Galvin Syllabus

PART ONE ■ OVERVIEW

Chapter 1 Introduction

Chapter 2 Operating-System Structures

PART TWO ■ PROCESS MANAGEMENT

Chapter 3 Processes

[redacted]

Chapter 4 Threads & Concurrency

Chapter 5 CPU Scheduling

PART THREE ■ PROCESS SYNCHRONIZATION

PART THREE ■ PROCESS SYNCHRONIZATION

Chapter 6 Synchronization Tools

Chapter 7 Synchronization Examples

Chapter 8 Deadlocks

PART FOUR ■ MEMORY MANAGEMENT

Chapter 9 Main Memory

+ Segmentation

Chapter 10 Virtual Memory

PART FIVE ■ STORAGE MANAGEMENT

PART FIVE ■ STORAGE MANAGEMENT

Chapter 11 Mass-Storage Structure

+ Disk structure & scheduling

Chapter 12 I/O Systems

[redacted]

PART SIX ■ FILE SYSTEM

slides

Chapter 13 File-System Interface

Chapter 14 File-System Implementation

Chapter 15 File-System Internals

OS Introduction

Definitions

- A **computer** is a general purpose device that can execute sequences of instructions presented in a formal format to perform numerical calculations and other tasks.
- ***Computer science*** is the study of computer systems and computing processes.
- ***Computer hardware*** is the collection of all physical elements of the computer system.
- The ***CPU*** is the hardware that executes the instructions whereas the ***processor*** is the physical chip which contains the CPU.
- The ***core*** is the component that executes instructions and registers for storing data locally. The one main CPU with its core is capable of executing a general-purpose instruction set, including instructions from processes. Thus core is the basic computation unit of CPU..

Operating System is a system software that manages all the resources of a computer system, both hardware and software, and provides an environment in which the user can execute his/her programs in a convenient and efficient manner.
It is not mandatory to be present in the system but it becomes very inefficient to manage resources and directly access hardware as hardware has very complex mechanisms, thus operating system should be present in complex systems.

Components of Operating System: The operating system includes the always running ***kernel***, ***middleware*** (a set of software frameworks that provide additional services such as database, multimedia and graphics to application developers) and ***system programs*** that aid in managing the system while it is running.

Need of Operating System

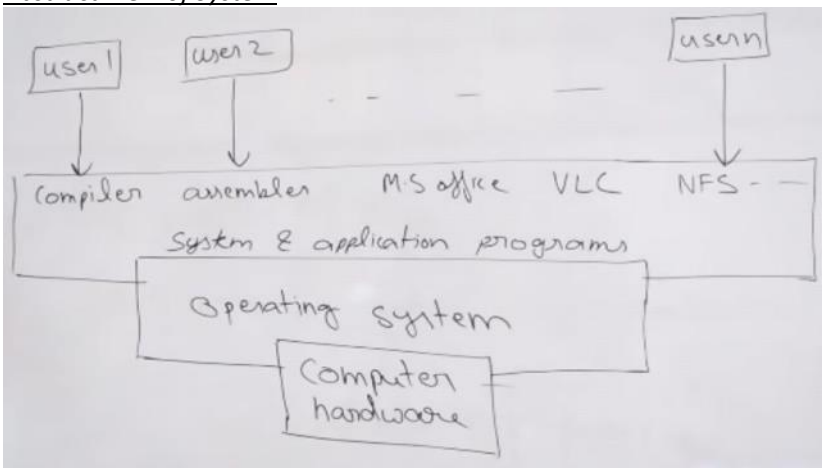
System View

1. It acts as an ***intermediatory between hardware and user***. Any application software cannot access hardware directly.
2. ***Resource Allocator & Manager***: It manage system resources(both hardware and software resources) in an unbiased fashion.
3. An operating system is a ***control program*** that manages the execution of user programs to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

User View

1. The operating system is designed mostly for ***ease of use***, to maximize the user's work with some attention paid to ***performance and security*** and none paid to resource utilization.
2. It provides a ***platform/environment*** on which other ***application programs*** are installed, thus facilitating the execution of application programs.

Abstract View of System



Goals of Operating System

1. *Primary Goal* : Convenience or User friendliness (Ease of Use, Minimal User Interference)
2. *Secondary Goal* : Efficiency(Proper Resource Allocation, High Performance, Energy Conservation)

Goals of how operating system should be designed are affected by both the choice of hardware and the type of operating system (distributed or network or real-time, etc)

User Goals: The system should be convenient to use, easy to learn and to use, reliable, safe, and fast.
Designer Goals: The system should be easy to design, implement and maintain, flexible, reliable, error free, and efficient.

General Principles in Software Engineering: Mechanism & Policies
Mechanism determine how to do something whereas *policies* determine what will be done.
One important principle is *separation of policy from mechanism*. Change in policy should not affect the mechanism of the system.

Implementation of Operating System
Traditionally, operating system were written in assembly language. However, now they are mostly written in high level languages like C/C++. Advantages of writing OS in high level languages are :

1. Code can be written faster
2. It is more compact
3. It is easier to understand & debug
4. It is easier to port from one hardware to another

Eg) MS DOS was written in Intel's Assembly language and thus can run on only intel's CPUs whereas Linux is written in C and thus can run in various CPU's.

Functions/Duties/Services of Operating System
Apart from resource allocation/management, and control program, operating system provides the following services:

1. **Process Management**
   - Creating & deleting user and system processes
   - Interprocess communication
   - Process & Thread Scheduling: Suspending & resuming processes
   - Process Synchronization
   - Deadlock Handling

2. **Memory Management**
   - Keeping track of which part of memory is being used by which job
   - Allocating and deallocating memory space
   - Deciding which processes (or parts of processes) and data to move into and out of memory

3. **I/O Operations & Device Management**
   - A memory-management component that includes buffering, caching, and spooling
   - A general device-driver interface
   - Drivers for specific hardware devices

4. **Storage Management**
   - File System Management
     - Creating and deleting files
     - Creating and deleting directories to organize files
     - Supporting primitives for manipulating files and directories
     - Mapping files onto mass storage
     - Backing up files on stable (nonvolatile) storage media

   - Mass Storage Management
     - Mounting and unmounting
     - Free-space management
     - Storage allocation
     - Disk scheduling
     - Partitioning
     - Protection

5. **Security & Protection:**
   The owners of information stored in a multiuser or networked computer system may want to control use of that

information. When several separate 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. Security of the system from outsiders is also important. Such security starts with requiring each user to authenticate himself to the system, usually by means of a password, to gain access to system resources. It extends to defending external I/O devices, including network adapters, from invalid access attempts and recording all such connections for detection of break-ins. If a system is to be protected and secure, precautions must be instituted throughout it. A chain is only as strong as its weakest link.

6. **Error Detection:**
   The operating system needs to be detecting and correcting errors constantly. Errors may occur in the CPU and memory hardware (such as a memory error or a power failure), in I/O devices (such as a parity error on disk, a connection failure on a network, or lack of paper in the printer), and in the user program (such as an arithmetic overflow or an attempt to access an illegal memory location). For each type of error, the operating system should take the appropriate action to ensure correct and consistent computing. Sometimes, it has no choice but to halt the system. At other times, it might terminate an error-causing process or return an error code to a process for the process to detect and possibly correct.

7. **User Interface:**
   Interface provided to the user by the operating system to interact with the computer system.
   There are two fundamental approaches for users to interface with the operating system:

   1. ***Command-Line Interface (CLE) or Command Interpreter***
   - It allows users to directly enter commands that are to be performed by the operating system.
   - Some operating systems include the command interpreter in the kernel like Linux. Others, such as Windows XP and UNIX, treat the command interpreter as a special program.
   - On systems with multiple command interpreters to choose from, the interpreters are known as ***shells***.
   - E.g. Bourne shell, C shell, Bourne-Again shell (BASH), Korn shell, etc.
   - The main function of the command interpreter is to get and execute the next user-specified command. These commands can be implemented in two general ways:
     a. In first approach, the command interpreter itself contains the code to execute the command. For example, a command to delete a file may cause the command interpreter to jump to a section of its code that sets up the parameters and makes the appropriate system call. In this case, the number of commands that can be given determines the size of the command interpreter, since each command requires its own implementing code.
     b. An alternative approach used by UNIX, implements most commands through system programs. In this case, the command interpreter does not understand the command in any way; it merely uses the command to identify a file to be loaded into memory and executed.

   2. **Graphical User Interface (GUI)**
      Users employ 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 (the desktop) 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—known as a folder—or pull down a menu that contains commands.

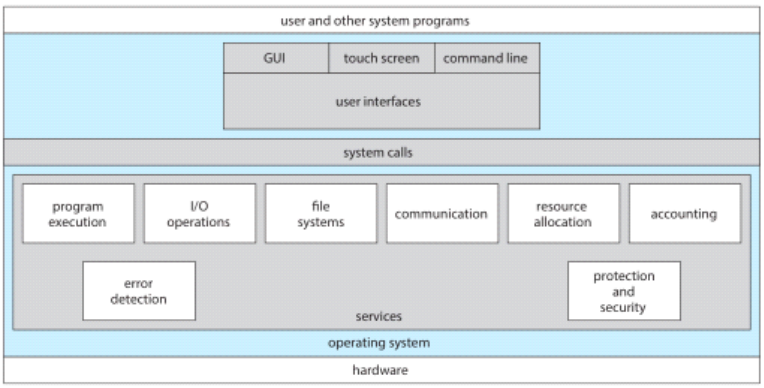


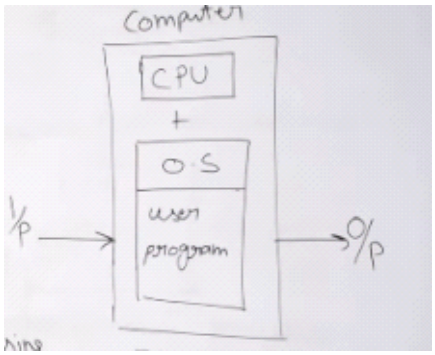
Figure 2.1 A view of operating system services.

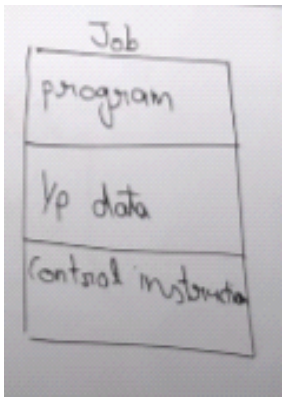
Types of Operating System

Single Processing Operating System

In starting mainframe computers:

- Common input & ouput devices were card readers & tape drivers.
- User prepared a *job* which consisted of program, input data and control instructions.
- Input Job was given in the form of *punch cards* and results also appeared in form of punch card after processing.
- So OS was very simple which was always present in memory and the major task was to transfer the control from one job to another.





Batch Processing Operating System

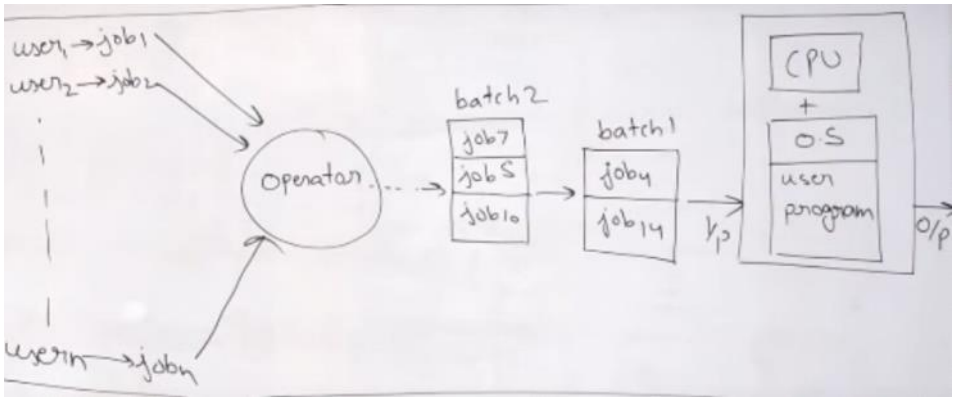
Jobs with similar needs are batched together and executed through the processor as a group.
Operator sort jobs as a deck of punch cards into batch with similar needs (like based on same compiler or programming language). Example) Payroll systems, Bank statements, etc.

Advantages:

- In a batch, job execute one after another saving time from activities like loading compiler.
- During a batch execution, no manual intervention is needed.
- It is easy to manage large work repeatedly in batch systems.

Disadvantages:

- *Memory Limitation*: Computer memory was still limited which confined efficiency.
- Interaction of input and output devices directly with CPU. CPU remains idle for most of the time as lot of time still gets wasted in loading or unloading of batches (punch cards) and grouping jobs in batches.
- The other jobs will have to wait for an unknown time if any job fails. Thus batch systems are hard to debug.
- The computer operators should be well known with batch systems

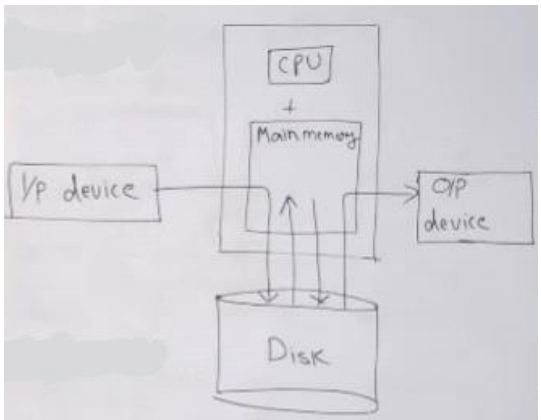


Spooling: *Simultaneous Peripheral Operations Online*
Peripheral Operations: Input/Output Operations
Online: Parallely during CPU Operations

- Input and Output devices are relatively slow compared to CPU (Digital).
- In spooling, data is stored first into the disk and then CPU interact with Disk (Digital) via main memory.
- Spooling is capable of overlapping i/o operations for one job with CPU operations of other jobs.

Advantages:
- No interaction of input & output devices with CPU
- CPU utilization time is increased as it is busy most of the time now which was idle before for majority of time.

Disadvantages:
- In starting, spooling was *uniprogramming* (synchronous programming). CPU used to wait for each program/process to finish to take up the next program/process from disk. This makes CPU idle when a program being processing depends on other events to get processed.



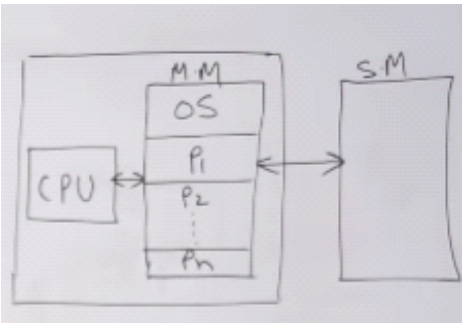
Mulitprogramming Operating System
- Multiprogramming means more than process in main memory which are ready to execute. The main memory is too small to accommodate all of these processes or jobs into it. Thus, these processes are initially kept in an area called ***job pool***. This job pool consists of all those processes awaiting allocation of main memory and CPU.
- CPU selects one job out of all these waiting jobs, brings it from the job pool to main memory and starts executing it. The processor executes one job until it is interrupted by some external factor or it goes for an I/O task.
- Process generally require CPU and I/O time, so if running process perform I/O or some other event which do not require CPU then instead of sitting idle, CPU make a context switch and pick some other process from main memory and continue processing in similar fashion.
- CPU never sits idle unless there is no process in main memory which is ready to execute at time of context switch, thus increasing CPU utilization.

Advantages:

- ***Main Objective***: Maximize CPU utilization { Reduce CPU's idle time and always have some process to be executing}
- Less waiting time, response time, etc.
- May be extended to multiple users
- Now-a-days useful when load is high

Disadvantages:

- Difficult Scheduling
- Main memory management is required
- Memory fragmentation
- Paging (Non-contiguous memory allocation)



Multitasking Operating System / Time Sharing Operating System */ Fair Share*

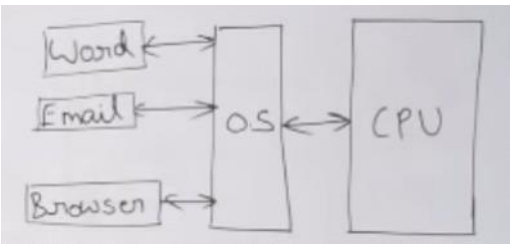
- Multitasking is multiprogramming with time-sharing or multiprogramming with Round-Robin.
- The time that each task gets to execute is called ***time quantum***.
- One CPU core can run only one process at an instant of time but it switches between processes so quickly that it gives an illusion that all processes are executing at same time.
- The task in multitasking may refer to multiple threads of the same program.
- It allows multiple users to share the computer system simultaneously.
- Main objective is better response time and switching CPU core among multiple processes so frequently so that user can interact with multiple processes at same time even while they are running (processing by CPU).
- Example : Unix OS

Advantages:

- Each task gets an equal opportunity
- Less chances of duplication of software
- CPU idle time can be reduced

Disadvantages:

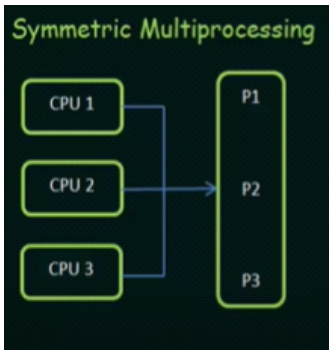
- Reliability problem
- One must have to take care of security and integrity of user programs and data
- Data communication problem



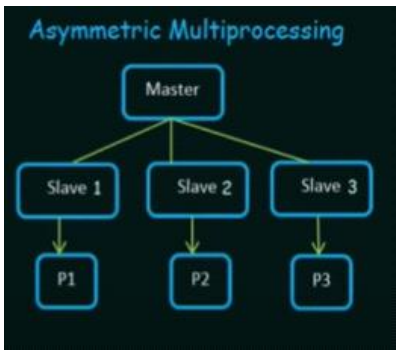
Multiprocessing Operating System

- Two or more CPU with in a single computer in close communication sharing the system bus, memory, clock and peripheral (I/O) devices.
- Different process may run on different CPU thus implementing ***true parallel execution***.
- Two types of Multiprocessing Operating System:

a. *Symmetric*: One OS control all CPU, each CPU has equal rights.



b. *Asymmetric*: *Master-slave architecture*, system task are processed on one processor and application programs on others. OR one CPU will handle all operations b/w hardware interrupt and I/O devices and rest CPU's will help in increasing computation of master CPU. It is easy to design, less circuit cost but less efficient.

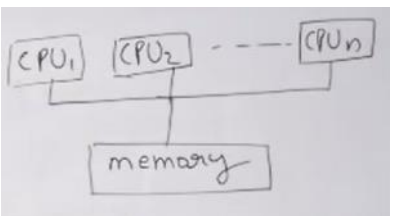


Advantages:

- *Increased throughput* (number of processes completed per unit time) as true parallel processing takes place
- *Increased reliability* (due to *graceful degradation*) : Even if some of the CPU's get malfunctioned or damaged, rest will work properly and processing will not stop.
- *Cost saving (Economic)*: All other computer resources like I/O devices , memory, etc. need to be same(shared by CPU's). Only multiple copies of CPU are to be embedded.
- *Battery Efficient*: During less load, only few of the CPU Processors are required and rest remains idle which reduces the battery consumption.

Disadvantages:

- *More Complex* (hardware designing and software compatibility)
- *Overhead or coupling* (coordination b/w different CPUs) reduces throughout
- To tackle needs of multi-core CPUs, *more memory* is required which increases the cost of the system.



Distributed Operating System / Loosely Coupled systems

- Various autonomous interconnected computers communicate each other using a shared communication network. Independent systems possess their own memory unit and CPU.
- These system's processors differ in size and function.

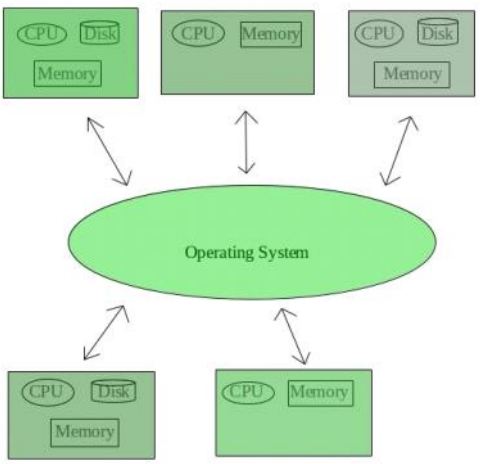
Example) LOCUS

Advantages:

- *Remote access* is enabled within the devices connected in that network. Users can access files or software present on any system connected to the network.
- Failure of one will not affect the other network communication, as all systems are independent from each other
- Electronic mail increases the data exchange speed
- Since resources are being shared, computation is highly fast and durable
- Load on host computer reduces
- These systems are easily scalable as many systems can be easily added to the network
- Delay in data processing reduces

Disadvantages:

- Failure of the main network will stop the entire communication
- To establish distributed systems the language which are used are not well defined yet
- These types of systems are not readily available as they are very expensive. Not only that the underlying software is highly complex and not understood well yet



Network Operating System/ Tightly Coupled Systems

- These systems run on a server and provide the capability to manage data, users, groups, security, applications, and other networking functions.
- They allow shared access of files, printers, security, applications, and other networking functions over a small private network.
- All the users are well aware of the underlying configuration, of all other users within the network, their individual connections etc. and that's why these computers are popularly known as *tightly coupled systems*.

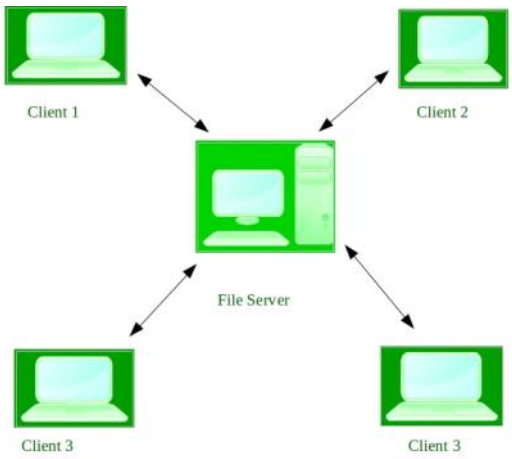
Example) Microsoft Windows Server 2003 & 2008, Mac OS X, UNIX, Linux, etc.

Advantages:

- Highly stable centralized servers
- Security concerns are handled through servers
- New technologies and hardware up-gradation are easily integrated to the system
- Server access are possible remotely from different locations and types of systems

Disadvantages:

- Servers are costly
- User has to depend on central location for most operations
- Maintenance and updates are required regularly



Real-Time Operating System

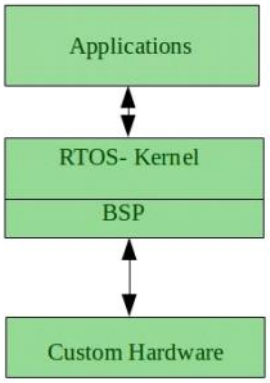
- The time interval required to process and respond to inputs is very small. This time interval is called ***response time***.
- Real-time systems are used when time constraints or requirements are very ***strict***(well defined and fixed) like missile systems, air traffic control systems, robots etc.
- There will be jobs with ***deadlines***. Results produced after deadline are treated as waste/undesired.
- There are two types of Real Time Operating System:
  a. *Hard Real-Time Operating System*: These OSs are meant for the applications where time constraints are very strict and even the shortest possible delay is not acceptable. These systems are built for saving life like automatic parachutes or air bags which are required to be readily available in case of any accident. Virtual memory is almost never found in these systems.
  b. *Soft Real-Time Operating System*: These OSs are for applications where for time-constraint is less strict.
- *Examples*: Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

Advantages:

- *Maximum Consumption*: Maximum utilization of devices and system,thus more output from all the resources
- *Task Shifting*: Time assigned for shifting tasks in these systems are very less. For example in older systems it takes about 10 micro seconds in shifting one task to another and in latest systems it takes 3 micro seconds.
- *Focus on Application*: Focus on running applications and less importance to applications which are in queue.
- *Real time operating system in embedded system*: Since size of programs are small, RTOS can also be used in embedded systems like in transport and others.
- *Error Free*: These types of systems are error free.
- *Memory Allocation*: Memory allocation is best managed in these type of systems.

Disadvantages:

- *Limited Tasks*: Very few tasks run at the same time and their concentration is very less on few applications to avoid errors.
- *Use heavy system resources*: Sometimes the system resources are not so good and they are expensive as well.
- *Complex Algorithms*: The algorithms are very complex and difficult for the designer to write on.
- *Device driver and interrupt signals*: It needs specific device drivers and interrupt signals to response earliest to interrupts.
- *Thread Priority*: It is not good to set thread priority as these systems are very less prone to switching tasks.

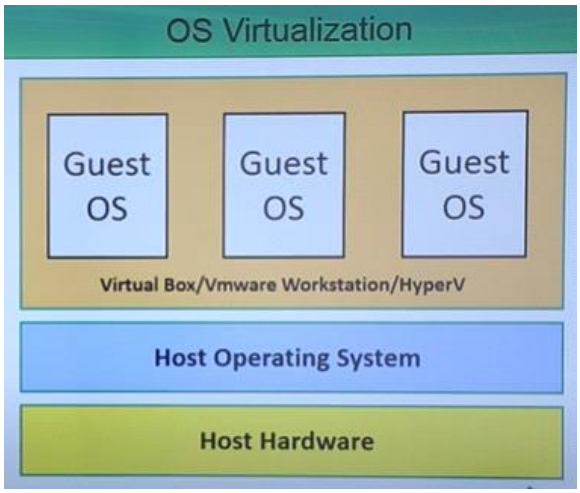


Virtualization

With operating system virtualization, nothing is preinstalled or permanently loaded on the local device and no hard-disk drive is needed. Everything runs from a network using one of the kind of virtual disk (private or shared).

Advantages:

- OS virtualization usually imposes very little or no overhead.
- It is capable of live migration.
- It can also use dynamic load balancing of containers between nodes and a cluster.



Key	Virtualization	Containerization
Basic	Virtualization is the technology which can simulate your physical hardware (such as CPU cores, memory, disk) and represent it as seperate machine	Containerization is os-level virtualization. It doesn't simulate the entire physical machine
Detaching Layer	It used Hypervisor to detach the physical machine	It used docker engine in case Docker
Isolation Level	It has hardware level isolation so fit is fully secured	It has process level isolation
LightWeight	It is heavyweight	It is very lightweight
Portable	It is not portable	It is very portable. We can build, ship and run anywhere

Application & System Programs

- ***Computer software*** is the collection of all programs stored in and executed by a computer system.

- ***Application software*** performs specific task for the user and solves certain common user problems. Knowledge of System's architecture is not necessary for application software. Such application programs include web browsers, word processors and text formatters, spreadsheets, database systems, compilers, plotting and statistical-analysis packages, and games.

- ***System software*** are general-purpose softwares which operates and controls the computer system, and provides a platform/environment to run application software. Knowledge of System's architecture is necessary for system software. Some portion of system software is written in low level language. Eg) Compiler, Assember, Interpreter etc.

High Level Language vs Low Level Language

S.NO	HIGH LEVEL LANGUAGE	LOW LEVEL LANGUAGE
1.	It is programmer friendly language.	It is a machine friendly language.
2.	High level language is less memory efficient.	Low level language is high memory efficient.
3.	It is easy to understand.	It is tough to understand.
4.	It is simple to debug.	It is complex to debug comparatively.
5.	It is simple to maintain.	It is complex to maintain comparatively.
6.	It is portable.	It is non-portable.
7.	It can run on any platform.	It is machine-dependent.
8.	It needs compiler or interpreter for translation.	It needs assembler for translation (for assembly language)
9.	It is machine independent.	It is machine dependent.
10.	Eg) C, C++, Java, Python etc.	Types) Machine languge and Assembly language

- ***Machine language*** is the only language which the machine can interpret on its own. It is represented in 0's and 1's and very difficult to interpret/learn by an user. No translator is required for machine language.
- ***Assembly language*** is somewhat intermediary language. Assembly languages use mnemonics (numbers, symbols, and abbreviations) instead of 0s and 1s. For example: for addition, subtraction and multiplications it uses symbols likes Add, sub and Mul, etc. Assembler is used as translator to convert mnemonics into machine understandable form.

- The programs which are written in high level languages (like C/C++, Java) are known as ***source code***. These source code cannot be executed directly by the computer and must be converted into machine language to be executed.
- The translator system software that is used to translate the program written in high-level language into machine code is called **Language Processor.**
- The program translated into machine language using language processors is known as ***object program or object code***.

Language Processors

1. ***Compiler***
   The language processor that reads the complete source program written in high level language as a whole in one go and translates it into an equivalent program in machine language is called as a Compiler.
   In a compiler, the source code is translated to object code successfully if it is free of errors. The compiler specifies the errors at the end of compilation with line numbers when there are any errors in the source code. The errors must be removed before the compiler can successfully recompile the source code again.
   Example: C, C++, C#, Java

2. ***Interpreter***
   The translation of single statement of source program into machine code is done by language processor and executes it immediately before moving on to the next line is called an interpreter. If there is an error in the statement, the interpreter terminates its translating process at that statement and displays an error message. The interpreter moves

on to the next line for execution only after removal of the error. An Interpreter directly executes instructions written in a programming or scripting language without previously converting them to an object code or machine code.
Example: Perl, Python and Matlab.

3. ***Assembler***
   The Assembler is used to translate the program written in Assembly language into machine code. The source program is a input of assembler that contains assembly language instructions. The output generated by assembler is the object code or machine code understandable by the computer.

Compiler vs Interpreter

COMPILER	INTERPRETER
A compiler is a program which coverts the entire source code of a programming language into executable machine code for a CPU.	interpreter takes a source program and runs it line by line, translating each line as it comes to it.
Compiler takes large amount of time to analyze the entire source code but the overall execution time of the program is comparatively faster.	Interpreter takes less amount of time to analyze the source code but the overall execution time of the program is slower.
Compiler generates the error message only after scanning the whole program, so debugging is comparatively hard as the error can be present any where in the program.	Its Debugging is easier as it continues translating the program until the error is met
Generates intermediate object code.	No intermediate object code is generated.
Examples: C, C++, Java	Examples: Python, Perl

Compiler vs Assembler

COMPILER	ASSEMBLER
Compiler converts the source code written by the programmer to a machine level language.	Assembler converts the assembly code into the machine code.
Compiler input source code.	Assembler input assembly language code.
It converts the whole code into machine language at a time.	But the Assembler can’t do this at once.
A Compiler is more intelligent than an Assembler.	But, an Assembler is less intelligent than a Compiler.
The compilation phases are lexical analyzer, syntax analyzer, semantic analyzer, intermediate code generated, a code optimizer, code generator, and error handler	Assembler makes two phases over the given input, first phase and the second phase.
The output of compiler is a mnemonic version of machine code.	The output of assembler is binary code.
C, C++, Java, and C# are examples of compiled languages.	GAS, GNU is an example of an assembler.

Assember vs Interpreter

S.NO.	ASSEMBLER	INTERPRETER
1.	It converts low-level language to the machine language.	It converts high-level language to the machine language.
2.	The program for an Assembler is written for particular hardware.	The program for an Interpreter is written for particular language.
3.	It is one to one i.e. one instruction translates to only one instruction.	It is one to many i.e. one instruction translates to many instruction.
4.	It translates entire program before running.	It translates program instructions line by line.
5.	Errors are displayed before program is running.	Errors are displayed for the every interpreted instruction (if any).

6.	It is used only one time to create an executable file.	It is used everytime when the program is running.
7.	Requirement of memory is less.	Requirement of memory is more.
8.	Programming language that it convert is Assembly language.	Programming language that it convert are PHP, Python, Perl, etc.

Types of System Programs or System Services

System services, 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 (Manage Files itself)***
   These programs create, delete, copy, rename, print, list, and generally access and manipulate files and directories.

2. ***Status Information***
   These programs ask the system for Date, Time, Amount of available memory or disk space, Number Of users, Detailed performance, Logging, and debugging information etc.

3. ***File Modification (Manage Contents of File)***
   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 Python) are often provided with the operating system.

5. ***Program Loading & Execution***
   Once a program is assembled or compiled, it must be loaded into memory to be executed. The system may provide *absolute loaders, relocatable loaders, linkage editors, and overlay loaders*. Debugging systems for either higher-level languages or machine language are needed 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***.
   Examples are network daemon, process schedulers, system error monitoring services, and print servers.

Loaders & Linkers

A program resides on disk as a binary executable file. To run on a CPU, the program must be brought into memory and placed in the context of a process. The steps in process from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core are:

- Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as an ***relocatable object file***.
- Next, the ***linker*** combines these relocatable object files into a single binary executable file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library.
- A ***loader*** is used to load the binary executable file into main memory using the address space of the newly created process, where it is eligible to run on a CPU core. After allocating the memory space to the executable module in main memory , it transfers control to the beginning instruction of the program.

- An activity associated with linking and loading is ***relocation***, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that the code can call library functions and access its variables as it executes.

Differences between Linking and Loading:

A) The key difference between linking and loading is that the linking generates the executable file of a program whereas, the loading loads the executable file obtained from the linking into main memory for execution.
B) The linking intakes the object module of a program generated by the assembler. However, the loading intakes the executable module generated by the linking.
C) The linking combines all object modules of a program to generate executable modules it also links the library function in the object module to built-in libraries of the high-level programming language. On the other hand, loading allocates space to an executable module in main memory.

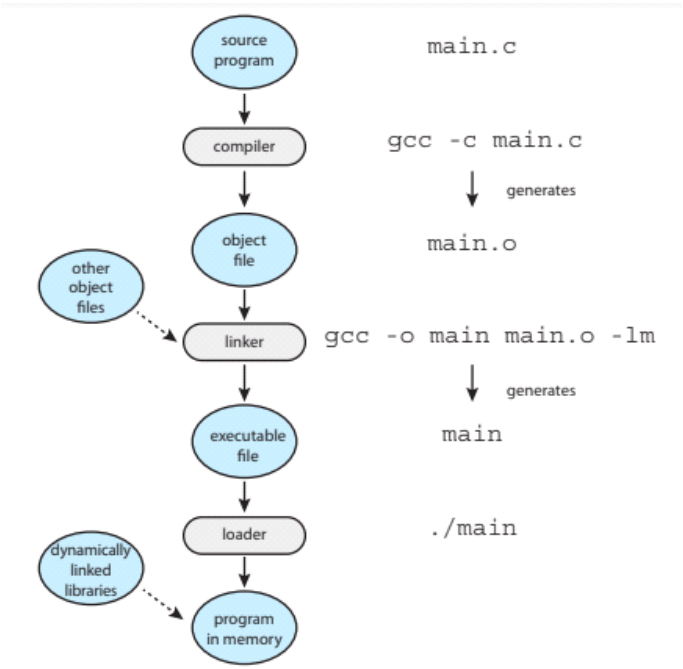


Figure 2.11 The role of the linker and loader.

Cross Platform Application Programs

Q) How is application program made available to run on multiple operating systems?

1. The application can be ***written in an interpreted language*** (such as Python or Ruby) that has an interpreter available for multiple operating systems.
   Performance suffers relative to that for native applications, and the interpreter provides only a subset of each operating system's features, possibly limiting the feature sets of the associated applications.

