disk drive must first request the tape drive and then the disk drive.

We can prove that if processes use this protocol then circular wait can never occur. We will prove this by contradiction. Let's assume that there is a cycle involving process P0 through Pk and that P_i is holding an instance of R_i, as shown below. The proof follows.

P0 → P1 → P2 → … → Pk → P0
R0 R1 R2 Rk R0
⇒ F(R0) < F(R1) < … < F(Rk) < F(R0)
⇒ F(R0) < F(R0), which is impossible.
⇒ There can be no circular wait.

6.4.2 Deadlock Avoidance

One method for avoiding deadlocks is to require additional information about how resources may be requested. Each request for resources by a process requires that the system consider the resources currently available, the resources currently allocated to the process, and the future requests and releases of each process, to decide whether the current request can be satisfied or must wait to avoid a possible future deadlock. The simplest and most useful model requires that each process declare the maximum number of resources of each type that it may need. Given a priori information about the maximum number of resources of each type that may be requested by each process, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state. A deadlock avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

6.4.2.1 Safe State

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally a system is in a safe state only if there exists a safe sequence. A sequence of processes <P_1, P_2… P_n> is a safe sequence for the current allocation state if, for each P_i, the resources that P_i can still request can be satisfied by the currently available resources plus all the resources held by all the P_j with j < i. In this situation, if the resources that P_i needs are not immediately available, then P_i can wait until all P_j have finished. When they have finished, P_i can obtain all of its needed resources, complete its designated task, return its allocated resources and terminate. When P_i terminates, P_{i+1} can obtain its needed resources and terminate. If no such sequence exists, then the system is said to be unsafe.

If a system is in a safe state, there can be no deadlocks. An unsafe state is not a deadlocked state; a deadlocked state is conversely an unsafe state. Not all unsafe states are deadlocks, however an unsafe state may lead to a deadlock state. Deadlock avoidance makes sure that a system never enters an unsafe state. The following diagram shows the relationship between safe, unsafe, and deadlock states.

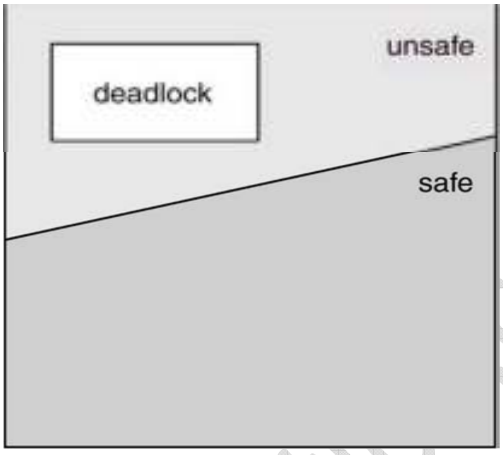


Figure 6.7: Safe, unsafe, and deadlocked state spaces.

Let' consider the following example to explain how a deadlock avoidance algorithm works. There is a system with 12 tape drives and three processes. The current system state is as shown in the following table. The available column shows that initially there are three tapes drives available and when process P1 finishes, the two tape drives allocated to it are returned, making the total number of tape drives 5. With 5 available tape drives, the maximum remaining future needs of P0 (of 5 tape drives) can be met. Once this happens, the 5 tape drives that P0 currently holds will go back to the available pool of drives, making the grand total of available tape drives 10. With 10 available drives, the maximum future need of P2 of 7 drives can be met. Therefore, system is currently in a safe state, with the safe sequence <P1, P0, P2>.

Process	Max Need	Allocated	Available
P_0	10	5	3
P_1	4	2	5
P_2	9	2	10

Now, consider that P2 requests and is allocated one more tape drive.
Assuming that the tape drive is allocated to P2, the new system state will be:

Process	Max Need	Allocated	Available
P_0	10	5	2
P_1	4	2	5
P_2	9	3	10

This new system is not safe. With two tape drives available, P1's maximum remaining future need can be satisfied which would increase the number of available tapes to 4. With 4 tapes available, neither P0's nor P2's maximum future needs can be satisfied. This means that if P2 request for an additional tape drive is satisfied, it would the system in an unsafe state. Thus, P2's request should be denied at this time.

6.4.2.2 Resource Allocation Graph Algorithm

In addition to the request and assignment edges, we introduce a new type of edge called a claim edge to resource allocation graphs. A claim edge Pi →Rj indicates that process Pi may request resource Rj at some time in the future. A dashed line is used to represent a claim edge. When Pi requests resource Rj the claim edge is converted to a request edge.

Suppose that Pi requests resource Rj. The request can be granted only if converting the request edge Pi →Rj into an assignment edge Rj →Pi does not result in the formation of a cycle. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state.

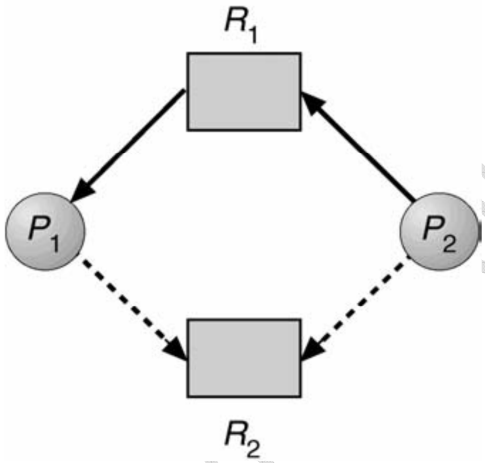
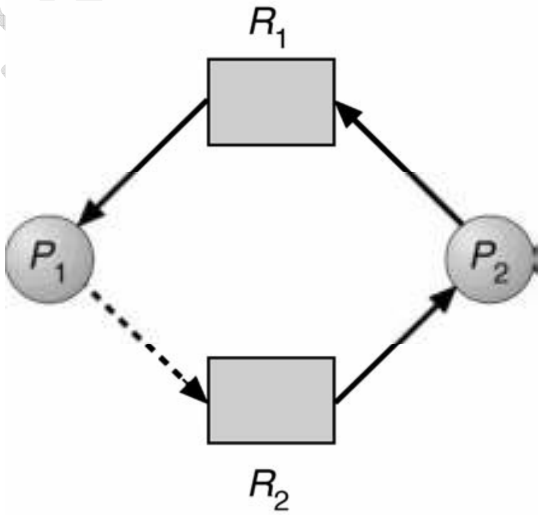


Figure 6.8 Resource-allocation graph for deadlock avoidance.

Suppose that P_i requests resource R_j. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ into an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle. If no cycle exists, then the allocation of the resource will leave the system in a safe state. If a cycle is found, then the allocation will put the system in an unsafe state. The following resource allocation graph shows that the system is in an unsafe state:



6.4.2.3 Banker's Algorithm

In this algorithm, when a new process enters the system, it must declare the maximum number of instances of each resource type that it may need, i.e., each process must a priori claim maximum use of various system resources. This number may not exceed the total number of instances of resources in the system, and there can be multiple instances of resources. When a process requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise the process must wait until some other process releases enough resources. We say that a system is in a safe state if all of the processes in the system can be executed to termination in some order; the order of process termination is called safe sequence.

When a process gets all its resources, it must use them and return them in a finite amount of time.

Let n be the number of processes in the system and m be the number of resource types. We need the following data structures in the Banker's algorithm:

- **Available:** A vector of length m indicates the number of available instances of resources of each type. Available[j] = = k means that there are k available instances of resource R_j.
- **Max:** An n x m matrix defines the maximum demand of resources of each process. Max[i,j] = = k means that process P_i may request at most k instances of resource R_j.
- **Allocation:** An n x m matrix defines the number of instances of resources of each type currently allocated to each process. Allocation[i,j] = = k means that P_i is currently allocated k instances of resource type R_j.
- **Need:** An n x m matrix indicates the remaining resource need of each process. Need[i,j] = = k means that P_i may need k more instances of resource type R_j to complete its task. Note that Need[i,j] = = Max[i,j] - Allocation[i,j].

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 1, 2, …, n.
2. Find an i such that both
   a) Finish[i] = = false
   b) $Need_i$ <= Work
   If no such i exists go to step 4.
3. Work = Work + $Allocation_i$
   Finish[i] = true
   Go to step 2
4. If Finish[i] = = true for all i, then the system is in a safe mode.

This algorithm may require an order of m x n^2 operations to decide whether a state is safe.

Resource Request Algorithm

Let $Request_i$ be the request vector for process P_i. if $Request_i[j]=k$, then process P_i wants k instances of resource R_j. When a request for resources is made by process P_i the following actions are taken:

1. If $Request_i <= Need_i$ go to step 2. Otherwise, raise an error condition since the process has exceeded its maximum claim.
2. If $Request_i <= Available$, go to step 3. Otherwise P_i must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process P_i by modifying the state as follows:

$Availabe = Available-Request_i$;

$Allocation_i = Allocation_i + Request_i$;

$Need_i = Need_i – Request_i$;

Invoke the Safety algorithm. If the resulting resource allocation graph is safe, the transaction is completed. Else, the old resource allocation state is restored and process P_i must wait for $Request_i$.

An illustrative example

We now show a few examples to illustrate how Banker's algorithm works. Consider a system with five processes P_0 through P_4 and three resource types: A, B, C. Resource type A has 10 instances, resource type B has 5 instances and resource type C has 7 instances. Suppose that at a time T_0 the following snapshot of the system has been taken:

	Allocation			**Max**			**Available**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	5	3	3	3	2
P_1	2	0	0	3	2	2			
P_2	3	0	2	9	0	2			
P_3	2	1	1	2	2	2			
P_4	0	0	2	4	3	3			

The content of the matrix Need is defined to be Max- Allocation and is:

	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

In the following sequence of tables, we show execution of the Safety algorithm for the given system state to determine if the system is in a safe state. We progressively construct a safe sequence.

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

We start our algorithm

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1>

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1, P_3>

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1	7	4	5
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1, P_3, P_4>

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1	7	4	5
P_4	0	0	2	4	3	1	7	5	5

Safe Sequence: < P_1, P_3, P_4, P_0>

The Safety algorithm concludes that the system is in a safe state, with < P_1, P_3, P_4, P_0, P_2 >being a safe sequence.

Suppose now that process P_1 requests one additional instance of resource type A and two instances of resource type C so Request $_1$ = (1, 0, 2). To decide whether this request can be fulfilled immediately, we invoke Banker's algorithm, which first check that Request$_1$ <= Available, which is true because (1,0,2)<=(3,3,2). It then pretends that this request has been fulfilled, and arrives at the following state:

	Allocation			**Need**			**Available**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Banker's algorithm then executes the Safety algorithm to determine if the resultant system will be in a safe state. Here is the complete working of Banker's algorithm. If P_1 requests (1,0,2), lets evaluate if this request may be granted immediately. Banker's algorithm takes the following steps.

1. Is Request1 ≤ Need1?

(1,0,2) ≤ (1,2,2) ⇒ true

2. Is Request1 ≤ Available?

(1,0,2) ≤ (3,3,2) ⇒ true

It then pretends that request is granted and updates the various data structures accordingly. It then invokes the Safety algorithm to determine if the resultant state is safe.

Here is sequence of steps taken by the Safety algorithm. The algorithm progressively constructs a safe sequence.

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0	5	3	2
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1 >

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1, P_3 >

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1	7	4	5
P_4	0	0	2	4	3	1			

Safe Sequence: < P_1, P_3 , P_4>

	Allocation			Need			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0	5	3	2
P_2	3	0	2	6	0	0	7	4	3
P_3	2	1	1	0	1	1	7	4	5
P_4	0	0	2	4	3	1	7	4	5

Safe Sequence: < P_1, P_3 , P_4, P_0>

Hence executing Safety algorithm shows that sequence <P1, P3, P4, P0, P2> satisfies the safety requirement and so P_1's request may be granted immediately. Note that safe sequence is not necessarily a unique sequence. There are several safe sequences for the above example. See lecture slides for more details.

Here is another example. P_0 requests (0,2,0). Should this request be granted? In order to answer this question, we again follow Banker's algorithm as shown in the following sequence of steps.

1. Is Request0 ≤ Need0?

(0,2,0) ≤ (7,4,3) ⇒ true

2. Is Request1 ≤ Available?

(0,2,0) ≤ (3,3,2) ⇒ true

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	3	2
P_1	2	0	0	1	2	2			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

The following is the updated system state. We run the Safety algorithm on this state now and show the steps of executing the algorithm.

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	1	2
P_1	2	0	0	1	2	2			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	1	2
P_1	2	0	0	1	2	2	5	2	3
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: $<P_3>$

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	1	2
P_1	2	0	0	1	2	2	5	2	3
P_2	3	0	2	6	0	0	7	2	3
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

Safe Sequence: $<P_3, P_1>$

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	1	2
P_1	2	0	0	1	2	2	5	2	3
P_2	3	0	2	6	0	0	7	2	3
P_3	2	1	1	0	1	1	10	2	5
P_4	0	0	2	4	3	1			

Safe Sequence: $<P_3, P_1, P_2>$

	Allocation			**Need**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	7	4	3	3	1	2
P_1	2	0	0	1	2	2	5	2	3
P_2	3	0	2	6	0	0	7	2	3
P_3	2	1	1	0	1	1	10	2	5
P_4	0	0	2	4	3	1	10	5	5

Safe Sequence: $<P_3, P_1, P_2, P_0, P_4>$

Hence executing the safety algorithm shows that sequence <P_3, P_1, P_2, P_0, P_4 > satisfies safety requirement. And so P_0's request may be granted immediately.

Suppose P_0 requests (0,2,0). Can this request be granted after granting P_1's request of (1,0,2)?

6.4.3 Deadlock Detection And Recovery:

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

- An algorithm that examines (perhaps periodically or after certain events) the state of the system to determine whether a deadlock has occurred
- A scheme to recover from deadlocks

Dead Lock Detection:

Single Instance of Each Resource Type:

If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource allocation graph, called a **wait-for graph**.

We obtain this graph from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. More precisely, an edge from P_i to P_j in a wait-for graph implies that process P_i is waiting for process P_j to release a resource that P_i needs. An edge $P_i \rightarrow P_j$ exists in a wait-for graph exists if and only if the corresponding resource allocation graph contains two edges for $P_i \rightarrow R_q$ and $R_q \rightarrow P_j$ some resource R_q.

As before, a deadlock exists in the system if and only if the wait for graph contains a cycle. To detect deadlocks the system needs to maintain the wait-for graph and periodically to invoke an algorithm that searches for a cycle in the graph. The following diagram shows a resource allocation graph and the corresponding wait-for graph. The system represented by the given wait-for graph has a deadlock because the graph contains a cycle.

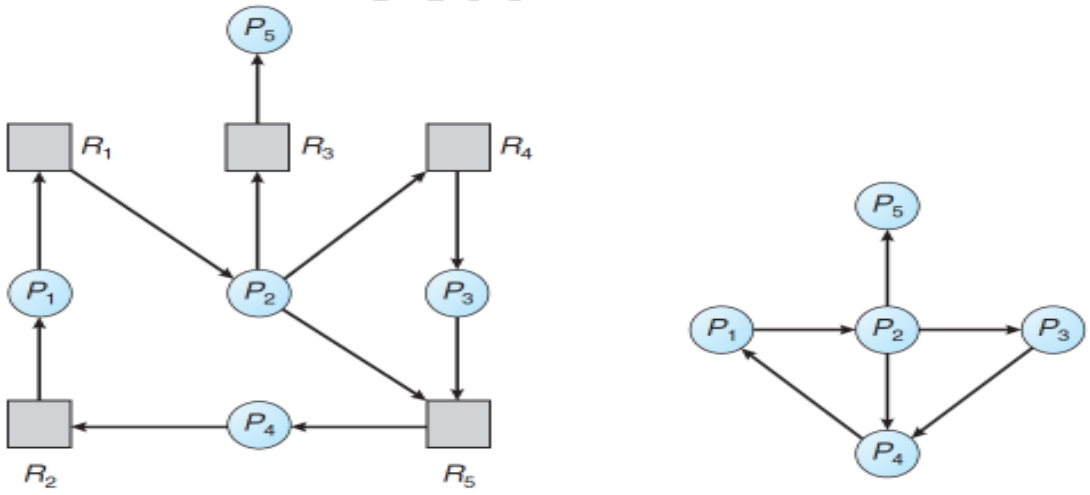


Figure 7.9 (a) Resource-allocation graph. (b)Corresponding wait-for graph.

Several Instances of a Resource Type

The wait for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. The deadlock detection algorithm described next is applicable to such a system. It uses the following data structures:

Available: A vector of length m indicates the number of available resources of each type.
Allocation: An n x m matrix defines the number of resources of each type currently allocated to each process.
Request: An n x m matrix indicates the current request of each process. If Request[i,j] = = k, then process P_i is requesting k more instances of resource type R_j.
The algorithm is:
1) Let Work and Finish be vectors of length m and n respectively. Initialize Work=Available. For i=1, 2,… , n if Allocation[i] ≠ 0 the Finish[i]=false; otherwise Finish[i]=true
2) Find an index i such that both
   a. Finish[i] = = false
   b. Request$_i$ ≤ Work
   c. If no such i exists go to step 4.
3) Work=Work + Allocation$_i$
   a. Finish[i]=true
   b. Go to step 2.
4) If Finish[i] = = false, for some i, 1≤ i ≤n, then the system is in a deadlock state. Moreover, if Finish[i] = = false, then P_i is deadlocked.
We show the working of this algorithm with an example. Consider the following system:
P = { P0, P1, P2, P3, P4 }
R = { A, B, C }
A: 7 instances
B: 2 instances
C: 6 instances

The system is currently in the following state. We want to know if the system has a deadlock. We find this out by running the above algorithm with the following state and construct a sequence in which requests for the processes may be granted.

	Allocation			**Request**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	2	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	Allocation			**Request**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	0	0	2	0	1	0
P_2	3	0	2	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Finish Sequence: < P_0>

	Allocation			**Request**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	0	0	2	0	1	0
P_2	3	0	2	0	0	0	3	1	2
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Finish Sequence: < P_0, P_2>

	Allocation			**Request**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	0	0	2	0	1	0
P_2	3	0	2	0	0	0	3	1	2
P_3	2	1	1	1	0	0	5	2	3
P_4	0	0	2	0	0	2			

Finish Sequence: < P_0, P_2, P_3>

	Allocation			**Request**			**Work**		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	0	0	2	0	1	0
P_2	3	0	2	0	0	0	3	1	2
P_3	2	1	1	1	0	0	5	2	3
P_4	0	0	2	0	0	2	5	2	5

Here is the sequence in which requests of processes P_0 through P_4 may be satisfied:
< P_0, P_2, P_3, P_4, P_1>. This is not a unique sequence. A few other possible sequences are the following.
< P_0, P_2, P_3, P_1, P_4,>
< P_0, P_2, P_4, P_1, P_3>
< P_0, P_2, P_4, P_3, P_1>

Now let us assume that P_2 requests an additional instance of C. Do we have a finish sequence? The work below shows that if this request is granted, the system will enter a deadlock. P_0's request can be satisfied with currently available resources, but request for no other process can be satisfied after that. Thus, a deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Process	**Request**		
	A	B	C
P_0	0	0	0
P_1	0	0	2
P_2	0	0	0
P_3	1	0	0
P_4	0	0	2

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	2	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

	Allocation			Request			Work		
	A	B	C	A	B	C	A	B	C
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2	0	1	0
P_2	3	0	2	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Detection Algorithm Usage

When should we invoke the deadlock detection algorithm? The answer depends on two factors:

1. How often is a deadlock likely to occur?
2. How many processes will be affected by deadlock when it happens?

Hence the options are:

Every time a request for allocation cannot be granted immediately—expensive but process causing the deadlock is identified, along with processes involved in deadlock

- Periodically, or based on CPU utilization
- Arbitrarily—there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery from Deadlock

When a deadlock detection algorithm determines that a deadlock exists, several alternatives exist. One possibility is to inform the operator that a deadlock has occurred, and to let the operator deal with the deadlock manually. The other possibility is to let the system recover from the deadlock automatically. There are two options for breaking a deadlock. One solution is simply to abort one or more processes to break the circular wait. The second option is to preempt some resources from one or more of the deadlocked processes.

