PS01CMCA54 Operating Systems

Dr. J. V. Smart

Table of Contents

- File System Manipulation Commands
  - Shell Globbing Patterns
- Plain Text Editors
- Brief Overview of Working with the Vim Editor
- The Shell Scripting Language of the *bash* Shell
  - Shell Variables
  - The `test` Command and the `[` Builtin
  - The Control Structures
  - Comments
  - The *Shebang* Line
  - Output
  - Input
  - Integer Arithmetic
  - String Operations
- Standard I/O Streams and I/O Redirection
  - Default Assignment of Standard I/O Streams
  - Standard Output Redirection
  - Standard Input Redirection
  - Standard Input and Standard Output Redirection
  - Standard Error Redirection
  - Supplying the standard input from the command line
  - Command Substitution
  - Pipes
  - Redirecting One File Descriptor to Another File Descriptor
- Filters
- Regular Expressions
  - Basic Regular Expressions
  - Extended Regular Expressions
- The grep Family of Commands
- Combining Multiple Commands using Logical Operators
- Vim Editor in More Detail
- Commands List

Syllabus

Syllabus with effect from the Academic Year 2021-2022

Course Code	PS01CMCA54	Title of the Course	OPERATING SYSTEMS
Total Credits of the Course	4	Hours per Week	4

Course Objectives:	1. To provide basic understanding of the role and functioning of an operating system.

	2. To introduce Linux shell environment and programming.

Course Content

Unit	Description	Weightage* (%)
1.	**Introduction to Operating Systems** - Understanding the role of operating systems - Operating system services - Operating system structure - The concepts of interrupt handling, system call, shell, operating system interface - Virtual machines - Linux Bash shell programming fundamentals - Command-line processing - Bash shell variables, control structures - input, output, integer arithmetic, string operations	25
2.	**Process Management** - The concept of a process - Scheduling of processes - Interprocess communication - Multithreading: concepts, advantages, models - Schedulers: long term, middle term, short term - CPU scheduling: criteria and algorithms - Multiprocessor scheduling - Introduction to process synchronization - The critical section problem and Peterson's solution - The concepts of semaphores and monitors - Introduction to deadlocks	25
3.	**Memory Management and File Systems** - Basic concepts of memory management - Paging - Segmentation - Virtual memory, demand paging - Page replacement - Introduction to file system management and directory structure - File system mounting - Disk scheduling	25
4.	***Linux Shell Programming* - The vim editor - File system manipulation commands - I/O redirection - Regular expressions - Basic filters - The sed and awk commands	25

Teaching-Learning	Blended learning approach incorporating traditional

Methodology	classroom teaching as well as online / ICT-based teaching practices

Evaluation Pattern

Sr. No.	Details of the Evaluation	Weightage
1.	Internal Written / Practical Examination (As per CBCS R.6.8.3)	15%
2.	Internal Continuous Assessment in the form of Practical, Viva-voce, Quizzes, Seminars, Assignments, Attendance (As per CBCS R.6.8.3)	15%
3.	University Examination	70%

Course Outcomes: Having completed this course, the learner will be able to	
1.	describe the role and functioning of an operating system.
2.	demonstrate understanding of fundamental concepts related to operating systems.
3.	understand process, memory and file system management.
4.	gain familiarity with Linux command line environment.
5.	use basic Linux commands.
6.	develop Linux shell scripts.

Suggested References:

Sr. No.	References
1.	Silbetschatz, Galvin, Gagne: Operating System Concepts, 8th edition, John Wiley and Sons, Inc., 2008
2.	Kochan S. G., Wood, P. : Unix Shell Programming, 4th edition, Addison Wesley, 2016
3.	Das S. : UNIX and Shell Programming, Tata McGraw-Hill Education, 2008
4.	Nutt G. : “Operating Systems” : 3rd Edition, Pearson Education, 2004
5.	Tanenbaum A. S., Woodhull A.S. : “Operating Systems Design and Implementation”, 3rd edition, Prentice Hall, 2006
6.	Shotts W. : “The Linux Command Line: A Complete Introduction Illustrated Edition”, 2nd Edition, No Starch Press, 2019

Glossary

Kernel

The operating system kernel is the one program running on the computer system at all times. The kernel provides the interface between the hardware and the programs

Shell

The shell provides an interface between the operating system and the user

Multiprocessor System

A computer system having multiple processors (CPUs)

Multiuser Operating System

An operating system that allows multiple users to work on the system simultaneously (at the same time)

Multiprogramming Operating System

An operating system that can run multiple programs simultaneously (at a time)

Multitasking (Time Sharing) Operating System

An operating system that can run multiple programs simultaneously (at a time) and switch between them so fast that users can use the programs interactively

Program

A computer program is a sequence of instructions to the computer. A program is stored in secondary storage (usually a hard disk). A program is a passive entity

Job

A job is a program that has been submitted for execution by a user. Jobs are kept in a job pool on the disk

Process

A process is a program in the state of execution. A process is an active entity

Shell

A shell is an interface between the user and the operating system. A shell provides either a Graphical User Interface (GUI) or a Command Line Interface (CLI)

Loader

The loader is the operating system component that loads a job into the main memory

Job Scheduler

When all the jobs submitted by the user(s) cannot be loaded into the main memory, the job scheduler selects the jobs to be loaded into main memory. The job scheduler is also known as the long term scheduler

CPU Scheduler

Whenever a CPU becomes free for executing a process, the CPU scheduler selects one process for execution out of all the processes that are currently loaded in the main memory and are ready to execute. It does this based on some criteria and according to some CPU scheduling algorithm. The CPU scheduler is also known as the short term scheduler

Swapper

When all the jobs cannot be loaded in to the main memory but the system does not want to keep any job waiting for an extended period of time because they are interactive jobs, swapping is used. With swapping, processes are loaded into the main memory, executed for some time, then copied back into a swap area on the disk so that another job or process could be loaded. All processes are continuously shuffled between the main memory and the swap area on the disk to give the illusion that all of

them have been loaded into the main memory at a time. Swapping is handled by the swapper, also known as medium term scheduler

Long Term Scheduler
See Job Scheduler

Medium Term Scheduler
See Sapper

Short Term Scheduler
See CPU Scheduler

Memory Protection
An operating system with memory protection does not allow a process to access any memory that is not allocated to it or shared with it. Memory protection prevents a process from seeing or modifying memory belonging to another process

CPU Protection
CPU protection prevents one process from monopolizing the CPU so that other processes do not get a chance to execute on the CPU for a long time

I/O Protection
To prevent the errors and problems that may be caused by multiple processes trying to access an input / output (I/O) device at the same time and to prevent unauthorized access to I/O devices by processes, all input output operations are controlled by the operating system. This is known as I/O protection

Dual Mode (Multimode) Operation
A modern CPU is capable of executing in two (dual mode) or more (multimode) modes. Different modes have different privileges (rights). Some instructions can only run in particular mode

Privileged / Supervisor / Kernel Mode
This mode can execute both privileged (for example, I/O instructions) and non-privileged (normal) instructions. The privileged instructions can only execute in this mode. Only the operating system kernel runs in this mode

Non-privileged / User Mode
Only non-privileged (normal) instructions can be executed in this mode. Privileged instructions cannot be run in this mode. All user processes run in this mode

Introduction

Defining Operating Systems

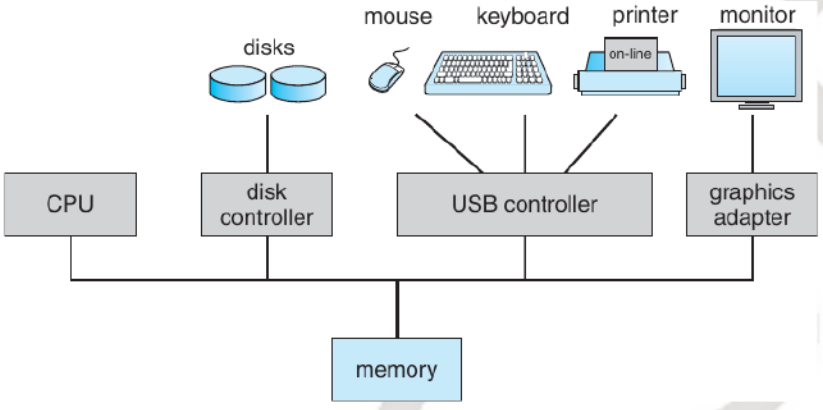
- The fundamental goal of computer systems is to execute user programs (application programs) and to make solving user problems easier
- Toward this goal, computer hardware is constructed. But bare hardware alone is extremely difficult to use without assistance from any ready-made software. Hence, some system programs are developed
- These programs require certain common operations, such as those controlling the I/O devices. The common functions of controlling and allocating resources are then

brought together into one piece of software: the operating system

- There is no universally accepted definition of what is part of the operating system
- A simple viewpoint is that it includes everything a vendor ships when you order *the operating system* (or everything that gets downloaded when you download *the operating system*)
- The features included, however, vary greatly across systems. Some systems take up less than 1 megabyte of space and lack even a full-screen editor, whereas others require gigabytes of space and are entirely based on graphical windowing systems
- A more common definition, and the one that we usually follow, is that the operating system is the one program running at all times on the computer-usually called the *kernel*
- Along with the kernel, there are two other types of programs
    - System programs, which are associated with the operating system but are not part of the kernel
    - Application programs, which include all programs not associated with the operation of the computer system

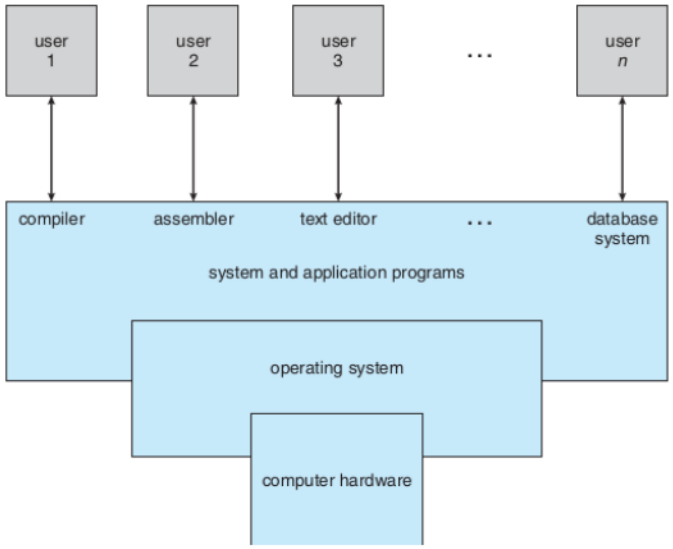
What is an operating system?

- An operating system acts as an intermediary between the user of a computer and the computer hardware (Note: you will learn later that the *operating system kernel* acts as an intermediary between the computer hardware and programs, while the *shell* acts as an intermediary between the operating system kernel and the users. Depending on which definition you use, the shell may or may not be considered part of the operating system)
- The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner
- An operating system is software that manages the computer hardware
- An operating system acts as the controller, allocator and manager of resources
- An operating system provides a basis (the basic and common functions) for application programs to execute



Architecture of a Computer System

What Operating Systems Do



Components of a Computer System

A computer system can be divided roughly into four components: the hardware, the operating system, the application programs and the users. The operating system controls the hardware and coordinates its use among the various application programs for the various users. An operating system performs no useful function by itself. It simply provides an environment within which other programs can do useful work.

Operating Systems from Different Viewpoints

User View

The user's view of the computer varies according to the interface being used.

- **Personal Computer** Most computer users sit in front of a PC, consisting of a monitor/ keyboard/ mouse, and system unit. Such a system is designed for one user to monopolize its resources. The goal is to maximize the work (or play) that the user is performing. In this case/ the operating system is designed mostly for ease of use, with some attention paid to performance and none paid to resource utilization - How various hardware and software resources are shared. Performance is, of course, important to the user; but such systems are optimized for the single-user experience rather than the requirements of multiple users.
- **Remote Terminal** In other cases, a user sits at a terminal connected to a mainframe or a minicomputer. Other users are accessing the same computer through other terminals. These users share resources and may exchange information. The operating system in such cases is designed to maximize resource utilization- to assure that all available CPU time, memory, and I/0 are used efficiently and that no individual user takes more than her fair share.
- **Workstations** In still other cases, users sit at workstations connected to networks of other workstations and servers. These users have dedicated resources at their disposal, but they also share resources such as networking and servers-file, compute, and print servers. Therefore, their operating system is designed to compromise between individual usability and resource utilization.
- **Handheld Computers** Recently, many varieties of handheld computers have come

into fashion. Most of these devices are standalone units for individual users. Some are connected to networks, either directly by wire or (more often) through wireless modems and networking. Because of power, speed, and interface limitations, they perform relatively few remote operations. Their operating systems are designed mostly for individual usability, but performance per unit of battery life is important as well.

- **Embedded Systems** Some computers have little or no user view. For example, embedded computers in home devices and automobiles may have numeric keypads and may turn indicator lights on or off to show status, but they and their operating systems are designed primarily to run without user intervention.

System View

From the computer's point of view, the operating system is the program most intimately involved with the hardware. In this context, we can view an operating system as a . A computer system has many resources that may be required to solve a problem: CPU time, memory space, file-storage space, I/0 devices, and so on. The operating system acts as the manager of these resources. Facing numerous and possibly conflicting requests for resources, the operating system must decide how to allocate them to specific programs and users so that it can operate the computer system efficiently and fairly. As we have seen, resource allocation is especially important where many users access the same mainframe or minicomputer. A slightly different view of an operating system emphasizes the need to control the various I/0 devices and user programs. An operating system is a manages the execution of user programs control program. A to prevent errors and improper use of the computer. It is especially concerned with the operation and control of I/O devices.

- Personal computers
  - Single user monopolizes the resources
  - Ease of use is very important
  - Performance and resource utilization are less important
- Mainframe / minicomputer
  - Several users share resources
  - Performance is a critical issue
  - Expensive computer systems. Hence resource utilization is very important
- Powerful workstations used to access Shared Resources on Server Computers
  - Powerful resources at the user's disposal, ease of use should be provided
  - Shared resources on the server computers must be utilized efficiently
  - A compromise between individual usability and resource utilization
- Hand-held devices
  - PDAs (Personal Digital Assistants), Smartphones, Tablets
  - Devices meant for use by one person
  - Nice user interface, usability / ease of use very important
  - Limited resources, performance and conservation of power also very important
- Embedded systems
  - Hardly any user interface
  - Very limited resources
  - Performance very important
  - Designed for use without user intervention

System View of an Operating System

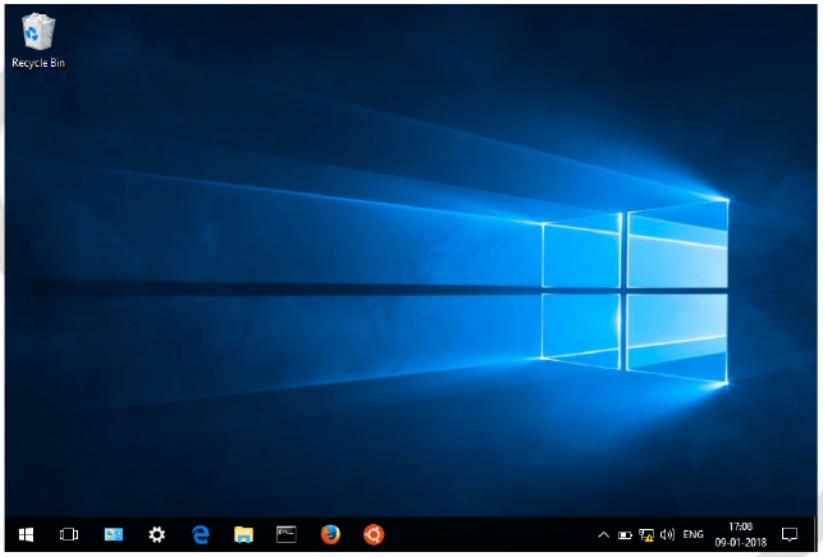
- Resource allocator / manager
- Control program

An Operating System is-

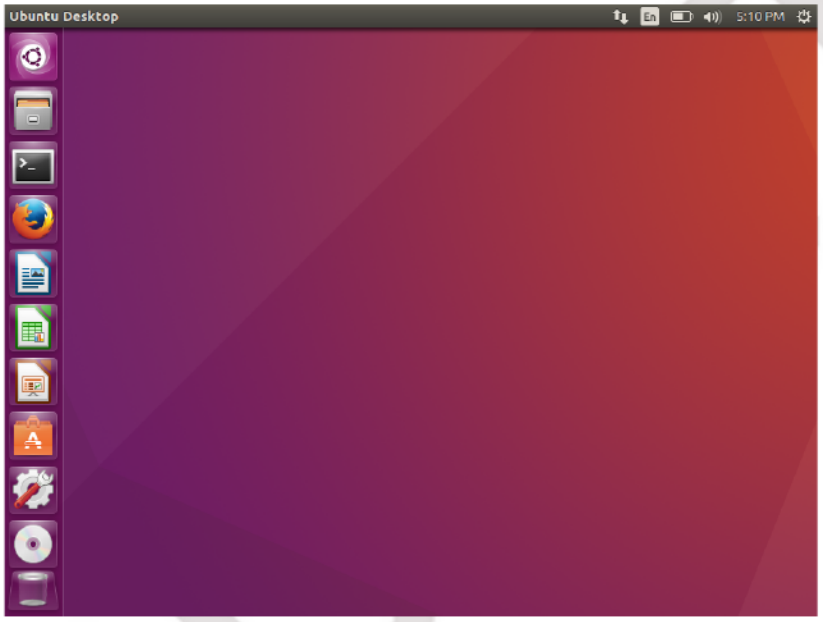
- A program that acts as an intermediary between a user of a computer and the computer hardware
- Goals of an Operating System
    - Make the computer system convenient to use
    - Execute user programs and make solving user problems easier
        - User programs require some common services like memory management, etc. These common services are bunched togather in the operating system
    - Use the computer hardware in an efficient manner
- Resource manager
- Control program
- Provider of common services
- Everything you get when you obtain an "operating system"
    - However, all that comes with an operating system may not be essential parts of an operating system
    - There is a wide variation in what comes with an operating system.
- The one program running at all times on the computer (the ***kernel***)
    - The *kernel* is the program that is the first to start execution when the computer boots and remains in execution till the computer shuts down
    - Everything else is either
        - A system program
        - An application program

The Shell

- The shell acts as an interface between the user and the operating system
- The shell provides the user interface through which the users can start and stop programs
- The shell can be text-mode (Command Line Interface - CLI) or graphical mode (Graphical User Interface - GUI)
- File Explorer / Windows Explorer is the graphical shell that comes with Microsoft Windows
- cmd is the text-mode shell supplied along with Microsoft Windows
- Linux systems offer a variety of shells in both the GUI mode (GNOME Shell, KDE, Unity, XFCE, LXDE, MATE, etc.) and the text mode (sh, bash, ksh, csh, etc.)

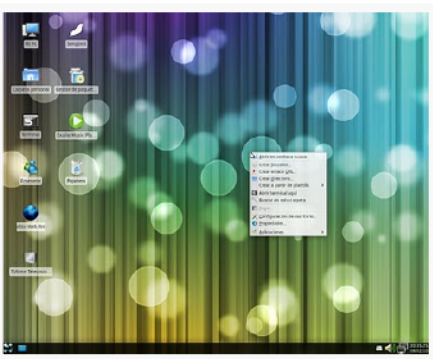


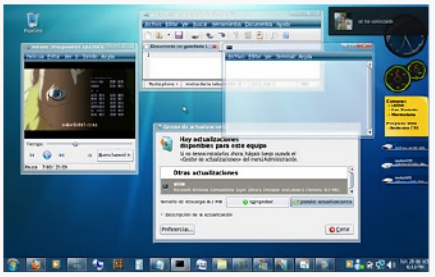
The Windows 10 File Explorer



The Ubuntu Unity Shell

- Most people identify the operating system with the shell because it is the shell that provides the user interface visible to the user. However, the shell is changeable. Hence, a particular shell cannot be considered an essential part of the operating system. This gives credence to the theory that only the kernel is the operating system and rest of the things supplied with the operating system should be considered system programs and application programs.

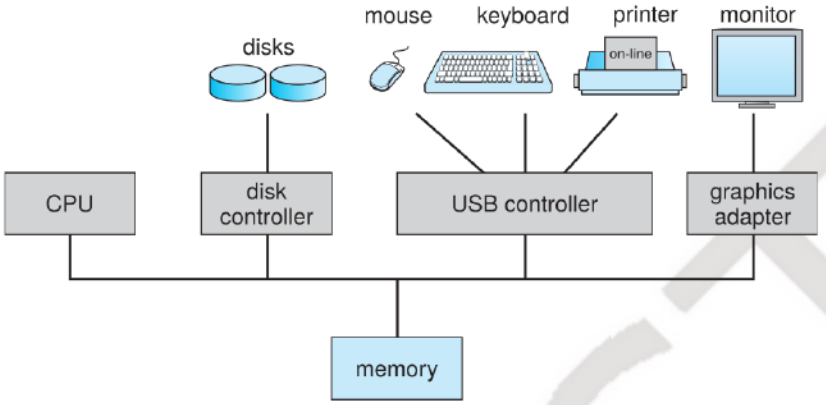




Fundamentals of Computer Architecture and operations

Computer System Operation

A modern general-purpose computer system consists of one or more CPUs, the main memory and a number of device controllers connected through a common bus.



Computer System Operation

Interrupts

- Only the CPU and DMA (Direct Memory Access) controller can access the main memory directly
- Whenever any event occurs at a controller, the CPU has to execute some code to process that event. For example, when a key is pressed, the code of the key that has been pressed needs to be stored into the keyboard buffer in the main memory. Similarly, when the network controller receives a packet from the network, its contents must be transferred into some main memory buffer. However, the CPU would be executing some other program at that time. Whenever an event occurs at a controller, the controller interrupts the CPU by putting a voltage on an interrupt line in the system bus. Different controllers are allotted different interrupt numbers, so that the CPU can identify which interrupt occurs. When this happens, the CPU suspends the execution of the currently running program. It saves the current state of execution. It then executes the ISR (Interrupt Service Routine) corresponding to the interrupt number. The Interrupt Service Routines for different interrupts are small pieces of software. After executing the ISR, the CPU restores the state of execution and resumes the execution of the program that was being executed from where it was left off. Hence the program executes as if it was never interrupted.

