Unit-I Fundamentals

Cloud Computing and DevOps

Ms. Vidya S. Gaikwad
vidya.gaikwad@viit.ac.in
Department of Computer Engineering

BRACT's, Vishwakarma Institute of Information Technology, Pune-48

(An Autonomous Institute affiliated to Savitribai Phule Pune University)
(NBA and NAAC accredited, ISO 9001:2015 certified)

Contents

- **Network Fundamentals:**
- The OSI Model
- TCP vs UDP
- IP addressing & Subnetting
- Routing & Firewall
- Storage Fundamentals: Block Storage, Object Storage, File storage, SAN, NAS
- **Linux Introduction and Essential Commands:**
- Introduction
- History
- Usage
- Flavours
- **Linux Commands Shell Scripting:** Basics, Arithmetic & Logical Operations, Cron, Loops

Linux Introduction and Essential Commands

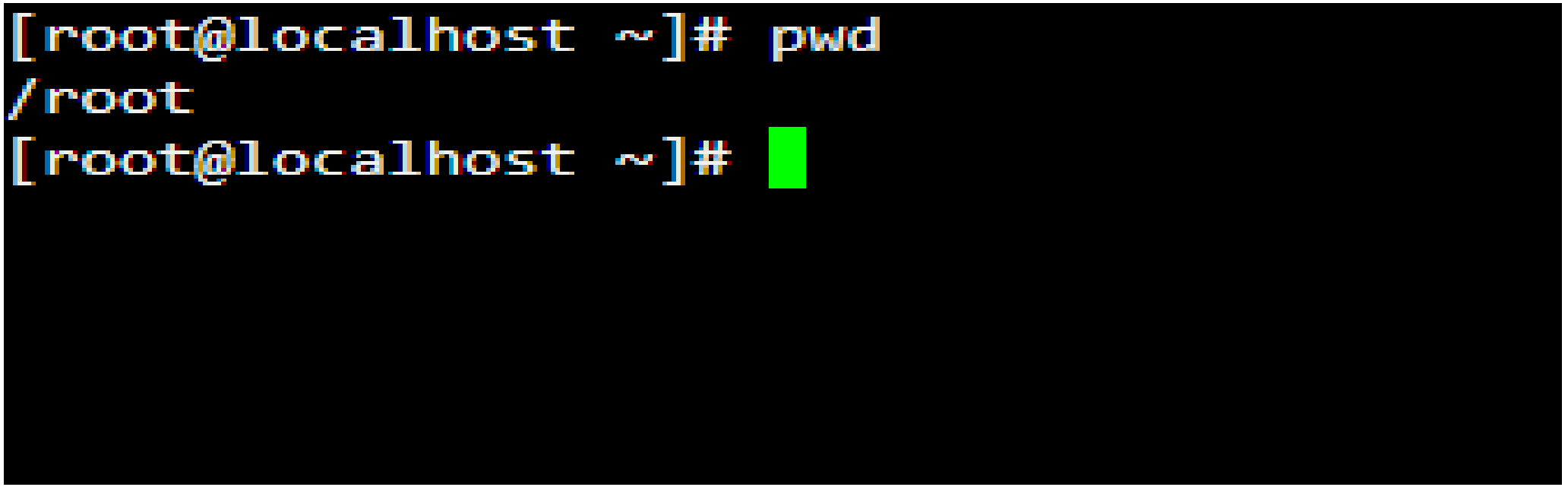
- The Linux command is **a utility of the Linux operating system**.
- All basic and advanced tasks can be done by executing commands.
- The commands are executed on the **Linux terminal**.
- The terminal is a command-line interface to interact with the system, which is similar to the command prompt in the Windows OS.
- Commands in Linux are **case-sensitive**.
- Linux provides a powerful command-line interface compared to other operating systems such as Windows and MacOS.
- We can do **basic work** and **advanced work** through its terminal.
- **Basic tasks such as creating a file, deleting a file, moving a file, and more**.
- **Advanced tasks** such as **administrative tasks (including package installation, user management), networking tasks (ssh connection), security tasks**, and many more.

Linux Introduction and Essential Commands

- **Linux Directory Commands**

1. pwd Command

- The '$pwd' command stands for '**print working directory**' and as the name says, it displays the directory in which we are currently working.

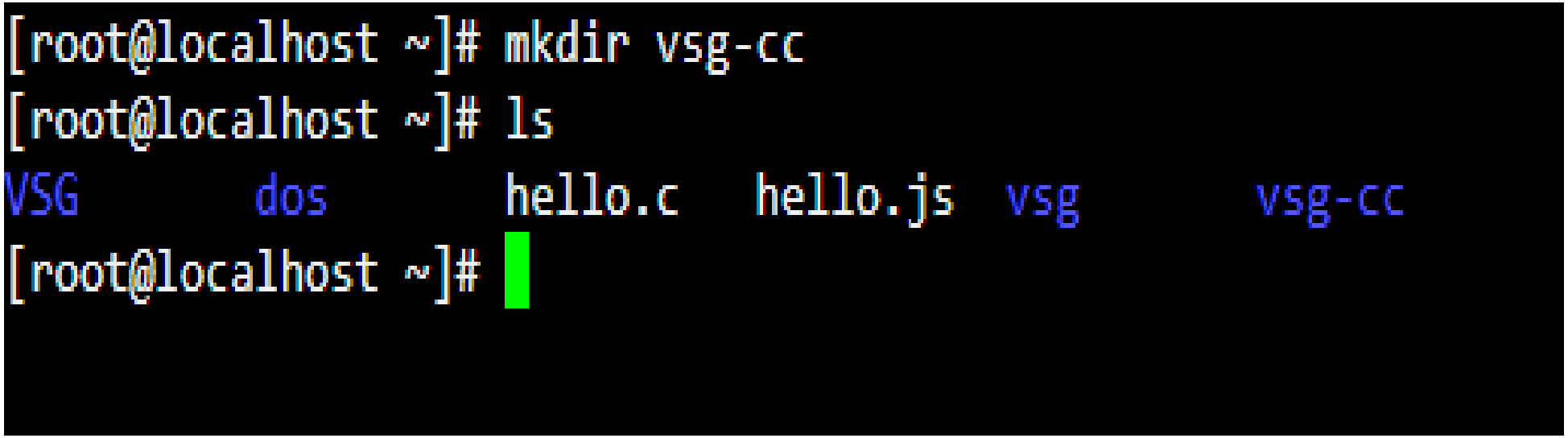


Linux Introduction and Essential Commands

2) mkdir:

The '$ mkdir' stands for '**make directory**' and **it creates a new directory**.

3) **ls :** The 'ls' command simply **displays the contents of a directory.**



Linux Introduction and Essential Commands

4) rmdir : The '$ rmdir' command **deletes any directory we want to delete and it stands for 'remove directory'**

```
[root@localhost ~]# rmdir vsg-cc
[root@localhost ~]# ls
VSG        dos        hello.c   hello.js  vsg
[root@localhost ~]#
```

Linux Introduction and Essential Commands

5) c d:

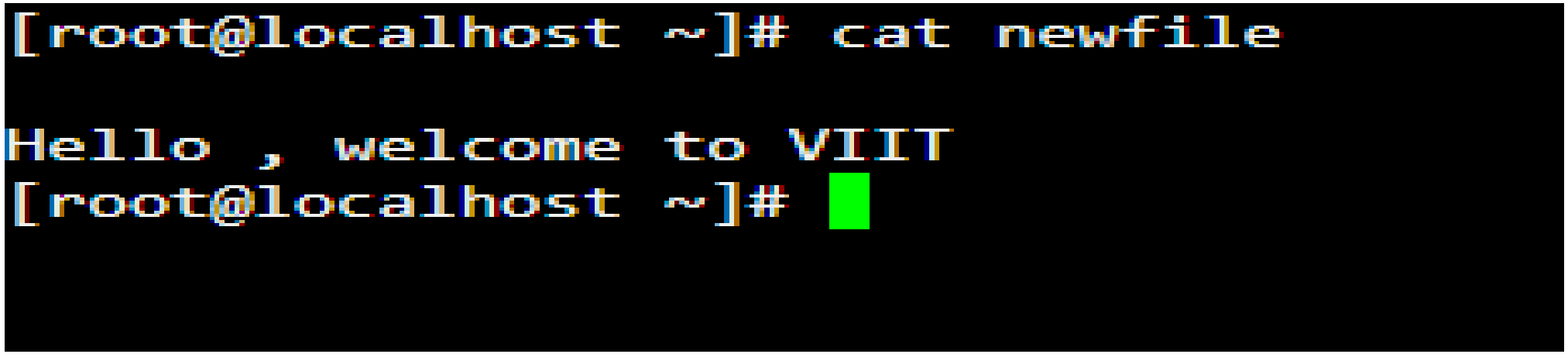
- The '$ cd' command stands for '**change directory**' and it changes your current directory to the 'newfolder' directory.

```
[root@localhost ~]# cd vsg
[root@localhost vsg]#
```

Linux Introduction and Essential Commands

6) c at command

- Concatenate, or **cat**, is one of the most frequently used Linux commands.
- It **lists, combines, and writes file content to the standard output**.
- To run the cat command, type **cat** followed by the file name and its extension.



Linux Introduction and Essential Commands

- **cp :** This '$ cp ' command stands for 'copy' and it simply copy/paste the file wherever you want to.

Linux Introduction and Essential Commands

- **c al :** The ‘$ cal’ means **calendar** and it simply **display calendar on to your screen**.

Contents

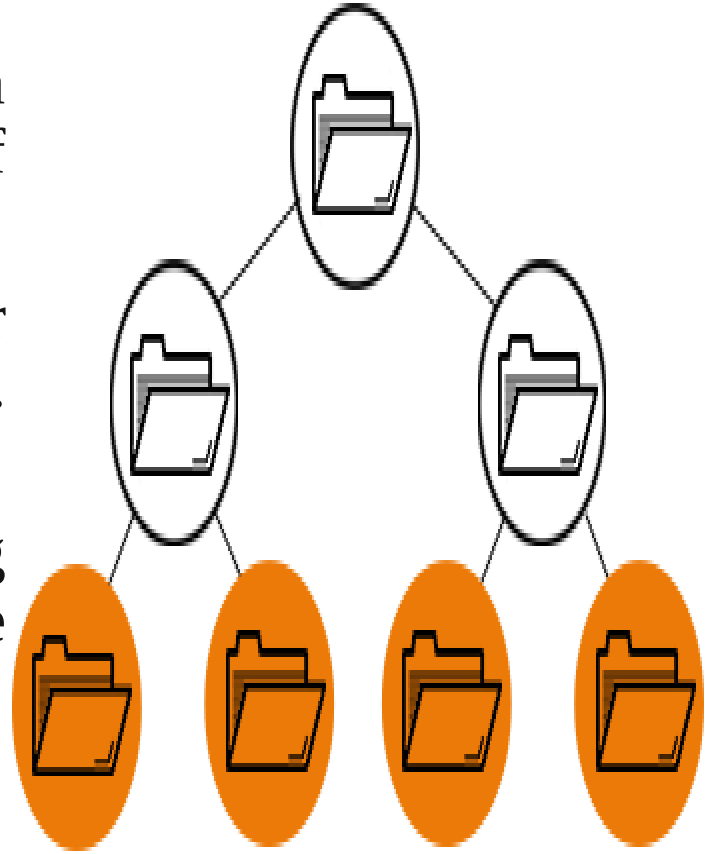
- **Network Fundamentals:**
- The OSI Model
- TCP vs UDP
- IP addressing & Subnetting
- Routing & Firewall
- **Storage Fundamentals: Block Storage, Object Storage, File storage, SAN, NAS**
- **Linux Introduction and Essential Commands:**
- Introduction
- History
- Usage
- Flavours
- **Linux Commands Shell Scripting:** Basics, Arithmetic & Logical Operations, Cron, Loops

Storage Fundamentals: Block Storage, Object Storage, File storage, SAN, NAS [1]

- Files, blocks, and objects are storage formats that hold, organize, and present data in different ways—each with their own capabilities and limitations.
- File storage organizes and represents data as a hierarchy of files in folders; block storage chunks data into arbitrarily organized, evenly sized volumes; and object storage manages data and links it to associated metadata.
- Containers are highly flexible and bring incredible scale to how apps and storage are delivered.

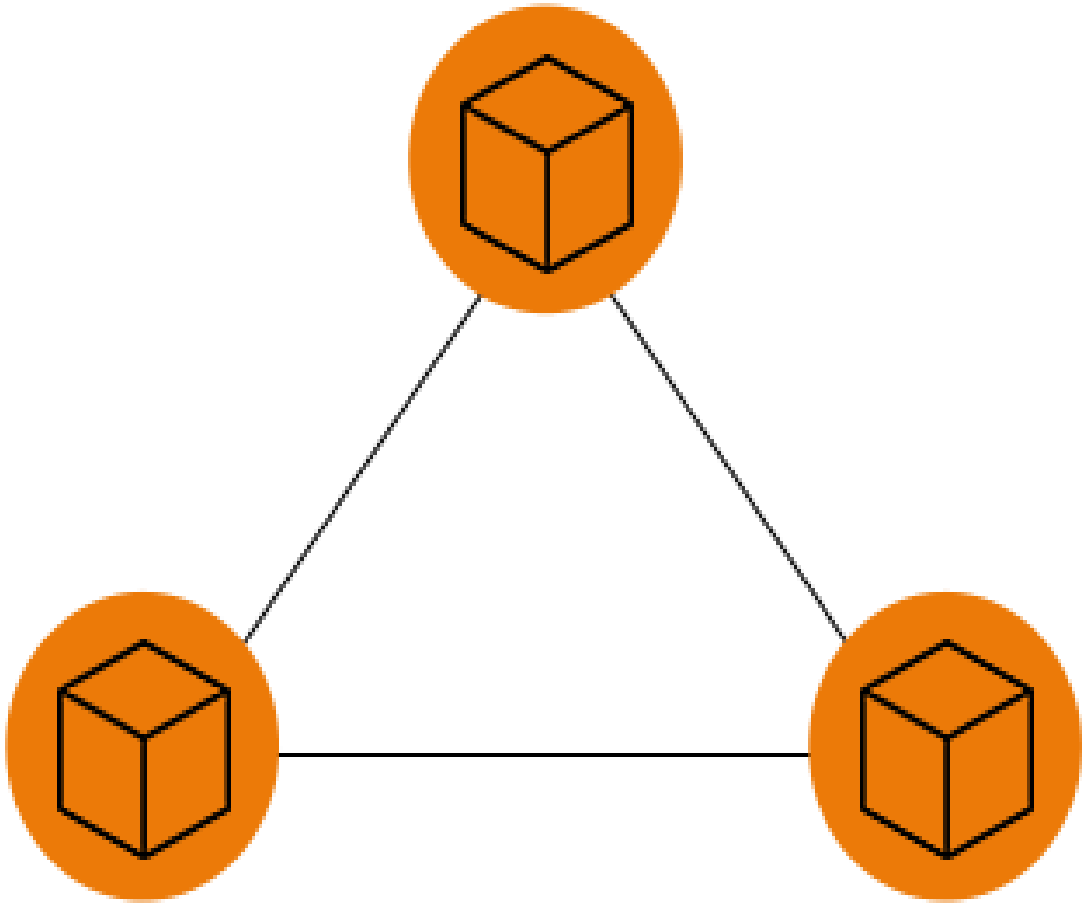
What is file storage?

- File storage, also called file-level or file-based storage.
- Data is stored as a single piece of information inside a folder, just like you'd organize pieces of paper inside a manila folder.
- When you need to access that piece of data, your computer needs to know the path to find it. (Beware—It can be a long, winding path.)
- Data stored in files is organized and retrieved using a limited amount of metadata that tells the computer exactly where the file itself is kept.
- It's like a library card catalogue for data files.



What is bloc k storage?

- Block storage chops data into blocks—get it?—and stores them as separate pieces.
- Each block of data is given a unique identifier, which allows a storage system to place the smaller pieces of data wherever is most convenient.
- That means that some data can be stored in a Linux® environment and some can be stored in a Windows unit.

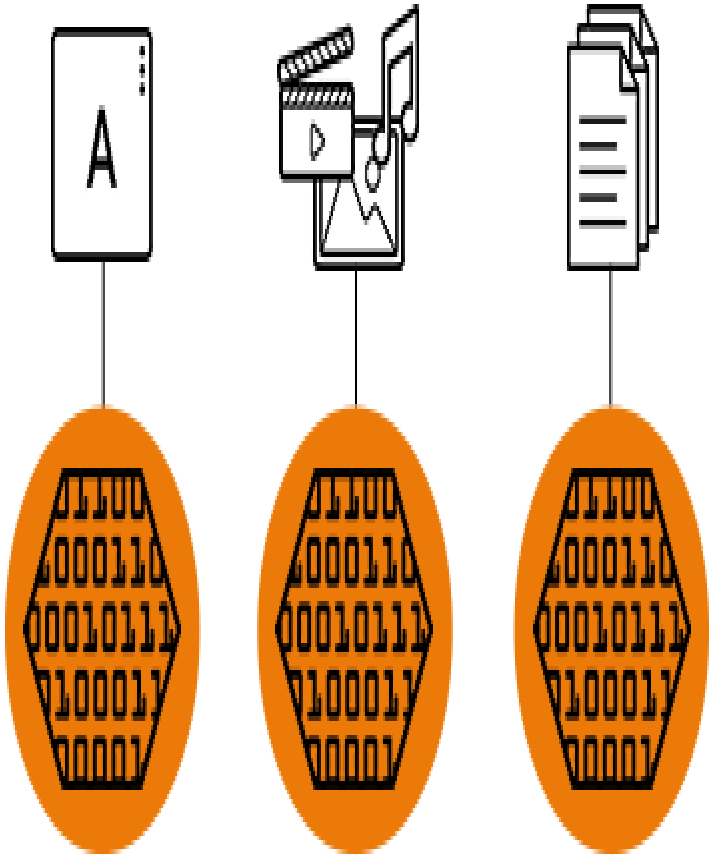


What is bloc k storage?

- Block storage is often configured to decouple the data from the user's environment and spread it across multiple environments that can better serve the data.
- And then, when data is requested, the underlying storage software reassembles the blocks of data from these environments and presents them back to the user.
- It is usually deployed in storage-area network (SAN) environments and must be tied to a functioning server.
- Block storage doesn't rely on a single path to data—like file storage does—it can be retrieved quickly.
- Block storage can be expensive.

What is objec t storage?

- Object storage, also known as object-based storage, is a flat structure in which files are broken into pieces and spread out among hardware.
- In object storage, the data is broken into discrete units called objects and is kept in a single repository, instead of being kept as files in folders or as blocks on servers.
- Object storage volumes work as modular units: each is a self-contained repository that owns the data, a unique identifier that allows the object to be found over a distributed system, and the metadata that describes the data.



What is objec t storage?

- To retrieve the data, the storage operating system uses the metadata and identifiers, which distributes the load better and lets administrators apply policies that perform more robust searches.
- Object storage requires a simple HTTP application programming interface (API), which is used by most clients in all languages.
- Object storage is cost efficient: you only pay for what you use.
- It can scale easily, making it a great choice for public cloud storage.
- It’s a storage system well suited for static data, and its agility and flat nature means it can scale to extremely large quantities of data.
- The objects have enough information for an application to find the data quickly and are good at storing unstructured data.

Capability	Object Storage	File Storage	Block Storage
Consistency	Eventual consistency	Strong consistency	Strong consistency
Structure	Unstructured	Hierarchically structured	Highly structured at block level
Access Level	Object level	File level	Block level

- Cloud Computing, like any computing, is a combination of CPU, memory, networking, and storage.
- **Infrastructure as a Service** (IaaS) platforms allow you to store your data in either Block Storage or Object Storage formats.
- **Use c ases for Block storage**
- **Ideal for databases**, since a DB requires consistent I/O performance and low-latency connectivity.
- Use block storage for **RAID Volumes**, where you combine multiple disks organized through stripping or mirroring.
- Any application which requires **service side processing**, like Java, PHP, and .Net will require block storage.
- Running **mission-critical** applications like Oracle, SAP, Microsoft Exchange, and Microsoft SharePoint.

- **Bloc k storage options in the c loud**

1.AWS Elastic Bloc k Storage (EBS):

1. Amazon EBS provides raw storage – just like a hard disk – which you can attach to your Elastic Cloud Compute (EC2) instances.
2. Once attached, you create a file system and get immediate access to your storage.
3. You can create EBS General Purpose (SSD) and Provisioned IOPS (SSD) volumes up to 16 TB in size, and slower, legacy magnetic volumes.

2.Rac kspace Cloud Bloc k Storage:

1. Rackspace provides raw storage devices capable of delivering super fast 10GbE(Gigabit Ethernet) internal connections.

3.Azure Premium Storage:

1. Premium Storage delivers high-performance, low-latency disk support for I/O intensive workloads running on Azure Virtual Machines. Volumes allow up to 32 TB of storage.

4.Google Persistent Disks:

1. Compute Engine Persistent Disks provide network-attached block storage, much like a high speed and highly reliable SAN, for Compute Engine instances.
2. You can remove a disk from one server and attach it to another server, or attach one volume to multiple nodes in read-only mode.
3. Two types of block storage are available: Standard Persistent Disk and Solid-State Persistent Disks.

- **Use Cases for Objec t Storage**
- Storage of **unstruc tured data** like music, image, and video files.
- Storage for backup files, database dumps, and log files.
- Large data sets. Whether you're storing pharmaceutical or financial data, or multimedia files such as photos and videos, storage can be used as your **big data** object store.
- Archive files in place of local tape drives. Media assets such as **video footage** can be stored in object storage and archived to AWS glacier.

- **Objec t storage options in the Cloud**

1.Amazon S3:

1. Amazon S3 stores data as objects within resources called “buckets.”
2. AWS S3 offers features like 99.999999999% durability, cross-region replication, event notifications, versioning, encryption, and flexible storage options (redundant and standard).

2.Rac kspace Cloud Files:

1. Cloud Files provides online object storage for files and media.
2. Cloud Files writes each file to three storage disks on separate nodes that have dual power supplies.
3. All traffic between your application and Cloud Files uses SSL to establish a secure, encrypted channel.
4. You can host static websites (for example: blogs, brochure sites, small company sites) entirely from Cloud Files with a global CDN.

- **Objec t storage options in the Cloud**

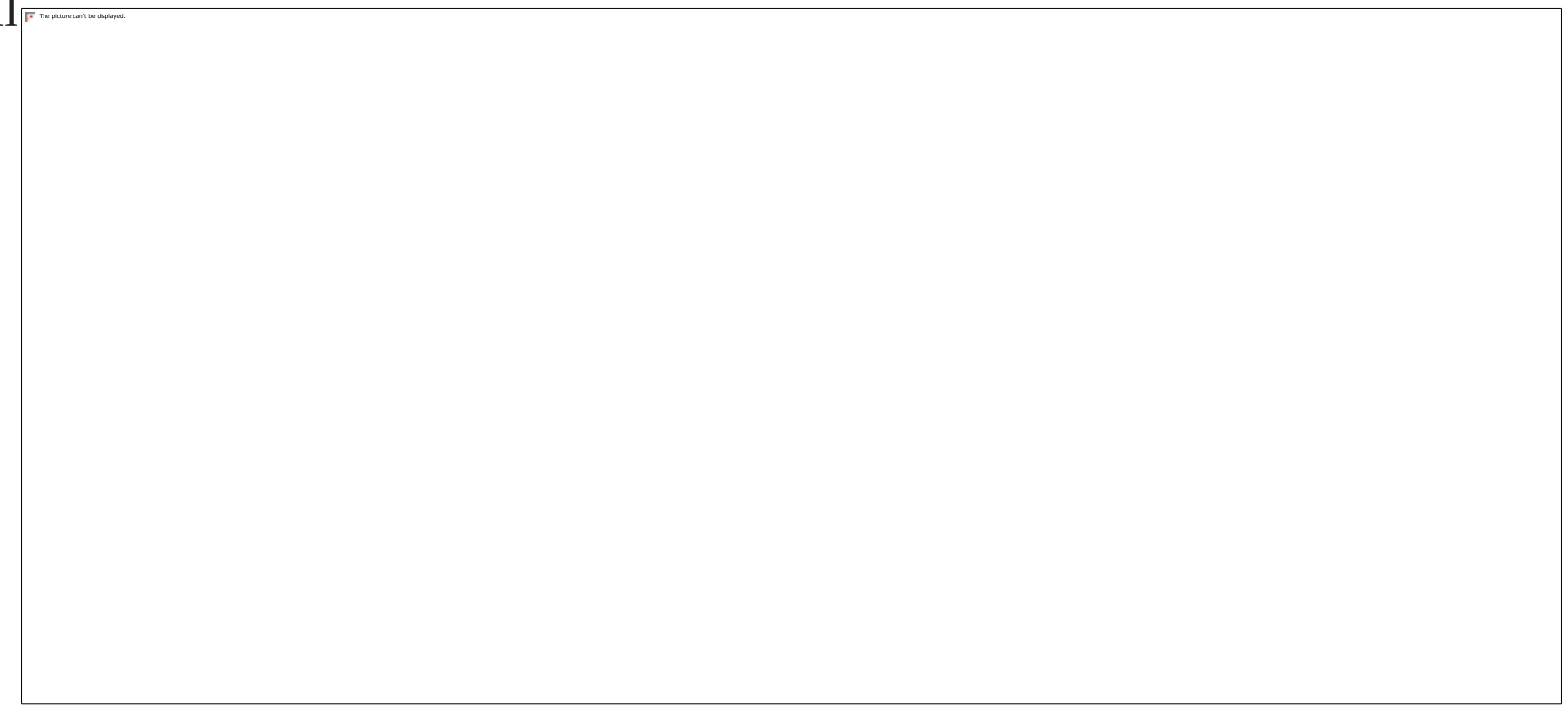
3.Azure Blob Storage:

1. For users with large amounts of unstructured data to store in the cloud, Blob storage offers a cost-effective and scalable solution.
2. Every blob is organized into a container with up to a 500 TB storage account capacity limit.

4.Google cloud storage:

1. Cloud Storage allows you to store data in Google's cloud.
2. Google Cloud Storage supports individual objects that are terabytes in size.
3. It also supports a large number of buckets per account.
4. Google Cloud Storage provides strong read-after-write consistency for all upload and delete operations.
5. Two types of storage class are available: Standard Storage class and Storage Near line class (with Near Line being **MUCH** cheaper).

- **What is Shell?**
- **Shell** is a UNIX term for an interface between a user and an operating system service.
- Shell provides users with an interface and accepts human-readable commands into the system and executes those commands which can run automatically and give the program's output in a shell script.
- An Operating is made of many components, but its two prime components are –
  - Kernel
  - Shell



- **What is Shell?**
- A Kernel is at the nucleus of a computer.
- It makes the communication between the hardware and software possible. While the Kernel is the innermost part of an operating system, a shell is the outermost one.
- A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output. It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.
- When you run the terminal, the Shell issues **a command prompt (usually $),** where you can type your input, which is then executed when you hit the Enter key. The output or the result is thereafter displayed on the terminal.
- The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name **Shell**.
- This Unix/Linux Shell Script tutorial helps understand shell scripting basics to advanced levels.

- **What is Shell?**
- A shell in a Linux operating system takes input from you in the form of commands, processes it, and then gives an output.
- It is the interface through which a user works on the programs, commands, and scripts. A shell is accessed by a terminal which runs it.
- When you run the terminal, the Shell issues **a command prompt (usually $),** where you can type your input, which is then executed when you hit the Enter key.
- The output or the result is thereafter displayed on the terminal.
- The Shell wraps around the delicate interior of an Operating system protecting it from accidental damage. Hence the name **Shell**.
- This Unix/Linux Shell Script tutorial helps understand shell scripting basics to advanced levels.

- **Shell Scripting**
- **Shell Scripting** is an open-source computer program designed to be run by the Unix/Linux shell.
- Shell Scripting is a program to write a series of commands for the shell to execute.
- It can combine lengthy and repetitive sequences of commands into a single and simple script that can be stored and executed anytime which, reduces programming efforts.