Process Termination

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

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

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

Aborting a process may not be so easy. If a process was in the midst of updating a file, terminating it will leave the system in an inconsistent state. If the partial termination method is used, then given a set of deadlocked processes, we must determine which process should be terminated in an attempt to break the deadlock. This determination is a policy decision similar to CPU scheduling problems. The question is basically an economic one; we should abort those processes the termination of which will incur the minimum cost.

Unfortunately, the term minimum cost is not a precise one. Many factors determine which process is chosen, including:

1. What the priority of the process is
2. How long the process has computed, and how much longer the process will compute before completing its designated task.
3. How many and what type of resources the process has used
4. How many resources the process needs in order to complete
5. How many processes will need to be terminated?
6. Whether the process is interactive or batch

Resource Preemption

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

1. **Selecting a victim:** Which resources and which processes are to be preempted?

As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlock process is holding, and the amount of time a deadlocked process has thus far consumed during its execution.

2. **Rollback:** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. However it is more effective to roll back the process only as far as necessary to break the deadlock. On the other hand, this method requires the system to keep more information about the state of all the running processes.

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

Chapter 07

Main Memory

Memory is central to the operation of a modern computer system. Memory consists of a large array of bytes, each with its own address. The CPU fetches instructions from memory according to the value of the program counter. These instructions may cause additional loading from and storing to specific memory addresses.

A typical instruction-execution cycle, for example, first fetches an instruction from memory. The instruction is then decoded and may cause operands to be fetched from memory. After the instruction has been executed on the operands, results may be stored back in memory. The memory unit sees only a stream of memory addresses; it does not know how they are generated (by the instruction counter, indexing, indirection, literal addresses, and so on) or what they are for (instructions or data). Accordingly, we can ignore ***how*** a program generates a memory address. We are interested only in the sequence of memory addresses generated by the running program.

7.1 Basic Memory Hardware:

The memory hierarchy includes:

- **Register:** Very small, extremely fast, extremely expensive, and volatile CPU registers
- **Cache:** Small, very fast, expensive, and volatile cache
- **Main Memory :** Hundreds of megabytes of medium-speed, medium-price, volatile main memory
- **Secondary Storage:** Hundreds of gigabytes of slow, cheap, and non-volatile secondary storage
- **Internet Storage:** Hundreds and thousands of terabytes of very slow, almost free, and non-volatile Internet storage (Web pages, Ftp repositories, etc.)

7.2 Address Binding:

Usually a program resides on a disk as a binary executable or script file. The program must be brought into the memory it to be executed. The collection of processes that is waiting on the disk to be brought into the memory for execution forms the **input queue.** The normal procedure is to select one of the processes in the input queue and to load that process into the memory. As the process is executed, it accesses instructions and data from memory. Eventually the process terminates and its memory space is become available for reuse.

In most cases, a user program will go through several steps–some of which may be optional–before being executed. These steps are shown in the figure 7.1. Addresses may

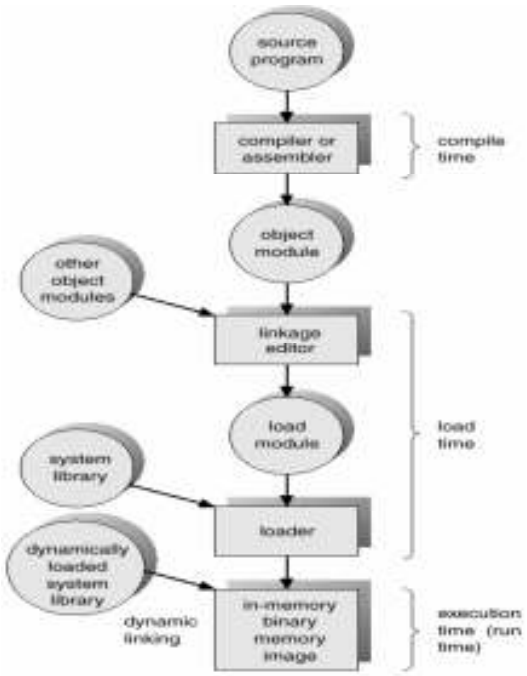


Figure 7.1 Multistep processing of a binding

be bound in different ways during these steps. Addresses in the source program are generally symbolic (such as an integer variable *count*). Address can be bound to instructions and data at different times, as discussed below briefly.

Compile time: if you know at compile where the process will reside in memory, the absolute addresses can be assigned to instructions and data by the compiler.

Load time: if it is not known at compile time where the process will reside in memory, then the compiler must generate Re-locatable code. In this case the final User program. is delayed until load time.

Execution time: if the process can be moved during its execution from one memory segment to another, then binding must be delayed until run time. Special hardware must be available for this to work.

In case of compile and load time binding, a program may not be moved around in memory at run time.

7.3. Logical- Versus Physical-Address Space

An address generated by the CPU is commonly referred to as a **logical address,** where as 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 the **physical address.** In essence, logical data refers to an instruction or data in the process address space where as the physical address refers to a main memory location where instruction or data resides. The compile time and load time binding methods generate identical logical and physical addresses, whereas the execution time binding method results in different physical and logical addresses. In this case we refer to the logical address as the **virtual address.** The set of all logical addresses generated by a program form the **logical address space** of a process; the set of all physical addresses corresponding to these logical addresses is a **physical address space** of the process. The total size of physical address space in a system is equal to the size of its main memory. The run-time mapping from virtual to physical addresses is done by a piece of hardware in the CPU, called the **memory management unit (MMU).**

Translation Examples

In the following two diagrams, we show two simple ways of translating logical addresses into physical address. In both case, there is a "base" register which is loaded with the address of the first byte in the program (instruction or data—in case of the second example, separate registers are used to point to the beginning of code, data, and stack portions of a program). In the first case, the base register is called the **relocation register**. The logical address is translated into the corresponding physical address by adding the logical address to the value of the relocation register, as shown in figure 7.2.

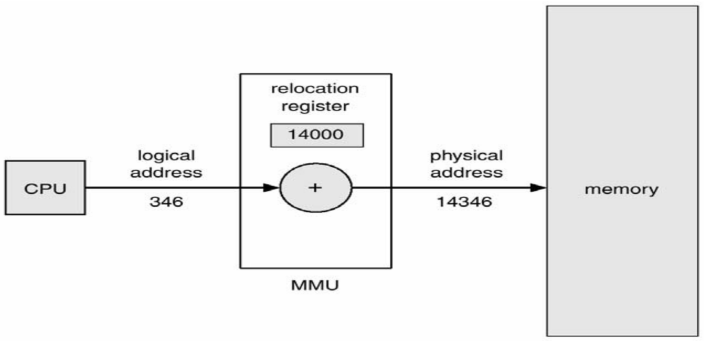
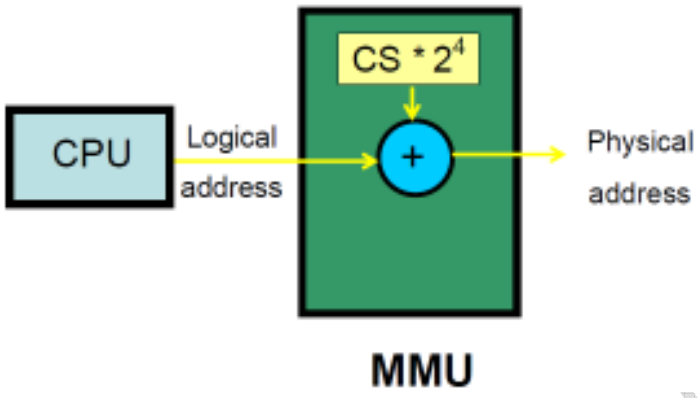


Figure 7.2: Logical to physical address translation

In i8086, the logical address of the next instruction is specified by the value of **instruction pointer** (IP). The physical address for the instruction is computed by shifting the **code segment register** (CS) left by four bits and adding IP to it,



In the following example, we show the logical address for a program instruction and computation of physical address for the given logical address.

- Logical address (16-bit)
  IP = 0B10h
  CS = D000h
- Physical address (20-bit)
  CS * 2^4 + IP = D0B10h

7.4. Various techniques for memory management

Here are some techniques of memory management, which are used in addition to the main techniques of memory management such as paging and segmentation.

7.4.1 Dynamic Loading

The size of a process is limited to the size of physical memory. To obtain better memory space utilization, we can use **dynamic loading**. With dynamic loading, a routine is not loaded until it is called. All routines are kept on a disk in a re-locatable format. The main program is loaded into memory and is executed. When a routine needs to call another routine, the calling routine first checks to see whether the other routine has been loaded or not. If not, the re-locatable linking loader is called to load the desired routine into the memory and to update the program's address tables to reflect this change. The control is then passed to the newly loaded routine.

The advantage of dynamic loading is that an unused routine is never loaded. This means that potentially less time is needed to load a program and less memory space is required. However the run time activity involved in dynamic loading is a disadvantage. Dynamic programming does not require special support from the operating system.

7.4.2 Dynamic Linking and Shared Libraries

Some operating systems support only **static linking** in which system language libraries are treated like any other object module and are combined by the loader into the binary proper image. The concept of dynamic linking is similar to that of dynamic loading. Rather than the loading being postponed until execution time, linking is postponed until run-time. This feature is usually used with system libraries. Without this facility, all programs on a system need to have a copy of their language library included in the executable image. This

requirement wastes both disk space and main memory. With dynamic linking, a stub is included in the image for each library-routine reference. This *stub* is a small piece of code that indicates how to locate the appropriate memory-resident library routine or how to load the library if the routine is not already present. During execution of a process, stub is replaced by the address of the relevant library code and the code is executed .If library code is not in memory, it is loaded at this time

This feature can be extended to update libraries. A library may be replaced by a new version and all programs that reference the library will automatically use the new version without any need to be re-linked. More than one version of a library may be loaded into the memory and each program uses its version information to decide which copy of the library to use. Only major changes increment the version number. Only programs that are compiled with the new library version are affected by the incompatible changes incorporated in it. Programs linked before the new library was installed will continue using the older library. This system is also known as **shared libraries**.

Dynamic linking requires potentially less time to load a program. Less disk space is needed to store binaries. However it is a time-consuming run-time activity, resulting in slower program execution. Dynamic linking requires help from the operating system. The **gcc** compiler invokes dynamic linking by default. The -static option allows static linking.

7.4.3 Overlays

To enable a process to be larger than the amount of memory allocated to it, we can use **overlays.** The idea of overlays is to keep in memory only those instructions and data that are needed at any given time. When other instructions are needed, they are loaded into space occupied previously by instructions that are no longer needed. We illustrate the concept of overlays with the example of a two-pass compiler. Here are the various specifications:

- 2-Pass assembler/compiler
- Available main memory: 150k
- Code size: 200k
  - Pass 1 …….. 70k
  - Pass 2 …….. 80k
  - Common routines ….. 30k
  - Symbol table …… 20k

Common routines, symbol table, overlay driver, and Pass 1 code are loaded into the main memory for the program execution to start. When Pass 1 has finished its work, Pass 2 code is loaded on top of the Pass 1 code (because this code is not needed anymore). This way, we can execute a 200K process in a 150K memory. The diagram below shows this pictorially.

The problems with overlays are that a) you may not be able to partition all problems into overlays, and b) programmer is responsible of writing the overlays driver.

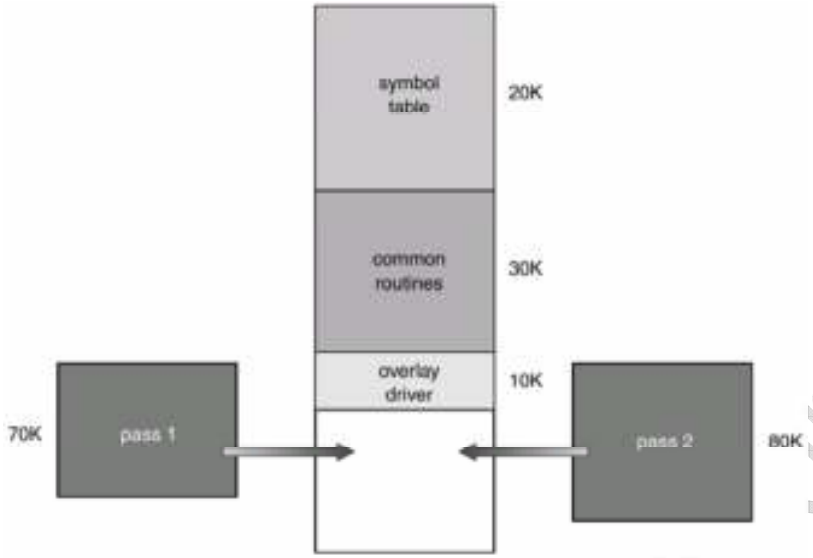


Figure 7.3: Overlays

7.4.4 Swapping

A process needs to be in the memory to be executed. A process, however, can be *swapped* temporarily out of memory to a *backing store*, and then brought back into memory for continued execution. Backing store is a fast disk large enough to accommodate copies of all memory images for all users; it must provide direct access to these memory images. The system maintains a *ready queue* of all processes whose memory images are on the backing store or in memory and are ready to run.

For example, assume a multiprogramming environment with a round robin CPU scheduling algorithm. When a quantum expires, the memory manager will start to swap out the process that just finished, and to swap in another process to the memory space that has been freed. A variant of this swapping policy can be used for priority-based scheduling algorithms. If a higher-priority process arrives and wants service, the memory manger can swap out the lower-priority process so that it can load and execute the higher-priority process. When the higher-priority process finishes, the lower-priority process can be swapped back in and continued. This technique is called **roll out, roll in**.

The major part of swap time is transfer time; the total transfer time is directly proportional to the *amount* of memory swapped. Swapping is constrained by factors like quantum for RR scheduler and pending I/O for swapped out process. Assume that I/O operation was queued because the device was busy. Then if we were to swap out P1, and swap in process P2, the I/O operation might attempt to access memory that now belongs to P2.The solution to this problem are never to swap out processes with pending I/O or to execute I/O in kernel space

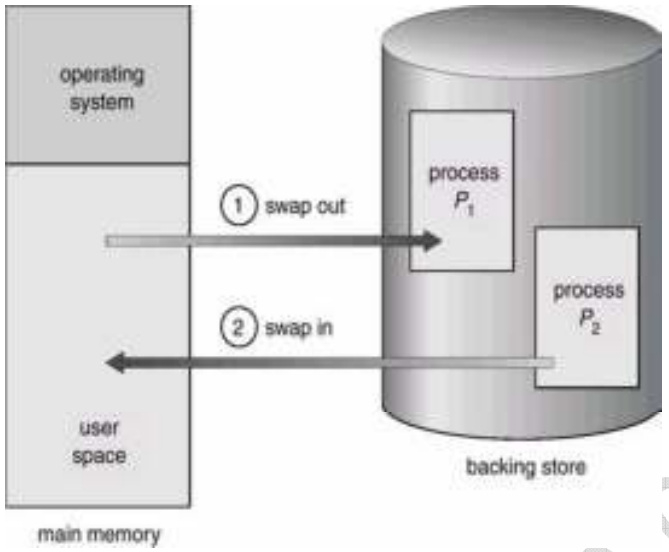


Figure 7.4: Schematic View of Swapping

Cost of Swapping

Process size	= 1 MB
Transfer rate	= 5 MB/sec
Swap out time	= 1/5 sec
	= 200 ms
Average latency	= 8 ms
Net swap out time	= 208 ms
Swap out + swap in	= 416 ms

Swapping on Mobile Systems

Although most operating systems for PCs and servers support some modified version of swapping, mobile systems typically do not support swapping in any form. Mobile devices generally use flash memory rather than more spacious hard disks as their persistent storage. The resulting space constraint is one reason why mobile operating-system designers avoid swapping. Other reasons include the limited number of writes that flash memory can tolerate before it becomes unreliable and the poor throughput between main memory and flash memory in these devices.

Instead of using swapping, when free memory falls below a certain threshold, Apple's iOS ***asks*** applications to voluntarily relinquish allocated memory. Read-only data (such as code) are removed from the system and later reloaded from flash memory if necessary. Data that have been modified (such as the stack) are never removed. However, any applications that fail to free up sufficient memory may be terminated by the operating system.

Android does not support swapping and adopts a strategy similar to that used by iOS. It may terminate a process if insufficient free memory is available. However, before terminating a process, Android writes its **application state** to flash memory so that it can be quickly restarted. Because of these restrictions, developers for mobile systems must carefully allocate and release memory to ensure that their applications do not use too much memory or

suffer from memory leaks. Note that both iOS and Android support paging, so they do have memory-management abilities.

7.5. Contiguous Memory Allocation

The main memory must accommodate both the operating system and the various user processes. We therefore need to allocate main memory in the most efficient way possible. This section explains one early method, contiguous memory allocation.

The memory is usually divided into two partitions: one for the resident operating system and one for the user processes. We can place the operating system in either low memory or high memory. The major factor affecting this decision is the location of the interrupt vector. Since the interrupt vector is often in low memory, programmers usually place the operating system in low memory as well.

We usually want several user processes to reside in memory at the same time. We therefore need to consider how to allocate available memory to the processes that are in the input queue waiting to be brought into memory. In **contiguous memory allocation**, each process is contained in a single section of memory that is contiguous to the section containing the next process.

7.5.1 Memory Protection

Before discussing memory allocation further, we must discuss the issue of memory protection. We can prevent a process from accessing memory it does not own by combining two ideas previously discussed. If we have a system with a relocation register, together with a limit register , we accomplish our goal. The relocation register contains the value of the smallest physical address; the limit register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). Each logical address must fall within the range specified by the limit register. The MMU maps the logical address dynamically by adding the value in the relocation register. This mapped address is sent to memory (Figure 7.5).

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process. The relocation-register scheme provides an effective way to allow the operating system's size to change dynamically. This flexibility is desirable in many situations. For example, the operating system contains code and buffer space for device drivers. If a device driver (or other operating-system service) is not commonly used, we do not want to keep the code and data in memory, as we might be able to use that space for other purposes. Such code is sometimes called **transient** operating-system code; it comes and goes as needed. Thus, using this code changes the size of the operating system during program execution.