- While some ISRs may be in the BIOS, most are installed by the operating system. Hence when an interrupt occurs, the control of execution indirectly transfers to the operating system.
- Interrupts can be generated in three ways
  - A hardware device may generate an interrupt by placing a voltage on some interrupt lines when it needs the attention of the CPU (hardware interrupt)
  - The CPU itself generates an interrupt when it detects an attempt to perform an illegal operation like division by zero or illegal memory access (trap)
- Interrupts can also be generated from software, using special machine language or assembly language instructions or *system calls* (software interrupt)

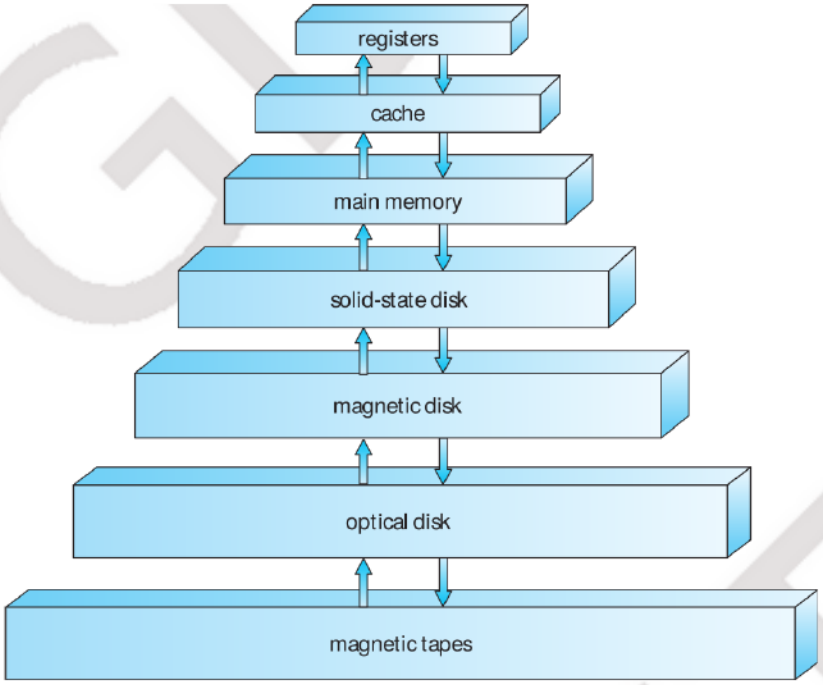
Device Drivers

- Different devices (even of the same type) require different coding for their operation
- The code to handle a particular type of device is a called device driver
- A device driver is a low-level software
- Device drivers for the common hardware come bundled with the operating system, while drivers for other devices may have to be installed separately

The System Boot Process

- The CPU and other components of the computer system are powered on
- The CPU executes a firmware program in ROM, either BIOS (Basic Input Output System) or UEFI (unified Extensible Firmware Interface)
- The firmware tests the hardware during POST (Power On Self Test)
- A boot loader is loaded from the secondary storage
- The boot loader may allow the user to choose the operating system to be booted by presenting a menu
- The boot loader loads an operating system kernel
- The kernel loads necessary device drivers and several other system programs
- The shell is launched, which acts as an interface between the user and the operating system and lets the user launch programs

The Storage Device Hierarchy



The Storage Device Hierarchy

The storage is divided into three main categories:

- The Primary Storage
  - Fast, but low capacity storage
  - Directly accessible to the CPU
  - Random access storage, individual bytes or memory words can be accessed
  - Includes registers, caches and main memory
  - Volatile, loses contents when the power supply is removed
- The Secondary Storage
  - Slow, but high capacity storage
  - Cannot be directly accessed by the CPU. To use its contents, the CPU must first bring them into main memory. This process is called *loading*
  - Random access storage, entire blocks can be accessed after loading into main memory. It is not possible to access bytes individually
  - Includes solid state disks, hard disks and optical disks
  - Non-volatile, does not contents even when the power supply is removed
- The Tertiary Storage
  - Sequential access devices
  - Magnetic tape is the prime example of tertiary storage
  - useful for backup

How a hard Drive Works (Slow Motion Video)

Single-Processor v/s Multiprocessor Systems

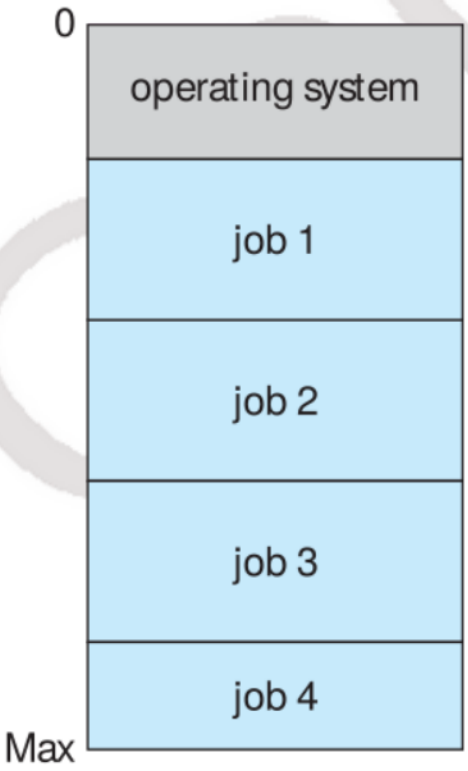
- Single-Processor Systems
- Multiprocessor Systems
  - Advantages
    - **Increased throughput:** By increasing the number of processors, we may get more work done in the given time frame. However, if we have *n* CPUs in the system, the increase in performance would be somewhat less than *n* due to the overhead of multiprocessing
    - **Economy of scale:** A single system with *n* processors may cost less than *n* systems with 1 processor
    - **Increased reliability:** If properly designed, failure of one system should not halt the system, but only result in a degradation of performance
  - Types of Multiprocessing
    - **Asymmetric Multiprocessing:** One of the CPUs acts as the *master* CPU, while other CPUs are *slave* CPUs. The master CPU controls the operations of the slave CPUs. If a slave CPU fails, the system can still survive; but if the master CPU fails, it cannot
    - **Symmetric Multiprocessing (SMP):** There is no master-slave relationship. All processors are peers. Achieving coordination among the CPUs is a complex task. But there no single point of failure. If any of the CPUs fail, the system can still survive
    - Almost all modern operating systems, including Windows, Mac OS X and

Linux support SMP

- Multicore Systems
  - These systems have multiple *processing cores* on a single processor die. It is like having multiple CPUs on a single chip. For most purposes, a system with *n* core is just like a multiprocessor system with *n* CPUs. Multicore systems are also multiprocessor systems

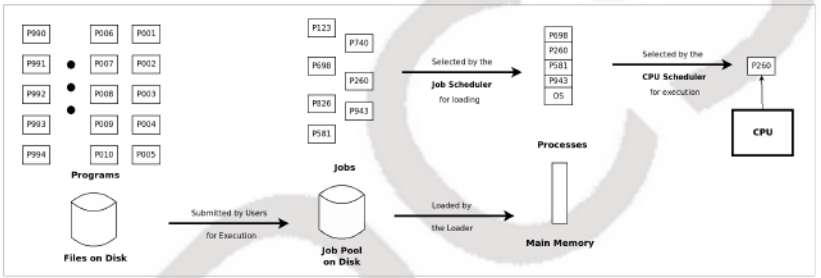
Operating System Structure

- The concept of a process
- Single-tasking Operating Systems
- Multiprogramming Operating Systems
  - CPU bursts and I/O bursts
  - Increase CPU utilization



Memory Layout for a Multiprogramming System

- Multitasking (Time-sharing)
  - Short response time
  - Designed for interactive systems
- Programs, jobs and processes



Programs, Jobs and Processes

- Job scheduling
- CPU scheduling
- Memory management
- I/O Management

Operating System Operation

- **Process management**
- **Memory management**
- **Storage management**
- **File-system management**
- **Caching**

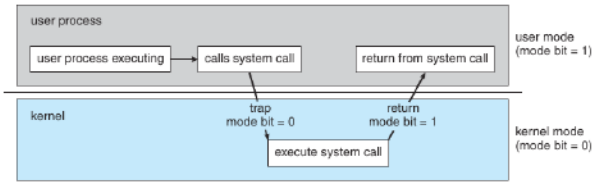
Protection and Security

CPU protection

- Hardware timer

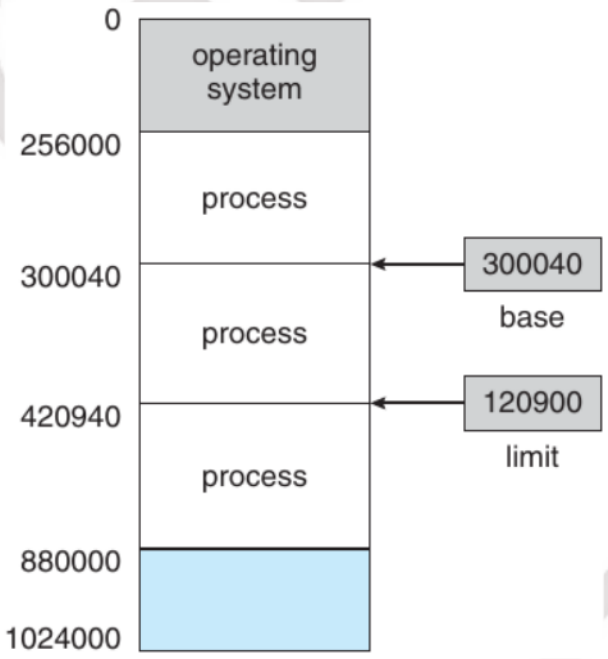
I/O protection

- Dual-mode / multimode operation

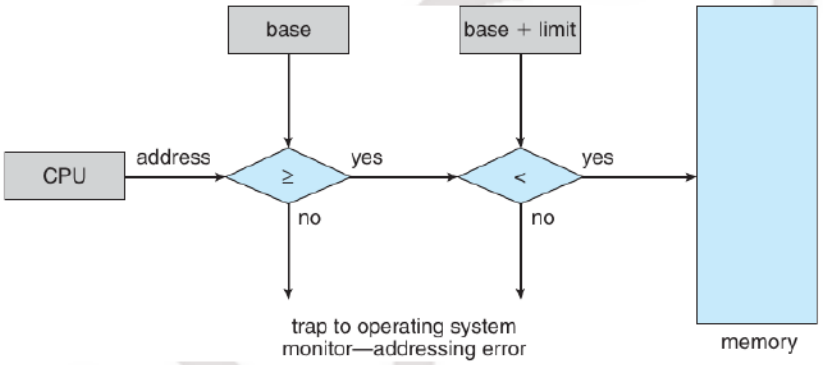


Dual Mode Operation

Memory protection



Memory management using Base and Limit Registers



Memory management using Base and Limit Registers

Security

Computing Environments

Process Scheduling

Non-preemptive v/s Preemptive Scheduling

- The situations in which we have to take a decision regarding which process should be given the CPU (i.e. the CPU scheduler will be invoked)
  i. When a process terminates
  ii. When the CPU burst time of the currently running process is over and it performs either some I/O operation or goes for waiting for some event
  iii. When a new process arrives in the ready queue
  iv. When the allocated time slice or time quantum of the currently running process is over

The CPU scheduler is always invoked in situations `i` and `ii`, whether the CPU scheduling is preemptive or non-preemptive. The CPU scheduler is invoked in situations `iii` and `iv` only in case of preemptive scheduling, not in case of non-preemptive scheduling

Process Synchronization

Peterson's Solutuion to the Two-process Critical Section Problem

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

    CRITICAL SECTION

    flag[i] = FALSE;
```

```
    REMAINDER SECTION
}
```

Process P_i

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

    CRITICAL SECTION

    flag[j] = FALSE;

    REMAINDER SECTION
}
```

Process P_j

Synchronization Hardware

- On a single-processor system, one may disable interrupts
- swap hardware instruction (atomic execution)
  - swap(*a*, *b*) will swap (interchange) the values of *a* and *b* atomically (as a single instruction)
- test-and-set hardware instruction (atomic execution)

Semaphores

- A semaphore is an integer variable
  - wait() (atomic decrement operation)
  - signal() (atomic increment operation)
- Binary semaphore
  - Only two possible values (0 and 1)
  - Also called *mutex* because it is frequently used to provide mutual exclusion
- Counting semaphore
  - Can hold *any* integer value

Solution to critical section problem using (binary) semaphores

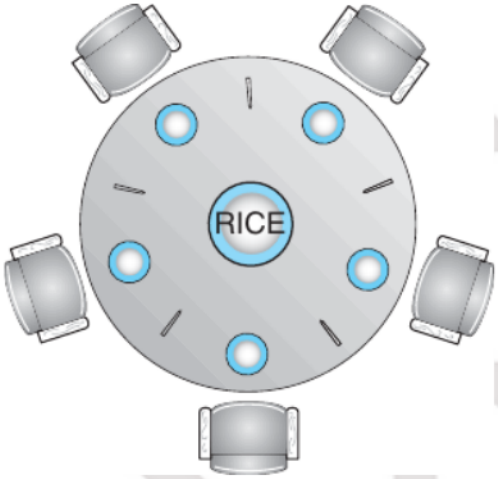
```
Semaphore S; // initialized to 1

wait (S);
    Critical Section
signal (S);
```

Classical Problems of Synchronization

- Bounded-Buffer Problem (Producer-consumer problem with finite buffer size)
  - There is a producer process that produces data items and puts them in a buffer

    - There is a consumer process that consumes (uses) data items produced by the producer process from the buffer
    - Both processes run ***concurrently*** (simultaneously)
    - The buffer is ***shared***
    - The buffer has a *finite size*
- Readers and Writers Problem
    - There is a ***shared*** data item
    - There are *m* reader processes that want to read the data item
    - There are *n* writer processes that want to write (modify) the data item
    - Both reader and writer processes run ***concurrently*** (simultaneously)
- Dining-Philosophers Problem
    - Mutual exclusion (one chopstick can be picked up by only one philosopher at a time)
    - Progress (if one or more philosophers want to eat, at least one of them should be able to eat)
    - Bounded waiting



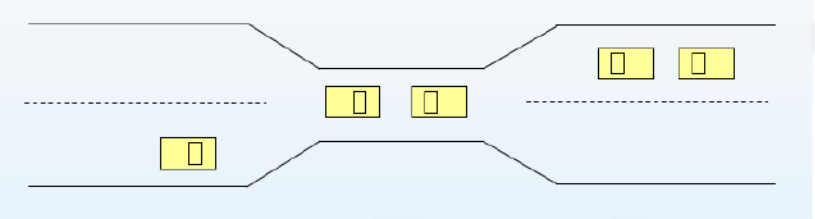
The Dining Philosophers Problem

Monitors

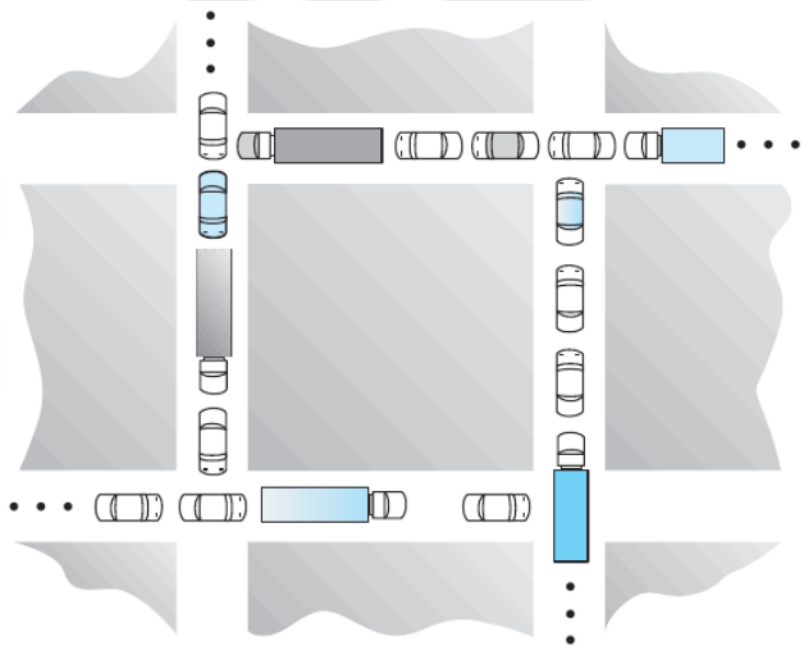
- A monitor is a high-level process synchronization tool
- To execute code in a critical section, a process must *acquire* the monitor on the shared object being used
- At a time only one process can acquire the monitor on a given object
- Used by various high-level languages for synchronization among threads. E.g. Java *synchronized* keyword

Deadlocks

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set is said to be *deadlocked*



Deadlock on a Bridge



Deadlock at Crossroads

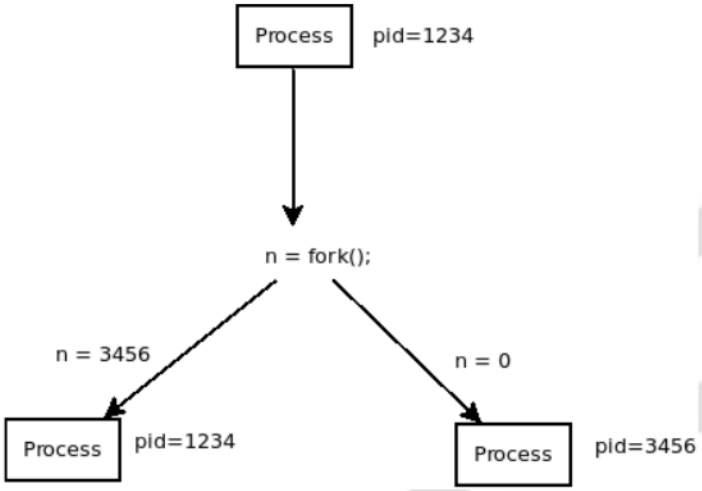
Characteristics of Deadlock

- There are more than one processes
- There are one or more types of resources
- There may be different number of resources of each type
- Processes require different number of resources of different types to function

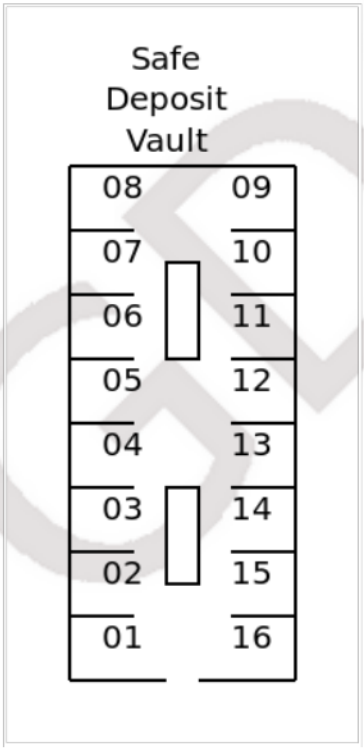
Deadlock can arise if the following four conditions hold *simultaneously*

- **Mutual exclusion** only one process at a time can use a resource
- **Hold and wait** a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption** a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait** there exists a set $\{P_0, P_1, ..., P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, ..., P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0