2. The application can be ***written in a language that includes a virtual machine*** containing the running application. The virtual machine is part of the language's full RTE.
   Eg) Java has an RTE that includes a loader, byte-code verifier, and other components that load the Java application into the Java virtual machine. This RTE has been ported, or developed, for many operating systems, from mainframes to smartphones, and in theory any Java app can run within the RTE wherever it is available.
   Systems of this kind have disadvantages similar to those of interpreter (limiting feature sets associated with the application).

3. The application developer can ***use a standard language or API*** in which the compiler generates binaries in a machine- and operating-system-specific language.

The application must be ported to each operating system on which it will run. This porting can be quite time consuming and must be done for each new version of the application, with subsequent testing and debugging.
Eg) POSIX API for variants of UNIX OS.

Q) Why are cross-platform application program still a challenging task

OR

why ***application programs are operating-system specific***?

1. At the application level, an application designed to call one set of APIs will not work on an operating system that does not provide those APIs.

2. Each operating system has a binary format for applications that dictates the layout of the header, instructions, and variables. Those components need to be at certain locations in specified structures within an executable file so the operating system can open the file and load the application for proper execution.

3. CPUs have varying instruction sets, and only applications containing the appropriate instructions can execute correctly.

4. Operating systems provide system calls that allow applications to request various activities, such as creating files and opening network connections. Those system calls vary among operating systems in many respects, including the specific operands and operand ordering used, how an application invokes the system calls, their numbering and number, their meanings, and their return of results.

System Operations

A modern general-purpose computer system consists of one or more CPUs and a number of device controllers connected through a common ***bus*** that provides access between components and shared memory.

Each ***device controller*** is in charge of a specific type of device. The CPU and the device controllers can execute concurrently, competing for memory cycles. A device controller maintains some ***local buffer storage*** and a ***set of special-purpose registers***. The device controller is responsible for moving the data between the peripheral devices that it controls and its local buffer storage.

Operating systems have a ***device driver*** for each device controller. This device driver understands the device controller and provides an uniform interface to the device to the rest of the operating system. The CPU and the device controllers can execute in parallel, competing for memory cycles. To ensure orderly access to the shared memory, a memory controller is provided whose function is to synchronize access to the memory. Drivers are ***hardware dependent*** and ***operating-system-specific***, though it need not be present in kernel of operating system.

For a computer to start running (powered up or rebooted), it needs to have an initial program to run. This initial program known as ***bootstrap program*** is stored within the computer hardware in firmware or ROM (specifically EEPROM). It initializes all aspects of the system, from CPU registers to device controllers to memory contents. The bootstrap program must know how to load the operating system and how to start executing that system. To accomplish this goal, the bootstrap program must locate the operating-system kernel and load it into memory.

Once the kernel is loaded and executing, it can start providing services to the system and its users. Some services are provided outside of the kernel by system programs that are loaded into memory at boot time to become ***system daemons***, which run the entire time the kernel is running. Once this phase is complete, the system is fully booted, and the system waits for some event to occur. Events are mostly signaled by the occurrence of an ***interrupt*** from hardware or software.

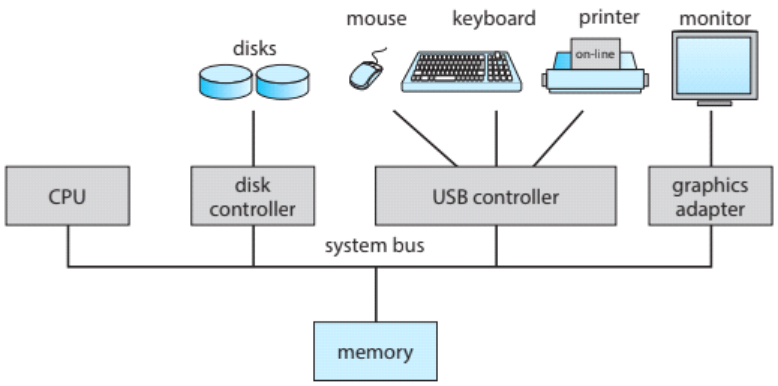


Figure 1.2 A typical PC computer system.

Interrupts

An important responsibility of OS is to manage hardware resources connected to the system. The OS needs to communicate with the system's individual devices. But there is an issue here, the processor speed >> hardware speed. Hence it is not ideal for CPU to issue an I/O request to hardware and then wait. It is better to free the CPU and handle some other work.
Some ways to overcome the issue are:

1. *Polling*: The kernel can check the status of hardware periodically. Here, the limitation is the overhead of checking regularly whether the hardware device is ready or not.

2. *Interrupt*: An interrput is an input signal to the processor (sent via the system bus) indicating an event needs immediate action. Hardware device has the ability to signal to the kernel when attention is needed.

Working of Interrupts

- When the CPU is interrupted, it stops what it is doing and immediately transfers execution to a fixed location. The fixed

location usually contains the starting address where the Service Routine of the interrupt is located. The Interrupt Service Routine (ISR) executes and on completion, the CPU resumes the interrupted computation.

- The method of managing the transfer of control to the appropriate ISR would be to invoke a general routine to examine the interrupt information. The routine, in turn, would call the interrupt-specific handler.
- Interrupts must be handled quickly, as they occur very frequently. A table of pointers to interrupt routines can be used instead to provide the necessary speed. The interrupt routine is called indirectly through the table, with no intermediate routine needed.
- The locations where table of pointers are stored, holds the addresses of the interrupt service routines for the various devices. This array, or ***interrupt vector***, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device.
- The interrupt architecture must also save the state information of whatever was interrupted, so that it can restore this information after servicing the interrupt. After the interrupt is serviced, the saved return address is loaded into the program counter, and the interrupted computation resumes as though the interrupt had not occurred.

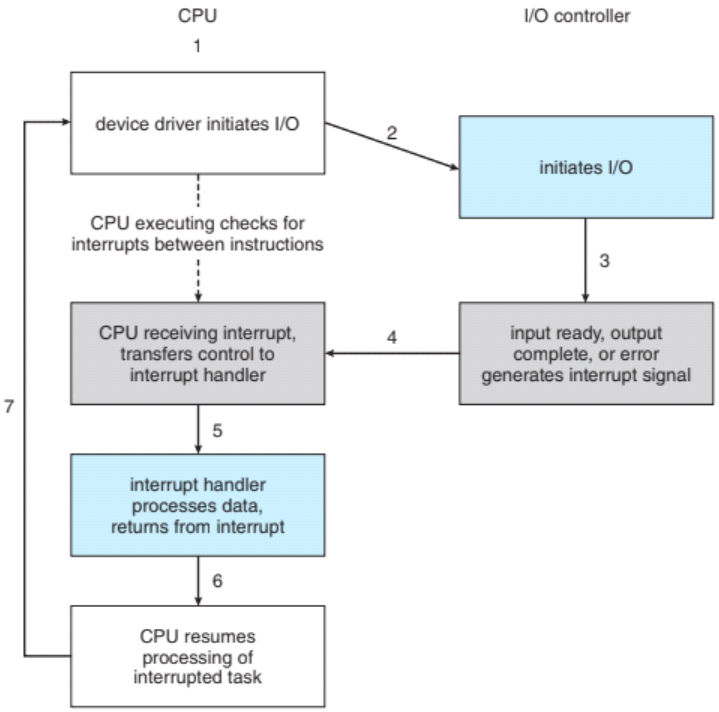
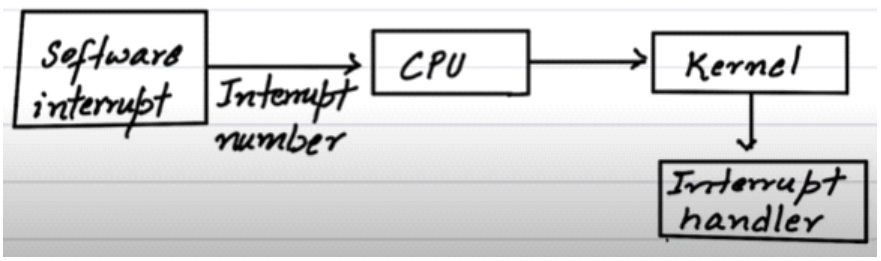


Figure 1.4 Interrupt-driven I/O cycle.

Types of Interrupts

1. **Hardware Interrupts** (Asynchronous Interrupts)
   - It is issued by a hardware device like I/O device (keyboard, mice, etc.), memory device (disk), etc.
   - These interrputs can arrive at any time even if the processor is in the middle of execution of a process, hence they are asynchronous in nature.

2. **Software Interrupts** (Synchronous Interrputs) : Exceptions/Faults and Traps
   - It is issued by the processor itself.
   - It is caused due to the process that is running on the processor i.e. it is caused by an instruction itself, hence they are synchronous (with the processor) in nature.



Types of Software Interrupts

a. *Traps*: They are generated intentionally by the program when they want to request an OS service. System calls are called from a program using traps.Eg) I/O management system calls like read(), write(), memory management system calls like requesting more heap memory etc.

b. *Faults*: They are generated when a program causes a recoverable error. Interrput handler is called which handles the error. Example) Segmentation fault, page fault, Division by zero etc.

c. *Abort*: They are generated when a program causes non-recoverable error. The interrput handler terminates the program. One cannot catch these types of errors. Eg) Hardware errors like corruption of disk

Storage Structure

- The CPU can load instructions only from memory, so any programs must first be loaded into memory to run. General-purpose computers run most of their programs from rewritable memory, called ***main memory (RAM)*** (commonly implemented in a semiconductor technology called *dynamic random-access memory (DRAM)*).

- ***Load or Store instructions:*** All forms of memory provide an array of bytes. Each byte has its own address. Interaction is achieved through a sequence of load or store instructions to specific memory addresses. The *load instruction* moves a byte or word from main memory to an internal register within the CPU, whereas the *store instruction* moves the content of a register to main memory. Aside from explicit loads and stores, the CPU automatically loads instructions from main memory for execution from the location stored in the program counter.

- ***Instruction - Execution Cycle:*** In a von Neumann architecture, an instruction–execution cycle first fetches an instruction from memory and stores that instruction in the ***instruction register***. The instruction is then decoded and may cause operands to be fetched from memory and stored in some internal register. After the instruction on the operands has been executed, the result may be stored back in memory.

- Ideally, we want the programs and data to reside in main memory permanently. This arrangement usually is not possible on most systems for two reasons:
  a. Main memory is usually too small to store all needed programs and data permanently.
  b. Main memory is volatile—it loses its contents when power is turned off or otherwise lost.

- Thus, most computer systems provide ***secondary storage*** as an extension of main memory. The main requirement for secondary storage is that it be able to hold large quantities of data and programs permanently.Most programs (system and application) are stored in secondary storage until they are loaded into memory. Many programs then use secondary storage as both the source and the destination of their processing. Eg) Hard Disk Drives (HDD), Solid State Drives (SSD), Non Volatile Memory (NVM) devices (like flash memory in smartphones), etc.

- Storage that are slow enough and large enough that they are used only for special purposes like to store backup copies of material stored on other devices are called ***tertiary storage***. Eg) Magnetic Tapes, Optical Disks, Optical Tapes, etc.

  Note: Non-Volatile storage can be classified into two distinct types:
  - ***Mechanical***: A few examples of such storage systems are HDDs, optical disks, holographic storage, and magnetic tape.
  - ***Electrical***: A few examples of such storage systems are flash memory, FRAM, NRAM, and SSD.

  Mechanical storage is generally larger and less expensive per byte than electrical storage. Conversely, electrical storage is typically costly, smaller and faster than mechanical storage.

- ***Cache Memory*** is a special very high-speed memory. It is used to speed up and synchronizing with high-speed CPU. Cache memory is costlier than main memory or disk memory but economical than CPU registers. Caches can be installed to improve performance where a large disparity in access time or transfer rate exists between two components.

- Each storage system provides the basic functions of storing a datum and holding that datum until it is retrieved at a later time. The main differences among the various storage systems lie in speed, size, and volatility. There is a ***trade-off between size and speed***, with smaller and faster memory closer to the CPU shown in hierarchy below:

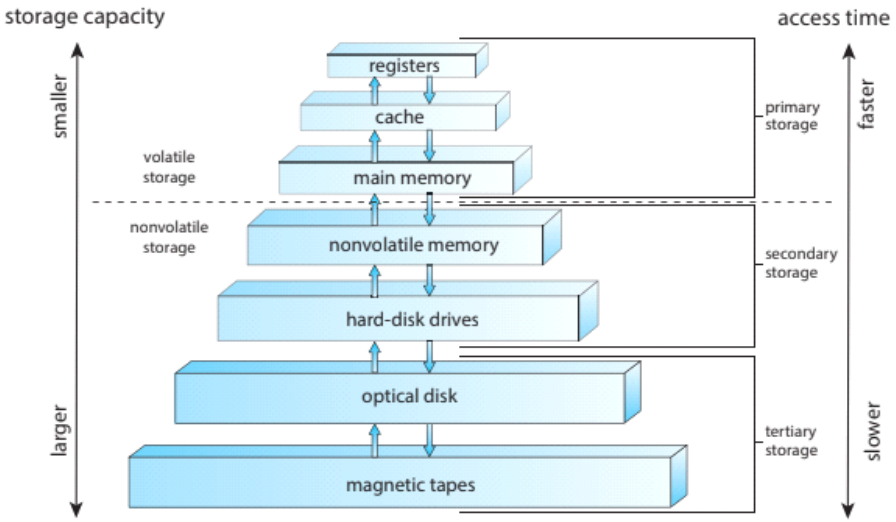


Figure 1.6 Storage-device hierarchy.

I/O Structure

A large portion of operating system code is dedicated to managing I/O, both because of its importance to the reliability and performance of a system and because of the varying nature of the devices.

To start an I/O operation, the device driver loads the appropriate registers within the device controller. The device controller , in turn, examines the contents of these registers to determine what action to take. The controller starts the transfer of data from the device to its local buffer. Once the transfer of data is complete, the device controller informs the device driver via an interrupt that it has finished its operation. The device driver then returns control to the operating system.

This form of interrupt-driven I/O is fine for moving small amounts of data but can produce high overhead when used for *bulk data movement*. To solve this problem, ***Direct Memory Access (DMA)*** is used. After setting up buffers, pointers, and counters for the I/O device, the device controller transfers an entire block of data directly to or from its own buffer storage to memory, with no intervention by the CPU. Only one interrupt is generated per block, to tell the device driver that the operation has completed. While the device controller is performing these operations, the CPU is available to accomplish other works. Hence it increases the CPU throughput but computer architecture (system buses, etc.) becomes more complex.

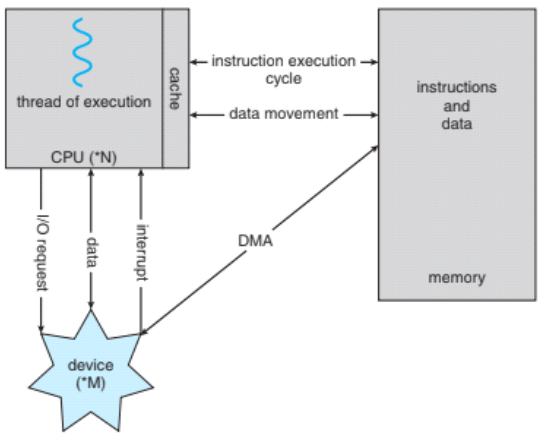


Figure 1.7 How a modern computer system works.

RAM vs ROM

Various Memory Devices

Storage	Speed	Cost	Capacity	Permanent
Register	Fastest	Highest	Lowest	No
Hard Disk	Moderate	Very Low	Very High	Yes
RAM	Very fast	High	Low to Moderate	No
ROM	Very Fast	High	Very Low	Yes
CD-ROM	Moderate	Very Low	High	Yes

Parameters	RAM	ROM
Usage	RAM allows the computer to read data quickly to run applications.	ROM stores all the application which is needed to boot the computer initially. It only allows for reading.
Volatility	RAM is volatile. So, its contents are lost when the device is powered off.	It is non-volatile, i.e., its contents are retained even if the device is powered off
Accessibility	Information stored in the RAM is easily accessed.	The processor can't directly access the information that is stored in the ROM. In order to access ROM information first, the information is transferred into the RAM, and then it can be executed by the processor.
Read/Write	Both R (read) and W (write) operations can be performed over the information which is stored in the RAM.	The ROM memory allows the user to read the information. But, the user can't alter the information.
Storage	RAM is used to store temporary information.	ROM memory is used to store permanent information, which is non-erasable.
Speed	The access speed of RAM is faster.	Its speed is slower in comparison with RAM. Therefore, ROM can't boost up the processor speed.
Cost	The price of RAM is quite high.	The price of ROM is comparatively low.
Chip size	Physical size of RAM chip is bigger than ROM chip.	Physical size of ROM chip is smaller than the RAM chip of same storage capacity.
Preservation of Data	Electricity is needed in RAM to flow and to preserve information	Electricity is not required to flow and preserving information
Structure	The RAM chip is in rectangle form and is inserted over the motherboard of the computer.	Read-only memory (ROM) is a type of storage medium that permanently stores data on personal computers (PCs) and other electronic devices.

RAM (Random Access Memory)

The information stored in RAM can be checked with the help of BIOS.
It is generally known as the main memory, or temporary memory or cache memory or volatile memory of the computer system.
Advantages:

- **Silent**: RAM does not have any moving parts, so its operation is completely silent.
- **Power-efficient:** RAM uses much less power than disk drives.
- **Saves your system battery:** It helps you to reduce carbon emissions and extend your battery life.
- **Fast**: It is much faster to read and write.

Disadvantages:

Volatile: The information stored in this type of memory is lost when the power supply to the PC or laptop is switched off.

Types of RAM

- **DRAM** -Dynamic RAM must be continuously refreshed, or otherwise, all contents are lost.
- **SRAM** - Static RAM is faster, needs less power but is more expensive. However, it does need to be refreshed like DRAM.
- **SDRAM** (Synchronous Dynamic RAM) - This type of RAM can run at very high clock speeds.

- **DDR** - Double Data Rate provide synchronous Random Access Memory

SRAM	DRAM
SRAM has lower access time, which is faster compared to DRAM.	DRAM has a higher access time. It is slower than SRAM.
SRAM is costlier than DRAM.	DRAM cost is lesser compared to SRAM.
SRAM needs a constant power supply, which means it consumes more power.	DRAM requires reduced power consumption as the information stored in the capacitor.
SRAM offers low packaging density.	DRAM offers a high packaging density.
Uses transistors and latches.	Uses capacitors and very few transistors.
L2 and L3 CPU cache units are some general application of an SRAM.	The DRAM is mostly found as the main memory in computers.
The storage capacity of SRAM is 1MB to 16MB.	The storage capacity of DRAM is 1 GB to 16GB.
SRAM is in the form of on-chip memory.	DRAM has the characteristics of off-chip memory.
The SRAM is widely used on the processor or lodged between the main memory and processor of your computer.	The DRAM is placed on the motherboard.
SRAM is of a smaller size.	DRAM is available in larger storage capacity.
This type of RAM works on the principle of changing the direction of current through switches.	This type of RAM works on holding the charges.

Advantage of SRAM

Here, are pros/benefits of using SRAM:

- SRAM performance is better than DRAM in terms of speed. It means it is faster in operation.
- SRAM used to create a speed-sensitive cache.
- It has medium power consumption.

Advantages of DRAM

Here, are pros/benefits of DRAM:

- Cheaper compared to SRAM.
- It has a higher storage capacity. Hence it is used to create a larger RAM space system.
- Offers simple structure
- It doesn't require to refresh the memory contents
- You don't need to refresh the memory contents and its access time is faster
- logic or circuitry is needed, so the memory module itself is simpler

Disadvantage of SRAM

Here, are drawbacks/cons of using DRAM

- It is comparatively slower than SRAM, so it takes more time for accessing data or information.
- You will losses data when power is OFF.
- It has higher power consumption compared to SRAM.

Disadvantages of DRAM

Here, are cons/drawbacks of using DRAM:

- It is costlier compared to DRAM.
- It is volatile, so you will lose the data when memory is not powered.
- DRAM does not offer to refresh programs.
- It has a low storage capacity.
- SRAM offers a more complex design.
- Reduces the memory density

ROM (Read Only Memory)

The computer manufacturer decides the information of ROM, and it is permanently stored at the time of manufacturing, which

cannot be overwritten by the user. However, once it is written, you can read it any number of times.

Advantages:

- Non-Volatile: It is a permanent type of memory. Its content are not lost when the power supply is switched off, which is its biggest advantage.
- It can't be changed accidentally
- It is *cheaper* than RAMs
- It is static and do not require refreshing

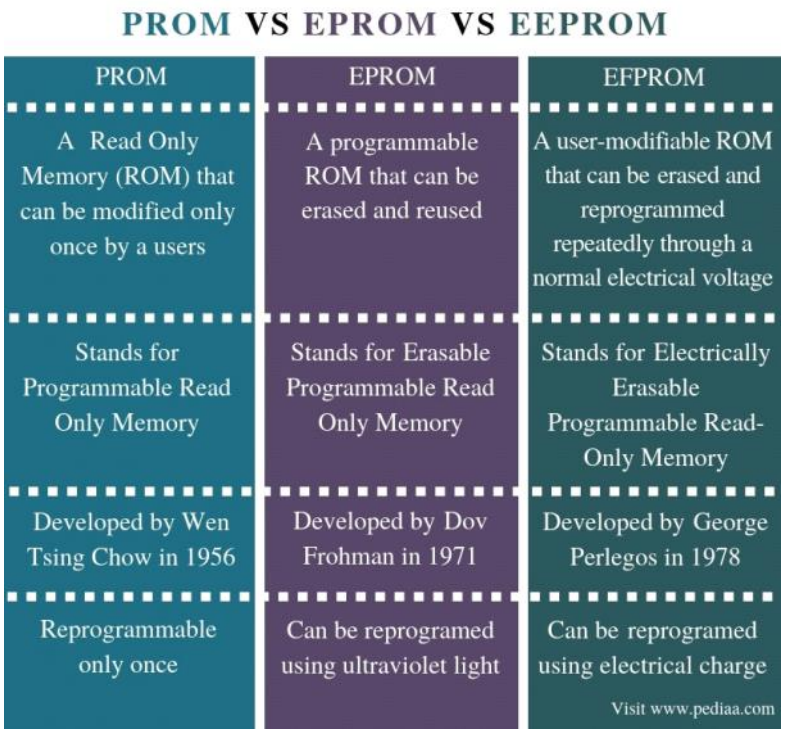
Types of ROM

- **EPROM**: The full form of EPROM is Erasable Programmable Read-only memory. It stores instructions, but you can erase only by exposing the memory to ultraviolet light. It can be reprogrammed. To reprogram it, erase all the previous data.

- **PROM:** The full form of PROM is Programmable Read-Only memory. This type of ROM is written or programmed using a particular device. It can be programmed by user. Once programmed, the data and instructions in it cannot be changed.

- **EEPROM** stands for electrically Erasable Programmable Read-Only Memory. It stores and deletes instructions on a special circuit. The data can be erased by applying electric field, no need of ultra violet light. We can erase only portions of the chip. EEPROM although can change but only frequently. It is low speed and it contains mostly static programs and data that aren't frequently used. For example, the iPhone uses EEPROM to store serial numbers and hardware information about the device.

- **Mask ROM** is a full form of MROM is a type of read-only memory (ROM) whose contents can be programmed only by an integrated circuit manufacturer.



System Calls

Dual Mode

In order to ensure the proper execution of operating system and protection from a malicious program or errant users, operating system distinguish between the execution of operating-system code and user-defined code.
Some unwanted program behaviors can be

a. Running user program can accidentaly wipe out the operating system by overwriting it with user data
b. Multiple processes can write in the same system at the same time, with disastrous results.

Computer system provide hardware support that allows differentiation among the two modes of execution: **User Mode (Unpriviliged Mode)** and **Kernel Mode (Supervisor Mode or System Mode or Privileged Mode)**.

A bit, called the ***mode bit***, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user.

Life Cycle of Instruction Execution

- When the computer system is executing on behalf of a user application, the system is in user mode. However, when a user application requests a service from the operating system (via a system call), the system must transition from user to kernel mode to fulfill the request.

- At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode. Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode (i.e. changes the state of the mode bit to 0). Thus, whenever the operating system gains control of the computer, it is in kernel mode. The system always switches to user mode (by setting the mode bit to 1) before passing control to a user program.

- Some machine instructions, if used improperly can cause harm to the operating system, are designated as ***privileged instructions*** by the hardware. The hardware allows privileged instructions to be executed only in kernel mode. If an attempt is made to execute a privileged instruction in user mode, the hardware does not execute the instruction but rather treats it as illegal and traps it to the operating system.

- Examples of privileged instructions):
  a. Instruction to switch to kernel mode
  b. I/O Control
  c. Timer management
  d. Interrupt management.

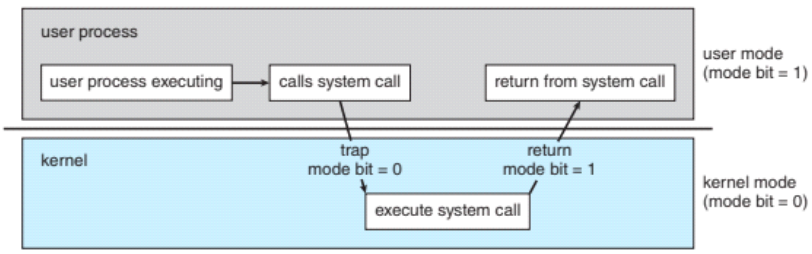


Figure 1.13 Transition from user to kernel mode.

Note:

- Operating system cannot allow a user program to get stuck in an infinite loop or to fail to call system services and never return control to the operating system. To accomplish this goal, we can use a ***timer***.
- A timer can be set to interrupt the computer after a specified period. The period may be fixed or variable. A variable timer is generally implemented by a fixed-rate clock and a counter. The operating system sets the counter. Every time the clock ticks, the counter is decremented. When the counter reaches 0, an interrupt occurs.
- Before turning over control to the user, the operating system ensures that the timer is set to interrupt. If the timer interrupts, control transfers automatically to the operating system, which may treat the interrupt as a fatal error or may give the program more time. Instructions that modify the content of the timer are privileged.

System Calls

- A system call is a mechanism using which a user program can request a service from the kernel for which it does not have the permission to perform.
- User programs typically do not have permission to perform operations like accessing I/O devices and communicating other programs. An user program invokes system calls when it requires such services.
- System calls provide an interface to the services made available by an operating system. These calls are generally available as functions written in C and C++, although certain low-level tasks may have to be written using assembly-language instructions.
- System calls occur in different ways, depending on the system. Usually, more information is required than simply the identity of the desired system call. The exact type and amount of information vary according to the particular operating system and call.

Working of System Call

When a system call is executed, it is typically treated by the hardware as a *software interrupt*. Control passes through the interrupt vector to a service routine in the operating system, and the mode bit is set to kernel mode. The system-call service routine is a part of the operating system. The kernel examines the interrupting instruction to determine what system call has occurred, a parameter indicates what type of service the user program is requesting. Additional information needed for the request may be passed in registers, on the stack, or in memory. The kernel verifies that the parameters are correct and legal, executes the request, and returns control to the instruction following the system call.

Example of System Calls: In order to copy contents from one file to another either the command *"cp in.txt out.txt"* in command interpreter can be used, or following GUI based approach involves a series of system calls:

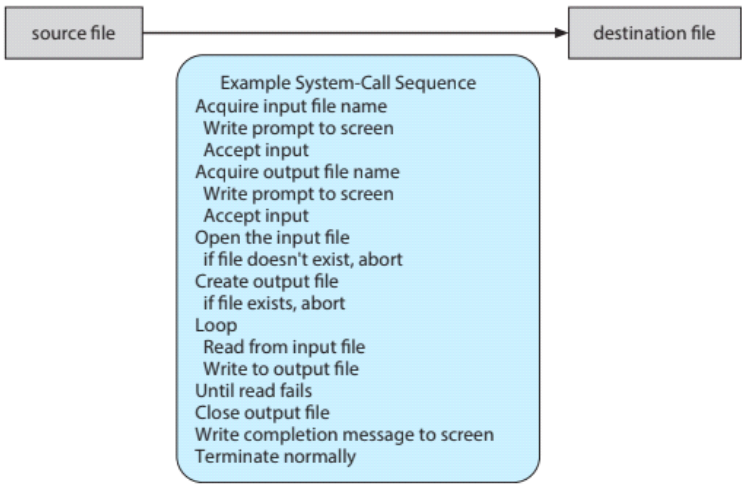


Figure 2.5 Example of how system calls are used.

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. A programmer accesses an API via a library of code provided by the operating system.

An API facilitates the programmers with an simpler and efficient way to develop their software programs. The functions that make up an API typically invoke the actual system calls on behalf of the application programmer.

Eg) Windows API , POSIX API, Java API (for programs on JVM) etc.

Q) Why application programmer prefer designing programs using API calls instead of actual system calls ?

1. *Program Portability*: A program designed using an API can compile and run on any system that supports the same API.
2. Actual system calls can often be more detailed and difficult to work with than the API available to an application programmer.
3. There exists a strong correlation between a function in the API and its associated system call within the kernel.

Run-Time Environment (RTE)

- RTE is the full suite of software needed to execute applications written in a given programming language, including its compilers or interpreters as well as other software, such as libraries and loaders.
- It 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 then invokes the intended system call in the operating-system kernel and returns the status of the system call.
- Most of the details of the operating-system interface (how system call is executed) are hidden from the programmer by the API and are managed by the RTE.
- Example) *JVM(Java Virtual Machine)* acts as a run-time engine to run Java applications. JVM is the one that actually calls the main method present in a java code. JVM is a part of *JRE(Java Runtime Environment)*.

Fig 2.6 depicts the *relationship among an API, the system-call interface, and the operating system*, using open() system call.

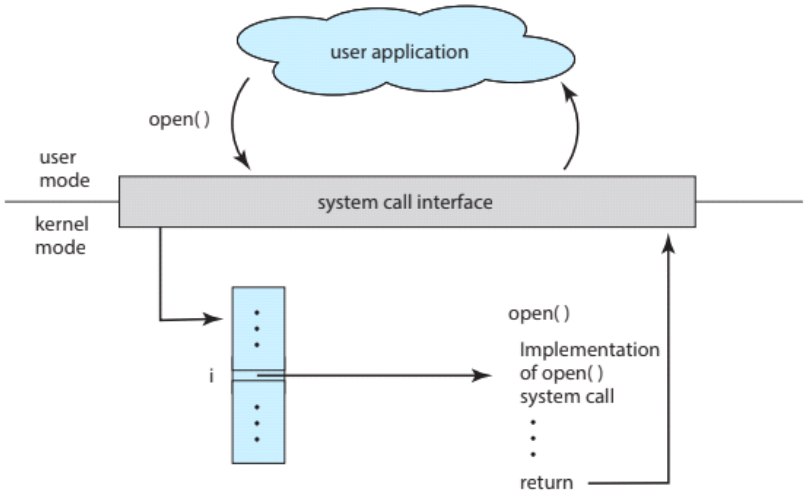


Figure 2.6 The handling of a user application invoking the `open()` system call.

Passing Parameters to Operating System

Three general methods are used to pass parameters to the operating system:

1. Pass the parameters in registers when the parameters are less than or equal to the registers.
2. In some cases, there may be more parameters than registers. In these cases, the parameters are generally stored in a block or table in memory and the address of the block is passed as a parameter in a register.
3. Parameters also can be placed, or pushed, onto a stack by the program and popped off the stack by the operating system.

Some operating systems prefer the block or stack method because those approaches do not limit the number or length of parameters being passed.

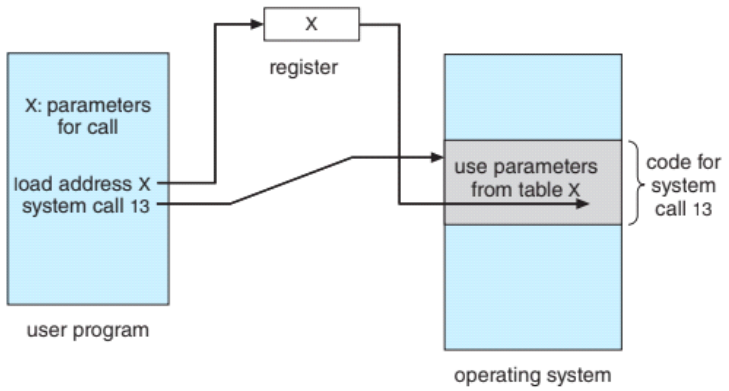


Figure 2.7 Passing of parameters as a table.

Types of System Calls

1. **Process control**

- create process, terminate process:
- load, execute

- get process attributes, set process attributes
- wait event, signal event
- allocate and free memory

2. **File management**
- create file (*create()*), delete file (*delete()*)
  They require the name of the file and some of the file's attributes
- open(), close()
- read(), write(), reposition()
- get_file_attributes() , set_file_attributes()
  File attributes include the file name, file type, protection codes, accounting information, and so on.

3. **Device management**
- request() device, release() device
- read(), write(), reposition()
- get_device_attributes() , set_device_attributes()
- logically attach or detach devices

4. **Information maintenance**
- get time or date, set time or date
- get system data, set system data
  (System data can be operating system version, amount of free space on memory, etc.)
- get process, file, or device attributes
- set process, file, or device attributes

5. **Communications**
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

6. **Protection**
- get file permissions
- set file permissions

System Structure

A ***kernel*** is that part of the operating system which interacts directly with the hardware and performs the most crucial tasks. It handlesmanagement/allocation of hardware & memory resources to system & application softwares. It is also responsible for CPU Scheduling. Hence it is the most important part of the operating system. It resides in the main memory all the time (after booting).

Monolithic Structure

It is the simplest structure for organizing the operating system, which is no structure at all i.e. compile all of the functionality of the kernel into a single, static binary obj file that runs in a single address space. Thus an enormous amount of functionality to be combined into one single address space.

Since all functionalities reside in the kernel itself, majority instructions will be privileged instructions which can be executed while in the kernel mode only.

It consists of two separable parts: the kernel and the system programs. The kernel is further separated into a series of interfaces and device drivers. 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.

Example 1) UNIX Operating System

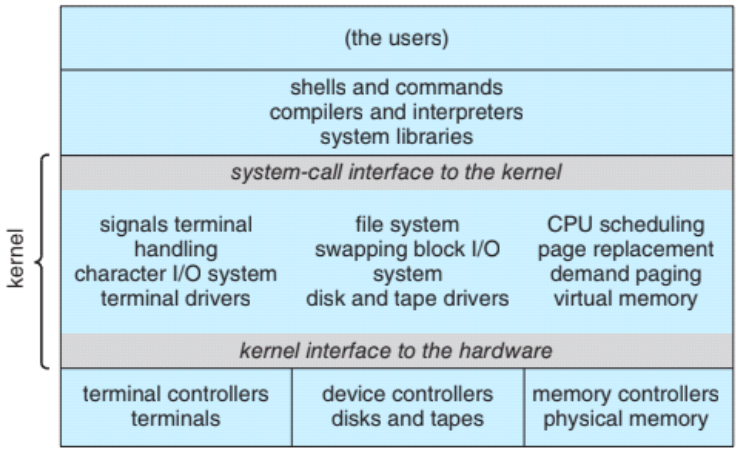


Figure 2.12 Traditional UNIX system structure.