- **Types of Shell**
- There are two main shells in Linux:

1. The **Bourne Shell**: The prompt for this shell is $ and its derivatives are listed below:

- POSIX shell also is known as sh
- Korn Shell also knew as sh
- **B**ourne **A**gain **SH**ell also knew as bash (most popular)

2. The C shell: The prompt for this shell is %, and its subcategories are:

- C shell also is known as csh
- Tops C shell also is known as tcsh

- **How to Write Shell Script in Linux/Unix**
- **Shell Scripts** are written using text editors.
- On your Linux system, open a text editor program, open a new file to begin typing a shell script or shell programming, then give the shell permission to execute your shell script and put your script at the location from where the shell can find it.
- Let us understand the steps in creating a Shell Script:

1. **Create a file using** a **vi** editor(or any other editor). Name script file with **extension .sh**
2. **Start** the script with **#! /bin/sh**
3. Write some code.
4. Save the script file as filename.sh
5. For **exec uting** the script type **bash filename.sh**

- “#!” is an operator called shebang which directs the script to the interpreter location.
- So, if we use”#! /bin/sh” the script gets directed to the bourne-shell.
- Let’s create a small script –
- #!/bin/sh
- ls

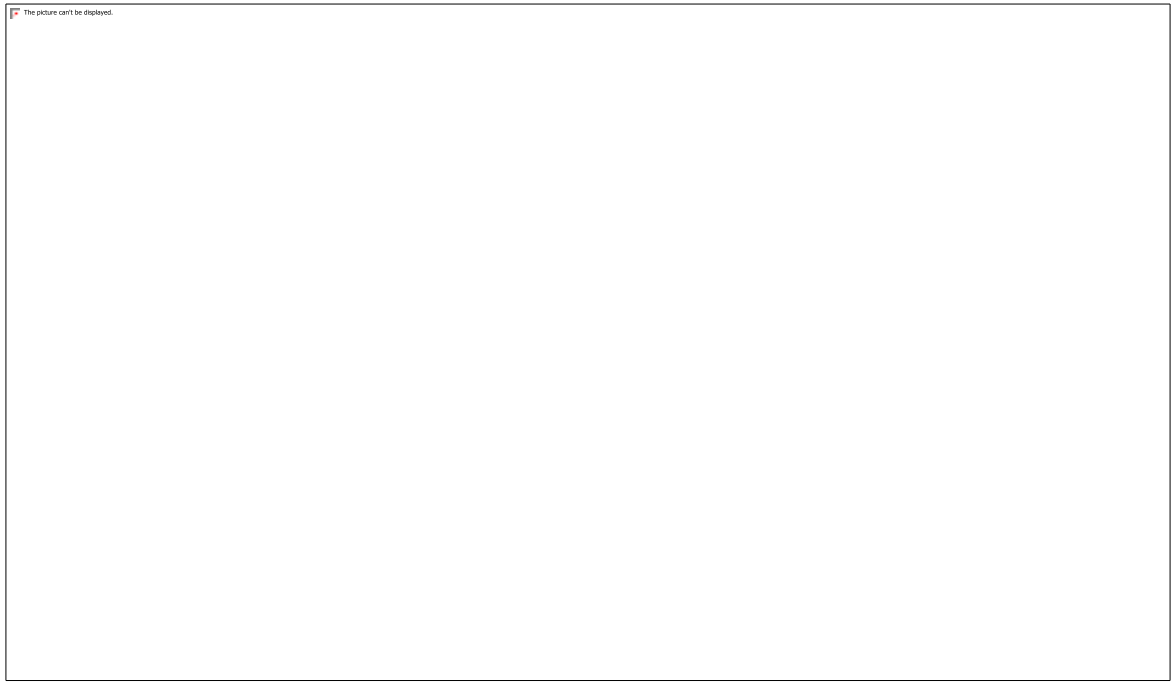
• Steps to Create Shell Script in Linux/Unix

Command ‘ls’ is executed when we execute the script sample.sh file.

Steps to Create Shell Script in Linux/Unix

- **Adding shell comments**
- Commenting is important in any program. In Shell programming, the syntax to add a comment is
- #comment

- Let understand this with an example.



- **echo** command in Linux is used to display line of text/string that are passed as an argument . This is a built in command that is mostly used in shell scripts and batch files to output status text to the screen or a file.
- echo [option] [string]
- **Displaying a text/string : Syntax :**
- echo [string]

- Defining Variables
- Variables are defined as follows −
- variable_name=variable_value
- For example −
- NAME=“VIIT“
- The above example defines the variable NAME and assigns the value “VIIT" to it.
- Variables of this type are called **scalar variables**.
- A scalar variable can hold only one value at a time.
- Shell enables you to store any value you want in a variable. For example −
- VAR1=“VIIT"
- VAR2=100

- Accessing Values
- To access the value stored in a variable, prefix its name with the dollar sign ($) −
- For example, the following script will access the value of defined variable NAME and print it on

- #!/bin/sh

- NAME=“VIIT"
- echo $NAME
- The above script will produce the following value −
- VIIT

- Shell Scripting Variables
- Scripts can contain variables inside the script.

- Look at the above snapshot, two variables are assigned to the script **$var1** and **$var2.**
- As scripts run in their own shell, hence variables do not survive the end of the script.

- Look at the above snapshot, two variables are assigned to the script **$var1** and **$var2.**
- As scripts run in their own shell, hence variables do not survive the end of the script.]

- Look at the above snapshot, **var1** and **var2** do not run outside the script.

- **Read-only Variables**
- Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.
- For example, the following script generates an error while trying to change the value of NAME −
- #!/bin/sh

- NAME=“VIIT"
- readonly NAME
- NAME=“CSE“
- The above script will generate the following result −
- /bin/sh: NAME: This variable is read only.

- **Variable Types**
- When a shell is running, three main types of variables are present −
- **Local Variables** − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

- Special Variables
- For example, the **$** character represents the process ID number, or PID, of the current shell −
- $echo $$
- The above command writes the PID of the current shell −
- 29949

- Shell supports a different type of variable called an **array variable**.
- This can hold multiple values at the same time.
- Arrays provide a method of grouping a set of variables.
- Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.
- All the naming rules discussed for Shell Variables would be applicable while naming arrays.
- **Defining Array Values**
- The difference between an array variable and a scalar variable can be explained as follows.
- Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows –

- NAME01="Zara"
- NAME02="Qadir"
- NAME03="Mahnaz"
- NAME04="Ayan"
- NAME05="Daisy
- We can use a single array to store all the above mentioned names.
- Following is the simplest method of creating an array variable.
- This helps assign a value to one of its indices.
- **array_name[index]=value**

- Here *array_name* is the name of the array, *index* is the index of the item in the array that you want to set, and value is the value you want to set for that item.
- As an example, the following commands −
- NAME[0]="Zara"
- NAME[1]="Qadir"
- NAME[2]="Mahnaz"
- NAME[3]="Ayan"
- NAME[4]="Daisy“
- If you are using the **ksh** shell, here is the syntax of array initialization −
- set -A array_name value1 value2 ... valuen

- If you are using the **bash** shell, here is the syntax of array initialization −
- array_name=(value 1 ... Value n)
- Accessing Array Values
- After you have set any array variable, you access it as follows −
- ${array_name[index]}
- Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept −

- #!/bin/sh

- NAME[0]="Zara"
- NAME[1]="Qadir"
- NAME[2]="Mahnaz"
- NAME[3]="Ayan"
- NAME[4]="Daisy"
- echo "First Index: ${NAME[0]}"
- echo "Second Index: ${NAME[1]}“

- The above example will generate the following result –
- $./test.sh
- First Index: Zara
- Second Index: Qadir

- You can access all the items in an array in one of the following ways −
- ${array_name[*]}
- ${array_name[@]}
- Here **array_name** is the name of the array you are interested in. Following example will help you understand the concept −

- #!/bin/sh

- NAME[0]=“VIIT"
- NAME[1]=“COMP"
- NAME[2]=“ENTC"
- NAME[3]=“CIVIL"
- NAME[4]=“IT"
- echo "First Method: ${NAME[*]}"
- echo "Second Method: ${NAME[@]}"

References

1) https://www.redhat.com/en/topics/data-storage/file-block-object-storage
2) https://www.guru99.com/introduction-to-shell-scripting.html
3) https://www.geeksforgeeks.org/echo-command-in-linux-with-examples/
4) https://www.tutorialspoint.com/execute_bash_online.php
5) https://www.tutorialspoint.com/unix/unix-using-arrays.htm

Firewalls

Presented By
Santosh Kumar

Outline

- Introduction
- Firewall Environments
- Type of Firewalls
- Future of Firewalls
- Conclusion

Introduction

- Firewalls control the flow of network traffic
- Firewalls have applicability in networks where there is no internet connectivity
- Firewalls operate on number of layers
- Can also act as VPN gateways
- Active content filtering technologies

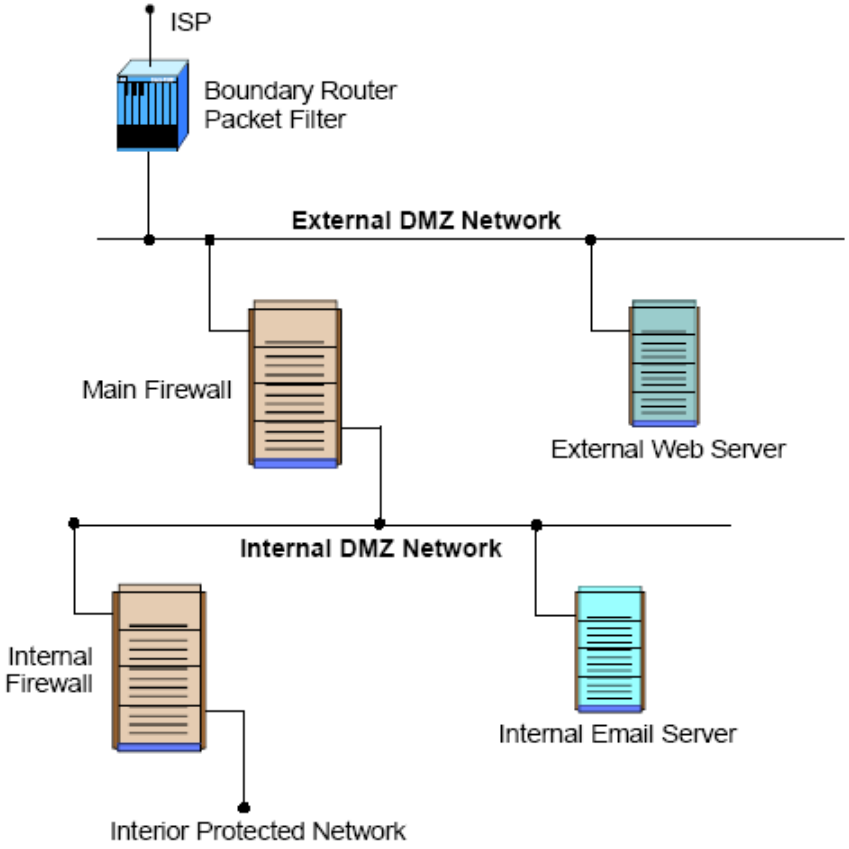
Firewall Environments

- There are different types of environments where a firewall can be implemented.
- Simple environment can be a packet filter firewall
- Complex environments can be several firewalls and proxies

DMZ Environment

- Can be created out of a network connecting two firewalls
- Boundary router filter packets protecting server
- First firewall provide access control and protection from server if they are hacked

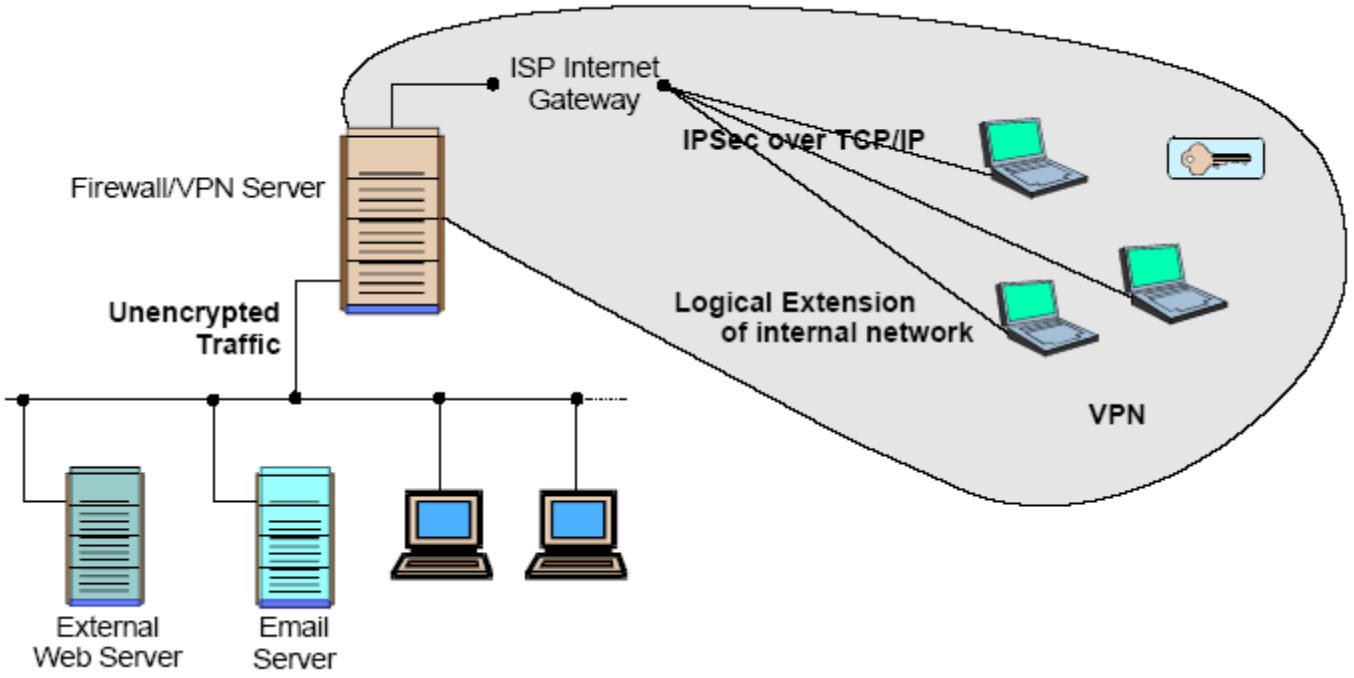
DMZ ENV


ISP
Boundary Router
Packet Filter
External DMZ Network
Main Firewall
External Web Server
Internal DMZ Network
Internal
Firewall
Internal Email Server
Interior Protected Network

VPN

- VPN is used to provide secure network links across networks
- VPN is constructed on top of existing network media and protocols
- On protocol level IPsec is the first choice
- Other protocols are PPTP, L2TP

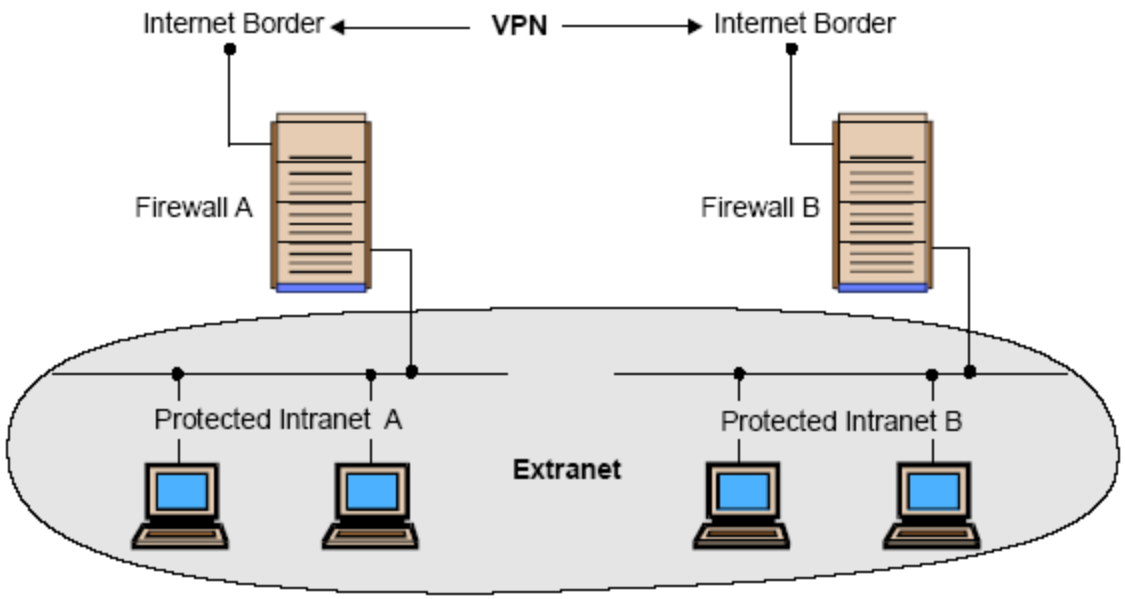
VPN


ISP Internet
Gateway
IPSec over TCP/IP
Firewall/VPN Server
Unencrypted
Traffic
Logical Extension
of internal network
VPN
External
Web Server
Email
Server

Intranets

- An intranet is a network that employs the same types of services, applications, and protocols present in an Internet implementation, without involving external connectivity
- Intranets are typically implemented behind firewall environments.

Intranets


Internet Border
VPN
Internet Border
Firewall A
Firewall B
Protected Intranet A
Protected Intranet B
Extranet

Extranets

- Extranet is usually a business-to-business intranet
- Controlled access to remote users via some form of authentication and encryption such as provided by a VPN
- Extranets employ TCP/IP protocols, along with the same standard applications and services

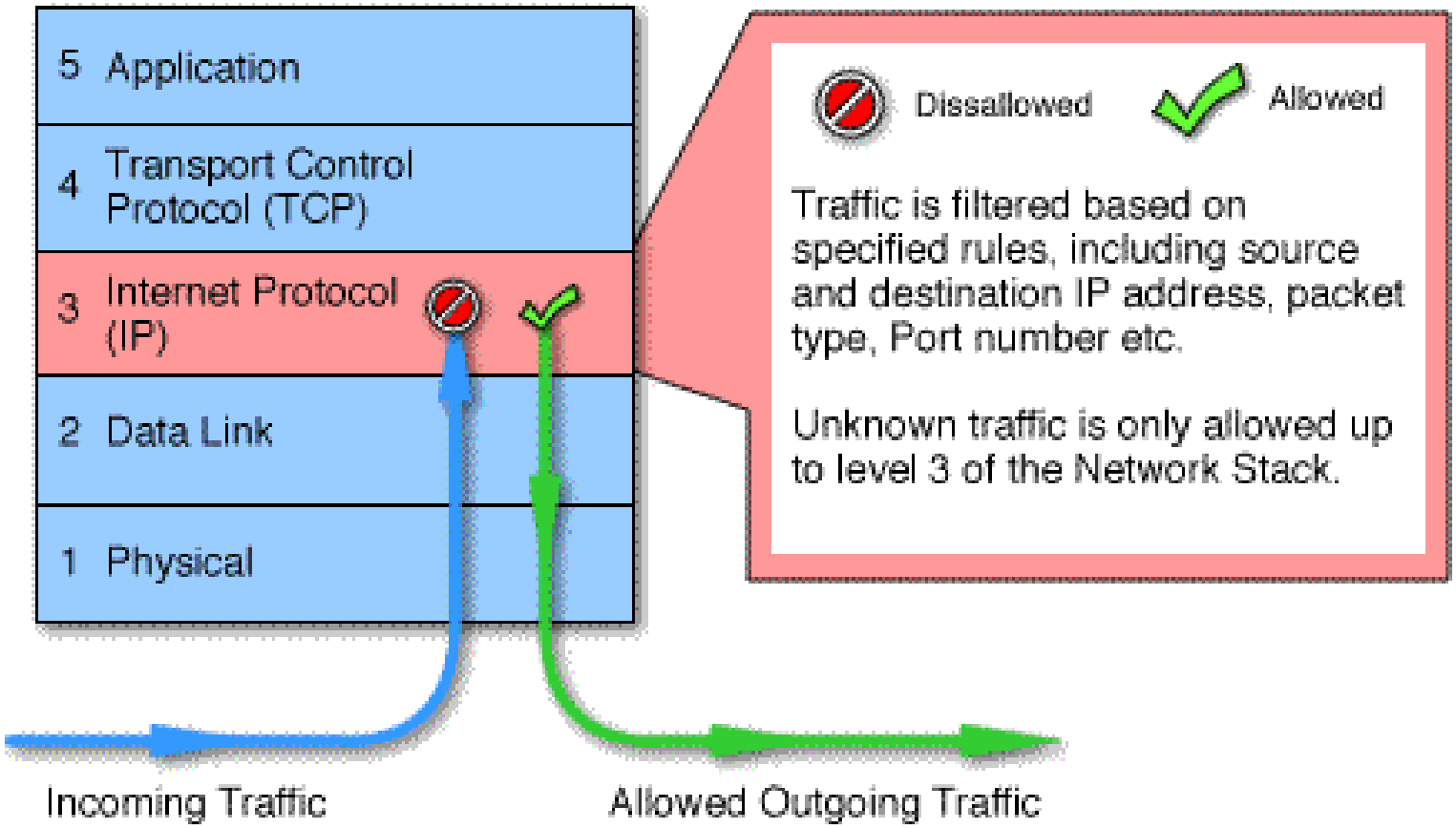
Type is Firewalls

- Firewalls fall into four broad categories
- Packet filters
- Circuit level
- Application level
- Stateful multilayer

Packet Filter

- Work at the network level of the OSI model
- Each packet is compared to a set of criteria before it is forwarded
- Packet filtering firewalls is low cost and low impact on network performance

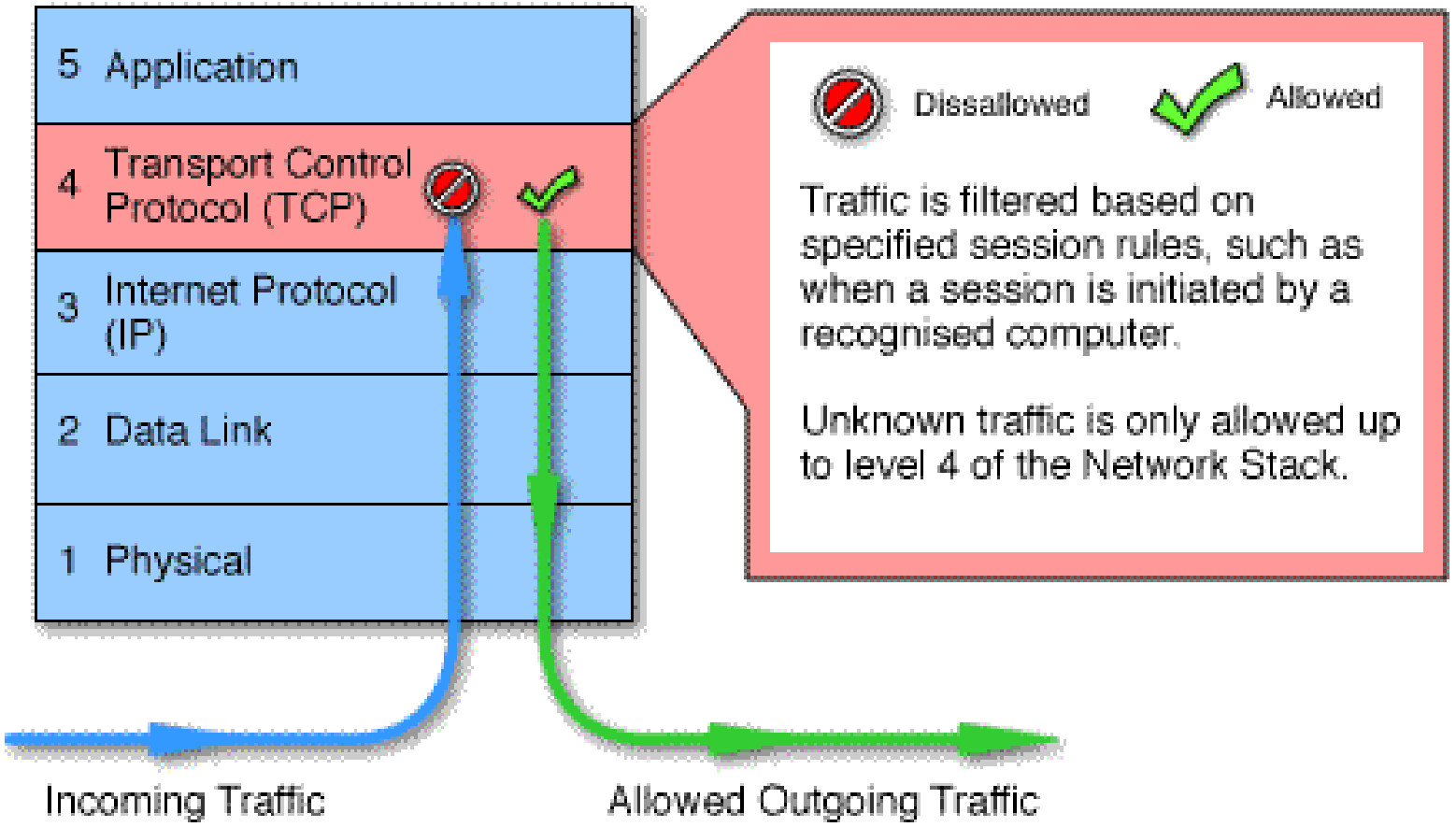
Packet Filtering



Circuit level

- Circuit level gateways work at the session layer of the OSI model, or the TCP layer of TCP/IP
- Monitor TCP handshaking between packets to determine whether a requested session is legitimate.