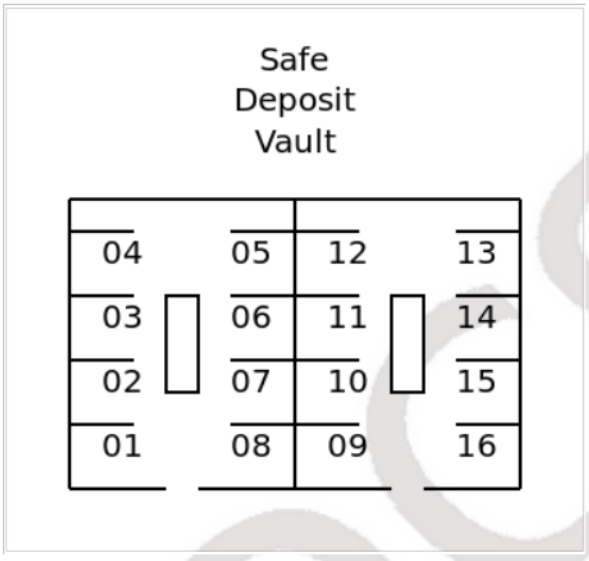
Figures



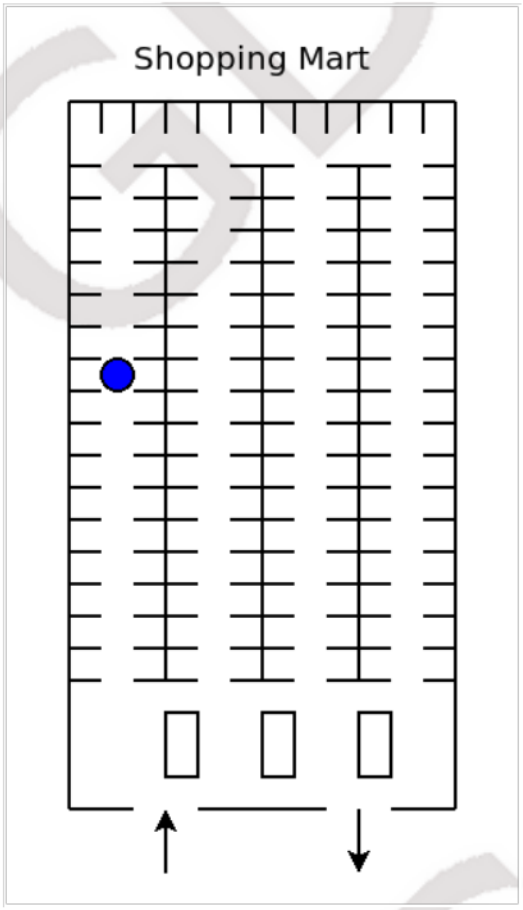
fork



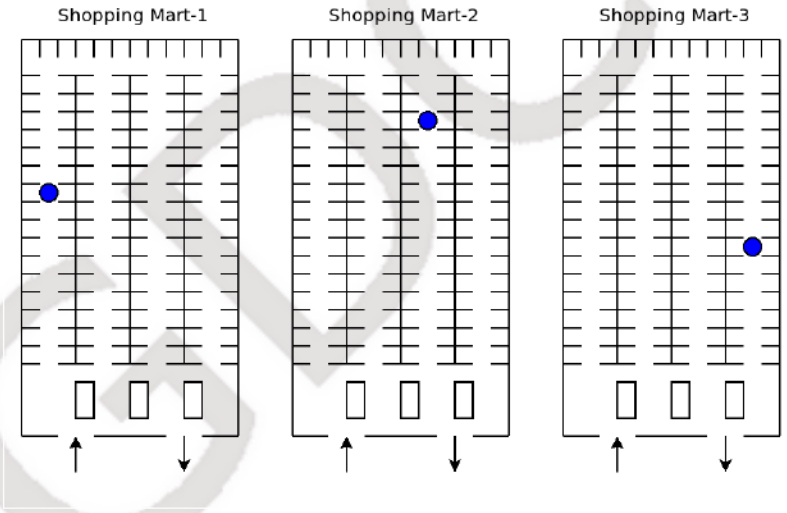
Multithreading Safe Deposit Vault-1



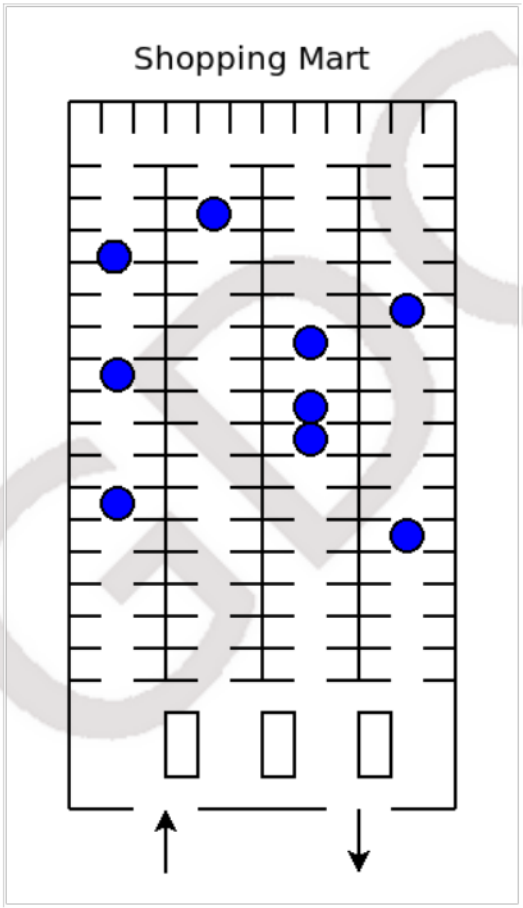
Multithreading Safe Deposit Vault-2



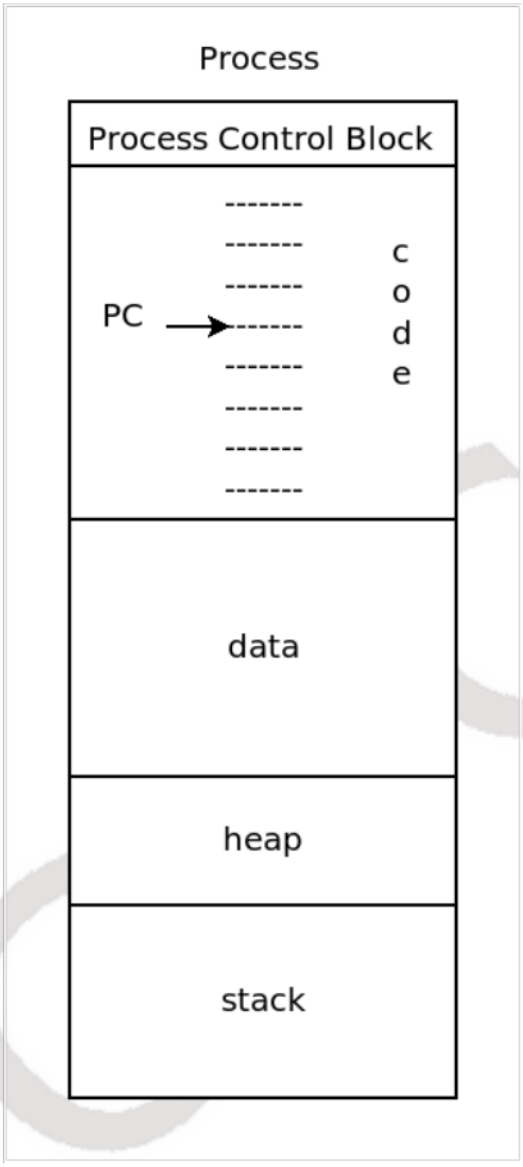
Multithreading Shopping Mart-1



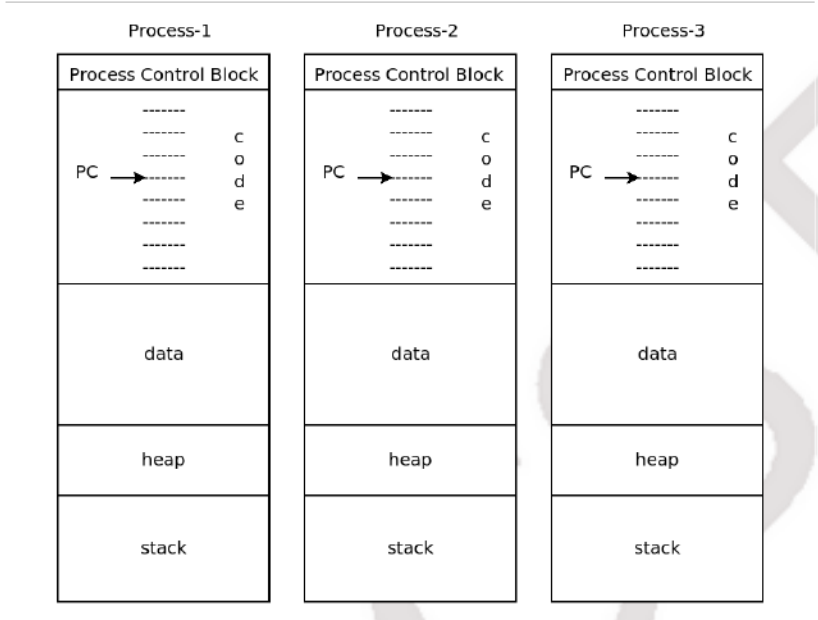
Multithreading Shopping Mart-2



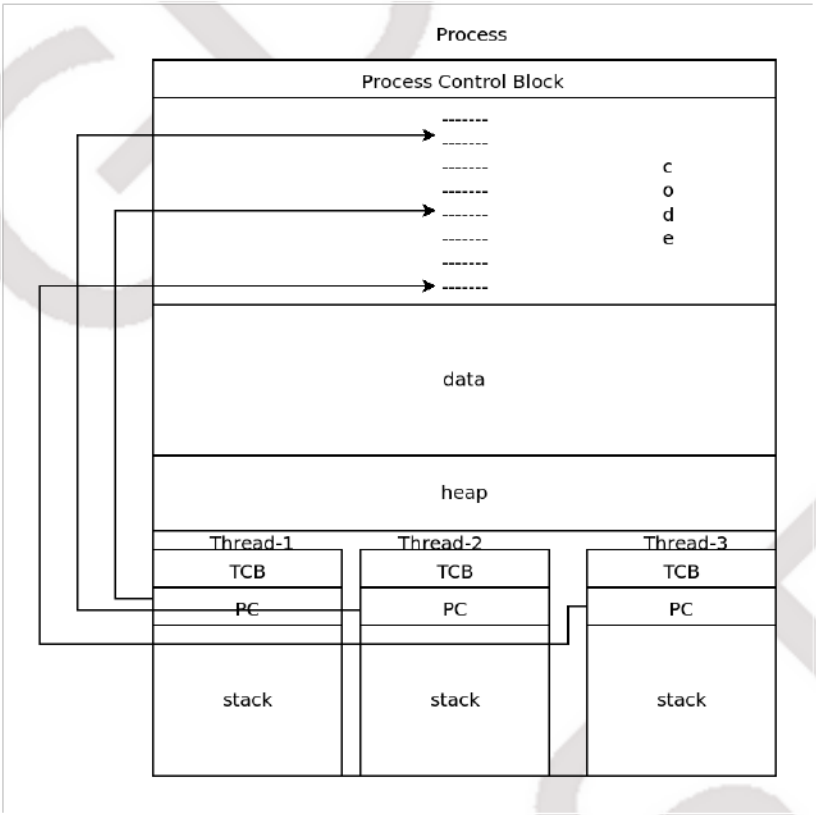
Multithreading Shopping Mart-3



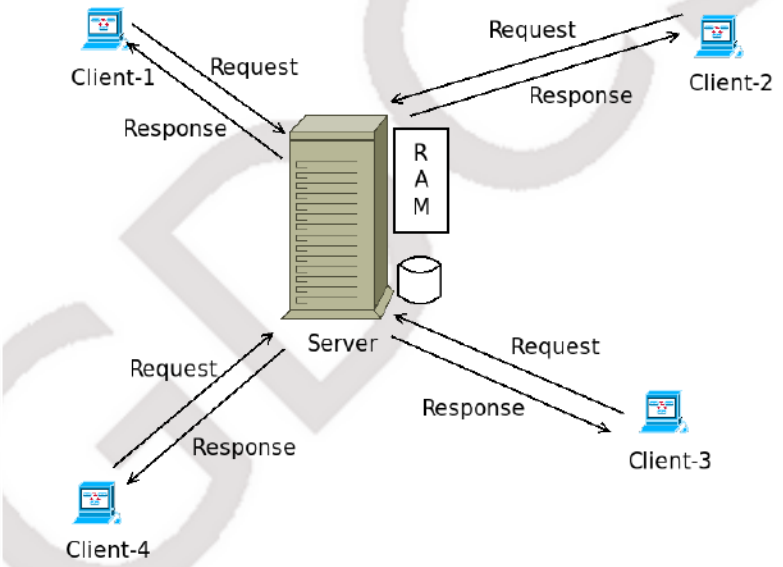
Multithreading-1



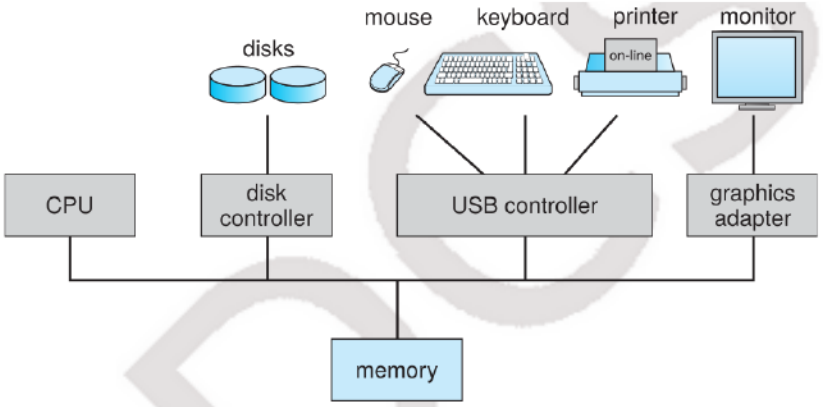
Multithreading-2



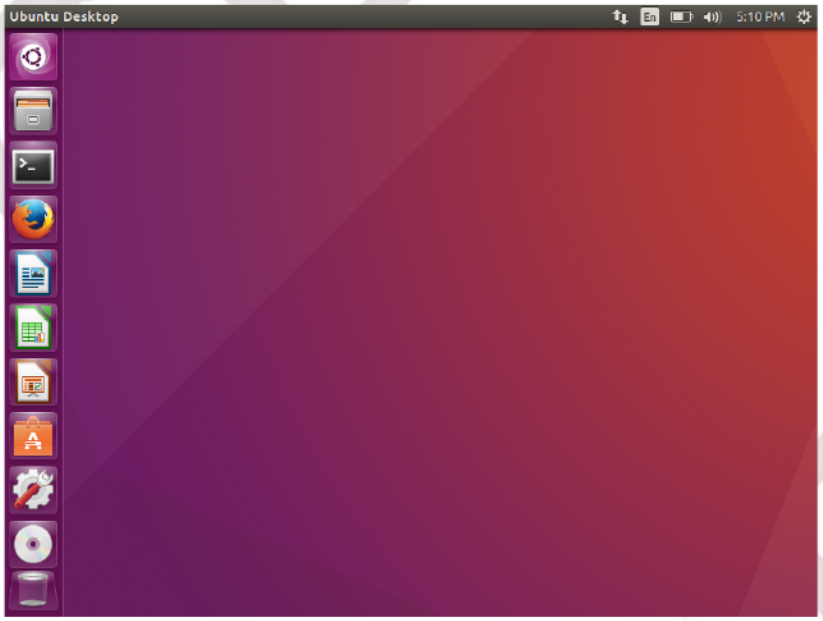
Multithreading-3



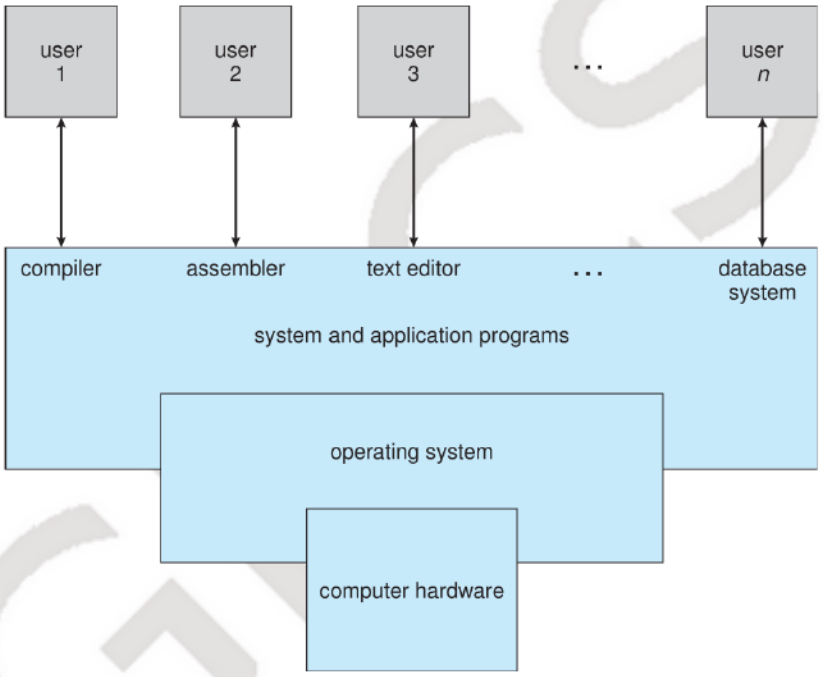
Multithreading-0



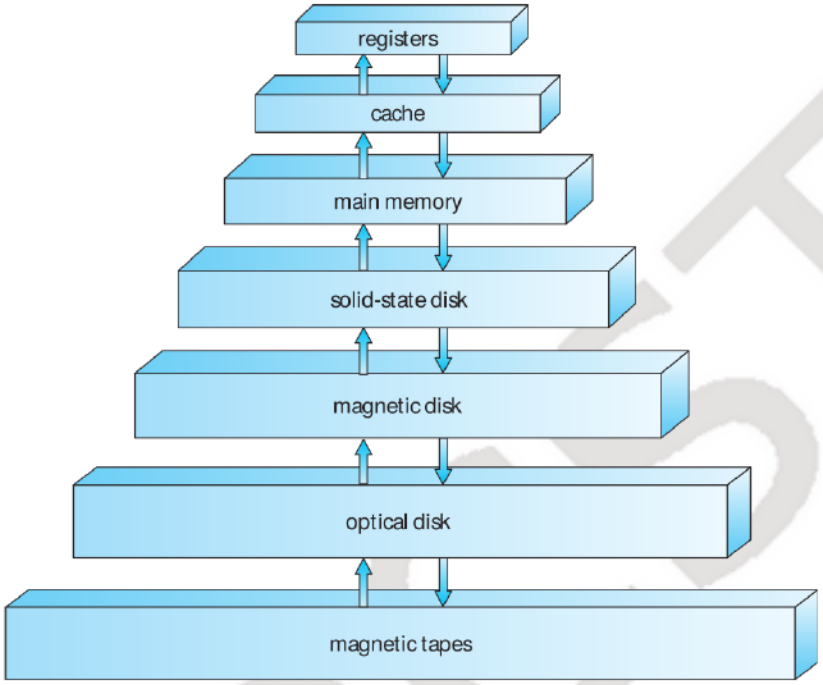
Computer_System_Operation



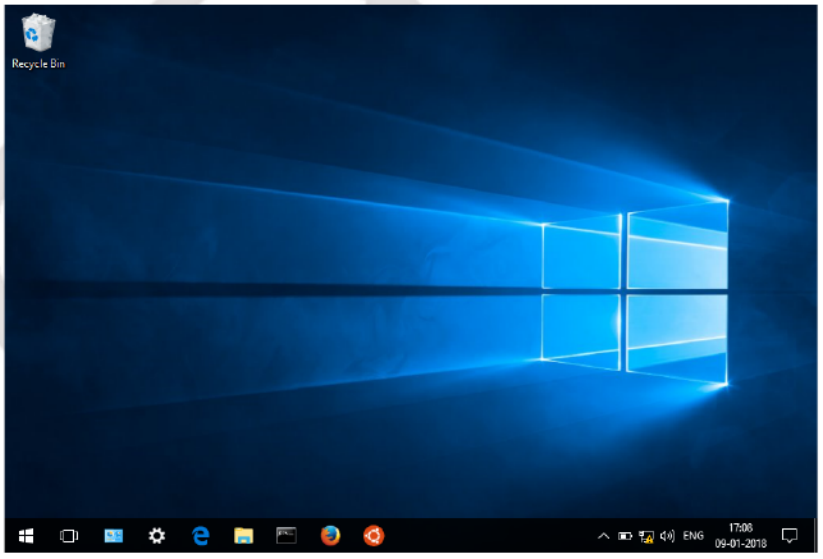
Ubuntu_16_04



Four_Components_of_a_Computer_System



The_Storage_Device_Hierarchy

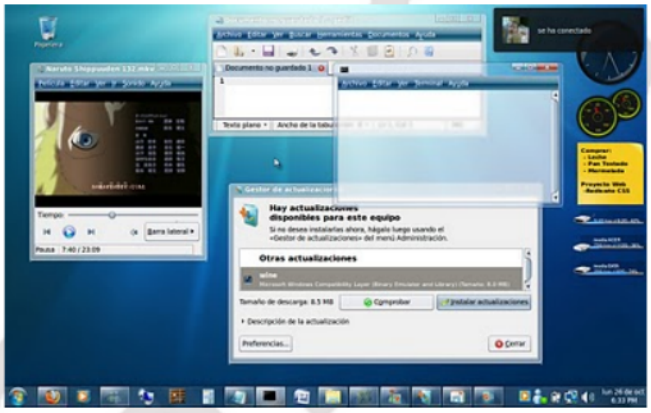


Windows_10

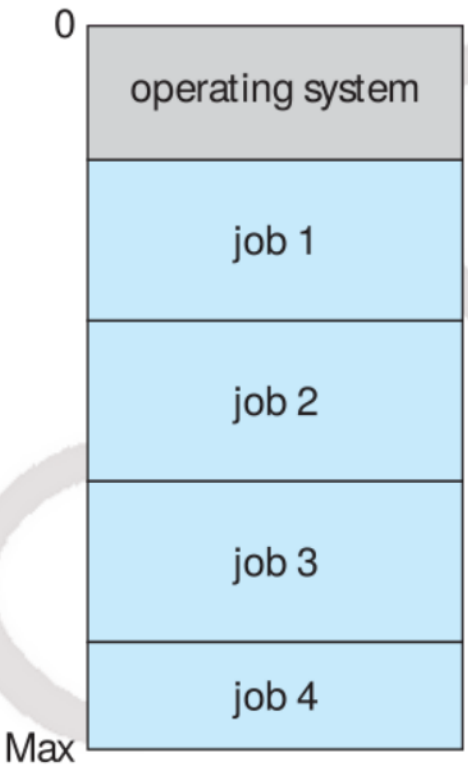
Linux_like_Windows-1

Linux_like_Windows-2

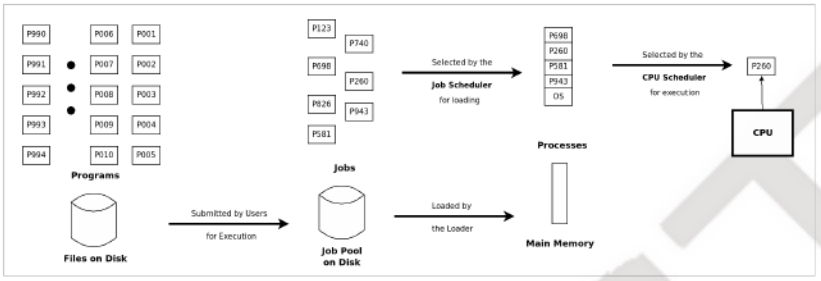
Windows_like_Linux-1



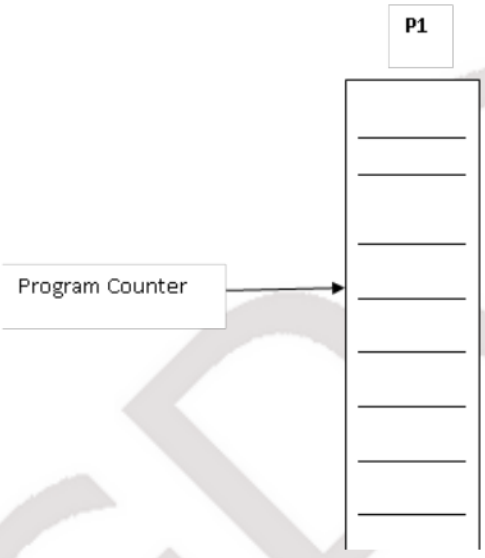
Windows_like_Linux-2



Memory_Layout_for_a_Multiprogramming_System



Programs_Jobs_and_Processes



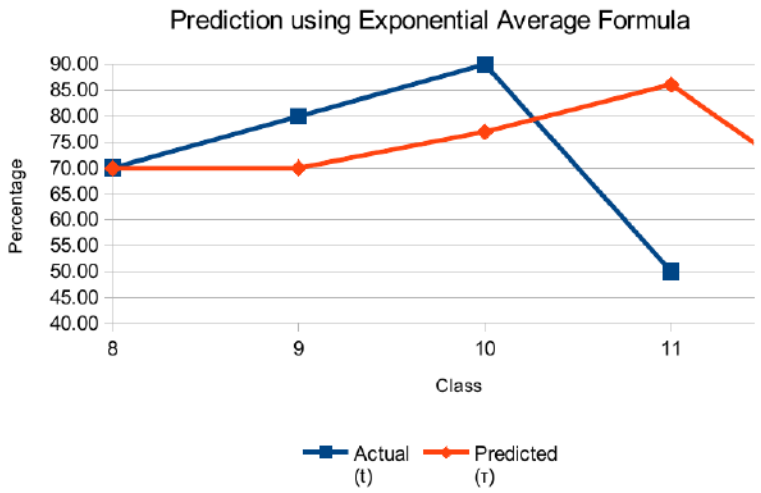
Program Counter

Exponential Average Formula: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