Example 2) Linux Operating System

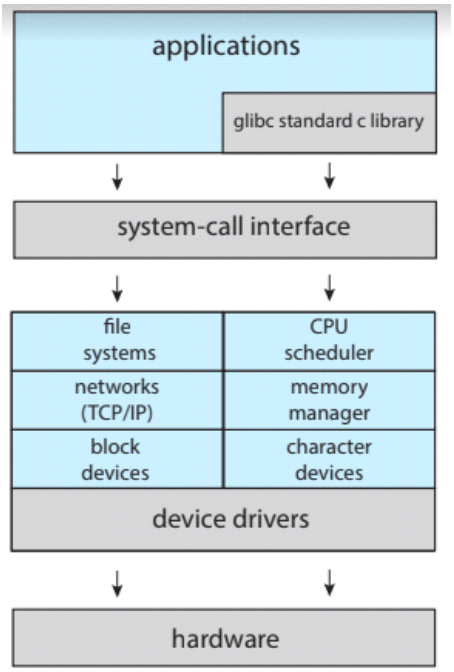


Figure 2.13 Linux system structure.

Advantages

Distinct Performance (Speed & Efficiency): There is very little overhead in the system-call interface, and communication within the kernel is fast.

Disadvantages

1. **Non-Extensiblity**: Since all the functionalities resides in the kernel itself, if any new functionality need to be added or existing functionality need to be modified, the entire kernel have to be re-compiled again.

2. **Less Portability**: Since kernel is enormous and it is architecture-specific, hence it will be very heavy job to make the same operating system be compatible on some other processor or hardware.

3. ***Less Security***: Since all functionalities reside in kernel itself, thus major instructions are all privileged which need to be executed in kernel mode. If any functionality malfunctions, whole operating system can stop executing since it is in kernel mode.

Layered Structure

- 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.
- 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 consists of data structures and a set of functions that can be invoked by layers higher than the current level ie. Each layer can invoke the layer below it.
- 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.
- 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.
- Disadvantages: There are challenges of appropriately defining the functionality of each layer. In addition, the overall performance of such systems is poor due to the overhead of requiring a user program to traverse through multiple layers to obtain an operating-system service.

Example) THE Operating System

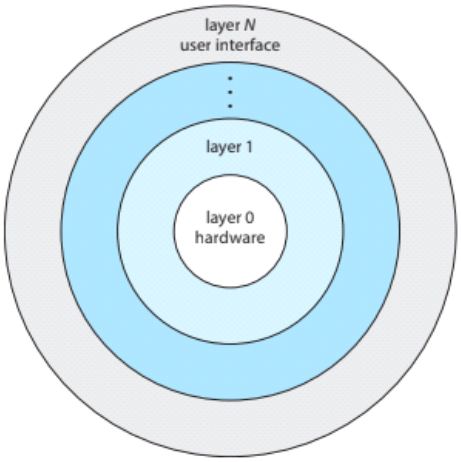


Figure 2.14 A layered operating system.

Microkernel Structure

This method structures the operating system by removing all nonessential components from the kernel and implementing them as user-level programs (which can be executed in user mod) that reside in separate address spaces.

Micro-kernel is much smaller(micro) in size than traditional kernel (in monolithic structure) and only core essential operating system functionalities are kept in the kernel.

The main function of the microkernel is to provide communication between the client program and the various services that are also running in user space. (Communication is provided through message passing). Microkernels provide minimal process and memory management, in addition to the communication facility.

Example) Windows 10, Mac OS

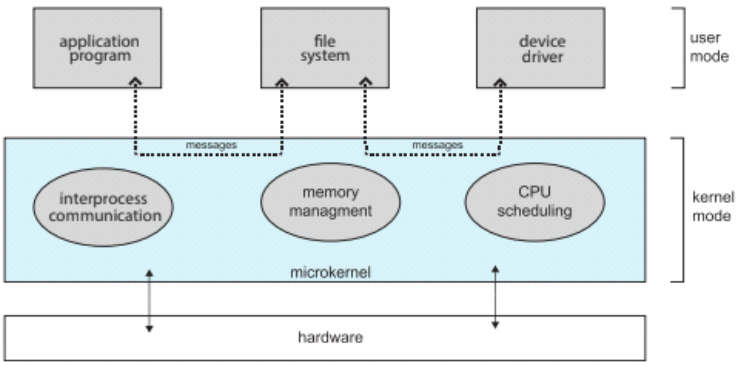


Figure 2.15 Architecture of a typical microkernel.

Advantages:

1. ***Extensibility:*** If new functionality need to added, then it can be added easily as microservice as it need not to be in the kernel part of the operating system. Even if it is core functionality which must be inside kernel, then recompiling micro-kernel is less costly than recompiling the monolithic kernel.

2. ***Portability:*** Since only kernel part is architecture specific, if same operating system need to be executed on some other processor, major part is architecture-independent and hence only microkernel need to be modified according to the available hardware/software resources.

3. ***Reliability & Security:*** Since major functionalities are outside kernel and thus in user mode which is non-privileged, thus operating system or the microkernel will work properly even if the microservice malfunctions.

Disadvantages: Low Performance

Q) *Why* windows is more monolithic and not microkernel?

R) Unfortunately, the performance of microkernels can suffer due to ***increased system-function overhead***. When two user-level services must communicate, messages must be copied between the services, which reside in separate address spaces. The operating system may have to switch from one process to the next to exchange the messages. The overhead involved in copying messages and switching between processes has been the largest impediment to the growth of microkernel-based operating systems.

Basis for Comparison	Microkernel	Monolithic Kernel
Size	Microkernel is smaller in size	It is larger than microkernel
Execution	Slow Execution	Fast Execution
Extendible	It is easily extendible	It is hard to extend
Security	If a service crashes, it does effects on working on the microkernel	If a service crashes, the whole system crashes in monolithic kernel.
Code	To write a microkernel more code is required	To write a monolithic kernel less code is required
Example	QNX, Symbian, L4Linux etc.	Linux,BSDs(FreeBSD,OpenBSD,NetBSD)etc.

System Boot

The process of starting a computer by loading the kernel is known as booting the system. On most systems, the boot process proceeds as follows:

1. A small piece of code known as the ***bootstrap program*** or ***boot loader*** locates the kernel.
2. The kernel is loaded into memory and started.
3. The kernel initializes hardware.
4. The root file system is mounted.

Detailed Process

- When the computer is first powered on, a small boot loader located in non-volatile firmware {or Read Only Memory (ROM)} known as ***BIOS (Basic Input/Output System)*** is run.

- This initial boot loader usually does nothing more than load a second boot loader, which is located at a fixed disk location called the ***boot block***. The program stored in the boot block loads the entire operating system into memory and begin its execution. It is simple code and knows only the address on disk and the length of the remainder of the bootstrap program.

- Many recent computer systems have replaced the BIOS-based boot process with ***UEFI (Unified Extensible Firmware Interface)***. UEFI has several advantages over BIOS, including better support for 64-bit systems and larger disks. The greatest advantage is that UEFI is a single, complete boot manager and therefore is faster than the multistage BIOS boot process.

- Whether booting from BIOS or UEFI, the bootstrap program can perform a variety of tasks. In addition to loading the file containing the kernel program into memory, it also runs ***diagnostics*** to determine the state of the machine — for example, inspecting memory and the CPU and discovering devices. If the diagnostics pass, the program can continue with the booting steps.

- These initial set of diagnostic tests are known as ***POST (Power-On Self Test)*** that run to determine if the attachments (keyboard, disk drives, memory etc.) works properly. It can detect some errors with keyboard, video cards, memory, motherboard, processors etc. Most BIOS chips use different beep codes to indicate the POST status or the problems find with devices.

- The bootstrap 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 and *mounts the root file system*. It is only at this point is the system said to be running.

- It is the last step in boot sequence. Once the kernel is loaded, find ***init*** and executes it. First thing that init does is reading initialization file i.e. /etc/inittab that helps init to read an initial configuration script for the environment which check the file system, sets the path etc.

Note: ***Booting Process in Windows***:
Windows allows a drive to be divided into partitions, and one partition identified as the ***boot partition*** contains the operating system and device drivers. The Windows system places its boot code in the first logical block on the hard disk or first page of the NVM device, which it terms the ***master boot record (MBR)***. Booting begins by running code that is resident in the system's firmware. This code directs the system to read the boot code from the MBR, understanding just enough about the storage controller and storage device to load a sector from it. In addition to containing boot code, the MBR contains a table listing the partitions for the drive and a flag indicating which partition the system is to be booted from. Once the system identifies the boot partition, it reads the first sector/page from that partition (called the ***boot sector***), which directs it to the kernel. It then continues with the remainder of the boot process, which includes loading the various subsystems and system services.

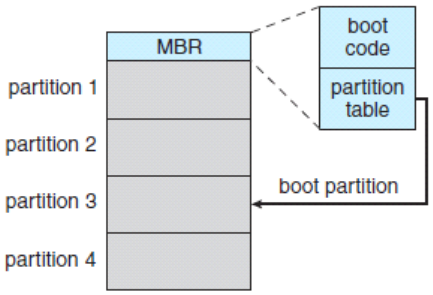


Figure 11.10 Booting from a storage device in Windows.

Advantages of UEFI over BIOS

- **Breaking Out Of Size Limitations :** The UEFI firmware can boot from drives of 2.2 TB or larger with the theoretical upper limit being 9.4 zettabytes, which is roughly 3 times the size of the total information present on the Internet. This is due to the fact that GPT uses 64-bit entries in it's table, thereby dramatically expanding the possible boot-device size.

- **Speed and performance :** UEFI can run in 32-bit or 64-bit mode and has more addressable address space than BIOS, which means your boot process is faster.

- **More User-Friendly Interface :** Since UEFI can run in 32-bit and 64-bit mode, it provides better UI configuration that has better graphics and also supports mouse cursor.

- **Security**: UEFI also provides the feature of Secure Boot. It allows only authentic drivers and services to load at boot time, to make sure that no malware can be loaded at computer startup. It also requires drivers and the Kernel to have digital signature, which makes it an effective tool in countering piracy and boot-sector malware.

Process

A process is a program under execution. The status of the current activity of a process is represented by the value of the *program counter* and the contents of the processor's registers.

Memory Layout of a Process

1. ***Text** Section*: The executable code in form of machine language(low-level language). It also contains constants, macros and it is read-only segment to prevent accidentally modification of an instruction. It is also sharable so that the so that another process can use this whenever it is required.

2. ***Data** Section*: It contains Static Variables & Global Variables (whose lifetime is until process completes execution) that are initialized by the programmer prior to the execution of a program. This segment is not read-only, as the value of the variables can be changed at the runtime.

3. ***Heap** Section*: It contains memory that is dynamically allocated during program at run time to variables whose size cannot be statically determined by the compiler before program execution. It is managed via system calls to malloc, calloc, free, delete etc.

4. ***Stack** Section*: Temporary data storage during recursive calls to functions such as function parameters, return addresses, and local variables. A stack pointer register keeps the tracks of the top of the stack i.e., how much of the stack area using by the current process, and it is modified each time a value is "pushed" onto the stack. If the stack pointer meets the heap pointer the available free memory is depleted.

Note: Sizes of Text & Data Section are fixed and do not change at run-time. However, the stack and heap sections can shrink and grow dynamically during program execution. Although the stack and heap sections grow toward one another, the operating system must ensure they do not overlap one another.

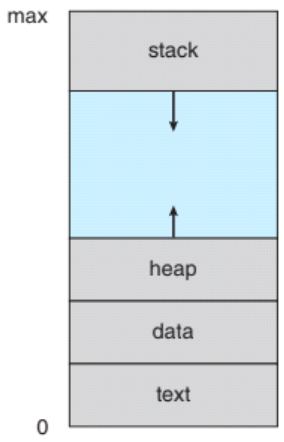


Figure 3.1 Layout of a process in memory.

Program vs Process

- A program by itself is not a process.
- **A program is a *passive entity***, as it is an executable file containing a list of instructions stored on secondary memory (hard disk).
- In contrast, **a process is an *active entity***, as it is a program code that has been loaded into a computer's memory so that it can be executed by CPU) with a program counter specifying the next instruction to execute and a set of associated resources.
- A program becomes a process when an executable file is loaded into memory, thus *Process is a program under execution.*
- ***One program can have several processes***. The two processes associated with the same program are considered two separate execution sequences as only text section(where program's executable code resides) is shared by the two processes and rest stack, heap & data sections are different for them.

Difference between Program and Process :

SR.NO.	PROGRAM	PROCESS
1.	Program contains a set of instructions designed to complete a specific task.	Process is an instance of an executing program.
2.	Program is a passive entity as it resides in the secondary memory.	Process is a active entity as it is created during execution and loaded into the main memory.
3.	Program exists at a single place and continues to exist until it is deleted.	Process exists for a limited span of time as it gets terminated after the completion of task.
4.	Program is a static entity.	Process is a dynamic entity.
5.	Program does not have any resource requirement, it only requires memory space for storing the instructions.	Process has a high resource requirement, it needs resources like CPU, memory address, I/O during its lifetime.
6.	Program does not have any control block.	Process has its own control block called Process Control Block.

Note: Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line.

Note: A process can itself be an execution environment for other code. Eg) An executable Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code and takes actions (via native machine instructions) on behalf of that code.

Process States

When a process executes, it undergoes transitions from various states during it's lifetime(from creation of process to its deletion) depending on the nature of the process.

1. ***New***: In this step, the process is about to be created but not yet created, it is the program which is present in secondary memory that will be picked up by OS to create the process.
2. ***Ready:*** After the creation of a process, the process enters the ready state i.e. the process is loaded into the main memory. The process here is ready to run and is waiting to get the CPU time for its execution. Processes that are ready for execution by the CPU are maintained in a queue for ready processes.
3. ***Run***: The process is chosen by CPU for execution and the instructions within the process are executed by any one of the available CPU cores.
4. ***Blocked or wait:*** Whenever the process requests access to I/O or needs input from the user or needs access to a critical region(the lock for which is already acquired) it enters the blocked or wait state. The process continues to wait in the main memory and does not require CPU. Once the I/O operation is completed the process goes to the ready state.
5. ***Terminated or completed***: Process is killed as well as PCB is deleted.
6. ***Suspend ready***: Process that was initially in the ready state but were swapped out of main memory and placed onto external storage by scheduler are said to be in suspend ready state. The process will transition back to ready state whenever the process is again brought onto the main memory.
7. ***Suspend wait or suspend blocked***: Similar to suspend ready but uses the process which was performing I/O operation and lack of main memory caused them to move to secondary memory. When work is finished it may go to suspend ready.

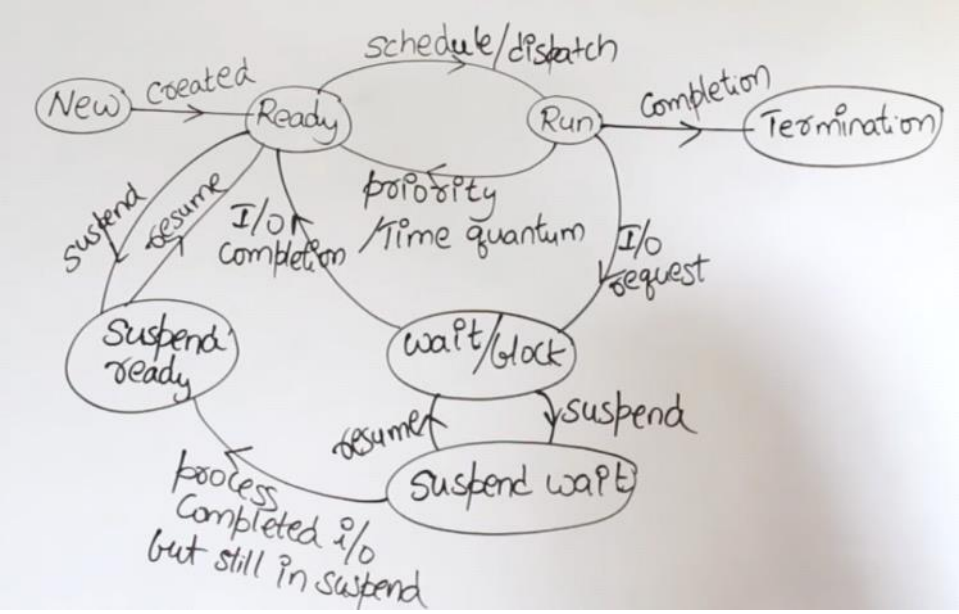
Q) Consider a system with 'N' CPU Processors/Cores and 'M' processes.

R)

State	Min	Max
Ready	0	M
Running	0	N

Block	0	M

Process State Transition Diagram



Types of Processes

1. ***Event Specific Process Categorization: CPU and IO Bound Processes:***
   If the process is intensive in terms of CPU computations/operations then it is called *CPU bound process*. It requires most of the time on CPU. Similarly, If the process is intensive in terms of I/O operations then it is called *IO bound process*. It requires most of the time on I/O devices or peripherals. Best performance system is which have a combination of CPU bound and I/O bound processes.

2. ***Nature Based Categorization: Independent and Cooperative Processes***:

- A process which does not need any other external factor to trigger is an *independent process*. They cannot effect or be affected by other processes in the system.
- Whereas a process which works on occurance of any event and the outcome effects any part of the rest of the system is a *cooperating process*. They can effect or get affected by other processes executing in the system. Eg) Any process which shares data with other process is cooperative process.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing.
- However, Concurrent access to shared data may result in data inconsistency, thus process synchronization mechanisms need to be implemented to ensure orderly execution of such processes so that data consistency is maintained.

Note: ***Degree of multiprogramming:*** The number of processes that can reside in the ready state at maximum decides the degree of multiprogramming, e.g., if the degree of programming = 100, this means 100 processes can reside in the ready state at maximum.

Note: There are two types of multiprogramming:

- Pre-emption – Process is forcefully removed from CPU. Pre-emption is also called as time sharing or multitasking.
- Non pre-emption – Processes are not removed until they complete the execution.

Process Control Block (PCB) *or Task Control Block*

- A process control block (PCB) contains information about the process, i.e. registers, quantum, priority, etc. and is used to track the process's execution status.
- Each block of memory contains information about the process state, program counter, stack pointer, status of opened files, scheduling algorithms, etc. All these information is required and must be saved when the process is switched from one state to another (context switch).
- When the process makes a transition from one state to another, the operating system updates its information in the

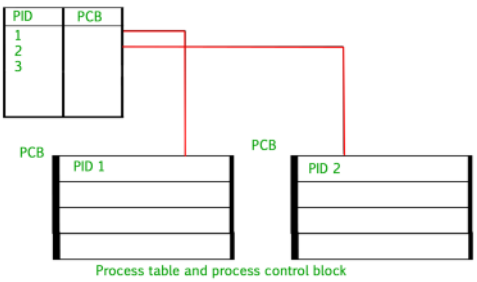
process's PCB.

Attributes of PCB:

1. ***Process ID:*** Every process is assigned with a unique id known as process ID or PID which stores the process identifier.
2. ***Process state:*** The state may be new, ready, running, waiting, halted, and so on.
3. ***Program counter***: The counter indicates the address of the next instruction to be executed for this process (after resuming in context switch).
4. ***CPU registers:*** They include accumulators, index registers, stack pointers, and general-purpose registers and any condition-code information. The registers vary in number and type, depending on the computer architecture.
5. ***CPU-scheduling information:*** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
6. ***Memory-management information:*** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system.
7. ***Accounting information:*** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
8. ***I/O status information:*** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

Process Control Block

Note: The ***process table*** is an array of pointers pointing to memory address of PCB's of processes which are currently in the system.

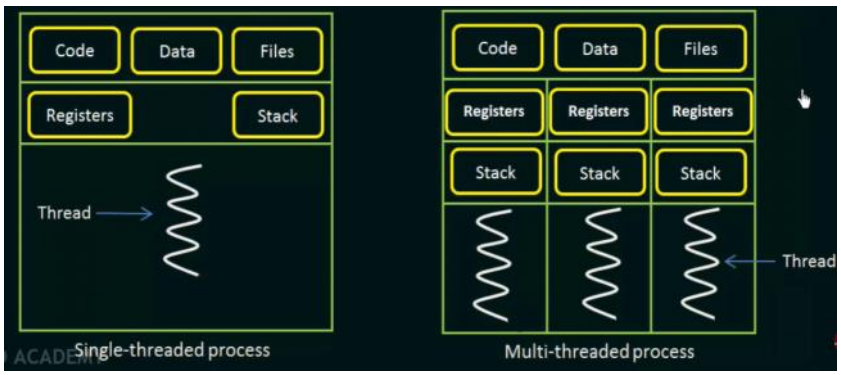


Process table and process control block

Threads

A ***thread*** is a basic unit of CPU utilization. It is only a segment of a process. It comprises of *a thread ID, a program counter, a register set and a stack.*

Multiple threads of the same process shares the code section, data section and operating system resources such as open files and signals.



Multi threading is an execution model that allows a single process to have multiple threads running concurrently within the context of that process.

Multi threading is the ability of a process to manage its use by more than one user at a time and to manage multiple requests by the same user without making multiple processes from same program.

Examples of Multithreaded System

- *Web Browsers* - A web browser can download any number of files and web pages (multiple tabs) at the same time and still lets you continue browsing. If a particular web page cannot be downloaded, that is not going to stop the web browser from downloading other web pages.

- *Web Servers* - A threaded web server handles each request with a new thread. There is a thread pool and every time a new request comes in, it is assigned to a thread from the thread pool.

- *Computer Games* - You have various objects like cars, humans, birds which are implemented as separate threads. Also playing the background music at the same time as playing the game is an example of multithreading.

- *Text Editors* - When you are typing in an editor, spell-checking, formatting of text and saving the text are done concurrently by multiple threads. The same applies for Word processors also.

- *IDE* - IDEs like Android Studio run multiple threads at the same time. You can open multiple programs at the same time. It also gives suggestions on the completion of a command which is a separate thread.

Advantages of Multi threading

- ***Increased responsiveness***: Since there are multiple threads in a program, so if one thread is taking too long to execute or if it gets blocked, the rest of the threads keep executing without any problem. Thus the whole program remains responsive to the user by means of remaining threads.

- ***Economic: (Less costly both wrt time & space)***: Creating brand new processes and allocating resources is a time consuming task, but since threads ***share resources***(data & text section) of the parent process, creating threads and switching between them is comparatively easy. Hence multi threading is the need of modern Operating Systems.

- ***Scalability:*** In multi-processing operating system, threads may run in parallel on different processors. Thus it increases ***concurrency*** of Multi-core CPU system.

Process vs Threads

S.NO	PROCESS	THREAD
1.	Process means any program in execution.	Thread means segment of a process.
2.	Process takes more time in creation as well as termination.	Thread takes less time in creation as well as termination.
3.	It also takes more time for context switching.	It takes less time for context switching.
4.	Process is less efficient in term of communication.	Thread is more efficient in term of communication.
5.	Process consume more resources.	Thread consume less resources.
6.	Process is isolated means it does not share memory with any other process.	Threads share memory , thus threads do not isolate.
7.	Process is called heavy weight process.	Thread is called light weight process.
8.	Process switching uses interface in operating system. It involves system calls.	Thread switching does not require to call a operating system and cause an interrupt to the kernel. It does not involve system calls. It is at user level.
9.	If one server process is blocked no other server process can execute until the first process unblocked.	Second thread in the same task could run, while one server thread is blocked.
10.	Process has its own Process Control Block, Stack and Address Space.	Thread has Parents' PCB, its own Thread Control Block and Stack and common Address space.

User Level Thread vs Kernel Level Thread

USER LEVEL THREAD	KERNEL LEVEL THREAD
User thread are implemented by users.	kernel threads are implemented by OS.
OS doesn't recognize user level threads.	Kernel threads are recognized by OS.
Implementation of User threads is easy.	Implementation of Kernel thread is complicated.
Context switch time is less.	Context switch time is more.
Context switch requires no hardware support.	Hardware support is needed.
If one user level thread perform blocking operation then entire process will be blocked.	If one kernel thread perform blocking operation then another thread can continue execution.
User level threads are designed as dependent threads.	Kernel level threads are designed as independent threads.
Example : Java thread, POSIX threads.	Example : Window Solaris

Multi-Threading Models

Since user threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system, there must exist relationship between user level threads and kernel level threads:

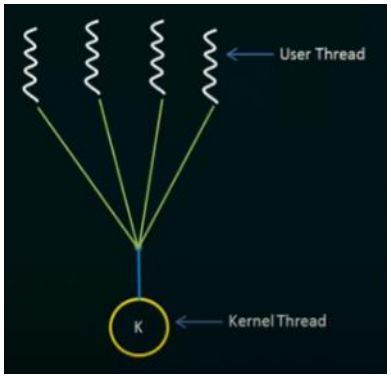
1. ***Many-to-One model***

   Advantages

   It maps many user-level threads to one kernel-level thread. Thread management is done by thread library in user space, so its efficient.

Disadvantages

The entire process will block if a thread makes a blocking system call. Because only one thread can access the kernel at a time, multiple threads are unable to run concurrently (parallelly) in multi-core system, thus it is not used anymore.



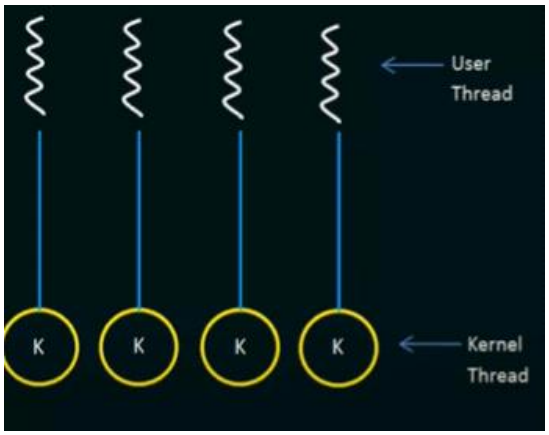
2. ***One-to-One model***

Advantages

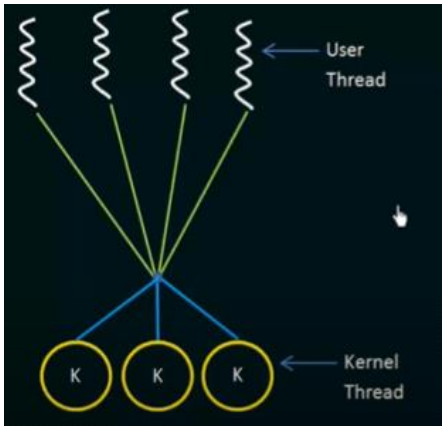
It Maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call It also allows multiple threads to run in parallel on multiprocessors. It is used in *Linux* OS and some versions of Windows OS.

Disadvantages

Creating a user thead requires creating the corresponding kernel thread which can prove to be costly. Because the overhead of creating kernel threads can burden the performance of an application, it restricts the number of threads supported by the system.



3. ***Many-to-Many model***
   - Multiplexes many user-level threads to a smaller or equal number of kernel threads.
   - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
   - Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.
   - 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 system with eight processing cores than a system with four cores.



Note: Many-to-one model was not able to do multi-processing and also entire process would block if any one thread made a blocking system call.
Although one-to-one model solved the multi-processing inability and entire process would run even if one of the threads would make a blocking system call, it was creating kernel threads corresponding to every user thread which is very expensive for system and thus limiting the number of threads.
Now many-to-many model solves all problems in many-to-one model and there is no overheads as in one-to-one model, thus **many-to-many model is** implemented in most of the systems and is called **the best multithreading model**.

Note: Since one core can run only one thread at an instant of time (just like a process), having more number of threads than number of cores in the system will not increase any performance efficiency.
Hence, the optimal number of threads is equal to number of cores in the system. Running CPU bound threads (which are not doing I/O) concurrently requires subsequent number of cores in the system.

Note: Global and Static variables in any programming language (C/C++ or Java) are considered evil or dangerous because:

1. Variables are given *much larger scope and lifetime* than they really need. Many processes/threads can read/write the same global variables as it is shared globally.
2. Use of global variables indicate *poor modularity.* If any one process/thread produces undesired result in these variables, then they become prone to tricky bugs and race conditions.
3. Since they are globally available, they can cause *namespace conflicts* like reassigning values to them.
4. They result in very *tight coupling of code* which makes testing very cumbersome as it is difficult to decouple them. (Coupling refers to the degree of direct knowledge that one element has of another. Tight coupling means changing one part causes change in other part.)

Process Scheduling

Definition

The ***process scheduling*** is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process Scheduling is an OS task that schedules processes of different states like ready, waiting, and running. Process scheduling allows OS to allocate a time interval of CPU execution for each process. Another important reason for using a process scheduling system is that it keeps the CPU busy all the time. This allows you to get the minimum response time for programs.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

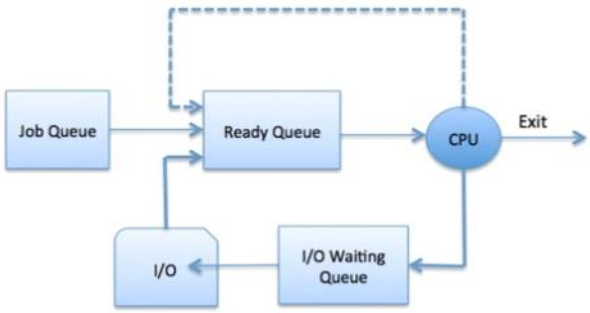
The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

These queues is generally stored as a *linked list*, where a ready-queue header contains pointers to the first PCB in the list, and each PCB includes a pointer field that points to the next PCB in the queue.

- ***Job queue*** – This queue keeps all the processes in the system. A new process which enters the system is always put in this queue.

- ***Ready queue*** – This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A process which is ready to execute (in ready state) is put in this queue.

- ***Device queues*** or ***I/O Wait Queue***– Processes that are waiting for a certain event to occur such as completion of I/O or processes which are blocked due to unavaibility of I/O device are placed in a wait queue.



Schedulers

Schedulers are special system software which handle process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run. Schedulers are of three types:

- ***Long term or Job Scheduler: Performance –*** Makes a decision about how many processes should be made to stay in the *ready state*, this *decides the degree of multiprogramming*. Once a decision is taken it lasts for a long time hence called long term scheduler. It increases efficiency by maintaining a balance between CPU bound and I/O bound processes.

- ***Short term or CPU Scheduler : Context switching time –*** Short term scheduler will *decide which process to be executed next* (from the processes in ready state) and then it will call dispatcher. All CPU Scheduling algorithms are used here only. The scheduler gives the dispatcher an ordered list of processes which the dispatcher moves to the CPU over time.

A ***dispatcher*** is a software that moves process from ready state to run state and vice versa , in other words, context switching. Dispatcher also performs Switching to user mode, Jumping to the proper location in the user program when it is newly loaded or resumed after an interrupt. Thus, the dispatcher is the module that gives a process control over the CPU after it has been selected by the short-term scheduler.
The dispatcher should be as fast as possible, since it is invoked during every context switch. The time dispatcher takes to stop one process and start another running is known as the ***dispatch latency***. Dispatch latency is a pure overhead.

- ***Medium term: Swapping time –*** Suspension & resuming decision is taken by medium term scheduler. Medium term scheduler is used for swapping that is moving the process from main memory to secondary and vice versa.
Swapping is necessary to improve the process mix or because a change in memory requirements has overcommitted available memory, requiring memory to be freed up. It is helpful in maintaining a perfect balance between the I/O bound and the CPU bound. Since it leads to suspension of processes, it reduces the degree of multiprogramming.

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.
3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.

Scheduler vs Dispatcher

PROPERTIES	DISPATCHER	SCHEDULER
Definition:	Dispatcher is a module that gives control of CPU to the process selected by short term scheduler	Scheduler is something which selects a process among various processes
Types:	There are no different types in dispatcher.It is just a code segment.	There are 3 types of scheduler i.e. Long-term, Short-term, Medium-term
Dependency:	Working of dispatcher is dependent on scheduler.Means dispatcher have to wait until scheduler selects a process.	Scheduler works independently. It works immediately when needed
Algorithm:	Dispatcher has no specific algorithm for its implementation	Scheduler works on various algorithm such as FCFS, SJF, RR etc.
Time Taken:	The time taken by dispatcher is called ***dispatch latency***.	Time taken by scheduler is usually negligible. Hence we neglect it.
Functions:	Dispatcher is also responsible for:Context Switching, Switch to user mode, Jumping to proper location when process again restarted	The only work of scheduler is selection of processes.

Context Switching

- A context switch is the mechanism to store and restore the state or context of a CPU in Process Control block so that a process execution can be resumed from the same point at a later time.

- Using this technique, a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

- When the scheduler switches the CPU from executing one process to execute another, the state from the current

running process is stored into the process control block. After this, the state for the process to run next is loaded from its own PCB and used to set the PC, registers, etc. At that point, the second process can start executing.

- Context switches are computationally intensive since register and memory state must be saved and restored. To avoid the amount of context switching time, some hardware systems employ two or more sets of processor registers. When the process is switched, the following information is stored for later use:
  - Program Counter
  - Scheduling information
  - Base and limit register value
  - Currently used register
  - Changed State
  - I/O State information
  - Accounting information

- Context Switching leads to an overhead cost because of TLB flushes, sharing the cache between multiple tasks, running the task scheduler etc. Context switching between two threads of the same process is faster than between two different processes as threads have the same virtual memory maps. Because of this TLB flushing is not required.

- *Voluntary vs Nonvoluntary Context Switch*: A voluntary context switch occurs when a process has given up control of the CPU because it requires a resource that is currently unavailable (such as blocking for I/O.) A nonvoluntary context switch occurs when the CPU has been taken away from a process, such as when its time slice has expired or it has been preempted by a higher-priority process.

Context Switching Triggers

- ***Multitasking***: In a multitasking environment, a process is switched out of the CPU so another process can be run. The state of the old process is saved and the state of the new process is loaded. On a pre-emptive system, processes may be switched out by the scheduler.

- ***Interrupt Handling***: The hardware switches a part of the context when an interrupt occurs. This happens automatically. Only some of the context is changed to minimize the time required to handle the interrupt.

- ***User and Kernel Mode Switching***: A context switch may take place when a transition between the user mode and kernel mode is required in the operating system.

Context Switching Steps

- Save the context of the process that is currently running on the CPU. Update the process control block and other important fields.
- Move the process control block of the above process into the relevant queue such as the ready queue, I/O queue etc.
- Select a new process for execution.
- Update the process control block of the selected process. This includes updating the process state to running.
- Update the memory management data structures as required.
- Restore the context of the process that was previously running when it is loaded again on the processor. This is done by loading the previous values of the process control block and registers.

Operations on Processes

1. ***Process Creation***

- A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a parent process, and 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.

- When a child process is created, it requires certain resources like CPU Time, memory files, I/O devices, initialization data (input) which are obtained directly from the operating system or from a subset of resources available to the parent process. Thus making child process only take some subset of the resources allocated to parent, we can avoid

overloading the system from too many resource allocations to processes.

- When a process creates a new process. two possibilities exist in terms of execution:
  - The parent continues to execute concurrently with its children.
  - 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:
  - The child process is a duplicate of the parent process (it has the same program and data as the parent).
  - The child process has a new program loaded into it.