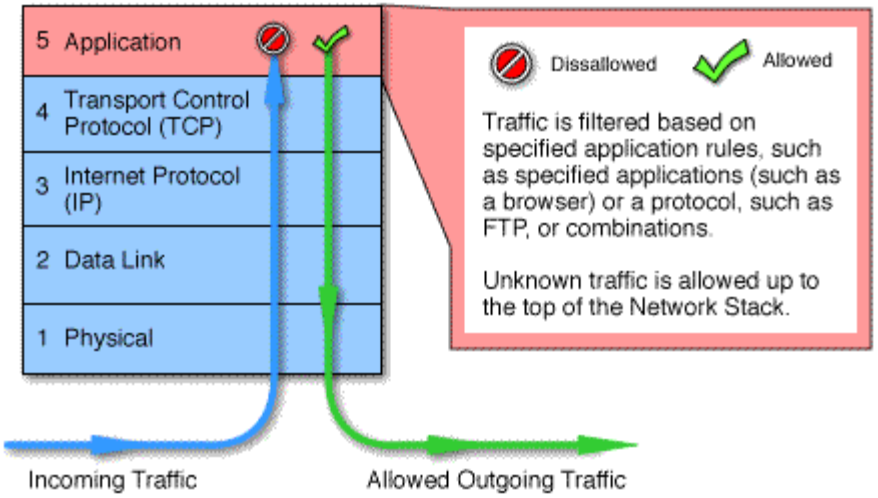
Circuit Level



Application Level

- Application level gateways, also called proxies, are similar to circuit-level gateways except that they are application specific
- Gateway that is configured to be a web proxy will not allow any ftp, gopher, telnet or other traffic through

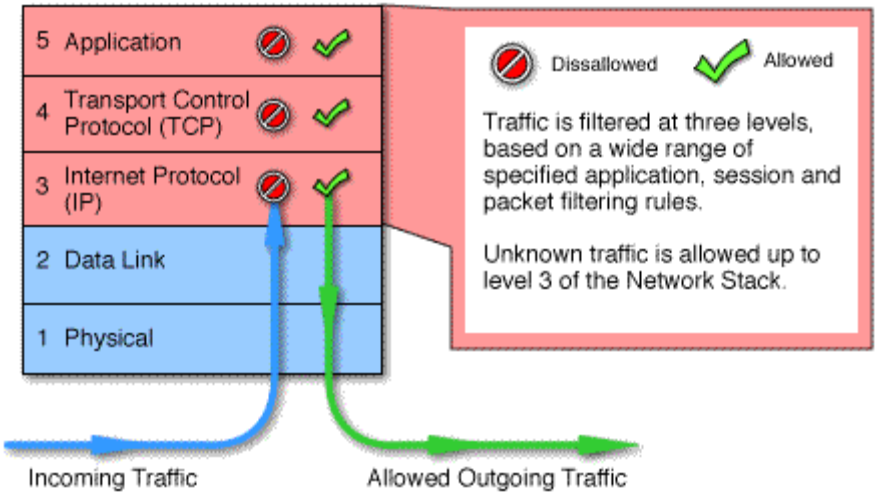
Application Level



Stateful Multilayer

- Stateful multilayer inspection firewalls combine the aspects of the other three types of firewalls
- They filter packets at the network layer, determine whether session packets are legitimate and evaluate contents of packets at the application layer

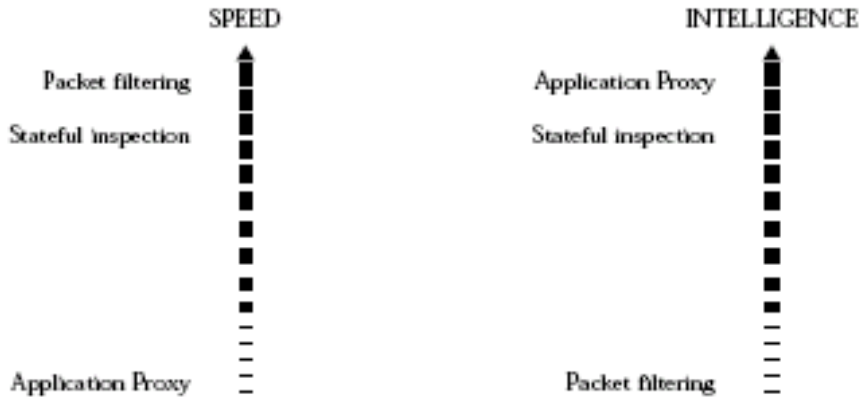
Stateful Multilayer



General Performance

FIREWALL PERFORMANCE SUMMARY

Technology	Speed	Flexibility	Intelligence
Packet filtering	V. Good	V.Good	Low
Application Proxy	Low	Low	V. Good
Stateful inspection	Good	Good	Good
Circuit gateway	Low	Low	Low



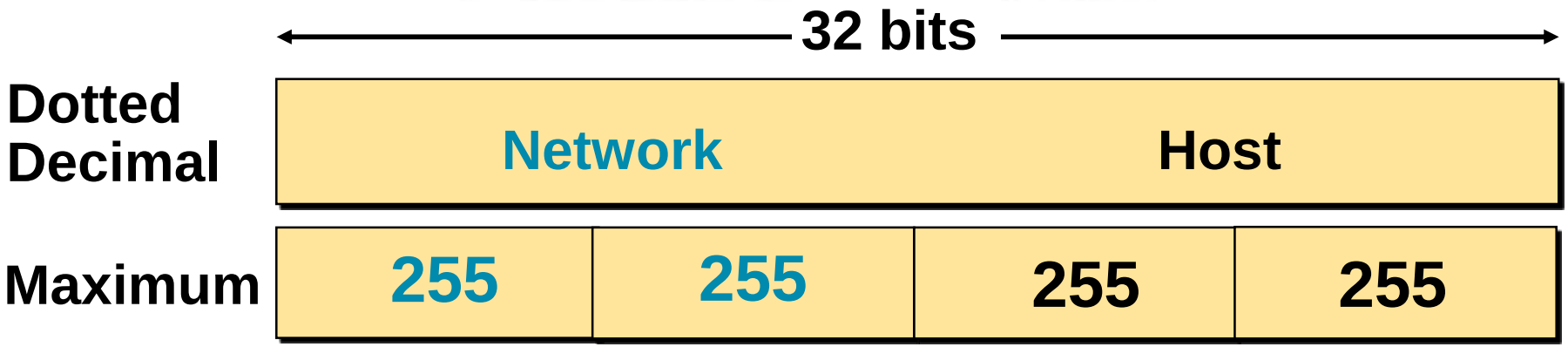
Future of Firewalls

- Firewalls will continue to advance as the attacks on IT infrastructure become more and more sophisticated
- More and more client and server applications are coming with native support for proxied environments
- Firewalls that scan for viruses as they enter the network and several firms are currently exploring this idea, but it is not yet in wide use

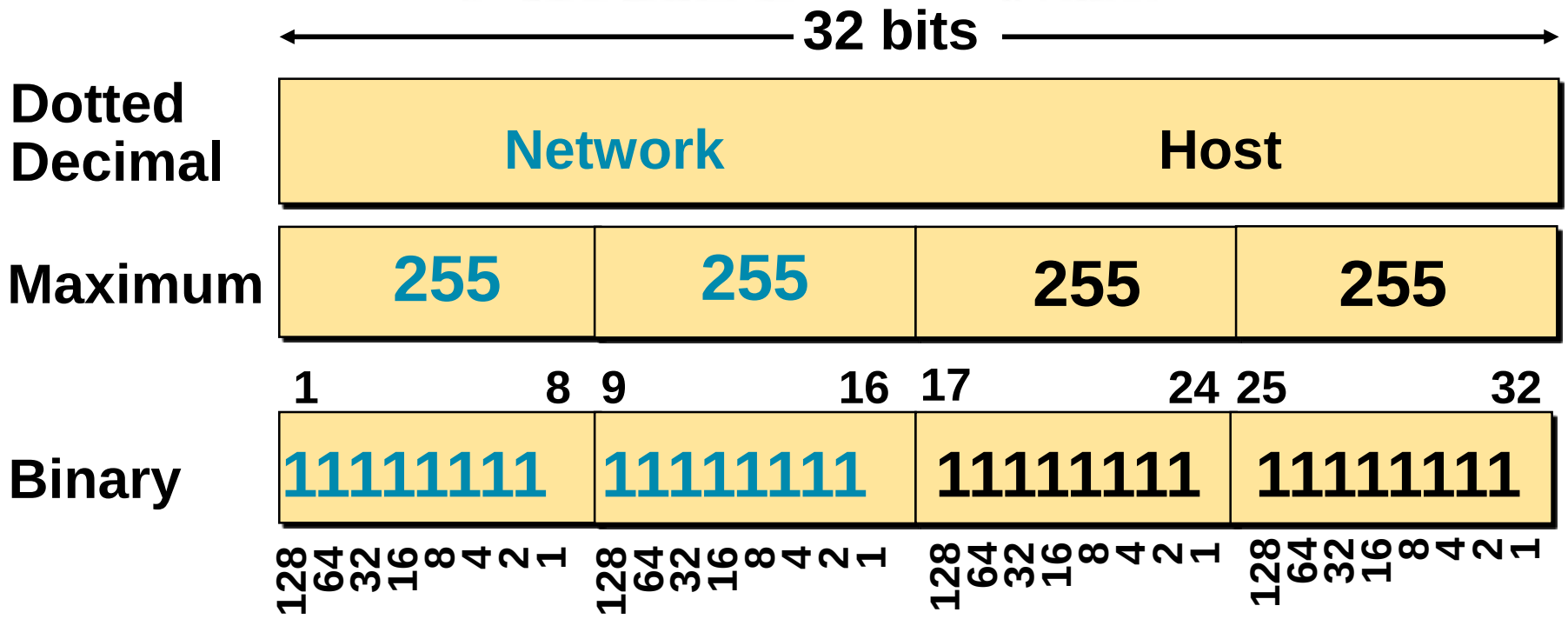
Conclusion

- It is clear that some form of security for private networks connected to the Internet is essential
- A firewall is an important and necessary part of that security, but cannot be expected to perform all the required security functions.

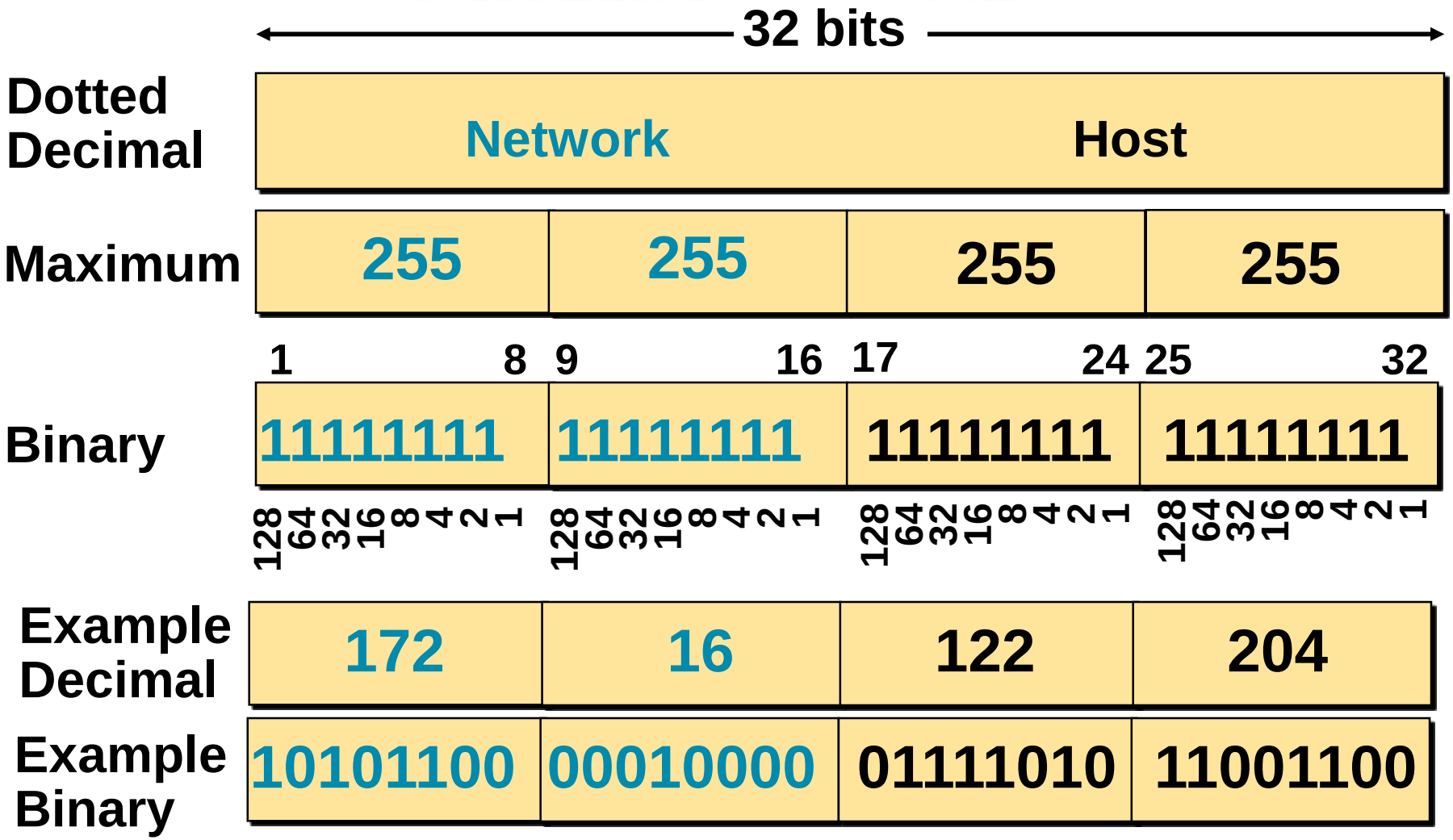
IP Addressing



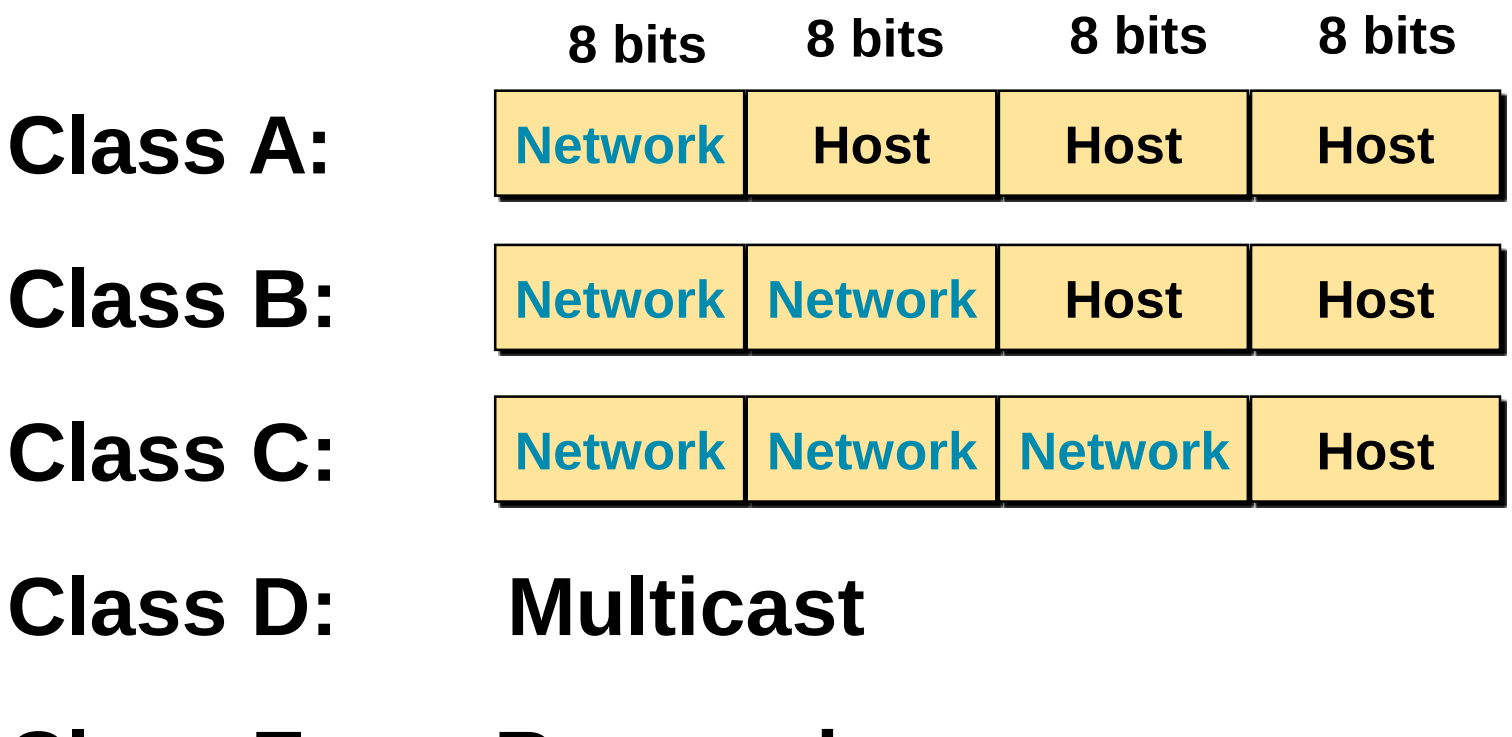
IP Addressing



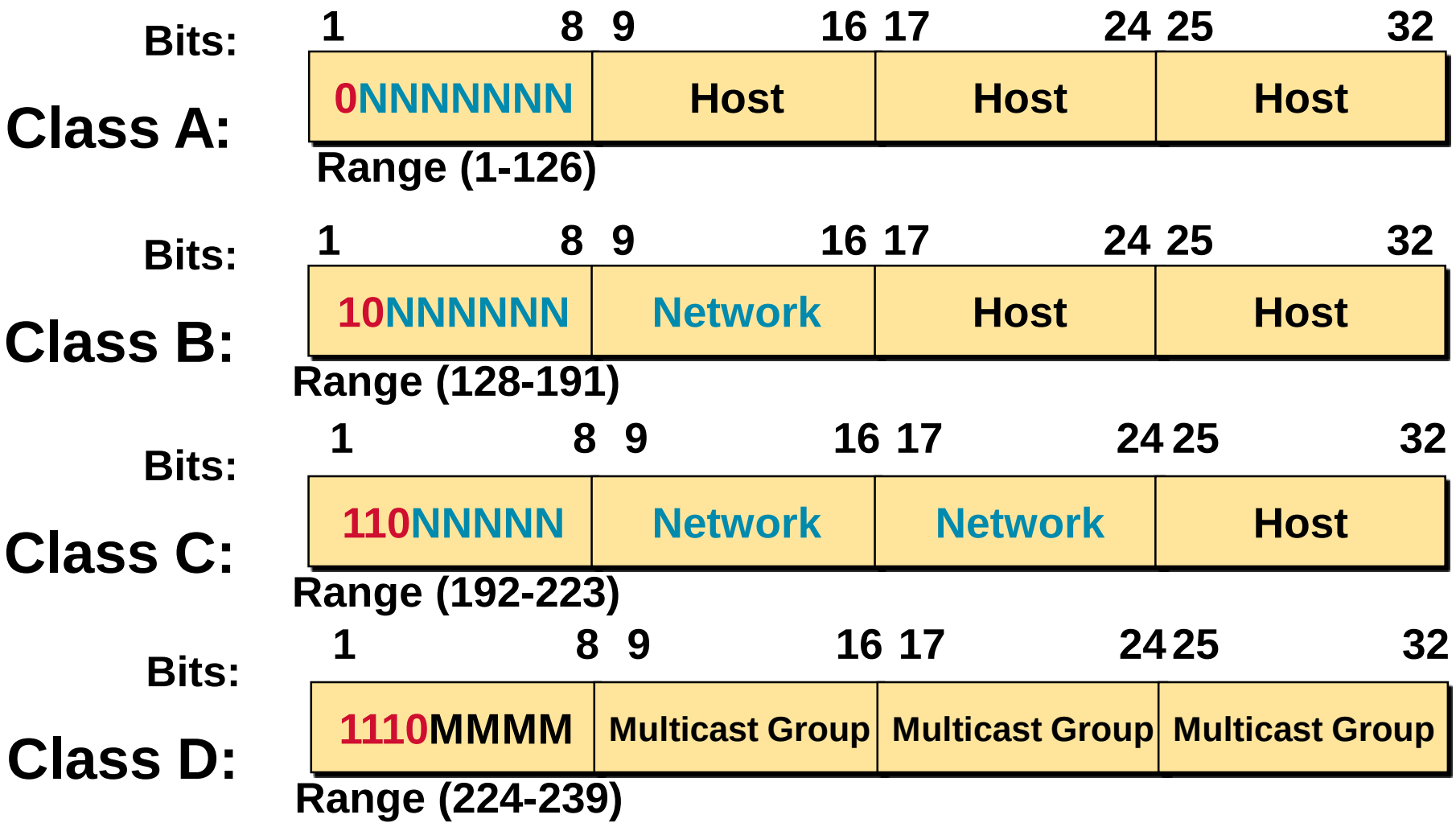
IP Addressing



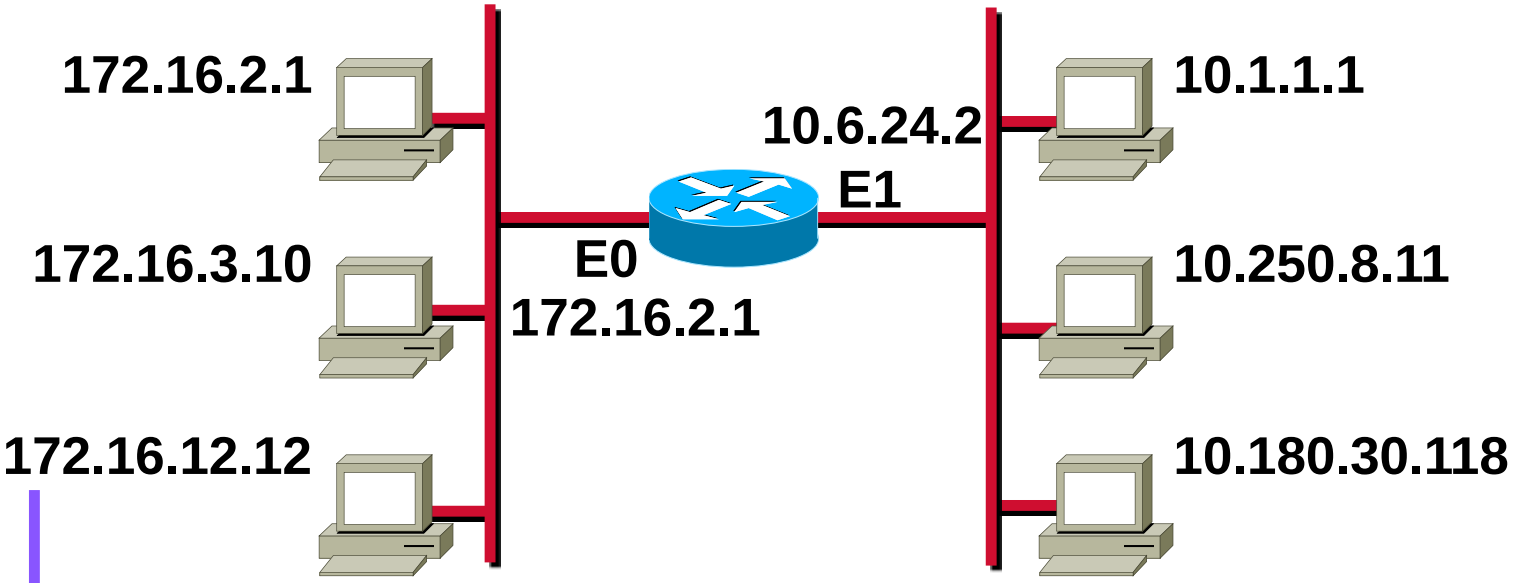
IP Address Classes



IP Address Classes



Host Addresses

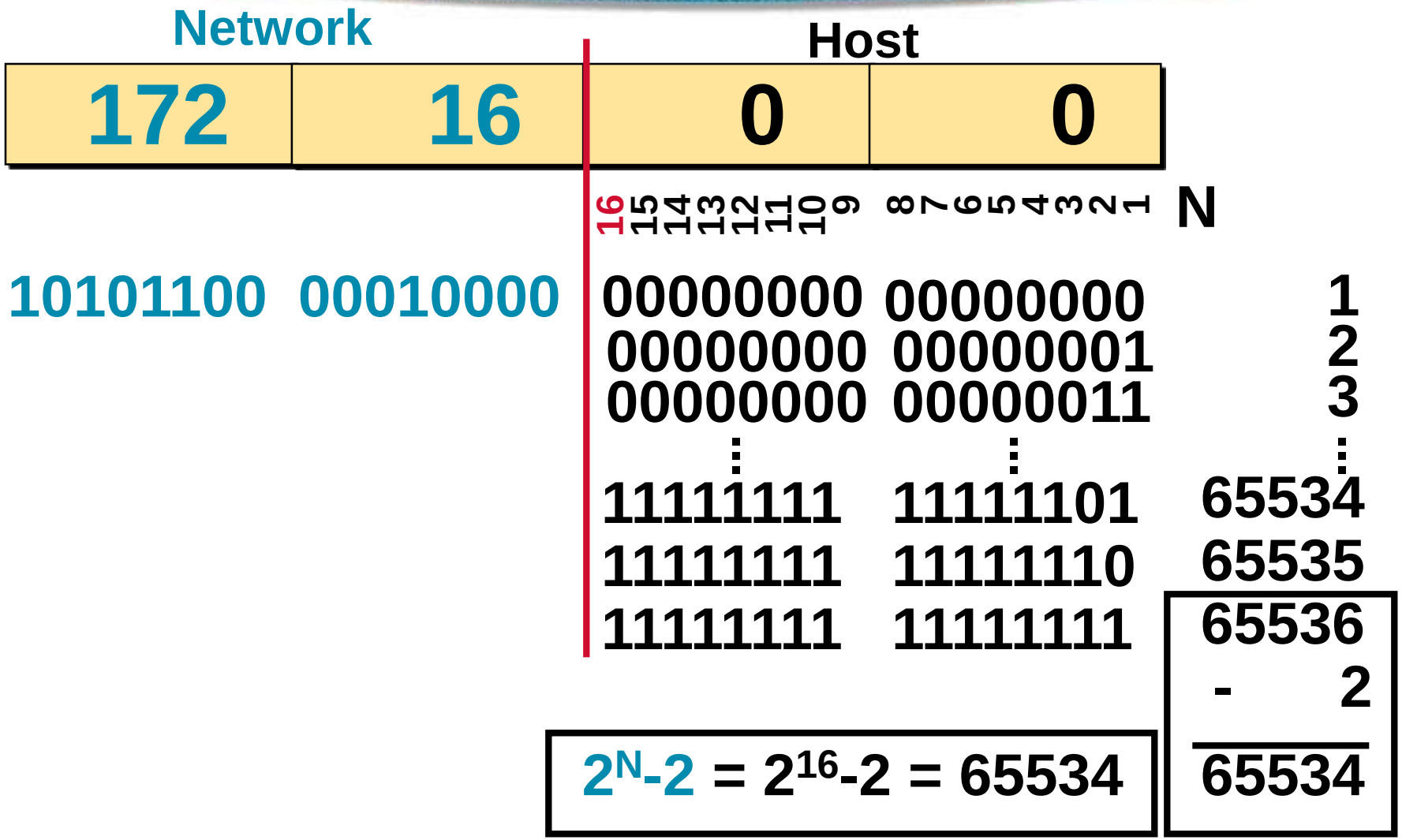


172.16 . 12 . 12

Network Host

Routing Table	
Network	Interface
172.16.0.0	E0
10.0.0.0	E1

Determining Available Host Addresses



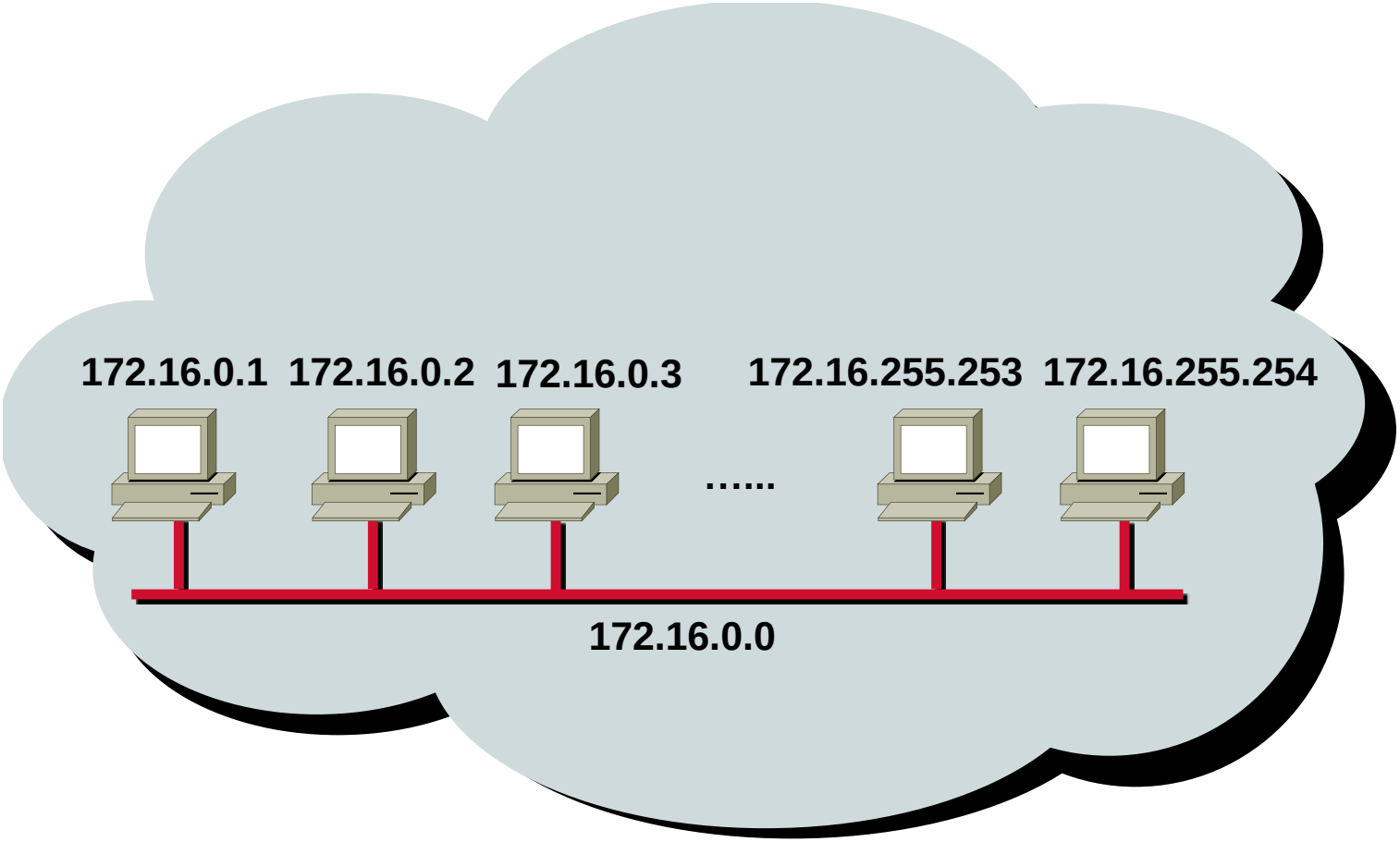
IP Address Classes Exercise

Address	Class	Network	Host
10.2.1.1			
128.63.2.100			
201.222.5.64			
192.6.141.2			
130.113.64.16			
256.241.201.10			