α	(1-α)
0.7	0.3

Class	Actual (t)	Predicted (τ)
8	70.00	70.00
9	80.00	70.00
10	90.00	77.00
11	50.00	86.10
12	?	60.83

Exponential Average Formula-1-Data



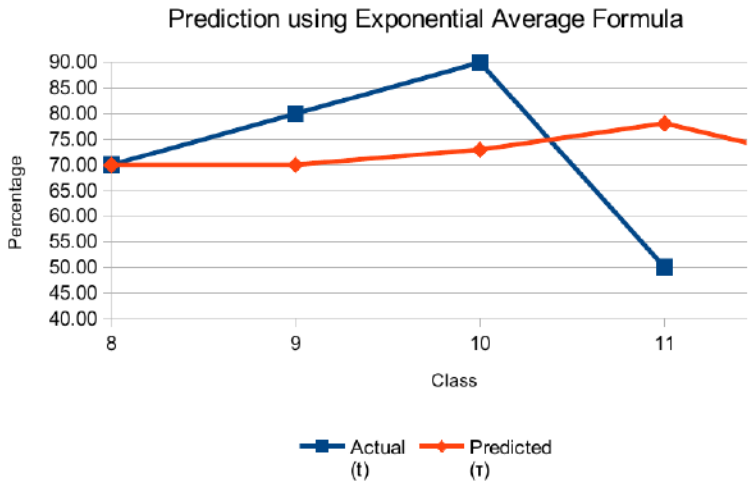
Exponential Average Formula-1-Chart

Exponential Average Formula: $\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n$

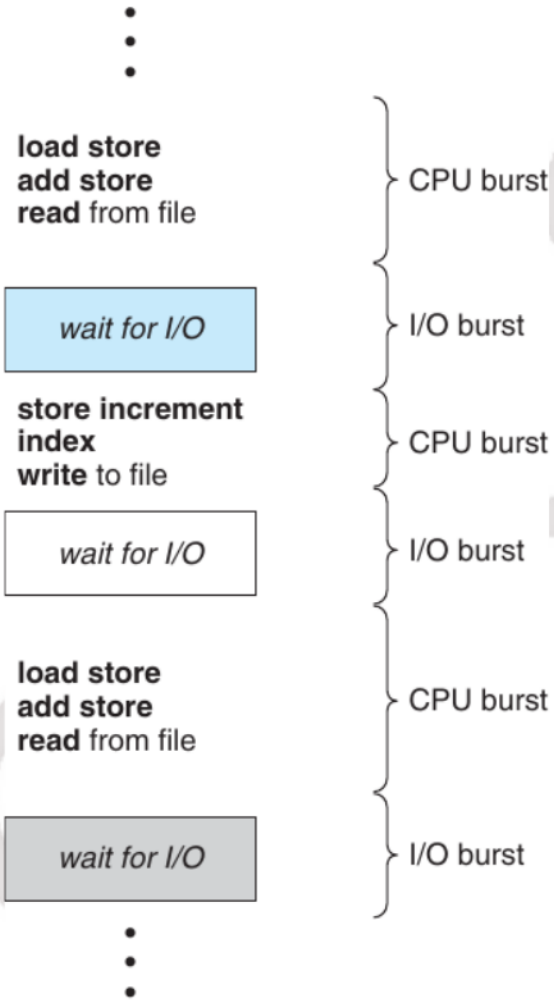
α	(1-α)
0.3	0.7

Class	Actual (t)	Predicted (τ)
8	70.00	70.00
9	80.00	70.00
10	90.00	73.00
11	50.00	78.10
12	?	69.67

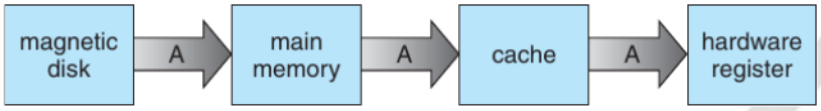
Exponential Average Formula-2-Data



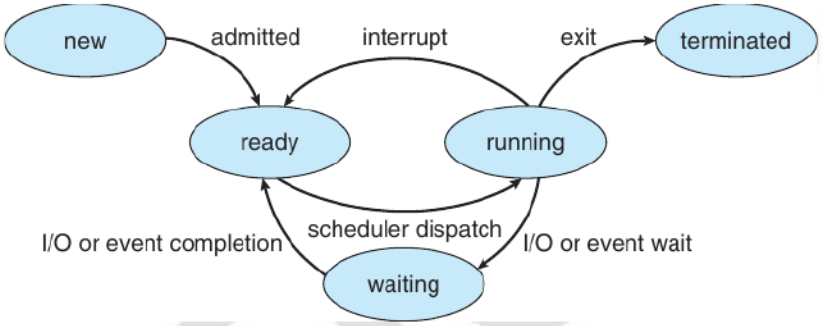
Exponential Average Formula-2-Chart



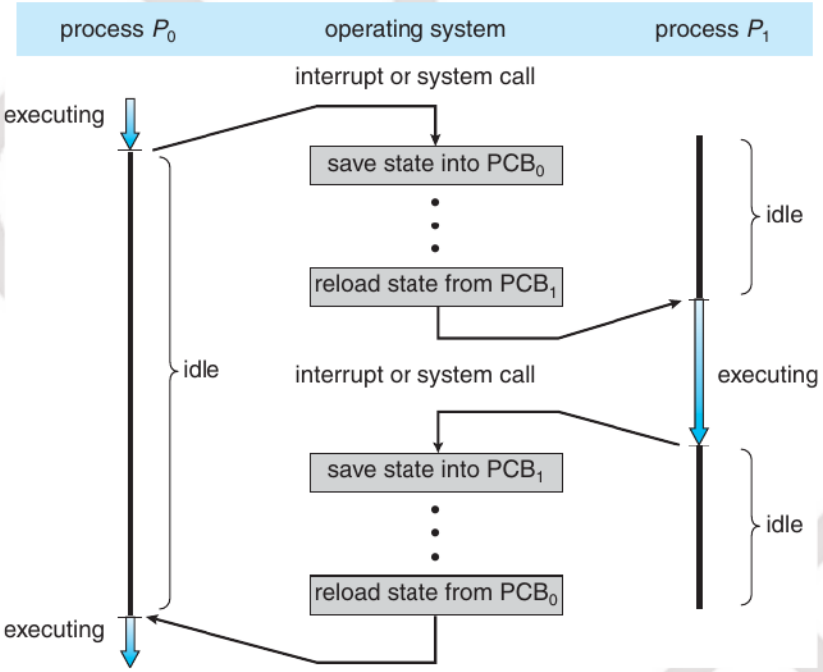
CPU Burst IO Burst Sequence



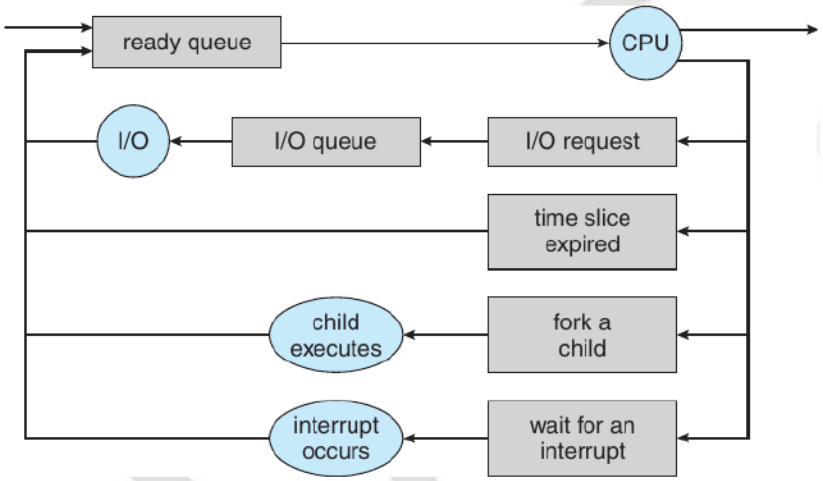
Cache Hierarchy



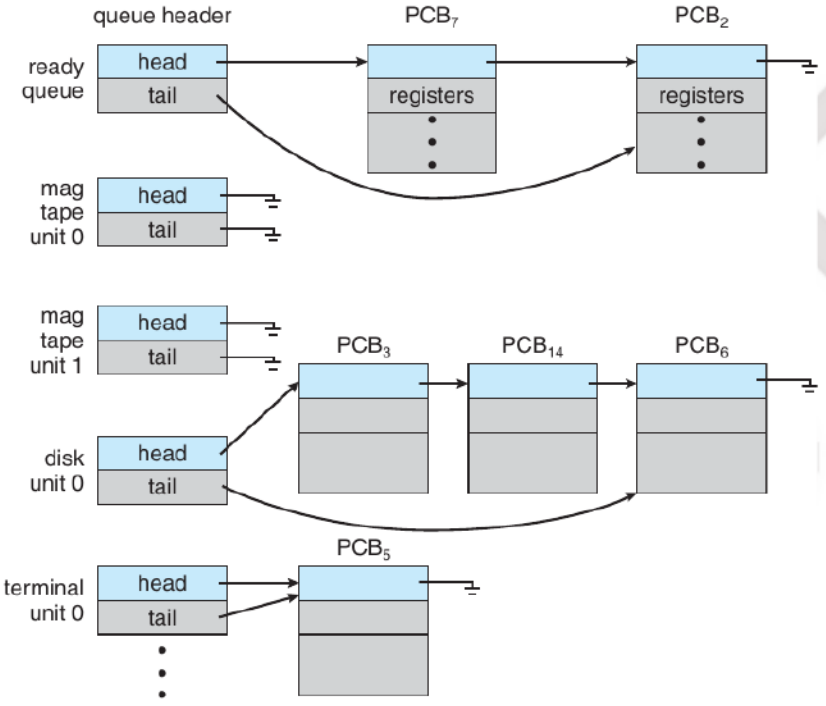
Process State Diagram



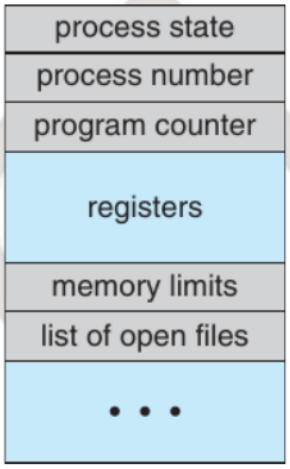
Context Switch



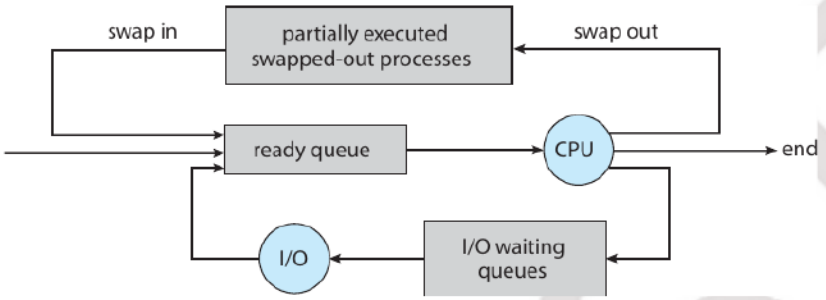
Queueing-diagram Representation of Process Scheduling



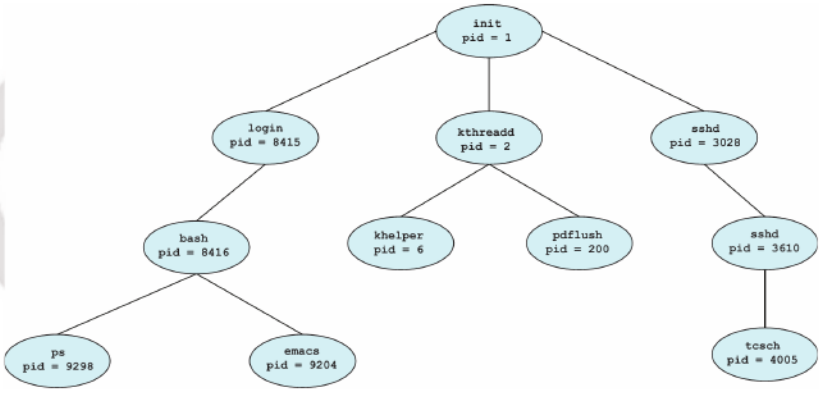
The Ready Queue and Various IO Device Queues



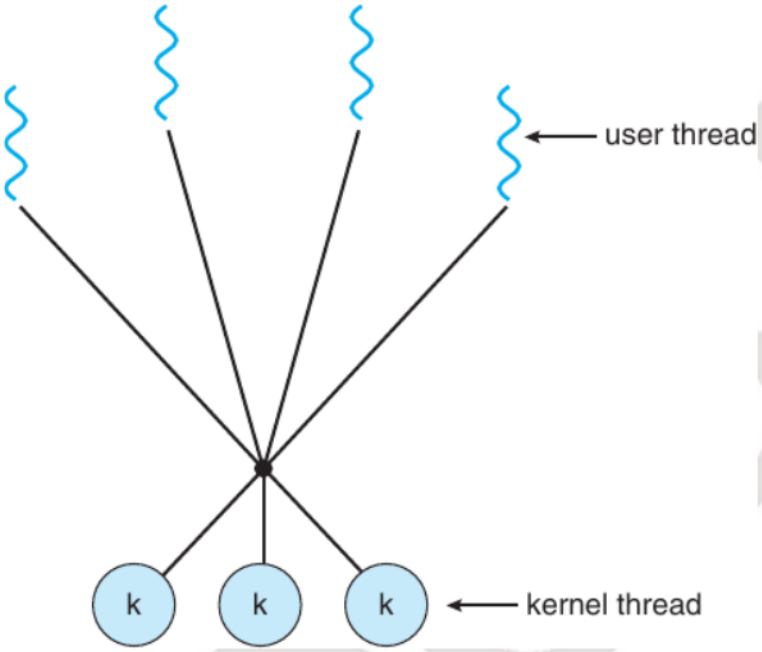
Process Control Block (PCB)



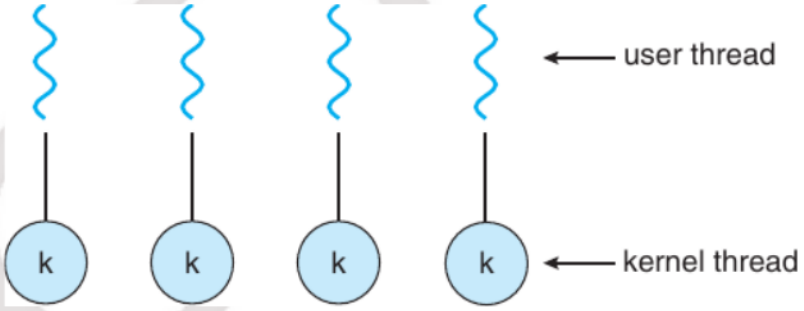
Addition of Medium-term Scheduling to the Queueing Diagram



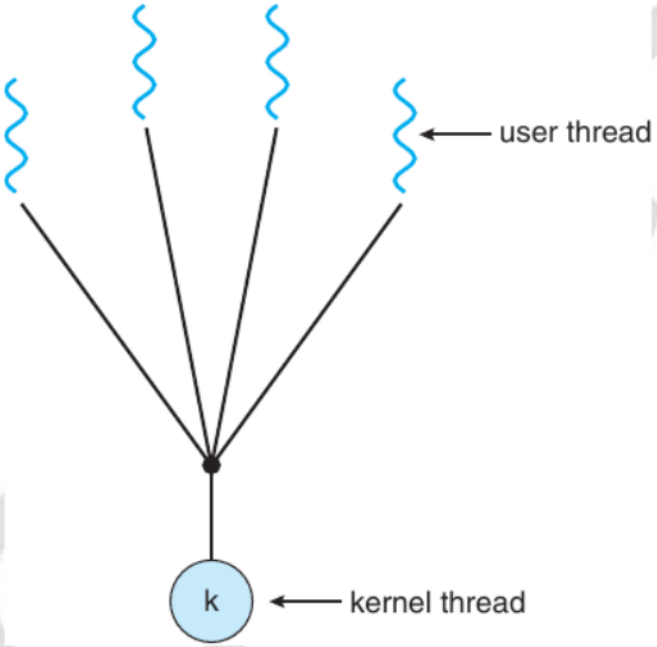
Process Hierarchy



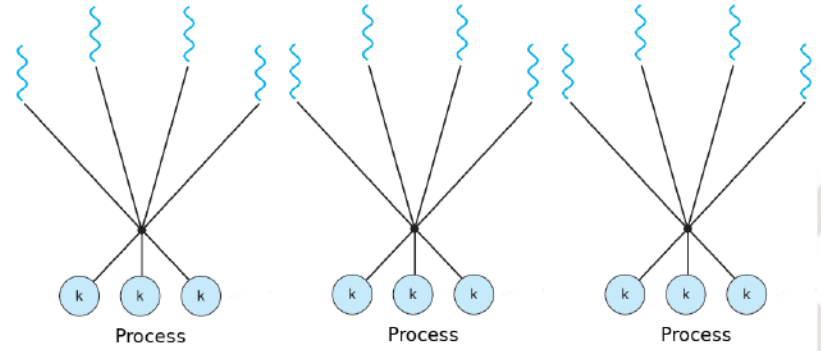
Thread Model Many-to-Many



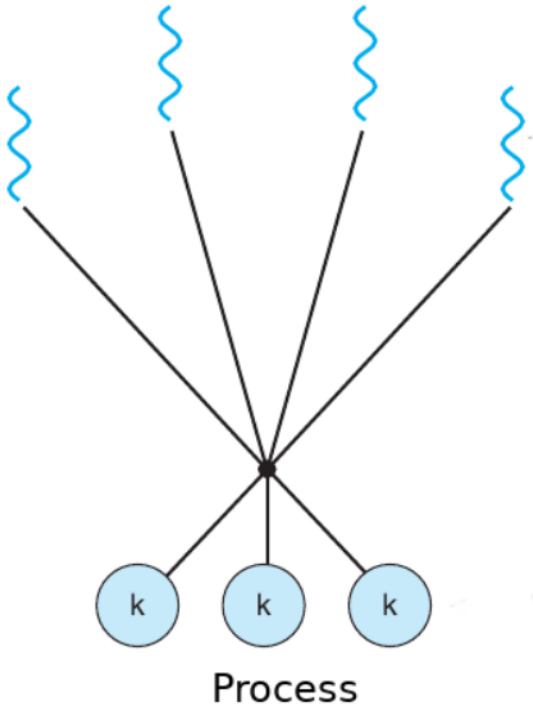
Thread Model One-to-One



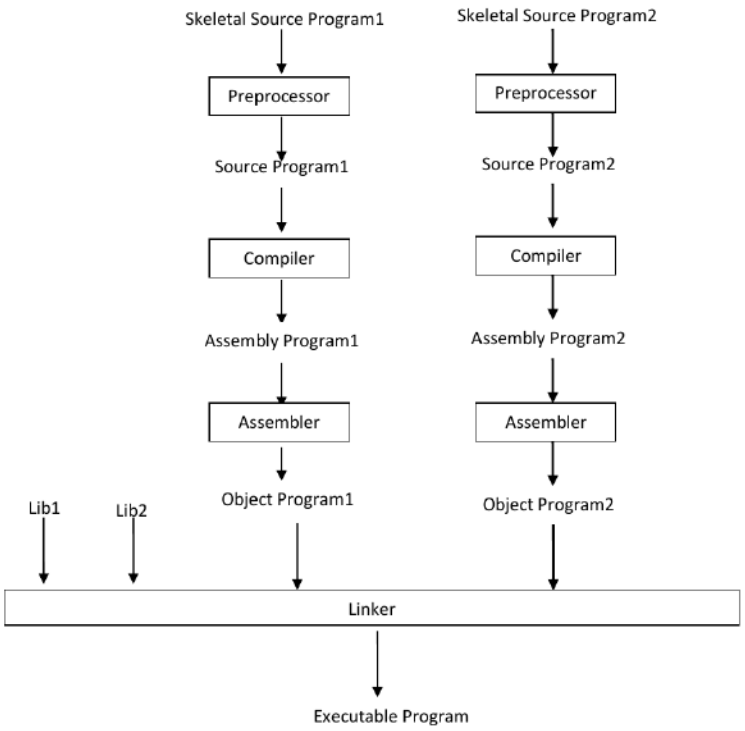
Thread Model Many-to-One



Thread Scheduling Contension Scope



Thread Scheduling Contension Scope-0



Compilation Linking Loading Concepts-1

Executable Program

Header
Object Program1
Object Program2
Actually used functions from Lib1
Actually used functions from Lib2
Symbol Table