- ***fork(), wait() & exec():***
  - A new process is created by the fork() system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: the return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.
  - After a fork() system call, one of the two processes typically uses the exec() system call to replace the process's memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec() 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. Because the call to exec() overlays the process's address space with a new program, exec() does not return control unless an error occurs.

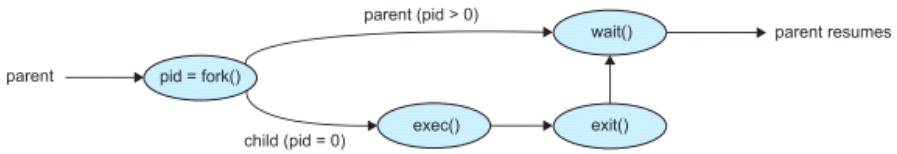


Figure 3.9 Process creation using the `fork()` system call.

Steps in Process Creation

1. When a new process is created, operating system assigns a unique ***Process Identifier (PID)*** to it and inserts a new entry in primary process table.
2. Then the required the required memory space for all the elements of process such as program, data and stack is allocated including space for its *Process Control Block (PCB)*.
3. Next, the various values in PCB are initialized such as,
   a. Process identification part is filled with PID assigned to it in step (1) and also its *parent's PID (PPID)*.
   b. The processor register values are mostly filled with zeroes, except for the stack pointer and program counter. Stack pointer is filled with the address of stack allocated to it in step (2) and program counter is filled with the address of its program entry point.
   c. The process state information would be set to 'New'.
   d. Priority would be lowest by default, but user can specify any priority during creation.

   In the beginning, process is not allocated to any I/O devices or files. The user has to request them or if this is a child process it may inherit some resources from its parent.
4. Then the operating system will link this process to scheduling queue and the process state would be changed from 'New' to 'Ready'. Now process is competing for the CPU.
5. Additionally, operating system will create some other data structures such as log files or accounting files to keep track of processes activity.

6. Process Termination

- A process terminates when it finishes executing its final statement and asks the operating system to delete it (via ***exit()*** system call). At that point, the process may return a status value (typically an integer) to its parent process (via the ***wait()*** system call). All the resources of the process—including physical and virtual memory, open files, and I/O buffers—are deallocated by the operating system.

- Termination can occur in other circumstances like: A process can cause the termination of another process via an appropriate system call (like TerminateProcess() in Windows).
  Note: Usually, such a system call can be invoked only by the parent of the process that is to be terminated. Otherwise, users could arbitrarily kill each other's jobs.

- A parent may terminate the execution of one of its children for a variety of reasons such as these:
  - *Unavailibity of Memory*: The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
  - *Normal Completion*: The task assigned to the child is no longer required.
  - *Parent Process Termination*: The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.
  - *Errors: Protection errors* (when a process is trying to use a resource (e.g. file) to which access is not granted or using it in an inappropriate manner such as writing to a read-only file), *Arithmetic Error* (division-by-zero or storing a number greater than the hardware capacity), Input/Output Failure, etc.

- ***exec() and wait()***:
  - We can terminate a process by using the exit() system call, providing an exit status as a parameter. Under normal termination, exit() will be called either directly (as *exit(1)*) or indirectly.
  - A parent process may wait for the termination of a child process by using the wait() system call. The wait() system call is passed a parameter that allows the parent to obtain the exit status of the child. This system call also returns the process identifier of the terminated child so that the parent can tell which of its children has terminated.
  - When a process terminates, its resources are deallocated by the operating system. However, its entry in the process table must remain there until the parent calls wait(), because the process table contains the process's exit status.

- ***Orphan Process***
  - Some systems do not allow a child to exist if its parent has terminated. Such child processes whose parents are terminated (and wait() system call for the child process is not called before parent process is terminated) are known as ***orphan processes***.
  - In such systems, if a process terminates (either normally or abnormally), then all orphan processes must also be terminated. This phenomenon is known as ***cascading termination*** and is initiated by the operating system.
  - Traditional UNIX systems addressed this scenario by assigning the *init process*(first process which runs on the system, it has PID = 1 and serves as the root of process hierarchy) as the new parent to orphan processes. The init process periodically invokes wait(), thereby allowing the exit status of any orphaned process to be collected and releasing the orphan's process identifier and process-table entry.

- ***Zombie Process or Defunct Process***
  - Zombie Process or Defunct Process are those Process which are in terminated state and has completed their execution by exit() system call but still has an entry in Process Table.
  - Zombie Process is neither completely dead nor completely alive but it has having some state in between.
  - All processes becomes a zombie process for a short duration of time when they terminate. Once the parent calls wait(), the process identifier of the zombie process and its entry in the process table are released.
  - Since, there is an entry for all the process in process table, even for Zombie Processes, it is obvious that size of process table is Finite. So, if zombie process is created in large amount, then Process Table will get filled up and program will stop without completing their task.

  Q) Find out Maximum Number of Zombie Process created so that Program will not stop its execution.
  R) Approach for this problem is to create a zombie process within a loop and count it until the program does not stop the execution.

Example of C Code

```c
#include<stdio.h>
#include<unistd.h>

int main()
{
    int count = 0;
    while (fork() > 0)
    {
        count++;
        printf("%d\t", count);
    }
}
```

After some number of increments in *count*, the program gets stopped as process table gets filled completely and no more zombie processes can be accomodated. Although it comes around *11834*, this number is not fixed as it will depend upon system configuration and strength.

Inter Process Communication

Reasons for providing an environment that allows process cooperation:

1. ***Information Sharing***: Since several applications may be interested in the same piece of information (like copying and pasting), we must provide an environment to allow concurrent access to such information.
2. ***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.
   Note: Such a speedup can be achieved only if the computer has multiple processing cores.
3. ***Modularity:*** We may want to construct the system in a modular fashion like dividing the system functions into separate processes or threads. This increases *convenience* for the user as well.
4. ***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.

- *Cooperative Processes* require an ***Inter Process Communication (IPC)*** mechanism that will alow them to exchange data & information.
- Inter process communication (IPC) is a mechanism which allows processes to communicate with each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.
- Cooperating processes can either directly share a logical address space (that is, both code and data) or be allowed to share data only through shared memory or message passing.

There are two fundamental models of IPC:

1. ***Shared Memory***:

- A region of memory is shared by cooperating processes so that they can exchange information by reading and writing data to the shared region.
- 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.
- 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.

Producer Consumer Problem:

- A producer process produces information that is consumed by a consumer process.
- For eg, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.
- To allow producer and consumer processes to run concurrently, we must have available a *buffer of items* that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by producer and consumer processes.
- A producer can produce one time while the consumer is consuimg another item.
- The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- There are two kinds of buffer:
  - *Unbounded buffer* which places no practical limit on size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
  - *Bounded buffer* which assumes a fixed buffer size. The consumer must wait if buffer is empty and the producer must wait if the buffer is full.

2. ***Message Passing***:

- Communication takes place by means of messages exchanged between the cooperating processes. The messages are sent to the kernel of OS which in turns send messages to other processes in cooperation.
- A message-passing facility provides at least two operations: *send (message) and receive (message)*. Two processes who need to communicate must send messages to and receive messages from each other.
- Messages sent by a process can be of either *fixed or variable size*. For fixed-size messages, the system level implementation is straighforward but it makes the task of programming more difficult. Variable size messages require a more complex system-level implementation but the programming task becomes simpler.
- There are several issues related with the link features like : *Naming, Synchronization and Buffering*.

- Processes need to have a *communication link* between them which can be implemented in various logical ways and the send()/receive() operations:

1. ***Direct or Indirect Communication***

   Under Direct Communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. This scheme employs ***symmetry*** in addressing.
   - 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 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.
   - Between each pair of processes, there exists exactly one link.

   Another variant of direct communication is where only the sender names the recipient, the recipient is not required to name the sender. This scheme employs ***asymmetry*** in addressing.
   - 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.

   The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions. Changing the identifier of a process may necessitate examining all other process definitions.

   Under Indirect Communication, messages are sent to and received from ***mailboxes***, or ports. A mailbox is an object into which messages can be placed by processes and from which messages can be removed. Each mailbox has a unique identification. Two processes can communicate only if the processes have a shared mailbox. A mailbox may be owned either by a process or by the operating system.
   - 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 of the pair have a shared mailbox.
   - A link may be associated with more than two processes.
   - Between each pair of communicating processes, there may bea number of different links, with each link corresponding to one mailbox.

   Q) Suppose processes P1, P2 and P3 all share mailbox A. Process P1 sends a message to A while both P2 & P3 execute a receive() from A. Which process will receive the message sent by P1?

   R) Answer depends on which of the following methods we chose:
   - Allowa link to be associated with two processes at most.
   - Allow at most one process at a time to execute a receive () operation.
   - Allow the system to select arbitrarily which one process will receive the message. The system also may define an algorithm for selecting which process will receive the message. The system may identify the receiver to the sender.

2. ***Synchronous or Aysnchronous Communication***

   ○ Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

   ○ Message passing may be either blocking or nonblocking also known as synchronous and asynchronous.

   - *Blocking send*: The sending process is blocked until the message is received by the receiving process or by the mailbox.
   - *Nonblocking send*: The sending process sends the message and resumes operation.
   - *Blocking receiver* : The receiver blocks until a message is available.
   - *Nonblocking receiver*: The receiver retrieves either a valid message or a null.

3. ***Automatic or Explicit Buffering***
   Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
   - *Zero capacity*: The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.
   - *Bounded capacity*: The 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 can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.
   - *Unbounded capacity*: The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

Advantages of Shared Memory Over Message Passing

Shared memory can be *faster* than message passing, since message-passing systems are typically implemented using system calls and thus require the more time-consuming task of kernel intervention. In shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Advantages of Message Passing Over Shared Memory

- Message passing is useful for exchanging smaller amounts of data, because no conflicts need to be resolved. In shared memory, processes need to ensure that they are not writing to the same location simultaneously.
- Message passing is used to implement in a distributed system than shared memory, because for shared memory, processes need to share a common address space which is not possible in remote machines connected through a network.
- The code for reading and writing the data from the shared memory should be written explicitly by the Application programmer, while no such code required here as the message passing facility provides mechanism for communication and synchronization of actions performed by the communicating processes.

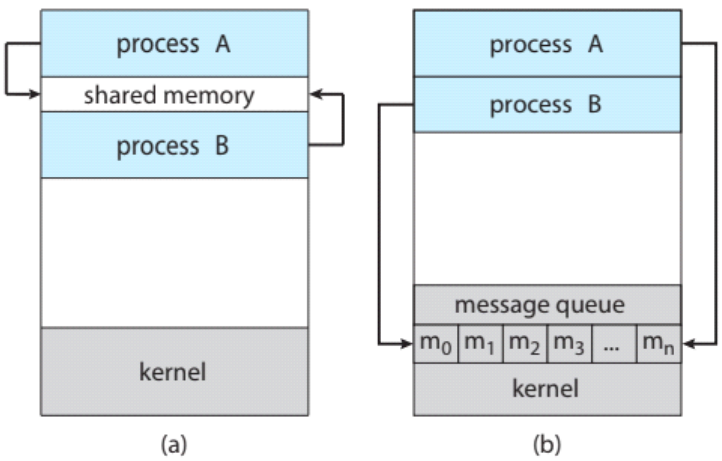


Figure 3.11 Communications models. (a) Shared memory. (b) Message passing.

Pipes in InterProcess Communication

A **Pipe** is a technique used for interprocess communication. A pipe is a mechanism by which the output of one process is directed into the input of another process. Thus it provides one way flow of data between two related processes.

Although pipe can be accessed like an **ordinary file**, the system actually manages it as **FIFO queue**. A pipe file is created using the pipe system call. A pipe has an input end and an output end. One can write into a pipe from input end and read from the output end. A pipe descriptor, has an array that stores two pointers, one pointer is for its input end and the other pointer is for its output end.

Suppose **two processes**, Process A and Process B, need to communicate. In such a case, it is important that the process which

writes, closes its read end of the pipe and the process which reads, closes its write end of a pipe. Essentially, for a communication from Process A to Process B the following should happen.

- Process A should keep its write end open and close the read end of the pipe.
- Process B should keep its read end open and close its write end. When a pipe is created, it is given a fixed size in bytes.

When a process attempts to write into the pipe, the write request is immediately executed if the pipe is not full. However, if pipe is full the process is blocked until the state of pipe changes. Similarly, a reading process is blocked, if it attempts to read more bytes that are currently in pipe, otherwise the reading process is executed. Only one process can access a pipe at a time.

Limitations **:**

- As a channel of communication a pipe operates in one direction only.
- Pipes cannot support broadcast i.e. sending message to multiple processes at the same time.
- The read end of a pipe reads any way. It does not matter which process is connected to the write end of the pipe. Therefore, this is very insecure mode of communication.
- Some plumbing (closing of ends) is required to create a properly directed pipe.

CPU Scheduling

CPU-I/O Burst Cycle

- A process execution consists of a cycle of CPU execution and I/O execution.
- Normally, every process ***begins with CPU*** burst that may be followed by I/O burst, then another CPU burst and then I/O burst and so on. Eventually, in the last it will ***end up on CPU burst***.
- A good CPU scheduling idea should choose the mixture of CPU bound & I/O bound processes so that both I/O devices and CPU can be utilized efficiently.
- *Frequency Curve:* It is exponential or hyperexponential curve which measures the duration of CPU bursts for a given process. An I/O bound process has large number of short CPU bursts whereas a CPU bound process has some few number of long CPU bursts.

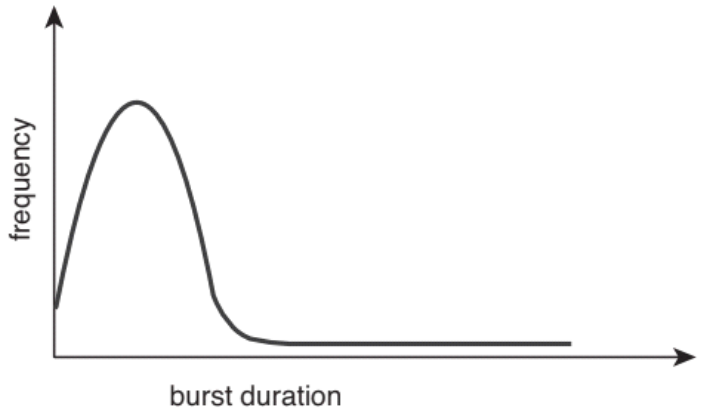


Figure 5.2 Histogram of CPU-burst durations.

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 CPU scheduler, which selects a process from the processes in memory that are ready to execute and allocates the CPU to that process.

CPU Scheduling may occur in following circumstances:

1. **Non-Preemptive or Cooperative**
   a. When a process completes it's execution i.e. terminates.
   b. When a process switches from the running state to the ready state: When a process leaves CPU voluntarily to perform some I/O operation or to wait for the termination of a child process.

   Non-preemptive scheduling is supported by Windows 3.1x.

2. **Preemptive**
   a. When a process enters in ready state either from new state (for eg a high priority process) or waiting state (for eg completion of I/O).
   b. When a process switches from running state to ready state (for eg time quantum expire or interrupt occurs)

   Pre-emptive scheduling is supported by Unix, Linux, Windows 95 & Higher.

Note: MS DOS does not support multiprogramming, hence no CPU scheduling.

CPU Scheduling Criteria

1. **CPU Utilization**
   CPU Utilization need to be as maximum as possible. CPU scheduling algorithm should reduce the CPU idleness as much as possible. It is given by percentage of time in which CPU was busy and percentage of time in which CPU was idle.

2. **Throughput:**
   Throughput is used to measure the amount of work CPU has done in executing the processes. It is the number of processes

that are completed per unit time. For long processes, this rate may be one process over several seconds, while for short transactions, it may be tens of processes per second.

3. **Turn Around Time (TAT):**
   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 in the ready queue, executing on the CPU, and doing I/O. Thus, it is the total time spent by the process in the system.

   - ***Burst Time/ Execution Time/ Running Time (BT)***: It is the time process require for running on CPU.
   - ***Arrival Time (AT)***: Time when a process enters the ready state.
   - ***Exit Time (ET)***: Time when process completes execution and terminates from the system.

   - Tun Around Time = Exit Time - Arrival Time
   - Turn Around Time = Burst Time + Waiting Time
   - **TAT = ET - AT = BT + WT**

4. **Average Waiting Time**
   The CPU-scheduling algorithm does not affect the amount of time during which a process executes or does I/O. The total execution time required for a process on a given system is fixed.
   CPU Scheduling algorithm affects only the amount of time that a process spends waiting in the ready queue. ***Waiting Time (WT)*** is the time spend by a process in ready state waiting for CPU.

5. **Average 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.
   Time between a process enters the ready queue and get scheduled on the CPU for the first time is known as ***Response Time***.
   Thus average response time for all the processes will be better scheduling criteria for interactive systems.

CPU Scheduling Algorithms

Scheduling algorithms are used to select a process for execution. There are several well known scheduling algorithms:

*Note**: Gantt chart*** is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes.

1. **First-Come First-Serve Scheduling (FCFS):**
   The process that requests the CPU first is allocated the CPU first. It is always non-preemptive in nature.

   Advantages:
   - 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.
   - It should be used for background processes where execution is not urgent.

   Disadvantages:
   - The average waiting time under the FCFS policy is often quite long. It is not minimal and may vary substantially if the processes' CPU burst times vary greatly.
   - It doesn't account any consideration of priority of process, or the burst time of the process.
   - Since it 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. Hence it should not be used for dynamic/interactive system, where it is important that each process 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.
   - Smaller (small burst time or I/O bound) process have to wait for long time for longer (long burst time or CPU bound) process to release CPU. This effect is known as ***convoy effect***. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.

Explanation of Convoy Effect

Consider system have one CPU bound and many I/O bound processes. 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.

2. Shortest-Job-First Scheduling (SJF):

- Out of all available processes, CPU is assigned to the process having smallest burst time requirement (no priority, no seniority). If the CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.
- The SJF algorithm can be either preemptive or nonpreemptive. 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.

Note:

- Since processes usually have multiple CPU burst, burst time refers to the next burst time (remaining burst time). A more appropriate term for this scheduling method should be *shortest-next-CPU-burst algorithm*, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.
- SJFS can be used in either non-preemptive approach or preemptive approach. If preemptive SJFS need to be referred, shortest-remaining-time-first is used, thus ***Pre-emptive Shortest Job First = Shortest Remaining Time First***
- If approach is not given , i.e. simpy Shortest Job First (SJF) is given, then it should be assumed to be of non-preemptive approach.

Advantages:

- The SJF scheduling algorithm is probably ***optimal***, as it guarantees the ***minimum average waiting time*** for a given set of processes, setting a standard among all algorithms in terms of average waiting time. Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases.
- It has better average response time when compared to First-Come First Serve Scheduling (FCFS) algorithm.

Disadvantages:

- Since there is no way to measure the exact burst time of a process, it cannot be implemented in real-sense and hence is a theoretical algorithm.
  Note: One approach to overcome the problem can be *approximate SJS scheduling.* It is expected that the next CPU burst will be similar in length to the previous ones. (The next CPU burst is generally predicted as an *exponential average* of the measured lengths of previous CPU bursts). By computing an approximation of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
- Processes with longer CPU burst time requirement will go into ***starvation*** (or ***indefinit blocking***) for CPU.
- Since there is no idea of priority of processes, processes with large burst time has poor response time.

3. Priority Scheduling

- A priority is associated with each process. At any instant of time, out of all available processes, CPU is allocated to the process which possess the highest priority (number can be higher or lower). Tie is broken using FCFS order.
- Priority scheduling can be either preemptive or nonpreemptive. 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 nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Internal vs External Priority: Priorities can be defined either internally or externally.

- Internally defined priorities use some measurable quantity(s) 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.
- 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 political factors.

Note: Shortest Job First (SJF) scheduling is a special type of priority scheduling where the priority of a process is inverse of its CPU burst time requirement. The larger the CPU burst, the lower the priority, and vice versa.

Advantages:
- It provides the facility for high priority especially for system processes.
- It also allow important process to run even if it is an user process.

Disadvantages:
- Process with low priority may go into **starvation** for CPU.
- There is no importance given to the burst time or the waiting time of a process.

Problem: Starvation or Indefinit blocking
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. For a heavily loaded computer system, the low priority process will eventually run when load reduces or the computer system will eventually crash and lose all unfinished low-priority processes.

Solutions to tackle Starvation:
- ***Ageing*** is a technique of gradually increasing the priority of process that wait for the CPU for long time. Eventually, even a process with an lowest priority initially would have the highest priority in the system and would be executed. This event of lowest priority process becoming high priority is known as ***priority inversion***.

- ***Combine round-robin and priority scheduling*** in such a way that the system executes the highest-priority process and runs processes with the same priority using round-robin scheduling.

4. Round-Robin Scheduling (RR)

- This algorithm is designed for *time-sharing system*, where it is not necessary to complete one process and start another, but to be responsive and divide the time of the CPU among the processes in ready state.
- Here, ready queue will be treated as circular queue. We will fix a ***time quantum or time slice*** upto which a process can hold the CPU in one go, with in which either a process terminates or process must release the CPU and re-enter in the circular queue and wait for it's next chance to get scheduled. Time quantum is generally 10 to 100 milliseconds in length.
- 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.
- It is always pre-emptive in nature. 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.
- 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.
- Performance depends heavily on *time quantum*. If the time quantum is extremely large, the RR policy is the same as the FCFS policy. Whereas if the time quantum is extremely small, RR approach can result into a large number of context switches. Hence it should be large enough to avoid heavy overhead of context switching and small enough to avoid FCFS kind of strategy.
- Turnaround time also depends on the size of the time quantum. The average turnaround time can be improved if most processes finish their next CPU burst in a single time quantum.

Advantages:
- It performs best in terms of average response time.
- It works well in cases of time-sharing system, client-server architecture and interactive system.
- It is a kind of SJF implementation.

Disadvantages:
- Longer processes may ***starve*** for CPU.
- Average waiting time under RR is often quite long.
- There is no idea of priority.

5. Multilevel Queue Scheduling

- There are separate queues for each distinct priority, and priority scheduling simply schedules the process in the highest-priority queue.
- It also works well when priority scheduling is combined with round-robin: if there are multiple processes in the highest-priority queue, they are executed in round-robin order.
- A priority is assigned statically based on the process type to each process, and a process remains in the same queue for the duration of its runtime.
- Processes with different priority have different response-time requirements and different scheduling needs. Each queue (with different priority processes) might have its own scheduling algorithm.
- In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling. Each queue has absolute priority over lower-priority queues.
- No process in the batch queue could run unless the queues for real-time processes, system processes, and interactive processes were all empty. If an interactive process entered the ready queue while a batch process was running, the batch process would be preempted.

Advantages: No scheduling overhead as each process will not move from one queue to another.
Disadvantages:

- It is inflexible as there are fixed queues based on process types.
- Lower priority processes will starve for CPU if high priority processes are in the high-level queue and there is absolute priority among different queues.

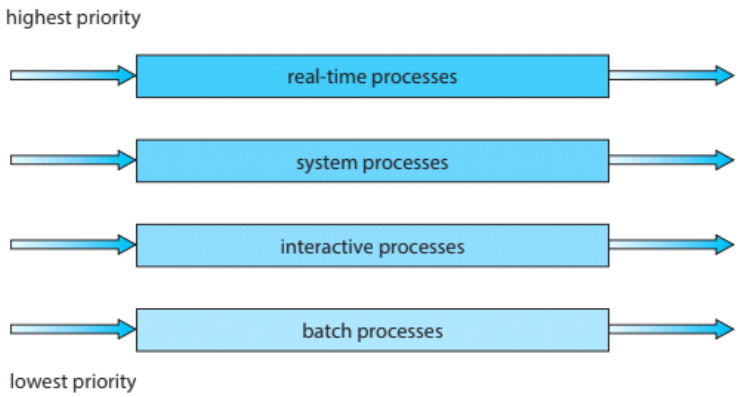


Figure 5.8 Multilevel queue scheduling.

6. **Multilevel Feedback Queue Scheduling**

- The multilevel feedback queue scheduling algorithm allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts. 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 which are typically characterized by short CPU bursts in the higher-priority queues.
- A process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
- This algorithm gives processes with short burst times the highest priority. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
- Long processes automatically sink to last queue and are served in FCFS order with any CPU cycles left over from above queues.

A multilevel feedback queue scheduler is defined by the following parameters:

- The number of queues
- The scheduling algorithm for each queue
- The method used to determine when to upgrade a process to a higher priority queue
- The method used to determine when to demote a process to a lower priority queue

- The method used to determine which queue a process will enter when that process needs service

Although it can be modified according to the needs of the system, it is ***very complex*** to build as process might need to be scheduled based on all of the parameters mentioned above.

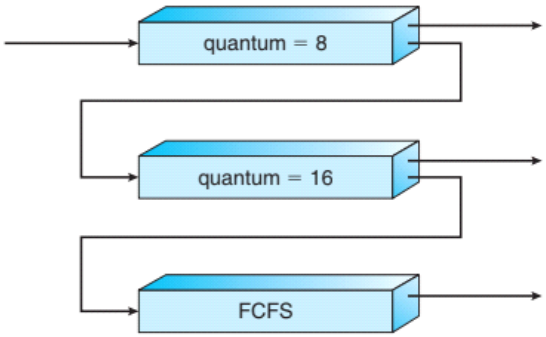


Figure 5.9 Multilevel feedback queues.

Note: Since systems in today's world uses real-time operating system which have multiple cores, hence ***multiprocessor scheduling*** is generally used in real world scenario. Each processor has it's own scheduling algorithms (master CPU can schedule processes for all slave CPUs or all CPU cores can self-schedule processes). It's major requirement is ***load balancing,*** i.e. to keep all processors equally loaded.

Critical Section Problem

Introduction

- *Concurrent Process Execution*: The CPU scheduler switches rapidly between processes to provide concurrent execution, i.e. one process may only partially complete execution before another process is scheduled. In fact, a process may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- *Parallel Process Execution:* In multicore programming, two instruction streams (representing different processes) execute simultaneously on separate processing cores.
- Concurrent or Parallel execution of cooperating sequential processes (or threads) can contribute to issues involving the integrity and *consistency of data resources* shared by them.
- *Process Synchronization* means sharing system resources by processes in a such a way that, concurrent access to shared data is handled which leads to minimize the chance of inconsistent data.

Producer - Consumer Problem:

Consider *Bounded Buffer* where maximum items that can be produced by the producer (if no item gets consumed by consumer) is BufferSize - 1. Maintain an integer variable *count* initialized with 0. count is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer

Producer

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

$register_1 = count$
$register_1 = register_1 + 1$
$count = register_1$

Consumer

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

$register_2 = count$
$register_2 = register_2 - 1$
$count = register_2$

- Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. Suppose count = 5 and that the producer and consumer processes concurrently execute the statements “count++” and “count--”.

- Concurrent execution of above two processes in different fashion (context-switching at different instances) may lead to three different final results of count = 4, 5, or 6. Though, the only correct result is count == 5, which is generated correctly if the producer and consumer execute separately.

- *register1* and *register2* are the local CPU registers. Even though register1 and register2 may be the same physical register, the contents of this register will be saved and restored by the interrupt handler.

- Example of sequential & concurrent execution of these processes, when interleaved (context-switched) in some arbitrary order resulting in count != 5 after the processes execution completes:

T_0:	*producer*	execute	$register_1 = count$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = count$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$count = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$count = register_2$	$\{count = 4\}$

T_0: *producer* execute $register_1 = count$ $\{register_1 = 5\}$
T_1: *producer* execute $register_1 = register_1 + 1$ $\{register_1 = 6\}$
T_2: *consumer* execute $register_2 = count$ $\{register_2 = 5\}$
T_3: *consumer* execute $register_2 = register_2 - 1$ $\{register_2 = 4\}$
T_4: *producer* execute $count = register_1$ $\{count = 6\}$
T_5: *consumer* execute $count = register_2$ $\{count = 4\}$

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable count concurrently. Such a situation is called *race condition*. To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable count.

Race Condition: A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.
In order to avoid such a condition, we need to synchronize the cooperative processes so that only one of them may access the shared resource at a given instant of time.

Example) Kernel data structure that maintains a list of all open files in the system must be modified when a new file is opened or closed (adding the file to the list or removing it from the list). If two processes were to open files simultaneously, the separate updates to this list could result in a race condition.

Other kernel data structures that are prone to possible race conditions include structures for maintaining memory allocation, for maintaining process lists, and for interrupt handling.

Critical Section Problem: The critical-section problem is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data.

A segment of code in a process in which the process may be accessing and updating data that is shared with at least one other process is known as ***critical section***. Thus, a critical section is group of instructions/statements or region of code that need to be executed *atomically (*uninterruptly*).* Example) Code for accessing a resource like file, input or output port, global data, etc.

Consider a system of n processes (P1, P2, …. Pn) which needs to share same data/resource. An important feature is when one process is executing in its critical section, no other process is allowed to execute in its critical section i.e. no two processes are executing in their critical sections at the same time.

Each process must request permission to enter its critical section. The section of code implementing this request is the ***entry section***. The critical section may be followed by an ***exit section***. The remaining code is the ***remainder section***.

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

Figure 6.1 General structure of a typical process.

Requirements of Synchronization Mechanisms for Solving Critical Selection Problem:
Primary (Necessary)

1. **Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

2. **Progress:** If one process doesn't need to execute into critical section then it should not stop other processes to get into the

critical section. Also, selection of a process which will go to critical selection cannot be postponed indefinitely, i.e. concurrent processes should not go into *deadlock*.

Secondary (Optional)

3. **Bounded Waiting:** There exists a bound, or limit, 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.

4. **Architectural Neutrality:** Our mechanism must be architectural natural. It means that if our solution is working fine on one architecture then it should also run on the other ones as well.

Methods for Solving Critical Selection Problem

Q) Can ***Disabling Interrupts*** work as a method to solve critical selection problem?

- We could prevent interrupts from occurring while a shared variable was being modified. Current Sequence of instructions would be allowed to execute in order without preemption.
- Though, this solution can solve the critical-section problem in a single-core environment, it is not as feasible in multiprocessor environment.
- Disabling interrupts on a multiprocessor can be time consuming, since the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases.

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

1. ***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.
   Thus, a nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

2. ***Preemptive kernels***: A preemptive kernel allows a process to be preempted while it is running in kernel mode.
   Preemptive kernels may face race-conditions, hence they must be carefully designed to ensure that shared kernel data are free from race conditions.

   Preemptive kernels are favoured even though Non-preemptive kernels are free from race conditions because:
   - 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 giving the processor to waiting processes.
   - 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.

   All process synchronization methods discussed will be solutions for the critical section problem for preemptive kernels.

Peterson's Solution & Hardware Solution

1. **Peterson's Solution**

It is a classic software-based solution which provides 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.

Peterson's solution is restricted to two processes (P_0 and P_1) that alternate execution between their critical sections and remainder sections. It requires the two processes to share two data items:

1. ***int turn***: The integer variable turn indicates whose turn it is to enter its critical section. For example, if turn == 0, then process P_0 is allowed to execute in its critical section, else process P_1.

2. ***boolean flag[2]:*** The boolean flag array is used to indicate if a process is ready to enter its critical section. For example, if flag[0] is true, P_0 is ready to enter its critical section, else not. Similarly, if flag[1] is true, P_1 is ready to enter its critical section, else not.

```
while (true) {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j)
        ;

        /* critical section */

    flag[i] = false;

        /*remainder section */
}
```

Figure 6.3 The structure of process P_i in Peterson's solution.

Peterson's Solution preserves mutual exclusion, progress requirement is satisified and bounded-waiting requirement is also set.

- P_0 enters its critical section only if either flag[1] == false or turn == 0. Similarly P_1 enters its critical section only if either flag[0] == false or turn == 1. If both processes try to enter at the same time, turn's value can be either 0 or 1 but not both, thus only one of the processes can enter critical section at a time and other process will remain in while loop until the former process leaves the critical section. This ensures *mutual exclusion* in the solution.

- To enter the critical section, process P_0 first sets flag[0] to be true and then sets turn to the value 1, thereby asserting that if the other process wishes to enter the critical section after it leaves it's critical section, the other process can do so.
  If P_1 do not require to enter the critical section, then flag[1] == false, and P_0 can enter its critical section again. This assures *progress* in the solution.

- If P_1 is stuck in while loop, then flag[0] == true and turn == 0. But whenever P_0, will leave the critical section, flag[0] will be set to false. Thus P_1 will break out of loop and enter it's critical section. Hence any process will enter the critical section after at-most one entry by other process, thus ensuring *bounded waiting*.

Note: Lower-level software-based solutions like Peterson's Solution will not work for modern-day architectures because of the way they perform basic machine-language instructions, such as *load* and *store*. To improve system performance, modern day processors and/or compilers may reorder read and write operations that have no dependencies. This may not effect correctness of Peterson's algorithm for single-threaded application, but for a multi-threaded application with shared data, the reordering of instructions may render inconsistent or unexpected results.

2. **Hardware/Architecture Based Solution**

Modern computer systems provide special atomic (uninterruptable) hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words.

The important characteristic of these instructions is that they are executed ***atomically*** , i.e. they cannot be interrupted and will run as single instruction unit. Thus, if two hardware instructions (*test_and_set()* or *compare_and_swap()*) are executed simultaneously (each on a different core), they will be executed sequentially and not in mixed fashion.

2.a Test and Set Instruction & Lock (TSL)

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

We can implement mutual exclusion and progress by declaring a boolean variable *lock* (initialized to *false*) and using *test_and_set()* instruction. Although this algorithm satisfies the mutual-exclusion requirement, it does not satisfy the bounded-waiting requirement. And since it is architecture based solution, it does not satisfy architectural neutrality also.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 6.6 Mutual-exclusion implementation with `test_and_set()`.

We can modify the above implementation to satisfy the *bounded-waiting* requirement in the following way:

```
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 (TRUE);
```

Figure 6.8 Bounded-waiting mutual exclusion with `TestAndSet()`.

2.b Compare And Swap Instruction (CAS)

The compare and swap() instruction (CAS), just like the test and set() instruction, operates on two words atomically, but uses a different mechanism that is based on swapping the content of two words.

The CAS instruction operates on three operands and the operand value is set to new value only if the expression (*value == expected) is true. CAS always returns the original value of the variable value.

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

Figure 6.7 The definition of the atomic `compare_and_swap()` instruction.

A global variable (*lock*) is declared and is initialized to 0. The first process that invokes compare and swap() will set lock to 1. It will then enter its critical section, because the original value of lock was equal to the expected value of 0. Subsequent calls to compare and swap() will not succeed, because lock now is not equal to the expected value of 0.When a process exits its critical section, it sets lock back to 0, which allows another process to enter its critical section.

```
while (true) {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
}
```

Figure 6.8 Mutual exclusion with the `compare_and_swap()` instruction.

Similar to Test And Set Instruction, above implementation ensures mutual exclusion and progress but do not fulfill bounded waiting and architectural neutrality requirements. Below modification can satisfy bounded-waiting requirement also:

```
while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;

      /* critical section */

   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;

   if (j == i)
      lock = 0;
   else
      waiting[j] = false;

      /* remainder section */
}
```

Figure 6.9 Bounded-waiting mutual exclusion with `compare_and_swap()`.