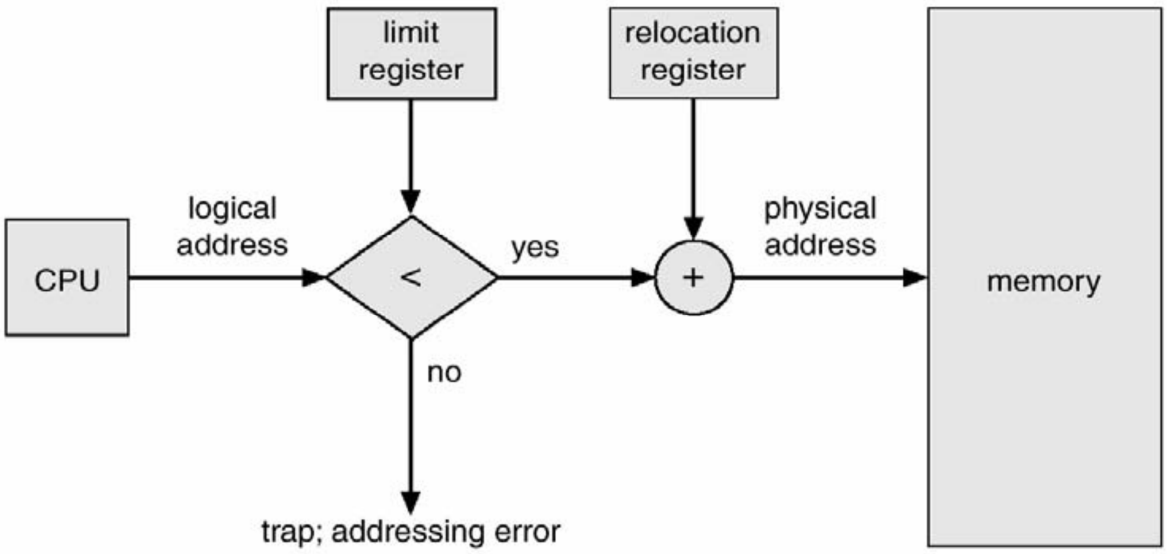


Figure 7.5: Hardware support for relocation and limit registers.

7.5.2 Memory Allocation

7.5.2.1 Multiprogramming with Fixed Tasks (MFT)

In this technique, memory is divided into several fixed-size partitions. Each partition may contain exactly one process. Thus the degree of multiprogramming is bound by the number of partitions. In this multiple partition method, when a partition is free, a process is selected from the input queue and is loaded in the free partition. When the process terminates, the partition becomes available for another process.

- This was used by IBM for system 360 OS/MFT (multiprogramming with a fixed number of tasks).
- Can have a single input queue instead of one for each partition.
- So that if there are no big jobs can use big partition for little jobs.
- Can think of the input queue(s) as the ready list(s) with a scheduling policy of FCFS in each partition.
- The partition boundaries are *not* movable and are set at boot time (must reboot to move a job).
- MFT can have large **internal fragmentation**, i.e., wasted space *inside* a region
- Each process has a single ``segment''
- No sharing between processes.
- No dynamic address translation.
- At load time must ``establish addressability''.
- Must set a base register to the location at which the process was loaded (the bottom of the partition).
- The base register is part of the programmer visible register set.
- This is an example of address translation during load time.
- Also called **relocation**.
- Storage keys are adequate for protection (IBM method).

- Alternative protection method is base/limit registers.
- An advantage of base/limit is that it is easier to move a job.
- But MFT didn't move jobs so this disadvantage of storage keys is moot.

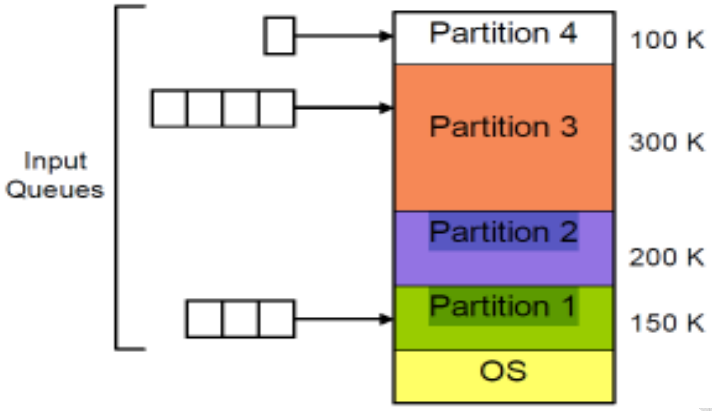


Figure 7.6: Multiprogramming with Fixed Tasks (MFT) with a queue per partition

MFT with multiple queues involves load-time address binding. In this technique, there is a potential for wasted memory space i.e. an empty partition but no process in the associated queue. However in MFT with single queue there is a single queue for each partition. The queue is searched for a process when a partition becomes empty. ***First-fit***, ***best-fit***, ***worst-fit*** space allocation algorithms can be applied here.

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

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

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

The following diagram shows MFT with single input queue.

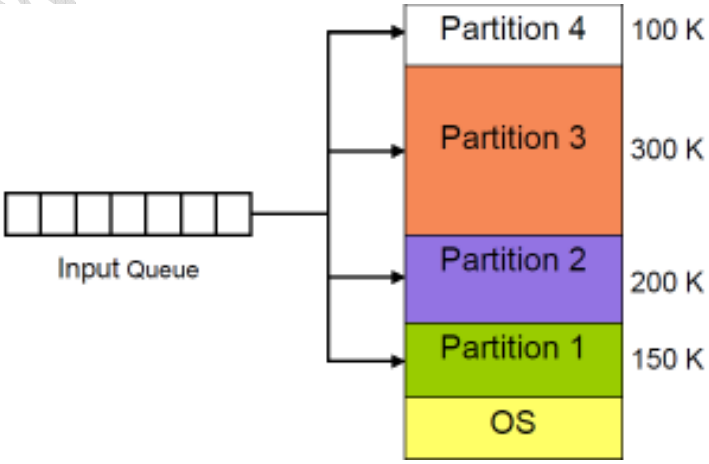


Figure 7.7: Multiprogramming with Fixed Tasks (MFT) with one input queue

7.5.2.2 Multiprogramming with Variable Tasks (MVT)

This is the generalization of the fixed partition scheme. It is used primarily in a batch environment. This scheme of memory management was first introduced in IBM OS/MVT (multiprogramming with a varying number of tasks). Here are the main characteristics of MVT.

- Both the number and size of the partitions change with time.
- Job still has only one segment (as with MFT) but now can be of any size up to the size of the machine and can change with time.
- A single ready list.
- Job can move (might be swapped back in a different place).
- This is dynamic address translation (during run time).
- Must perform an addition on every memory reference (i.e. on every address translation) to add the start address of the partition.
- **Eliminates internal fragmentation**.
    - Find a region the exact right size (leave a hole for the remainder).
    - Not quite true, can't get a piece with 10A755 bytes. Would get say 10A760.

  But internal fragmentation is *much* reduced compared to MFT. Indeed, we say that internal fragmentation has been eliminated.
- **Introduces external fragmentation**, i.e., holes *outside* any region.
- What do you do *if no hole is big enough* for the request?
- Can compact memory
    - Transition from bar 3 to bar 4 in diagram below.
    - This is expensive.
    - Not suitable for real time systems.
- Can swap out one process to bring in another
    - Bars 5-6 and 6-7 in the following diagram

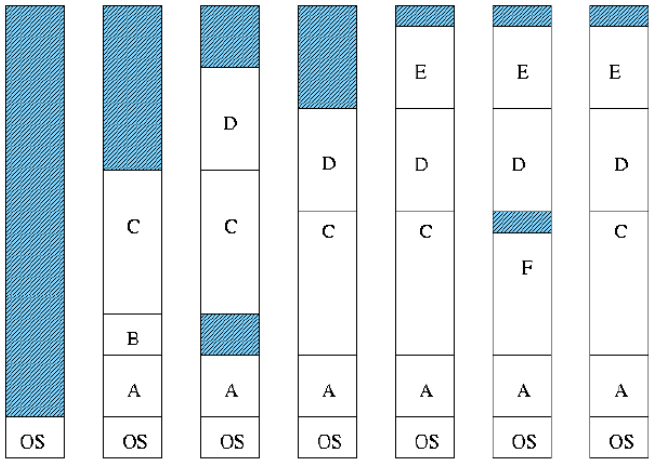


Figure 7.8: Multi-programming with Variable Tasks (MVT), external fragmentation, and compaction

7.5.3 External fragmentation

As processes come and go, *holes* of free space are created in the main memory. External Fragmentation refers to the situation when free memory space exists to load a process in the memory but the space is not contiguous. Compaction eliminates external fragmentation by shuffling memory contents (processes) to place all free memory into one large block. The cost of compaction is slower execution of processes as compaction takes place.

7.6. Paging:

In the memory management techniques discussed so far, Paging is a memory management scheme that permits the physical address space of a process to be noncontiguous. It avoids the considerable problem of fitting the various sized memory chunks onto the backing store, from which most of the previous memory-management schemes suffered. When some code fragments or data residing in main memory need to be swapped out, space must be found on the backing store. The fragmentation problems discussed in connection with main memory are also prevalent with backing store, except that access is much slower so compaction is impossible.

Physical memory is broken down into fixed-sized blocks, called **frames**, and logical memory is divided into blocks of the same size, called **pages**. The size of a page is a power of 2. It is important to keep track of all free frames. In order to run a program of size *n* pages, we find *n* free frames and load program pages into these frames. In order to keep track of a program's pages in the main memory a **page table** is used. The typical page table size lies between 1K and16K. Thus when a process is to be executed, its pages are loaded into any available memory frames from the backing store. Figure 7.9 shows process address space with pages (i.e., logical address space), physical address space with frames, loading of paging into frames, and storing mapping of pages into frames in a page table.

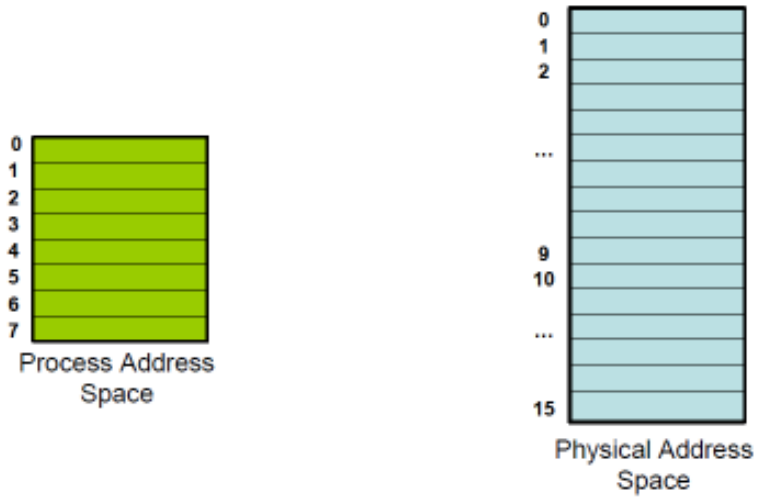


Figure 7.9 a): Logical and physical address spaces

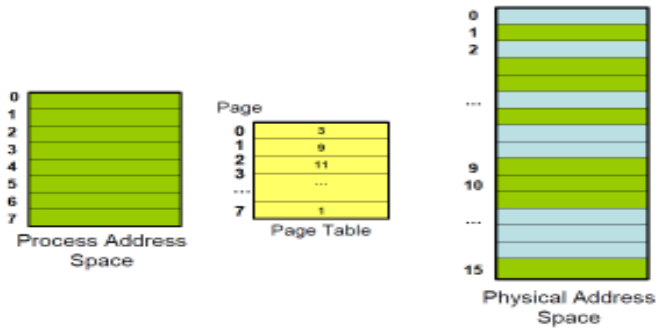


Figure 7.9 b): Mapping paging in the logical into the frames in the physical address space and keeping this mapping in the page table

Every **logical address** generated by the CPU is divided into two parts: a page number (p) and a page offset (d). The page table contains the base address (frame number) of each page in physical memory. The frame number is combined with the page offset to obtain the physical memory address of the memory location that contains the object addressed by the corresponding logical address. Here p is used to index the process page table; page table entry contains the frame number, f, where page p is loaded. The **physical address** of the location referenced by (p,d) is computed by appending d at the end of f, as shown below:

F	d

The hardware support needed for this address translation is shown below.

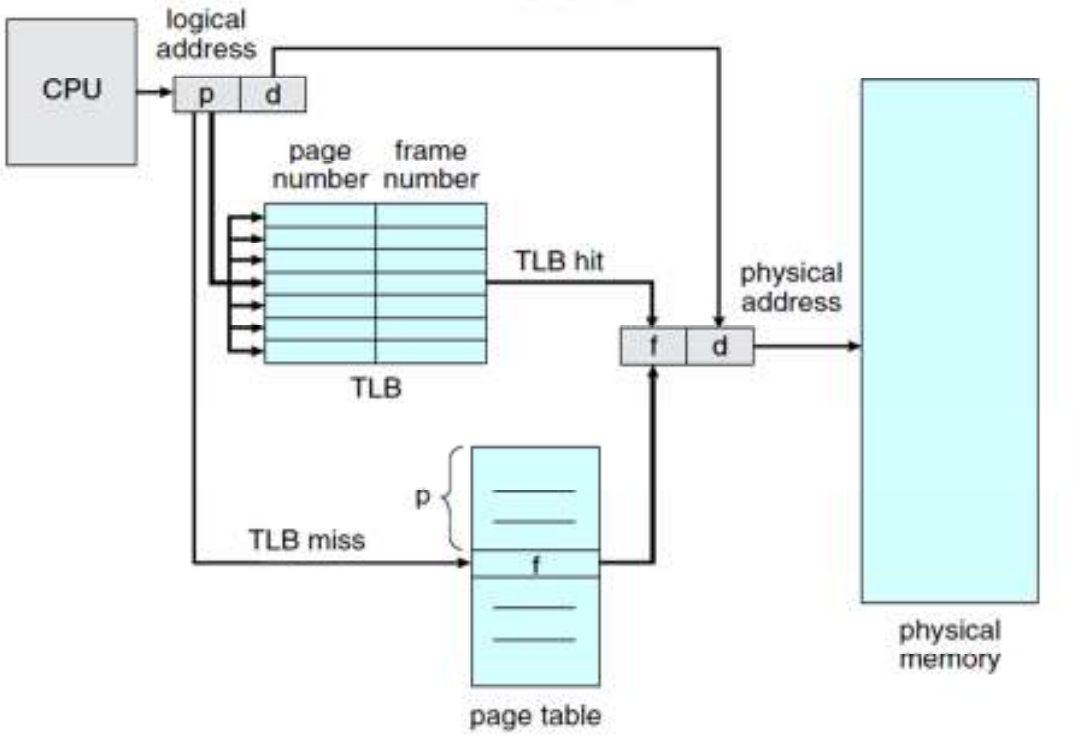


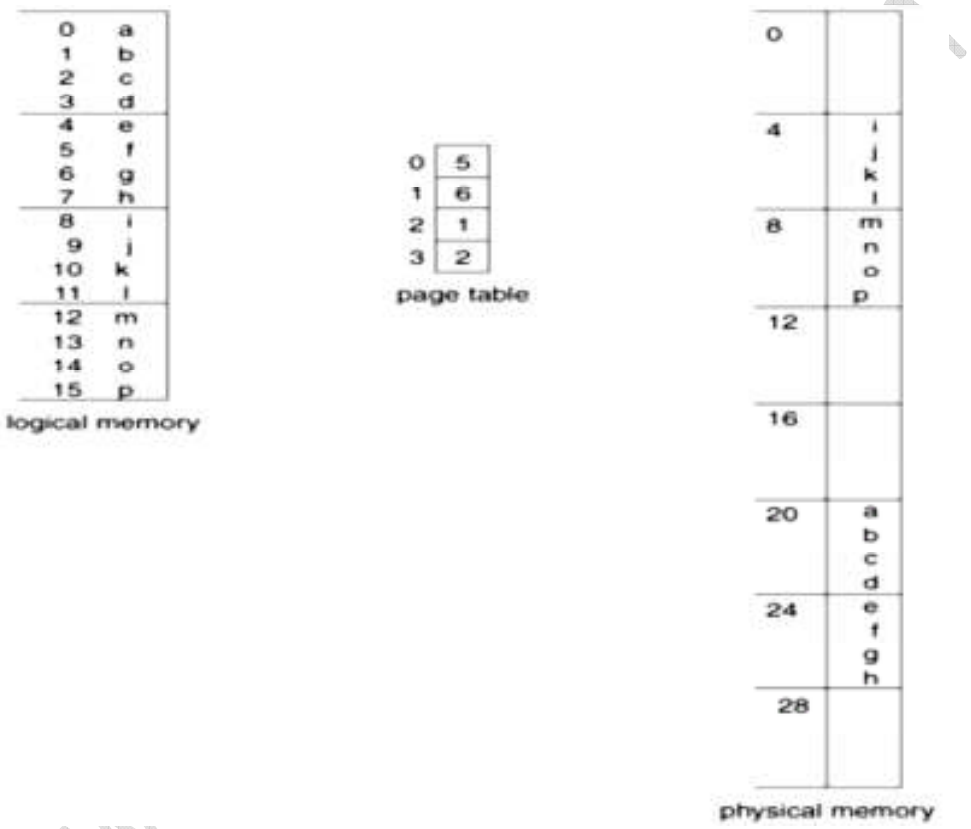
Figure 7.10: Paging hardware with TLB.

Paging itself is a form of dynamic relocation. When we use a paging scheme, we have no external fragmentation; however we may have **internal fragmentation**. An important aspect of paging is the clear separation between the user's view of memory and the

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

Paging Example

- Page size = 4 bytes
- Process address space = 4 pages
- Physical address space = 8 frames
- Logical address: (1,3) = 0111
- Physical address: (6,3) = 1011



7.6.1. Addressing in Paging

The page size is defined by the CPU hardware. If the size of logical address space is 2^m and a page size is 2^n addressing units (bytes or words) , then the high-order *m-n* bits of a logical address designate the page number and the *n* low order bits designate offset within the page. Thus, the logical address is as follows:

page number	page offset
P	D
m-n bits	n bits

Example:

Assume a **logical address space** of 16 pages of 1024 words, each mapped into a physical memory of 32 frames. Here is how you calculate the various parameters related to paging.