Compilation Linking Loading Concepts-2

P1.OBJ

0.	int x=10;
1.	int y=20;
2.	mov ax, x(0)
3.	add ax, y(1)
4.	mov z, ax
5.	call fn2
6.	mov z, 50
7.	fn3 proc
8.	
9.	
10.	ret
RELOCTAB	2
	3
LINKTAB	{y, PD, 1}
	{fn3, PD, 7}
	{z, EXT, 4}
	{fn2, EXT, 5}
	{z, EXT, 6}

P2.OBJ

0.	int z=30;
1.	int a=40;
2.	mov ax, a(1)
3.	add ax, y
4.	mov z(0), ax
5.	call fn3
6.	mov z(0), 50
7.	fn2 proc
8.	
9.	
10.	Ret
RELOCTAB	2
	4
	6
LINKTAB	{z, PD, 0}
	{fn2, PD, 7}
	{y, EXT, 3}
	{fn3, EXT, 5}

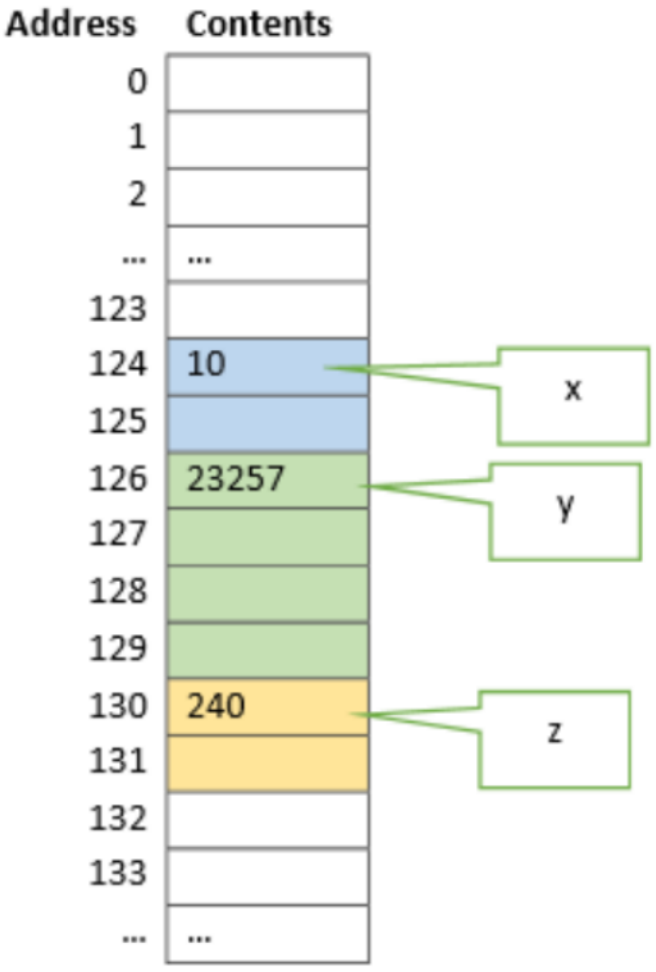
P.EXE

100.	int x=10;
101.	int y=20;
102.	mov ax, x
103.	add ax, y
104.	mov z, ax
105.	call fn2
106.	mov z, 50
107.	fn3 proc
108.	
109.	
110.	ret
111.	int z=30;
112.	int a=40;
113.	mov ax, a
114.	add ax, y
115.	mov z, ax
116.	call fn3
117.	mov z, 50
118.	fn2 proc
119.	
120.	
121.	ret

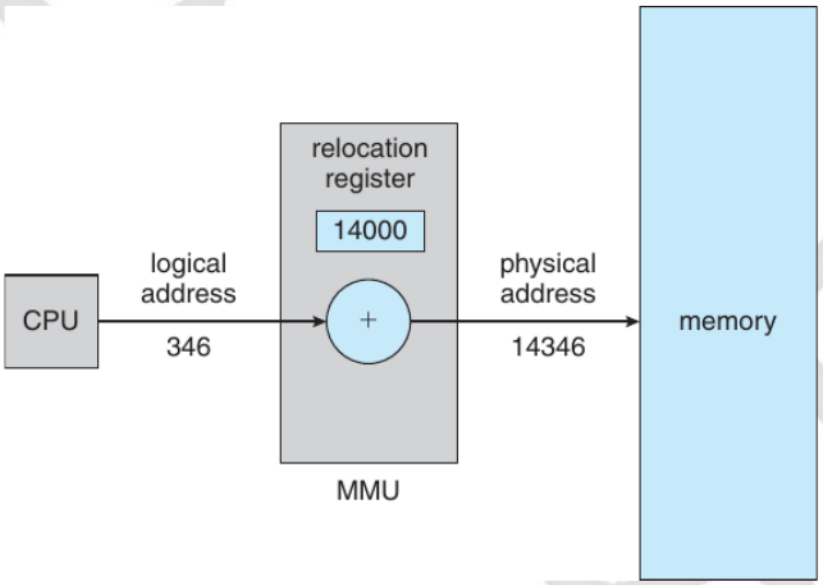
Compilation Linking Loading Concepts-4

```
_main:
100000f40:  55    pushq %rbp
100000f41:  48 89 e5    movq  %rsp, %rbp
100000f44:  48 83 ec 10    subq  $16, %rsp
100000f48:  c7 45 fc 00 00 00 00   movl  $0, -4(%rbp)
100000f4f:  c7 45 f8 00 00 00 00   movl  $0, -8(%rbp)
100000f56:  83 7d f8 0a    cmpl  $10, -8(%rbp)
100000f5a:  0f 8d 1f 00 00 00   jge   31 <_main+3F>
100000f60:  48 8d 3d 43 00 00 00   leaq  67(%rip), %rdi
100000f67:  b0 00    movb  $0, %al
100000f69:  e8 1a 00 00 00    callq 26
100000f6e:  89 45 f4    movl  %eax, -12(%rbp)
100000f71:  8b 45 f8    movl  -8(%rbp), %eax
100000f74:  83 c0 01    addl  $1, %eax
100000f77:  89 45 f8    movl  %eax, -8(%rbp)
100000f7a:  e9 d7 ff ff ff    jmp   -41 <_main+16>
100000f7f:  8b 45 fc    movl  -4(%rbp), %eax
100000f82:  48 83 c4 10    addq  $16, %rsp
100000f86:  5d    popq  %rbp
100000f87:  c3    retq
```

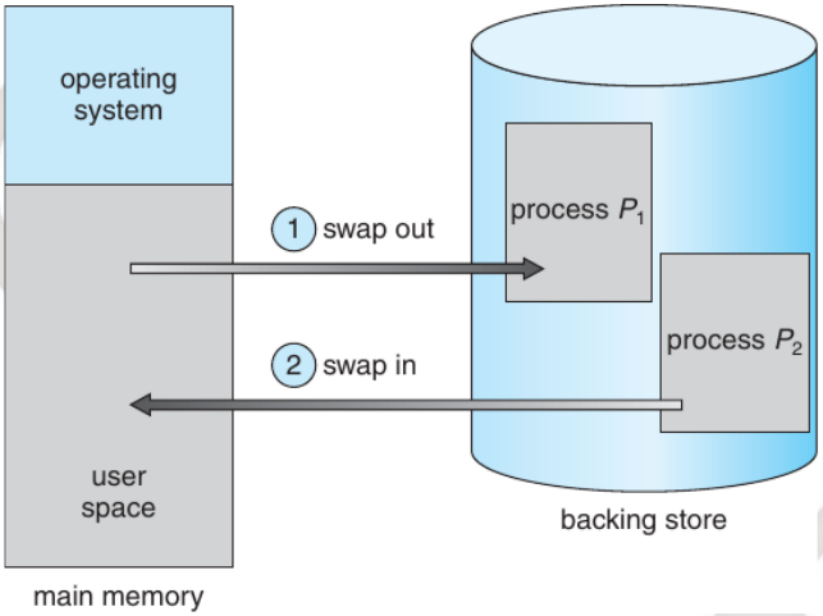
Symbols Variables and Memory Addresses-2



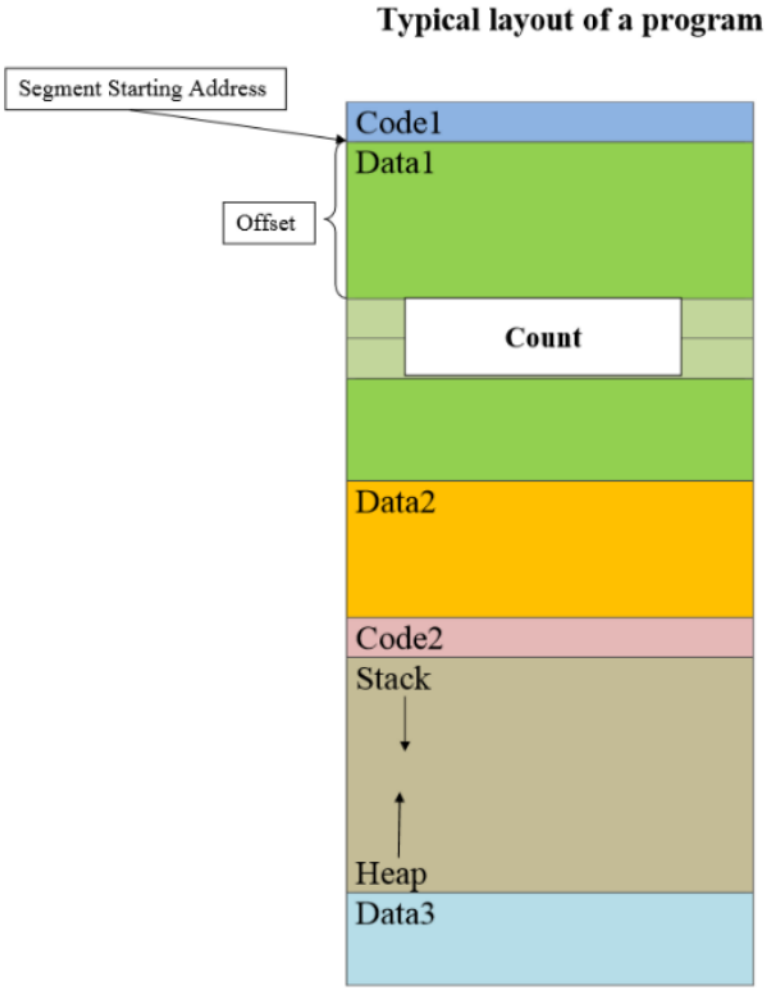
Symbols Variables and Memory Addresses-1



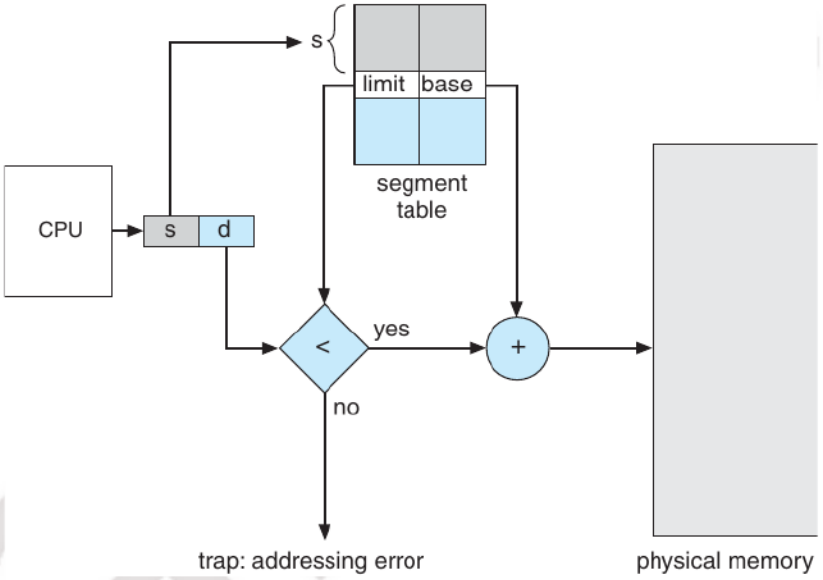
Dynamic Relocation using a Relocation Register



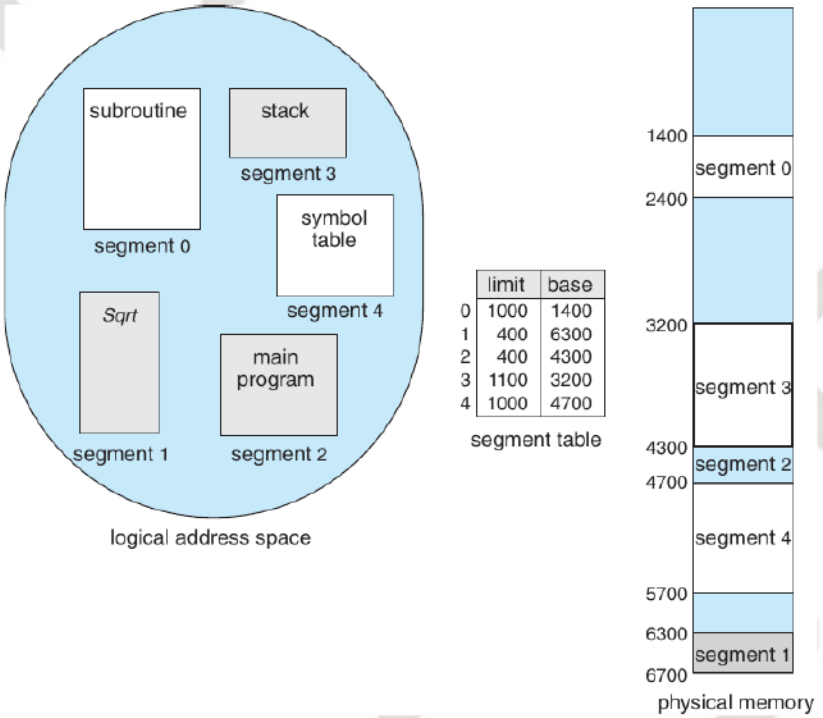
Swapping



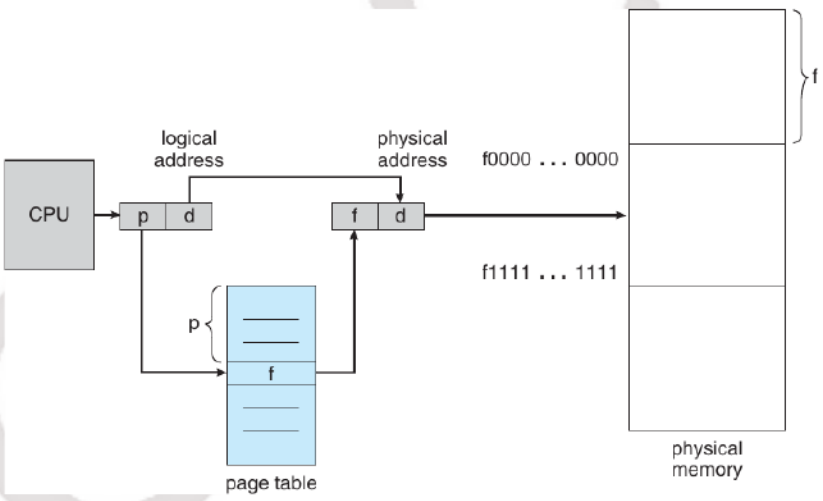
Typical Layout of a Program



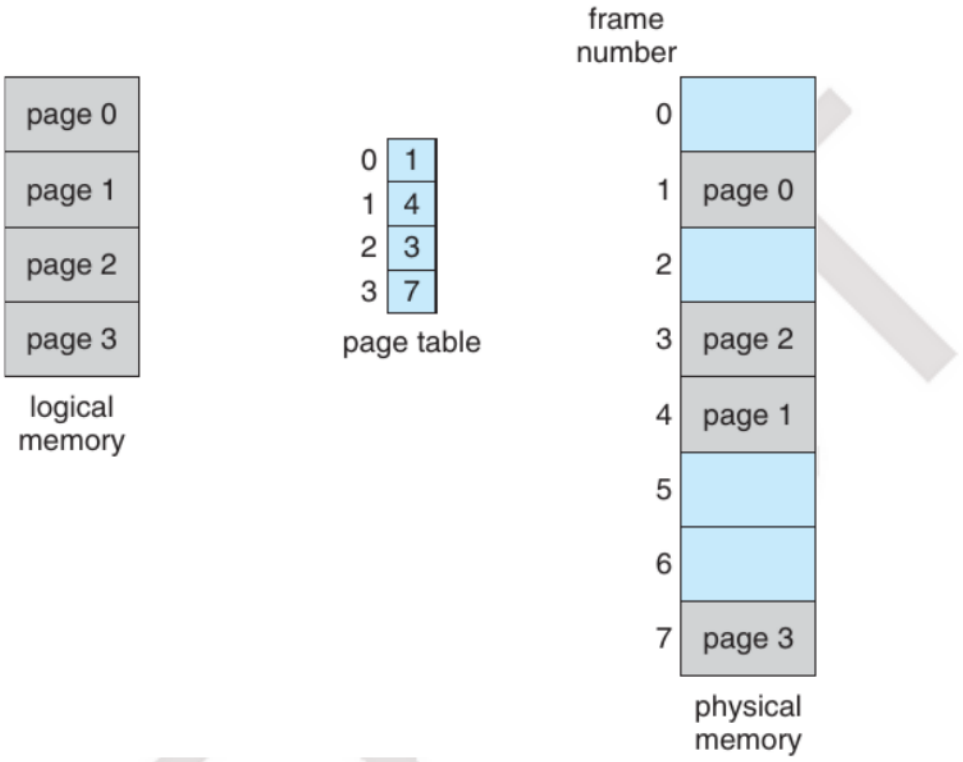
Segmentation Hardware



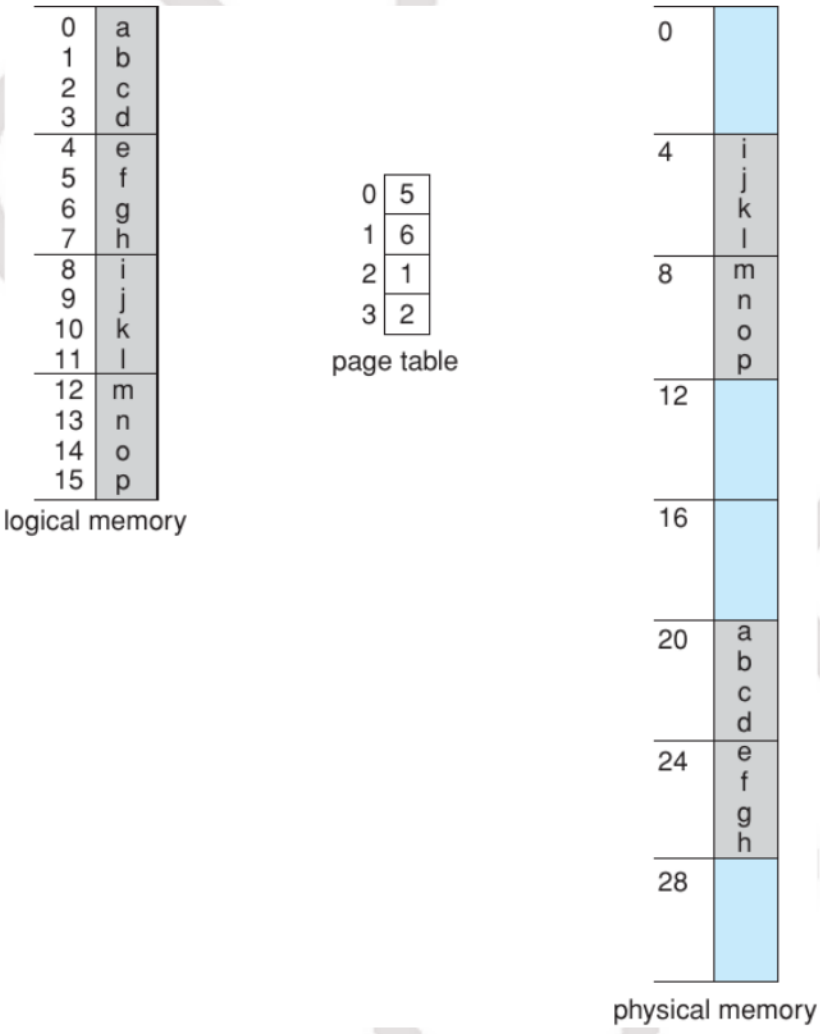
Example of Segmentation



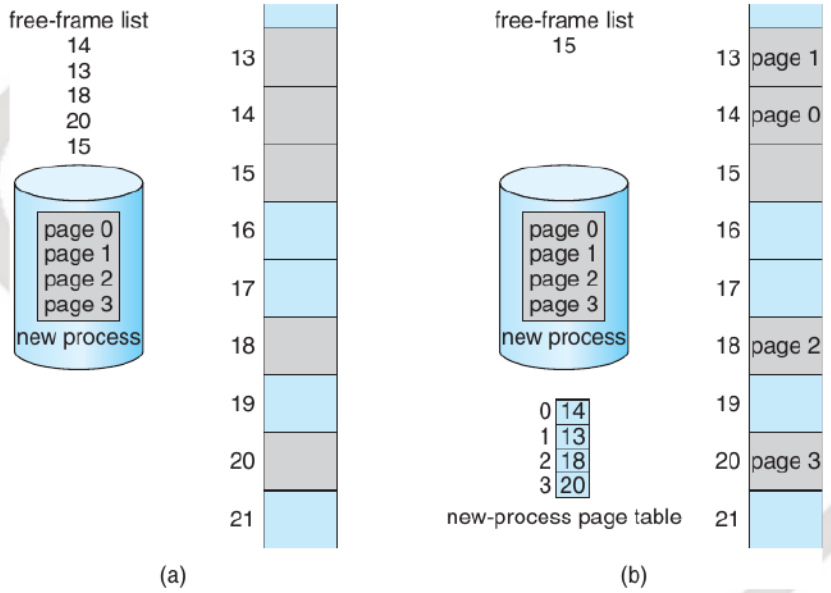
Paging Hardware



The Paging Model



Paging Example-1



Paging - Memory Allocation

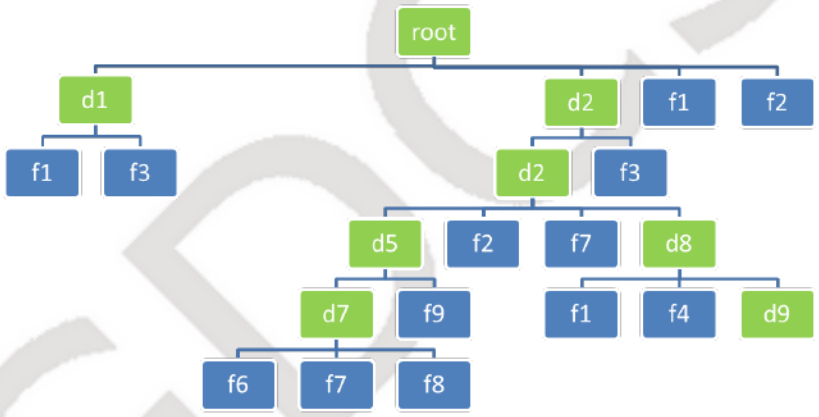
Linux Shell Programming Fundamentals

The File System

The general purpose computers also have storage facilities. The secondary storage is organized in blocks and, like everything else with computers; the blocks are accessed by block numbers. Today's hard disks may have over a billion blocks. This method, though natural for computers, is quite problematic for humans. It is impossible for a human being to remember what was stored in which block number. When some piece of information is large requiring many blocks, one must remember a series of block numbers (that need not be sequential). There is no marking on the blocks to indicate which one is in use and which one is not. One must be careful not to commit the mistake of writing some data to a block to which earlier some other useful data was written; as in that case the previously written data would be silently overwritten and lost. In case of the same system being used by multiple users, of course, there is no way for one user to know which blocks are in use by the other users.