Note: The hardware-based solutions to the critical-section problem are complicated as well as generally inaccessible to application programmers. Hence, operating-system designers build higher-level software tools to solve the critical-section problem.

Mutex Locks & Semaphores

Mutex Locks

- Mutex Lock is a higher-level software-based solution to protect critical sections and thus prevent race conditions. A process must acquire the lock before entering a critical section and release the lock when it exits the critical section.
- The *acquire()* function acquires the lock, and the *release()* function releases the lock. A mutex lock has a boolean variable *available* whose value indicates if the lock is available or not.
- If the lock is available, a call to *acquire()* succeeds, and the lock is then considered unavailable. A process that attempts to acquire an unavailable lock is blocked until the lock is released.
- *Note*: Calls to instructions *acquire()* and *release()* are performed *atomically* (uninterruptly).

```
while (true) {
    acquire lock
        critical section
    release lock
        remainder section
}
```

Figure 6.10 Solution to the critical-section problem using mutex locks.

The definition of `acquire()` is as follows:

```
acquire() {
    while (!available)
       ; /* busy wait */
    available = false;
}
```

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

Busy Waiting:

While a process is in its critical section, any other process that tries to enter its critical section must loop continuouslyin the call to *acquire()*. This continual looping is clearly a problem in a real multi-programming system, where a single CPU core is shared among many processes. Busy waiting also wastes CPU cycles that some other process might be able to use productively. Thus busy waiting is a disadvantage for mutex locks.

Spin Lock:

Mutex lock is also called a *spin lock* which causes a process trying to acquire the lock to simply wait in a loop ("spin") while repeatedly checking if the lock is available. Since the process remains active but is not performing a useful task, the use of such a lock results in busy waiting.

Q) Why spin lock is not appropriate for single-core processing systems?
R) It is not preferred in single-core systems because of busy waiting and the CPU utilization time being wasted. Consider a high priority thread which attempts to claim a spinlock already held by a lower priority thread: it will just loop forever and the lower priority thread will never get another chance to run and release the spinlock.

Q) Though spin lock causes busy waiting in single-core systems, yet they are preferred choice for modern multi-core systems. Why ?
R) Although, they may cause busy waiting, spin locks have an advantage because no context switch is required when a process

wait to acquire a lock . Since a context switch may take considerable time, spinlocks are the preferable choice for locking in multicore systems. If a lock is to be held for a short duration, one thread can *spin* on one processing core while another thread performs its critical section on another core. Hence on modern multi-core computing systems, spinlocks are widely used in many operating systems.

Semaphores

- Semaphore S is an integer variable that, apart from initialization, is accessed only through two standard ***atomic*** operations: *wait()* and *signal()*.
- Since these operations are executed atomically, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. Also within *wait()* block, the testing of the integer value of S in while($S \leq 0$), as well as its possible modification (S--), must be executed without interruption.
- *Note: wait()* operation is also known as *P()* which means to-test and decrement, while *signal()* operation is also known as *V()* which means to increment.

Wait()

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

Signal()

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

Two Types of Semaphores:

1. *Binary Semaphore*
   It's value can only be either 0 or 1. Thus, binary semaphores behave similarly to mutex locks. In fact, on systems that do not provide mutex locks, binary semaphores can be used instead for providing mutual exclusion.

2. *Counting Semaphore*
   - Its value can range over an unrestricted domain.
   - They can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of available resources.
   - Each process that wishes to use a resource performs a *wait()* operation on the semaphore (thereby decrementing the count). When a process releases a resource, it performs a *signal()* operation (incrementing the count).
   - When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

Advantages of Semaphores

- Semaphores allow only one process into the critical section. They follow the mutual exclusion principle strictly and are much more efficient than some other methods of synchronization.
- There is no resource wastage because of busy waiting in semaphores as processor time is not wasted unnecessarily to check if a condition is fulfilled to allow a process to access the critical section.
- Semaphores are implemented in the machine independent code of the microkernel. So they are machine independent.

Disadvantages of Semaphores

- Semaphores are complicated so the wait and signal operations must be implemented in the correct order to prevent deadlocks.
- Semaphores are impractical for last scale use as their use leads to loss of modularity. This happens because the wait and signal operations prevent the creation of a structured layout for the system.
- Semaphores may lead to a priority inversion where low priority processes may access the critical section first and high priority processes later.

Applications of Semaphores:

1. *Process Synchronization*

2. *Order of Execution of Processes*
   Consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0.
   Process P1:

```
S1;
signal(synch);
```

   Process P2:

```
wait(synch);
S2;
```

3. *Resource Management*

Implementation of Semaphores (with reduced Busy Waiting):
The previous implementation of *wait()* and *signal()* faces the problem of busy waiting. To overcome this problem, we can modify the definition of the wait() and signal() operations.

- When a process executes the *wait()* operation and finds that the semaphore value is not positive, it must wait. However, rather than engaging in busy waiting, the process can *suspend* itself.
- The suspend 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 suspended, 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 and is placed in the ready queue.
- The *sleep()* operation suspends the process that invokes it. The *wakeup(P)* operation resumes the execution of a suspended process P. These two operations are provided by the operating system as basic *system calls*.
- The list of waiting processes can be implemented by a pointer to a list of PCBs which is treated as FIFO queue. Semaphore contains both head and tail pointers to the queue.

Structure of Semaphore (to avoid busy waiting)

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

Wait() Operation

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            sleep();
        }
}
```

Signal() Operation

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

Note:

- Since, semaphore operations need to be executed atomically, no two processes can execute wait() and signal() operations on the same semaphore at the same time. This is a critical section problem, and in a single-processor environment, we can solve it by simply disabling interrupts during the time the wait() and signal() operations are executing. This ensures sequential order of execution of processes to avoid race condition.
- In a multicore environment, interrupts must be disabled on every processing core. Otherwise, instructions from different processes (running on different cores) may interleave in random fashion leading to race condition. Disabling interrupts on every core can be a difficult task and can seriously decrease performance. Therefore, mutex locks (spin locks) are used to implement semaphores so that wait and signal operations are implemented atomically.

Busy Waiting Analysis

- We have not completely eliminated busy waiting even with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the *wait()* and *signal()* operations.
- If these sections are short, the critical section is almost never occupied, and busy waiting occurs rarely or for only a short time. But for application programs whose critical sections may be long or may almost always be occupied, busy waiting is extremely inefficient.

Deadlock Situation

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event (signal() operation) that can be caused only by one of the waiting processes. When such a state is reached, these processes are said to be ***deadlocked***.

Example) Consider two processes P_0 and P_1 associated with two semaphores S and Q initialized with 1.

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

Suppose that P_0 executes wait(S) and then P_1 executes wait(Q).When P_0 executes wait(Q), it must wait until P_1 executes signal(Q). Similarly, when P_1 executes wait(S), it must wait until P_0 executes signal(S). Since these *signal()* operations cannot be executed, P_0 and P_1 are deadlocked.

Priority Inversion

- When a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process, since kernel data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource.

- Let there are 3 processes— L, M, and H —with priority order L < M < H. Assume that process H requires a semaphore S, which is currently being accessed by process L. Process H would wait for L to finish using resource S. Now, process M preempts process L, thus indirectly, a process with a lower priority (process M) has affected higher priority process (process H) to wait for shared resource S for longer.

- This problem is known as ***priority inversion***, which can occur in system with atleast 3 processes of different priorities.

- Solution to this problem is known as ***priority inheritance***. All processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the shared resources.When they are finished, their priorities revert to their original values.

- Thus process L will temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L had finished using resource S, it would relinquish its inherited priority from H and assume its original priority.

Because resource S would now be available, process H (not M) would run next.

Q) Important: Difference between mutex and semaphores?
R) https://www.geeksforgeeks.org/mutex-vs-semaphore/

Classical Problems

1. ***Bounded Buffer Problem (or Producer - Consumer Problem)***

The producer and consumer processes share the following data structures:

int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;

Pool consists of n buffers, each capable of holding one item. The mutex binary 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. The semaphore empty is initialized to the value n while the semaphore full is initialized to the value 0.

Producer Operation

```
while (true) {
     . . .
   /* produce an item in next_produced */
     . . .
   wait(empty);
   wait(mutex);
     . . .
   /* add next_produced to the buffer */
     . . .
   signal(mutex);
   signal(full);
}
```

Figure 7.1 The structure of the producer process.

- A producer first waits until there is atleast one empty slot.
- Then it decrements the **empty** semaphore because, there will now be one less empty slot, since the producer is going to insert data in one of those slots.
- Then, it acquires lock on the buffer, so that the consumer cannot access the buffer until producer completes its operation.
- After performing the insert operation, the lock is released and the value of **full** is incremented because the producer has just filled a slot in the buffer.

Consumer Operation

```
while (true) {
   wait(full);
   wait(mutex);
     . . .
   /* remove an item from buffer to next_consumed */
     . . .
   signal(mutex);
   signal(empty);
     . . .
   /* consume the item in next_consumed */
     . . .
}
```

Figure 7.2 The structure of the consumer process.

- The consumer waits until there is atleast one full slot in the buffer.
- Then it decrements the **full** semaphore because the number of occupied slots will be decreased by one, after the consumer completes its operation.
- After that, the consumer acquires lock on the buffer.

- Following that, the consumer completes the removal operation so that the data from one of the full slots is removed.
- Then, the consumer releases the lock.
- Finally, the **empty** semaphore is incremented by 1, because the consumer has just removed data from an occupied slot, thus making it empty.

2. ***Reader Writer Problem***

Problem Statement:
Consider a database that need to be shared among several concurrent processes. Some of these processes may want only to read the database known as ***readers***, whereas others may want to update (that is, read and write) the database known as ***writers***. Two readers can access the shared data simultaneously without caring any critical section problem. However, if a writer and some other process (either a reader or a writer) access the database simultaneously, race condition can arise.
We require that the writers have exclusive access to the shared database while writing to the database. Also, no reader need to be kept waiting unless a writer has already obtained permission to use the shared object, i.e. in simple words, reader has more priority over the writer.

Solution: The reader processes share the following data structures:

semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;

- The binary semaphores *mutex* and *rw_mutex* are initialized to 1 while *read_count* is a counting semaphore initialized to 0.
- The ***mutex*** semaphore is used to ensure mutual exclusion when the variable *read_count* is updated.
- The ***read_count*** variable keeps track of how many processes are currently reading the object.
- The binary semaphore ***rw_mutex*** is common to both reader and writer processes. It functions as a mutual exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers that enter or exit while other readers are in their critical sections.

If a writer is in the critical section and n readers are waiting, then one reader is queued on *rw_mutex*, and n − 1 readers are queued on *mutex*. Also when a writer executes signal (rw_mutex), we may resume the execution of either the waiting readers or a single waiting writer. The selection is made by the scheduler.

Writer Operation

```
while (true) {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

- Writer requests the entry to critical section.
- If allowed i.e. *wait()* gives a true value, it enters and performs the write. If not allowed, it keeps on waiting.
- It exits the critical section.

Reader Operation

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

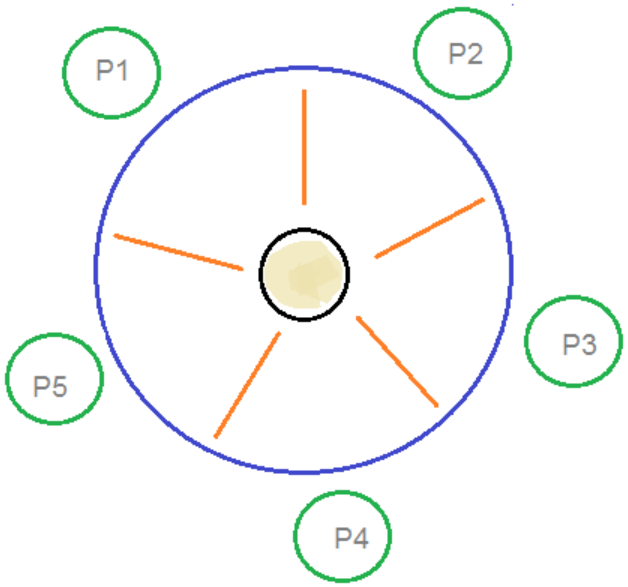
Figure 7.4 The structure of a reader process.

- Reader requests the entry to critical section.
- If allowed:
  - it increments the count of number of readers (*read_count*) inside the critical section.
  - If this reader is the first reader entering, it locks the *rw_mutex* semaphore to restrict the entry of writers if any reader is inside.
  - It then, signals *mutex* as any other reader is allowed to enter while others are already reading.
  - After performing reading, it exits the critical section.
  - When exiting, it checks if no more reader is inside, it signals the semaphore *rw_mutex* as now, writer can enter the critical section.
- If not allowed, it keeps on waiting

3. ***Dining Philophoser Problem***

Problem Statement

Consider five philosophers who spend their lives thinking and eating. The philosophers share a 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. 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 a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.



Dining Philosophers Problem

Solution

Represent each chopstick with a semaphore. A philosopher tries to grab a 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]*** where all the elements of chopstick are initialized to 1.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Figure 7.6 The structure of philosopher *i*.

Although this solution ensures *mutual exclusion* i.e. no two neighbouring philosophers can eat simultaneously, it does not ensures *progress* as the situation of ***deadlock*** may arise.

Consider situation where all five philosophers become hungry at the same time and each grabs her left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab her right chopstick, she will be delayed forever.

Modifications to avoid Deadlock & ensure progress

1. Allow at most four philosophers to be sitting simultaneously at the table.
2. Allow a philosopher to pick up her chopsticks only if both chopsticks are available, i.e. don't allow *hold and wait* nature of philosopher.

Above modifications require to mold the problem statement, hence different modification need to be proposed.

3. Using Extra Semaphore, we can make a philosopher to either pick-up both chopsticks, or wait for the first one also. This

solution is also not acceptable as it requires extra semaphore handling.

The following *Assymetric* Solutions are considered better and appropriate modification to avoid deadlock.

4. An odd-numbered philosopher picks up first her left chopstick and then her right chopstick, whereas an even numbered philosopher picks up her right chopstick and then her left chopstick.
5. All philosophers will first pick left chopstick then right chopstick, except the last philosopher (5th one) who picks right chopstick first, then only left chopstick.

Even though the solution is free from deadlock condition, it does not ensure that there will not be ***starvation***. One of the philosopher may always starve to death.

Deadlock

Definition

A set of processes are said to be in a deadlocked state when "every process in the set is waiting for an event that can be caused only by another process in the set."

In a multi-programming environment, several processes may compete for a finite number of resources. A process requests resources and if the resources are not available at that time, the process enters a waiting state. If the process is unable to changes it's waiting state indefinitely, because the resources requested by it are held by another waiting process, then the system is said to be in deadlock.

System Model

The resources are partitioned into several types each consisting of some number of identical instances. Example of resource types are CPU cycles, files, and I/O devices (such as network interfaces). Synchronization tools like mutex locks and semaphores are also type of system resources and are most common sources of deadlock.

Sequence of Resource Utilization by a Process

1) ***Request:*** The thread requests the resource. If the request cannot be granted immediately, then the requesting thread must wait until it can acquire the resource. Number of instances of a resource requested cannot be greater than total available resources in the system.
2) ***Use:*** If resource is allocated to the process, the process can operate on the resource.
3) ***Release:*** Process releases the resource after use.

Note: The request and release of resources may be system calls like *request()* and *release()* of a device, *open()* and *close()* of a file, *allocate()* and *free()* memory system calls , *wait()* and *signal()* operations on semaphores and through *acquire()* and *release()* of a mutex lock.

System Table records whether each resource is free or allocated. For each resource that is allocated, the table also records the process to which it is allocated. If a process requests a resource that is currently allocated to another process, it can be added to a queue of process waiting for this resource.

Necessary Conditions for Deadlock Situation

A deadlock situation can arise if the all four conditions hold simultaneously in a system:

1. **Mutual Exclusion:** At least one resource must be held in a non-sharable mode i.e. only one process at a time can use the resource. If another process requests that resource, the requesting process must wait for the resource until it has been released.

2. **Hold and Wait:** A process is holding at least one resource and waiting to acquire additional resources that are currently being held by other process.

3. **No Preemption:** Resources cannot be preempted from a process by any other process. A resource can be released only voluntarily by the process holding it after completing it's task.

4. **Circular Wait:** A set $\{T_0, T_1, ..., T_n\}$ of waiting processes exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, ..., T_{n-1} is waiting for a resource held by T_n, and T_n is waiting for a resource held by T_0.

Resource Allocation Graph

- System Resource Allocation Graph is a directed graph which is used to describe deadlock situations in a system. It consists of V vertices and E edges. Vertices are either of resource type Ri or of process (or thread) type Ti.

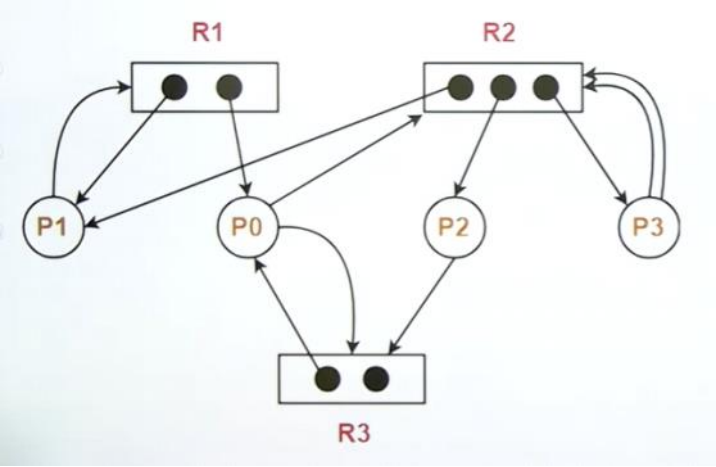
- If an edge is from process Pi to a resource type Rj i.e. Pi -> Rj , then it is ***request edge*** which depicts that the process Pi is requesting an instance of resource Rj and is currently waiting for that resource. Number of edges from a process to a resource depicts that process is requesting that many instances of that resource.

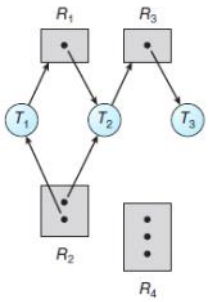
- If an edge if from resource type Ri to process Pj i.e. Ri -> Pj , then it is ***assignment edge*** which signifies that an instance of

resource type Ri has been allocated to process Pj. Number of edges from a resource type to a process signifies, that many number of instance of the resource are allocated to the process.

- When process Pi requests an instance of resource type Rj, a *request edge* is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is instantaneously transformed to an *assignment edge*.When the process no longer needs access to the resource, it releases the resource. As a result, the assignment edge is deleted.

- If a resource-allocation graph does not have a cycle, then the system is not in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state.





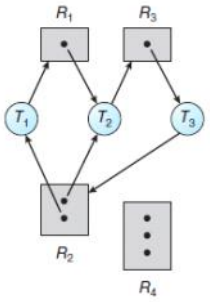


Figure 8.4 Resource-allocation graph. **Figure 8.5** Resource-allocation graph with a deadlock.

Deadlock Handling

There are four methods to handle deadlocks:

1. **Prevention**
   It designs such a system which violates alteast one of the four necessary conditions of deadlock, thus ensuring independence from deadlock situation.

2. **Avoidance**
   System maintains a set of data (provided in advance) using which it can take a decision whether to entertain a new resource request or not so that the system remains in *safe state*.
   To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

3. **Detection and Recovery**
   System waits until the deadlock situation occurs and once it is detected, it applies certain algorithm to recover from the deadlock to a previous safe state. Example) Databases

4. **Ignorance or Ostrich's Algorithm**

System ignores the deadlock situation and pretend as if deadlock does not exist or it may never occur. Example) Most Operating Systems like Windows and Linux.

Deadlock Prevention & Avoidance

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock.

1. ***Mutual Exclusion***
   - If we want to violate mutual exclusion, we have to make atleast one resource sharable among processes i.e. it is to made to be used simultaneously by more than one processes.

   - But such a violation is not possible because non-sharable nature of a resource is its intrinsic property which cannot be changed, i.e. the resource which is non-sharable is due to its hardware configuration and hence cannot be made sharable.

2. ***Hold and Wait***
   - To violate hold and wait condition, we have to make sure that whenever a process requests a resource, it does not hold any other resources already. It can be violated using one of the following protocol:

   1. ***Conservative Approach****:* A process is allowed to start execution if and only if it has accquired all the requested resources. Hence, a process is allocated the resources it require before the start of execution. Although this approach seems to be easy to ensure deadlock independence, this solution is *practically not possible* due to dynamic nature of requesting the resources and hence inefficient.

   2. ***Do Not Hold****:* Process request any resource when it is not holding any other resource. Before a process can request any additional resources, it must release all the resources that it are currently allocated to it.

   3. ***Wait Timeouts***: We place a maximum time upto which a process can wait for resource(s), after which process will release all the resources which it already holds.

   - Hold and wait violation has following ***diadvantages***:
     1. *Resource utilization is low*, since resources may be allocated but unused for a long period. For example, a process may be allocated a mutex lock for its entire execution, yet only require it for a short duration.
     2. *Starvation* is possible. A process that needs many 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***

   - One protocol: If a process is holding some resources and requests another resource that cannot be immediately allocated to it, then all resources held by this process are preempted (implicitly released). The preempted resources are added to the list of resources for which the process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

   - Another protocol:
     - If a process requests some resources, first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources.
     - If so, we forcefully preempt the desired resources from the waiting process and allocate them to the requesting thread. The waiting process from which the resources are forcefully preempted is known as ***victim process***.
     - If the resources are neither available nor held by a waiting process, the requesting thread must wait. While it is waiting, some of its resources may be forcefully preempted by some other requesting process.
     - A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources if they were preempted while it was waiting.

   - This method of ***forceful preemption*** may be used by high priority process or system processes. Also, only the processes which are in waiting state should be selected the *victim process* and not the process in running state.

   - Forceful preemption is usually applied to resources whose state can be easily saved and restored later, such as CPU

registers and database transactions. It cannot be applied to resources such as mutex locks and semaphores, where deadlock occurs most commonly.

4. ***Circular Wait***
    - Circular Wait violation is the most practical and efficient approach for deadlock prevention. To violate circular wait condition, we give a natural number ordering to all the resource types available in the system, which allows us to compare two resources and to determine whether one precedes another in our ordering.

    - All processes are made to request resources only in the increasing (or decreasing) order of the resource type number. If a process requires a resource having a lesser number, then it must release all the resources having number higher than required number.

    - Both protocols need to be implemented to avoid circular wait, i.e. if only resources are ordererd with unique natural numbers and a process request resources in any order, then deadlock may occur.

Note: ***Disadvantages*** of using deadlock prevention technique are *low device utilization* and *reduced system throughput*.

Deadlock Avoidance

- If the system have prior information available about the resource requirements of each process, available resources and resources already allocated to the processes, it can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.

- Each process should declare the maximum number of each resource types that it *may* require during its execution. A deadlock avoidance algorithm dynamically examines the *resource-allocation state* to ensure that a circular-wait condition can never exist.

❋ **R*esource-allocation state*** is defined by the number of available and allocated resources and the maximum demands of the processes.

❋ ***Safe Sequence:*** A sequence of processes <P1, P2, ..., Pn> is a safe sequence for the current allocation state if, for each Pi, the requests that Pi will make can be satisfied by the currently available resources plus the resources held by all Pj, with j < i. In this situation, if the resources that Pi needs are not immediately available, then Pi can wait until all Pj have finished. When they have finished, Pi can obtain all of its needed resources, complete its execution, return its allocated resources, and terminate. When Pi terminates, Pi+1 can obtain its needed resources, and so on.

❋ ***Safe State:*** A system is in a *safe state* only if there exists atleast one safe sequence. A state is hence said to be safe, if the system can allocate resources to each process (upto it's maximum need) in some order and still avoid a deadlock. A safe state can never lead to deadlock or deadlock can never happen when state of the system is safe.

❋ ***Unsafe State:*** On contrary, If no safe sequence exists, then the system is said to be in *unsafe state*. If the system is in unsafe state, then deadlock may or may not occur (as we consider the *maximum* need of resources in worst case by a process and worst case scenario of resource requirement may not occur). However, if a deadlock has occurred, then the system must have been in unsafe state.

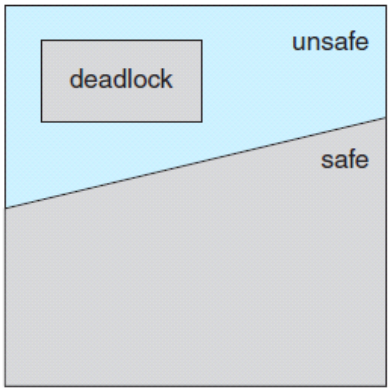


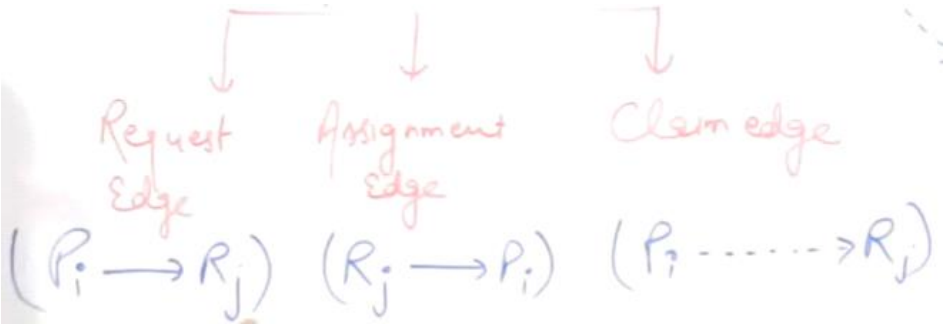
Figure 8.8 Safe, unsafe, and deadlocked state spaces.

- Initially, the system is in a safe state. Whenever a process requests a resource that is currently available, the system must decide whether the resource can be allocated immediately or the thread must wait. The request is granted only if the allocation leaves the system in a safe state.

- Since we consider maximum need of resource (which it may never request), we can get unsafe state and have to make the process wait, which leads to lower resource utilization than expected.

Resource Allocation Graph Algorithm

- This algorihtm uses a variation of Resource Allocation Graph and can only be applied when there is only *one instance of each resource type* in the system and cannot be applied when any one resource type has multiple instances.
- In addition to *request edge* and *assignment edge*, this variation introduces third type of edge known as ***claim edge***. A claim edge Pi → Rj indicates that process Pi may request resource Rj at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a *dashed line*.
- When process Pi requests resource Rj, the claim edge Pi → Rj is converted to a request edge. Similarly, when a resource Rj is released by Pi, the assignment edge Rj → Pi is reconverted to a claim edge Ti → Rj.
- A request by process Pi to accquire resource Rj can be granted only if converting the request edge Pi → Rj to an assignment edge Rj → Pi does not result in the formation of a cycle in the resource-allocation graph.
- *Note*: Cycle Detection Algorithm runs in O(N^2) Time where N is number of vertices (Number of Processes + Number of Resource Types in the system).
- 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.



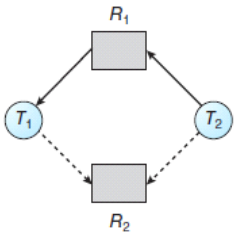


Figure 8.9 Resource-allocation graph for deadlock avoidance.

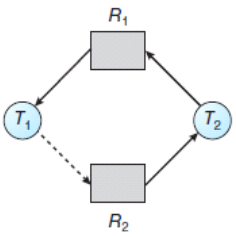


Figure 8.10 An unsafe state in a resource-allocation graph.

Banker's Algorithm

Idea

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number should not exceed the total number of resources in the system. When a user 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.

Data Structures

N = Number of Processes in the System
M = Number of Resource types in the System

1. **int Available[M]:** This vector denotes the number of available resources of each type. If Available[i] = k for some i in [1, M], it denotes that k instance of resource type Ri are currently available in the system.

2. **int Max[N][M]:** This matrix defines the maximum need of each process. If Max[i][j] = k for some i in [1,N] and j in [1, M], it denotes that Process Pi has may need a maximum of k instances of resource type Rj during its entire execution.

3. **int Allocation[N][M]:** This matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] = k for some i in [1,N] and j in [1, M], it denotes that Process Pi is currently allocated k instance of resource type Rj. The row vector Allocation[i] (or $Allocation_i$) represents resources allocated to Process Pi.

4. **int Need[N][M]:** This matrix defines the remaining (or current) resources need of each process. If Need[i][j] = k for some i in [1,N] and j in [1, M], it denotes that Process Pi may need k more instances of resource type Rj to complete it's execution. The row vector Need[i] (or $Need_i$) represents resources which may be needed by Process Pi in future.
   Note: Need matrix can be written as *Current Need* or *Request* or *Demand* or *Requirement* matrix.

   $$Need[i][j] = Max[i][j] - Allocation[i][j].$$

8.6.3.2 Resource-Request Algorithm

Next, we describe the algorithm for determining whether requests can be safely granted. Let $Request_i$ be the request vector for thread T_i. If $Request_i[j] == k$, then thread T_i wants k instances of resource type R_j. When a request for resources is made by thread T_i, the following actions are taken:

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

$$Available = Available - Request_i$$
$$Allocation_i = Allocation_i + Request_i$$
$$Need_i = Need_i - Request_i$$

If the resulting resource-allocation state is safe, the transaction is completed, and thread T_i is allocated its resources. However, if the new state is unsafe, then T_i must wait for $Request_i$, and the old resource-allocation state is restored.

8.6.3.1 Safety Algorithm

We can now present the algorithm for finding out whether or not a system is in a safe state. This algorithm 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 $i = 0, 1, ..., n - 1$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Need_i \leq Work$

   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 state.

This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Deadlock Detection And Recovery

If there is no deadlock prevention or avoidance method implemented in the system, then deadlock may occur in the system. Hence to remove deadlock, devise two algorithms:

1. *Detection of Deadlock*: Algorithm will examine the current state of the system and find if the system is in deadlock or not.

2. *Recovery from Deadlock*: Once deadlock is detected, the algorithm is implemented to recover the system from deadlock to either a previous safe state or restart from beginning.

Note: ***Disadvantage***: Detection and recovery scheme requires overhead that includes not only the run-time costs of maintaining the necessary information and executing the detection algorithm but also the potential losses inherent in recovering from a deadlock.

Deadlock Detection

1. ***Wait - For Graph***
   - 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. It cannot be applied to detect deadlock for system if any resource type has multiple instances.
   - Wait-For graph is obtained from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges. If there is a request edge Pi -> Rj and an allocation edge Rj -> Pk, then resource Rj have to be removed in wait for graph and an edge from process Pi to Pk (Pi -> Pk) is drawn which depicts Pi is waiting for the process Pk to release a resource that Pi needs for its execution.
   - 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 invoke an algorithm* that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires O(N^2) operations, where N is the number of vertices(processes) in the graph.

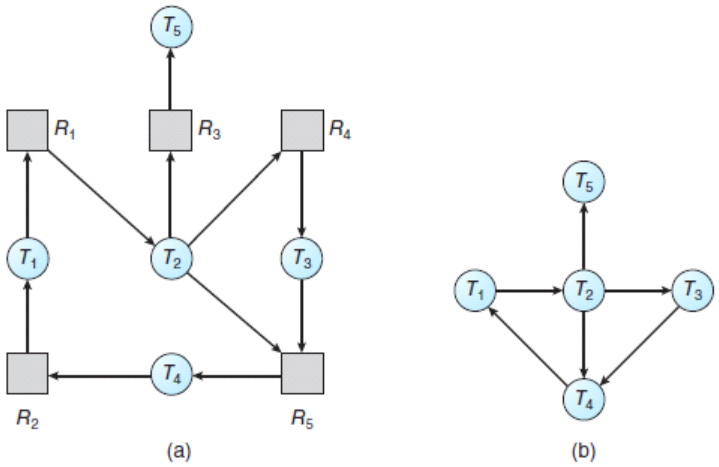


Figure 8.11 (a) Resource-allocation graph. (b) Corresponding wait-for graph.

2. ***Banker's Algorithm***

   The banker's detection algorithm simply investigates every possible allocation sequence for the processes that remain to be completed and check whether deadlock can occur in any of the sequence.

   1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
   2. Find an index i such that both
      a. $Finish[i] == false$
      b. $Request_i \leq Work$

      If no such i exists, go to step 4

1. Let *Work* and *Finish* be vectors of length m and n, respectively. Initialize $Work = Available$. For $i = 0, 1, ..., n-1$, if $Allocation_i \neq 0$, then $Finish[i] = false$. Otherwise, $Finish[i] = true$.
2. Find an index i such that both
   a. $Finish[i] == false$
   b. $Request_i \leq Work$

   If no such i exists, go to step 4.
3. $Work = Work + Allocation_i$
   $Finish[i] = true$
   Go to step 2.
4. If $Finish[i] == false$ for some $i, 0 \leq i < n$, then the system is in a deadlocked state. Moreover, if $Finish[i] == false$, then thread T_i is deadlocked.

This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state.

Q) When should be invoke deadlock detection algorithm ?

1. If deadlocks occur frequently, then the detection algorithm should be invoked frequently. Resources allocated to deadlocked processes will be idle until the deadlock can be broken. In addition, the number of processes involved in the deadlock cycle may grow.
2. Deadlocks occur only when some thread makes a request that cannot be granted immediately. Hence in *extreme*,we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately. In this case, we can identify not only the deadlocked set of processes but also the specific thread which caused the deadlock. However, invoking the deadlock-detection algorithm for every resource request will incur considerable overhead in computation time.
3. A less expensive alternative is simply to invoke the algorithm at defined intervals (eg once per hour) or whenever CPU utilization drops below 40 percent. If the detection algorithm is invoked at arbitrary points in time, the resource graph may contain many cycles and it will be very difficult to find the process which caused the deadlock (at first).

Deadlock Recovery

Either system can recover from the deadlock *automatically* by applying certain deadlock recovery algorithm or we can report that deadlock has occurred in the system to the computer system operator/user and let him deal with the situation manually. If deadlock does occur, a system can attempt to recover from the deadlock by either one of the following:

1. ***Process Termination:*** Aborting one or more of the processes in the circular wait to break circular wait/cycle.

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

   a. *Abort all deadlocked processes*. However, this will be very expensive task as many processes might be on verge of their termination before going in deadlock. Aborting all of them will delete all partial computations and hence they have to be computed again from start.

   b. *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.

   Note: ***Disadvantage***: Aborting a process is not an easy task. If the process was in the midst of updating a file, terminating it may leave that file in an incorrect state. Similarly, if a process working in critical section(on sharable resources) gets aborted, then it may lead to race condition and make data inconsistent.

Q) If processes are aborted one by one, then how to select which process should be aborted before others?

R) We should abort those processes whose termination will incur the *minimum cost*. It depends on many factors like:

a. Process should be of lower priority than others.
b. Process should not have been computed for long time and it's remaining execution time on CPU should be higher than other processes.

c. Process to be aborted should have accquired those resources which are easy to forcefully preempt.
d. Process to be aborted should be waiting for more resources to complete its execution than others.

2. ***Resource Preemption:*** Preempting some resources that have been assigned to one or more deadlocked process(es).