IP Address Classes Exercise Answers

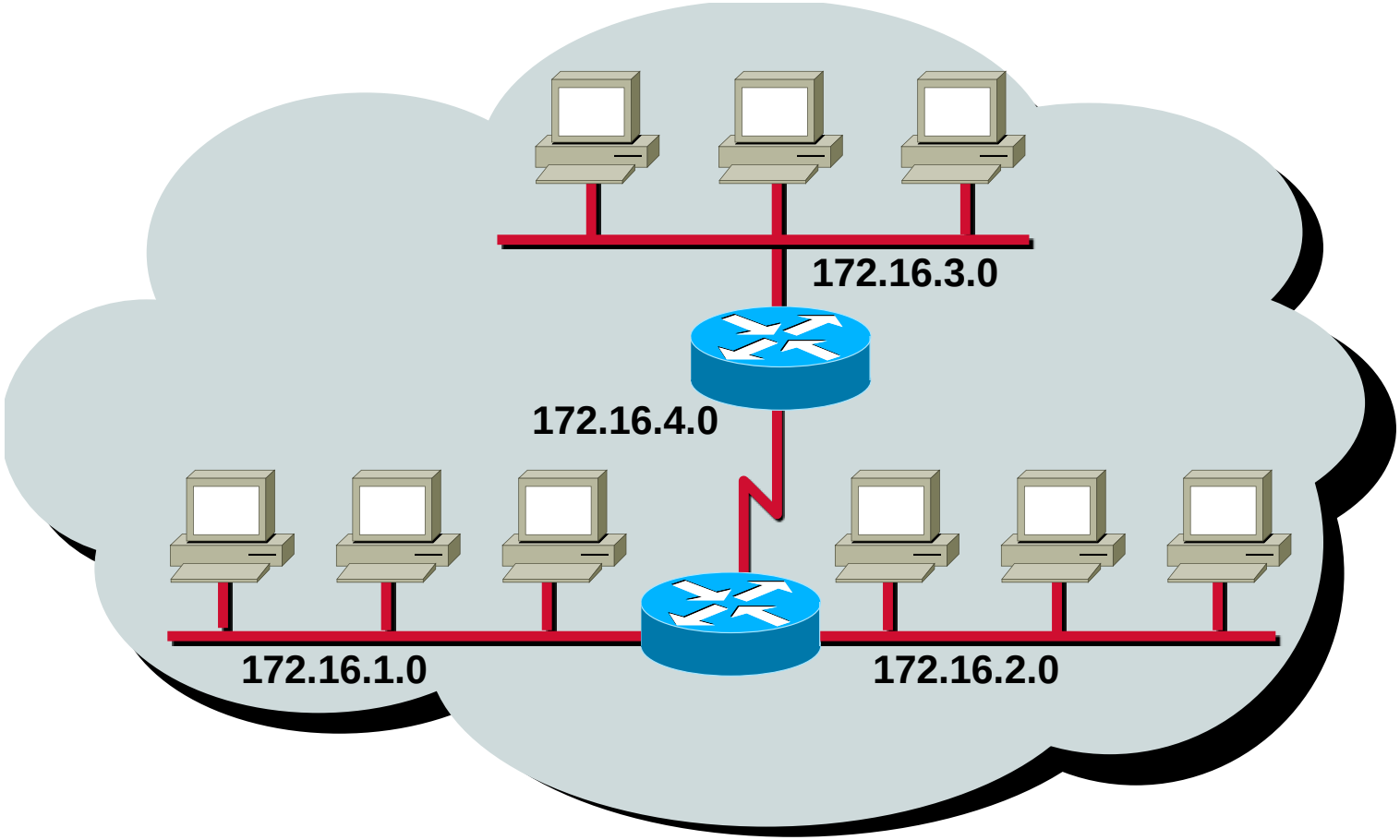
Address	Class	Network	Host
10.2.1.1	A	10.0.0.0	0.2.1.1
128.63.2.100	B	128.63.0.0	0.0.2.100
201.222.5.64	C	201.222.5.0	0.0.0.64
192.6.141.2	C	192.6.141.0	0.0.0.2
130.113.64.16	B	130.113.0.0	0.0.64.16
256.241.201.10	Nonexistent		

Addressing without Subnets



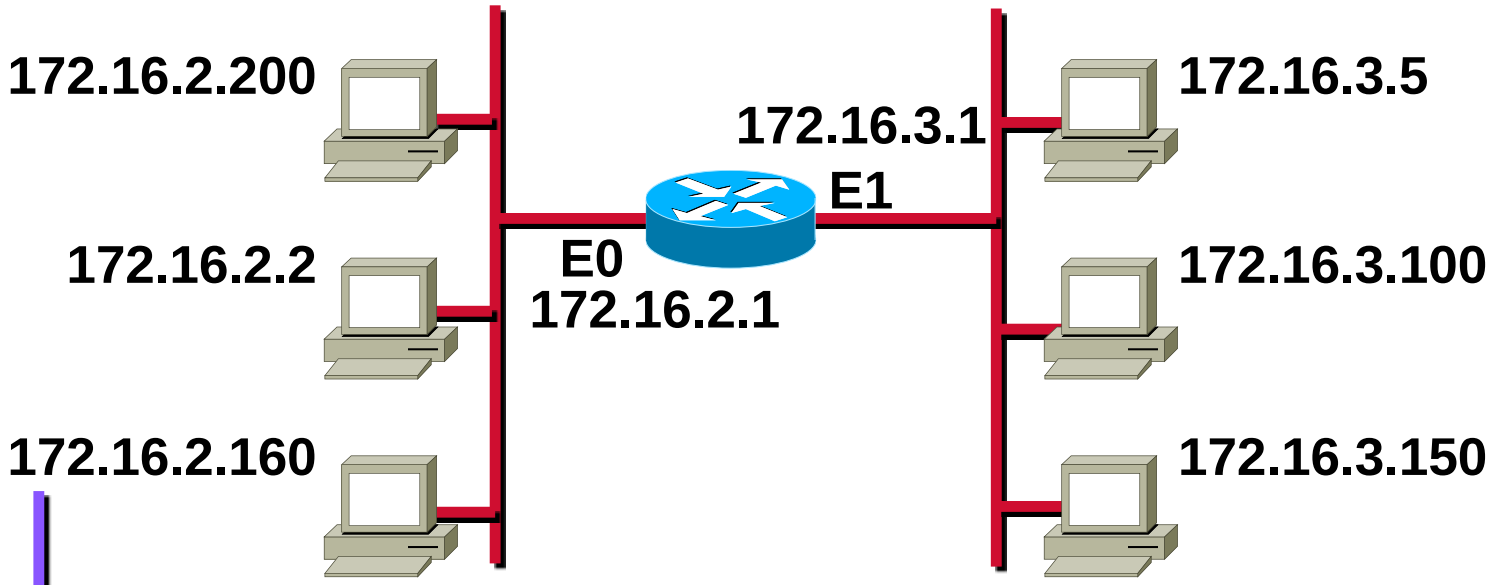
Network 172.16.0.0

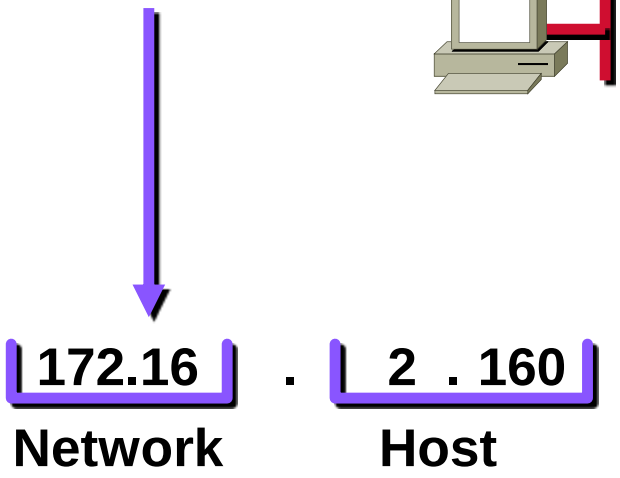
Addressing with Subnets



Network 172.16.0.0

Subnet Addressing

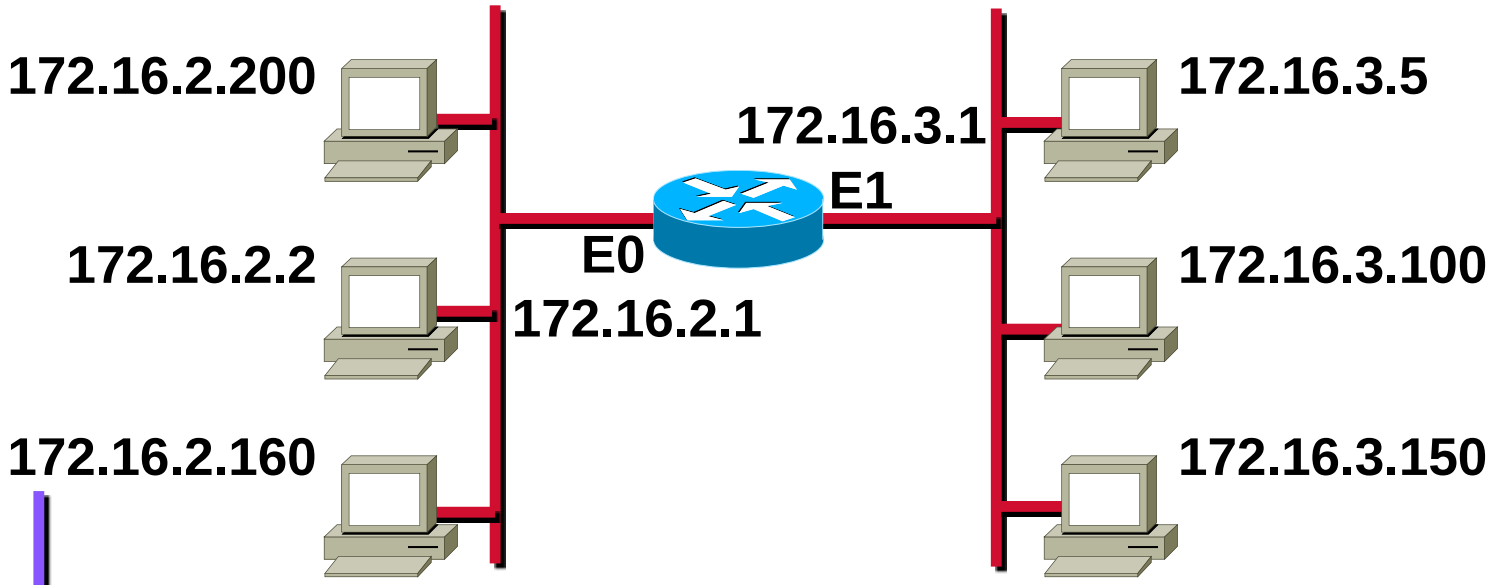


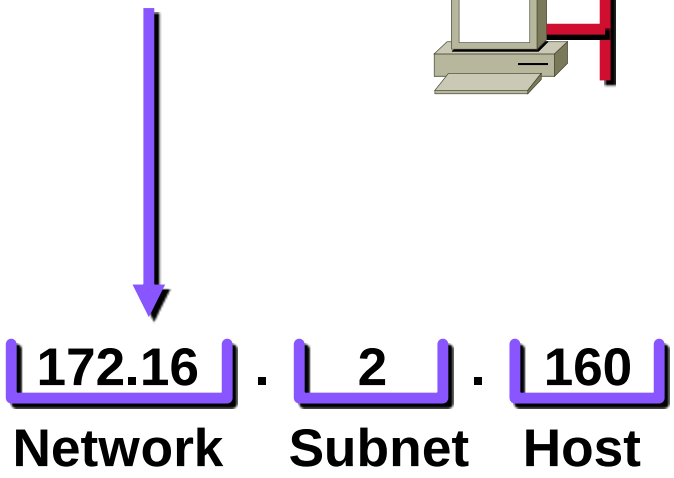


New Routing Table

Network	Interface
172.16.0.0	E0
172.16.0.0	E1

Subnet Addressing

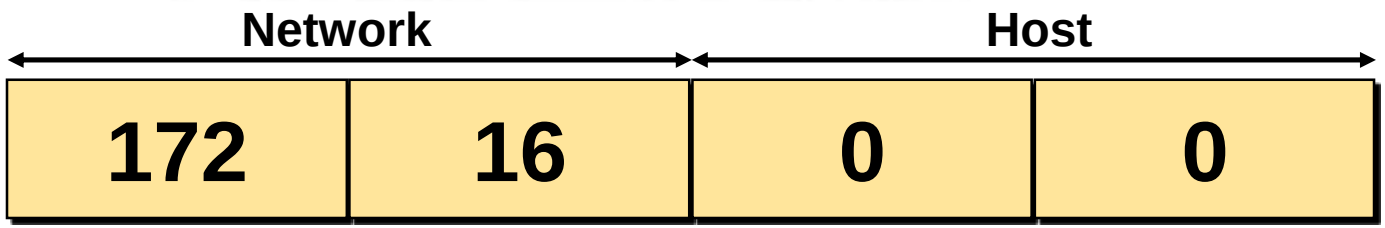


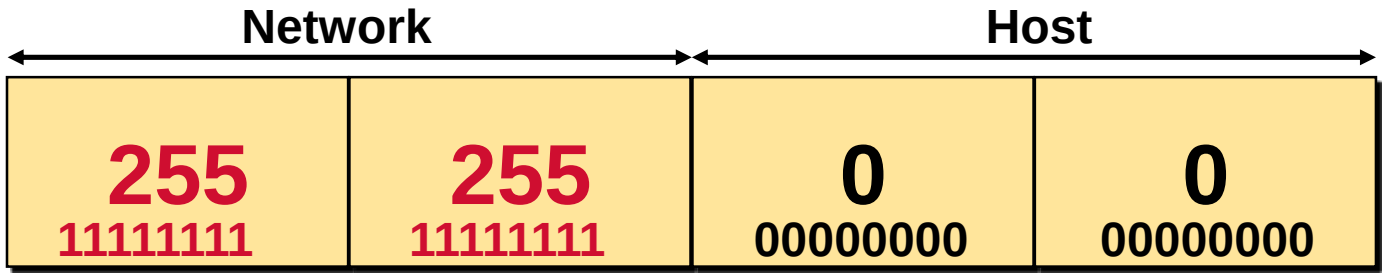


New Routing Table

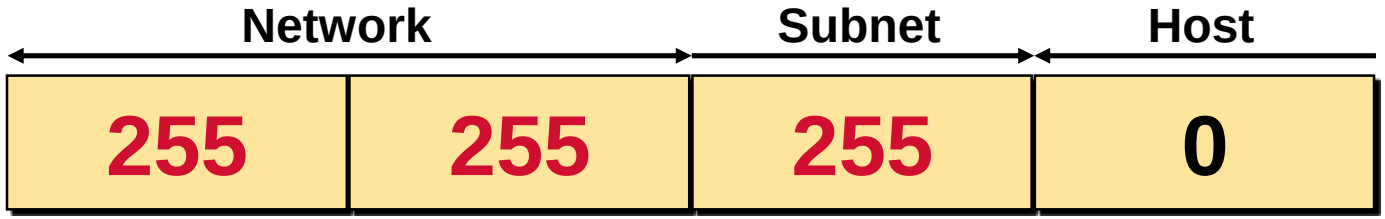
Network	Interface
172.16.2.0	E0
172.16.3.0	E1

Subnet Mask



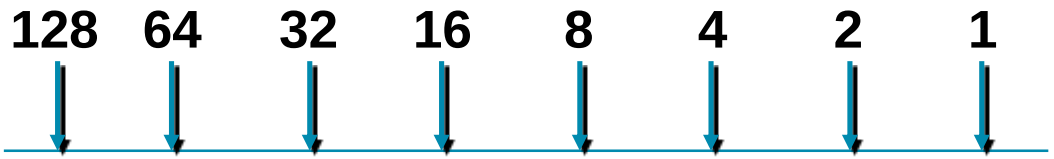


Also written as "/16" where 16 represents the number of 1s in the mask.



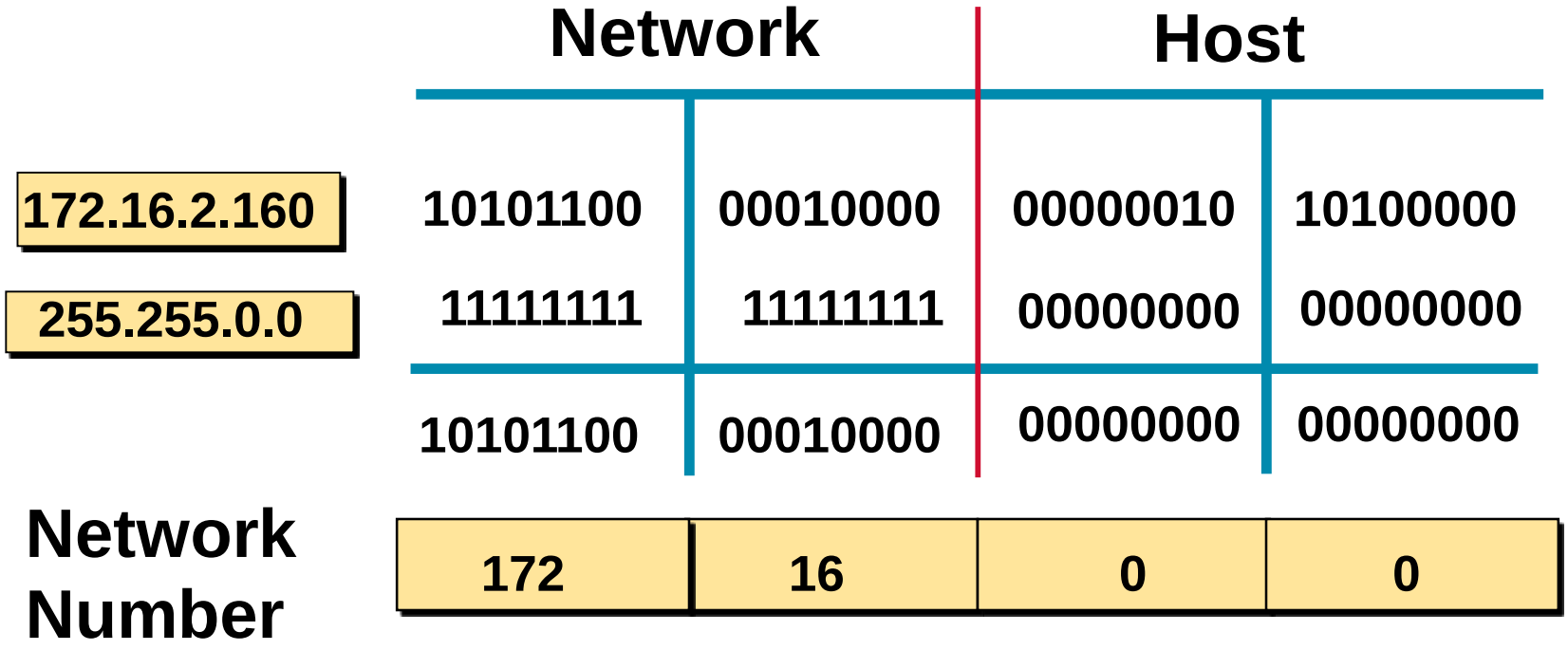
Also written as "/24" where 24 represents the number of 1s in the mask.

Decimal Equivalents of Bit Patterns



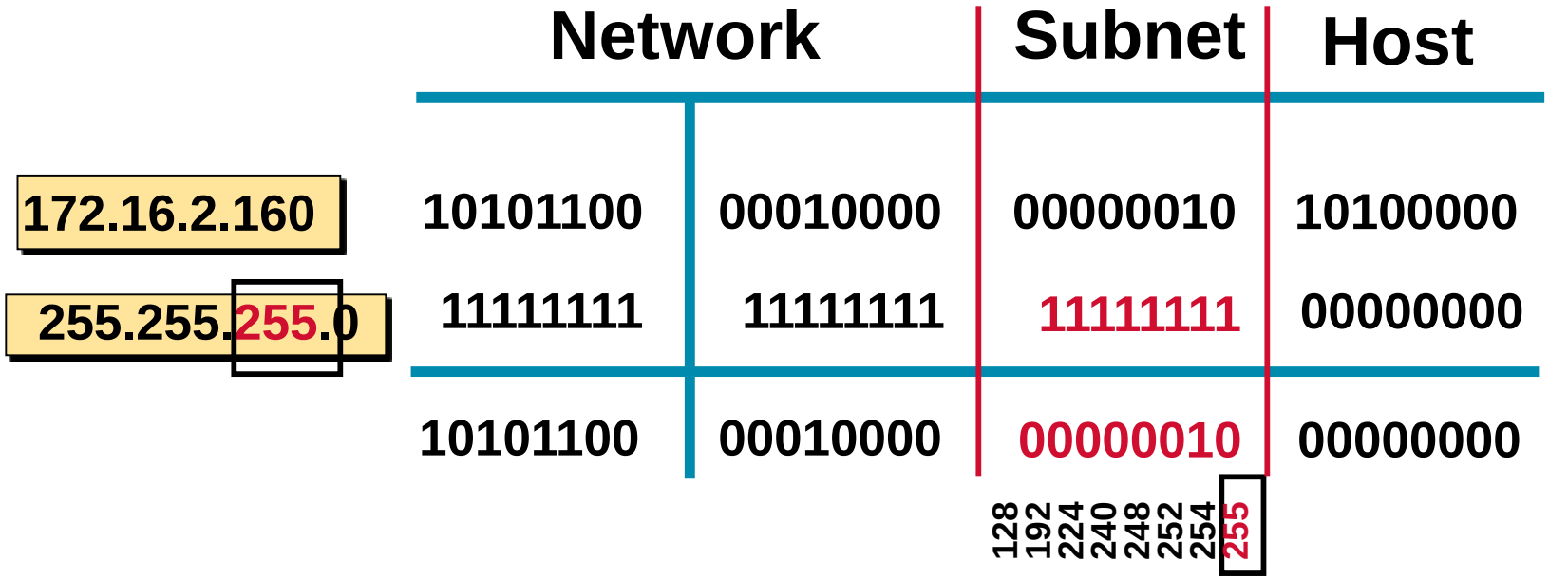
128	64	32	16	8	4	2	1		
1	0	0	0	0	0	0	0	=	128
1	1	0	0	0	0	0	0	=	192
1	1	1	0	0	0	0	0	=	224
1	1	1	1	0	0	0	0	=	240
1	1	1	1	1	0	0	0	=	248
1	1	1	1	1	1	0	0	=	252
1	1	1	1	1	1	1	0	=	254
1	1	1	1	1	1	1	1	=	255

Subnet Mask without Subnets



Subnets not in use—the default

Subnet Mask with Subnets

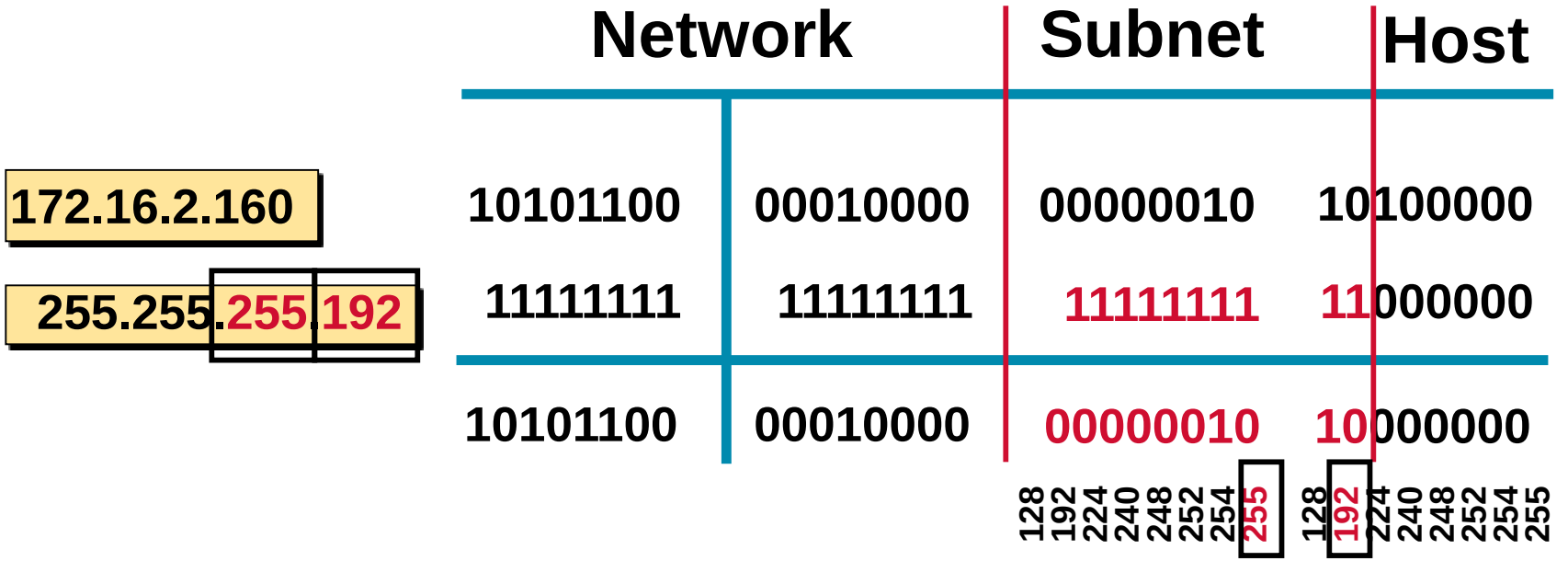


Network Number

172	16	2	0

Network number extended by eight bits

Subnet Mask with Subnets (cont.)