To avoid all these problems, the operating system provides a file system interface to secondary storage. The concept of a file system is modeled after the filing cabinets commonly found in offices, but with new twists. A file system chiefly contains two types of objects – files and directories (also known as folders). A file is the basic unit of secondary data storage on computers. Any data that the user wants to store will go in some file in the file systems. The files are identified by their names, which are much easier for humans to remember than block numbers. As a disk may have a large number of files, directories are used to organize them. A directory is nothing but a container that may contain files as well other directories known as its subdirectories. The subdirectories may, in turn, contain files as well as subsubdirectories and these may contain files as well as further directories and so on. In fact, there is no theoretical limit to such nesting (putting one container object inside another). However, every file system starts with what is called its root directory and then the root directory may contain files as well as subdirectories and it can go on and on like that. The following figure illustrates this concept. Here the green nodes represent directories, while the blue ones represent files. As an added convenience, directory names

begin with d (except root) and filenames begin with f, though there is no such rule.



The file system structure

Internally, the data is still accessed by disk block numbers, but now the file system component of the operating system keeps track of which file or directory is stored in which block number(s) and which blocks are free and which ones are in use. This takes a great burden off the human users. To maintain this information, some data structures are created on the disk.

The file systems follow certain basic rules. Each file system has a single root directory that is the starting point of the file system. Each directory in the file system contains a number of objects (files and directories), each of which must have a name unique within that directory. Thus, there can never be two objects with the same name in the same directory. However, two different directories can have two different objects with the same name. As long as you know the directory containing the object you want to use, knowing its name may be enough as it is guaranteed to be unique within its directory. Otherwise, you will have to specify the absolute (or full) path leading to the object. That is, you start with the root directory, then the subdirectory, then the sub-subdirectory, and so on, until you reach to the directory that contains the object in question and finally the object itself. Each of these components is separated by a special character that is not permitted in file or directory names. This path uniquely identifies an object in the entire file system.

The file system structure is generally described as a tree structure, with the root directory forming the only root of the tree, all other directories forming branches and the files forming the leaves. If a directory is contained inside another directory, the former is said to be the child of the latter and the latter is said to be the parent of the former. Every directory, except the root directory, has a parent directory. The files cannot hold further objects (files or directories) in them, and hence are the end of the branches. They are known as leaves. With computers, tree structures are typically drawn upside down, with the root at the top.

A blank disk initially contains no file system. The initial blank file system structure (empty data structures) is created by an operation known as formatting the disk. Formatting a disk that already contains a file system destroys the existing file system and replaces it with a new blank one. Certain types of disks such as the hard disks can contain multiple partitions. Each partition may be formatted to contain a separate and independent file system. Formatting one partition does not affect the others. The operating system provides utilities for viewing, modifying and formatting the disks and their partitions.

The command line interface provides commands to navigate the directory structure, while the graphical environment provides a file browser to explore the file system. Common operations provided at the file system level with both the user interfaces include creating new files and directories, copying a file or directory to another part of the file system, moving a file or directory to another part of the file system, renaming a file or directory, deleting (removing) a file or directory (including all the contents), etc. Often security constraints may be in place to prevent a user from carrying out certain operations on certain files or directories.

Different devices and different operating systems use different file systems. For example, Microsoft Microsoft Windows family of operating systems typically use a file system called FAT (File Allocation Table) along with its variants and a file system called NTFS (New Technology File System). The first one is very simple in nature, but limited in power and performance. It also does not provide any security. The latter is far more advanced than the former and provides a host of features, including security. Both file systems are case insensitive, i.e. capital letters and small letters are treated as same. Thus, you cannot have two files with the names myletter and MyLetter in the same directory. Both generally divide the filename into two parts – the filename proper followed by a . (dot) and the extension. Both use the \ (backward slash or backslash) character as the path separator character. The extension part of the filename is used to signify the type of the file. Different Unix systems use different file systems. Linux generally uses some version of the extended file system (ext2, ext3 or ext4), though several others are also available and in use. These are quite powerful and feature rich file systems. The extended file system is case sensitive, i.e. capital and small letters are treated as two different characters. So you may have two files with the names f1 and F1 in the same directory. It uses the / (slash) character as the path separator. The concept of using the extension part of the filename to signify its type is not mandatory and is weakly used. USB flash disks and memory cards used with mobile phones usually come formatted with the FAT file system when the size is upto 32GB and the exFAT file system when the size is > 32GB (for example, the micro SDXC cards). CDs generally use the ISO9660 file system, while DVDs use the UDF file system. These different file systems have different characteristics.

The Unix/Linux File System

- Hierarchical file system
  - While the Windows file systems is actually a forest, the UNIX file systems is a proper tree (actually a directed acyclic graph)
- The root directory is identified by the / (slash) character
- Directories and files
- Identifying a file system object uniquely using path
- Components of the path are separated by the / (forward slash) character
- / (forward slash) v/s \ (backslash)
- Filenames are case sensitive
- Extensions not an essential part of file name (executable files usually have no extension)
- Any character except / can be used in file names
- The notion of current directory and relative paths
- Absolute path v/s relative path
- Interactive users have a home directory. A user has full permissions on one's home

directory. The ~ (tilde sign) is a shortcut for the user's home directory in file system manipulation commands

- Hidden files and directories - any file or directory whose name starts with a . is considered hidden. Hidden files and directories are not displayed in the nautilus file browser (GUI) or by the ls command (CUI). However, there are options to display them

Accessing Multiple File Systems

A computer system may have many storage devices in it. Also, removable devices may be inserted or attached and removed at any time. Each device has its own file system on it. A device that can have multiple partitions, like the hard disk, has a separate file system on each partition. How does one access these file systems? Operating systems like Microsoft Windows assign a separate drive letter (like C:, D:, E:, etc.) to each file system. However, Linux and other Unix-like systems have a single file system tree starting with the root directory, denoted by / (the slash character). The file system contained on the partition from which Ubuntu boots is called the root file system. The root directory of this file system becomes / - the root of the entire file system tree. Initially this is the only file system available.

We may access any other file system by mounting it on any existing directory (this directory is called the mount point). Once mounted, the contents of that file system appear as the contents of the mount point directory. If the mount point previously contained some contents (files and subdirectories), they are masked (hidden) for the duration of the mount. Now we may access (and modify) the contents of that file system from the mount point directory. When we no longer need to use the file system, we may unmount it. At this point, the original contents of the mount point directory get unmasked (become visible again). This process is depicted in the following figures a, b, c and d. Figure a shows the root file system. Figure b shows the file system on another device. Figure c shows the situation after mounting the file system of figure b onto the directory d3 of Figure a. The original contents of d3 are now masked and the contents of the file system mounted there appear as if they are the contents of d3. Figure d shows the situation after unmounting the second file system. The original contents of the directory d3 now become visible again.

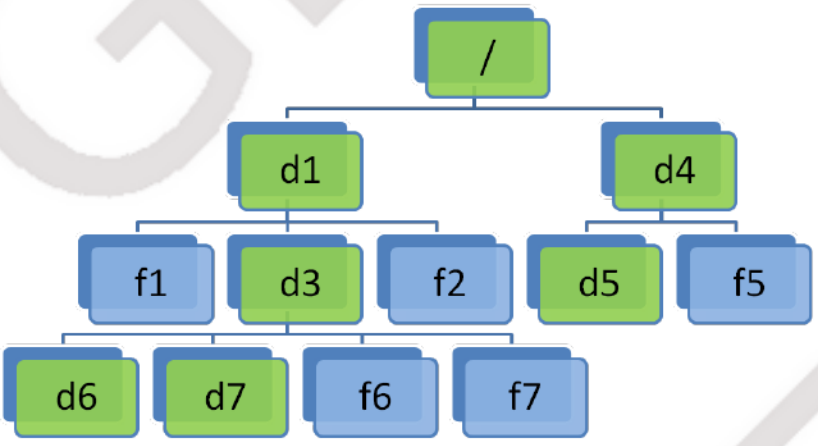


Figure a: The root file system

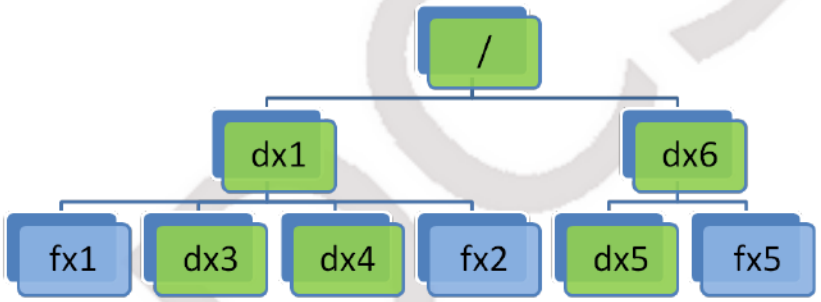


Figure b: File system on another device

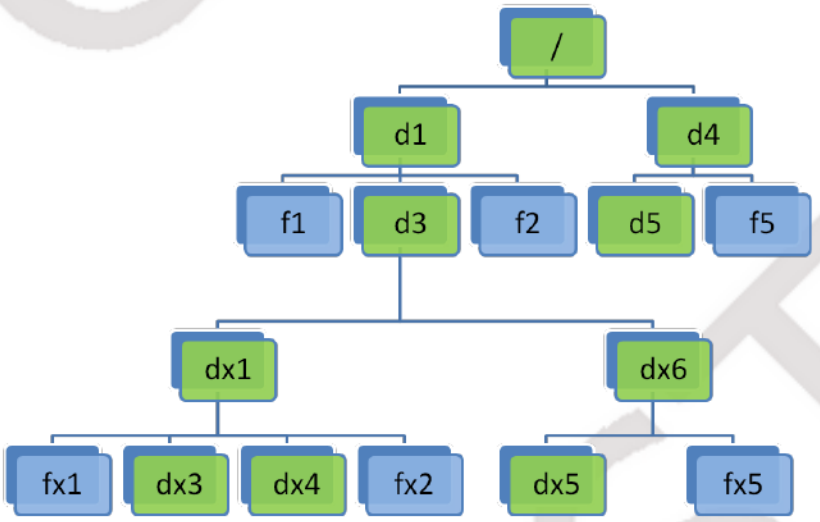


Figure c: After mounting the second file system on directory d3

Figure d: After unmounting the second file system from directory d3

Figure d: After unmounting the second file system from directory d3

However, the common practice is to mount file systems onto empty directories. In the default configuration, Ubuntu automatically detects other fixed devices in the system and shows them in the places menu and the left pane of the file browser. They are mounted when we first try to access them. The file browser shows a triangular icon alongside all mounted file systems other than the root file system. The file system can be unmounted by clicking on this icon. It may be mounted again when the user tries to access it again next time. The root file system cannot be unmounted. Removable devices are automatically mounted when they are inserted. Read-only media like optical disks can be unmounted by simply removing or ejecting them. Media on which writing is possible (like USB flash disks) must be unmounted by clicking the unmount icon in the file browser or by right clicking its icon on the desktop and selecting “safely remove device” option. This causes any data cached in main memory for improved performance to be flushed to the disk. A message at the end of this process announces that it is now safe to remove the device and the unmount icon disappears from besides the device’s entry in the file browser’s left pane. Only after this the device can be safely detached from the system. Failure to observe this

procedure may result in loss of data or damage to the file system on the removable media. While this method of accessing the other storage devices in the system may sound unnecessarily complicated, it is more flexible and powerful and has several distinct advantages.

Not only local file systems, even the remote file systems can also be accessed in the same way. For example, Windows shares can be mounted on local directories using the SMB (Server Message Block) protocol and then be accessed just like local content. Similarly, NFS (Network File System), FTP server directories can also be mounted on a directory and accessed as a part of the file system.

Of course, all these procedures can also be carried out using commands. It is also possible to configure the system to mount some file systems at particular mount points automatically every time the system boots. By default, the system mounts other fixed and removable devices in the system in directories under the /media directory.

The Shells

Unix offers a variety of shells in both text mode and graphical mode. The graphical mode shells include the GNOME shell, Unity, KDE, XFCE, LXDE, MATE, etc. The commonly used text mode shells include sh (the original Bourne shell), ksh (Korn shell) csh (C Shell), bash (Bourne-Again SHell), etc.

The Bourne Shell

- **Internal commands** are built into the shell itself. There is no separate executable file for them
- **External commands** are not built into the shell. There is a separate executable file for them
- Search path
    - Since it is not practical to search the entire hard disk for the executable file corresponding to a possible external command entered by the user, only the directories in the search path are searched for the same. The search path is stored in the built-in shell variable PATH as string containing one or more directories separated by the : (colon) character
    - The current directory is not searched for a command if it is not in the search path
- CTRL-D at the beginning of a line indicates end-of-file (or end-of-input)
- CTRL-C is used to terminate the currently running foreground process
- CTRL-S is used to pause the terminal
- CTRL-Q is used to resume the terminal
- To copy text from the terminal, select the text using the mouse and press CTRL-SHIFT-C
- To paste text into the terminal, press CTRL-SHIFT-V. The text is inserted as if it had been typed by the user
- The shell maintains the history of the commands entered earlier in memory. When the shell session terminates, this command history is saved in the file ~/.bash_history. The command history can be accessed using the UP / DOWN arrow keys at the prompt. It is also available in subsequent sessions and survives reboots
- The shell has its own full-fledged built-in programming language with variables, control

structures, input, output, etc. It is a dynamic language in nature
- Case sensitivity
- Command completion using TAB and TAB-TAB

The Command Line

- **Components of the command line and word splitting** The command line contains one (or more) command(s), arguments to the command, operators, etc. The shell performs word splitting on the command line using white space (space / tab characters) as the separators
- In general, order of components of a command line is not important. However, there are several exceptions
- **Options**
  - **Short options** Short options consist of - (hyphen) followed by a single character
  - **Long options** Long options are specified using -- (two consecutive hyphens)
  - **Options with arguments** Some options have their own arguments. In such cases, the argument(s) must immediately follow the option
  - **Combining short options** Multiple short options can be combined by writing them immediately after a single hyphen: `ls -lh` . Only the last short option can have argument(s) in such case; and the argument, if any, is immediately specified after the combined options: `tar -czf backup.tar.gz directory`
- **Quoting and escaping** Strings on the command line can be specified without using any quotes. However, if a string contains white space or some other character having a special meaning to the shell interpreter, the string must be enclosed in quotes or the white spaces / special characters must be escaped
  - **Escaping** A character having special meaning to the shell can be escaped by preceding it with \ (backspace). This removes its special meaning. The special meaning of the backslash itself can also be removed by preceding it with another backslash `\\`
  - **Single Quotes** Almost no processing is carried out inside strings enclosed in single quotes `'aaa bbb ccc'`
  - **Double Quotes** Double quotes also behave like single quotes, but variable / parameter expansion *is* carried out inside double quotes: `$variable_name` is converted into the value of the variable. Hence if the value of variable *name* is *Scotty*, `echo 'Beam me up, $name'` outputs `Beam me up, $name` but `echo "Beam me up, $name"` outputs `Beam me up, Scotty`
- **Operators** (Input/Output Redirection, Combining Multiple Commands using Logical Operators)

File System Manipulation Commands

Note: *File system object* means either a file or a directory

- **pwd** (present working directory)
  - **pwd** Outputs the current working directory
- **cd** (change directory)
  - **cd directory** Changes the current directory to the given directory
  - **cd** changes the current directory to the home directory of the user. This behaviour

is different from Windows where `cd` displays the current directory

- **ls** (list)
  - **ls** Outputs the names of the file system objects in the current directory
  - **ls arguments** Outputs the information about the argument(s). In case an argument is a file, its name is displayed. In case an argument is a directory, the names of file system objects inside the directory are displayed. If the argument file system object does not exist, an error message is output
  - **Options**
    - **-l** (*long* listing) Outputs a long listing with more information about the file system objects
    - **-d** (*directory*) When an argument is a directory, outputs information about the directory itself rather than its contents
    - **-R** (*recursive*) Outputs information about each argument along with child file system objects recursively
- **mkdir** (make directory)
  - **mkdir directory** Creates the given directory if it does not exist
- **rmdir** (remove directory)
  - Removes (deletes) the given directory if it is empty
- **cat** (concatenate)
  - **cat file(s)** Outputs a concatenation of the given files. Also used to output a single file
- **cp** (copy)
  - **cp arguments** There must be at least two arguments. The last argument is the target. All penultimate arguments are sources. If the target is a directory, the sources are copied into the target. If the target is a file, there must be only one source and the source file is copied onto the target file. If the target does not exist, it is assumed to be a file and is created. *cp* does not copy directories by default. There are no warning or prompts if a file is going to be overwritten
  - **Options**
    - **-r** (recursive) This option is used to copy an entire file system tree rooted at the given node (i.e. a directory along with all its contents recursively)
    - **-i** (interactive) Prompts before overwriting a file
- **rm** (remove)
  - **rm arguments** Removes (deletes) the given arguments. While the files and directories deleted from the GUI go into the trash, files and directories deleted from the command prompt are permanently lost. Does not delete directories by default. Does not prompt before deleting a file
  - **Options**
    - **-r** (recusrive) Deletes the arguments recursively. Directories in the arguments are deleted with all the contents recursively
    - **-i** (interactive) Prompts before deleting a file
- **mv** (move)
  - **mv arguments** Moves or renames as per the number and types of arguments
  - **How is the target of the move is determined**
    - If the destination is the name of an existing file, the target of move is assumed to be that file
    - If the destination is the name of an existing directory, the target is assumed

to an object with the same name as the source inside that directory
• If the destination does not exist, the destination name itself is the target. It is considered a file name if the source is a file and a directory name if the source is a directory

- **When a move operation is performed**
  • If the directory containing the source and the directory containing the target are different, the source object is moved
- **When a rename operation is performed**
  • If the source name and target name are not same, a rename operation is performed
- **When a file is overwritten**
  • If the target is a file and it already exists, it is overwritten
- **Operations that are not allowed**
  • If there are more than one sources and the destination is not a directory
  • If the source is a directory and the target is an existing file
  • If the source is a filename and the target refers to the same file
- **Options**
  • **-i** (interactive) Prompts before overwriting a file

Shell Globbing Patterns

Pattern	Matches with
?	Any single character
*	Any number of any characters
[abcde]	Any single character from among a, b, c, d and e
[^abcde]	Any one character other than a, b, c, d and e
[aeiou]	Any single vowel
[^aeiou]	Any one character other than a vowel
[abc^de]	Any one character from among a, b, c, ^, d and e
[a-z]	Any single lower case alphabetic character
[a-zA-Z]	Any single lower or upper case alphabetic character
[a-zA-Z_%]	Any single character from among alphabetic characters, _ and %
[a-zA-Z0-9]	Any single alphanumeric character
[^a-zA-Z0-9]	Any single non-alphanumeric character

Examples

Pattern	Matches with
p*	Anything starting with p
p*e	Anything that starts with p and ends with e, including pe
*~	All backup files

Pattern	Matches with
b??h	b, followed by exactly two characters, followed by h
b*h	Anything that starts with b and ends with h, including bh
dir*	Anything that starts with dir, including dir
dir?	dir followed by exactly one character

Plain Text Editors

- Plain text editors
  - Graphical editors
    - gedit / Text Editor (default)
    - gvim
    - leafpad
  - Text mode editors
    - vi (default)
    - vim (VI iMproved)
    - nano, pico
    - emacs (Editing MACroS)

Brief Overview of Working with the Vim Editor

- vim stands for VI iMproved
- It is an improved version of the vi editor
- vim has several *modes*
  - *Normal* (*command*) mode
    - When vim starts, it is in this mode
    - In this mode, the keys pressed by the user are interpreted as commands and corresponding actions are taken
  - *Insertion* mode
    - This mode is entered by pressing the *i* command in the command mode. Several other commands also switch to insertion mode
    - in this mode, the keys pressed by the user are treated as characters to be inserted into the text being edited at the curent cursor position
    - One may switch back to the command mode by pressing `ESC` in the insertion mode
  - *Last line* mode
    - This mode is entered when using certain commands in the command mode, like `:`
    - It is used for *ex commands* and long commands
    - The cursor moves to the last line where the rest of the command is entered
    - The command is completed (and executed) by pressing `ENTER` or `ESC`
    - To cancel the command, press `CTRL+C`
- Cursor movement keys like the arrow keys, HOME, END, PgUp, PgDn, etc. work as expected in both the command mode and the insertion mode in vim
- When vim is installed, the command *vi* also invokes *vim* only

- A file is opened by passing its name on the command line

```
vi filename
```

- A file is saved by typing `:w` (write) in the command mode
- To save the current file and close vi, type `:wq` (write and quit) in the command mode
- To close vi without saving the current file, type `:q!` (quit, with force) in the command mode
- An entire line can be deleted by pressing `dd` in the command mode
- *n* lines can be deleted by pressing `ndd` in the command mode. Here *n* is an integer and is known as the repeat count
- Deleting line(s) actually *cuts* them, so they can be pasted immediately afterwards
- `yy` (yank = copy) copies a line of text
- *n* lines can be copied by pressing `nyy` in the command mode.
- Text that is copied or cut (deleted) can be *put* (pasted) after the cursor using the command `p` (put) and before the cursor using the command `P`
- `u` (undo) can be used to undo the operations, while `CTRL-R` (redo) can be used to redo the operations that were previously undone

The Shell Scripting Language of the *bash* Shell

Shell Variables

- The shell variables need not be declared. A variable is created automatically the first time it is assigned a value. All variables are of type *string*; but, in some contexts, may be interpreted as numbers. The variable assignment takes the form

```
variable_name=value
```

There must not be spaces around the = (assignment operator). Also, there is no `$` in front of the variable name when assigning value to it.