   To eliminate deadlocks using resource preemption, we successively preempt some resources fromprocesses and give these resources to other processes until the deadlock cycle is broken. This method requires to tackle the following issues:

   a. Victim Selection: Processes whose resources will be preempted, resources which will be preempted and order in which resources to be preempted from resource should be selected in order to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed.

   b. Rollback: Since victim process cannot continue normal execution, we must roll back the process to some previous safe state and restart it from that state. Although, it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes. Since, determing the previous safe state is difficult, simple solution can be total rollback i.e. aborting the process and restarting it.

   c. Starvation: In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its execution, and the victim process may go into starvation. Hence, we must ensure that a process should be picked as a victim only a fewer number of times. The most common solution is to include the number of rollbacks in the cost factor.

Deadlock Ignorance (Ostrich's Algorithm)

Disadvantage: In the absence of algorithms to detect and recover from deadlocks, we may arrive at a situation in which the system is in a deadlocked state yet has no way of recognizing what has happened. In this case, the undetected deadlock will cause the system's performance to deteriorate, because resources are being held by threads that cannot run and because more and more threads, as they make requests for resources, will enter a deadlocked state. Eventually, the system will stop functioning and will need to be restarted manually.

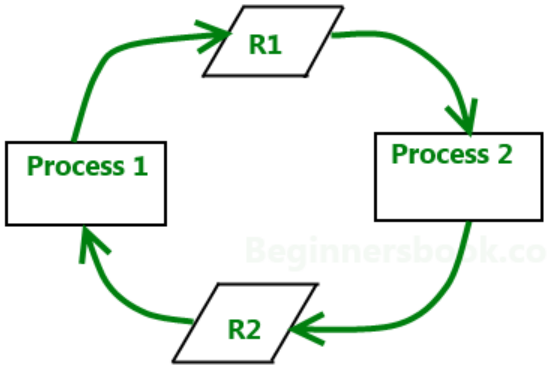
Q) Even though deadlock situation can be very disadvantageous for the system, deadlock ignorance is used in most operating systems like Linux and Windows. Why ?

1. ***Expense***: Ignoring the possibility of deadlocks is cheaper than the other approaches. Since in many systems, deadlocks occur infrequently (say, once per month), the extra expense of the other methods may not seem worthwhile.

2. ***Performance:*** If the operating system has a deadlock prevention or detection system in place, this will have a negative impact on performance (slow the system down) because whenever a process or thread requests a resource, the system will have to check whether granting this request could cause a potential deadlock situation.

3. ***Solving Livelock:*** Methods used to recover from other liveness conditions, such as ***livelock***, may be used to recover from deadlock. In some circumstances, a system is suffering from a liveness failure but is not in a deadlocked state.
   For eg) a real-time thread running at the highest priority may never return control to the operating system. The system must have manual recovery methods for such conditions and may simply use those techniques for deadlock recovery.

Livelock: Livelock is a variant of deadlock which is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.
These processes are not in the waiting state, and they are running concurrently. This is different from a deadlock because in a deadlock all processes are in the waiting state.



Process P1 holds resource R2 and requires R1 while Process P2 holds resource R1 and requires R2.

Address Binding

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 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.

Basic Hardware

- CPU can directly access only the main memory (RAM + ROM) and the registers (built into each processing core). It cannot access any other general-purpose storage like secondary memory (Hard Disk). Machine instructions always takes memory addresses as arguments and not disk addresses. Therefore, data which is stored in secondary memory and will be used by instructions in execution should be brought to *direct-access storage* (main memory or registers) before CPU executes the instructions.

- Registers are accessible within one cycle of CPU clock but completing main memory access requires many cycles of CPU clock, which leads the processor to *stall*, since it does not have the data required to complete the instruction that it is executing. To reduce this situation, we can add fast memory between the CPU and main memory (typically on the CPU chip) for fast access and this fast but expensive memory is known as *cache memory*.

- For proper system operation, we must protect the operating system from access by user processes, as well as protect user processes fromone another. This protection must be provided by the hardware, because the operating system doesn't intervene between the CPU and its memory accesses.

Base Register Scheme

- Each process must have separate memory space because processes should be protected from each other and to implement multi-core programming, they need to executed concurrently and loaded in memory. To separate memory spaces for processes, there should be range of legal addresses so that any access outside this range is treated illegal so that process can access only it's own memory space. To define range of legal addresses for a process, two registers namely, *base register* and *limit register* are used.

- The ***base register*** holds the smallest legal physical memory address; the ***limit register*** specifies the size of the range. For example, if the base register holds 300040 and the limit register is 120900, then the program can legally access all addresses from 300040 through 420939 (inclusive).

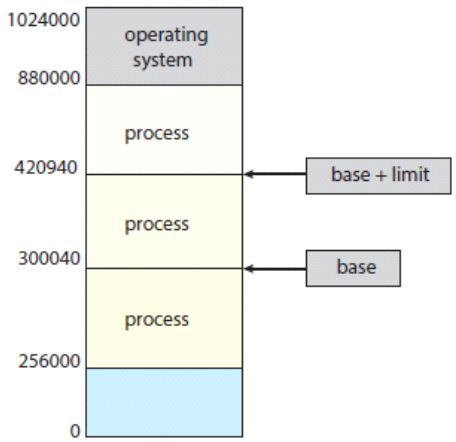


Figure 9.1 A base and a limit register define a logical address space.

- CPU will compare every address generated in user mode and check if access is legal or not. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a *trap* to the operating system, which treats the attempt as a fatal error. This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.

- Content of base and limit registers can only be changed by privileged instructions. Since, privileged instructions are available in kernel mode, registers can be modified by only operating system and not any user process. Hence operating system is given access to load users' programs into users' memory, to dump out those programs in case of errors, to access and modify parameters of system calls, to perform I/O to and from user memory, and to provide many other services.

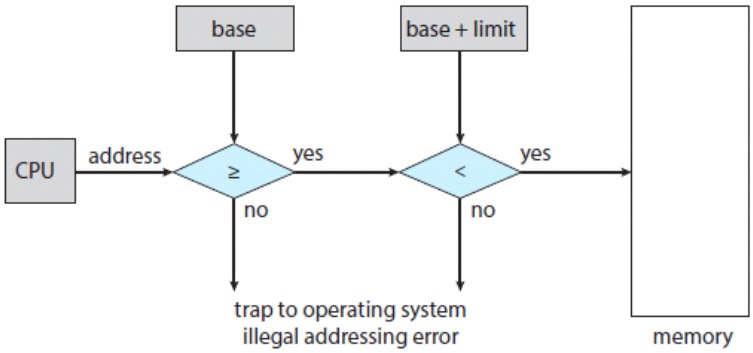


Figure 9.2 Hardware address protection with base and limit registers.

Address Binding

A user process goes through several steps before it's execution. Addresses may be represented in different ways during these steps. Addresses in the source program are called ***symbolic addresses*** (such as variables). A compiler typically binds these symbolic addresses to ***relocatable addresses*** (such as "14 bytes from the beginning of this module"). The linker or loader in turn binds the relocatable addresses to ***absolute addresses*** (such as 74014). Each binding is a mapping from one address space to another.

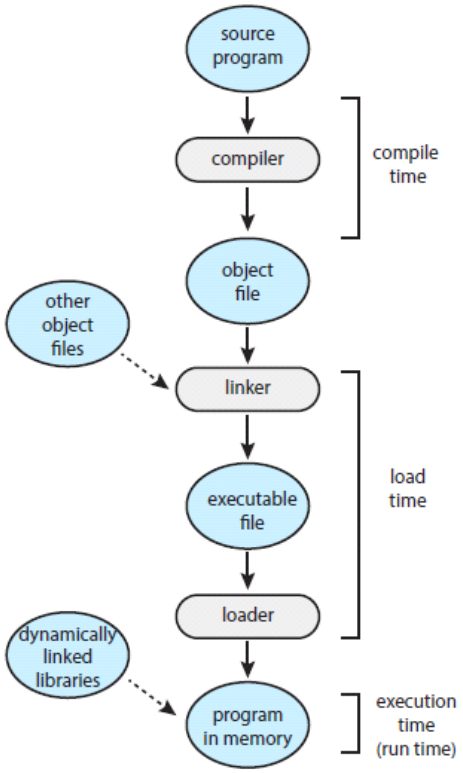


Figure 9.3 Multistep processing of a user program.

Binding of instructions and data to memory addresses can be done at any step given below:

1. *Compile Time*
   If it is known where the code will reside in memory at compile time, then *absolute code* can be generated. For example, if it is known that a user process will reside starting at location R, then the generated compiler code will start at that location and

extend up from there. If, at some later time, the starting location changes, then it will be necessary to recompile this code.

2. *Load Time*
   If it is not known at compile time where the process will reside in memory, then the compiler must generate *relocatable code*. In this case, final binding is delayed until load time. If the starting address changes, we only need to reload the user code to incorporate this changed value.

3. *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 scheme to work and most operating system use this method.

Logical vs Physical Address Space

- Binding addresses at either compile or load time generates identical logical and physical addresses. However, the execution-time address-binding scheme results in different logical and physical addresses.

- An address generated by the CPU is commonly referred to as a ***logical address.*** In execution-time address-binding scheme, logical address is also referred as ***virtual address***. An address seen by the memory unit i.e. loaded into the memory-address register of the memory is commonly referred to as a ***physical address***.

- The set of all logical addresses generated by a program is a ***logical address space***, whereas the set of all physical addresses corresponding to these logical addresses is a ***physical address space***. In execution-time addressing-binding scheme, these two spaces differ.

- The run-time mapping from virtual to physical addresses is done by a hardware device called the ***memory-management unit (MMU)***. There are various methods of mapping and simplest of them is the generalization of base-register scheme.

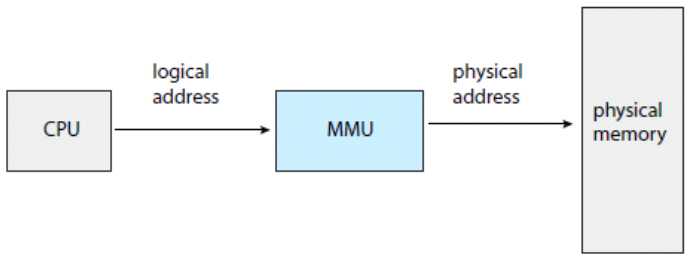


Figure 9.4 Memory management unit (MMU).

- The base register is now called a ***relocation register***. The value in the relocation register is added to every address generated by a user process at the time the address is sent to memory. If the value of base register is R, then the logical address will range from 0 to *max* and the physical address will range from R to R + *max*.

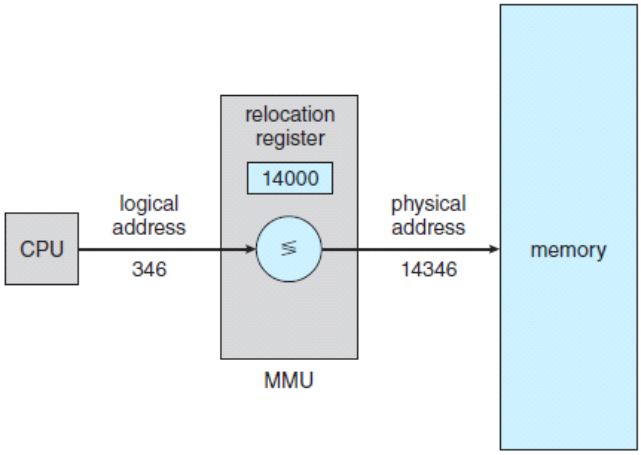


Figure 9.5 Dynamic relocation using a relocation register.

- A user program never access the real physical address and always deals with virtual/logical addresses because if it gets access

to real address, it can store it in a pointer, manipulate it and security of data comes in danger.

Differences Between Logical Address and Physical Address

Paramenter	LOGICAL ADDRESS	PHYSICAL ADDRESS
Basic	generated by CPU	location in a memory unit
Address Space	Logical Address Space is set of all logical addresses generated by CPU in reference to a program.	Physical Address is set of all physical addresses mapped to the corresponding logical addresses.
Visibility	User can view the logical address of a program.	User can never view physical address of program.
Generation	generated by the CPU	Computed by MMU
Access	The user can use the logical address to access the physical address.	The user can indirectly access physical address but not directly.

Dynamic Linking & Loading, Swapping

Dynamic Loading

- Dynamic loading is used to obtain better memory utilization. With dynamic loading, a routine is not loaded until it is called. All routines are kept on disk in a relocatable load 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. If it has not, the relocatable linking loader is called to load the desired routine into memory and to update the program's address tables to reflect this change. Then control is passed to the newly loaded routine.

- *Advantage*: Routine is loaded only when it is needed. This method is particularly useful when large amounts of code are needed to handle infrequently occurring cases, such as error routines. In such a situation, although the total program size may be large, the portion that is used (and hence loaded) may be much smaller.

- Dynamic loading does not require special support from the operating system. However, operating system may help the programmer by providing library routines to implement dynamic loading.

Dynamic Linking

- ***Static linking***: System libraries are treated like any other object module and are combined by the loader into the binary program image. Each program on a system must include a copy of its language library in the executable code. This requirement not only increases the size of an executable code but also may waste main memory.

- ***Dynamically linked libraries (DLLs)*** are system libraries that are linked to user programs when the programs are run. Eg) standard C language library. These libraries can be shared among multiple processes, so that only one instance of the DLL in main memory is required. Hence, they are also called ***shared libraries*** and used in various operating systems like Linux and Windows.

- ***Dynamic Linking:*** Linking of system libraries (DLLs) to user program is postponed until execution time. When a program references a routine that is in a dynamic library, the loader locates the DLL, loading it into memory if necessary. It then adjusts addresses that reference functions in the dynamic library to the location in memorywhere the DLL is stored.

- *Advantage: Library updates such as bug fixes:* A library may be replaced by a new version, and all programs that reference the library will automatically use the new version. Without dynamic linking, all such programs would need to be relinked to gain
- access to the new library. To make programs not accidentally execute new, incompatible versions of libraries, version information is included in both the program and the library. More than one version of a library may be loaded into memory, and each program uses its version information to decide which copy of the library to use. Thus, only programs that are compiled with the new library version are affected by any incompatible changes incorporated in it. Other programs linked before the new library was installed will continue using the older library.

- Unlike dynamic loading, dynamic linking and shared libraries generally require help from the operating system. If the processes in memory are protected from one another, then the operating system is the only entity that can check to see whether the needed routine is in another process's memory space or that can allow multiple processes to access the same memory addresses.

Difference Between Static vs Dynamic Linking & Loading

Static	Dynamic
• Loading the entire program into the main memory before start of the program execution is called as static loading.	• Loading the program into the main memory on demand is called as dynamic loading.
• Inefficent utilization of memory because whether it is required or not required entire program is brought into the main memory.	• Efficent utilization of memory.
• Program execution will be faster.	• Program execution will be slower.
• Statically linked program takes constant load time every time it is loaded into the memory for execution.	• Dynamic linking is performed at run time by the operating system.
• If the static loading is used then accordingly static linking is applied.	• If the dynamic loading is used then accordingly dynamic linking is applied.

- Static linking is performed by programs called linkers as the last step in compiling a program. Linkers are also called link editors.
- In static linking if any of the external programs has changed then they have to be recompiled and re-linked again else the changes won't reflect in existing executable file.
- In dynamic linking this is not the case and individual shared modules can be updated and recompiled.This is one of the greatest advantages dynamic linking offers.
- In dynamic linking load time might be reduced if the shared library code is already present in memory.

Swapping

- Process instructions and the data they operate on must be in memory to be executed. However, a process, or a portion of a process, can be swapped temporarily out of memory to a ***backing store*** and then brought back into memory for continued execution.
- The backing store is commonly fast secondary storage. It must be large enough to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images.
- When a process or part is swapped to the backing store, the data structures associated with the process must be written to the backing store. The operating system must also maintain metadata for processes that have been swapped out, so they can be restored when they are swapped back in to memory.
- Idle processes are good candidates for swapping. Any memory that has been allocated to these inactive processes can then be dedicated to active processes. If an inactive process that has been swapped out becomes active once again, it must then be swapped back in.
- *Disadvantage*: The total time taken by swapping process includes the time it takes to move the entire process to a secondary disk and then to copy the process back to memory, as well as the time the process takes to regain main memory. Thus swapping involves high overhead of context-swithcing time.
- *Advantage*: Swapping makes it possible for the total physical address space of all processes to exceed the real physical memory of the system, thus increasing the degree of multiprogramming in a system. Although the process of swapping affects the performance of the system, it helps to run larger and more than one process. This is the reason why swapping is also referred to as ***memory compaction***.

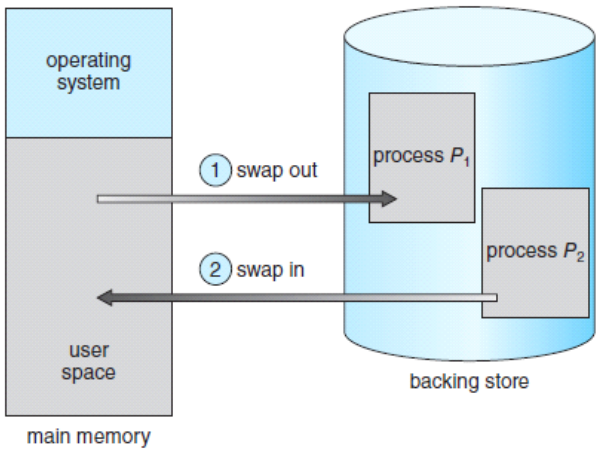


Figure 9.19 Standard swapping of two processes using a disk as a backing store.

Contiguous Memory Allocation

- In *contiguous memory allocation*, each process is contained in a single section of memory that is contiguous to the section containing the next process.

- The operating system keeps a table indicating which parts of memory are available and which are occupied. Initially, all memory is available for user processes and is considered one large block of available memory, a hole.

- As processes enter the system, the operating system takes into account the memory requirements of each process and the amount of available memory space in determining which processes are allocated memory.

- When a process is allocated space, it is loaded into memory, where it can then compete for CPU time. When a process terminates, it releases its memory, which the operating system may then provide to another process.

- When sufficient memory is not available for the arriving processes, they are placed into a wait queue. When memory is later released, the operating system checks the wait queue to determine if it will satisfy the memory demands of a waiting process.

Note: The memory is usually divided into two partitions: one for the operating system and one for the user processes. We can place the operating system in either low memory addresses or high memory addresses. Many operating systems (including Linux and Windows) place the operating system in high memory.

Memory Protection

- It is accomplished using the *relocation register* which contains the value of the smallest physical address and the *limit register* which contains the range of logical addresses. 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, and this mapping is sent to the memory.

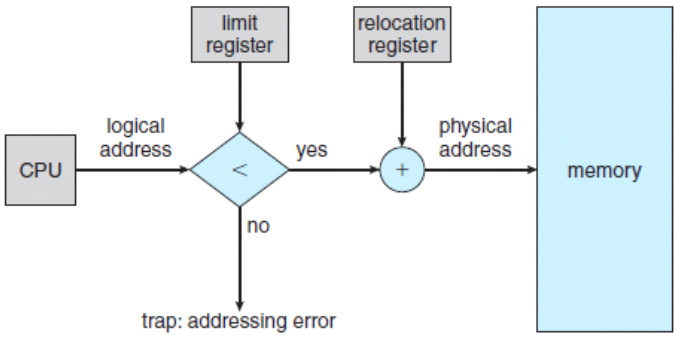


Figure 9.6 Hardware support for relocation and limit registers.

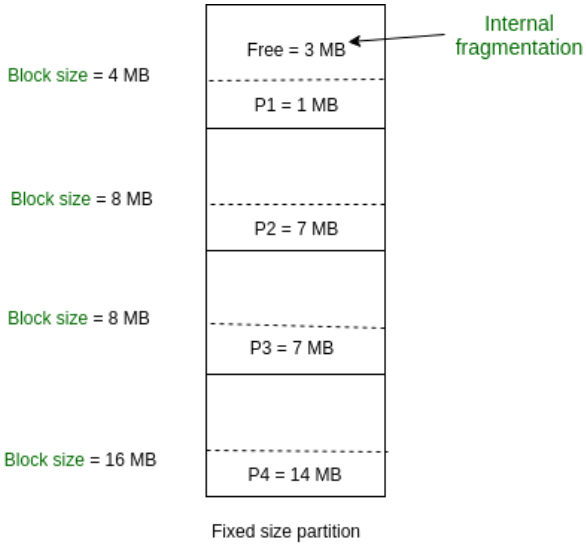
- When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by a CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process.

- This *relocation-register scheme* provides flexibility by allowing operating system's size to change dynamically. For example, operating system contains code and buffer space for device drivers. A device driver can be loaded into memory only when it is needed and can be removed when no longer needed and it's memory can be allocated for other needs.

Fixed Size Partitioning

- In this partitioning, main memory is divided into several fixed-sized partitions. Although the number of partitions is fixed and each partition is of fixed size, the size of all partitions may not be same. It is one of the simplest and oldest methods for allocating memory. It was used in batch operating systems and not used now-a-days.

- Each partition may contain exactly one process. Thus, the degree of multiprogramming is bound by the number of partitions. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.



Fixed size partition

Advantages of Fixed Partitioning

1. **Easy to implement:**
   Algorithms needed to implement Fixed Partitioning are easy to implement. It simply requires putting a process into certain partition without focussing on the emergence of Internal and External Fragmentation.

2. **Little OS overhead:**
   Processing of Fixed Partitioning require lesser excess and indirect computational power.

Disadvantages of Fixed Partitioning

1. **Internal Fragmentation:**
   Main memory use is inefficient. Any program, no matter how small, occupies an entire partition. This can cause internal fragmentation.

2. **External Fragmentation:**
   The total unused space (as stated above) of various partitions cannot be used to load the processes even though there is space available but not in the contiguous form (as spanning is not allowed).

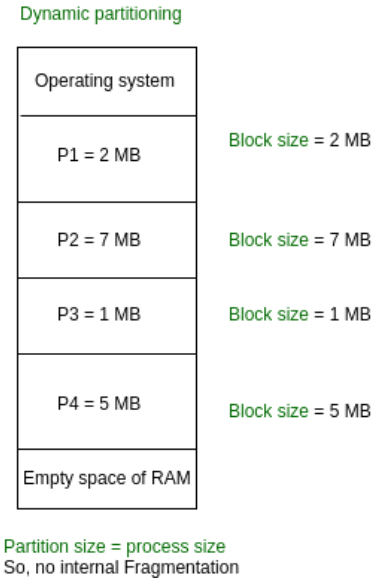
3. **Limit process size:**
   Process of size greater than size of partition in Main Memory cannot be accommodated. Partition size cannot be varied according to the size of incoming process's size.

4. **Limitation on Degree of Multiprogramming:**
   Partition in Main Memory are made before execution or during system configure. Main Memory is divided into fixed number of partition. Number of processes greater than number of partitions in RAM is invalid in Fixed Partitioning.

Variable Size Partitioning

The memory blocks available comprise a set of *holes* of various sizes scattered throughout memory. When a process arrives and needs memory, the system searches the set for a hole that is large enough for this process. If the hole is too large, it is split into two parts. One part is allocated to the arriving process; the other is returned to the set of holes. When a process terminates, it releases its block of memory, which is then placed back in the set of holes. If the new hole is adjacent to other holes, these adjacent holes are merged to form one larger hole.

- Initially RAM is empty and partitions are made during the run-time according to process's need instead of partitioning during system configure.
- The size of partition will be equal to incoming process. The partition size varies according to the need of the process so that the internal fragmentation can be avoided to ensure efficient utilisation of RAM.
- Number of partitions in RAM is not fixed and depends on the number of incoming process and Main Memory's size.



Advantages of Variable Partitioning

1. **No Internal Fragmentation:**
   In variable Partitioning, space in main memory is allocated strictly according to the need of process, hence there is no case of internal fragmentation. There will be no unused space left in the partition.

2. **No restriction on Degree of Multiprogramming:**
   More number of processes can be accommodated due to absence of internal fragmentation. A process can be loaded until the memory is empty.

3. **No Limitation on the size of the process:**
   In Fixed partitioning, the process with the size greater than the size of the largest partition could not be loaded and process can not be divided as it is invalid in contiguous allocation technique. Here, In variable partitioning, the process size can't be restricted since the partition size is decided according to the process size.

Disadvantages of Variable Partitioning

1. **Difficult Implementation:**
   Implementing variable Partitioning is difficult as compared to Fixed Partitioning as it involves allocation of memory during run-time rather than during system configure.

2. **External Fragmentation:**
   There will be external fragmentation inspite of absence of internal fragmentation.
   The empty space in memory cannot be allocated when there is no hole greater than or equal to the size of process. Since, there are holes of smaller size whose size when added becomes greater than or equal to size of process and still the process will have to wait for memory, it results in External Fragmentation.

Differences B/W Fixed Partitioning and Variable Partitioning

	Fixed partitioning	**Variable partitioning**
1.	In multi-programming with fixed partitioning the main memory is divided into fixed sized partitions.	In multi-programming with variable partitioning the main memory is not divided into fixed sized partitions.
2.	Only one process can be placed in a partition.	In variable partitioning, the process is allocated a chunk of free memory.
3.	It does not utilize the main memory effectively.	It utilizes the main memory effictively.
4.	There is presence of internal fragmentation and external fragmentation.	There is external fragmentation.
5.	Degree of multi-programming is less.	Degree of multi-programming is higher.

6.	It is more easier to implement.	It is less easier to implement.
7.	There is limitation on size of process.	There is no limitation on size of process.

There are various strategies of contiguous memory allocation, i.e. how a process of memory requirement of given size should be given memory from the list of available holes.

1. ***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.

2. ***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.

3. ***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.

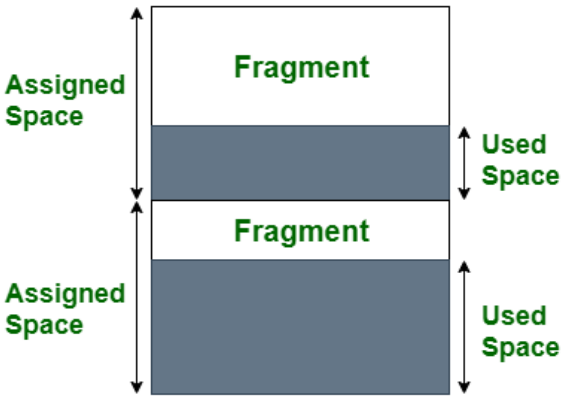
Note:

- For fixed-size partitioning, best-fit proves to be better strategy than worst-fit since there will be less internal fragmentation, while for variable-size partitioning, worst-fit proves to be better strategy than best-fit.

- Both best-fit and worst-fit are slower strategies as they check entire list of holes to find the largest/smallest hole when compared to first-fit which allocates nearest available memory space/hole.

Internal Fragmentation

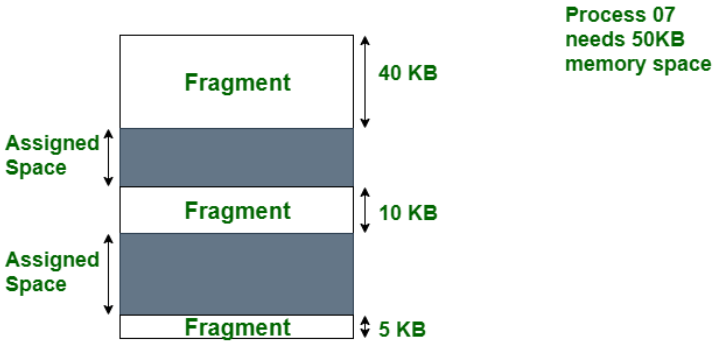
Internal fragmentation happens when the memory is split into fixed size partitions. Whenever a process request for the memory, the suitable partition is allotted to the process. If the memory allotted to the process is larger than the memory requested, then the memory with size equal to difference between allotted and requested memory gets wasted. It occurs minimum for best-fit strategy. Solution to resolve internal fragmentation is variable-sized partitioning.



Internal Fragmentation

External Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. External fragmentation exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous: storage is fragmented into a large number of small holes. It occurs minimum for worst-fit strategy.



Solution to resolve external fragmentation can be:

1. ***Compaction:***
   - The goal is to shuffle the memory contents so as to place all free memory together in one large block.
   - It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address.
   - The simplest compaction algorithm is to move all processes toward one end of memory; all holes move in the other direction, producing one large hole of available memory. This scheme can be expensive.

2. ***Non Contiguous Memory Allocation***
   In the *non-contiguous memory allocation* the available free memory space are scattered and all the free memory space is not at one place. A process will acquire the memory space which it is not at one place but at the different locations according to the process requirement. This technique of reduces the wastage of memory hence resolving internal and external fragmentation. It can be implemented using either ***Paging*** or ***Segmentation***.

Difference between Internal fragmentation and External fragmentation

	Internal fragmentation	***External fragmentation***
1.	In internal fragmentation fixed-sized memory, blocks square measure appointed to process.	In external fragmentation, variable-sized memory blocks square measure appointed to method.
2.	Internal fragmentation happens when the method or process is larger than the memory.	External fragmentation happens when the method or process is removed.
3.	The solution of internal fragmentation is best-fit block.	Solution of external fragmentation is compaction, paging and segmentation.
4.	Internal fragmentation occurs when memory is divided into fixed sized partitions.	External fragmentation occurs when memory is divided into variable size partitions based on the size of processes.
5.	The difference between memory allocated and required space or memory is called Internal fragmentation.	The unused spaces formed between non-contiguous memory fragments are too small to serve a new process, is called External fragmentation .

Paging

- Paging involves breaking physical memory (main memory) into fixed-sized blocks called ***frames*** and breaking logical memory (secondary storage) into blocks of the same size called ***pages***. When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store).

- Paging is a form of *dynamic relocation* as every logical address is bound by the paging hardware to some physical address. Using paging is similar to using a table of base (or relocation) registers, one for each frame of memory.

- Every address generated by the CPU is divided into two parts: a ***page number (p)*** and a ***page offset (d)*** (displacement within the page). The page number is used as an index into a ***page table***. The page table contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address.

- When a process arrives in the system to be executed, its size (expressed in pages) is examined. Each page of the process needs one frame. Thus, if the process requires n pages, at least n frames must be available in memory. If n frames are available, they are allocated to this arriving process. The first page of the process is loaded into one of the allocated frames, and the frame number is put in the page table for this process. The next page is loaded into another frame, its frame number is put into the page table, and so on.

- Since the operating system is managing physical memory, it must be aware of the allocation details of physical memory—which frames are allocated, which frames are available, how many total frames there are, and so on. This information is kept in a single, system-wide data structure called a ***frame table***. The frame table has one entry for each physical page frame, indicating whether it is free or allocated to some process.

- The operating system maintains a copy of the page table for each process, just as it maintains a copy of the instruction counter and register contents. This copy is used to translate logical addresses to physical addresses whenever the operating system maps a logical address to a physical address manually. It is also used by the CPU dispatcher to define the hardware page table when a process is to be allocated the CPU. Paging therefore increases the context-switch time.

- Page table is generated for each process separately, thus a pointer to the page table is stored with the other register values in the process control block (PCB) of each process. This pointer is known as ***page-table base register (PTBR)***. When the CPU scheduler selects a process for execution, it must reload the user registers and the appropriate hardware page-table values from the stored user page table. Page table is stored in main memory, which reduces context-switching time.

Address generated by CPU is divided into

- **Page number(p):** Number of bits required to represent the pages in Logical Address Space or Page number
- **Page offset(d):** Number of bits required to represent particular word in a page or page size of Logical Address Space or word number of a page or page offset.

Physical Address is divided into

- **Frame number(f):** Number of bits required to represent the frame of Physical Address Space or Frame number.
- **Frame offset(d):** Number of bits required to represent particular word in a frame or frame size of Physical Address Space or word number of a frame or frame offset.

Steps taken by MMU to convert a logical address generated by CPU to a physical address are:

1. Extract the page number p and use it as an index into the page table.
2. Extract the corresponding frame number f fromthe page table.
3. Replace the page number p in the logical address with the ***frame number f*** . As the offset d does not change, it is not replaced, and the frame number and offset now comprise the physical address.

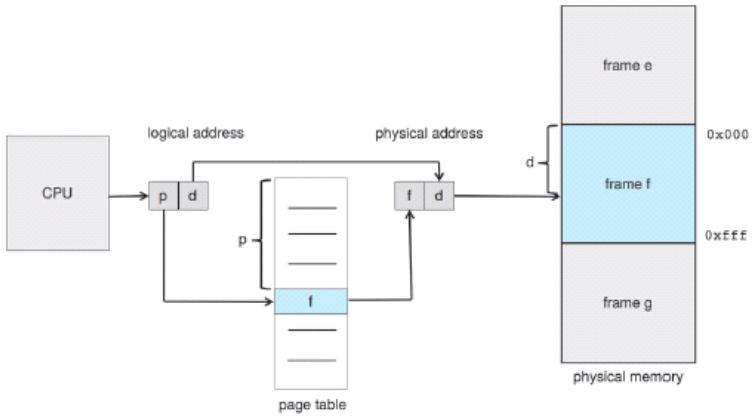


Figure 9.8 Paging hardware.

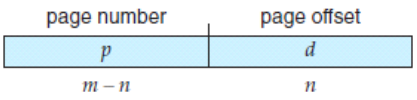
Advantage:

A) The logical address space is now totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

B) There is no external fragmentation possible in paging as it is an non-contiguous memory allocation technique. Also, there can be internal fragmentation upto just 1 frame getting wasted per process as in worst case, a process can need memory of size = memory of n pages + 1 byte. To allocate 1 extra byte, entire frame/page has to be allocated. Hence, although there is internal fragmentation, but very negligible when compared to contiguous memory allocation.

Page Size

The page size (and the frame size) is defined by the hardware. The size of a page is selected as a power of 2 to make the translation of a logical address into a page number and page offset easy. If the size of the logical address space is 2^m, and a page size is 2^n bytes, then the page number will be m−n bits, and the page offset will be of n bits size.
Thus, the physical/logical address is as follows:



If we will increase page size, then since one process can cause internal fragmentation of atmost 1 frame, then more memory will be wasted. Thus *smaller page size will be desirable*. But overhead is involved in each page table entry, and this overhead is reduced as the size of the pages increases. Also, disk I/O is more efficient when the amount of data being transferred is larger. Hence *larger page size will be desirable*.

Disadvantage of using just Page Table

Although storing the page table in main memory can yield faster context switches, it may also result in slower memory access times. To access location *i*, we must first index into the page table, using the value in the PTBR offset by the page number for i. This task requires one memory access. It provides us with the frame number, which is combined with the page offset to produce the actual address.We can then access the desired place in memory. With this scheme, two memory accesses are needed to access data (one for the page-table entry and one for the actual data), slowing the memory access by factor of 2.

Translation Look Aside Buffer (TLB)

- Transaltion Look Aside Buffer is used to reduce the slower memory access using page tables. It is a special, small, fast-lookup hardware cache used for faster memory access.
- Each entry in the TLB consists of two parts: a key (or tag) and a value. When the associative memory is presented with an item, the item is compared with all keys simultaneously. If the item is found, the corresponding value field is returned.
- Searching for key-value pair is fast in TLB lookup is a part of instruction pipeline, which reduces performance penality to almost negligible.