Network Number

172	16	2	128

Network number extended by ten bits

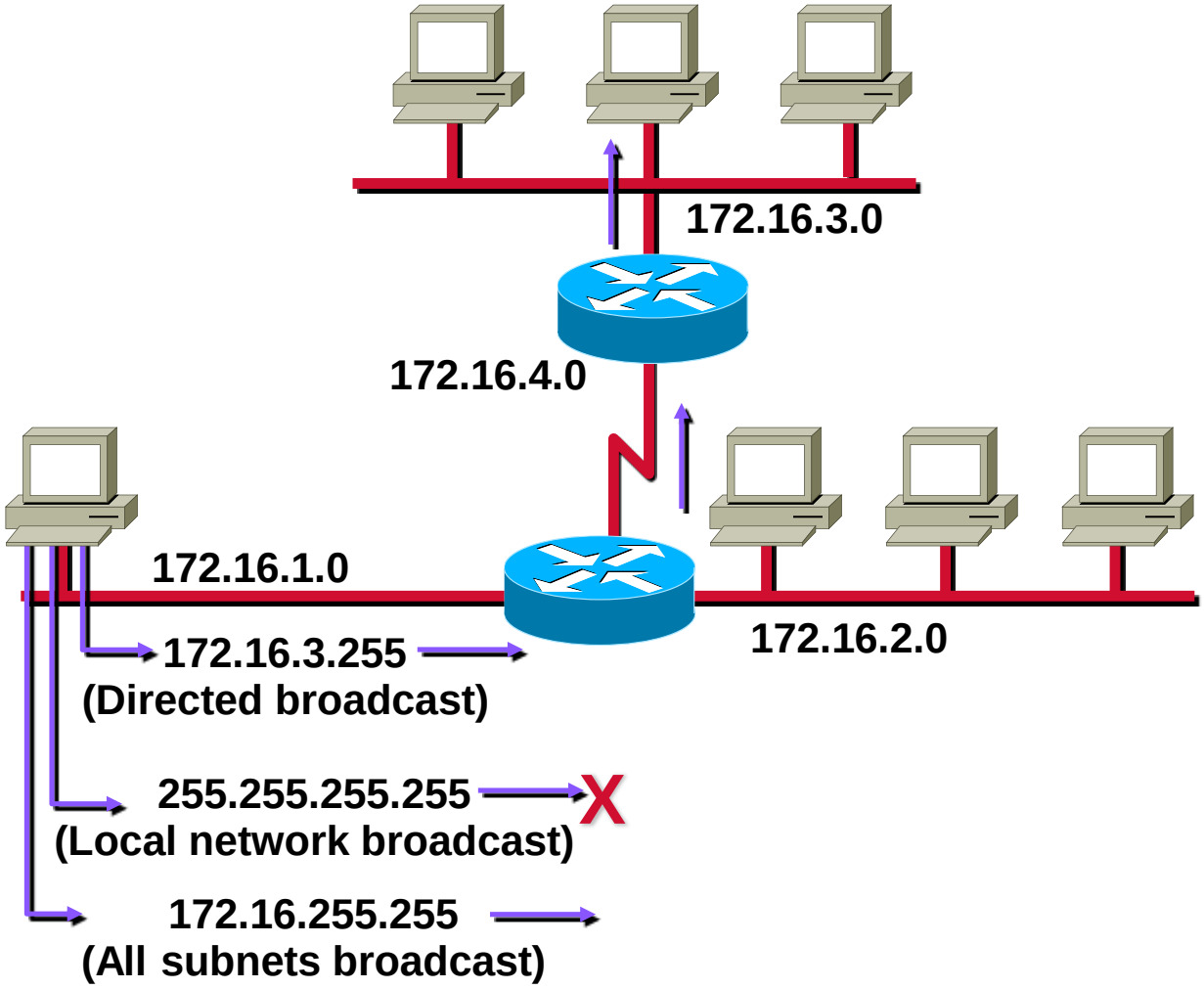
Subnet Mask Exercise

Address	Subnet Mask	Class	Subnet
172.16.2.10	255.255.255.0		
10.6.24.20	255.255.240.0		
10.30.36.12	255.255.255.0		

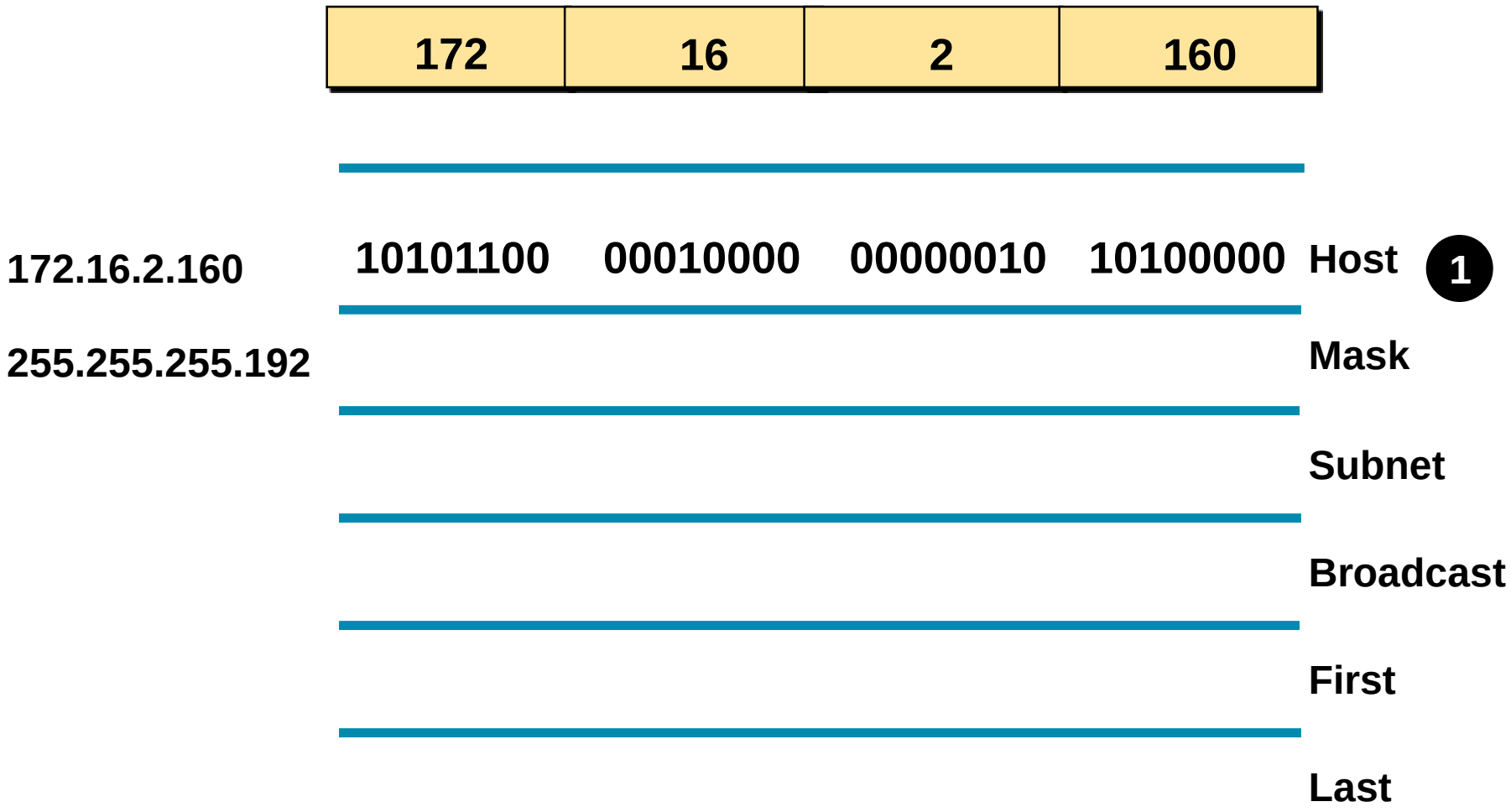
Subnet Mask Exercise Answers

Address	Subnet Mask	Class	Subnet
172.16.2.10	255.255.255.0	B	172.16.2.0
10.6.24.20	255.255.240.0	A	10.6.16.0
10.30.36.12	255.255.255.0	A	10.30.36.0