- The value of a variable can be accessed in the following ways:

```
$variable_name
${variable_name}
```

- It is ***not*** an error to attempt to access the value of an undefined variable. Such an attempt simply returns an empty string (a string of zero length)
- **Tip** Whenever you have a doubt that a variable may not exist, or that its value may be empty string or that its value may contain spaces in it, access its value using `"` (double quotes). For example, use `test -n "$v1"` instead of `test -n $v1`. You may also choose to always use double quotes
- **Built-in variables of the shell** The shell has a number of built-in variables. By convention, their names are in ALL_CAPS. For example, PATH, HOME, SHELL, TERM, etc.
    - **SHELL** : The current shell

    - **HOME** : Home directory of the currently logged in user
    - **PATH** : A : (clolon) separated list of directories that are searched for external commands
    - **TERM** : The type of the current terminal
    - **PS1** : The primary prompt
    - **PS2** : The secondary prompt
- **Special parameters** The shell has several special parameters
    - **# ($#)**
    - *** and @ ($*, "$*", $@, "$@")**
    - **$?**
- **Debugging the shell script**
    - Use `set -xv` to turn debugging on
    - Use `set +xv` to turn debugging off

The `test` Command and the `[` Builtin

- The test command
    - The command `test` tests some condition and returns exit status accordingly (0 means success / true and non-zero means failure / false). It does not produce any output
    - The spacial parameter `?` can be used to output the exit status of the last foreground process
    - The syntax `[ condition ]` can be used in place of test condition (`[` is a shell builtin)
    - There **must be** at least one space between each and every argument and `[` , `]`

```
echo $?
```

- String tests
    - **-z *string*** Returns true if *string* has zero length (empty string)
    - **-n *string*** Returns true if *string* has non-zero length
    - ***string1* = *string2*** Returns true if *string1* is equal to *string2*
    - ***string1* != *string2*** Returns true if *string1* is not equal to *string2*
- Integer tests
    - ***no1* -gt *no2*** Returns true if *no1* is greater than *no2*
    - ***no1* -ge *no2*** Returns true if *no1* is greater than or equal to *no2*
    - ***no1* -lt *no2*** Returns true if *no1* is less than *no2*
    - ***no1* -le *no2*** Returns true if *no1* is less than or equal to *no2*
    - ***no1* -eq *no2*** Returns true if *no1* is equal to *no2*
    - ***no1* -ne *no2*** Returns true if *no1* is not equal to *no2*
- Logical operations on tests
    - ***test1* -a *test2*** (and) Returns true if both the tests evaluate to true
    - ***test1* -o *test2*** (or) Returns true if any one of the tests evaluates to true
    - **!*condition*** (not) Negation / logical inversion of condition

The Control Structures

- The *if* statement

```
if command1
then
    statement1
    statement2
    ...
elif command2
then
    statement3
    statement4
    ...
elif command3
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if test condition1
then
    statement1
    statement2
    ...
elif test condition2
then
    statement3
    statement4
    ...
elif test condition3
then
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

```
if [ condition1 ]
then
    statement1
    statement2
    ...
elif [ condition2 ]
then
    statement3
    statement4
    ...
elif [ condition3 ]
then
```

```
    statement5
    statement6
    ...
 ...
 ...
else
    statement7
    statement8
    ...
fi
```

- The *while* statement

```
while command
do
    statement(s)
done
```

```
while test condition
do
    statement(s)
done
```

```
while [ condition ]
do
    statement(s)
done
```

- The *for* statement

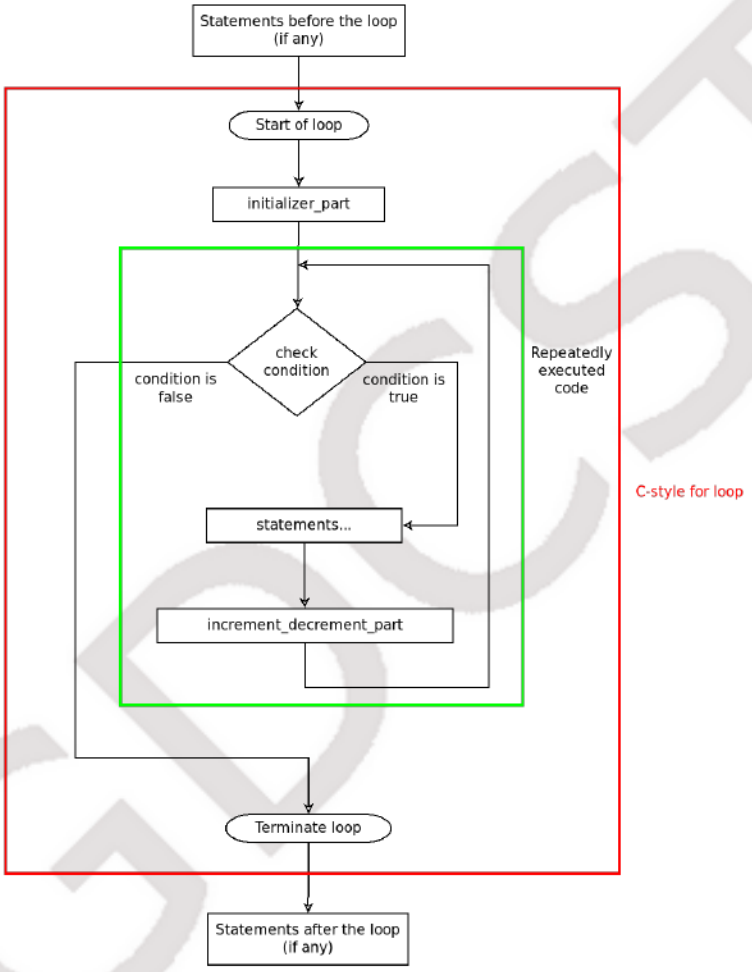
```
for variable in white-space-separated-list
do
    statement(s)
done
```

- The *C-style for* statement

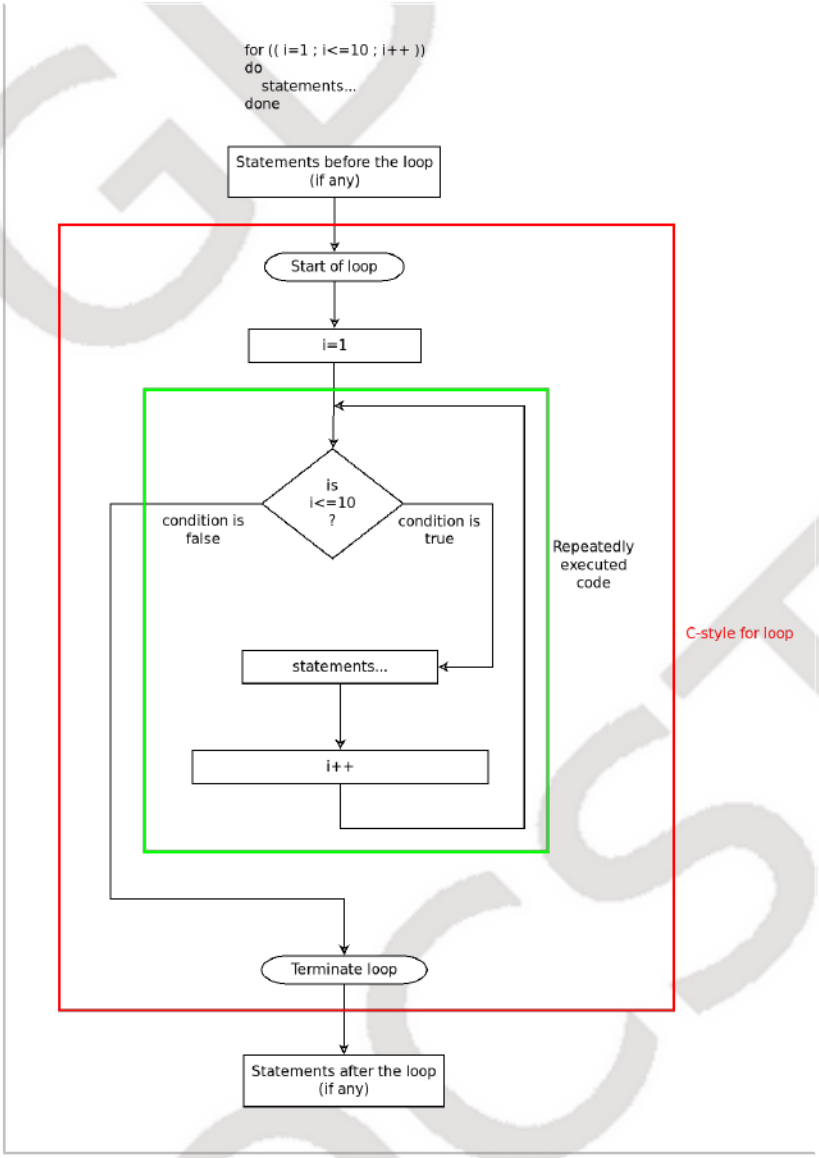
```
for ((initializer; test; increment))
do
    statement(s)
done
```

```
for ((i=0;i<10;i++))
do
    echo $i
done
```

```
for (( initializer_part ; condition ; increment_decrement_part ))
do
   statements...
done
```



The C-style for loop



The C-style for loop example

- New line in syntax
  - A `;` (semicolon) can be used wherever the syntax requires a new line
- break and continue
  - `break` can be used in the loops to terminate the loop and jump to the next statement after the loop
  - `continue` can be used to terminate the current iteration of the loop and jump to the start of the loop
- The command `true` does nothing, but returns with an exit status of zero (success / true)
- The command `false` does nothing, but returns with a non-zero exit status (failure / false)

```
i=0
while true
do
    echo $i
    let i++
    if [ $i -ge 10 ]
    then
        break
    fi
```

```
done
```

- The *case* statement
  - cases are matched in sequence, and only the first matching case is executed
  - All cases except the last one must end with ;;
  - Multiple cases may be combined using | (the vertical bar character)
  - Shell globbing patterns may be used in cases
  - The pattern `*` stands for the default case. it must be the last case
  - The case statement ends with `esac`, reverse of *case*

```
read -p "Enter your option: " option
case $option in
1|4) echo one or four
     echo "(any one of them)"
   ;;
2) echo two
   ;;
3) echo three
   ;;
??) echo two characters
   ;;
a*) echo begins with an a
    ;;
*) echo other
esac
```

Comments

- The **#** character is used for writing single-line comments
- All the characters starting from the # upto the end of the line are treated as comment and are ignored by the shell interpreter
- There is no syntax for multi-line comments

The *Shebang* Line

- Unix-like systems offer several shells as well as interpreters for many different programming languages. The scripts can be executed as stand-alone commands (without mentioning the shell or the interpreter) as long as the shell or interpreter is present in the path. When a script is executed as a command, the shell tries to guess the interpreter to be used for its execution based on the contents, because the concept of file extensions is not strictly enforced in unix-like systems. Sometimes, such guess may be wrong also. To inform the shell which interpreter to use for executing a script in an explicit way within the shell script itself, a *shebang* line is used
- The *shebang* line is a special comment line mentioning the interpreter to be used for executing the script

```
#!/bin/bash
#!/bin/csh
#!/bin/ksh
```

```
#!/usr/bin/perl
```

```
#!/usr/bin/python
#!/usr/bin/env python
```

- The *shebang* line must be the first line in a script

Output

- *echo*
  - `echo` is used to output its arguments
  - By default, echo appends a new line at the end of its output
  - To suppress that new line, use the `-n` option:
- *printf*
  - `printf` is used for formatted output
  - It syntax is identical to the printf function in the standard C library, but it is a command
  - It does not append a new line at the end of its output by default

```
printf "%03d %20s %5.2f\n" $no $name $marks
```

Input

- *read*
  - The `read` shell builtin can be used to read the values of one or more variables
  - `read x` reads a line from the input and assigns it to the variable x
  - `read x y z` reads a line from the input, separates the words in the line and assigns them in order to the variables
  - If there are less words than variables, the remaining variables are not assigned
  - If there are more words than variables; all but the last variable are assigned one word while the last variable is assigned the rest of the line
  - The `-p` option can be used to display a prompt for reading

```
read -p "Enter a number: " no
```

Integer Arithmetic

- *expr*
  - The `expr` command can be used to evaluate complex expressions involving *integers*
  - It outputs the result of the evaluation
  - Usual operators are available
  - All the operators and arguments *must* be separated by white spaces
  - Rules of precedence and associativity are followed
  - Brackets can be used to group subexpressions

    - Charactes having a special meaning in the shell (like `(` and `*`) must be either quoted or escaped
    - The output of `expr` can be assigned to a variable using Command Substitution

```
expr 10 + 20
expr $x + $y
expr $x + $y \* $z
expr \( $x + $y \) \* $z
expr "(" $x + $y ")" "*" $z
expr 10 + \( $y - $x - \( $x + $y - \( 5 \* 3 \) \) \) \* 3
```

```
x=`expr $y + $z`
x=$(expr $y + $z)
```

- *let*
    - `let` is a shell builtin for integer aritmetic
    - `let` expects an expression as a single argument (so the expression must not have spaces in it; or it should be quoted)
    - `let` evaluates its argument just like `$((...))
    - `let` is not a form of Command Substitution, so assignment must be used inside the let expression to store the calculated value in some variable
    - `let` supports the increment operator `++` and the decrement operator `--`. `let i++` adds `1` to the value of the variable `i`. It is same as writing `i=i+1`. `let i--` subtracts `1` from the value of the variable `i`. It is same as writing `i=i-1`.

```
let x=10*20
let x=(10+20)*20
let x=y++
let i++
```

String Operations

- String concatenation
    - Two strings can be concatenated by simply placing them side-by-side without any intervening space

```
"abc""xyx"
$x$y
abc$x
${x}abc
```

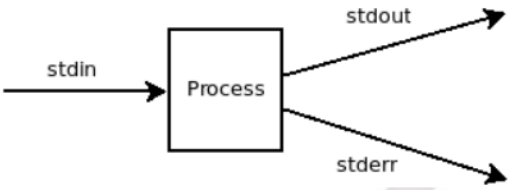
- String length
    - Using `expr`: `expr length string` outputs the length of the string
    - Using bash parameter substitution: `${#variable}` results in the length of the variable
- Substring
    - Using `expr`: `expr substr str pos len` outputs a substring of maximum length *len* from the string *str*, starting at position *pos* (the first character has the position 1)

    - Using bash parameter substitution: `${variable:off:len}` returns a substring of maximum length *len* from the value of *variable*, starting from offset *off* (the first character has the offset 0)
- String matching
    - Using `expr : expr match str regexp` performs an anchored match of the string *str* with the regular expression *regexp*

Standard I/O Streams and I/O Redirection

Every process has a PCB (Process Control Block) associated with it. One of the items in the PCB is a table of open files. It lists all the files opened by the process. In UNIX systems, a file is opened using the *open()* system call. This system call returns an integer called the *file descriptor* (fd), which is the index of the entry of that file in the table of open files.

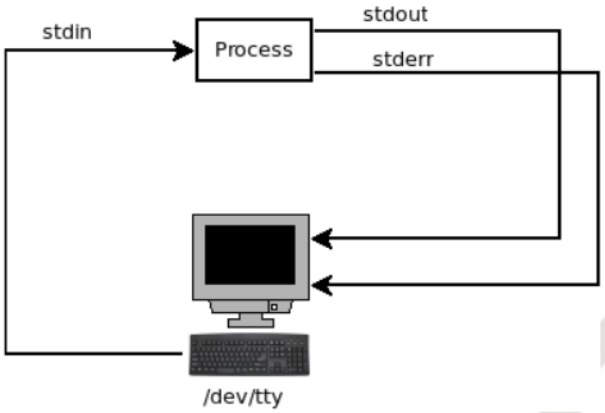
Table of Open Files
/dev/tty
/dev/tty
/dev/tty
f1
f2



Standard Console I/O Streams

Default Assignment of Standard I/O Streams

```
$ p1
```



Default Assignment of Standard I/O Streams

Table of Open Files for process p1

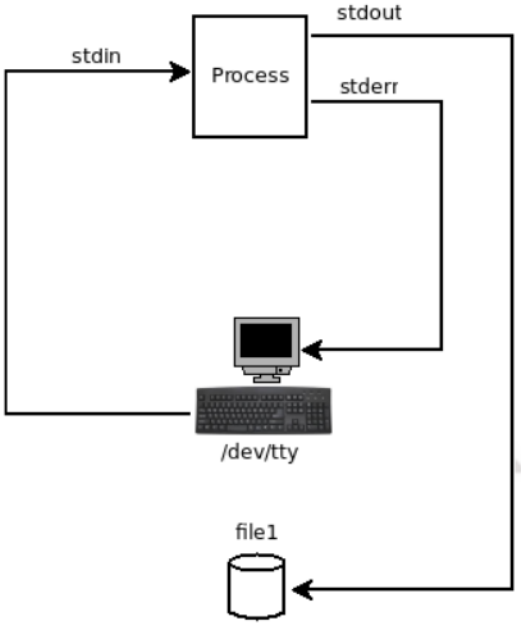
FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	/dev/tty
2	stderr	/dev/tty

Standard Output Redirection

```
$ p1 1>file1

$ p1 >file1

$ >file1 p1
```



Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	/dev/tty

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

```
$ >t1
```

This will create the file t1, if it does not exist; and will truncate the file t1, if it does.