No. of bits needed for **p** = ceiling [log2 16] bits = 4 bits
No. of bits neede for **f** = ceiling [log2 32] bits = 5 bits
No. of bits needed for **d** = ceiling [log2 2048] bits = 11 bits
Logical address size = |p| + |d| = 4+11 bits = 15 bits
Physical address size = |f| + |d| = 5+11 bits = 16 bits

Page Table Size

Page table size = NP * PTES , where NP is the number of pages in the process address space and PTES is the page table entry size (equal to |f| based on our discussion so far).
Page table size = 16 * 5 bits (for the above example; assuming a byte size page table entry)

Paging in Intel P4

32-bit *linear* address
4K page size
Maximum pages in a process address space = 232 / 4K
Number of bits needed for **d** = $\log_2$ 4K bits = 12 bits
Number of bits needed for **p** = 32 – 12 bits =20

Paging in PDP-11

16-bit logical address
8K page size
Maximum pages in a process address space = 216 / 8K = 8
|**d**| = log2 8K = 13 bits
|**p**| = 16 – 13 = 3 bits

Another Example

Logical address = 32-bit
Process address space = 232 B = 4 GB
Main memory = RAM = 512 MB
Page size = 4K
Maximum pages in a process address space = 232 / 4K = 1M
|**d**| = $\log_2$ 4K = 12 bits
|**p**| = 32 – 12 = 20 bits
No. of frames = 512 M / 4 K = 128 K
|**f**| = ceiling [$\log_2$ 128 K] bits = 17 bits ≈ 4 bytes (rounding to next even-numbered byte)
Physical address = 17+12 bits

7.6.2. Implementation of Page table

- **In the CPU registers**

This is OK for small process address spaces and large page sizes. It has the advantage of having *effective memory access time* ($T_{effective}$) about the same as memory access time (T_{mem}). An example of this implementation is in PDP-11.

- **In the main memory**

A page table base register (PTBR) is needed to point to the page table. With page table in main memory, the effective memory access time, $T_{effective}$, is $2T_{mem}$, which is not acceptable because it would slow down program execution by a factor of two.

- **In the translation look-aside buffer (TLB)**

A solution to this problem is to use special, small, fast lookup hardware, called translation look-aside buffer (TLB), which typically has 64–1024 entries. Each entry is (key, value). The key is searched for in parallel; on a hit, value is returned. The (key,value) pair is (p,f) for paging. For a logical address, (p,d), TLB is searched for p. If an entry with a key p is found, we have a hit and f is used to form the physical address. Else, page table in the main memory is searched.

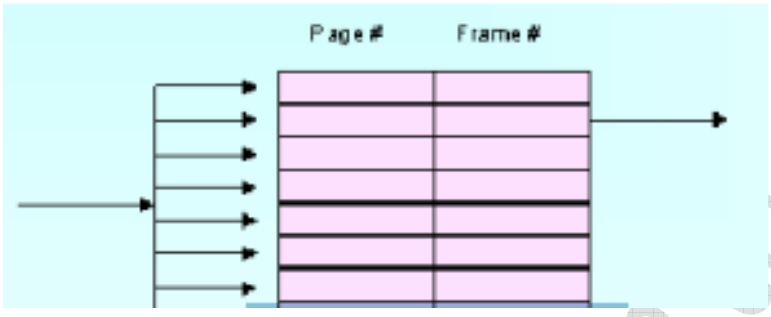


Figure 7.11: TLB –Logical address: (p,d)

The TLB is loaded with the (p,f) pair so that future references to p are found in the TLB, resulting in improved hit ratio. On a context switch, the TLB is flushed and is loaded with values for the scheduled process. Here is the hardware support needed for paging with part of the page table stored in TLB.

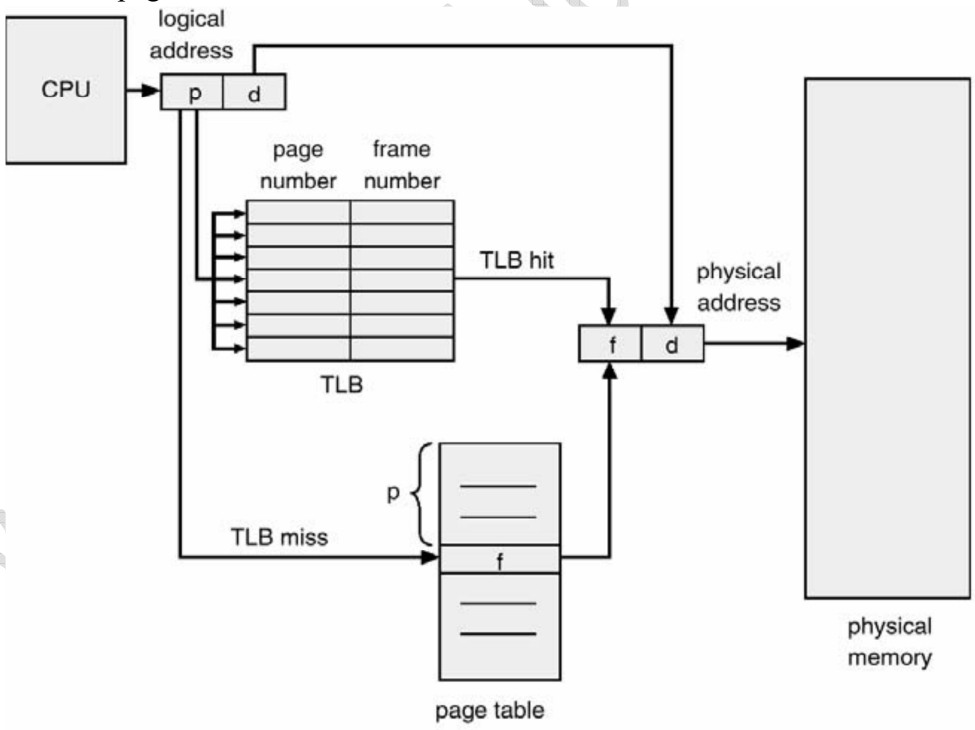


Figure 7.11: Paging Hardware with TLB

7.6.3 Performance of Paging

We discuss performance of paging in this section. The performance measure is the effective memory access time. With part of the page table in the TLB and the rest in the

main memory, the effective memory access time on a hit is Tmem + TTLB and on a miss is 2Tmem + TTLB.

If HR is hit ratio and MR is miss ratio, the effective access time is given by the following equation

$T_{effective}$ = HR (TTLB + Tmem) + MR (TTLB + 2Tmem)

We give a few examples to help you better understand this equation.

Example 1

Tmem = 100 nsec

TTLB = 20 nsec

Hit ratio is 80%

$T_{effective}$ = 0.8 (20 + 100) + 0.2 (20 + 2*100) nanoseconds = 140 nanoseconds

This means that with 80% chances of finding a page table entry in the TLB, the effective access time becomes 40% worse than memory access time without paging.

Example 2

Tmem = 100 nsec

TTLB = 20 nsec

Hit ratio is 98%

$T_{effective}$ = 0.98 (20 + 100) + 0.02 (20 + 2*100) nanoseconds = 122 nanoseconds

This means that with 98% chances of finding a page table entry in the TLB, the effective access time becomes 22% worse than memory access time without paging. This means that with a small cache and good hit ratio, we can maintain most of the page table in the main memory and get much better performance than keeping the page table in the main memory and not using any cache.

7.6.4 Protection under Paging

Memory protection in paging is achieved by associating protection bits with each page. These bits are associated with each page table entry and specify protection on the corresponding page. The primary protection scheme guards against a process trying to access a page that does not belong to its address space. This is achieved by using a valid/invalid (v) bit. This bit indicates whether the page is in the process address space or not. If the bit is set to invalid, it indicates that the page is not in the process's logical address space. Illegal addresses are trapped by using the valid-invalid bit and control is passed to the operating system for appropriate action. The following diagram shows the use of v bit in the page table. In this case, logical address space is six pages and any access to pages 6 and 7 will be trapped because the v bits for these pages are set to invalid.

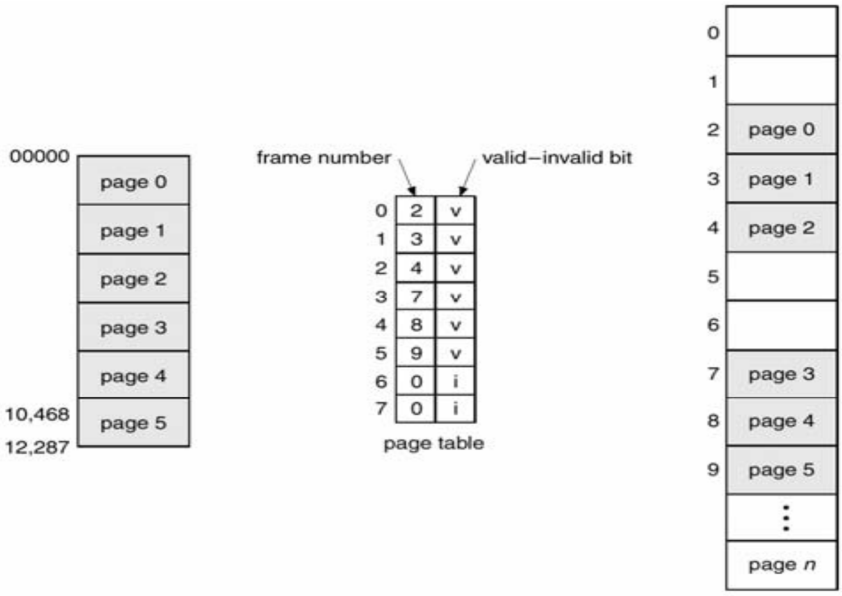


Figure 7.12: Use of valid/invalid (v) bit for protection under paging

One bit can define the page table to be read and write or read only. Every reference to memory goes through the page table to find the correct frame number. At the same time that the physical address is being computed, the protection bits can be checked to verify that no writes are being made to a read only page. An attempt to write to a read-only page causes a hardware trap to the operating system (memory-protection violation).

This approach can be expanded to provide a finer level of protection. Read, write, and execute bits (r, w, x) can be used to allow a combination of these accesses, similar to the file protection scheme used in the UNIX operating system. Illegal attempts will be trapped to the operating system.

7.6.5 Structure of the Page Table

As logical address spaces become large (32-bit or 64-bit), depending on the page size, page table sizes can become larger than a page and it becomes necessary to page the page the page table. Additionally, large amount of memory space is used for page table. The following schemes allow efficient implementations of page tables.

- **Hierarchical / Multilevel Paging**
- **Hashed Page Table**
- **Inverted Page Table**

Hierarchical/Multilevel Paging

Most modern computers support a large logical address space: (2^{32} to 2^{64}). In such an environment, the page table itself becomes excessively large. Consider the following example:

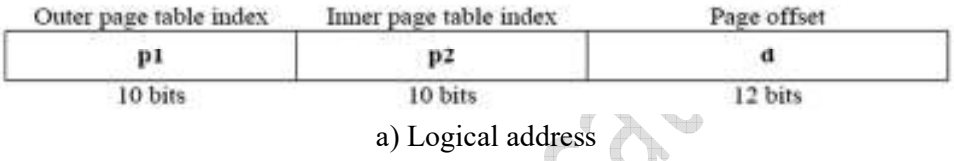
- Logical address = 32-bit
- Page size = 4K bytes (2^{12} bytes)
- Page table entry = 4 bytes
- Maximum pages in a process address space = 2^{32} / 4K = 1M
- Maximum pages in a process address space = 2^{32} / 4K = 1M

- Page table size = 4M bytes

This page table cannot fit in one page. One solution is to page the page table, resulting in a 2-level paging. A page table needed for keeping track of pages of the page table— called the outer page table or page directory. In the above example:

- No. of pages in the page table is 4M / 4K = 1K
- Size of the outer page table is 1K * 4 bytes = 4K bytes ⇒ outer page will fit in one page

In the 32-bit machine described above, we need to partition p into two parts, p1 and p2. p1 is used to index the outer page table and p2 to index the inner page table. Thus the logical address is divided into a page number consisting of 20 bits and a page offset of 12 bits. Since we page the page table, the page number is further divided into a 10-bit page number, and a 10-bit page offset. This is known as **two-level paging**. The following diagram shows division of the logical address in 2-level paging and hierarchical views of the page table.



a) Logical address

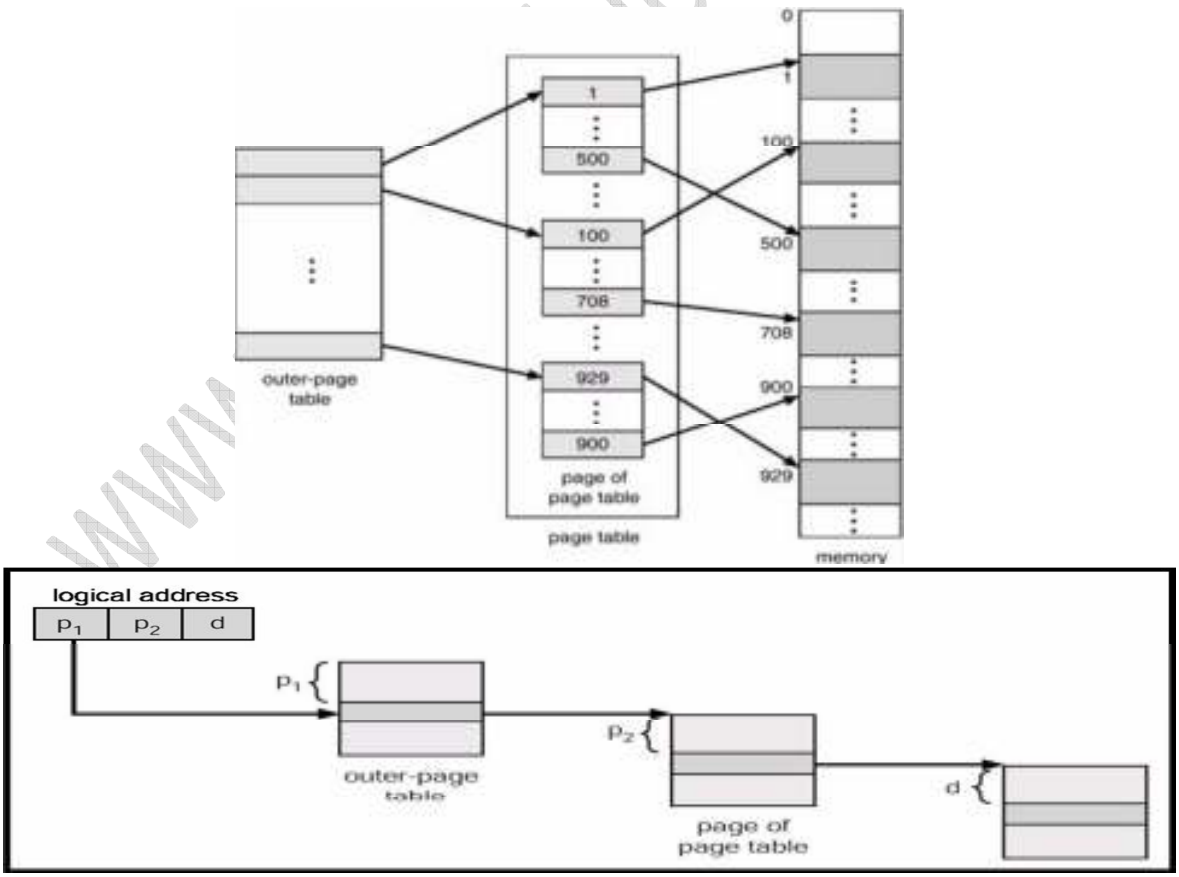


Figure 7.13: Two views of address translation for a two-level paging architecture

Another Example: DEC VAX

- Logical address = 32 bits
- Page size = 512 bytes = 2^9 bytes
- Process address space divided into four equal sections
- Pages per section = $2^{30} / 2^9 = 2^{21} = 2M$
- Size of a page table entry = 4 bytes
- Bits needed for page offset = $\log_2 512 = 9$ bits
- Bits to specify a section = $\log_2 4 = 2$ bits
- Bits needed to index page table for a section = $\log_2 2^{21} = 21$ bits
- Size of a page table = $2^{21} * 4 = 8$ MB
- MB page table is paged into 8MB / 512 = 2 K pages

Size of the outer page table (2K * 4 = 8 KB) is further paged, resulting in 3-level paging per section

Section	Page number	Page offset
s	p	d
2	21	9

More Examples

- 32-bit Sun SPARC supports 3-level paging
- 32-bit Motorola 68030 supports 4-level paging
- 64-bit Sun UltraSPARC supports 7-level paging – too many memory references needed for address translation

Hashed Page Table

This is a common approach to handle address spaces larger then 32 bits .Usually open hashing is used. Each entry in the linked list has three fields: page number, frame number for the page, and pointer to the next element—(p, f, next). The page number in the logical address (specified by p) is hashed to get index of an entry in the hash table. This index is used to search the linked list associated with this entry to locate the frame number corresponding to the given page number. The advantage of hashed page tables is smaller page tables.

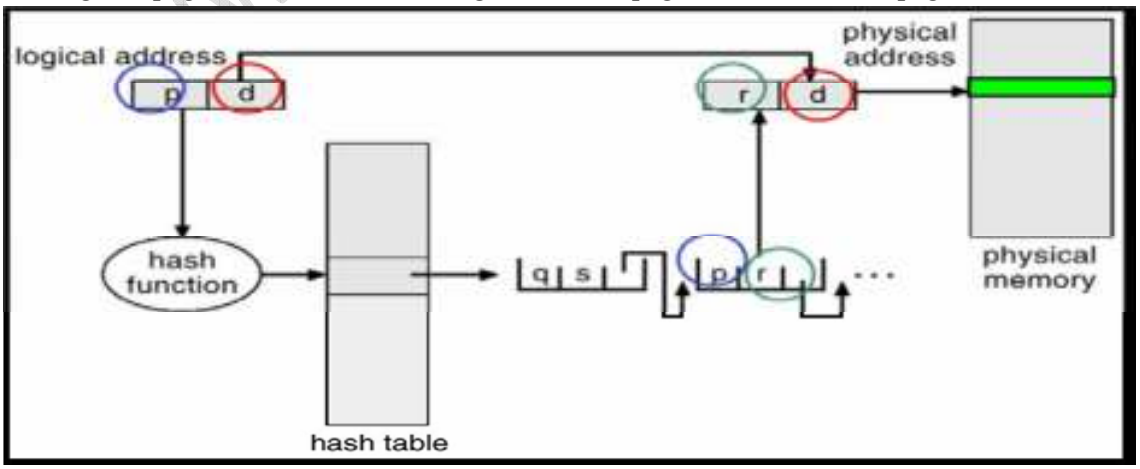


Figure 7.13: Hashed Page Table

Inverted Page Table

Usually each process has a page table associated with it. The page table has one entry for each page in the address space of the process. For large address spaces (32-bit and above), each page table may consist of millions of entries. These tables may consume large amounts of physical memory, which is required just to keep track of how the mapping of logical address spaces of processes onto the physical memory.

A solution is to use an inverted page table. An **inverted page table** has one entry for each real page (frame) of memory. Each entry consists of the virtual address of the page stored in the in that real memory location, with information about the process that own the page.

Page table size is limited by the number of frames (i.e., the physical memory) and not process address space. Each entry in the page table contains (pid, p). If a page 'p' for a process is loaded in frame 'f', its entry is stored at index 'f' in the page table. We effectively index the page table with frame number; hence the name inverted page table.

Examples of CPUs that support inverted pages tables are 64-bit UltraSPARC and PowerPC. The following diagram shows how logical addresses are translated into physical addresses.