Working:

A) The TLB contains only a few of the page-table entries. When a logical address is generated by the CPU, the MMU first checks if its page number is present in the TLB.
B) If the page number is found (known as *TLB hit*), its frame number is immediately available and is used to access memory.
C) If the page number is not in the TLB (known as a *TLB miss*), logical to physical memory address translation occurs by referencing the page table. When the frame number is obtained, we can use it to access memory and also we add the page number and frame number to the TLB, so that they will be found quickly on the next reference.
D) If the TLB is already full of entries, an existing entry must be selected for replacement. Replacement policies range from least recently used (LRU) to round-robin policy.
E) Every time a new page table is selected like during context-switch, the TLB must be flushed (erased) to ensure that the next executing process does not use the wrong translation information (page-frame pairs of previous process).

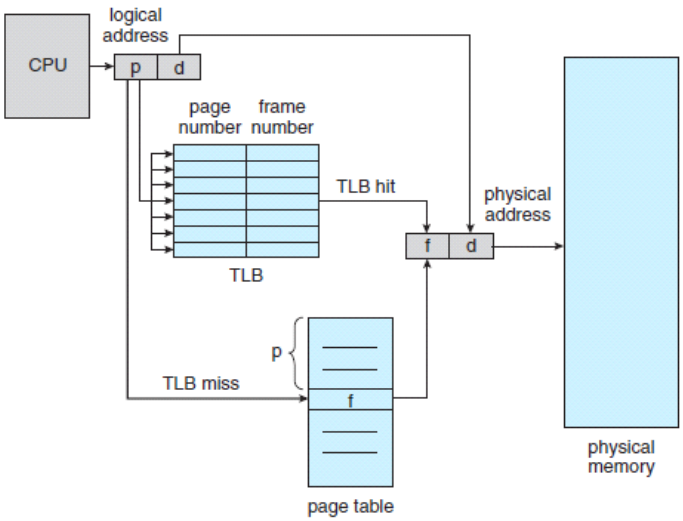


Figure 9.12 Paging hardware with TLB.

Note: Some TLBs allow certain entries to be *wired down*, meaning that they cannot be removed from the TLB during context-switch. Typically, TLB entries for key kernel code(operating system) are wired down.

TLB Ratio: The percentage of times that the page number of interest is found in the TLB is called the hit ratio.
If we know the TLB ratio, TLB lookup time and memory access time, then we can find the effective memory-access time when TLB is used.

Effective Access Time = (TLB Ratio) * (TLB Lookup + Memory Access) + (1 - TLB Ratio) * (TLB Lookup + 2 * Memory Access)

If there is TLB hit, then memory will be accessed only once (for physical address) while if there is TLB miss, memory access will be twice (one for accessing page table and other for accessing physical address of frame). There will be TLB lookup time in both cases.

This effective access time will be far better than access time required when no TLB would have been used because TLB ratio usually ranges from 90-95%. Thus hardware feature like TLB can have a significant effect on memory performance and operating-system improvements.

Protection

1. Memory protection in a paged environment is accomplished by ***protection bit*** associated with each frame, which are kept in the page table. One bit can define a page to be read–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.

2. One additional bit, ***valid-invalid bit***, may also be attached to each entry in the page table. When this bit is set to valid, the associated page is in the process's logical address space and is thus a legal (or valid) page. When the bit is set to invalid, the page is not in the process's logical address space. Illegal addresses are trapped by use of the valid–invalid bit. The operating system sets this bit for each page to allow or disallow access to the page.

3. There is still the problem of ***internal fragmentation*** in the above scheme. A process rarely use all of its address range. In fact, many processes use only a small fraction of the address space available to them. It would be wasteful in these cases to create a page table with entries for every page in the address range. Most of this table would be unused but would take up valuable memory space. Some systems provide hardware, in the form of a ***page-table length register (PTLR)***, to indicate the size of the page table. This value is checked against every logical address to verify that the address is in the valid range for the process. Failure of this test causes an error trap to the operating system.

Shared Pages

- An advantage of paging is the possibility of sharing common code in multi-processing operating system. ***Re-entrant code*** (or pure code) is non-self-modifying code, 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's execution. The data for two different processes will, of course, be different.

- Example) Consider a system that supports 40 users, each of whom executes a *text editor*. If the text editor consists of 150 KB of code and 50 KB of data space, we need 8,000 KB to support the 40 users. But using shared pages, only one copy of the editor need be kept in physical memory. Each user's page table maps onto 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 (150 KB), plus 40 copies of the 50 KB of data space per user. The total space required is now 2,150 KB instead of 8,000 KB-a significant savings.

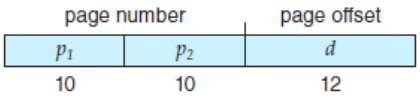
- Applications) Compilers, window system, run-time libraries, database systems, shared memory in inter-process communication, etc. Note: To be sharable, the code must be reentrant.

Structure of Page Tables

Since most modern computer systems support a large logical address space, the page table itself becomes excessively large. We would not want to allocate the page table contiguously in main memory. One simple solution to this problem is to divide the page table
into smaller pieces. This division of page table can be done in various ways:

1. **Two Level Paging Algorithm** or **Forward-Mapped Page Table**:

In this algorithm, the page table is itself also paged. For example, consider a system with a 32-bit logical address space and a page size of 4 KB. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus a logical address is as follows:



where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table. Because address translation works from the outer page table inward, this scheme is also known as a *forward-mapped page table*.

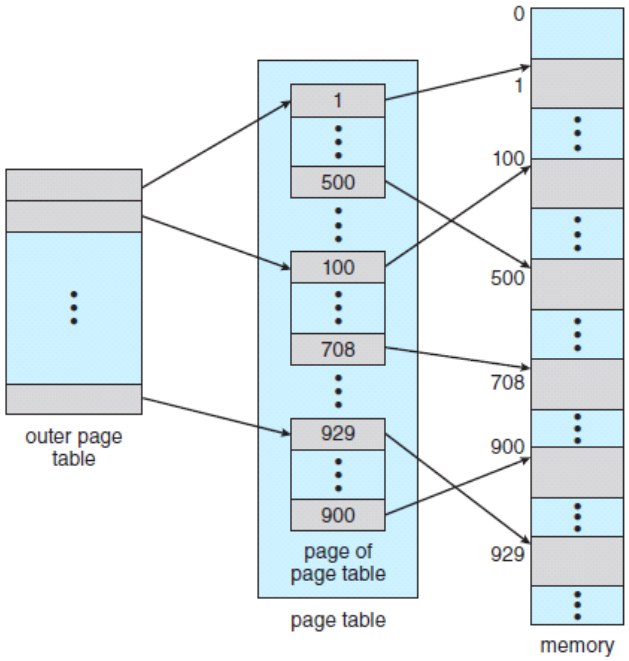


Figure 9.15 A two-level page-table scheme.

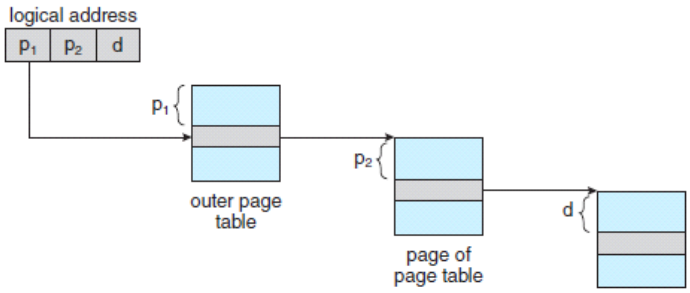
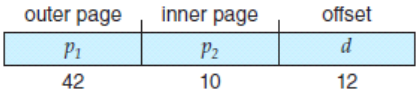


Figure 9.16 Address translation for a two-level 32-bit paging architecture.

Note: ***Multilevel Paging*****:** For a system with a 64-bit logical address space, a two-level paging scheme is no longer appropriate. Let's suppose that the page size in such a system is 4 KB (2^12). In this case, the page table consists of up to 2^52 entries. If we use a two-level paging scheme, then the inner page tables can conveniently be one page long, or contain 2^10 4-byte entries. The addresses look like this:



The outer page table consists of 2^42 entries (or 2^44 bytes). The obvious way to avoid such a large table is to divide the outer page table into smaller pieces. We can divide the outer page table in various ways like paging the outer page table, giving us a three-level paging scheme, and so on. Since more the level of paging, more will be number of main memory access reducing the overall time access.
Hence, for 64-bit architectures, hierarchical page tables (multi-level paging) are generally considered *inappropriate*.

1. **Hashed Page Tables**

Hashed page tables can be used to handle address spaces larger than 32 bits where the hash value of hash function is the virtual page number. Each entry in the hash table contains a linked list of elements that hash to the same location (to handle collisions). Each element consists of three fields:

- Virtual page number
- Value of the mapped page frame
- Pointer to the next element in the linked list

Algorithm

- The virtual page number in the virtual address is hashed into the hash table.
- The virtual page number is compared with field 1 in the first element in the linked list.
- If there is a match, the corresponding page frame (field 2) is used to form the desired physical address.
- If there is no match, subsequent entries in the linked list are searched for a matching virtual page number.

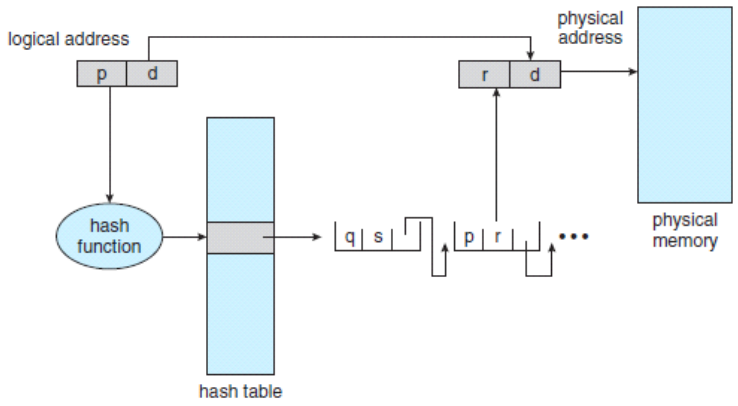


Figure 9.17 Hashed page table.

Note: A variation of Hashed Page Tables: ***Clustered page tables***: They are similar to hashed page tables except that each entry in the hash table refers to several pages rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. Clustered page tables are particularly useful for sparse address spaces, where memory references are noncontiguous and scattered throughout the address space.

Advantage & Disadvantage of Page Tables:

- *Advantage*: Since CPU generates logical address which needs to be translated to physical address by MMU, the table is sorted by virtual address(page numbers), the operating system is able to calculate where in the table the associated physical address entry is located and to use that value directly.
- *Disadvantage*: Each process has an associated page table and the page table has one entry for each page that the process is using. Each page table may consist of millions of entries. These tables may consume large amounts of physical memory just to keep track of how other physical memory is being used.

1. **Inverted Page Tables**

An inverted page table has one entry for each frame of main memory. Each entry consists of the virtual address of the page stored in that real memory location, with information about the process that owns the page. Thus, only one page table is in the system, and it has only one entry for each page of physical memory.

Inverted page tables requires an address-space identifier to be stored in each entry of the page table, since the table usually contains several different address spaces mapping physical memory. Storing the address-space identifier ensures that a logical page for a particular process is mapped to the corresponding physical page frame.

Although this scheme decreases the amount of memory needed to store each page table, it increases the amount of time needed to search the tablewhen a page reference occurs. Because the inverted page table is sorted by physical address, but lookups occur on virtual addresses, the whole table might need to be searched before a match is found. This search would take far too long.

To reduce this problem, we use a hash table to limit the search to few page-table entries. Of course, each access to the hash table adds amemory reference to the procedure, so one virtual memory reference requires at least two real memory reads: one for the hash-table entry and one for the page table.

Systems that use inverted page tables have difficulty implementing shared memory. With standard paging, each process has its own page table, which allows multiple virtual addresses to be mapped to the same physical address. This method cannot be used with inverted page tables; because there is only one virtual page entry for every physical page, one physical page cannot have two (or more) shared virtual addresses. Therefore, with inverted page tables, only one mapping of a virtual address to the shared physical address may occur at any given time. A reference by another process sharing the memory will result in a page fault and will replace the mapping with a different virtual address.

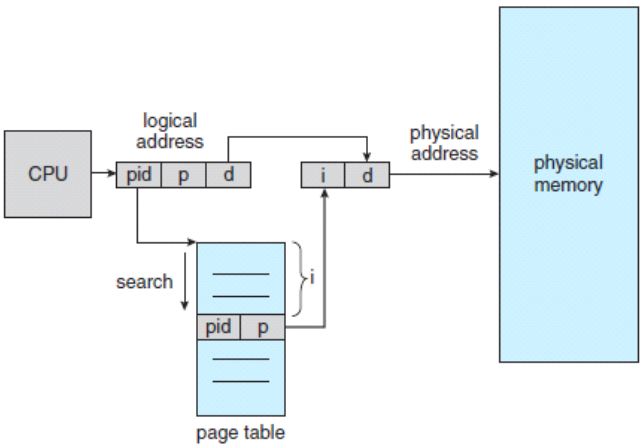


Figure 9.18 Inverted page table.

Extra Questions

Q) Why paging increases the context-switching time?
R) Context Switch basically means, taking CPU from one process and giving it to another process (which is in ready queue and hence already loaded into memory).

Paging is a way to manage memory efficiently by bringing those parts of program into memory which are required and keeping the rest of the disk free and bringing them as per need. Pages of other processes, which were brought into memory earlier might be evicted to make space for new processes.

So in current situation, during a context switch, the new processes might find that most of its pages have been evicted, which needs to be brought back, which will take time and hence the time to context switch is increased.

Imaging if there were no paging, then the whole program has to be kept into the memory and during a context switch, the new process will always find its all pages in the memory and hence this will be very fast but unefficient use of the memory, as we have to keep everything in the memory even if it is not required as of now.

Q) What is difference between *pages* and *frames*?
R) In paging, the Main Memory is divided into parts called as 'Frames' and the process is divided into 'Pages' so that a part of process(a page) can be accommodated in a frame(part of Main Memory). A Page Table keeps track of the pages and where they are present in the Main Memory. The pages can be present in any frame. The frames need not be contiguous. This is

how External Fragmentation is prevented.It is important to note that Page Size = Frame Size.

Q) What is the difference between Cache Miss, TLB miss and page fault?

R) The CPU generates the logical address, which contains the page number and the page offset. The page number is used to index into the page table, to get the corresponding page frame number, and once we have the page frame of the physical memory(also called main memory), we can apply the page offset to get the right word of memory.

Why TLB(Translation Look Aside Buffer): The thing is that page table is stored in physical memory, and sometimes can be very large, so to speed up the translation of logical address to physical address , we use TLB, which is made of expensive and faster associative memory, So instead of going into page table first, we go into the TLB and use page number to index into the TLB, and get the corresponding page frame number and if it is found, we completely avoid page table(because we have both the page frame number and the page offset) and form the physical address.

TLB Miss: If we don't find the page frame number inside the TLB, it is called a TLB miss only then we go to the page table to look for the corresponding page frame number.

TLB Hit: If we find the page frame number in TLB, its called TLB hit, and we don't need to go to page table.

Page Fault: Occurs when the page accessed by a running program is not present in physical memory. It means the page is present in the secondary memory but not yet loaded into a frame of physical memory.

Cache Hit: Cache Memory is a small memory that operates at a faster speed than physical memory and we always go to cache before we go to physical memory. If we are able to locate the corresponding word in cache memory inside the cache, its called cache hit and we don't even need to go to the physical memory.

Cache Miss: It is only after when mapping to cache memory is unable to find the corresponding block(block similar to physical memory page frame) of memory inside cache (called cache miss), then we go to physical memory and do all that process of going through page table or TLB.

Overall Flow

a. First go to the cache memory and if its a **cache hit**, then we are done.
b. If its a **cache miss**, first go to TLB and if its a **TLB hit**, go to physical memory using physical address formed, we are done.
c. If its a **TLB miss**, then go to page table to get the frame number of your page for forming the physical address.
d. If page is found, it is a **page hit**, else page is not found, its a **page fault**. Use one of the page replacement algorithms if all the frames are occupied by some page else just load the required page from secondary memory to physical memory frame.

Segmentation

- *User View of Memory:* Users view memory as a collection of variable-sized segments, with no necessary ordering among segments. For eg) A program is a main program with a set of methods, procedures, or functions. It may also include various data structures: objects, arrays, stacks, variables, and so on. Each of these modules or data elements is referred to by name.

- Segmentation is a memory-management scheme that supports this user view of memory. A logical address space is a collection of segments. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment. The user therefore specifies each address by two quantities: a segment name and an offset.

- 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 pair: ***<segment-number, offset>***.

  Address generated by the CPU is divided into:

  - **Segment number (s):** Number of bits required to represent the segment.
  - **Segment offset (d):** Number of bits required to represent the size of the segment.

- The user program is compiled, and the compiler automatically constructs segments reflecting the input program. Libraries that are linked in during compile time might be assigned separate segments. The loader would take all these segments and assign them segment numbers. C compiler might create separate segments for the following:
  - The code
  - Global variables
  - The heap, from which memory is allocated
  - The stacks used by each thread
  - The standard C library

- *Hardware:*

Although the user can now refer to objects in the program by a two-dimensional address, the actual physical memory is still a one-dimensional sequence of bytes. Thus, we must define an implementation to map two-dimensional user-defined addresses into one-dimensional physical addresses.

This mapping is implemented by a ***segment table***. Each entry in the segment table has a ***segment 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.

A logical address consists of two parts: a *segment number s*, and an *offset into that segment d*. The segment number is used as an index to the segment table.

The offset d of the logical address must be between 0 and the segment limit. If it is not, we trap to the operating system (logical addressing attempt beyond end of segment).

When an offset is legal, it is added to the segment base to produce the address in physical memory of the desired byte. The segment table is thus essentially an array of base-limit register pairs.

The segment table has a separate entry for each segment, giving the beginning address of the segment in physical memory (or base) and the length of that segment (or limit).

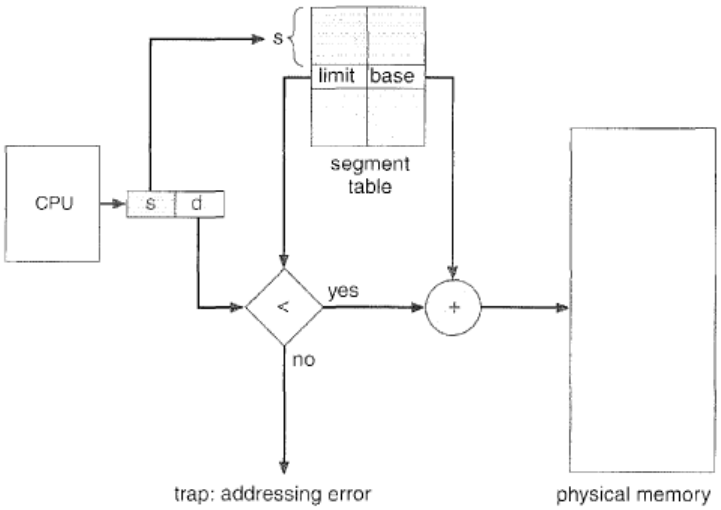


Figure 8.19 Segmentation hardware.

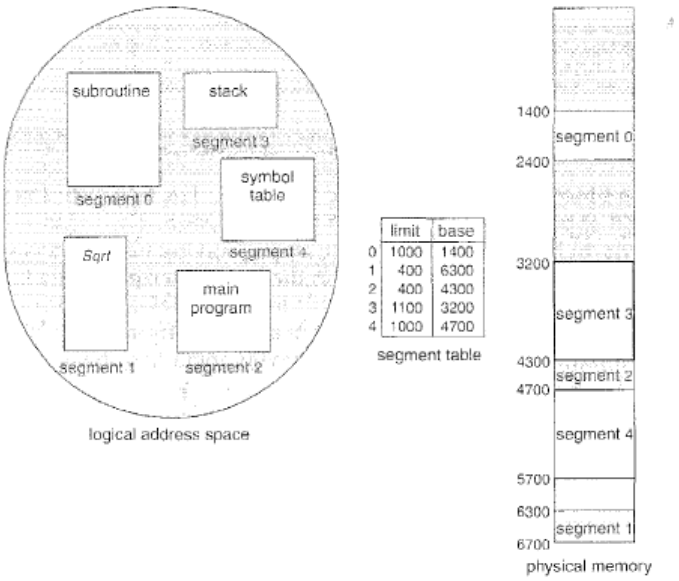


Figure 8.20 Example of segmentation.

Advantages of Segmentation:

- No Internal fragmentation.
- Segment Table consumes less space in comparison to Page table in paging.

Disadvantage of Segmentation:

- As processes are loaded and removed from the memory, the free memory space is broken into little pieces, causing *External fragmentation*.

Differences between Paging and Segmentation

S.NO	Paging	Segmentation
1.	In paging, program is divided into fixed or mounted size pages.	In segmentation, program is divided into variable size sections.
2.	For paging operating system is accountable.	For segmentation compiler is accountable.

3.	Page size is determined by hardware.	Here, the section size is given by the user.
4.	It is faster in the comparison of segmentation.	Segmentation is slow.
5.	Paging could result in internal fragmentation.	Segmentation could result in external fragmentation.
6.	In paging, logical address is split into page number and page offset.	Here, logical address is split into section number and section offset.
7.	Paging comprises a page table which encloses the base address of every page.	While segmentation also comprises the segment table which encloses segment number and segment offset.
8.	Page table is employed to keep up the page data.	Section Table maintains the section data.
9.	In paging, operating system must maintain a free frame list.	In segmentation, operating system maintain a list of holes in main memory.
10.	Paging is invisible to the user.	Segmentation is visible to the user.
11.	In paging, processor needs page number, offset to calculate absolute address.	In segmentation, processor uses segment number, offset to calculate full address.

Virtual Memory

Virtual memory is a technique that allows the execution of processes that are not completely in memory. Virtual memory abstracts main memory into an extremely large, uniform array of storage, separating logical memory as viewed by the programmer from physical memory. Virtual Memory can be implemented using *Demand Paging or Demand Segmentation*.

- *Need of Virtual Memory:* The instructions being executed must be in physical memory. One approach can be placing the entire logical address space in physical memory. It limits the size of a program to at maximum size of physical memory. The problem with this approach is that the entire program is not needed in many cases and even in those cases where the entire program is needed, it may not all be needed at the same time.

- Executing programs while they are loaded only partially in memory is beneficial for both the system and it's users in the following ways:
  a. A program can be of size larger than the size of available physical memory. Users would be able to write programs for an extremely large virtual address space, simplifying the programming task.

  b. Because each program could take less physical memory, more programs could be run at the same time,with a corresponding increase in CPU utilization and throughput but with no increase in response time or turn around time.

  c. Less I/O would be needed to load or swap portions of programs into memory, so each program would run faster.

- Virtual memory involves the separation of logical memory as perceived by developers 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 much easier, because the programmer no longer needs to worry about the amount of physical memory available.

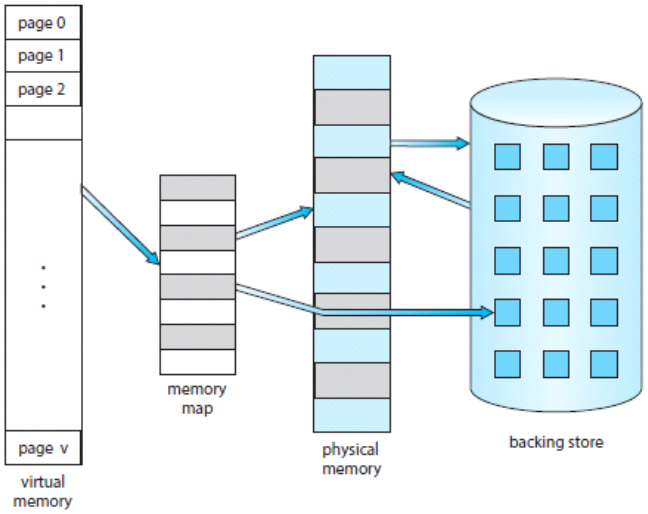


Figure 10.1 Diagram showing virtual memory that is larger than physical memory.

- ***Virtual Address Space***
  - The virtual address space of a process refers to the logical (or virtual) view of how a process is stored in memory. A process begins at a certain logical address and exists in contiguous memory.
  - Physical memory is organized in page frames and the physical page frames assigned to a process may not be contiguous. It is up to the memory management unit (MMU) to map logical pages to physical page frames in memory.
  - We allow the heap to grow upward in memory as it is used for dynamic memory allocation. Similarly, we allow for the stack to grow downward in memory through successive function calls.
  - The large blank space (or hole) between the heap and the stack is part of the virtual address space but will require actual physical pages only if the heap or stack grows. Virtual address spaces that include holes are known as ***sparse address spaces***.
  - Using a sparse address space is beneficial because the holes can be filled as the stack or heap segments grow or if we wish

to dynamically link libraries during program execution.

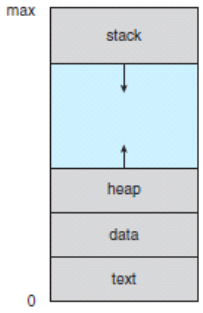


Figure 10.2 Virtual address space of a process in memory.

- Virtual memory allows files and memory to be shared by two or more processes through page sharing, which leads to following benefits:

  a. System libraries (like standard C library) can be shared by several processes through mapping of the shared object into a virtual address space. Although each process considers the libraries to be part of its virtual address space, the actual pages where the libraries reside in physical memory are shared by all the processes. Typically, a library is mapped read-only into the space of each process that is linked with it.

  b. Similarly, processes can share memory. Virtual memory allows one process to create a region of memory that it can share with another process. Processes sharing this region consider it part of their virtual address space, yet the actual physical pages of memory are shared.

  c. Pages can be shared during process creation with the fork() system call, thus speeding up process creation.

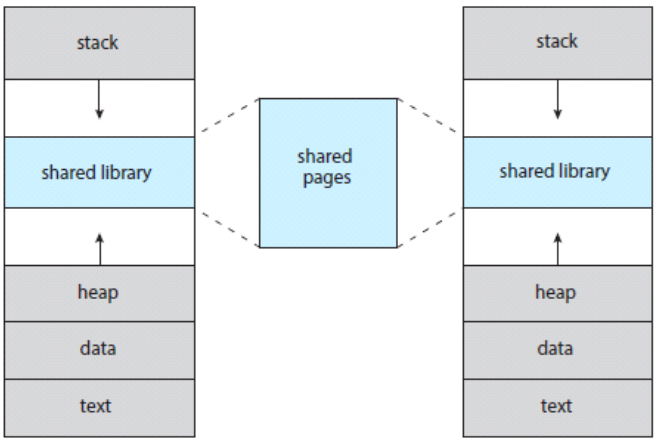


Figure 10.3 Shared library using virtual memory.

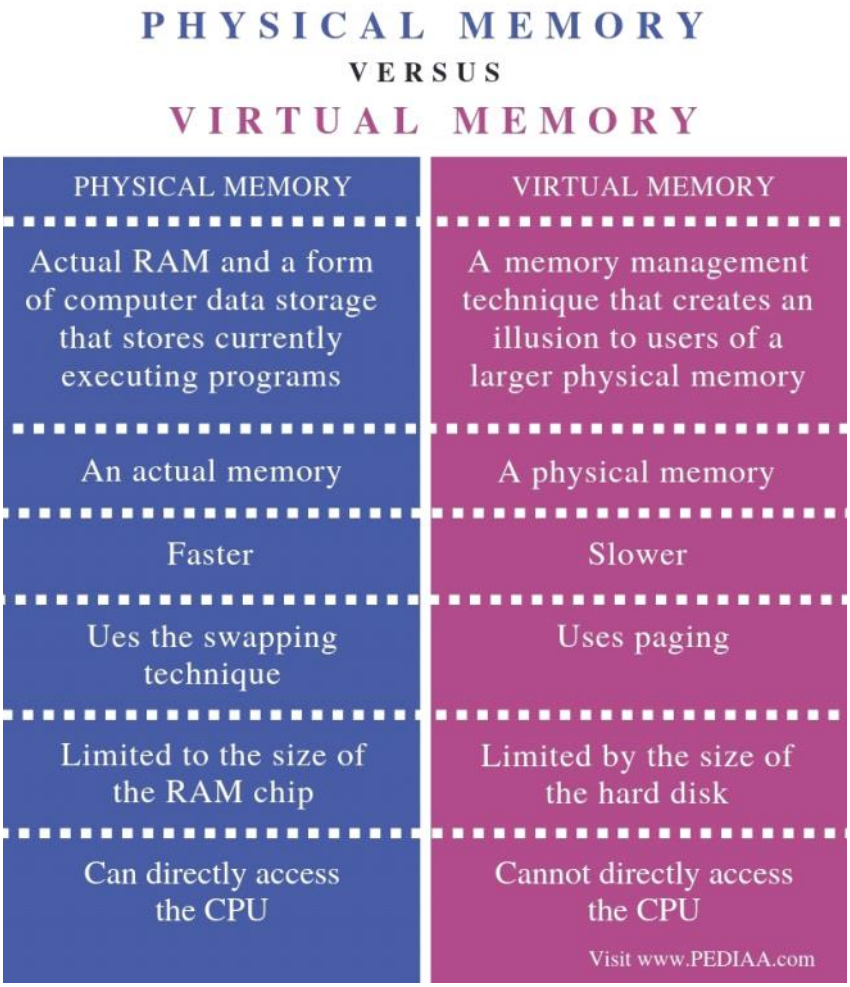
Advantages of Virtual Memory

- Virtual memory helps to gain speed when only a particular segment of the program is required for the execution of the program.
- It is very helpful in implementing a multiprogramming environment.
- It allows you to run more applications at once.
- It helps you to fit many large programs into smaller programs.
- Common data or code may be shared between memory.
- Process may become even larger than all of the physical memory.
- Data / code should be read from disk whenever required.
- The code can be placed anywhere in physical memory without requiring relocation.
- More processes should be maintained in the main memory, which increases the effective use of CPU.

- Each page is stored on a disk until it is required after that, it will be removed.
- It allows more applications to be run at the same time.
- There is no specific limit on the degree of multiprogramming.
- Large programs should be written, as virtual address space available is more compared to physical memory.

Disadvantages of Virtual Memory

- Applications may run slower if the system is using virtual memory.
- Likely takes more time to switch between applications.
- Offers lesser hard drive space for your use.
- It reduces system stability.
- It allows larger applications to run in systems that don't offer enough physical RAM alone to run them.
- It doesn't offer the same performance as RAM.
- It negatively affects the overall performance of a system.
- Occupy the storage space, which may be used otherwise for long term data storage.



Demand Paging

Demand paging is a strategy used to load only those pages of an executable program from secondary memory to the primary(main) memory, which are currently needed/demanded during execution.
Pages that are never accessed are thus never loaded into physical memory. A demand-paging system is combination of paging and swapping.
It is commonly used in virtual memory systems. By loading only the portions of programs that are needed, demand paging explains one of the primary benefits of virtual memory which is using the memory more efficiently.

Working

- Since, only some pages of a program are loaded to the main memory, and the rest will be in the secondary memory, there should be a hardware support to distinguish between two types of pages.

- The valid-invalid bit scheme can be used for this purpose. When the bit is set to *valid*, the associated page is both legal and in memory. If the bit is set to *invalid*, the page either is not valid (not in the logical address space of the process) or is valid but is currently in secondary storage.

- 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 not currently in memory is simply marked invalid.

- If the process tries to access a page that was not brought into memory, access to a page marked invalid causes a ***page fault***. The paging hardware, in translating the address through the page table, will cause a trap to the operating system on seeing the valid-invalid bit set as invalid.

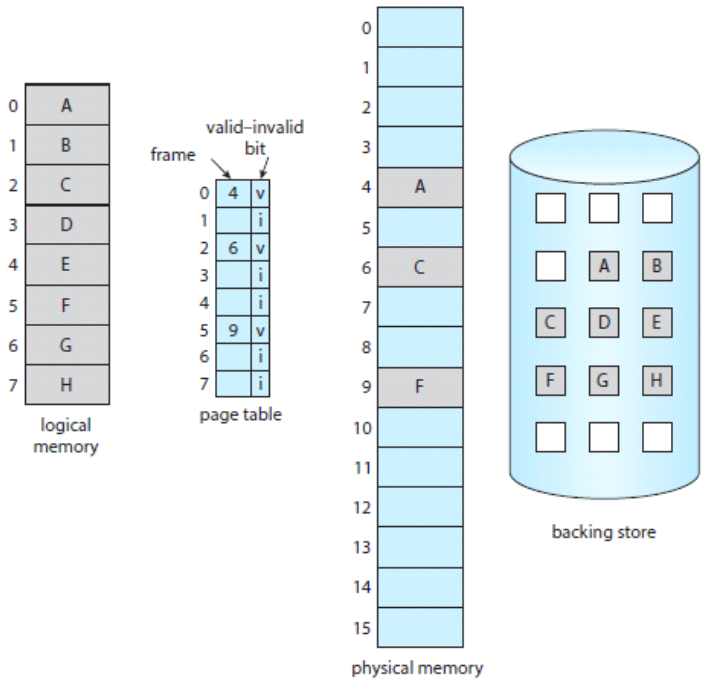


Figure 10.4 Page table when some pages are not in main memory.

Page Fault Handling

1. We check an internal table (usually kept with the process control block) for this process to determine whether the reference was a valid or an invalid memory access.
2. If the reference was not in the logical address space of process, we terminate the process. If it was valid i.e. we have not brought that page in main memory, *page fault* has occurred. We find a free frame from the ***free-frame list***, a pool of free frames for satisfying such requests. Zero-fill-on-demand frames are zeroed-out before being allocated, thus erasing their previous contents to avoid potential security implications of not clearing the content.

3. We schedule a secondary storage operation to read the desired page into the newly allocated frame.
   a. Wait in a queue 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.
4. While waiting for secondary storage, allocate the CPU core to some other process and save the registers and process state for the other process.
5. When the storage read is complete, receive an interrupt from the secondary storage system (I/O completed), and then modify the internal table kept with the process and the page table to indicate that the page is now in main memory.
6. Wait for the CPU core to be allocated to this process again, and when allocated, restore the registers, process state, and new page table, and then resume the interrupted instruction.

Pure Demand Paging*:* In this scheme, a page is never brought into memory until it is required. Such a swapper is known as ***lazy swapper***. In the extreme case, we can 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 page not in memory , 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.

Locality of Reference: Some programs could access several new pages of memory with each instruction execution (one page for the instruction and many for data), possibly causing multiple page faults per instruction. This situation would result in unacceptable system performance. Programs tend to have locality of reference which results in reasonable performance from demand paging.

Performance of Demand Paging:

Effective Access Time for a demand paged memory is
Effective Access Time = **(1 — P) X Memory Access Time + P X Page Fault Time**
Where, P = Probability of a Page Fault

Page Fault Rate $0 \leq p \leq 1$

- if p = 0 no page faults
- if p = 1, every reference is a fault

The effective access time is directly proportional to the ***page-fault rate***. 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.

Hardware

The ***hardware*** 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 a 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 or NVM device. It is known as the ***swap device***, and the section of storage used for this purpose is known as ***swap space***.