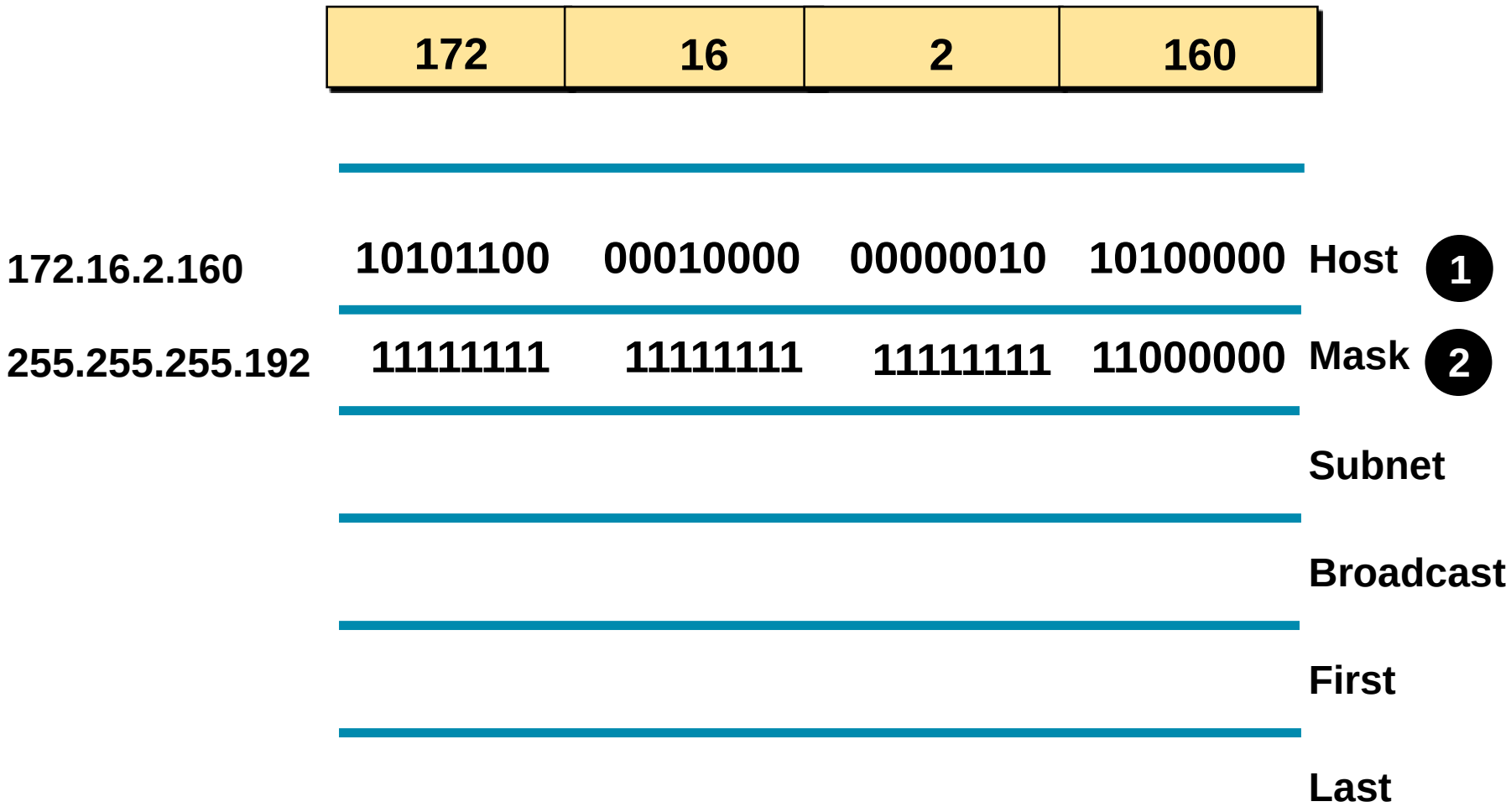
Broadcast Addresses



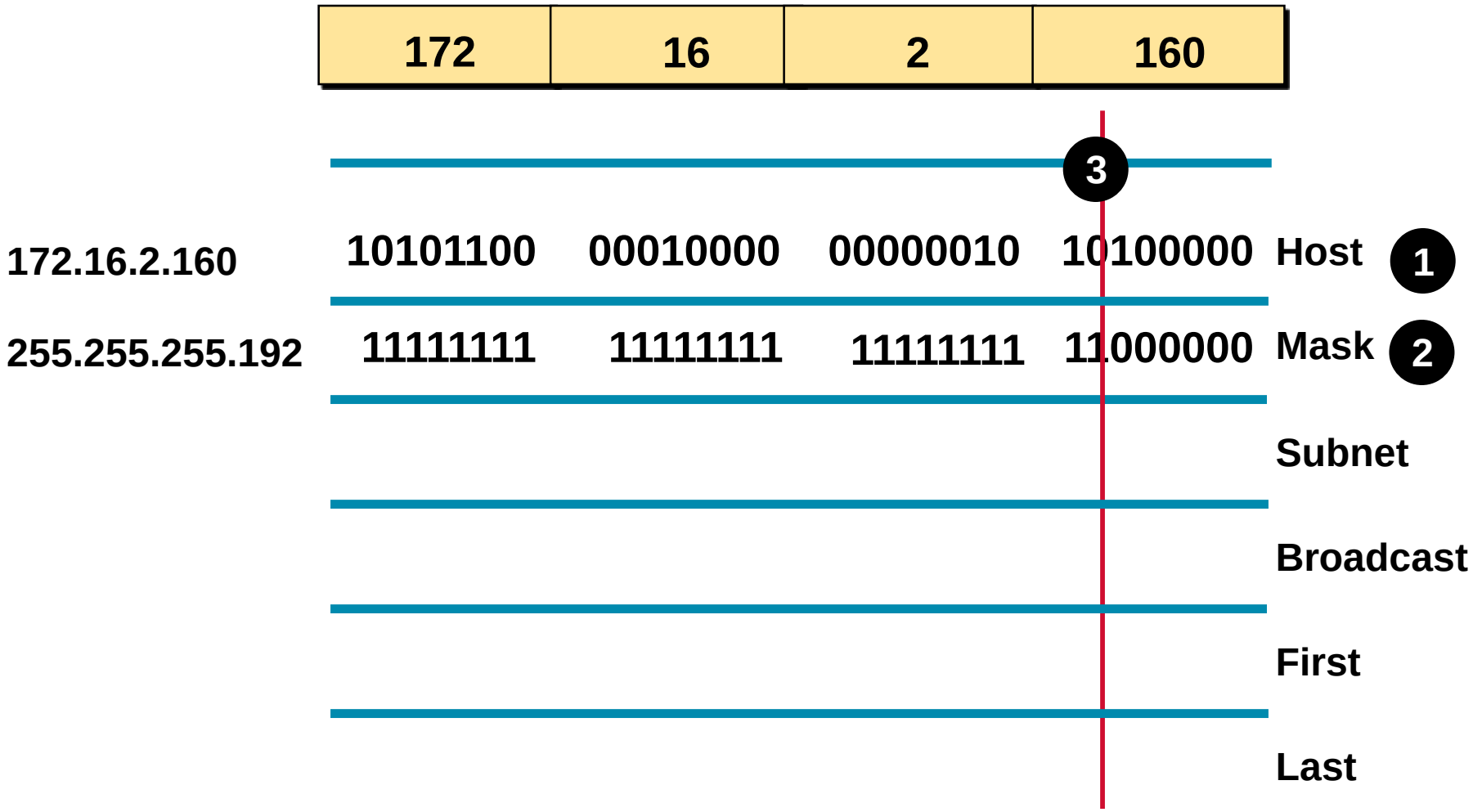
Addressing Summary Example



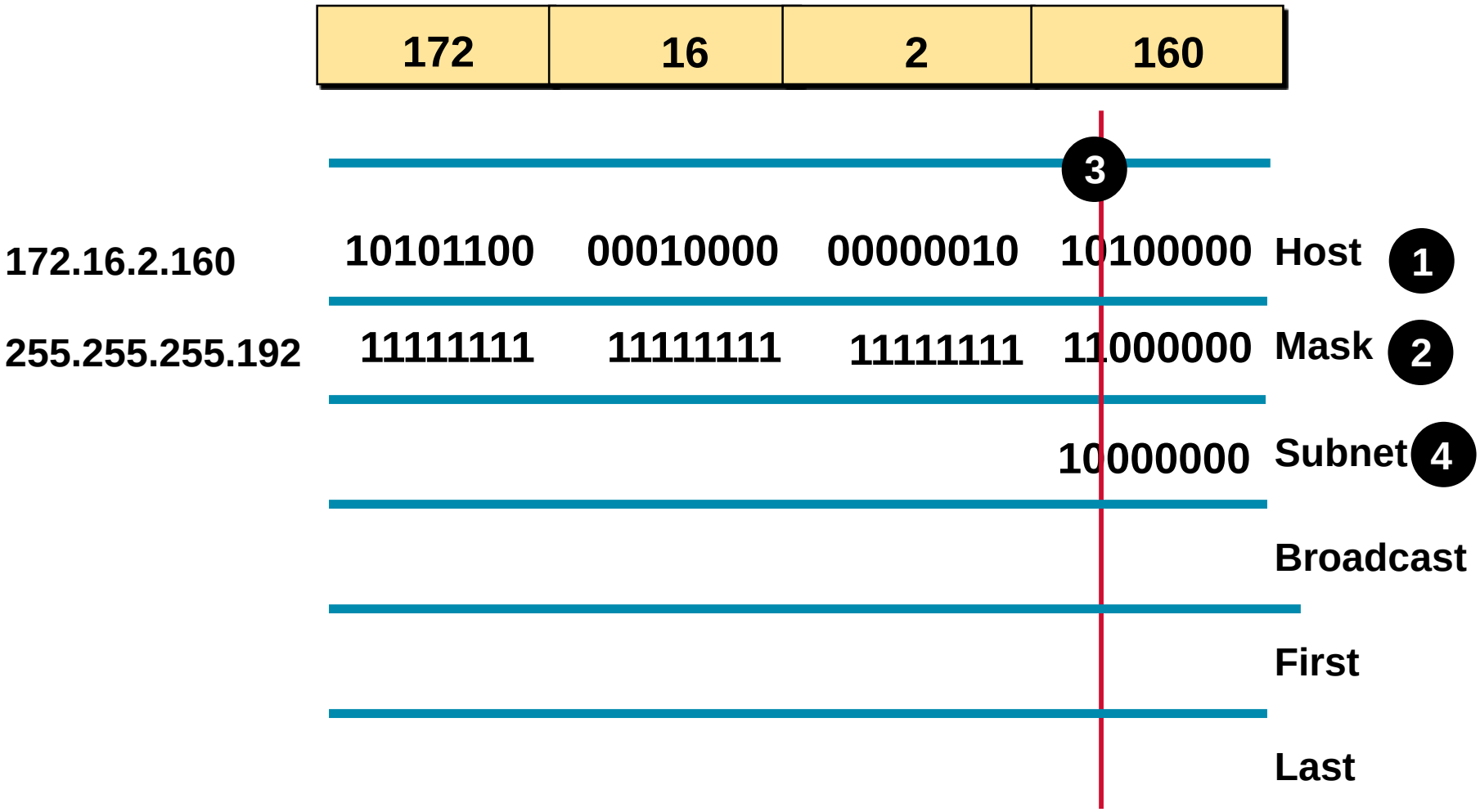
Addressing Summary Example



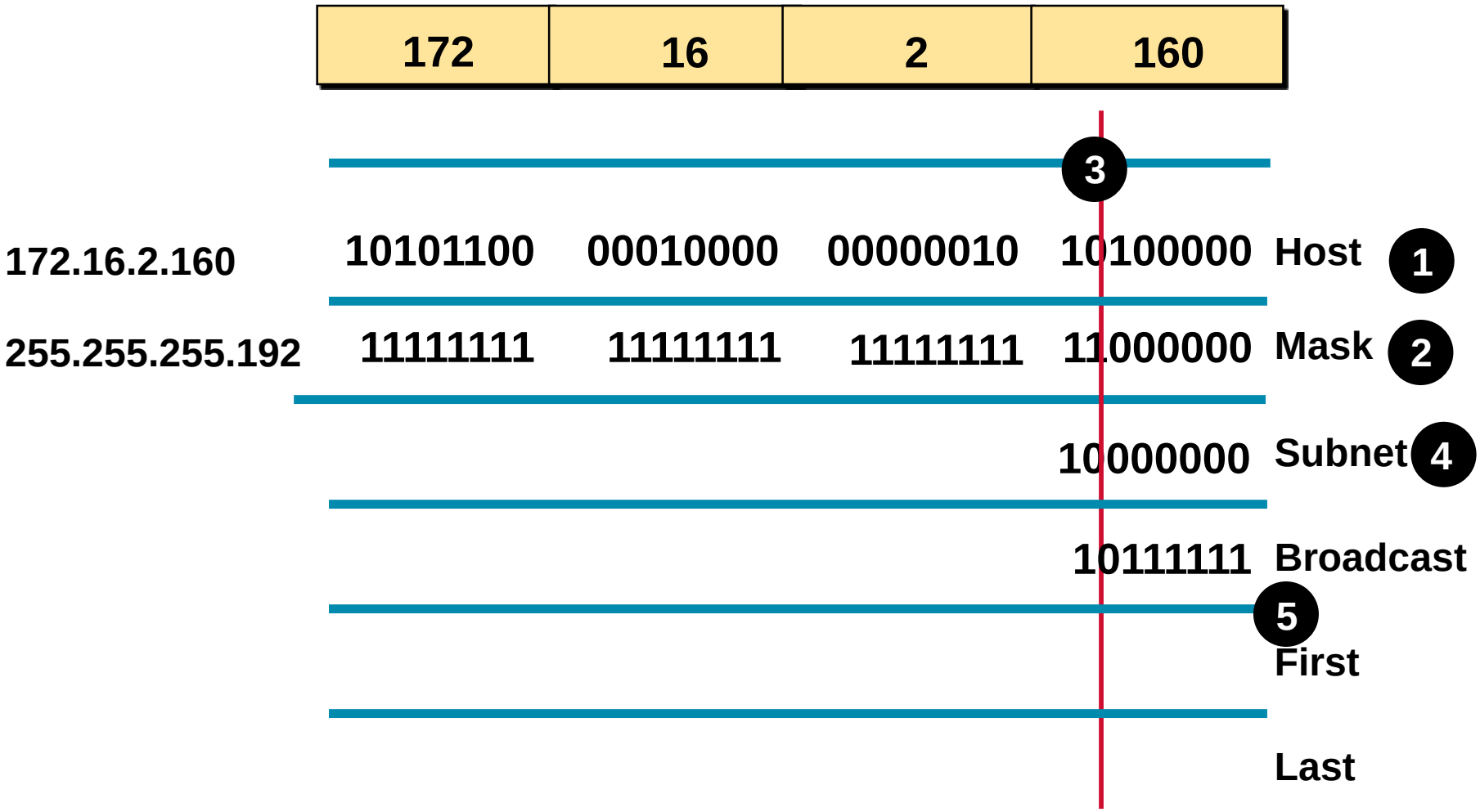
Addressing Summary Example



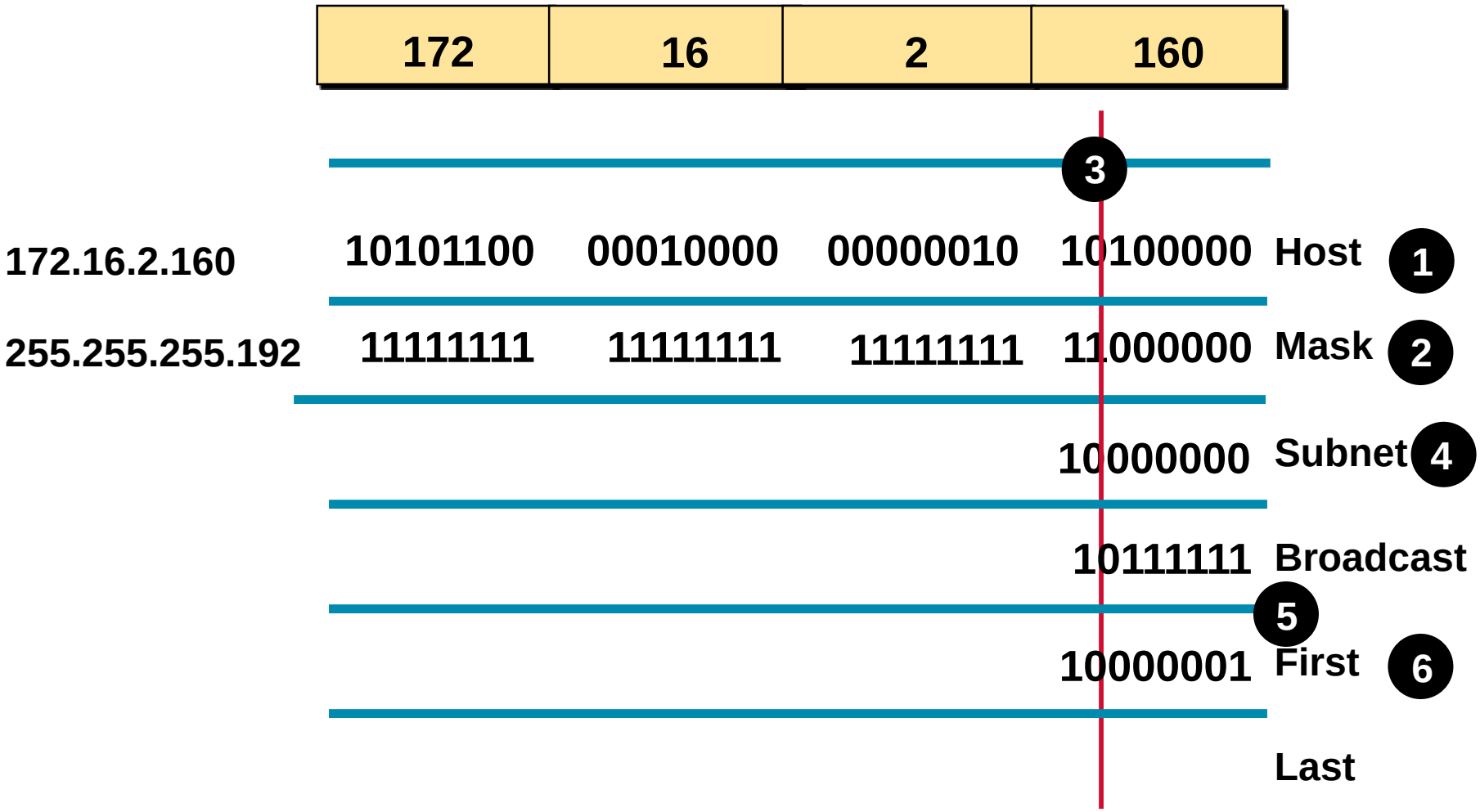
Addressing Summary Example



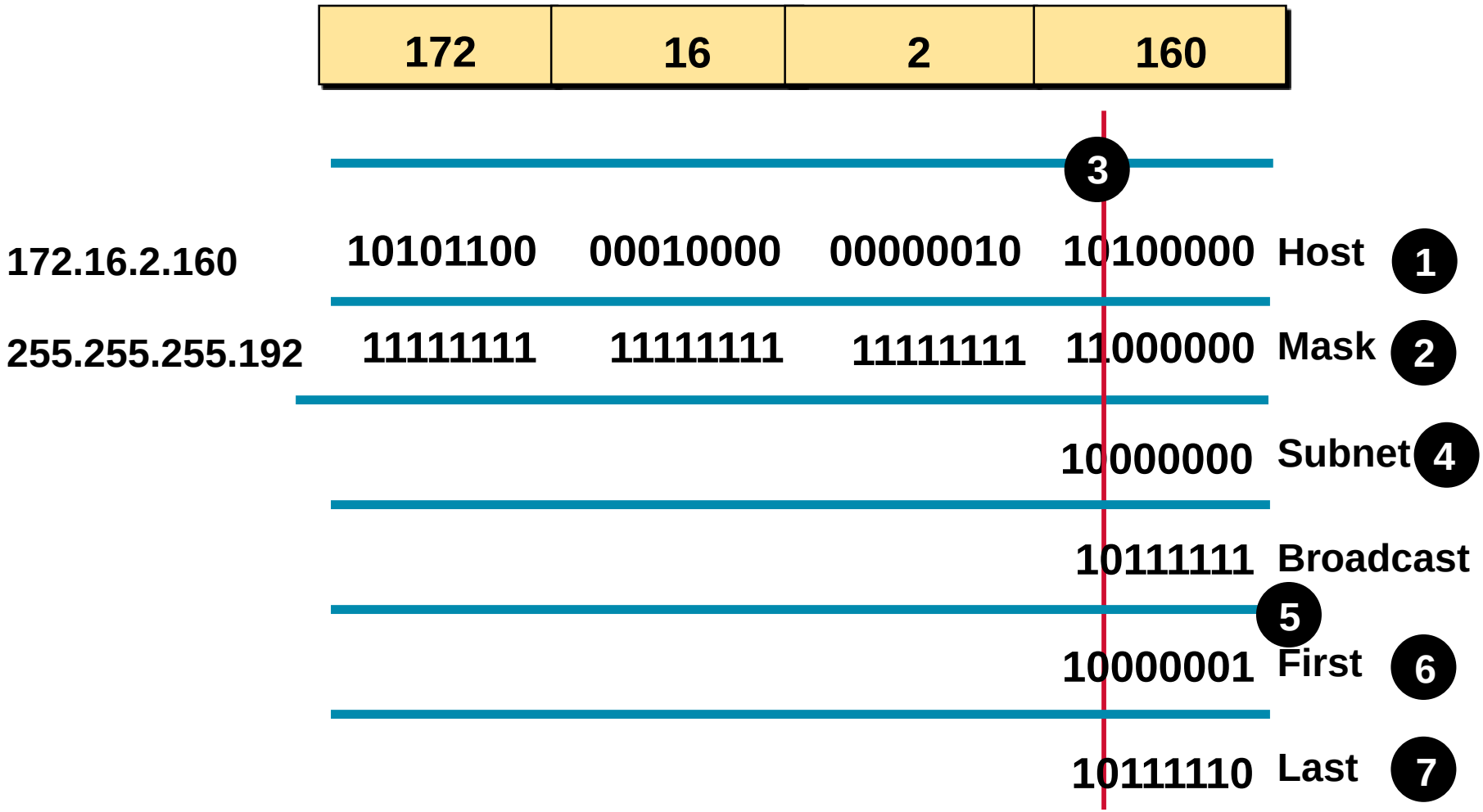
Addressing Summary Example



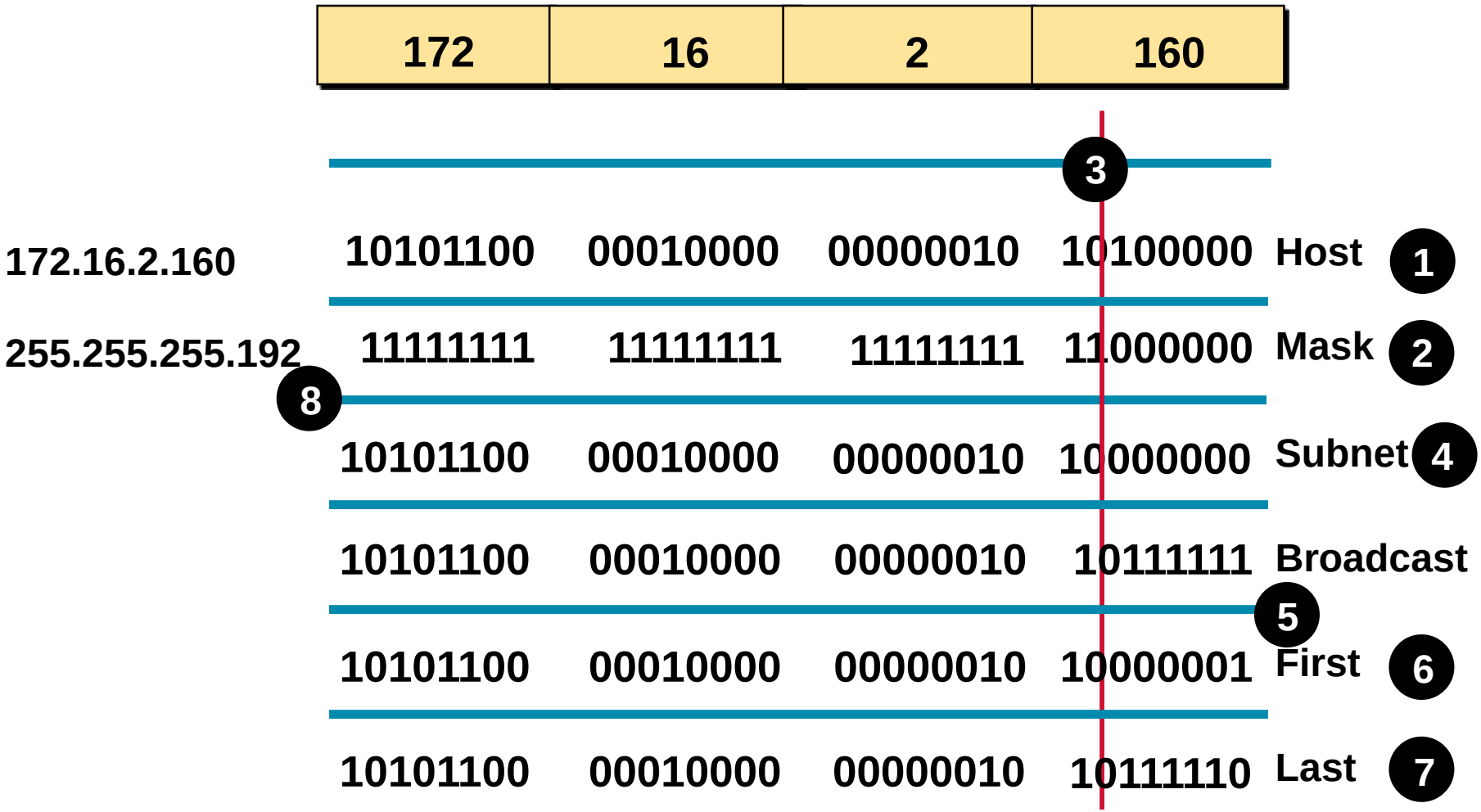
Addressing Summary Example



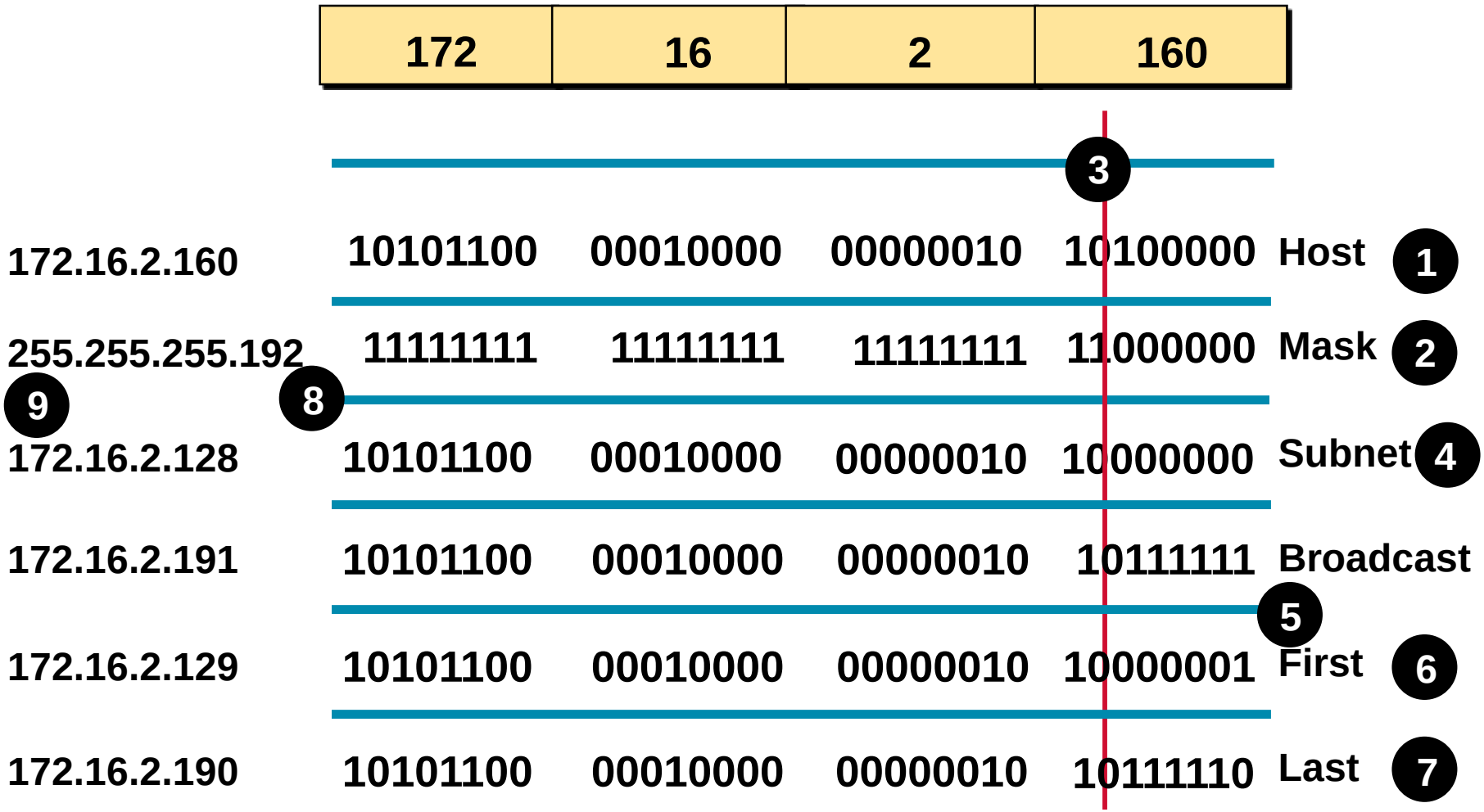
Addressing Summary Example



Addressing Summary Example



Addressing Summary Example



Class B Subnet Example

IP Host Address: 172.16.2.121
Subnet Mask: 255.255.255.0

	Network	Network	Subnet	Host
172.16.2.121:	10101100	00010000	00000010	01111001
255.255.255.0:	11111111	11111111	11111111	00000000
Subnet:	10101100	00010000	00000010	00000000
Broadcast:	10101100	00010000	00000010	11111111

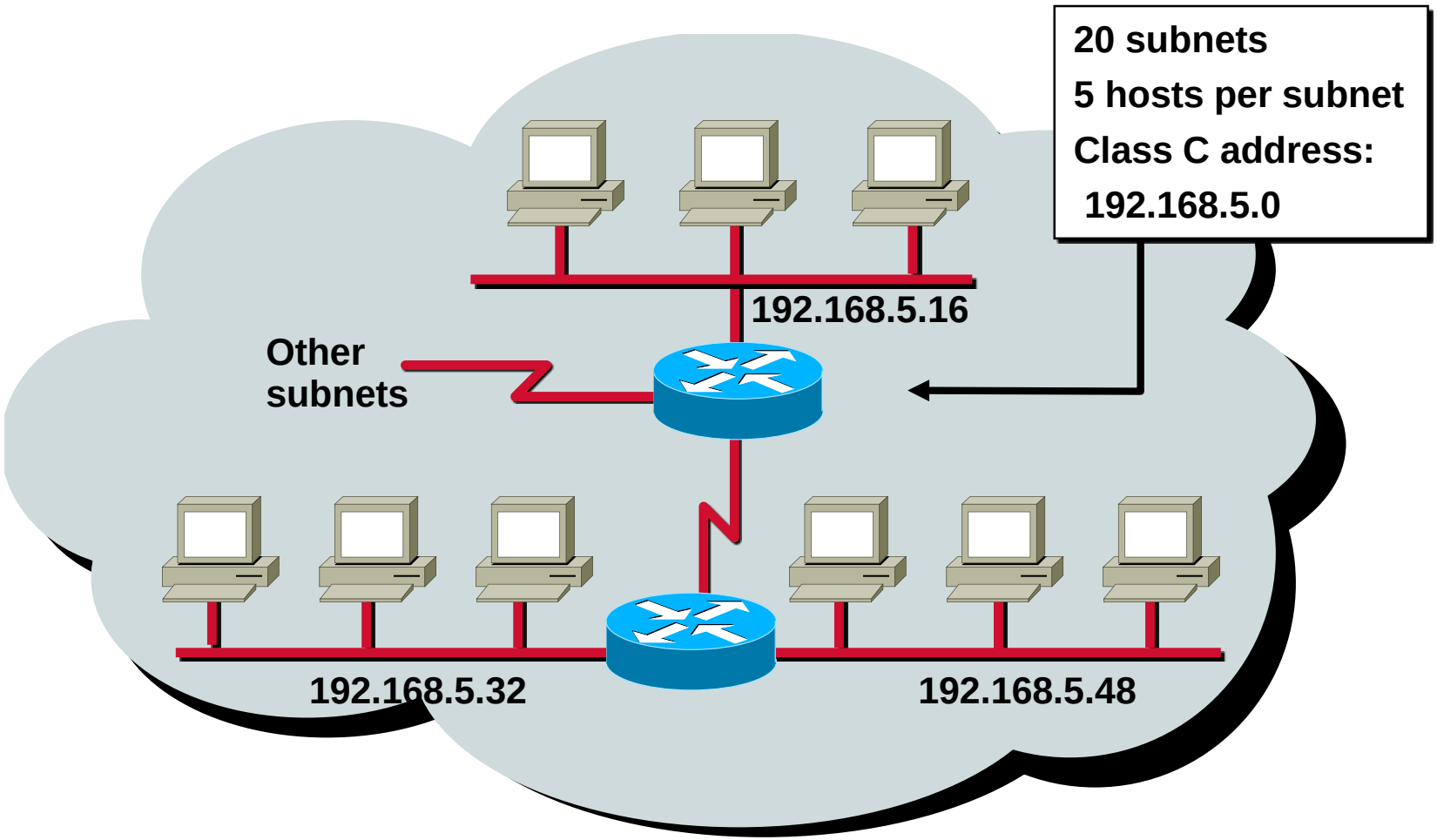
Subnet Address = 172.16.2.0

Host Addresses = 172.16.2.1–172.16.2.254

Broadcast Address = 172.16.2.255

Eight bits of subnetting

Subnet Planning



Class C Subnet Planning Example

IP Host Address: 192.168.5.121
Subnet Mask: 255.255.255.248

	Network	Network	Network	Subnet	Host
192.168.5.121:	11000000	10101000	00000101	01111001	
255.255.255.248:	11111111	11111111	11111111	11111000	
Subnet:	11000000	10101000	00000101	01111000	
Broadcast:	11000000	10101000	00000101	01111111	

Subnet Address = 192.168.5.120

Host Addresses = 192.168.5.121–192.168.5.126

Broadcast Address = 192.168.5.127

Five Bits of Subnetting

Broadcast Addresses Exercise

Address	Subnet Mask	Class	Subnet	Broadcast
201.222.10.60	255.255.255.248			
15.16.193.6	255.255.248.0			
128.16.32.13	255.255.255.252			
153.50.6.27	255.255.255.128			

Broadcast Addresses Exercise Answers

Address	Subnet Mask	Class	Subnet	Broadcast
201.222.10.60	255.255.255.248	C	201.222.10.56	201.222.10.63
15.16.193.6	255.255.248.0	A	15.16.192.0	15.16.199.255
128.16.32.13	255.255.255.252	B	128.16.32.12	128.16.32.15
153.50.6.27	255.255.255.128	B	153.50.6.0	153.50.6.127

Network Design Fundamentals

Internet Protocol: Subnetting

Presented by,

Jack Crowder, CCIE

Agenda

- Review
  - IP Addressing
    - Format
    - Classfull
    - Reserved
- Subnetting
  - Terms
  - Binary calculations
  - IP subnet calculation

Review

- RFC 1466 (IPv4)
- Address format:
  - 4 octets, dotted decimal notation
  - Example: 192.168.005.100
  - Applied to any interface that wants to communicate in an IP network
- Purpose:
  - Addresses logically grouped: Subnet
    - Routers “deal in” subnets

Classfull Boundaries – 1st Octet

00000000	0	Class A	/
01111111	127		
10000000	128	Class B	/
10111111	191		
11000000	192	Class C	/
11011111	223		
11100000	224	Class D	
11101111	239		

RFC 1878 and 1918

- reserved: 0.0.0.0 – 0.255.255.255 /8
- Class A: 1.0.0.0 - 127.255.255.255
  - reserved: 10.0.0.0 - 10.255.255.255 /8 (private)
  - reserved: 127.0.0.0 - 127.255.255.255 /8 (loopback)
- Class B: 128.0.0.0 - 191.255.255.255
  - reserved: 128.0.0.0 - 128.255.255.255 /8
  - reserved: 172.16.0.0 - 172.31.255.255 /12 (private)
- Class C: 192.0.0.0 - 223.255.255.255
  - reserved: 192.168.0.0 - 192.168.255.255 /16 (private)
- Class D: 224.0.0.0 – 239.255.255.255
  - reserved: 224.0.0.0 - 225.255.255.255 (Multicast)
- reserved: 255.255.255.255 (Broadcast) /32

Subnetting Terms

- Address
- Subnet
- Subnet Mask
- Network field (as determined by the Subnet Mask)
- Host field (as determined by the Subnet Mask)
- Subnet ID: all 0's (in the Host field)
- Broadcast ID: all 1's (in the Host field)
- Unicast (useable address on a subnet)
- Host route

Binary Calculation
2 to the power of X

- 4 Octets (a.k.a. Bytes) in a IPv4 address
  - 8 bits in an octet
  - 32 bits in an address
- 8 Bits in each octet
  - $2^8 = 256$
    - Decimal number range:

      0 – 255 (256 numbers counting “0”)
    - Binary range:

      00000000 – 11111111 (256 combinations of 1’s and 0’s)

Decimal to Binary

255 = 11111111
128 = 10000000
64 = 01000000
32 = 00100000
16 = 00010000
8 = 00001000
4 = 00000100
2 = 00000010
1 = 00000001
0 = 00000000

255 = 11111111
252 = 11111100
248 = 11111000
240 = 11110000
224 = 11100000
192 = 11000000
128 = 10000000
0 = 00000000

Binary Template

always start counting from 0

$2^x =$	128	64	32	16	8	4	2	1
x =	7	6	5	4	3	2	1	0
Position =	8	7	6	5	4	3	2	1

Subnet Mask

- Purpose:
  - Apply the Mask to the IP Address to determine:
    - Network bits
    - Host bits
    - Subnet ID, Broadcast ID & Unicast range
- Format:
  - 4 octets, dotted decimal notation (same as IP address)
  - Contiguous binary 1's starting from the left
- Examples:
  - 255.255.255.0 (typical for LAN segment)
  - 255.255.255.252 (typical for WAN pvc)
  - 255.255.255.1 (incorrect)

Subnet Mask in Binary

- 255.255.255.0
- 11111111.11111111.11111111.00000000

- 255.255.255.252
- 11111111.11111111.11111111.11111100

- 255.255.255.1 - incorrect
- 11111111.11111111.11111111.00000001

Subnet Calculation

- Step 1 – Convert:
  - decimal address & mask format to binary address & mask format
- Step 2 – Apply:
  - binary subnet mask to the binary IP address using the "and" function
- Step 3 – Calculate:
  - Subnet ID
  - Broadcast ID
  - Unicast range (usable subnet addresses)

DECIMAL

Step 1

XXX = 0 - 255

XXX.XXX.XXX.XXX

Network bits Host bits

address 185.213.22.219

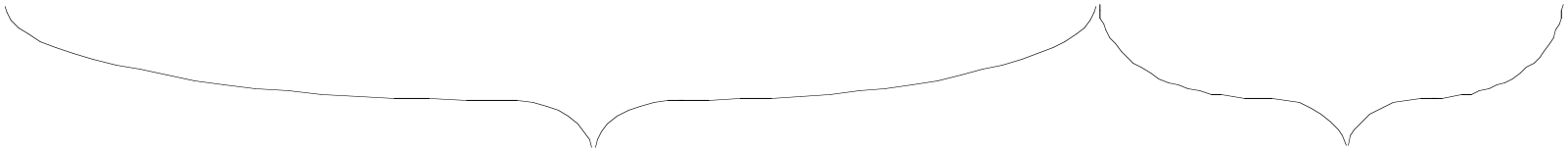
subnet mask 255.255.255.0

Step 1

XXXXXXXX = 00000000 - 1111111

xxxxxxxx.xxxxxxxx.xxxxxxxx.xxxxxxxx

BINARY



Network bits Host bits

10111001.11010101.00010110.11011011

11111111.11111111.11111111.00000000

Step 2: IP Subnet Calculation

Subnet MASK

<AND> IP Addres

IP Subnet

<AND> Rules:

1 and 1 = 1

1 and 0 = 0

0 and 1 = 0

0 and 0 = 0

Another way:

1 and X = X

0 and X = 0

Step 3 – IP Subnet – Example 1

185.213.022.219 255.255.255.000 /24

11111111.11111111.11111111.00000000

_____.11010101.00010110.11011011

Subnet ID

Broadcast ID

Subnet ID: ____.____.____.____

Broadcast ID: ____.____.____.____

Unicast: ____.____.____.____ - ____

IP Subnet – Example 2

185.213.022.219 255.255.255.252 /30

11111111.11111111.11111111.11111100

_____.11010101.00010110.11011011

Subnet ID

Broadcast ID

Subnet ID: ____.____.____.____

Broadcast ID: ____.____.____.____

Unicast: ____.____.____.____ - ____

Useable IP Address Calculations

1) 32 bits in address
2) 32 - network bits = host bits
3) 2 to the power of host bits = addresses on subnet
4) addresses - 2 (Broadcast and Subnet ID)
   = usable addresses on subnet

255.255.255.240 = /28
32 - 28 = 4
2^4 = 16 Addresses on Subnet
16 - 2 = 14 Unicast addresses on Subnet

Host Route

- Purpose:
  - Used on Loopback and Dial-up interfaces
- Example:
  - 10.5.109.22 255.255.255.255 (/32)

IP Subnet example 3

- IP Address: _____._____._____._____
- Subnet Mask: _____._____._____._____ _____
  - Network bits _____ Host bits _____

- Subnet: _____._____._____._____
- Broadcast: _____._____._____._____
- Usable Range: _____._____._____._____
- through _____._____._____._____

Linux Shell Scripting Tutorial Ver. 1.0

Written by Vivek G Gite

I N D E X

- Introduction
  - Kernel
  - Shell
  - How to use Shell
  - Common Linux Command Introduction
- Process
  - Why Process required
  - Linux commands related with process
- Redirection of Standard output/input
  - Redirectors
  - Pipes
  - Filters
- Shell Programming
  - Variables in Linux
  - How to define User defined variables
  - Rules for Naming variable name
  - How to print or access value of UDV (User defined variables)
  - How to write shell script
  - How to Run Shell Scripts
  - Quotes in Shell Scripts

- Shell Arithmetic
- Command Line Processing (Command Line Arguments)
- Why Command Line arguments required
- Exit Status
- Filename Shorthand or meta Characters (i.e. wild cards)

- Programming Commands
  - echo command
  - Decision making in shell script (i.e. if command)
  - test command or [expr]
  - Loop in shell scripts
  - The case Statement
  - The read Statement
- More Advanced Shell Script Commands
  - /dev/null - Use to send unwanted output of program
  - Local and Global Shell variable (export command)
  - Conditional execution i.e. && and ||
  - I/O Redirection and file descriptors
  - Functions
  - User Interface and dialog utility
  - trap command
  - getopts command
  - More examples of Shell Script (Exercise for You :-)

Introduction

This tutorial is designed for beginners only and This tutorial explains the basics of shell programming by showing some examples of shell programs. Its not help or manual for the shell. While reading this tutorial you can find manual quite useful (type man bash at $ prompt to see manual pages). Manual contains all necessary information you need, but it won't have that much examples, which makes idea more clear. For that reason, this tutorial contains examples rather than all the features of shell. I assumes you have at least working knowledge of Linux i.e. basic commands like how to create, copy, remove files/directories etc or how to use editor like vi or mcedit and login to your system. Before Starting Linux Shell Script Programming you must know

- Kernel
- Shell
- Process
- Redirectors, Pipes, Filters etc.

What's Kernel

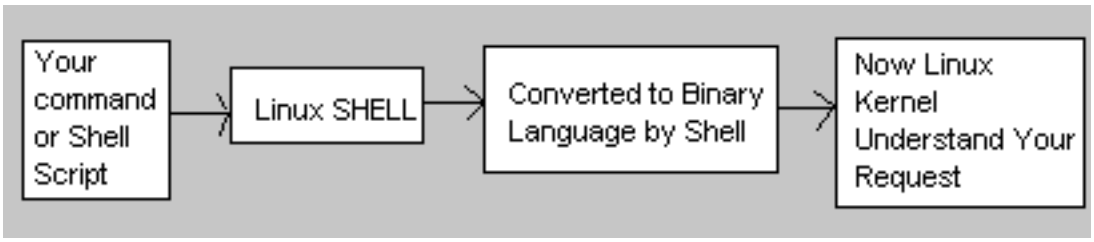
Kernel is hart of Linux O/S. It manages resource of Linux O/S. Resources means facilities available in Linux. For eg. Facility to store data, print data on printer, memory, file management etc . Kernel decides who will use this resource, for how long and when. It runs your programs (or set up to execute binary files) It's Memory resident portion of Linux. It performance following task :-

- I/O management
- Process management
- Device management
- File management
- Memory management

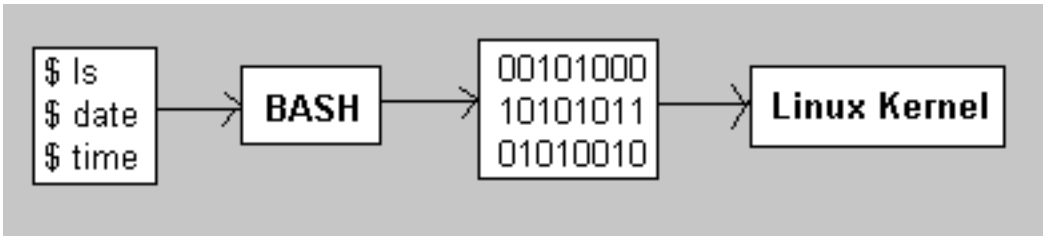
What's Linux Shell

Computer understand the language of 0's and 1's called binary language, In early days of computing, instruction are provided using binary language, which is difficult for all of us, to read and write. So in O/s there is special program called Shell. Shell accepts your instruction or commands in English and translate it into computers native binary language.

This is what Shell Does for US



You type Your command and shell convert it as



It's environment provided for user interaction. Shell is an command language interpreter that executes commands read from the standard input device (keyboard) or from a file. Linux may use one of the following most popular shells (In MS-DOS, Shell name is COMMAND.COM which is also used for same purpose, but it's not as powerful as our Linux Shells are!)

Shell Name	Developed by	Where	Remark
BASH (Bourne-Again SHell)	Brian Fox and Chet Ramey	Free Software Foundation	Most common shell in Linux. It's Freeware shell.
CSH (C SHell)	Bill Joy	University of California (For BSD)	The C shell's syntax and usage are very similar to the C programming language.
KSH (Korn SHell)	David Korn	AT & T Bell Labs	

Any of the above shell reads command from user (via Keyboard or Mouse) and tells Linux O/s what users want. If we are giving commands from keyboard it is called command line interface (Usually in-front of $ prompt, This prompt is depend upon your shell and Environment that you set or by your System Administrator, therefore you may get different prompt).
NOTE: To find your shell type following command
$ echo $SHELL

How to use Shell

To use shell (You start to use your shell as soon as you log into your system) you have to simply type commands. Following is the list of common commands.