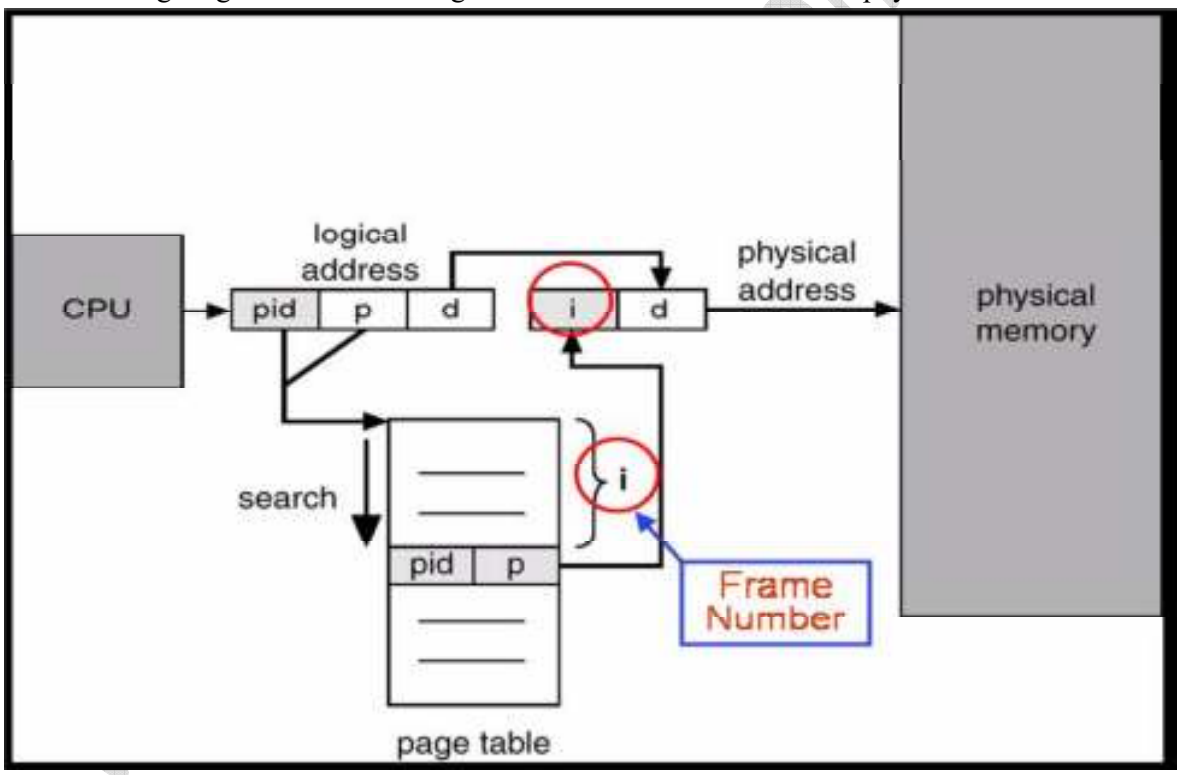


Figure 7.14: Address translation with inverted page table

7.6.6 Sharing in Paging

Another advantage of paging is the possibility of *sharing* common code. Reentrant (read only) code pages of a process address can be shared. If the code is reentrant, it never changes during execution. Thus two or more processes can execute the same code at the same time. Each process has its own copy of registers and data storage to hold the data for the process' execution. The data for two different processes will, of course, vary for each process. Consider the case when multiple instances of a text editor are invoked. Only one

copy of the editor needs to be kept in the physical memory. Each user's page table maps on to the same physical copy of the editor, but data pages are mapped onto different frames. Thus to support 40 users, we need only one copy of the editor, which results in saving total space.

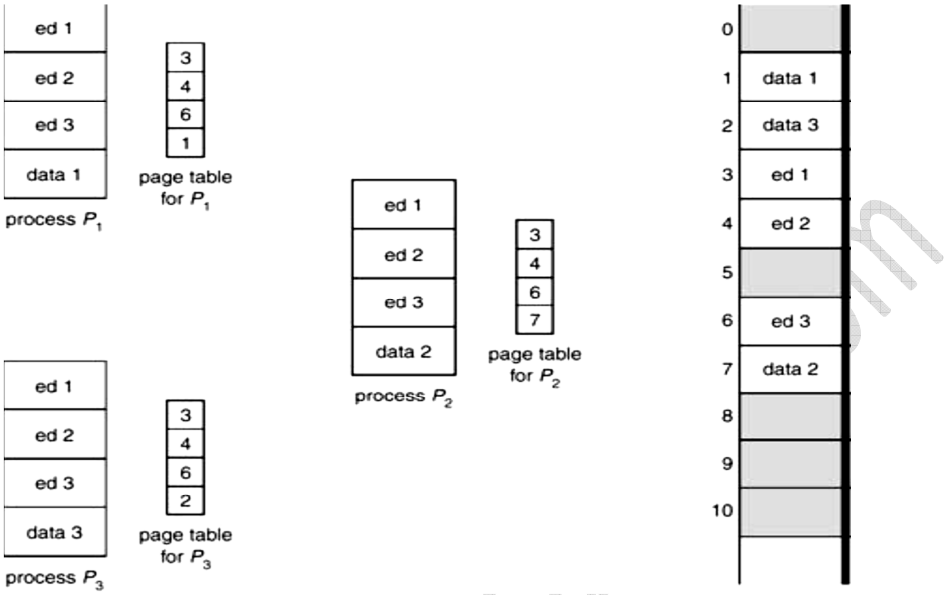


Figure 7.15: Sharing in paging

7.7. Segmentation

Segmentation is a memory management scheme that supports programmer's view of memory. A logical-address space is a collection of segments. A segment is a logical unit such as: main program, procedure, function, method, object, global variables, stack, and symbol table. Each segment has a name and length. The addresses specify both the segment name and the offset within the segment. An example of the logical address space of a process with segmentation is shown below.

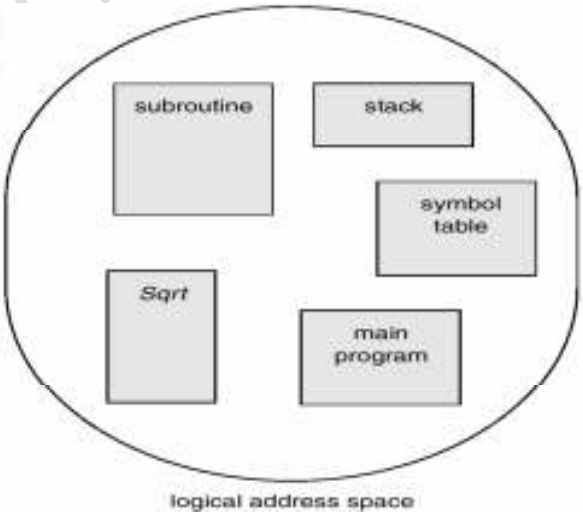


Figure 7.16: Logical address space with segmentation

For simplicity of implementation, segments are numbered and are referred to by a segment number, rather than by a segment name. Thus a logical address consists of a two tuple:

<segment-number, offset> or <s,d>

The segment table maps the two-dimensional logical addresses to physical addresses. Each entry of a segment table has a *base* and a *segment limit*. The segment base contains the starting physical address where the segment resides in memory, whereas the segment limit specifies the length of the segment.

There are two more registers, relevant to the concept of segmentation:

Segment-table base register (STBR) points to the segment table's location in memory.

Segment-table length register (STLR) indicates number of segments used by a program

Segment number s is legal if s < STLR, and offset, d, is legal if d < limit. The following diagram shows the hardware support needed for translating a logical address into the physical address when segmentation is used. This hardware is part of the MMU in a CPU.

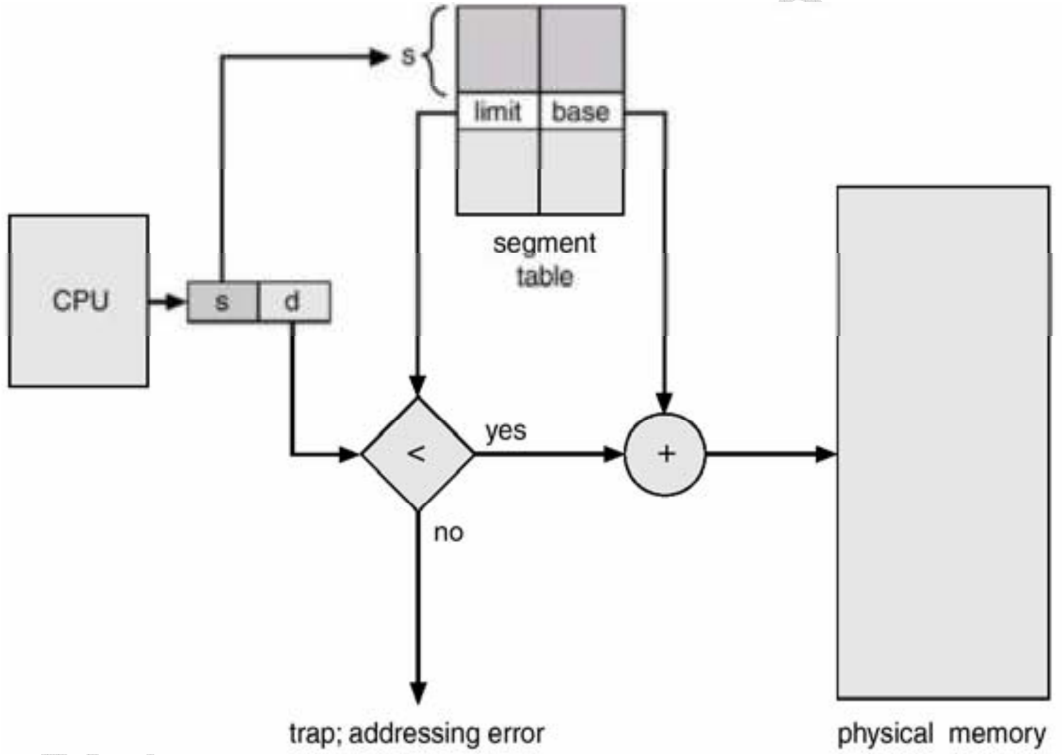


Figure 7.17: Hardware support for segmentation

For logical to physical address conversion, segment number, s, is used to index the segment table for the process. If d < limit, it is added to the base value to compute the physical address for the given logical address. The segment base and limit values are used to relocate and bound check the reference at runtime.

7.7.1 Sharing of Segments

Another advantage of segmentation is sharing of code or data. Each process has a segment table associated with it, which the dispatcher uses to define the hardware segment table when this process is given the CPU. Segments are shared when entries in the

segment tables of two different processes point to the same physical location. The sharing occurs at segment level, thus, any information defined as a segment can be shared.

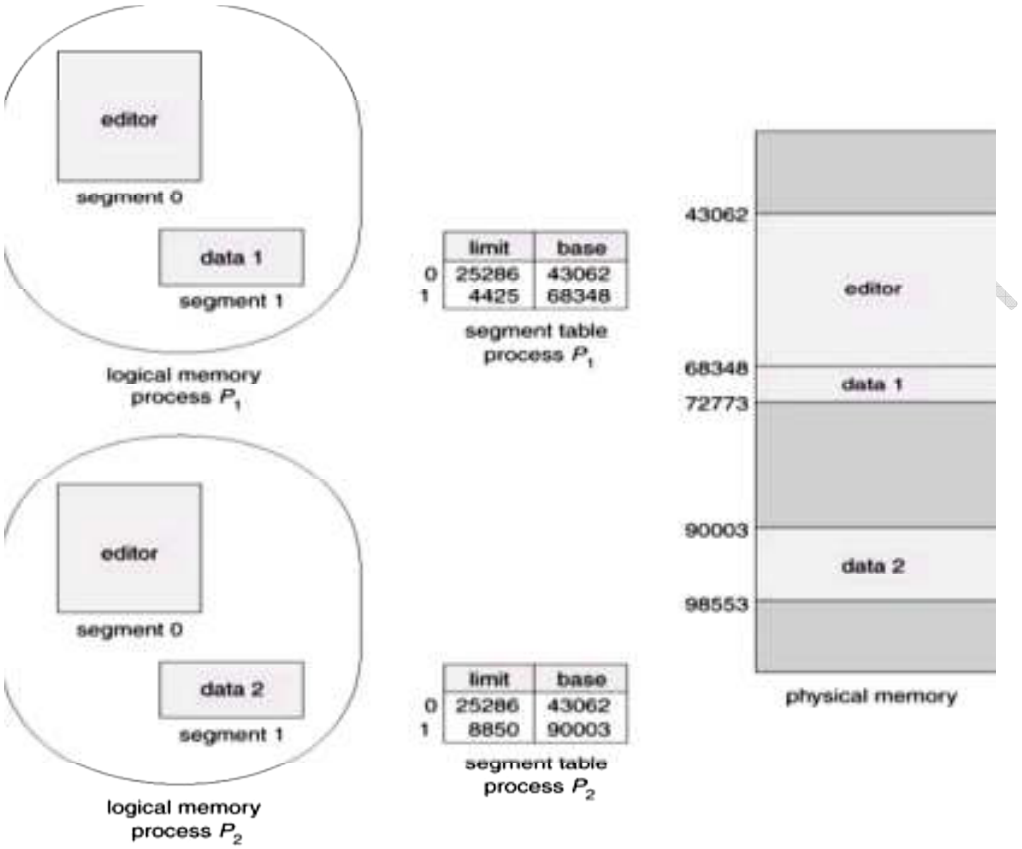


Figure 7.18: Sharing in segmentation

The long-term scheduler must find and allocate memory for all the segments of a user program. This situation is similar to paging except that the segments are of *variable* length; pages are all the same size. Thus memory allocation is a dynamic storage allocation problem, usually solved with a best fit or worst fit algorithm.

7.7.2 Protection

A particular advantage of segmentation is the association of protection with segments. Because the segments represent a defined portion of the program, it is likely that the entries will be used the same way. Hence, some segments are instructions, whereas other segments are data. In a modern architecture, instructions are non-self-modifying so they can be defined as read only. Or execute only. The memory mapping hardware will check the protection bits associated with each segment-table entry to prevent illegal access to memory, such as attempts to write into a read only segment. By placing an array in its own segment, the memory management hardware will automatically check that array indexes are legal and do not stray outside array boundaries.

The bits associated with each entry in the segment table, for the purpose of protection are:

- Validation bit : if the validation bit is 0, it indicates an illegal segment
- Read, write, execute bits

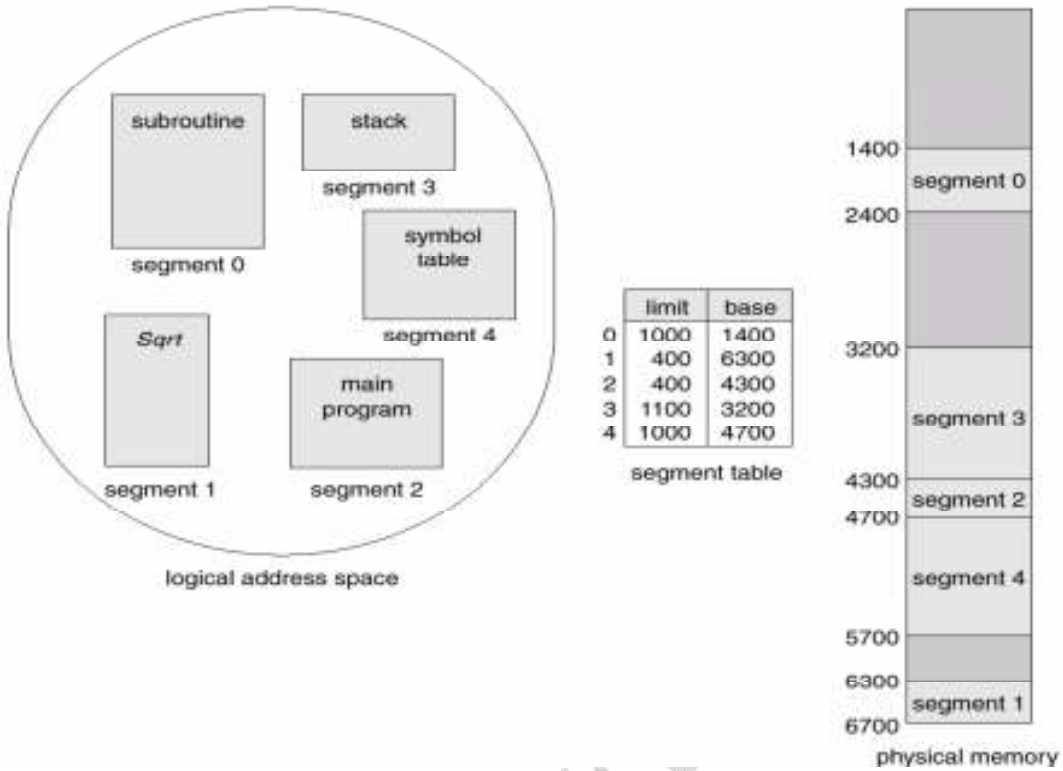


Figure 7.19 Example of segmentation.

7.7.3 Issues with Segmentation

Segmentation may then cause external fragmentation (i.e. total memory space exists to satisfy a space allocation request for a segment, but memory space is not contiguous), when all blocks of memory are too small to accommodate a segment. In this case, the process may simply have to wait until more memory becomes available or until compaction creates a larger hole. Since segmentation is by nature a dynamic relocation algorithm, we can compact memory whenever we want. If we define each process to be one segment, this approach reduces to the variable sized partition scheme. T the other extreme, every byte could be put in its own segment and relocated separately. This eliminates external fragmentation altogether, however every byte would need a base register for its relocation, doubling memory use. The next logical step- fixed sized, small segments, is paging i.e. paged segmentation. Also it might latch a job in memory while it is involved in I/O. To prevent this I/O should be done only into OS buffers.

Chapter 08

Virtual Memory

8.1 Basic Concept

An examination of real programs shows that in many cases the existence of the entire program in memory is not necessary:

- Programs often have code to handle unusual error conditions. Since these errors seldom occur in practice, this code is almost never executed.
- Arrays, lists and tables are often allocated more memory than they actually need. An array may be declared 100 by 100 elements even though it is seldom larger than 10 by 10 elements.
- Certain options of a program may be used rarely.

Even in cases where the entire program is needed, it may not be all needed at the same time. The ability to execute a program that is only partially in memory confers many benefits.

- A program would no longer be constrained by the amount of physical memory that is available. Users would be able to write programs for an extremely large virtual address space simplifying the programming task.
- Because each user program could take less physical memory, more programs could be run at the same time, with a corresponding increase in CPU utilization and throughput with no increase in response time or turnaround time.
- Less I/O would be needed to load or swap each user program into memory, so each user program would run faster.

Thus running a program that is not entirely in memory would benefit both the system and the user.

Virtual Memory:

Virtual Memory is the separation of user logical memory from physical memory. This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Virtual memory makes the task of programming easier because the programmer need not worry about the amount of physical memory, or about what code can be placed in overlays; she can concentrate instead on the problem to be programmed. In addition to separating logical memory from physical memory, virtual memory also allows files and memory to be shared by several different processes through page sharing. The sharing of pages further allows performance improvements during process creation. Virtual memory is commonly implemented as demand paging. It can also be implemented in a segmentation system. One benefit of virtual memory is efficient process creation. Yet another is the concept of memory mapped files.