Swap Space Management

- I/O to swap space is generally faster than that to the file system even when both are on secondary storage because swap space is allocated in larger contiguous secondary memory space and thus less management is required than file system.
- One approach for paging is copying an entire file image into the swap space at process startup and then performing demand paging from the swap space.
- Another approach is to demand-page 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 read from the file system but that all subsequent paging is done from swap space. This approach is used in most OS including Linux & Windows.

Note: Some OS limits the amount of swap space, and in those cases, demand pages for binary executable files are swapped in main memory directly from the file system and thus file system itself acts as the backing store. However, swap space must still be used for pages not associated with any file known as ***anonymous memory***.

Copy on Write

- Process creation using the ***fork()*** system call may bypass the need for demand paging by using a technique similar to page sharing which provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.

- 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.

- We can use a technique known as copy-on-write, which works by allowing the parent and child processes initially to 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.

- If the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write, the operating system will obtain a frame from the free-frame list and create a copy of this page, mapping it to the address space of the child process. The child process will then modify its copied page and not the page belonging to the parent process.

Note: Only pages that can be modified need be marked as copy-on-write. Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.
Note: This technique is used by most operating systems like Windows, Linux, and macOS.

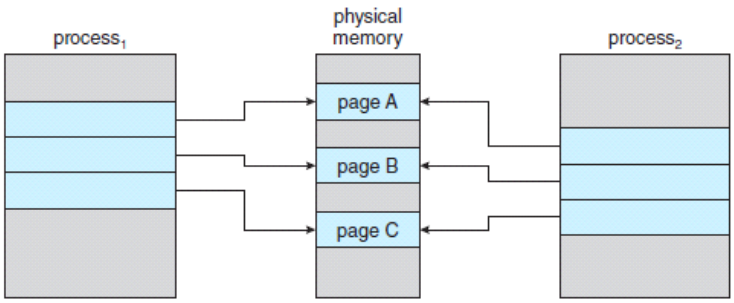


Figure 10.7 Before process 1 modifies page C.

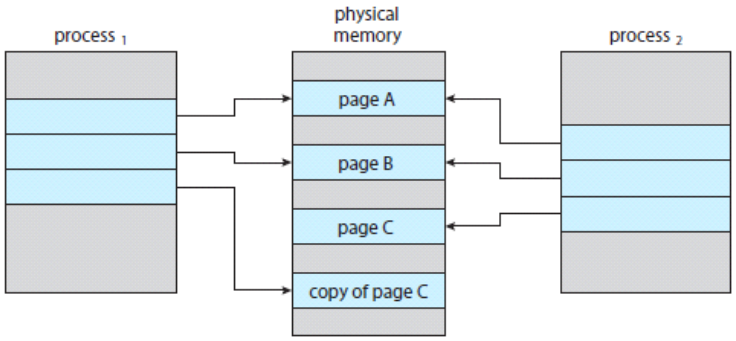


Figure 10.8 After process 1 modifies page C.

Page Replacement

Need of Page Replacement

Over-Allocation of Memory: While a process is executing, a page fault occurs. The operating system determines where the desired page is residing on secondary storage but then finds that there are no free frames on the free-frame list; all memory is in use. This situation is known as over-allocation of memory and occurs due to need of increase in degree of multiprogramming.

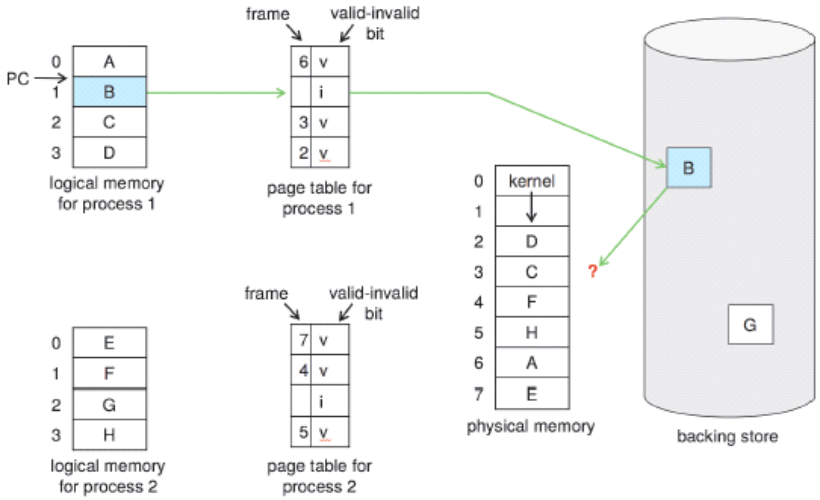


Figure 10.9 Need for page replacement.

Operating System can tackle this problem in one of the following ways:

1. System can simply terminate the process. However, demand paging is the operating system's attempt to improve the computer system's utilization and throughput, thus it is not the best choice.

2. The operating system could use standard swapping and swap out a process, freeing all its frames and reducing the level of multiprogramming. However, standard swapping is no longer used by most operating systems due to the overhead of copying entire processes between memory and swap space.

3. Most operating systems now combine swapping pages with ***page replacement***. If no frame is free, we find one that is not currently being used and free it. We can free a frame by writing its contents to swap space and changing the page table (and all other tables) to indicate that the page is no longer in memory. We can now use the freed frame to hold the page for which the process faulted.

Note: Page replacement is basic necessity for demand paging as it completes the separation between logical memory and physical memory. With this mechanism, an enormous virtual memory can be provided for programmers on a smaller physical memory.

Note: Apart from appropriate page replacement algorihtm, demand paging also requires efficient ***frame allocation algorithm***. If we have multiple processes in memory, we must decide how many frames to allocate to each process.

Page Fault Service Routine (supporting Page Replacement)

1. Find the location of the desired page on secondary storage.
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***.
   c. Write the victim frame to secondary storage (if necessary) and change the page and frame tables accordingly.
3. Read the desired page into the newly freed frame; change the page and frame tables.
4. Continue the process from where the page fault occurred.

Modify Bit or Dirty Bit

- Need: If no frames are free, two page transfers , both for page-out and page-in are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly.

- Each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified.

- When we select a page for replacement, we examine its modify bit.
  - If the bit is set i.e. page is modified, since it was read in from secondary storage, we must write the page to storage.
  - If the modify bit is not set i.e. the page has not been modified, since it was read into memory, we need not write the memory page to storage as it is already there.

- This scheme can significantly reduce the time required to service a page fault, since it reduces I/O time by one-half if the page has not been modified.

Criteria for Selecting Page Replacement Algorithm

- Page replacement algorithm to be used should have ***lowest page-fault rate***.
- We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a ***reference 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. Adding physical memory increases the number of frames and as the number of frames available increases, the number of page faults decreases.

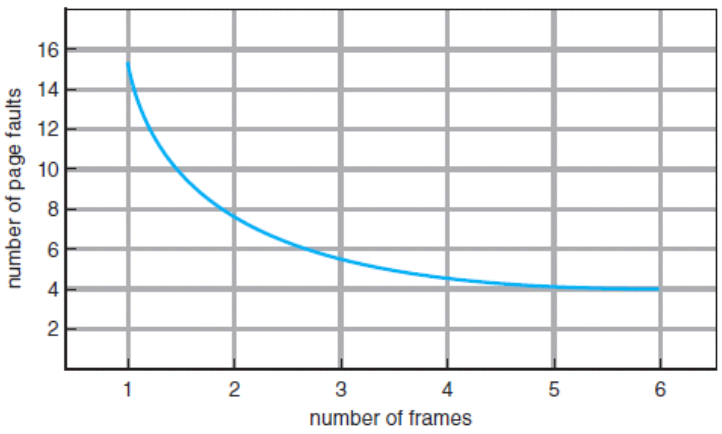


Figure 10.11 Graph of page faults versus number of frames.

1. **FIFO (First In First Out) Page Replacement 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.

- 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 it at the tail of the queue.

- The FIFO page-replacement algorithm is easy to understand and program. However, its performance is not always good. If we select a page that is in active use for replacement with a new one, after replacing it, a fault occurs almost immediately to retrieve the active page. Some other page must be replaced to bring the active page back into memory. Thus, a bad replacement choice *increases the page-fault rate* and slows process execution.

- ***Belady's anomaly***: For some page-replacement algorithms (like the FIFO algorithm), the page-fault rate may increase as the number of allocated frames increases. We would expect that giving more memory to a process would improve its performance, but converse happens.

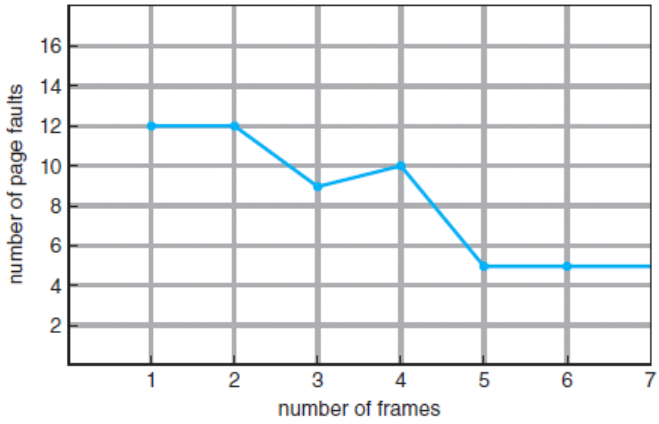


Figure 10.13 Page-fault curve for FIFO replacement on a reference string.

2. **Optimal Page Replacement (OPT) Algorithm**

- *Algorithm*: Replace the page that will not be used for the longest period of time.
- It is the algorithm that has the lowest page-fault rate (for a fixed number of frames) of all algorithms and will never suffer from Belady's anomaly.
- However, this algorithm does not exists as it requires future knowledge of the reference string. Thus it is only theoretical algorithm and used for comparison studies of new algorithms.

Note: The key distinction between the FIFO and OPT algorithms are looking backward versus forward in time and that the FIFO algorithm uses the time when a page was brought into memory, whereas the OPT algorithm uses the time when a page is to be used.

3. **LRU (Least Recenlty Used) Page Replacement Algorithm**

- LRU replacement associates with each page the time of that page's last use. When a page must be replaced, LRU chooses the page that has not been used for the longest period of time. Thus we use the recent past as an approximation of the near future.
- The LRU policy is often used as a page-replacement algorithm and is considered to be good. It can be implemented by using substantial hardware-assistance in one of the following ways:

  a. ***Counters:*** We associate a *time-of-use* field with each page table entry and add a logical clock or counter to the CPU. 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-table entry and thus we can have the "time" of the last reference to each page. This scheme requires a search of the page table to find the LRU page and a write to time-of-use field for each memory access. The times must also be maintained when page tables are changed (due to CPU scheduling). Overflow of the clock must be considered.

  b. ***Stack***: We can keep a stack of page numbers. Whenever a page is referenced, it is removed from the stack and put on the top. In this way, the most recently used page is always at the top of the stack, and the least recently used page is always at the bottom. Because entries must be removed from the middle of the stack, it is best to implement this approach by using a ***doubly linked list*** with a head pointer and a 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. This approach is particularly appropriate for software or microcode implementations of LRU replacement.

- This strategy is same as the optimal page-replacement algorithm looking backward in time, rather than forward. If S_R be the reverse of a reference string S, then the page-fault rate for the OPT algorithm on S is the same as the page-fault rate for the

LRU algorithm on S_R.

Note: ***Stack Algorithms*:**

- A stack algorithm is an algorithm for which the set of pages in memory for n frames is always a subset of the set of pages that would be in memory with n + 1 frames.
- These algorithms does not suffer from Belady's anomaly. Examples of stack algorithms are optimal page replacement and LRU page replacement algorithms.
- For LRU replacement, the set of pages in memory would be the n most recently referenced pages. If the number of frames is increased, these n pages will still be the most recently referenced and so will still be in memory.

4. **Least Frequently Used (LFU) Algorithm**

- Least frequently used (LFU) page-replacement algorithm 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.

- However, a problem arises when 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 usage count.

5. **Most frequently used (MFU) Algorithm**

It is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

Thrashing

Definition

If a process does not have the minimum number of frames it needs to support pages in the working set, it will quickly page-fault. However, since all its pages are in active use, it will have to replace a page that will be needed again right away. Consequently, it quickly faults again, and again, and again, replacing pages that it must bring back in immediately. This high paging activity is called thrashing. A process is thrashing if it is spending more time paging than executing. Thrashing results in severe performance problems.

Effects & Cause of Thrashing

CPU utilization is plotted against the degree of multiprogramming. As the degree of multiprogramming increases, CPU utilization also increases, although more slowly, until a maximum is reached. If the degree of multiprogramming is increased further, thrashing sets in, and CPU utilization drops sharply. At this point, to increase CPU utilization and stop thrashing, we must decrease the degree of multiprogramming.

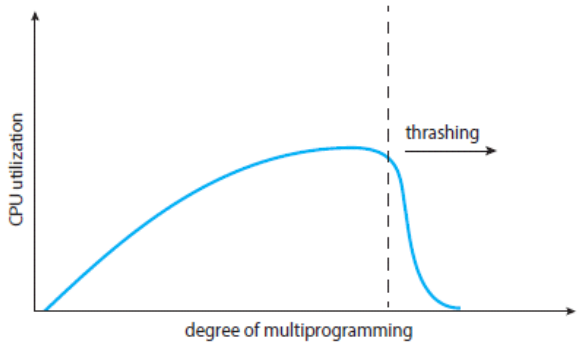


Figure 10.20 Thrashing.

The effects of thrashing and also the extent to which thrashing occurs will be decided by the type of page replacement policy.

1. **Global Page Replacement**: The paging algorithm is applied to all the pages of the memory regardless of which process "owns" them. A page fault in one process may cause a replacement from any process in memory. Thus, the size of a partition may vary randomly.

If **global** page replacement is used, situations worsens very quickly. CPU thinks that CPU utilization is decreasing, so it tries to increase the degree of multiprogramming. Hence bringing more processes inside memory, which in effect increases the thrashing and brings down CPU utilization further down. The CPU notices that utilization is going further down, so it increases the degree of multiprogramming further and the cycle continues.

2. **Local Page Replacement**: The memory is divided into partitions of a predetermined size for each process and the paging algorithm is applied independently for each region. A process can only use pages in its partition.

The solution can be **local** page replacement where a process can only be allocated pages in its own region in memory. If the swaps of a process increase also, the overall CPU utilization does not decrease much. If other transactions have enough page frames in the partitions they occupy, they will continue to be processed efficiently. Thus, if one process starts thrashing, it cannot steal frames from another process and cause the latter to thrash as well.

But, the problem is still not solved. 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 because of the longer average queue for the paging device. Thus, the effective access time will increase even for a process that is not thrashing.

Locality Model:

- 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 running program is generally composed of several different localities, which may overlap.
- Localities are defined by the program structure and its data structures. The locality model states that all programs will exhibit this basic memory reference structure.
- Suppose we allocate enough frames to a process to accommodate its current locality. It will fault for the pages in its locality until all these pages are in memory; then, it will not fault again until it changes localities.

- If we do not allocate enough frames to accommodate the size of the current locality, the process will thrash, since it cannot keep in memory all the pages that it is actively using.

Thrashing Handling

To prevent thrashing, we must provide a process with as many frames as it needs. It can be provided in the following ways:

1. **Working Set Model**

This model is based on the above-stated concept of the Locality Model. The basic principle states that if we allocate enough frames to a process to accommodate its current locality, it will only fault whenever it moves to some new locality. But if the allocated frames are lesser than the size of the current locality, the process is bound to thrash.

According to this model, based on a parameter A, the working set is defined as the set of pages in the most recent 'A' page references. Hence, all the actively used pages would always end up being a part of the working set.

The accuracy of the working set is dependant on the value of parameter A. If A is too large, then working sets may overlap. On the other hand, for smaller values of A, the locality might not be covered entirely.

If ***D*** is the total demand for frames and ***WSSi*** is the working set size for a process ***i***, then

$$D = \Sigma WSSi$$

Now, if ***m*** is the number of frames available in the memory, there are 2 possibilities:

- ***D>m*** : Total demand exceeds the number of frames, then thrashing will occur as some processes would not get enough frames.
- ***D<=m***: Then there would be no thrashing.

If there are enough extra frames, then some more processes can be loaded in the memory. On the other hand, if the summation of working set sizes exceeds the availability of frames, then some of the processes have to be suspended(swapped out of memory).

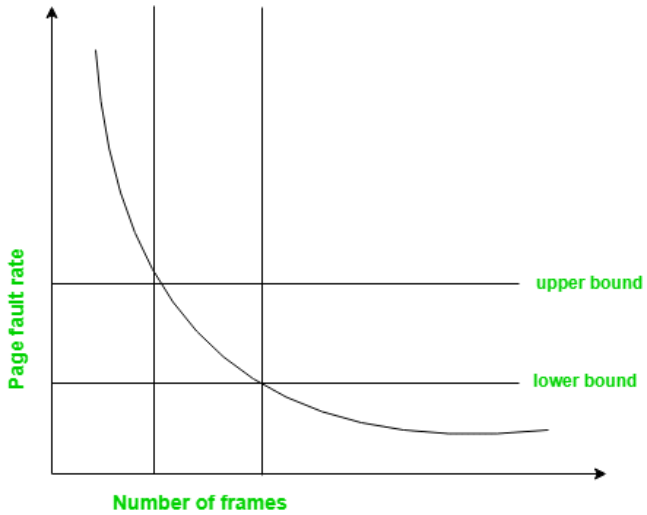
This technique prevents thrashing along with ensuring the highest degree of multiprogramming possible. Thus, it optimizes CPU utilisation.

2. **Page Fault Frequency Model**

A more direct approach to handle thrashing is the one that uses Page-Fault Frequency concept.

The problem associated with Thrashing is the high page fault rate and thus, the concept here is to control the page fault rate.

If the page fault rate is too high, it indicates that the process has too few frames allocated to it. On the contrary, a low page fault rate indicates that the process has too many frames.



Upper and lower limits can be established on the desired page fault rate as shown in the diagram.

If the page fault rate falls below the lower limit, frames can be removed from the process. Similarly, if the page fault rate exceeds the

upper limit, more number of frames can be allocated to the process.

In other words, the graphical state of the system should be kept limited to the rectangular region formed in the given diagra m.

Here too, if the page fault rate is high with no free frames, then some of the processes can be suspended and frames allocate d to them can be reallocated to other processes. The suspended processes can then be restarted later.

Storage Management

"I will speak ill of no one and speak all the good I know of everybody."
...Andrew Jackson

CHAPTER 8

I/O System

Learning Objectives

After reading this chapter, you will know:

1. Kernel I/O Subsystem
2. Disk Management

Introduction

The role of the operating system in computer I/O is to manage and control I/O operations and I/O devices.

Kernel I/O Subsystem

Kernels provide many services related to I/O. Several services like scheduling, buffering, caching, spooling, reservation and error handling are provided by the kernel I/O subsystem and are built on the hardware and device driver infrastructure.

(i) **I/O Scheduling:** To schedule a set of I/O requests means to determine a good order in which to execute them. The order in which application issues system calls rarely is the best choice. Scheduling can improve overall system performance, can share device access fairly among processes and can reduce the average waiting time for I/O to complete.

(ii) **Buffering:** A buffer is a memory area that stores data while they are transferred between two devices or between a device and an application.

Buffering is done for three reasons. One reason is to cope with a speed mismatch between the producer and consumer of a data stream.

And second use of buffering is to adapt between devices that have different data transfer rate. Such disparitions are especially common in computer networking where buffers are used widely for fragmentation and reassembling of message.

A third use of buffering is to support copy semantics for application I/O.

(iii) **Caching:** A cache is a region of fast memory that holds copies of data. Access to the cached copy is more efficient than access to the original. Caching and buffering are distinct functions but sometimes a region of memory can be used for both purposes.

(iv) **Spooling and Device Reservation:**

A spool is a buffer that holds output for a device, such as a printer, that cannot accept interleaved data streams. Although a printer can serve only one job at a time, several applications may wish to print their output concurrently, without having their output mixed together.

The operating system solves this problem by intercepting all output to the printer. Each applications output is spooled to a separate disk file.

When an application finishes printing, the spooling system queues the corresponding spool file for output to the printer.

Disk Structure

A hard disk is a collection of platters each disk platter has a flat circular shape, like a compact disk (CD), common platter diameter ranges from 1.8 to 5.25 inches. Two surfaces of a platter are covered with magnetic material. We store information by recording it magnetically on the platters.

The space of platters are logically divided in to circular 'Tracks'. The tracks are subdivided into 'sectors'. The set of tracks that's at one arm position forms a 'cylinder'.

Disk Performance Parameter

Seek time: The time required to reach the desired track by Read-Write head is the 'Seek Time'. The linear formula for seek time is given below.

$T_s = m * n + S$

T_s = Estimated Seek Time

n = Number of Tracks Traversed

m = Constant that depends on the disk drive

S = Startup Time.

Rotational Delay: The time required to rotate the desired sector under read-write head is called Rotational Delay.

Access time: The sum of seek time and rotational delay is the access time.

Transfer time: The transfer time is depends on the rotation speed of the disk. The formula for the transfer time is,

$T = B/R * N$

T = Transfer Rate

B = Number of bytes to be transferred.

N = Number of bytes on tracks.

R = Rotation speed in revolution per second.

Thus the total average access time can be express as

$Ta = Ts + \frac{1}{2}R + B/RN$ where, Ts is the average seek time

Disk Scheduling

The Seek Time is the time for the disk arm to move the heads to the cylinder containing the desired sector.

The Rotational Latency is the additional time waiting for the disk to rotate the desired sector to the disk head.

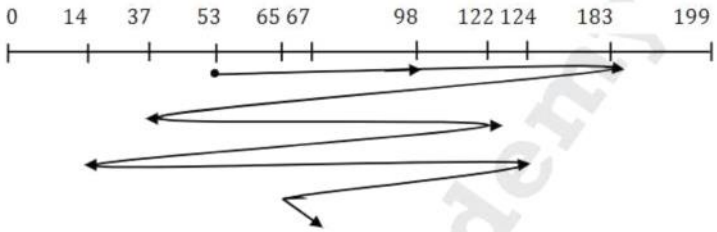
The Disk Bandwidth is the total number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer.

We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in good order.

FCFS : (First-Come First-Serve) Scheduling

This is the simplest disk scheduling. As its name FCFS, the request for block that comes first is served first.

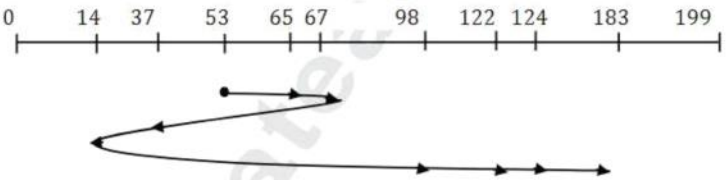
For example ,the request for I/O to blocks are on following cylinder -98, 183, 37, 122, 14, 124, 65, 67, in that order. If the disk head is initially at cylinder 53, it will first move from 53 to 98 then to 183, 37, 122, 14, 124, 65 and finally to 67, for a total head moment to 640 cylinders.



SSTF (Shortest-Seek-Time-First) Scheduling

The SSTF algorithms select the request with the minimum seek time from the current head position

For above example



This strategy reduces the total head movement to 236 cylinders. So, it is more improved form of scheduling than FCFS.

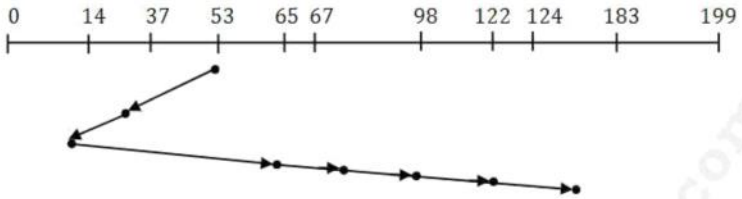
SCAN Algorithm

In the scan algorithm, the disk arm starts at one end of the disk and moves towards the other end, servicing requests as it reaches each cylinder, until it gets to other end of the disk. At the other end, the direction of head movement is reversed and servicing continues. The head continuously scans back and forth across the disk.

We again use our example

Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65 and 67 we need to know the direction of head movement, in addition to the head's current position (53). If the disk arm is moving toward 0, the head will service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124 and 183. Now total head movement is 236.

124 and 183. Now total head movement is 236.



C-SCAN Scheduling

Circular SCAN(C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time.

Like SCAN, C-SCAN moves the head from one end of the disk to other, servicing request along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

LOOK and C-LOOK Scheduling

As we have described, both SCAN and C-SCAN move the disk arm across the full width of the disk.

In practice, neither algorithm is implemented this way. More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

These versions of SCAN and C-SCAN are called LOOK and C-LOOK scheduling, because they look for a request before continuing to move in a given direction.

Disk Management

Disk Formatting

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low level formatting (or physical formatting). Low-level formatting fills the disk with a special data structure for each sector. The data structure for a sector typically consists of a header, a data area (usually 512 bytes in size) and a trailer.

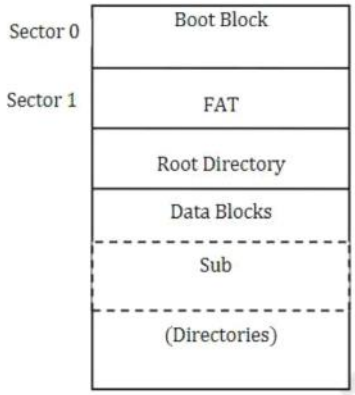
Logical Formatting (Or Creation of a File System)

In this step, the operating system stores the initial file-system data onto the disk. These data structures may include maps of free and allocated space (a FAT or I nodes) and an initial empty directory.

Boot Block

It contains code required to boot the operating system.

For example, MS DOS uses one 512-byte block for its boot program.



FAT: File allocation table, which stores the position of each file in the directory tree.

Root directory: Every file within the directory hierarchy can be specified by giving its path name from the top of the directory hierarchy, the root directory. Such absolute path names consist of the list of directories that must be traversed from the root directory to get the file, with slashes separating the components. The leading slash indicates that the path is absolute that is, starting at the root directory.

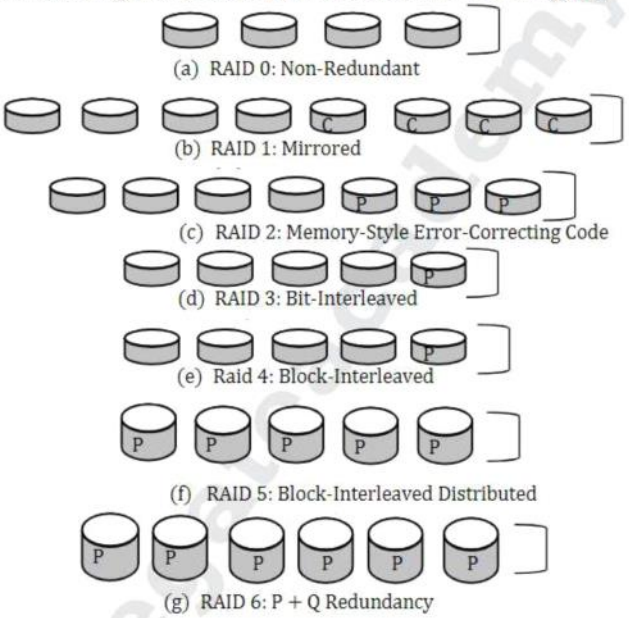
RAID (Redundant Array of Independent Data disk)

The basic idea behind the RAID is to combine multiple small, inexpensive disk drives into array to accomplish performance or redundancy goals not attainable with one large and expensive drive. This array of drives will appear to the computer as single logical storage unit. The main functions of RAID are

- Storing the same data in different places on multiple hard disks and improving the storage performance.
- Providing better throughput.
- Data fault tolerance.
  RAID support 7 levels, RAID 0 to RAID 6, now we will discuss one by one.
- **RAID 0:** The idea behind this level is data to be stored is divided into number of parts called stripe and loaded these strips across the number of disks of array.
- **RAID 1:** RAID Level 1 uses at least two duplicate hard drives and stores the exact same block of information between them. So, it is often called "Mirroring".
- **RAID 2 (Error Correcting Code):** Redundancy scheme of RAID Level 2 is "Hamming Code". The strips are very small in this level, often as a single byte or word. The hamming code is calculated across corresponding bit position on each data disk and the of the code is stored in corresponding bit position on multiple disks.
- **RAID 3:** In RAID Level 3 instead of error correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks. In the event of drive failure, the parity drive is accessed and data is reconstructed from the remaining devices.

- **RAID 4:** This Level uses dedicated parity drive to protect Data disk are striped, as in RAID Level 0. Parity information for the stripe is calculated, and store on a parity disk. If one of the data disk failed then information will be re-build on a separate disk using parity information.
- **RAID 5:** RAID Level 5 is same as RAID Level 4 but only difference is that RAID 5 distribute the parity stripes across all disks.
- **RAID 6:** In RAID Level 6 two different parity calculations are carried out and stored in separate blocks on different disks. So this scheme needs N+2 disks (2 for parity). The advantage of RAID 6 is that it provides high data availability. Three disks would have to fail within the MTTR (Mean time to repair) interval to cause data to become unavailable.

For better understanding about different RAID Level consider below figure.



Solved Examples

Example 1

On a system using a disk cache, the mean access time is 41.2ms, the mean cache access time is 2ms, the mean disk access time is 100ms, and the system has 8Mb of cache memory. If memory is doubled, the miss rate is halved. How much memory must be added to reduce the mean access time to 20ms? Assume the amount of memory may only increase by doubling (8Mb, 16Mb, 32Mb, 64Mb, etc).

Solution:

The Hit Ratio can be computed from the equation

$$2 \times (1 - m) + 100m = 41.2$$
$$2 - 2m + 100m = 41.2$$
$$98m = 39.2$$
$$m = 0.40$$

Doubling memory to 16Mb would lower the Miss Rate to 0.20

$= 2 \times 0.80 + 100 \times 0.20 = 21.6$

An additional doubling to 32Mb would be needed to lower the mean access time below 20ms.

$= 2 \times 0.90 + 100 \times 0.10 = 11.8$

Example 2

A disk has 8 sectors per track and spins at 600rpm. It takes the controller 10ms from the end of one I/O operation before it can issue a subsequent one. How long does it take to read all 8 sectors using the following interleaving systems?

(A) No Interleaving
(B) Single Interleaving
(C) Double Interleaving

Solution:

The disk makes 10 revolutions per second or one revolution in 100ms. In 10ms, less than one sector will have passed under the read/write head.

(A) The next sector cannot be read until the disk makes almost a complete revolution. It will require 8 revolutions to read all 8 sectors. At 100 ms per revolution, it will take 800ms

(B) The next sector will spin under the read/write head almost as soon as the next I/O operation is issued. Two revolutions will be needed to read all 8 sectors, making the total read time 200 ms.

(C) A total of 2.75 revolutions will be needed to read all 8 sectors, for a total time of 275ms.

Example 3

On a disk with 1000 cylinders, number 0 to 999, compute the number of tracks the disk arm must move to satisfy all the requests in the disk queue. Assume the last request serviced was at track 345 and the head is moving toward track 0. The queue in FIFO

serviced was at track 345 and the head is moving toward track 0. The queue in FIFO

order contains requests for the following tracks: 123, 874, 692, 475, 105, 376. Perform the computation for the following scheduling algorithms.

(A) FIFO
(B) SSTF
(C) SCAN
(D) LOOK
(E) C-SCAN
(F) C-LOOK

Solution:

(A) 2013
The tracks traveled will be 345, 123, 874, 692, 475, 105, and 376, making the total distance 222 + 751 + 182 + 217 + 370 + 271 = 2013.

(B) 1298
The tracks traveled will be 345, 376, 475, 692, 874, 123, and 105, making the total distance 529 + 769 = 1298.

(C) 1219
The tracks traveled will be 345, 123, 105, 0, 376, 475, 692, and 874, making the total distance 345 + 874 = 1219

(D) 1009
The tracks traveled will be 345, 123, 105, 376, 475, 692, and 874, making the total distance 240 + 769 = 1009.

(E) 1967
The tracks traveled will be 345, 123, 105, 0, 999, 874, 692, 475, and 376, making the total distance 345 + 999 +623 = 1967.

(F) 1507
The tracks traveled will be 345, 123, 105, 874, 692, 475, and 376, making the total distance 240 + 769 + 498 = 1507.

Example 4

On a system using a disk cache, the mean access time is dependent on the mean cache access time, the mean disk access time and the hit rate. For the following what is the mean access time?

(A) Cache: 1ms; Disk: 100ms; Hit rate: 25%
(B) Cache: 1ms; Disk: 100ms; Hit rate: 50%

Solution:

(A) $100\times0.75 + 1\times0.25 = 75.25$
(B) $100\times 0.50 + 1\times 0.50 = 50.50$

Example 5

A disk has 19,456 cylinders, 16 heads, and 63 sectors per track. The disk spins at 5400rpm seek time between adjacent tracks is 2ms. Assuming the read/write head is already positioned at track 0, how long does it takes to read the entire disk?

Solution:

Each track can be read in one revolution.
5400 revolution in 60 seconds

1 revolution in $\Rightarrow \frac{60 \text{ seconds}}{5400}$

1 revolution in $\Rightarrow \frac{60 \text{ seconds}}{5400}$

$$= \frac{60 \times 1000\text{ms}}{5400} = 11.11 \text{ ms}$$

A track can be read in 11.11 ms
To Read all 19456 × 16 tracks requires approximately 3459 seconds
Seek Time = $(19456 - 1) \times 2 = 39$ s
Total Time = (3459+39) seconds =3498s = 58.3 minute ≈ 58 minutes

Example 6

When multiple interrupts from different devices appear at about the same time, a priority scheme could be used to determine the order in which the interrupts would be serviced. Discuss what issues need to be considered in assigning priorities to different interrupts

Solution:

A number of issues need to be considered in order to determine the priority scheme to be used to determine the order in which the interrupts need to be serviced. First, interrupts raised by devices should be given higher priority then traps generated by the user program; a device interrupt can therefore interrupt code used for handling system calls.
Second, interrupts that simply perform tasks such as copying data served up a device to user/kernel buffers, since such tasks can always be delayed.
Third, devices that have real-time constraints on when its data is handled should be given higher priority than other devices.
Also, devices that do not have any form of buffering for its data would have to be assigned higher priority since the data could be available only for a short period of time.

Example 7

What are the various kinds of performance overheads associated with servicing an interrupt?

Solution:

When an interrupt occurs the currently executing process is interrupted and its state is stored in the appropriate process control block. The interrupt service routine is then dispatched in order to deal with the interrupt.
On completion of handling of the interrupt, the state of the process is restored and other process will be resumed. Therefore, the performance over heads include the cost of saving and restoring process state and the cost of flushing the instruction pipeline and restoring the instructions into the pipeline when process is restarted.

Example 8

Which disk-scheduling algorithm would best optimize the performance of a RAM disk?

Solution:

All the disk-scheduling algorithm would yield the same device performance. A RAM disk is a virtual device created from main memory. The order in which requests are handled has no effect on performance.

Example 9

SSTF favors tracks in the center of the disk on a system how might this affect the design of the file system?

Solution:

Certain blocks in the file system tend to get heavier usage than others. File system data structure and directories located near the top of the file system design should place these blocks near the center of the disk.