Linux Common Commands

NOTE that following commands are for New users or for Beginners only. The purpose is if you use this command you will be more familiar with your shell and secondly, you need some of these command in your Shell script. If you want to get more information or help for this command try following commands For e.g. To see help or options related with date command try
$ date --help
or To see help or options related with ls command (Here you will screen by screen help, since help of ls command is quite big that can't fit on single screen)

```
$ ls --help | more
```

Syntax: *command-name --help*
Syntax: *man command-name*
Syntax: *info command-name*

See what happened when you type following

```
$ man ls
$ info bash
```

NOTE: In MS-DOS, you get help by using /? clue or by typing help command as

```
C:\> dir /?
C:\> date /?
C:\> help time
C:\> help date
C:\> help
```

Linux Command

For this Purpose	Use this Command Syntax	Example (In front of $ Prompt)
To see date	date	$ date
To see who's using system.	who	$ who
Print working directory	pwd	$ pwd
List name of files in current directory	ls or dirs	$ ls
To create text file NOTE: Press and hold **CTRL** key and press **D** to stop or to end file (**CTRL+D**)	cat > { file name }	$ cat > myfile *type your text* *when done press ^D*
To text see files	cat {file name }	$ cat myfile
To display file one full screen at a time	more {file name }	$ more myfile
To move or rename file/directory	mv {file1} {file2}	$ mv sales sales.99
To create multiple file copies with various link. After this both oldfile newfile refers to same name	ln {oldfile} {newfile}	**$ ln Page1 Book1**
To remove file	rm file1	$ rm myfile

Remove all files in given directory/subdirectory. Use it very carefully.	rm -rf {dirname}	**$ rm -rf oldfiles**
To change file access permissions u - User who owns the file g - Group file owner o - User classified as other a - All other system user + Set permission - Remove permission r - Read permission w - Write permission x - Execute permission	chmod {u|g|o|a} {+|-} {r|w|x} {filename}	**$ chmod u+x,g+wx,o+x myscript** NOTE: This command set permission for file called 'myscript' as User (Person who creates that file or directory) has execute permission (u+x) Group of file owner can write to this file as well as execute this file (g+wx) Others can only execute file but can not modify it, Since we have not given w (write permission) to them. (o+x).
Read your mail.	mail	$ mail
To See more about currently login person (i..e. yourself)	who am i	$ who am i
To login out	logout (OR press CTRL+D)	$ logout *(Note: It may ask you password type your login password, In some case this feature is disabled by System Administrator)*
Send mail to other person	mail {user-name}	$ mail ashish
To count lines, words and characters of given file	wc {file-name}	$wc myfile
To searches file for line that match a pattern.	grep {word-to-lookup} {filename}	$ grep fox myfile
To sort file in following order -r Reverse normal order -n Sort in numeric order -nr Sort in reverse numeric order	sort -r -n -nr {filename}	$sort myfile

To print last | first line of given file	tail - | + { linenumber } {filename}	$tail +5 myfile
To Use to compare files	cmp {file1} {file2} OR diff {file1} {file2}	$cmp myfile myfile.old
To print file	pr {file-name}	$pr myfile

What is Processes

Process is any kind of program or task carried out by your PC. For e.g. $ ls -lR , is command or a request to list files in a directory and all subdirectory in your current directory. It is a process. A process is program (command given by user) to perform some Job. In Linux when you start process, it gives a number (called PID or process-id), PID starts from 0 to 65535.

Why Process required

Linux is multi-user, multitasking o/s. It means you can run more than two process simultaneously if you wish. For e.g.. To find how many files do you have on your system you may give command like

```
$ ls / -R | wc -l
```

This command will take lot of time to search all files on your system. So you can run such command in Background or simultaneously by giving command like

```
$ ls / -R | wc -l &
```

The ampersand (&) at the end of command tells shells start command (ls / -R | wc -l) and run it in background takes next command immediately. An instance of running command is called process and the number printed by shell is called process-id (PID), this PID can be use to refer specific running process.

Linux Command Related with Process

For this purpose	Use this Command	Example
To see currently running process	ps	$ ps
To stop any process i.e. to kill process	kill {PID}	$ kill 1012
To get information about all running process	ps -ag	$ ps -ag
To stop all process except your shell	kill 0	$ kill 0
For background processing (With &, use to put particular command and program in background)	linux-command &	$ ls / -R | wc -l &

NOTE that you can only kill process which are created by yourself. A Administrator can almost kill 95-98% process. But some process can not be killed, such as VDU Process.

Redirection of Standard output/input or Input - Output redirection

Mostly all command gives output on screen or take input from keyboard, but in Linux it's possible to send output to file or to read input from file. For e.g. $ ls command gives output to screen; to send output to file of ls give command , $ ls > filename. It means put output of ls command to filename. There are three main redirection symbols >,>>,<

(1) > Redirector Symbol

Syntax: *Linux-command > filename*

To output Linux-commands result to file. Note that If file already exist, it will be overwritten else new file is created. For e.g. To send output of ls command give

```
$ ls > myfiles
```

Now if 'myfiles' file exist in your current directory it will be overwritten without any type of warning. (What if I want to send output to file, which is already exist and want to keep information of that file without loosing previous information/data?, For this Read next redirector)

(2) >> Redirector Symbol

Syntax: *Linux-command >> filename*

To output Linux-commands result to END of file. Note that If file exist , it will be opened and new information / data will be written to END of file, without losing previous information/data, And if file is not exist, then new file is created. For e.g. To send output of date command to already exist file give

```
$ date >> myfiles
```

(3) < Redirector Symbol

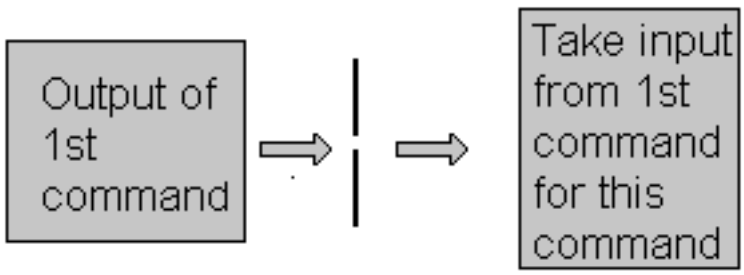
Syntax: *Linux-command < filename*

To take input to Linux-command from file instead of key-board. For e.g. To take input for cat command give

```
$ cat < myfiles
```

Pips

A pipe is a way to connect the output of one program to the input of another program without any temporary file.



A pipe is nothing but a temporary storage place where the output of one command is stored and then passed as the input for second command. Pipes are used to run more than two commands (Multiple commands) from same command line.
Syntax: *command1 | command2*

Command using Pips	Meaning or Use of Pipes
$ ls | more	Here the output of ls command is given as input to more command So that output is printed one screen full page at a time
$ who | sort	Here output of who command is given as input to sort command So that it will print sorted list of users
$ who | wc -l	Here output of who command is given as input to wc command So that it will number of user who logon to system
$ ls -l | wc -l	Here output of ls command is given as input to wc command So that it will print number of files in current directory.
$ who | grep raju	Here output of who command is given as input to grep command So that it will print if particular user name if he is logon or nothing is printed (To see for particular user logon)

Filter

If a Linux command accepts its input from the standard input and produces its output on standard output is know as a filter. A filter performs some kind of process on the input and gives output. For e.g.. Suppose we have file called 'hotel.txt' with 100 lines data, And from 'hotel.txt' we would like to print contains from line number 20 to line number 30 and store this result to file called 'hlist' then give command

```
$ tail +20 < hotel.txt | head -n30 >hlist
```

Here head is filter which takes its input from tail command (tail command start selecting from line number 20 of given file i.e. hotel.txt) and passes this lines to input to head, whose output is redirected to 'hlist' file.

Introduction to Shell Programming

Shell program is series of Linux commands. Shell script is just like batch file is MS-DOS but have more power than the MS-DOS batch file. Shell script can take input from user, file and output them on screen. Useful to create our own commands that can save our lots of time and to automate some task of day today life.

Variables in Linux

Sometimes to process our data/information, it must be kept in computers RAM memory. RAM memory is divided into small locations, and each location had unique number called memory location/address, which is used to hold our data. Programmer can give a unique name to this memory location/address called memory variable or variable (Its a named storage location that may take different values, but only one at a time). In Linux, there are two types of variable

1) System variables - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.

2) User defined variables (UDV) - Created and maintained by user. This type of variable defined in lower LETTERS.

Some System variables

You can see system variables by giving command like $ set, Some of the important System variables are

System Variable	Meaning
BASH=/bin/bash	Our shell name
BASH_VERSION=1.14.7(1)	Our shell version name
COLUMNS=80	No. of columns for our screen
HOME=/home/vivek	Our home directory
LINES=25	No. of columns for our screen
LOGNAME=students	Our logging name
OSTYPE=Linux	Our o/s type : -)
PATH=/usr/bin:/sbin:/bin:/usr/sbin	Our path settings
PS1=[\u@\h \W]\$	Our prompt settings
PWD=/home/students/Common	Our current working directory
SHELL=/bin/bash	Our shell name
USERNAME=vivek	User name who is currently login to this PC

NOTE that Some of the above settings can be different in your PC. You can print any of the above variables contain as follows

```
$ echo $USERNAME
$ echo $HOME
```

Caution: Do not modify System variable this can some time create problems.

How to define User defined variables (UDV)

To define UDV use following syntax

Syntax: *variablename=value*

NOTE: Here 'value' is assigned to given 'variablename' and Value must be on right side = sign For e.g.

```
$ no=10     # this is ok
$ 10=no     # Error, NOT Ok, Value must be on right side of = sign.
```

To define variable called 'vech' having value Bus

```
$ vech=Bus
```
To define variable called n having value 10
```
$ n=10
```

Rules for Naming variable name (Both UDV and System Variable)

(1) Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character. For e.g. Valid shell variable are as follows
HOME
SYSTEM_VERSION
vech
no

(2) Don't put spaces on either side of the equal sign when assigning value to variable. For e.g.. In following variable declaration there will be no error
```
$ no=10
```
But here there will be problem for following
```
$ no    =10
$ no=    10
$ no    =    10
```

(3) Variables are case-sensitive, just like filename in Linux. For e.g.
```
$ no=10
$ No=11
$ NO=20
$ nO=2
```
Above all are different variable name, so to print value 20 we have to use $ echo $NO and Not any of the following
```
$ echo $no      # will print 10 but not 20
$ echo $No      # will print 11 but not 20
$ echo $nO      # will print 2 but not 20
```

(4) You can define NULL variable as follows (NULL variable is variable which has no value at the time of definition) For e.g.
```
$ vech=
$ vech=""
```
Try to print it's value $ echo $vech , Here nothing will be shown because variable has no value i.e. NULL variable.

(5) Do not use ?,* etc, to name your variable names.

How to print or access value of UDV (User defined variables)

To print or access UDV use following syntax
Syntax: *$variablename*
For eg. To print contains of variable 'vech'
```
$ echo $vech
```
It will print 'Bus' (if previously defined as vech=Bus) ,To print contains of variable 'n' $ echo $n
It will print '10' (if previously defined as n=10)
Caution: Do not try $ echo vech It will print vech instead its value 'Bus' and $ echo n, It will print n instead its value '10', You must use $ followed by variable name.

Q.1.How to Define variable x with value 10 and print it on screen
```
$ x=10
$ echo $x
```

Q.2.How to Define variable xn with value Rani and print it on screen
```
$ xn=Rani
$ echo $xn
```

Q.3.How to print sum of two numbers, let's say 6 and 3
```
$ echo 6 + 3
```
This will print 6 + 3, not the sum 9, To do sum or math operations in shell use expr, syntax is as follows Syntax: *expr op1 operator op2*
Where, op1 and op2 are any Integer Number (Number without decimal point) and operator can be
+ Addition
- Subtraction
/ Division
% Modular, to find remainder For e.g. 20 / 3 = 6 , to find remainder 20 % 3 = 2, (Remember its integer calculation)
* Multiplication
```
$ expr 6 + 3
```
Now It will print sum as 9 , But
```
$ expr 6+3
```
will not work because space is required between number and operator (See Shell Arithmetic)

Q.4.How to define two variable x=20, y=5 and then to print division of x and y (i.e. x/y)
```
$x=20
$ y=5
$ expr x / y
```

Q.5.Modify above and store division of x and y to variable called z
```
$ x=20
$ y=5
$ z=`expr x / y`
$ echo $z
```
Note : For third statement, read Shell Arithmetic.

How to write shell script

Now we write our first script that will print "Knowledge is Power" on screen. To write shell script you can use in of the Linux's text editor such as vi or mcedit or even you can use cat command. Here we are using cat command you can use any of the above text editor. First type following cat command and rest of text as its

```
$ cat > first
#
# My first shell script
#
clear
echo "Knowledge is Power"
```

Press Ctrl + D to save. Now our script is ready. To execute it type command
```
$ ./first
```
This will give error since we have not set Execute permission for our script first; to do this type command
```
$ chmod +x first
$ ./first
```
First screen will be clear, then Knowledge is Power is printed on screen. To print message of variables contains we user echo command, general form of echo command is as follows
echo "Message"
echo "Message variable1, variable2....variableN"

How to Run Shell Scripts

Because of security of files, in Linux, the creator of Shell Script does not get execution permission by default. So if we wish to run shell script we have to do two things as follows

(1) Use chmod command as follows to give execution permission to our script

Syntax: *chmod +x shell-script-name*

OR Syntax: *chmod 777 shell-script-name*

(2) Run our script as

Syntax: *./your-shell-program-name*

For e.g.

```
$ ./first
```

Here '.'(dot) is command, and used in conjunction with shell script. The dot(.) indicates to current shell that the command following the dot(.) has to be executed in the same shell i.e. without the loading of another shell in memory. Or you can also try following syntax to run Shell Script

Syntax: *bash &nbsh;&nbsh; your-shell-program-name*

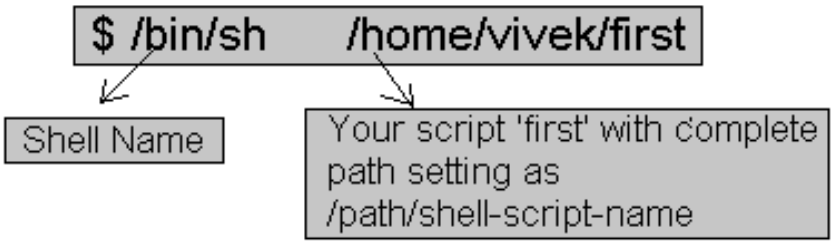
OR */bin/sh &nbsh;&nbsh; your-shell-program-name*

For e.g.

```
$ bash first
$ /bin/sh first
```

Note that to run script, you need to have in same directory where you created your script, if you are in different directory your script will not run (because of path settings), For eg. Your home directory is (use $ pwd to see current working directory) /home/vivek. Then you created one script called 'first', after creation of this script you moved to some other directory lets say /home/vivek/Letters/Personal, Now if you try to execute your script it will not run, since script 'first' is in /home/vivek directory, to Overcome this problem there are two ways First, specify complete path of your script when ever you want to run it from other directories like giving following command

```
$ /bin/sh /home/vivek/first
```



Now every time you have to give all this detailed as you work in other directory, this take time and you have to remember complete path. There is another way, if you notice that all of our programs (in form of executable files) are marked as executable and can be directly executed from prompt from any directory (To see executables of our normal program give command $ ls -l /bin or ls -l /usr/bin) by typing command like

```
$ bc
$ cc myprg.c
$ cal
```

etc, How this happed? All our executables files are installed in directory called /bin and /bin directory is set in your PATH setting, Now when you type name of any command at $ prompt, what shell do is it first look that command in its internal part (called as internal command, which is part of Shell itself, and always available to execute, since they do not need extra executable file), if found as internal command shell will execute it, If not found It will look for current directory, if found shell will execute command from current directory, if not found, then Shell will Look PATH setting, and try to find our requested commands executable file in all of the directories mentioned in PATH settings, if found it will execute it, otherwise it will give message "bash: xxxx :command not found", Still there is one question remain can I run my shell script same as these executables. Yes you can, for

this purpose create bin directory in your home directory and then copy your tested version of shell script to this bin directory. After this you can run you script as executable file without using $ ./shell script-name syntax, Following are steps

```
$ cd
$ mkdir bin
$ cp first ~/bin
$ first
```

Each of above command Explanation

Each of above command	Explanation
$ cd	Go to your home directory
$ mkdir bin	Now created bin directory, to install your own shell script, so that script can be run as independent program or can be accessed from any directory
$ cp first ~/bin	copy your script 'first' to your bin directory
$ first	Test whether script is running or not (It will run)

In shell script comment is given with # character. This comments are ignored by your shell. Comments are used to indicate use of script or person who creates/maintained script, or for some programming explanation etc. Remember always set *Execute permission for you script*.

Commands Related with Shell Programming

(1)*echo [options] [string, variables...]*

Displays text or variables value on screen.

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following backslash escaped characters in the strings:

\a alert (bell)
\b backspace
\c suppress trailing new line
\n new line
\r carriage return
\t horizontal tab
\\ backslash

For eg. $ echo -e "An apple a day keeps away \a\t\tdoctor\n"

(2)*More about Quotes*

There are three types of quotes

" i.e. Double Quotes
' i.e. Single quotes
` i.e. Back quote

1."Double Quotes" - Anything enclose in double quotes removed meaning of that characters (except \ and $).

2. 'Single quotes' - Enclosed in single quotes remains unchanged.

3. `Back quote` - To execute command.

For eg.

```
$ echo "Today is date"
```

Can't print message with today's date.

```
$ echo "Today is `date`".
```

Now it will print today's date as, Today is Tue Jan ....,See the `date` statement uses back quote, (See also Shell Arithmetic NOTE).

(3) *Shell Arithmetic*

Use to perform arithmetic operations For e.g.

```
$ expr 1 + 3
$ expr 2 - 1
$ expr 10 / 2
$ expr 20 % 3 # remainder read as 20 mod 3 and remainder is 2)
$ expr 10 \* 3 # Multiplication use \* not * since its wild card)
$ echo `expr 6 + 3`
```

For the last statement not the following points

1) First, before expr keyword we used ` (back quote) sign not the (single quote i.e. ') sign. Back quote is generally found on the key under tilde (~) on PC keyboards OR To the above of TAB key.
2) Second, expr is also end with ` i.e. back quote.
3) Here expr 6 + 3 is evaluated to 9, then echo command prints 9 as sum
4) Here if you use double quote or single quote, it will NOT work, For eg.

```
$ echo "expr 6 + 3"    # It will print expr 6 + 3
$ echo 'expr 6 + 3'
```

Command Line Processing

Now try following command (assumes that the file "grate_stories_of" is not exist on your disk)

```
$ ls grate_stories_of
```

It will print message something like -

grate_stories_of: No such file or directory

Well as it turns out ls was the name of an actual command and shell executed this command when given the command. Now it creates one question What are commands? What happened when you type $ ls grate_stories_of? The first word on command line, ls, is name of the command to be executed. Everything else on command line is taken as arguments to this command. For eg.

```
$ tail +10 myf
```

Here the name of command is tail, and the arguments are +10 and myf.

Now try to determine command and arguments from following commands:

```
$ ls    foo
$ cp    y    y.bak
$ mv    y.bak    y.okay
$ tail    -10    myf
$ mail    raj
$ sort    -r    -n    myf
$ date
$ clear
```

Command	No. of argument to this command	Actual Argument
ls	1	foo
cp	2	y and y.bak
mv	2	y.bak and y.okay
tail	2	-10 and myf
mail	1	raj
sort	3	-r, -n, and myf
date	0	
clear	0	

NOTE: $# holds number of arguments specified on command line. and $* or $@ refer to all arguments in passed to script. Now to obtain total no. of Argument to particular script, your $# variable.

Why Command Line arguments required

Let's take rm command, which is used to remove file, But which file you want to remove and how you will you tail this to rm command (Even rm command does not ask you name of file that would like to remove). So what we do is we write as command as follows

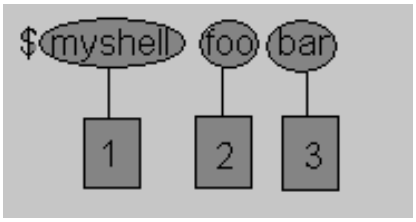
```
$ rm {file-name}
```

Here rm is command and file-name is file which you would like to remove. This way you tail to rm command which file you would like to remove. So we are doing one way communication with our command by specifying file-name. Also you can pass command line arguments to your script to make it more users friendly. But how we address or access command line argument in our script. Lets take ls command

```
$ ls -a /*
```

This command has 2 command line argument -a and /* is another. For shell script,

```
$ myshell foo bar
```



1. Shell Script name i.e. myshell
2. First command line argument passed to myshell i.e. foo
3. Second command line argument passed to myshell i.e. bar

In shell if we wish to refer this command line argument we refer above as follows

1. myshell it is $0
2. foo it is $1
3. bar it is $2

Here $# will be 2 (Since foo and bar only two Arguments), Please note At a time such 9 arguments can be used from $0..$9, You can also refer all of them by using $* (which expand to `$0,$1,$2...$9`) Now try to write following for commands, Shell Script Name ($0), No. of Arguments (i.e. $#), And actual argument (i.e. $1,$2 etc)

```
$ sum 11 20
$ math 4 - 7
$ d
$ bp -5 myf +20
$ ls *
$ cal
$ findBS 4 8 24 BIG
```

Shell Script Name	No. Of Arguments to script	Actual Argument ($1,..$9)				
$0	$#	$0	$1	$2	$3	$4
sum	2	11	20			
math	3	4	-	7		
d	0					

bp	3	-5	myf	+20		
ls	1	*				
cal	0					
findBS	4	4	8	24	BIG	

For e.g. now will write script to print command ling argument and we will see how to access them

```
$ cat > demo
#!/bin/sh
#
# Script that demos, command line args
#
echo "Total number of command line argument are $#"
echo "$0 is script name"
echo "$1 is first argument"
echo $2 is second argument"
echo "All of them are :- $*"
```

Save the above script by pressing ctrl+d, now make it executable

```
$ chmod +x demo
$ ./demo Hello World
$ cp demo ~/bin
$ demo
```

Note: After this, For any script you have to used above command, in sequence, I am not going to show you all of the above.

(5)*Exit Status*

By default in Linux if particular command is executed, it return two type of values, (Values are used to see whether command is successful or not) if return value is zero (0), command is successful, if return value is nonzero (>0), command is not successful or some sort of error executing command/shell script. This value is know as Exit Status of that command. To determine this exit Status we use $? variable of shell. For eg.

```
$ rm unknow1file
```

It will show error as follows

```
rm: cannot remove `unkowm1file': No such file or directory
```

and after that if you give command `$ echo $?` it will print nonzero value(>0) to indicate error. Now give command

```
$ ls
$ echo $?
```

It will print 0 to indicate command is successful. Try the following commands and not down there exit status

```
$ expr 1 + 3
$ echo $?

$ echo Welcome
$ echo $?

$ wildwest canwork?
$ echo $?

$ date
$ echo $?

$ echon $?
```

```
$ echo $?
```

(6)*if-then-fi for decision making is shell script* Before making any decision in Shell script you must know following things Type bc at $ prompt to start Linux calculator program

```
$ bc
```

After this command bc is started and waiting for you commands, i.e. give it some calculation as follows type 5 + 2 as

```
5 + 2
7
```

7 is response of bc i.e. addition of 5 + 2 you can even try

```
5 - 2
5 / 2
```

Now what happened if you type 5 > 2 as follows

```
5 > 2
0
```

0 (Zero) is response of bc, How? Here it compare 5 with 2 as, Is 5 is greater then 2, (If I ask same question to you, your answer will be YES) In Linux (bc) gives this 'YES' answer by showing 0 (Zero) value. It means when ever there is any type of comparison in Linux Shell It gives only two answer one is YES and NO is other.

Linux Shell Value	Meaning	Example
Zero Value (0)	Yes/True	0
NON-ZERO Value (> 0)	No/False	-1, 32, 55 anything but not zero

Try following in bc to clear your Idea and not down bc's response

```
5 > 12
5 == 10
5 != 2
5 == 5
12 < 2
```

Expression	Meaning to us	Your Answer	BC's Response (i.e. Linux Shell representation in zero & non-zero value)
5 > 12	Is 5 greater than 12	NO	0
5 == 10	Is 5 is equal to 10	NO	0
5 != 2	Is 5 is NOT equal to 2	YES	1
5 == 5	Is 5 is equal to 5	YES	1
1 < 2	Is 1 is less than 2	Yes	1

Now will see, if condition which is used for decision making in shell script, If given condition is true then command1 is executed.

Syntax:

```
if condition
then
        command1 if condition is true or if exit status
        of condition is 0 (zero)
        ...
        ...
```

fi

Here *condition* is nothing but comparison between two values, for compression we can use test or [expr] statements or even exist status can be also used. An *expression* is nothing but combination of values, relational operator (such as >,<, <> etc) and mathematical operators (such as +, -, / etc). Following are all examples of expression:

```
5 > 2
3 + 6
3 * 65
a < b
c > 5
c > 5 + 30 -1
```

Type following command (assumes you have file called foo)

```
$ cat foo
$ echo $?
```

The cat command return zero(0) on successful, this can be used in if condition as follows, Write shell script as

```
$ cat > showfile
#!/bin/sh
#
#Script to print file
#
if cat $1
then
   echo -e "\n\nFile $1, found and successfully echoed"
fi
```

Now run it.

```
$ chmod +x showfile
$./showfile foo
```

Here

```
$ ./showfile  foo
```

Our shell script name is showfile($0) and foo is argument (which is $1).Now we compare as follows if cat $1 (i.e. if cat foo)

Now if cat command finds foo file and if its successfully shown on screen, it means our cat command is successful and its exist status is 0 (indicates success) So our if condition is also true and hence statement echo -e "\n\nFile $1, found and successfully echoed" is proceed by shell. Now if cat command is not successful then it returns non-zero value (indicates some sort of failure) and this statement echo -e "\n\nFile $1, found and successfully echoed" is skipped by our shell.

Now try to write answer for following

1) Create following script

```
cat > trmif
#
# Script to test rm command and exist status
#
if rm $1
then
   echo "$1 file deleted"
fi
```

(Press Ctrl + d to save)

```
$ chmod +x trmif
```

Now answer the following

A) There is file called foo, on your disk and you give command, $ ./trmfi foo what will be output.

B) If bar file not present on your disk and you give command, $ ./trmfi bar what will be output.
C) And if you type $ ./trmfi, What will be output.