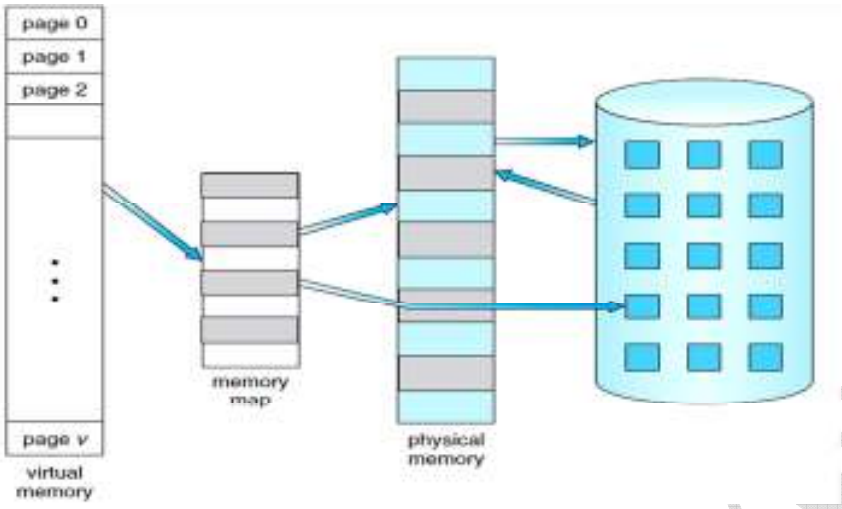


Figure 8.1: Diagram showing virtual memory that is larger than physical memory.

8.2 Demand Paging

A demand paging system is similar to a paging system with swapping. Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, however we use a lazy swapper. A lazy swapper never swaps a page into memory unless that page will be needed. Since we are now viewing a process as a sequence of pages rather than as one large contiguous address space, use of swap is technically incorrect. A swapper manipulates entire processes, whereas a pager is concerned with the individual pages of a process. Thus the term pager is used in connection with demand paging.

8.2.1 Basic Concepts

When a process is to be swapped in, the paging software guesses which pages would be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.

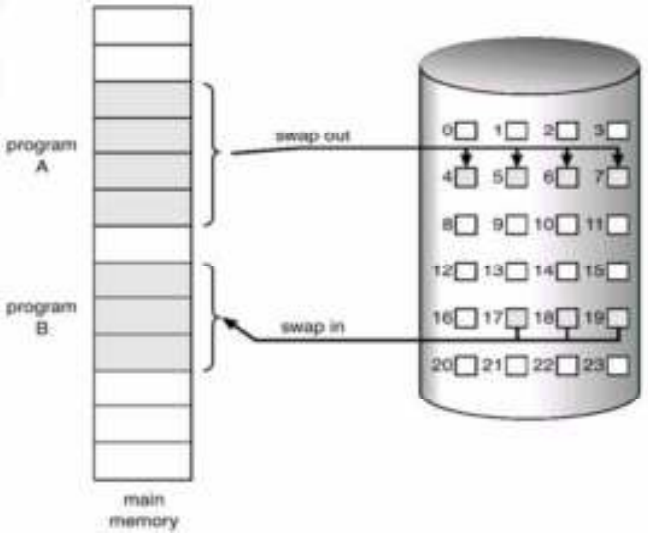


Figure 8.2 Transfer of a paged memory to contiguous disk space.

With this scheme, we need some form of hardware support to distinguish which pages are in memory and which are on disk. The valid-invalid bit scheme described in previous lectures can be used. This time however when the bit is set to valid, this value indicates that the associated page is both legal and in memory. If the bit is set to invalid this value indicates that the page either is invalid or valid but currently on the disk. The page table entry for a page that is brought into memory is set as usual but the page table entry for a page that is currently not in memory is simply marked invalid or contains the address of the page on disk.

Notice that marking a page invalid will have no effect if the process never attempts to access that page. Hence if we guess right and page in all and only those pages that are actually needed, the process will run exactly as though we had brought in all pages. While the process executes and accesses pages that are memory resident, execution proceeds normally.

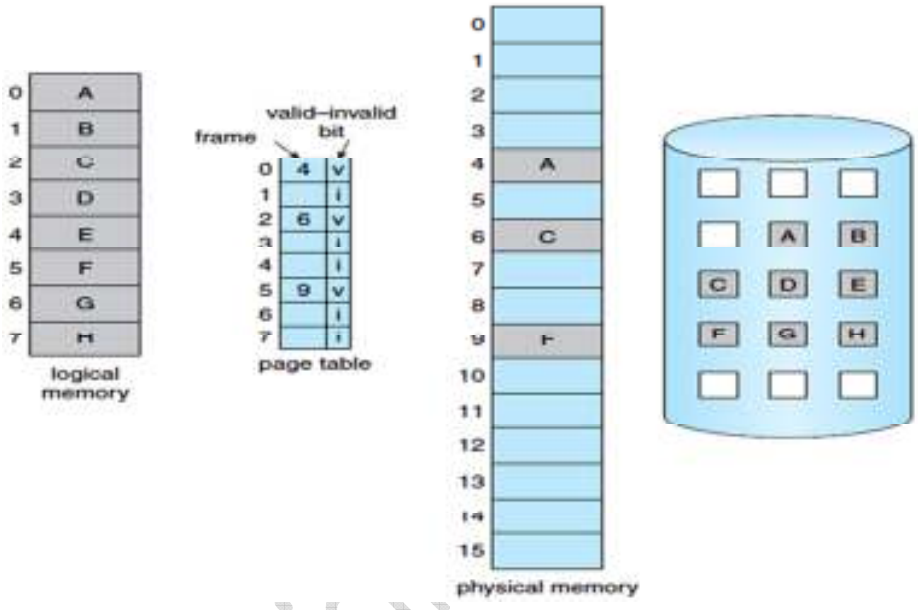


Figure 8.3: Page table when some pages are not in main memory.

8.2.2 Page Fault

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a page fault trap. The paging hardware in translating the address through the page table will notice that the invalid bit is set, causing a trap to the operating system. This trap is the result of the operating system's failure to bring the desired page into memory (in an attempt to minimize disk transfer overhead and memory requirements) rather than an invalid address error as a result of an attempt to use an illegal memory address. The procedure for handling a page fault is straightforward:

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was valid or invalid memory access.
2. If the reference was invalid we terminate the process. If it was valid, but we have not yet brought in that page, we now page it in.
3. We find a free frame (by taking one from the free-frame list, for example)
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.

6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.

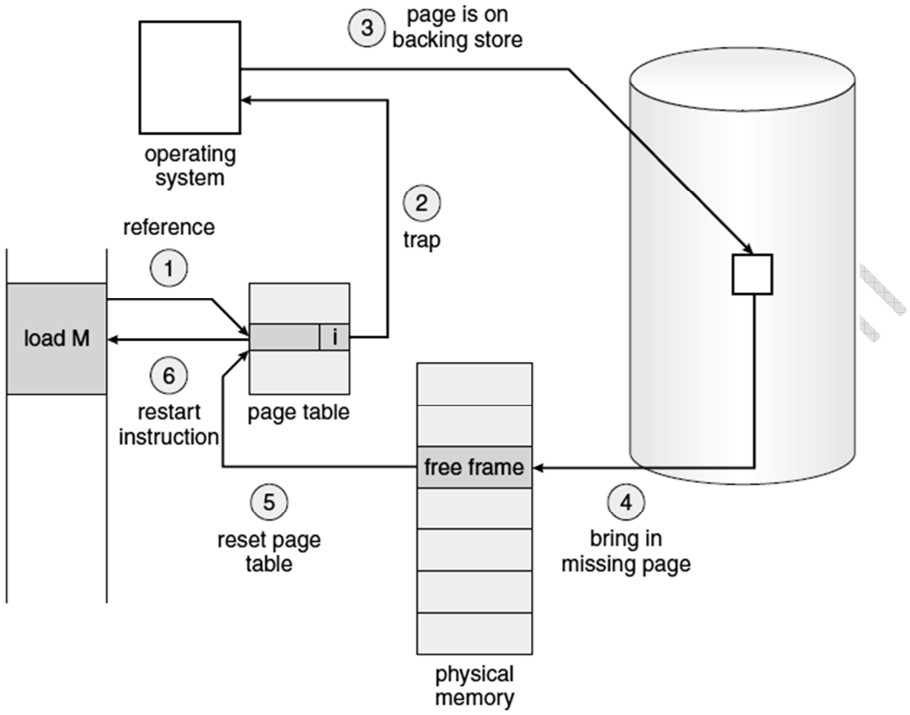


Figure 8.4: Steps in handling a page fault.

Since we save the state (registers, condition code, instruction counter) of the interrupted process when the page fault occurs, we can restart the process in exactly the same place and state except that the desired page is now in memory and is accessible. In this way we are able to execute a process even though portions of it are not yet in memory. When the process tries to access locations that are not in memory, the hardware traps the operating system (page fault). The operating system reads the desired into memory and restarts the process as though the page had always been in memory.

In the extreme case, we could start executing a process with no pages in memory. When the operating system sets the instruction pointer to the first instruction of the process, which is on a non-memory resident page, the process immediately faults for the page. After this page is brought into memory, the process continues to execute faulting as necessary until every page that it needs is in memory. At that point, it can execute with no more faults. This scheme is called pure demand paging: never bring a page into memory until it is required.

The hardware needed to support demand paging is the same as the hardware for paging and swapping:

Page table: This table has the ability to mark an entry invalid through a valid-invalid bit or special value of protection bits.

Secondary memory: This memory holds those pages that are not present in main memory. The secondary memory is usually a high speed disk. It is known as the swap device, and the section of disk used for this purpose is called the swap space.

In addition to this hardware, additional architectural constraints must be imposed. A crucial one is the need to be able to restart any instruction after a page fault. In most cases this is easy to meet, a page fault occurs while we are fetching an operand, we must fetch and decode the instruction again, and then fetch the operand. A similar problem occurs in machines that use special addressing modes, including auto increment and auto decrement modes. These addressing modes use a register as a pointer and automatically increment or decrement the register. Auto decrement automatically decrements the register before using its contents as the operand address; auto increment increments the register after using its contents. Thus the instruction

MOV (R2) +, -(R3)

Copies the contents of the location pointed to by register2 into that pointed to by register3. Now consider what will happen if we get a fault when trying to store into the location pointed to by register3. To restart the instruction we must reset the two registers to the values they had before we started the execution of the instruction.

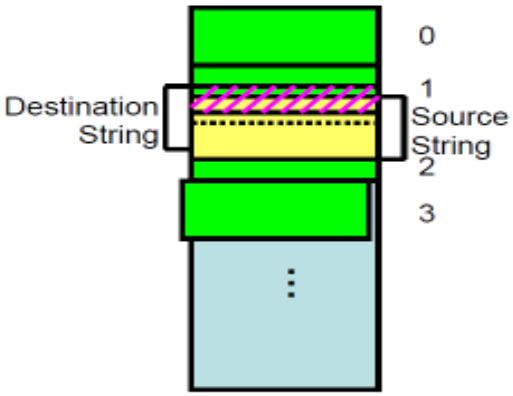


Figure 8.5: Execution of a block (string) move instruction causing part of the source to be overwritten before a page fault occurs

Another problem occurs during the execution of a block (string) move instruction. If either source or destination straddles a page boundary a page fault might occur after the move is partially done. In addition if the source and destination blocks overlap the source block may have been modified in which case we cannot simply restart the instruction, as shown in the diagram on the previous page.

Performance of demand paging

Demand paging can have a significant effect on the performance of a computer system. To see why, let us compute the effective access time for a demand paged memory. For most computer systems, the memory access time, denoted ma now ranges from 10 to 200 nanoseconds. As long as we have no page faults, the effective access time is equal to the memory access time. If, however a page fault occurs, we must first read the relevant page from disk, and then access the desired word.

Let p be the probability of a page fault ($0 \leq p \leq 1$). We would expect p to be close to zero, that is, there will be only a few page faults. The effective access time is then:

Effective access time = (1-p) * ma + p * page fault time

To compute the effective access time, we must know how much time is needed to service a page fault. A page fault causes the following sequence to occur:

1. Trap to the operating system
2. Save the user registers and process states
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on disk
5. Issue a read from the disk to a free frame:
    a. Wait in a queue for this device until the read request is serviced
    b. Wait for the device seek and/or latency time
    c. Begin the transfer of the page to a free frame
6. While waiting, allocate the CPU to some other user (CU scheduling; optimal)
7. Interrupt from the disk (I/O completed)
8. Save the registers and process state for the other user (if step 6 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state and new page table

Not all these steps are necessary in every case. For example we are assuming that in step 6, the CPU is allocated to another process while the I/O occurs. This arrangement allows multiprogramming to maintain CPU utilization, but requires additional time to resume the page fault service routine when the I/O transfer is complete.

In any case we are faced with three major components of the page fault service time:

1. Service the page fault interrupt
2. Read in the page
3. Restart the process

The first and third tasks may be reduced, with careful coding, to several hundred instructions. These tasks may take from 1 to 100 microseconds each. The page switch time, on the other hand, will probably be close to 24 milliseconds. A typical hard disk has an average latency of 8 milliseconds, a seek of 15 milliseconds, and a transfer time of 1 millisecond. Thus, the total paging time would be close to 25 milliseconds, including hardware and software time. Remember that we are looking at only the device service time. If a queue of processes is waiting for the device we have to add device queuing time as we wait for the paging device to be free to service our request, increasing even more the time to swap. If we take an average page fault service time of 25 milliseconds and a memory access time of 100 nanoseconds, then the effective access time in nanoseconds is

Effective access time = (1-p) * (100) + p (25 milliseconds)
= (1-p) * 100 + p * 25,000,000
= 100 + 24,999,900 * p

Here the effective access time is directly proportional to the page fault rate. If one access out of 1,000 causes a page fault, the effective access time is 25 microseconds. The computer would be slowed down by a factor of 250 because of demand paging! If we want less than 10 percent degradation, we need:

110 > 100 + 25,000,000 * p
10 > 25,000,000 * p

$p < 0.0000004$

That is, to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault. It is important to keep the slowdown due to paging to a reasonable level, we can allow only less than one memory access out of 2,500,000 to page fault.

It is important to keep the page fault rate low in a demand-paging system. Otherwise the effective access time increases, slowing process execution dramatically. One additional aspect of demand paging is the handling and overall use of swap space. Disk I/O to swap space is generally faster than that to the file system. It is faster because swap space is allocated in much larger blocks, and file lookups and indirect allocation methods are not used. It is therefore possible for the system to gain better paging throughput by copying an entire file image into the swap space at process startup, and then performing demand paging from the swap space. Another option is to demand pages from the from the file system initially, but to write the pages to swap space as they are replaced. This approach will ensure that only needed pages are ever read from the file system, but all subsequent paging is done from swap space.

Some systems attempt to limit the amount of swap space when binary files are used. Demand pages for such files are brought directly from the file system. However, when page replacement is called for, these pages can simply be overwritten and read in from the file system again if ever needed. Using this approach, the file system itself serves as the backing store. However swap space must still be used for pages not associated with a file; these pages include the stack and heap for a process. This technique is used in several systems including Solaris.

Performance of Demand Paging with Page Replacement

When there is no free frame available, page replacement is required, and we must select the pages to be replaced. This can be done via several replacement algorithms, and the major criterion in the selection of a particular algorithm is that we want to minimize the number of page faults. The victim page that is selected depends on the algorithm used, it might be the least recently used page, or the most frequently used etc depending on the algorithm.

Another Example

- Effective memory access is 100 ns
- Page fault overhead is 100 microseconds = 105 ns
- Page swap time is10 milliseconds = 107 ns
- 50% of the time the page to be replaced is “dirty”
- Restart overhead is 20 microseconds = 2 x 104 ns

Effective access time = 100 * (1-p) + (105 + 2 * 104 + 0.5 * 107 + 0.5 * 2 * 107) * p
= 100 * (1-p) + 15,120,000 * p

What is a Good Page Fault Rate?

For the previous example suppose p is 1%, then EAT is
= 100 * (1-p) + 15,120,000 * p
= 151299 ns

Thus a slowdown of 151299 / 100 = 1513 occurs.

For the luxury of virtual memory to cost only 20% overhead, we need

$120 > 100 * (1-p) + 15{,}120{,}000 * p$

$120 > 100 - 100\,p + 15{,}120{,}000\,p$

$p < 0.00000132$

⇒ Less than one page fault for every 755995 memory accesses!

Process Creation and Virtual Memory

Paging and virtual memory provide other benefits during process creation, such as copy on write and memory mapped files.

Copy on Write fork ()

Demand paging is used when reading a file from disk into memory and such files may include binary executables. However, process creation using fork() may bypass initially the need for demand paging by using a technique similar to page sharing. This technique provides for rapid process creation and minimizes the number of new pages that must be allocated to newly created processes. Recall the fork() system call creates a child process as a duplicate of its parent.

Traditionally fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent. However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary. Alternatively we can use a technique known as copy on write. This works by allowing the parent and child processes to initially share the same pages. These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created. For example assume a child process attempts to modify a page containing portions of the stack; the operating system recognizes this as a copy-on-write page. The operating system will then create a copy of this page mapping it to the address space of the child process. Therefore the child page will modify its copied page, and not the page belonging to the parent process. Using the copy-on-write technique it is obvious that only the pages that are modified by either process are copied; all non-modified pages may be shared by the parent and the child processes. Note that only pages that may be modified are marked as copy-on-write. Pages that cannot be modified (i.e. pages containing executable code) may be shared by the parent and the child. Copy-on-write is a common technique used by several operating systems such as Linux, Solaris 2 and Windows 2000.

When it is determined a page is going to be duplicated using copy-on-write it is important to note where the free page will be allocated from. Many operating systems provide a pool of free pages for such requests. These free pages are typically allocated when the stack or heap for a process must expand or for managing copy-on-write pages. Operating systems typically allocate these pages using a technique known as zero-fill-on demand.

Zero-fill-on-demand pages have been zeroed out before allocating, thus deleting the previous contents on the page. With copy-on-write the page being copied will be copied to a zero-filled page. Pages allocated for the stack or heap are similarly assigned zero-filled pages.

vfork()

Several versions of UNIX provide a variation of the fork() system call—vfork() (for virtual memory fork). vfork() operates differently than fork() with copy on write. With vfork() the parent process is suspended and the child process uses the address space of the parent. Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes. Therefore, vfork() must be used with caution, ensuring that the child process does not modify the address space of the parent. vfork() is intended to be used when the child process calls exec() immediately after creation. Because no copying of pages takes place, vfork() is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.

Page replacement

While a user process is executing, a page fault occurs. The hardware traps to the operating system, which checks its internal tables to see that this page is a genuine one rather than an illegal memory access. The operating system determines where the desired page is residing on the disk, but then finds that there are no free frames on the free frame list: All memory is in use.