```
$ xyz >file1     #there is no command called xyz in the path
bash: xyz: command not found
```

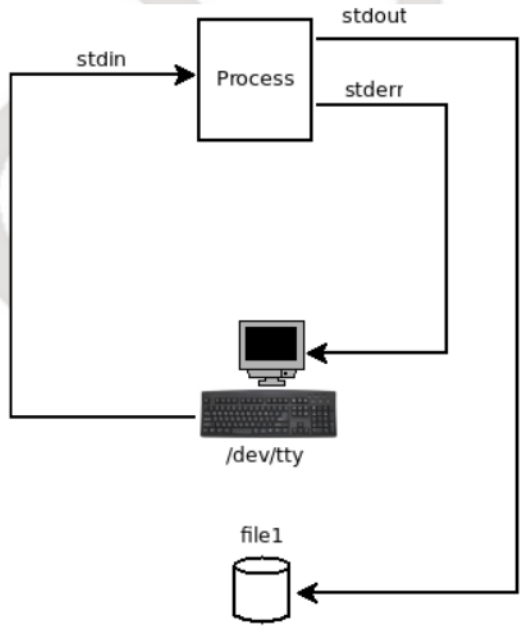
But the file file1 is still overwritten (truncated in this case).

Standard Output Redirection (Append Mode):

```
$ p1 1>>file1

$ p1 >>file1

$ >>file1 p1
```



Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	/dev/tty

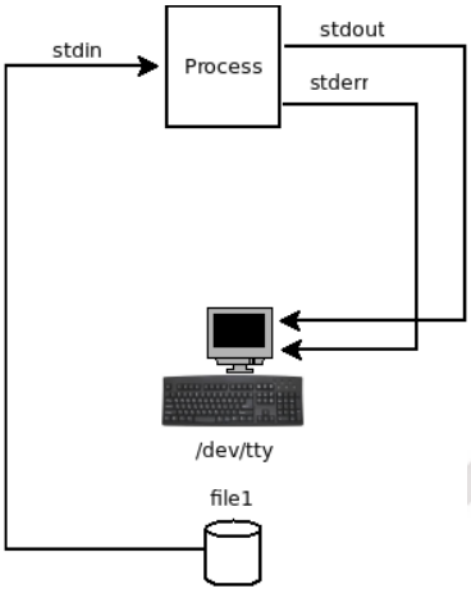
The file *file1* will be opened in *append* mode by the shell *before* execution of the command.

Standard Input Redirection

```
$ p1 0<file1

$ p1 <file1

$ <file1 p1
```



Standard Input Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	~~/dev/tty~~ file1
1	stdout	/dev/tty
2	stderr	/dev/tty

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

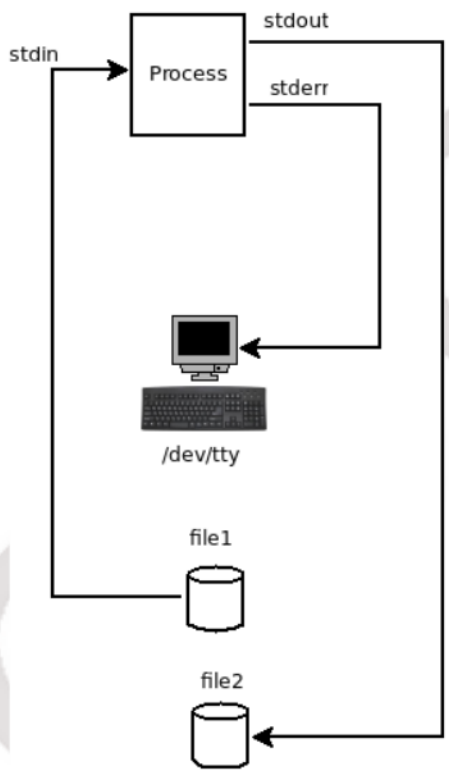
Standard Input and Standard Output Redirection

```
$ p1 0<file1 1>file2

$ p1 <file1 >file2

$ p1 >file2 <file1

$ >file2 <file1 p1
```



Standard Input and Standard Output Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	~~/dev/tty~~ file1
1	stdout	~~/dev/tty~~ file2
2	stderr	/dev/tty

The file *file1* will be opened in *read* mode by the shell *before* execution of the command. If the file does not exist, it is an error.

The file *file2* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

file1 and *file2* must not be same files. If the following command is executed:

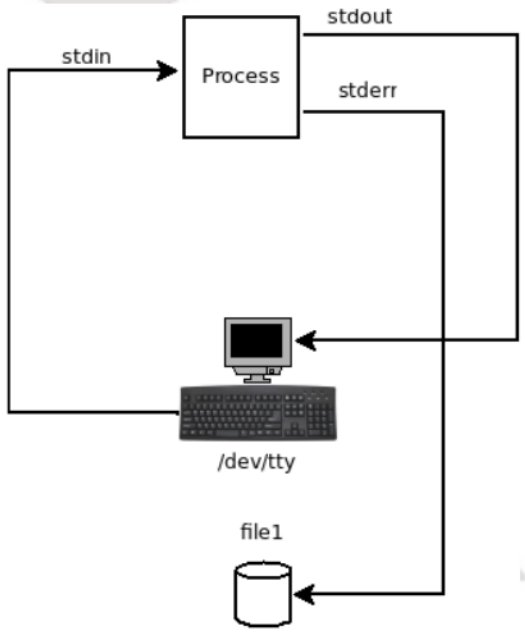
```
$ p1 <zzz >zzz
```

The file *zzz* will be overwritten *before* the execution begins and it will be an empty file.

Standard Error Redirection

```
$ p1 2>file1

$ 2>file1 p1
```



Standard Error Redirection

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	/dev/tty
2	stderr	~~/dev/tty~~ file1

The file *file1* will be opened in *write* mode by the shell *before* execution of the command. If it exists, it will be truncated.

If we use 2>> operator in place of 2> operator, the file will be opened in append mode.

```
$ p1 2>>file1
```

/dev/null is the null device. Any output sent to it is simply discarded. If we try to perform input from it, we immediately get EOF (End-of-File).

```
$ find / -name '*.php' 2>/dev/null
```

Supplying the standard input from the command line

```
$ p1 << END
INPUT LINE 1
INPUT LINE 2
INPUT LINE 3
INPUT LINE 4
END
```

This way of providing input from the command line is known as *the here document*.

Command Substitution

Command substitution means constructing the command line by substituting a command in it by the standard output of that command.

```
$ p1 p1_arg1 p1_arg2 `p2 p2_arg1 p2_arg2 p2_arg3 ...` p1_argx p1_argy p1_argz
```

The command

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

will be executed as an independent command and the standard output from it will replace the whole backquoted string in the original command line. Then the modified command line will be executed.

E.g., if the standard output of

```
p2 p2_arg1 p2_arg2 p2_arg3 …
```

is

```
aaa bbb ccc
```

then the modified command line becomes

```
p1 p1_arg1 p1_arg2 aaa bbb ccc p1_argx p1_argy p1_argz
```

Which will then be executed.

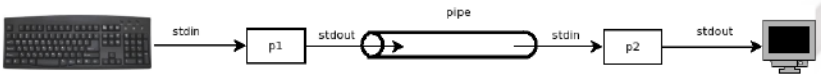
```
 x=10
y=20
z=`expr $x + $y`
+ expr 10 + 20
z=30
```

New Syntax

```
$ p1 p1_arg1 p1_arg2 $(p2 p2_arg1 p2_arg2 p2_arg3 ...) p1_argx p1_argy p1_argz
```

Pipes

```
$ p1 | p2
```



Pipe

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ pipe1
2	stderr	/dev/tty

Table of Open Files for process p2

FD	Name	Associated with
0	stdin	~~/dev/tty~~ pipe1
1	stdout	/dev/tty
2	stderr	/dev/tty

The processes p1 and p2 will be run concurrently as child processes of the shell, with the standard output of p1 redirected to a pipe and p2 taking its standard input from that pipe.

```
$ p1 | p2 | p3 | p4 | p5
```

p1 will take its standard input from the terminal (keyboard). Standard output of p1 will be supplied as standard input to p2. Standard output of p2 will be supplied as standard input to p3. Standard output of p3 will be supplied as standard input to p4. Standard output of p4 will be supplied as standard input to p5. Standard output of p5 will appear on the terminal (monitor). In a chain of commands, each process carries out one specific stage of text processing.

Redirecting One File Descriptor to Another File Descriptor

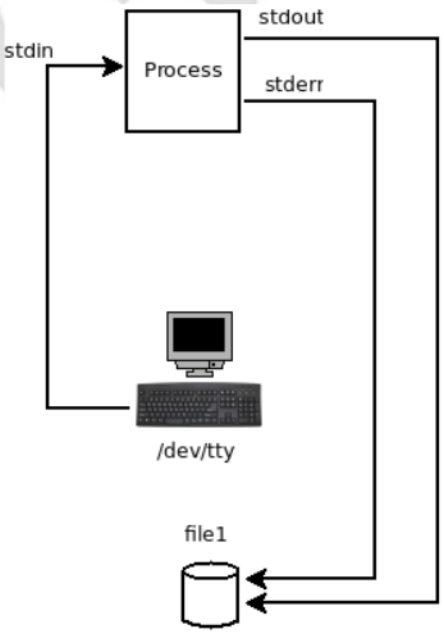
```
$ p1 >file1 2>file1
```

This will not have the desired effect.

```
$ p1 >file1 2>&1
```

This means send standard error wherever File Descriptor 1 (standard output) is going.

First the standard output will be redirected to file1 and then the standard error will be redirected to wherever the standard output is going, i.e. file1. Hence, both standard output and standard error will be redirected to file1.



Standard Output and Standard Error Redirection to the Same File

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1
2	stderr	~~/dev/tty~~ file1

In this case order of redirection *is* significant!

Suppose, instead of

```
$ p1 >file1 2>&1
```

we run

```
$ p1 2>&1 >file1
```

Table of Open Files for process p1

FD	Name	Associated with
0	stdin	/dev/tty
1	stdout	~~/dev/tty~~ file1

FD	Name	Associated with
2	stderr	/dev/tty

First the standard error will be redirected to wherever the standard output is going, i.e. the terminal and then the standard output will be redirected to file1.

Arguments for Different Redirection Operators

command1	>	file1
command1	>>	file1
Command1	2>	file1
command1	2>>	file1
command1	<	file1
command1	|	command2
command1	<<	WORD

Filters

A *filter* is a command that takes its input from standard input, processes it and produces the output on standard output. The benefit of a filter is that its input and output can be redirected independently and flexibly. Also, several filters can be chained in a pipeline.

```
cat a4 | grep '\^\[0-9\]\*,\[0-9\]\*,12'
```

Regular Expressions

Basic Regular Expressions

Pattern	Matches with
.	Any single character
*	Previous element 0 or more times
^	An imaginary anchor at the beginning of line (only when ^ is at the beginning of the pattern)
$	An imaginary anchor at the end of line (only when $ is at the end of the pattern)

\ is the *escape character* - it removes the special meaning of the next character; also adds special meaning to some characters in some circumstances as described below.

Character Classes

Pattern	Matches with
[abcd]	Any one character from a, b, c or d
[a-z]	Any one character between a and z, inclusive
[a-z0-9]	Any one character from between a and z, inclusive; or from between 0 and 9, inclusive
[a-zA-Z0-9]	Any one character from between a and z, inclusive; or from between A and Z, inclusive; or from between 0 and 9, inclusive
[a-z0-9ABCD]	Any one character from between a and z, inclusive or from between 0 and 9, inclusive or A or B or C or D
[^a-z0-9]	Any one character other than those between a and z, inclusive and those between 0 and 9, inclusive
[a-z^0-9]	Any one character from between a and z, inclusive or ^ or from between between 0 and 9, inclusive

Pattern	Matches with
\t	The TAB character
\n	The newline character
\\	The (backslash) character
\w	equivalent to [a-zA-Z0-9]
\W	equivalent to [^a-zA-Z0-9]

Predefined Character Classes

Pattern	Matches with	Equivalent to
[:alpha:]	Any alphabetic character	[a-zA-Z]
[:digit:]	Any digit	[0-9]
[:alnum:]	Any alphanumeric character	[a-zA-Z0-9]
[:lower:]	Any lower case letter	[a-z]
[:upper:]	Any upper case letter	[A-Z]
[:space:]	Any white space character	space/tab/new line
[:punct:]	Any punctuation character	
[:xdigit:]	Any hexadecimal digit	[0-9a-fA-F]
[:print:]	Any printable character	
[:cntrl:]	Any control character	
[:graph:]	Any ASCII graph character	

Word Matching

Pattern	Matches with
\<	An imaginary anchor at the beginning of every word
\>	An imaginary anchor at the end of every word
\<pat1\>	The input matches with the pattern \<pat1\> only if the input matches with pattern pat1 and is a whole word (not part of a larger word)

Extended Regular Expressions

The meta characters used in extended regular expressions are as follows. These meta characters can be used as given (individually) when extended regular expressions are enabled (e.g. grep -E). The grep command also supports them in basic regular expressions in their escaped form (e.g. \+ instead of +, \? instread of ?, \{ instread of {, etc.).

Pattern	Matches with
+	Previous element 1 or more times
?	Previous element 0 or 1 time

Repetition

{*m*} in a pattern means the previous element *m* times. {*m*,*n*} in a pattern means the previous element between *m* and *n* times. {*m*,} in a pattern means the previous element *m* or more times.

Notes:

- The input abc will match with the pattern .{3} because the pattern .{3} is treated as equivalent to the pattern ...
- The pattern matchers are usually greedy; they try to match the given pattern or part of the pattern with the maximum possible amount of input.

Alternation

pattern1 | pattern2 matches any input that matches with either pattern1 or pattern2

Subexpressions / Groups / Tags and Back References

Any part of the search pattern enclosed between (and) is treated as a subexpression / group (considered to have been tagged). The exact input that matches with a subexpression is remembered and can be referenced later as *n*, where *n* is the subexpression number (you can have multiple subexpressions in your search pattern and they are numbered serially starting from 1, so \1 means the input that matched with the first subexpression, \2 means the input that matched with the second subexpression and so on).

Examples:

Palindrome strings of 5 characters:

```
^\(.\)\(.\).\2\1$
```

Line containing only valid numbers (integers, floating point numbers in usual notation, numbers in scientific notation):

```
^[+-]\?[0-9]\+\(\.[0-9]*\)\?\([eE][-+]\?[0-9]\+\)\?$
```

The grep Family of Commands

- **grep:** regular grep, did not support multiple patterns and extended regular expressions
- **egrep:** extended grep, supported multiple patterns and extended regular expressions
- **fgrep:** fast grep, did not support patterns (can search only for fixed strings), but supported multiple search strings

Note: under GNU/Linux, the functionalities of all three commands have been combined into a single command - grep

grep -E is equivalent to egrep

grep -F is equivalent to fgrep

Feature	grep	egrep (extended)	fgrep (fast)
Regular Expressions	Yes	Yes	No
Extended Regular Expressions	No	Yes	No
Multiple Patterns	No	Yes	Yes
Speed	Medium	Medium	Fast

Combining Multiple Commands using Logical Operators

Note: An exit status of 0 means success, while any non-zero exit status means failure

- `cmd1 ; cmd2` cmd1 is executed and then cmd2 is executed (always)
- `cmd1 && cmd2` cmd1 is executed and then cmd2 is executed only if cmd1 is successful
- `cmd1 || cmd2` cmd1 is executed and then cmd2 is executed only if cmd1 is unsuccessful
- `(cmd1;cmd2)` the command list is executed in a child shell
- `{cmd1;cmd2;}` the command list is executed in the present shell

Vim Editor in More Detail

Commands List

- **File System Handling**
  - **pwd** Displays the present working directory
  - **ls** Displays list of files
    - **-a** Display all files, including hidden ones and the implied . and .. directories
    - **-A** Display all files, including hidden ones but excluding the implied . and .. directories
    - **-d** When an argument is a directory, display the directory (default is to display the contents of the directory)
    - **-h** Display human-readable size (20M, 1.5G, etc.)
    - **-l** Display a long list
    - **-r** Use reverse (descending) order while sorting
    - **-R** Display the contents of the arguments recursively (in case of a directory, display the subdirectories, subsubdirectories, etc.)
    - **-S** Sort list on file size
    - **-t** Sort list on modification time
    - **-X** Sort list on filename extension
    - **--color=always|never|auto** Produce color output always/never/decide automatically based on the standard output destination
  - **cat** Output standard input / a file / concatenate multiple files and output
  - **cd** Change directory
  - **mkdir** Make one or more directories
  - **rmdir** Remove one or more directories (must be empty)
  - **rm** Remove file(s) (and directories)
    - **-r** Remove file(s) and directories recursively
    - **-i** Remove interactively (prompt before deleting each file/directory)
    - **-f** Ignore non-existant files, never prompt
  - **cp** Copy file(s)
    - **-r** Copy file(s) and directories recursively
    - **-i** Copy interactively (prompt before deleting each file/directory)
  - **mv** Move/rename files and directories
    - **-i** Move/rename files and directories interactively (prompt before deleting each file/directory)
    - **-f** Move/rename forcefully, do not prompt before overwriting
  - **mount** Mount a file system onto a directory
    - **-t** Specify the file system type
    - **-a** Mount all file systems from /etc/fstab having the auto option
    - **-o** Supply mount options (e.g. -o loop, -o rw)
    - **-r** Mount read only
    - **-w** Mount for read/write
  - **umount** Unmount a file system
- **Editing**
  - **vim**
- **Process Management**
  - **ps** process Status; display a list of running processes (by default, only processes in the current session)
    - **-a** (all) processes belonging to the current user sessions except the session leaders (like the shell process that is the root of the process hierarchy for the session)

    - **-e** (every) Displays all the interactive as well as non-interactive processes of all the users, including the system processes
    - **-f** (full) Show the full information including parent process id and the full command line
  - **kill** Send a signal to another process
    - **-9** Send the kill signal to another process to kill it
- **Basic Filters**
  - **cut** Extract one or more fields (columns) from input
    - **-c** Specify column positions (for use with fixed-width input)
    - **-d** Specify delimiter (for use with delimited input)
    - **-f** Specify fields to be extracted (for use with delimited input)
  - **grep (egrep, fgrep)**
    - **-E** Enable extended regular expression
    - **-e** Specify a pattern (used to specify multiple patterns)
    - **-F** Use fixed strings (work like fgrep)
    - **-f** Specify a file containing patterns, one per line
    - **-i** Ignore case when matching patterns
    - **-c** Output a count of matching lines
    - **-v** Invert match, output lines other than those having a match for the pattern(s)
    - **-w** word match, pattern must match with a whole word in the line
    - **-l** Output only a list of files that have one or more matches
    - **-n** Output line numbers along with matching lines
    - **-o** Output only the matched (non-empty) parts of a matching line, with each such part on a separate output line
    - **--color=always|none|auto** Produce color output always/never/decide automatically based on the standard output destination
  - **wc** Word Count; count the number of lines, words and characters in standard input or a file
    - **-l** Output line count
    - **-w** Output line count
    - **-c** Output line count
  - **head** Display the first few lines of a file or standard input (10 by default)
    - **-n** number of lines from the beginning to be displayed
  - **tail** Display the last few lines of a file or standard input (10 by default)
    - **-n** number of lines from the end to be displayed
    - **-f** continue to wait for further input even after reaching end of file; useful for watching a continuously growing file
  - **sort** Sort the input
    - **-k** Specify key (starting column, ending column) on which to sort. In case of delimited input, specify starting and ending field number, e.g. -k2,2. In case of fixed-width input, specify 1.starting character position and 1.ending character position, e.g. -k1.5,1.10 May be repeated for sorting on multiple keys
    - **-r** sort in reverse (descending) order
    - **-t** Specify the delimiter
    - **-d** Sort as per dictionary (lexical) order (default)
    - **-f** Fold lowercase to uppercase (ignore case while sorting)
    - **-n** Sort numerically

        - **-h** Sort numerically, human readable sizes (K, M, G, etc.)
    - **uniq** Output unique or repeated lines (assumes the input is sorted)
        - **-c** Prefix lines by a count of how many times the line is repeated
        - **-d** Only output duplicated (repeated) lines
        - **-i** ignore case
        - **-u** Only output unique lines (default)
    - **tr** Translate - convert or delete individual characters
        - **-s** Squeeze - convert multiple consecutive occurrences of the given characters into a single occurrence
        - **-d** Delete the given characters from the input
    - **tee** Copy standard input to given file as well as to standard output
        - **-a** append to file rather than overwrite
- **Other Commands**
    - **clear** clear the screen
    - **more** displays a file or standard input one page at a time. Press ENTER for next line, SPACE for next page, b for previous page (not supported with pipes), q to quit, /patternENTER search, n/N to search again in forward/backward direction
    - **less** More powerful version of more, supports two-way scrolling with pipes as well
    - **echo** outputs its arguments followed by a newline
        - **-n** do not output a newline at the end
    - **read var1 var2 var3...** reads a line, performs word-splitting and assigns each word to corresponding variable. If there are less words than the number of variables then the remaining variables remain unassigned. If there are more words than the number of variables then one word is assigned to each variable except the last one; the last variable holds all remaining input
        - **-p** display the argument as a prompt if the input comes from a terminal
        - **-s** do not echo (display) the characters as they are read (useful for input of passwords)
    - **test** test various conditions
        - **string tests**
            - **-z *str*** True of the string *str* is zero length (empty string)
            - **-n *str*** True of the string *str* is non-zero length
            - ***str1* = *str2*** True if *str1* is equal to *str2* (string comparison)
            - ***str1* != *str2*** True if *str1* is not equal to *str2* (string comparison)
        - **integer tests**
            - ***no1* -gt *no2*** True if *no1* is greater than *no2*
            - ***no1* -ge *no2*** True if *no1* is greater than or equal to *no2*
            - ***no1* -lt *no2*** True if *no1* is less than *no2*
            - ***no1* -le *no2*** True if *no1* is less than or equal to *no2*
            - ***no1* -eq *no2*** True if *no1* is equal to *no2* (integer comparison)
            - ***no1* -ne *no2*** True if *no1* is not equal to *no2* (integer comparison)
        - **file related tests**
            - **-e *file1*** True if the file *file1* exists
            - **-d *file1*** True if the file *file1* exists and is a directory
            - **-f *file1*** True if the file *file1* exists and is a regular file
            - **-r *file1*** True if the file *file1* exists and is readable for the current user
            - **-w *file1*** True if the file *file1* exists and is writable for the current user

        - **-x *file1*** True if the file *file1* exists and is executable for the current user
        - **-s *file1*** True if the file *file1* exists and has a size greater than zero
    - **expr** evaluate expressions
        - **arithmetic expressions** expr
        - **string expressions** String Operations
    - **bc** arbitrary precision integer and floating arithmetic tool
    - **printf** Works just like the `printf()` function in the C programming language. As it is a command, its syntax is slightly different: `printf format-string arg1 arg2 arg3 ...`
    - **man** show the manual page for the argument from the first section in which it exists
        - **-a** show manual pages for the argument from all the sections in which it exists one after another
- Advanced Filters
    - **sed** Stream editor. Used to edit a file programmatically (from script)
        - **-n** Do not perform default output
        - **-f** Specify sed script filename
        - **commands**
            - **s** Substitute
            - **i** Insert line(s) before
            - **a** Append line(s) after
            - **d** delete line
            - **p** print (output) line
    - **awk** Powerful data manipulation program
        - **-f** Specify awk script filename
        - **Built-in Variables**
            - **FNR** File Number of Record (line number of the current line in the current file)
            - **FS** Field separator
            - **NF** Number of fields in the current line
            - **NR** Number of Record (a cumulative line number for all the lines from all the files processed so far)
            - **OFS** Output field separator (inserted between fields by print)
        - **Built-in Functions** (Note: s/t → string, r → regular expression, n → number, i → index, a → array)
            - **length(s)** length of string s
            - **index(s, t)** 1-based index of substring t in s, 0 if t is not found in s
            - **match(s, r)** returns the 1-based index of the first longest match of regular expression r in s, 0 otherwise
            - **split(s, a, r)** Splits string s into fields using the separators in r (if omitted, default is white space) and stores them into array a as elements starting with index 1
            - **sub(r, s, t)** Substitution. First match of r in t is replaced by s
            - **gsub(r, s, t)** Global substitution. Every match of r in t is replaced by s
            - **substr(s, i, n)** Return substring of s, starting at index i (1-based) and of length n. If n is omitted, returns a suffix of s starting at i
            - **tolower(s)** Converts s to lower case
            - **toupper(s)** Converts s to upper case
            - **print(expr-list)** Print (output) each argument separated by OFS

› **printf(format, expr_list)** Just like the printf function in the C programming language