The operating system has several options at his point. It could terminate the user process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput. Users' should not be aware that their processes are running on a paged system – paging should be logically transparent to the user. So this option is not the best choice. The operating system could swap out a process, but that would reduce the level of multiprogramming. So we explore page replacement. This means that if no free frame is available on a page fault, we replace a page in memory to load the desired page. The page-fault service routine is modified to include page replacement. We can free a frame by writing its contents to swap space, and changing the page table to indicate that the page is no longer in memory. The modified page fault service routine is:

1. Find the location of the desired page on the disk
2. Find a free frame
    a) If there is a free frame use it.
    b) If there is no free frame, use a page replacement algorithm to select a victim frame.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Restart the user process.

The following diagram shows theses steps pictorially.

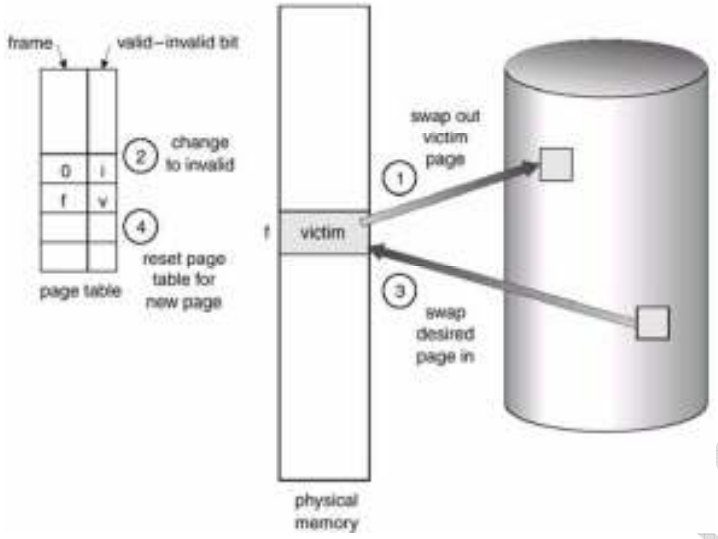


Figure 8.6: Steps needed for page replacement

We can reduce overhead by using a *modify* bit (or *dirty* bit). Each page or frame may have a modify bit associated with it in hardware. The modify bit is set by the hardware whenever any word or byte in the page is written into, indicating that the page has been modified. When we select a page for replacement we examine it's modify bit. If the bit is set, we know that the page has been modified since it was read in from the disk. In this case we must write that page to the disk. If the modify bit is not set however, the page has not been modified since it was read into memory, and hence we can avoid writing that page to disk. In the following figure we show two processes with four pages each, main memory having eight frames, with two used for resident part of operating system (leaving six frames for user processes). Both processes have three of their pages in memory and therefore there is no free frame. When the upper process (user 1) tries to access its fourth page (page number 3), a page fault is caused and page replacement is needed.

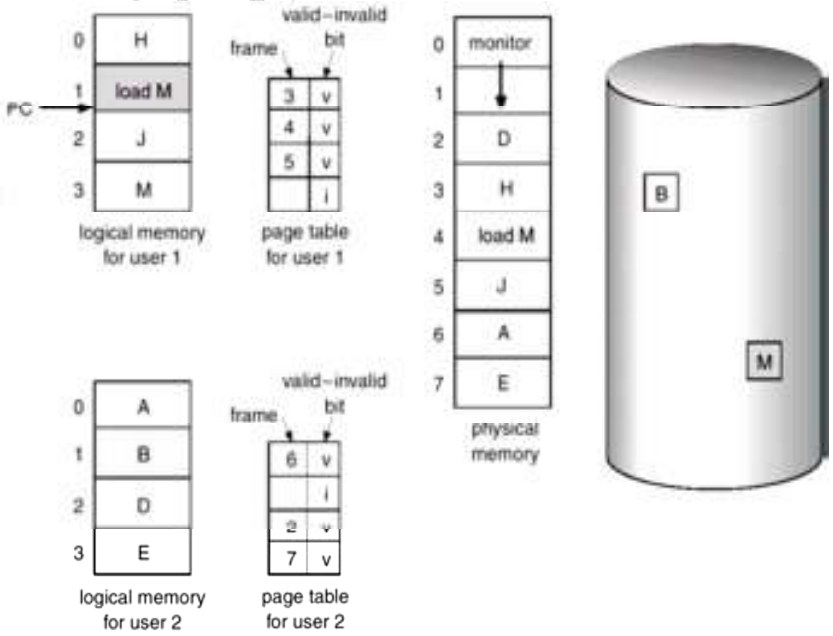


Figure 8.7: Page fault and page replacement

Page Replacement Algorithms

In general we want a page replacement algorithm with the lowest page-fault rate. We evaluate an algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string. To determine the number of page faults for a particular reference string and page replacement algorithm, we also need to know the number of page frames available. Obviously as the number of frames available increases, the number of page faults decreases.

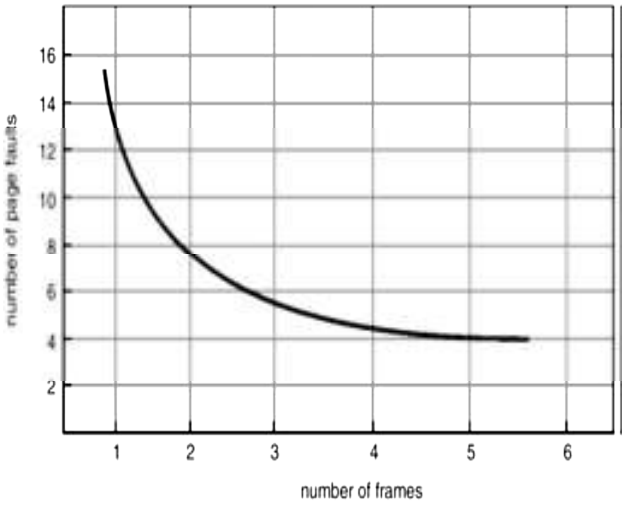


Figure 8.8: Expected relationship between number of free frames allocated to a process and the number of page faults caused by it FIFO Page Replacement

The simplest page-replacement algorithm is a FIFO algorithm. A FIFO replacement algorithm associates with each page the time when that page was brought into memory. When a page must be replaced, the oldest page is chosen. Notice that it is not strictly necessary to record the time when a page is brought in. We can create a FIFO queue to hold all pages in memory. We replace the page at the head of the queue. When a page is brought into memory we insert t at the tail of the queue. Consider the following example, in which the number of frames allocated is 4, and the reference string is 1, 2, 3, 4, 5, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4. The number of page faults caused by the process is nine, as shown below.

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 5, 4, 5, 4, 4

1	5	5	5	5	8	8	8	8
2	2	1	1	1	1	9	9	9
3	3	3	6	6	6	6	5	5
4	4	4	4	7	7	7	7	4

Figure 8.9: Example for the FIFO page replacement algorithm

The problem with this algorithm is that it suffers from Belady's anomaly: For some page replacement algorithms the page fault rate may increase as the number of allocated frames increases, whereas we would expect that giving more memory to a process would improve its performance.

Optimal Algorithm

An optimal page-replacement algorithm has the lowest page fault rate of all algorithms, and will never suffer from the Belay's algorithm. This algorithm is simply to replace the page that will not be used for the longest period of time. Use of this algorithm guarantees the lowest possible page-fault rate for a fixed number of frames. In case of the following example (which uses the same replacement string as the example for the FIFO algorithm), the number of page faults caused by the process is seven.

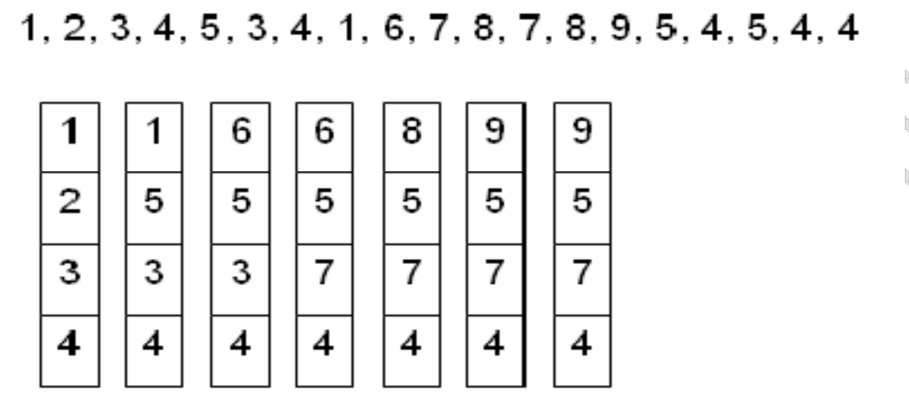


Figure 8.10: Optimal page replacement algorithm

Unfortunately this algorithm is difficult to implement because it requires future knowledge of the reference string. As a result this algorithm is used mainly for comparison.

LRU Page Replacement

If we use the recent past as an approximation of the near future, then we will replace the page that has not been used for the longest period of time. This approach is the least recently used algorithms. The following example illustrates the working of LRU algorithm.

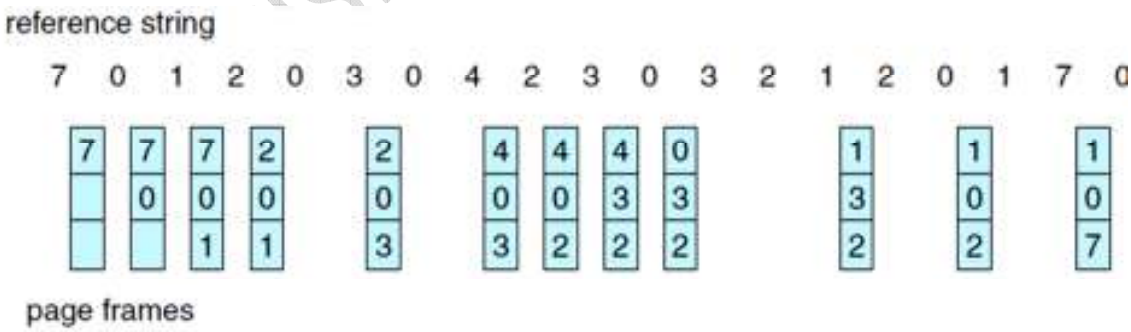
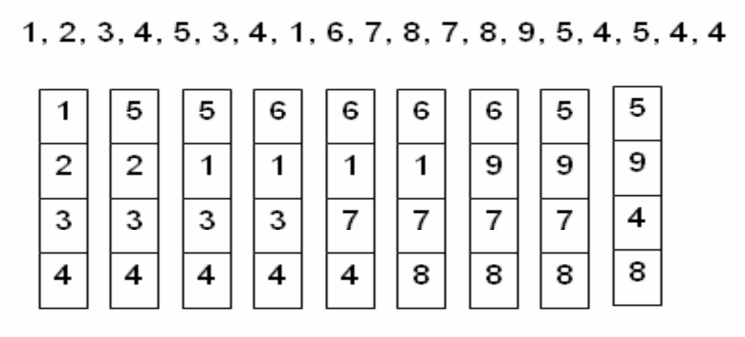


Figure 8.11: Example for the LRU page replacement algorithm

Here is another example, which uses the same reference string as used in the examples for the FIFO and optimal replacement algorithms. The number of page faults in this case is nine.



An LRU page replacement may require substantial hardware assistance. The problem is to determine an order for the frames defined by the time of last use. Two implementations are feasible:

Counter-based Implementation of LRU

In the simplest case we associate with each page table entry a time-of-use field and add to the CPU a logical clock or counter. The clock is incremented for every memory reference. Whenever a reference to a page is made, the contents of the clock register are copied to the time-of-use field in the page entry for that page. In that way we always have the time of the last reference to each page. We replace the page that has the smallest time value. This scheme requires a search of the page table to find the LRU page and a write to memory for each memory access. The times must also be maintained when page tables are changed. Overflow of the clock must be considered.

Stack-based Implementation of LRU

Another approach to implementing the LRU algorithm is to keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on top. In this way, the top of the stack is always the most recently used page and the bottom is the LRU page. Because entities must be removed from the middle of the stack, it is best implementing by a doubly linked list with a head and tail pointer. Removing a page and putting it on the top of the stack then requires changing six pointers at worst. Each update is a little more expensive, but there is no search for a replacement the tail pointer points to the bottom of the stack which is the LRU page. The following diagram shows the working of stack-based implementation of the LRU algorithm.

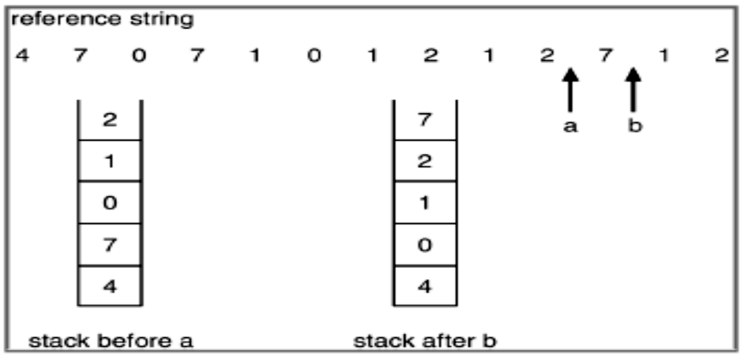


Figure 8.12: Stack based implementation of the LRU page replacement algorithm

LRU Approximation Algorithm

Few computer systems provide sufficient hardware support for true LRU page replacement. Some systems provide no hardware support and other page replacement algorithms must be used. Many systems provide some help however, in the form of a reference bit. The reference bit for a page is set by the hardware whenever that page is referenced. Reference bits are associated with each entry in the page table. Initially all bits are cleared by the operating system. As a user process executes the bit associated with each page referenced is set to 1 by the hardware. After some time we can determine which pages have been used and which have not been used by examining the reference bits. We do not know the order of use however, but we know which pages were used and which were not used.

Least Frequently Used Algorithm

This algorithm is based on the locality of reference concept— the least frequently used page is not in the current locality. LFU requires that the page with the smallest count be replaced. The reason for this selection is that an actively used page should have a large reference count. This algorithm suffers from the situation in which a page is used heavily during the initial phase of a process, but then is never used again. Since it was used heavily it has a large count and remains in memory even though it is no longer needed. One solution is to shift the counts right by 1 bit at regular intervals, forming an exponentially decaying average user count.

Most Frequently Used

The MFU page replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used; it will be in the **locality** that has just started.

Page Buffering Algorithm

The OS may keep a pool of free frames. When a page fault occurs a victim page is chosen as before. However the desired page is read into a free frame from the pool before the victim is written out. This allows the process to restart as soon as possible, without waiting for the victim to be written out. When the victim is later written out, its frame is added to the free frame pool. Thus a process in need can be given a frame quickly and while victims are selected, free frames are added to the pool in the background An expansion of this idea is to maintain a list of modified pages. Whenever the paging device is idle, a modified page is selected and is written to disk. Its modify bit is then reset. This scheme increases the probability that a page will be clean when it is selected for replacement and will not need to be written out.

Another modification is to keep a pool of free frames, but to remember which page was in which frame. Since the frame contents are not modified when a frame is written to disk, the old page can be reused directly from the free-frame pool if it is needed before that frame is reused. No I/O is needed in this case. When a page fault occurs we check whether the desired page is in the free-frame pool. If it is not we must select a free frame and read into it. This method is used together with FIFO replacement in the VAX/VMS operating system.

Local vs Global Replacement

If process P generates a page fault, page can be selected in two ways:

- Select for replacement one of its frames.
- Select for replacement a frame from a process with lower priority number.

Global replacement allows a process to select a replacement frame from the set of all frames, even if that frame belongs to some other process; one process can take a frame from another. Local replacement requires that each process select from only its allocated frames.

Consider an allocation scheme where we allow high priority processes to select frames from low priority processes for replacement. A process can select a replacement from among its own frames or the frames of any lower priority process. This approach allows a high priority process to increase its frame allocation at the expense of the low priority process.

Allocation of frames

Each process needs a minimum number of frames so that its execution may be guaranteed on a given machine. Let's consider the MOV X,Y instruction. The instruction is 6 bytes long (16-bit offsets) and might span 2 pages. Also, two pages to handle source and two pages are required to handle destination (assuming 16-bit source and destination).

Minimum frames required to guarantee execution of the MOV X,Y instruction

There are three major allocation schemes:

Fixed allocation

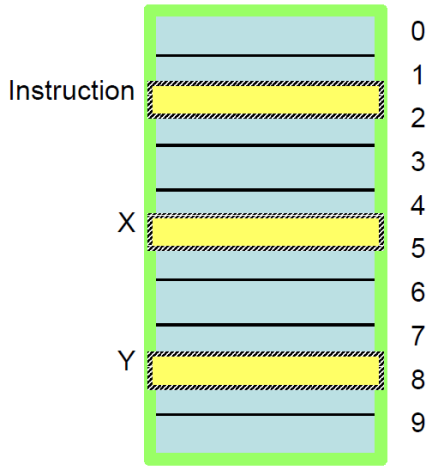
In this scheme free frames are equally divided among processes

Proportional Allocation

Number of frames allocated to a process is proportional to its size in this scheme.

Priority allocation

Priority-based proportional allocation



Here is an example of frame allocation:

Number of free frames = 64

Number of processes = 3

Process sizes: P1 = 10 pages; P2 = 40 pages; P3 = 127 pages

Fixed allocation

64/3 = 21 frames per process and one put in the free frames list

Proportional Allocation

- si = Size of process Pi
- S = Σ si
- m = Number of free frames
- ai = Allocation for Pi = (si / S) * m
  - a1 = (10 / 177) * 64 = 3 frames
  - a2 = (40 / 177) * 64 = 14 frames

- a3 = (127 / 177) * 64 = 45 frames

➢ Two free frames are put in the list of free frames

Thrashing

If a process does not have enough frames, it will quickly page fault. At this point, if a free frame is not available, one of its pages must be replaced so that the desired page can be loaded into the newly vacated frame. However since all its pages are in active use, the replaced page will be needed right away. Consequently it quickly faults again and again. The process continues to fault, replacing pages for which it then faults and brings back in right away. This high paging activity is called **thrashing**. In this case, *only one process is thrashing*. A process is thrashing if it is spending more time paging than executing.

Thrashing results on severe performance problems. The operating system monitors CPU utilization and, if CPU utilization is too low, the operating system increases the degree of multiprogramming by introducing one or more new processes to the system. This decreases the number of frames allocated to each process currently in the system, causing more page faults and further decreasing the CPU utilization. This causes the operating system to introduce more processes into the system. As a result CPU utilization drops even further and the CPU scheduler tries to increase the degree of multiprogramming even more. Thrashing has occurred and system throughput plunges.

The page fault rate increases tremendously. As a result the effective memory access time increases. Along with low CPU utilization, there is high disk utilization. There is low utilization of other I/O devices. No work is getting done, because the processes are spending all their time paging and the system spend most of its time servicing page fault. Now *the whole system is thrashing*—the CPU utilization plunges to almost zero, the paging disk utilization becomes very high, and utilization of other I/O devices becomes very low.

If a global page replacement algorithm is used, it replaces pages with no regard to the process to which they belong. Now suppose that a process enters a phase in its execution and needs more frames. It starts faulting and taking frames away from other processes. These processes need those pages however and so they also fault taking frames away from other processes. These faulting processes must use the paging device to swap pages in and out. As they queue up for the paging device, the ready queue empties. As processes wait for the paging device, CPU utilization decreases.

The following graph shows the relationship between the degree of multiprogramming and CPU utilization.

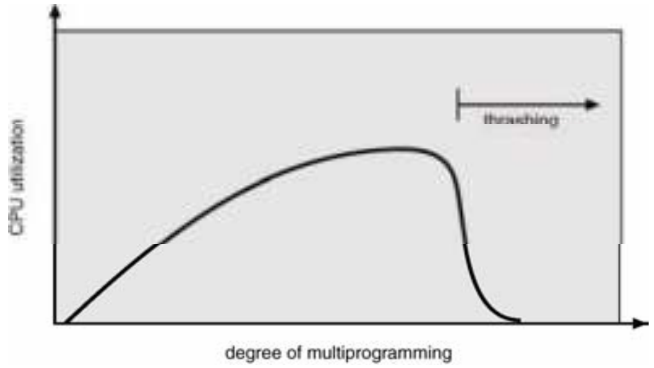


Figure 8.13: Multiprogramming and CPU utilization.

Thus in order to stop thrashing, the degree of multiprogramming needs to be reduced. The effects of thrashing can be reduced by using a local page replacement. With local replacement if one process starts thrashing it cannot steal frames from another process and cause the latter to thrash also. Pages are replaced with regard to the process if which they are a part. However, if processes are thrashing they will be in the queue for the paging device most of the time. The average service time for a page fault will increase due to the longer average queue for the paging device. Thus the effective access time will increase even for a process that is not thrashing, since a thrashing process is consuming more resources

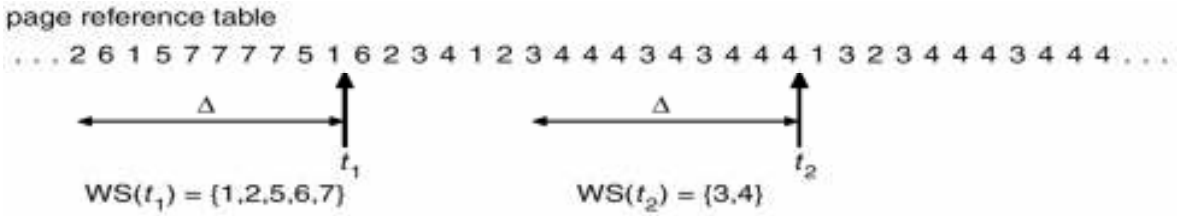
Locality of Reference

The locality model states that as a process executes it moves from locality to locality. A locality is a set of pages that are actively used together. A program is generally composed of several different localities, which may overlap. The following diagram shows execution trace of a process, showing localities of references during the execution of the

Figure 8.14: Process execution and localities of reference

Working Set Model

The working set model is based on the assumption of locality. This model uses a parameter Δ to define the working set window. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is called the working set. If a page is in active use it will be in the working set. If it no longer being used it will drop from the working set Δ time units after its last reference. Thus the working set is an approximation of the program's locality. In the following example, we use a value of Δ to be 10 and identify two localities of reference, one having five pages and the other having two pages.



We now identify various localities in the process execution trance given in the previous section. Here are the first two and last localities are: L1 = {18-26, 31-34}, L2 = {18-23, 29-31, 34}, and Last = {18-20, 24-34}. Note that in the last locality, pages 18-20 are referenced right in the beginning only and are effectively out of the locality.

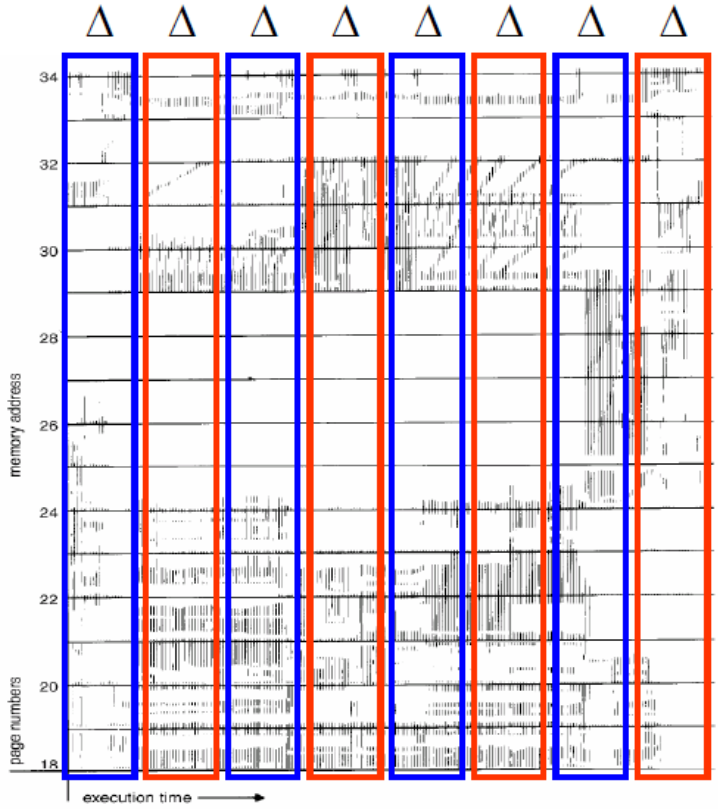


Figure 8.15: Process execution trace and localities of reference

The accuracy of the working set model depends on the selection of Δ. If Δ is too small, it will not encompass the entire locality; if Δ is too large, it may overlap several localities. In the extreme if Δ is infinite, the working set is the set of pages touched during the process execution. The most important property of the working set is its size. If we compute the working set size, WSSi for each process in the system we can consider

$D = \Sigma WSSi$

where, D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process *i* needs WSSi frames. If the total demand is greater than the total number of frames (D > m), thrashing will occur, because some processes will not have enough frames.

Use of the working set model is then simple, the operating system monitors the working set of each process and allocates to that working set enough frames to provide it with its working set size. If there are enough extra frames another process can be initiated. If the sum of the working set sizes increases, exceeding the total number of available frames, the operating system selects a process to suspend. The process' pages are written out and its frames are reallocated to other processes. The suspended process can be restarted later.

The difficulty with the working set model is to keep track of the working set. The working set window is a moving size window. At each memory reference a new reference appears at one end and the oldest reference drops off the other end. We can approximate the working set model with a fixed interval timer interrupt and a reference bit.

For example, assume Δ = 10,000 references and the timer interrupts every 5000 references. When we get a timer interrupt we copy and clear the reference bit values for each page. Thus if a page fault occurs we can examine the current reference bit and 2 in memory bits to determine whether a page was used within the last 10,000 to 15,000 references. If it was used at least one of these bits will be on, otherwise they will be off. Thus after Δ references, if one of the bits in memory = 1 then the page is in the working set. Note that this arrangement is not completely accurate because we cannot tell where within an interval of 5,000 a reference occurred. We can reduce the uncertainty by increasing the number of our history bits and the frequency of interrupts. However the cost to service these more frequent interrupts will be correspondingly higher.

Page Fault Frequency

Page fault frequency is another method to control thrashing. Since thrashing has a high page fault rate, we want to control the page fault frequency. When it is too high we know that the process needs more frames. Similarly if the page-fault rate is too low, then the process may have too many frames. The operating system keeps track of the upper and lower bounds on the page-fault rates of processes. If the page-fault rate falls below the lower limit, the process loses frames. If page-fault rate goes above the upper limit, process gains frames. Thus we directly measure and control the page fault rate to prevent thrashing. The following diagram shows the working of this scheme.

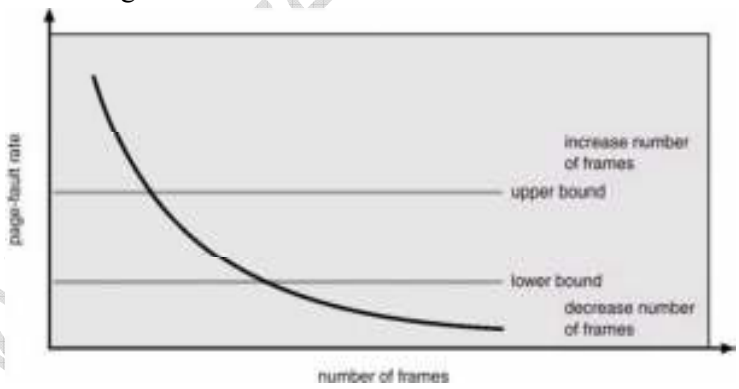


Figure 8.16: Controlling thrashing with page fault frequency

Allocating Kernel Memory

When a process running in user mode requests additional memory, pages are allocated from the list of free page frames maintained by the kernel. This list is typically populated using a page-replacement algorithm and most likely contains free pages scattered throughout physical memory, as explained earlier. Remember, too, that if a user process requests a single byte of memory, internal fragmentation will result, as the process will be granted an entire page frame.

Kernel memory is often allocated from a free-memory pool different from the list used to satisfy ordinary user-mode processes. There are two primary reasons for this:

1. The kernel requests memory for data structures of varying sizes, some of which are less than a page in size. As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation. This is especially important because many operating systems do not subject kernel code or data to the paging system.
2. Pages allocated to user-mode processes do not necessarily have to be in contiguous physical memory. However, certain hardware devices interact directly with physical memory—without the benefit of a virtual memory interface—and consequently may require memory residing in physically contiguous pages.

There are two strategies for managing free memory that is assigned to kernel processes:

1. **Buddy System**
2. **Slab Allocation.**

Buddy System

The buddy system allocates memory from a fixed-size segment consisting of physically contiguous pages. Memory is allocated from this segment using a **power-of-2 allocator**, which satisfies requests in units sized as a power of 2 (4 KB, 8 KB, 16 KB, and so forth). A request in units not appropriately sized is rounded up to the next highest power of 2. For example, a request for 11 KB is satisfied with a 16-KB segment.

Let's consider a simple example. Assume the size of a memory segment is initially 256 KB and the kernel requests 21 KB of memory. The segment is initially divided into two **buddies**—which we will call *AL* and *AR*—each 128 KB in size. One of these buddies is further divided into two 64-KB buddies— *BL* and *BR*. However, the next-highest power of 2 from 21 KB is 32 KB so either *BL* or *BR* is again divided into two 32-KB buddies, *CL* and CR. One of these buddies is used to satisfy the 21-KB request. This scheme is illustrated in Figure 8.17, where *CL* is the segment allocated to the 21-KB request.

An advantage of the buddy system is how quickly adjacent buddies can be combined to form larger segments using a technique known as **coalescing**. For example, when the kernel releases the *CL* unit it was allocated, the system can coalesce *CL* and *CR* into a 64-KB segment. This segment, *BL*, can in turn be coalesced with its buddy *BR* to form a 128-KB segment. Ultimately, we can end up with the original 256-KB segment.

The obvious drawback to the buddy system is that rounding up to the next highest power of 2 is very likely to cause fragmentation within allocated segments. For example, a 33-KB request can only be satisfied with a 64- KB segment. In fact, we cannot guarantee that less than 50 percent of the allocated unit will be wasted due to internal fragmentation. In the following section, we explore a memory allocation scheme where no space is lost due to fragmentation.

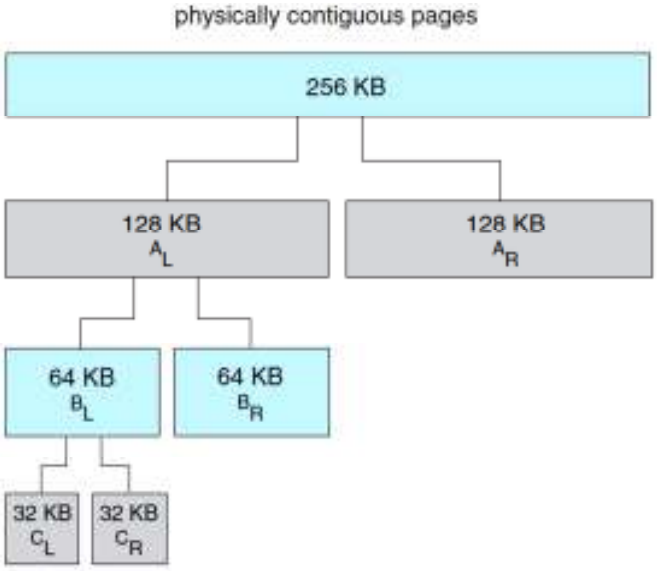


Figure 8.17: Buddy system allocation.

Slab Allocation

A second strategy for allocating kernel memory is known as **slab allocation**. A **slab** is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs. There is a single cache for each unique kernel data structure —for example, a separate cache for the data structure representing process descriptors, a separate cache for file objects, a separate cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, and so forth.

The relationship among slabs, caches, and objects is shown in Figure 8.18. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size, each stored in a separate cache.

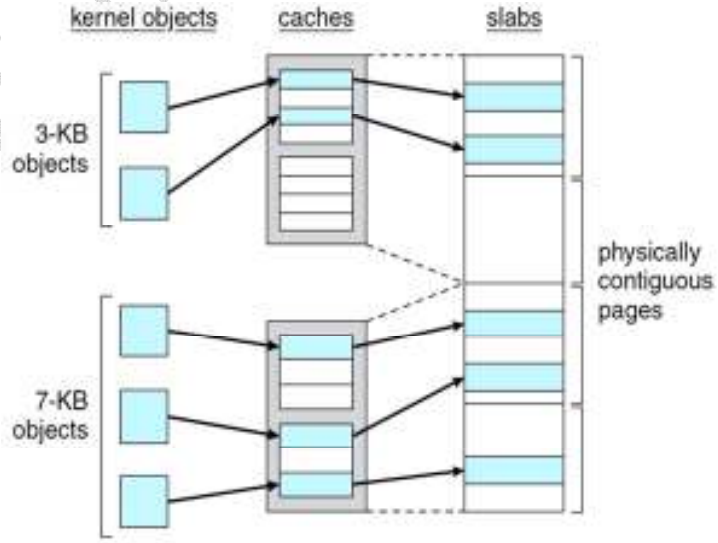


Figure 8.18: Slab Allocation.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as free—are allocated to the cache. The number of objects in the cache depends on the size of the associated slab. For example, a 12-KB slab (made up of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as used.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type struct task struct, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the struct task struct object from its cache. The cache will fulfill the request using a struct task struct object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. Full. All objects in the slab are marked as used.

2. Empty. All objects in the slab are marked as free.

3. Partial. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exists, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

Advantages of Slab Allocator

The slab allocator provides two main benefits:

1. No memory is wasted due to fragmentation. Fragmentation is not an issue because each unique kernel data structure has an associated cache, and each cache is made up of one or more slabs that are divided into chunks the size of the objects being represented. Thus, when the kernel requests memory for an object, the slab allocator returns the exact amount of memory required to represent the object.

2. Memory requests can be satisfied quickly. The slab allocation scheme is thus particularly effective for managing memory when objects are frequently allocated and de-allocated, as is often the case with requests from the kernel. The act of allocating—and releasing—memory can be a time-consuming process. However, objects are created in advance and thus can be quickly allocated from the cache. Furthermore, when the kernel has finished with an object and releases it, it is marked as free and returned to its cache, thus making it immediately available for subsequent requests from the kernel.

The slab allocator first appeared in the Solaris 2.4 kernel. Because of its general-purpose nature, this allocator is now also used for certain user-mode memory requests in Solaris. Linux originally used the buddy system; however, beginning with Version 2.2, the Linux kernel adopted the slab allocator. Recent distributions of Linux now include two other kernel memory allocators— the SLOB and SLUB allocators. (Linux refers to its slab implementation
as SLAB.)

The SLOB allocator is designed for systems with a limited amount of memory, such as embedded systems. SLOB (which stands for Simple List of Blocks) works by maintaining three lists of objects: ***small*** (for objects less than 256 bytes), ***medium*** (for objects less than 1,024 bytes), and ***large*** (for objects less than 1,024 bytes). Memory requests are allocated from an object on an appropriately sized list using a first-fit policy.

Beginning with Version 2.6.24, the SLUB allocator replaced SLAB as the default allocator for the Linux kernel. SLUB addresses performance issues with slab allocation by reducing much of the overhead required by the SLAB allocator. One change is to move the metadata that is stored with each slab under SLAB allocation to the page structure the Linux kernel uses for each page. Additionally, SLUB removes the per-CPU queues that the SLAB allocator maintains for objects in each cache. For systems with a large number of processors, the amount of memory allocated to these queues was not insignificant. Thus, SLUB provides better performance as the number of processors on a system increases.

Other considerations

Many other things can be done to help control thrashing. We discuss some of the important ones in this section.

Pre-paging

An obvious property of a pure demand paging system is the large number of page faults that occur when a process is started. This situation is the result of trying to get the initial locality into memory. Pre-paging is an attempt to prevent this high level of initial paging. The strategy is to bring into memory at one time all the pages that will be needed. Pre-paging may be an advantage in some cases. The question is simply whether the cost of using pre-paging is less than the cost of the servicing the corresponding page faults.

Page Size

How do we select a page size? One concern is the size of the page table. For a given virtual memory space, decreasing the page size increases the number of pages and hence the size of the page table. Because each active process must have its own copy of the page table, a large page size is desirable. On the other hand, memory is better utilized with smaller pages. If a process is allocated memory starting at location 00000, and continuing till it has as much as it needs, it probably will not end exactly on a page boundary. Thus, a part of the final page must be allocated. This causes internal fragmentation and to minimize this, we need a small page size.

Another problem is the time required to read or write a page. I/O time is composed of seek, latency and transfer times. Transfer time is proportional to the amount transferred, and this argues for a small page size. However, latency and seek times usually dwarf transfer times, thus a desire to minimize I/O times argues for a larger page size. I/O overhead is also reduced with small page size because locality improves. This is because a smaller page size allows each page to match program locality more accurately. Some factors (internal fragmentation, locality) argue for a small page size, whereas others (table size, I/O time) argue for a large page size. There is no best answer. However the historical trend is towards larger pages.

Program Structure

Demand paging is designed to be transparent to the user program. However, in some cases system performance can be improved if the programmer has an awareness of the underlying demand paging and execution environment of the language used in the program. We illustrate this with an example, in which we initialize a two dimensional array (i.e., a matrix).

Consider the following program structure in the C programming language. Also note that arrays are stored in row-major order in C (i.e., matrix is stored in the main memory row by row), and page size is such that each row is stored on one page.

Program 1

```
int A[1024][1024];
for (j = 0; j < 1024; j++)
for (i = 0; i < 1024; i++)
A[i,j] = 0;
```

Since this code snippet initializes the matrix column by column, it causes 1024 page faults while initializing one column. This means that execution of the code causes **1024 x 1024** page faults.

Now consider the following program structure.

Program 1

```
int A[1024][1024];
for (i = 0; i< 1024; i++)
for (j= 0; j < 1024; j++)
A[i,j] = 0;
```

In this case, matrix is accessed row by row, causing 1 page fault per row. This means that execution of the code causes **1024** page faults.