(7)*test command or [expr]*
test command or [expr] is used to see if an expression is true, and if it is true it return zero(0), otherwise returns nonzero(>0) for false. Syntax: *test expression* OR *[expression]*
Now will write script that determine whether given argument number is positive. Write script as follows

```
$ cat > ispostive
#!/bin/sh
#
# Script to see whether argument is positive
#
if test $1 -gt 0
then
   echo "$1 number is positive"
fi
```

Run it as follows

```
$ chmod +x ispostive
$ ispostive 5
```

Here o/p : 5 number is positive

```
$ispostive -45
```

Here o/p : Nothing is printed

```
$ispostive
```

Here o/p : ./ispostive: test: -gt: unary operator expected
The line, if test $1 -gt 0 , test to see if first command line argument($1) is greater than 0. If it is true(0) then test will return 0 and output will printed as 5 number is positive but for -45 argument there is no output because our condition is not true(0) (no -45 is not greater than 0) hence echo statement is skipped. And for last statement we have not supplied any argument hence error ./ispostive: test: -gt: unary operator expected is generated by shell , to avoid such error we can test whether command line argument is supplied or not. (See command 8 Script example). test or [expr] works with
1.Integer (Number without decimal point)
2.File types
3.Character strings
For Mathematics use following operator in Shell Script

Math- ematical Operator in Shell Script	Meaning	Normal Arithmetical/ Mathematical Statements	But in Shell	
			For test statement with if command	For [expr] statement with if command
-eq	is equal to	5 == 6	if test 5 -eq 6	if expr [5 -eq 6]
-ne	is not equal to	5 != 6	if test 5 -ne 6	if expr [5 -ne 6]
-lt	is less than	5 < 6	if test 5 -lt 6	if expr [5 -lt 6]
-le	is less than or equal to	5 <= 6	if test 5 -le 6	if expr [5 -le 6]
-gt	is greater than	5 > 6	if test 5 -gt 6	if expr [5 -gt 6]
-ge	is greater than or equal to	5 >= 6	if test 5 -ge 6	if expr [5 -ge 6]

NOTE: == is equal, != is not equal.
For string Comparisons use

Operator	Meaning
string1 = string2	string1 is equal to string2
string1 != string2	string1 is NOT equal to string2
string1	string1 is NOT NULL or not defined
-n string1	string1 is NOT NULL and does exist
-z string1	string1 is NULL and does exist

Shell also test for file and directory types

Test	Meaning
-s file	Non empty file
-f file	Is File exist or normal file and not a directory
-d dir	Is Directory exist and not a file
-w file	Is writeable file
-r file	Is read-only file
-x file	Is file is executable

Logical Operators
Logical operators are used to combine two or more condition at a time

Operator	Meaning
! expression	Logical NOT
expression1 -a expression2	Logical AND
expression1 -o expression2	Logical OR

(8)*if...else...fi*
If given condition is true then command1 is executed otherwise command2 is executed.
Syntax:

```
if  condition
then
        command1 if condition is true or if exit status
        of condition is 0(zero)
        ...
        ...
else
        command2 if condition is false or if exit status
        of condition is >0 (nonzero)
         ...
         ...
fi
```

For eg. Write Script as follows

```
$ cat > isnump_n
#!/bin/sh
#
```

```
# Script to see whether argument is positive or negative
#
if [ $# -eq 0 ]
then
   echo "$0 : You must give/supply one integers"
   exit 1
fi


if test $1 -gt 0
then
   echo "$1 number is positive"
else
   echo "$1 number is negative"
fi
```

Try it as follows

```
$ chmod +x isnump_n
$ isnump_n 5
Here o/p : 5 number is positive
$ isnump_n -45
Here o/p : -45 number is negative
$ isnump_n
Here o/p : ./ispos_n : You must give/supply one integers
$ isnump_n 0
Here o/p : 0 number is negative
```

Here first we see if no command line argument is given then it print error message as "./ispos_n : You must give/supply one integers". if statement checks whether number of argument ($#) passed to script is not equal (-eq) to 0, if we passed any argument to script then this if statement is false and if no command line argument is given then this if statement is true. The echo command i.e.

```
echo "$0 : You must give/supply one integers"
      |          |
      |          |
      1          2
```

1 will print Name of script
2 will print this error message
And finally statement exit 1 causes normal program termination with exit status 1 (nonzero means script is not successfully run), The last sample run $ isnump_n 0 , gives output as "0 number is negative", because given argument is not > 0, hence condition is false and it's taken as negative number. To avoid this replace second if statement with if test $1 -ge 0.

(9)*Multilevel if-then-else*
Syntax:

```
if condition
then
            condition is zero (true - 0)
            execute all commands up to elif statement
elif condition1
            condition1 is zero (true - 0)
            execute all commands up to elif statement
elif condition2
            condition2 is zero (true - 0)
            execute all commands up to elif statement
```

```
else
        None of the above condtion,condtion1,condtion2 are true (i.e.
        all of the above nonzero or false)
        execute all commands up to fi
fi
```

For e.g. Write script as $ cat > elf #!/bin/sh # # Script to test if..elif...else # # if [$1 -gt 0] then echo "$1 is positive" elif [$1 -lt 0] then echo "$1 is negative" elif [$1 -eq 0] then echo "$1 is zero" else echo "Opps! $1 is not number, give number" fi Try above script with $ chmod +x elf $ ./elf 1 $ ./elf -2 $ ./elf 0 $ ./elf a Here o/p for last sample run: ./elf: [: -gt: unary operator expected ./elf: [: -lt: unary operator expected ./elf: [: -eq: unary operator expected Opps! a is not number, give number Above program gives error for last run, here integer comparison is expected therefore error like "./elf: [: -gt: unary operator expected" occurs, but still our program notify this thing to user by providing message "Opps! a is not number, give number". (10)*Loops in Shell Scripts*

Computer can repeat particular instruction again and again, until particular condition satisfies. A group of instruction that is executed repeatedly is called a loop.

(a) for loop Syntax:

```
for { variable name } in { list }
do
        execute one for each item in the list until the list is
        not finished (And repeat all statement between do and done)
done
```

Suppose,

```
$ cat > testfor
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

Run it as,

```
$ chmod +x testfor
$ ./testfor
```

The for loop first creates i variable and assigned a number to i from the list of number from 1 to 5, The shell execute echo statement for each assignment of i. (This is usually know as iteration) This process will continue until all the items in the list were not finished, because of this it will repeat 5 echo statements. for e.g. Now try script as follows

```
$ cat > mtable
#!/bin/sh
#
#Script to test for loop
#
#
if [ $# -eq 0 ]
then
    echo "Error - Number missing form command line argument"
    echo "Syntax : $0 number"
    echo " Use to print multiplication table for given number"
    exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
```

```
do
   echo "$n * $i = `expr $i \* $n`"
done
```

Save and Run it as

```
$ chmod +x mtable
$ ./mtable 7
$ ./mtable
```

For first run, Above program print multiplication table of given number where i = 1,2 ... 10 is multiply by given n (here command line argument 7) in order to produce multiplication table as

```
7 * 1 = 7
7 * 2 = 14
...
..
```

7 * 10 = 70 And for Second run, it will print message -

```
Error - Number missing form command line argument
Syntax : ./mtable number
        Use to print multiplication table for given number
```

This happened because we have not supplied given number for which we want multiplication table, Hence we are showing Error message, Syntax and usage of our script. This is good idea if our program takes some argument, let the user know what is use of this script and how to used it. Note that to terminate our script we used 'exit 1' command which takes 1 as argument (1Indicates error and therefore script is terminated)

(b)while loop

Syntax:

```
while [ condition ]
do
        command1
        command2
        command3
        ..
        ....
 done
```

Loop is executed as long as given condition is true. For eg. Above for loop program can be written using while loop as

```
$cat > nt1
#!/bin/sh
#
#Script to test while statement
#
#
if [ $# -eq 0 ]
then
   echo "Error - Number missing form command line argument"
   echo "Syntax : $0 number"
   echo " Use to print multiplication table for given number"
   exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
```

```
do
   echo "$n * $i = `expr $i \* $n`"
   i=`expr $i + 1`
done
```

Save it and try as
```
$ chmod +x nt1
$./nt1 7
```
Above loop can be explained as follows

n=$1	Set the value of command line argument to variable n. (Here it's set to 7)
i=1	Set variable i to 1
while [$i -le 10]	This is our loop condition, here if value of i is less than 10 then, shell execute all statements between do and done
do	Start loop
echo "$n * $i = `expr $i \* $n`"	Print multiplication table as 7 * 1 = 7 7 * 2 = 14 .... 7 * 10 = 70, Here each time value of variable n is multiply be i.
i=`expr $i + 1`	Increment i by 1 and store result to i. (i.e. i=i+1) Caution: If we ignore (remove) this statement than our loop become infinite loop because value of variable i always remain less than 10 and program will only output 7 * 1 = 7 ... ... E (infinite times)
done	Loop stops here if i is not less than 10 i.e. condition of loop is not true. Hence loop is terminated.

From the above discussion not following points about loops
(a) First, the variable used in loop condition must be initialized, Next execution of the loop begins.
(b) A test (condition) is made at the beginning of each iteration.
(c) The body of loop ends with a statement that modifies the value of the test (condition) variable.

(11) *The case Statement*
The case statement is good alternative to Multilevel if-then-else-fi statement. It enable you to match several values against one variable. Its easier to read and write.
Syntax:

```
case  $variable-name  in
     pattern1)   command
                      ...
                      ..
                      command;;
     pattern2)   command
```

```
                        ...
                        ..
                        command;;
        patternN)   command
                        ...
                        ..
                        command;;
        *)              command
                        ...
                        ..
                        command;;
    esac
```

The $variable-name is compared against the patterns until a match is found. The shell then executes all the statements up to the two semicolons that are next to each other. The default is *) and its executed if no match is found. For eg. Create script as follows

```
$ cat > car
#
# if no vehicle name is given
# i.e. -z $1 is defined and it is NULL
#
# if no command line arg

if [ -z $1 ]
then
   rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
# otherwise make first arg as rental
   rental=$1
fi

case $rental in
   "car") echo "For $rental Rs.20 per k/m";;
   "van") echo "For $rental Rs.10 per k/m";;
   "jeep") echo "For $rental Rs.5 per k/m";;
   "bicycle") echo "For $rental 20 paisa per k/m";;
   *) echo "Sorry, I can not gat a $rental for you";;
esac
```

Save it by pressing CTRL+D

```
$ chmod +x car
$ car van
$ car car
$ car Maruti-800
```

Here first we will check, that if $1(first command line argument) is not given set value of rental variable to "*** Unknown vehicle ***",if value given then set it to given value. The $rental is compared against the patterns until a match is found. Here for first run its match with van and it will show output For van Rs.10 per k/m. For second run it print, "For car Rs.20 per k/m". And for last run, there is no match for Maruti-800, hence default i.e. *) is executed and it prints, "Sorry, I can not gat a Maruti-800 for you". Note that esac is always required to indicate end of case statement.

(12)*The read Statement*

Use to get input from keyboard and store them to variable.

Syntax: *read varible1, varible2,...varibleN*
Create script as

```
$ cat > sayH
#
#Script to read your name from key-board
#
echo "Your first name please:"
read fname
echo "Hello $fname, Lets be friend!"
```

Run it as follows

```
$ chmod +x sayH
$ ./sayH
```

This script first ask you your name and then waits to enter name from the user, Then user enters name from keyboard (After giving name you have to press ENTER key) and this entered name through keyboard is stored (assigned) to variable fname.

(13)*Filename Shorthand or meta Characters (i.e. wild cards)*
* or ? or [...] is one of such shorthand character.

* Matches any string or group of characters.
For e.g. $ ls * , will show all files, $ ls a* - will show all files whose first name is starting with letter 'a', $ ls *.c ,will show all files having extension .c $ ls ut*.c, will show all files having extension .c but first two letters of file name must be 'ut'.

? Matches any single character.
For e.g. $ ls ? , will show one single letter file name, $ ls fo? , will show all files whose names are 3 character long and file name begin with fo

[...] Matches any one of the enclosed characters.
For e.g. $ ls [abc]* - will show all files beginning with letters a,b,c
[..-..] A pair of characters separated by a minus sign denotes a range;
For eg. $ ls /bin/[a-c]* - will show all files name beginning with letter a,b or c like

```
/bin/arch          /bin/awk          /bin/bsh     /bin/chmod          /bin/cp
/bin/ash           /bin/basename     /bin/cat     /bin/chown          /bin/cpio
/bin/ash.static    /bin/bash         /bin/chgrp   /bin/consolechars   /bin/csh
```

But

```
$ ls /bin/[!a-o]
$ ls /bin/[^a-o]
```

If the first character following the [is a ! or a ^ then any character not enclosed is matched i.e. do not show us file name that beginning with a,b,c,e...o, like

```
/bin/ps            /bin/rvi          /bin/sleep  /bin/touch     /bin/view
/bin/pwd           /bin/rview        /bin/sort   /bin/true      /bin/wcomp
/bin/red           /bin/sayHello     /bin/stty   /bin/umount    /bin/xconf
/bin/remadmin      /bin/sed          /bin/su     /bin/uname     /bin/ypdomainname
/bin/rm            /bin/setserial    /bin/sync   /bin/userconf  /bin/zcat
/bin/rmdir         /bin/sfxload      /bin/tar    /bin/usleep
/bin/rpm           /bin/sh           /bin/tcsh   /bin/vi
```

(14)*command1;command2*
To run two command with one command line.For eg. $ date;who ,Will print today's date followed

by users who are currently login. Note that You can't use $ date who for same purpose, you must put semicolon in between date and who command.

More Advanced Shell Script Commands

/dev/null - Use to send unwanted output of program

This is special Linux file which is used to send any unwanted output from program/command.
Syntax: *command > /dev/null*
For e.g. $ ls > /dev/null , output of this command is not shown on screen its send to this special file. The /dev directory contains other device files. The files in this directory mostly represent peripheral devices such disks liks floppy disk, sound card, line printers etc.

Local and Global Shell variable (export command)

Normally all our variables are local. Local variable can be used in same shell, if you load another copy of shell (by typing the /bin/bash at the $ prompt) then new shell ignored all old shell's variable. For e.g. Consider following example

```
$ vech=Bus
$ echo $vech
Bus
$ /bin/bash
$ echo $vech

NOTE:-Empty line printed
$ vech=Car
$ echo $vech
Car
$ exit
$ echo $vech
Bus
```

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ /bin/bash	Now load second shell in memory (Which ignores all old shell's variable)
$ echo $vech	Print the contains of variable vech
$ vech=Car	Create new local variable 'vech' with Car as value in second shell
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech (Now you can see first shells variable and its value)

We can copy old shell's variable to new shell (i.e. first shells variable to seconds shell), such variable is know as Global Shell variable. To do this use export command
Syntax: *export variable1, variable2,.....variableN*
For e.g.

```
$ vech=Bus
$ echo $vech
Bus
$ export vech
$ /bin/bash
$ echo $vech
Bus
$ exit
$ echo $vech
```

Bus

Command	Meaning
$ vech=Bus	Create new local variable 'vech' with Bus as value in first shell
$ echo $vech	Print the contains of variable vech
$ export vech	Export first shells variable to second shell
$ /bin/bash	Now load second shell in memory (Old shell's variable is accessed from second shell, *if they are exported*)
$ echo $vech	Print the contains of variable vech
$ exit	Exit from second shell return to first shell
$ echo $vech	Print the contains of variable vech

Conditional execution i.e. && and ||

The control operators are && (read as AND) and || (read as OR). An AND list has the
Syntax: command1 && command2
Here command2 is executed if, and only if, command1 returns an exit status of zero. An OR list has the
Syntax: command1 || command2
Here command2 is executed if and only if command1 returns a non-zero exit status. You can use both as follows
command1 && comamnd2 if exist status is zero || command3 if exit status is non-zero
Here if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed. For e.g.

```
$ rm myf && echo File is removed successfully || echo File is not removed
```

If file (myf) is removed successful (exist status is zero) then "echo File is removed successfully" statement is executed, otherwise "echo File is not removed" statement is executed (since exist status is non-zero)

I/O Redirection and file descriptors

As you know I/O redirectors are used to send output of command to file or to read input from file. (See Input/Output redirection). Now consider following examples

```
$ cat > myf
   This is my file
   ^D
```

Above command send output of cat command to myf file

```
$ cal
```

Above command prints calendar on screen, but if you wish to store this calendar to file then give command

```
$ cal > mycal
```

The cal command send output to mycal file. This is called output redirection

```
$ sort
10
-20
11
2
^D
-20
2
10
11
```

Here sort command takes input from keyboard and then sorts the number, If we wish to take input from file give command as follows

```
$ cat > nos
10
```

```
-20
11
2
^D
$ sort < nos
-20
2
10
11
```

First we have created the file nos, then we have taken input from this file and sort command prints sorted numbers. This is called input redirection. In Linux (And in C programming Language) your keyboard, screen etc are treated as files. Following are name of such files

Standard File	File Descriptors number	Use	Example
stdin	0	as Standard input	Keyboard
stdout	1	as Standard output	Screen
stderr	2	as Standard error	Screen

By default in Linux every program has three files associated with it, (when we start our program these three files are automatically opened by your shell) The use of first two files (i.e. stdin and stdout) , are already seen by us. The last file stderr (numbered as 2) is used by our program to print error on screen. You can redirect the output from a file descriptor directly to file with following

Syntax: file-descriptor-number>filename

For e.g.

```
$ rm bad_file_name111
```

rm: cannot remove `bad_file_name111': No such file or directory ,is the output (error) of the above program. Now if we try to redirect this error-output to file, it can not be send to file

```
$ rm bad_file_name111 > er
```

Still it prints output on stderr as rm: cannot remove `bad_file_name111': No such file or directory, And if you see er file as $ cat er , This file is empty, since output is send to error device and you can not redirect it to copy this error-output to your file 'er'. To overcome this we have to use following command

```
$ rm bad_file_name111 2>er
```

Note that no space are allowed between 2 and >, The 2>er directs the standard error output to file. 2 number is default number of stderr file. Now consider another example, here we are writing shell script as follows

```
$ cat > demoscr
if [ $# -ne 2 ]
then
echo "Error : Number are not supplied"
echo "Usage : $0 number1 number2"
exit 1
fi
ans=`expr $1 + $2`
echo "Sum is $ans"
```

Try it as follows

```
$ chmod +x demoscr
$ ./demoscr
Error : Number are not supplied
Usage : ./demoscr number1 number2
$ ./demoscr > er1
$ ./demoscr 5 7
Sum is 12
```

Here for first sample run , our script prints error message indicating that we have not given two number. For second sample run, we have redirect output of our script to file, since it's error we have to show it to

user, It means we have to print our error message on stderr not on stdout. To overcome this problem replace above echo statements as follows

```
echo "Error : Number are not supplied" 1>&2
echo "Usage : $0 number1 number2" 1>&2
```

Now if you run as

```
$ ./demoscr > er1
Error : Number are not supplied
Usage : ./demoscr number1 number2
```

It will print error message on stderr and not on stdout. The 1>&2 at the end of echo statement, directs the standard output (stdout) to standard error (stderr) device.

Syntax: *from>&destination*

Functions

Function is series of instruction/commands. Function performs particular activity in shell. To define function use following

Syntax:

```
function-name ( )
{
     command1
     command2
     .....
     ...
     commandN
     return
}
```

Where function-name is name of you function, that executes these commands. A return statement will terminate the function. For e.g. Type SayHello() at $ prompt as follows

```
$ SayHello()
{
echo "Hello $LOGNAME, Have nice computing"
return
}
```

Now to execute this SayHello() function just type it name as follows

```
$ SayHello
Hello xxxxx, Have nice computing
```

This way you can call your function. Note that after restarting your computer you will loss this SayHello() function, since its created for that session only. To overcome this problem and to add you own function to automat some of the day today life task, your function to /etc/bashrc file. Note that to add function to this file you must logon as root. Following is the sample /etc/bashrc file with today() function , which is used to print formatted date. First logon as root or if you already logon with your name (your login is not root), and want to move to root account, then you can type following command , when asked for password type root (administrators) password

```
$ su
password:
```

Now open file as (Note your prompt is changed to # from $ to indicate you are root)

```
# vi /etc/bashrc
```

OR

```
# mcedit /etc/bashrc
```

At the end of file add following in /etc/bashrc file

```
#
# today() to print formatted date
#
# To run this function type today at the $ prompt
# Added by Vivek to show function in Linux
#
```

```
today()
{
echo This is a `date +"%A %d in %B of %Y (%r)"`
return
}
```

Save the file and exit it, after all this modification your file may look like as follows

```
# /etc/bashrc

# System wide functions and aliases
# Environment stuff goes in /etc/profile

# For some unknown reason bash refuses to inherit
# PS1 in some circumstances that I can't figure out.
# Putting PS1 here ensures that it gets loaded every time.

PS1="[\u@\h \W]\\$ "

#
# today() to print formatted date
#
# To run this function type today at the $ prompt
# Added by Vivek to show function in Linux
today()
{
echo This is a `date +"%A %d in %B of %Y (%r)"`
return
}
```

To run function first completely logout by typing exit at the $prompt (Or press CTRL + D, Note you may have to type exit (CTRL +D) twice if you login to root account by using su command) ,then login and type $ today , this way today() is available to all user in your system, If you want to add particular function to particular user then open .bashrc file in your home directory as follows

```
# vi .bashrc
```

OR

```
# mcedit .bashrc
```

At the end of file add following in .bashrc file

```
SayBuy()
{
   echo "Buy $LOGNAME ! Life never be the same, until you log again!"
   echo "Press a key to logout. . ."
   read
   return
}
```

Save the file and exit it, after all this modification your file may look like as follows

```
# .bashrc
#
# User specific aliases and functions
# Source global definitions

if [ -f /etc/bashrc ]; then
. /etc/bashrc
fi

SayBuy()
```

```
{
  echo "Buy $LOGNAME ! Life never be the same, until you log again!"
  echo "Press a key to logout. . ."
  read
  return
}
```

To run function first logout by typing exit at the $ prompt (Or press CTRL + D) ,then logon and type $ SayBuy , this way SayBuy() is available to only in your login and not to all user in system, Use .bashrc file in your home directory to add User specific aliases and functions only. (Tip: If you want to show some message or want to perform some action when you logout, Open file .bash_logout in your home directory and add your stuff here For e.g. When ever I logout, I want to show message Buy! Then open your .bash_logout file using text editor such as vi and add statement

```
echo "Buy $LOGNAME, Press a key. . ."
read
```

Save and exit from the file. Then to test this logout from your system by pressing CTRL + D (or type exit) immediately you will see message "Buy xxxxx, Press a key. . .", after pressing key you will be exited.)

User Interface and dialog utility

Good program/shell script must interact with users. There are two ways to this one is use command line to script when you want input, second use statement like echo and read to read input into variable from the prompt. For e.g. Write script as

```
$ cat > userinte
#
# Script to demo echo and read command for user interaction
#
echo "Your good name please :"
read na
echo "Your age please :"
read age
neyr=`expr $age + 1`
echo "Hello $na, next year you will be $neyr yrs old."
```

Save it and run as

```
$ chmod +x userinte
$ ./userinte

Your good name please :
Vivek
Your age please :
25
Hello Vivek, next year you will be 26 yrs old.
```

Even you can create menus to interact with user, first show menu option, then ask user to choose menu item, and take appropriate action according to selected menu item, this technique is show in following script

```
$ cat > menuui
#
# Script to create simple menus and take action according to that selected
# menu item
#
while :
do
    clear
    echo "-----------------------------------"
    echo " Main Menu "
    echo "-----------------------------------"
    echo "[1] Show Todays date/time"
    echo "[2] Show files in current directory"
```

```
    echo "[3] Show calendar"
    echo "[4] Start editor to write letters"
    echo "[5] Exit/Stop"
    echo "========================="
    echo -n "Enter your menu choice [1-5]: "
    read yourch
  case $yourch in
      1) echo "Today is `date` , press a key. . ." ; read ;;
      2) echo "Files in `pwd`" ; ls -l ; echo "Press a key. . ." ; read ;;
      3) cal ; echo "Press a key. . ." ; read ;;
      4) vi ;;
      5) exit 0 ;;
      *) echo "Opps!!! Please select choice 1,2,3,4, or 5";
        echo "Press a key. . ." ; read ;;
  esca

done
```

Above all statement explained in following table

Statement	Explanation
while :	Start infinite loop, this loop will only break if you select 5 (i.e. Exit/Stop menu item) as your menu choice
do	Start loop
clear	Clear the screen, each and every time
echo "----------" **echo " Main Menu "** **echo "----------"** **echo "[1] Show Todays date/time"** **echo "[2] Show files in current directory"** **echo "[3] Show calendar"** **echo "[4] Start editor to write letters"** **echo "[5] Exit/Stop"** **echo "=========="**	Show menu on screen with menu items
echo -n "Enter your menu choice [1-5]: "	Ask user to enter menu item number
read yourch	Read menu item number from user
case $yourch in **1) echo "Today is `date` , press a key. . ." ; read ;;** **2) echo "Files in `pwd`" ; ls -l ;** **echo "Press a key. . ." ; read ;;** **3) cal ; echo "Press a key. . ." ; read ;;** **4) vi ;;** **5) exit 0 ;;** ***) echo "Opps!!! Please select choice 1,2,3,4, or 5";** **echo "Press a key. . ." ; read ;;** **esca**	Take appropriate action according to selected menu item, If menu item is not between 1 - 5, then show error and ask user to input number between 1-5 again
done	Stop loop , if menu item number is 5 (i.e. Exit/Stop)

User interface usually includes, menus, different type of boxes like info box, message box, Input box etc. In Linux shell there is no built-in facility available to create such user interface, But there is one utility supplied with Red Hat Linux version 6.0 called dialog, which is used to create different type of boxes like info box, message box, menu box, Input box etc. Now try dialog utility as follows :

```
$ cat > dia1
dialog --title "Linux Dialog Utility Infobox" --backtitle "Linux Shell Script\
Tutorial" --infobox "This is dialog box called infobox, which is used\
to show some information on screen, Thanks to Savio Lam and\
Stuart Herbert to give us this utility. Press any key. . . " 7 50 ; read
```

Save the shell script and run as

```
$ chmod +x dia1
$ ./dia1
```

After executing this dialog statement you will see box on screen with titled as "Welcome to Linux Dialog Utility" and message "This is dialog....Press any key. . ." inside this box. The title of box is specified by --title option and info box with --infobox "Message" with this option. Here 7 and 50 are height-of-box and width-of-box respectively. "Linux Shell Script Tutorial" is the backtitle of dialog show on upper left side of screen and below that line is drawn. Use dialog utility to Display dialog boxes from shell scripts.

Syntax:

```
dialog --title {title} --backtitle {backtitle} {Box options}
where Box options can be any one of following
--yesno      {text}  {height} {width}
--msgbox     {text}  {height} {width}
--infobox    {text}  {height} {width}
--inputbox   {text}  {height} {width} [{init}]
--textbox    {file}  {height} {width}
--menu       {text}  {height} {width} {menu} {height} {tag1} item1}...
```

msgbox using dialog utility

```
$cat > dia2
dialog --title "Linux Dialog Utility Msgbox" --backtitle "Linux Shell Script\
Tutorial" --msgbox "This is dialog box called msgbox, which is used\
to show some information on screen which has also Ok button, Thanks to Savio Lam\
and Stuart Herbert to give us this utility. Press any key. . . " 9 50
```

Save it and run as

```
$ chmod +x dia2
$ ./dia2
```

yesno box using dialog utility

```
$ cat > dia3
dialog --title "Alert : Delete File" --backtitle "Linux Shell Script\
Tutorial" --yesno "\nDo you want to delete '/usr/letters/jobapplication'\
file" 7 60
sel=$?
case $sel in
   0) echo "You select to delete file";;
   1) echo "You select not to delete file";;
   255) echo "Canceled by you by pressing [ESC] key";;
esac
```

Save it and run as

```
$ chmod +x dia3
$ ./dia3
```

Above script creates yesno type dialog box, which is used to ask some questions to the user , and answer to those question either yes or no. After asking question how do we know, whether user has press yes or no button ? The answer is exit status, if user press yes button exit status will be zero, if user press no button exit status will be one and if user press Escape key to cancel dialog box exit status will be one 255. That is what we have tested in our above shell as

Statement	Meaning
sel=$?	Get exit status of dialog utility
case $sel in 0) echo "You select to delete file";; 1) echo "You select not to delete file";; 255) echo "Canceled by you by pressing [Escape] key";; esac	Now take action according to exit status of dialog utility, if exit status is 0 , delete file, if exit status is 1 do not delete file and if exit status is 255, means Escape key is pressed.

inputbox using dialog utility

```
$ cat > dia4
dialog --title "Inputbox - To take input from you" --backtitle "Linux Shell\
Script Tutorial" --inputbox "Enter your name please" 8 60 2>/tmp/input.$$

sel=$?
na=`cat /tmp/input.$$`
case $sel in
   0) echo "Hello $na" ;;
   1) echo "Cancel is Press" ;;
   255) echo "[ESCAPE] key pressed" ;;
esac
rm -f /tmp/input.$$
```

Inputbox is used to take input from user, Here we are taking Name of user as input. But where we are going to store inputted name, the answer is to redirect inputted name to file via statement 2>/tmp/input.$$ at the end of dialog command, which means send screen output to file called /tmp/input.$$, letter we can retrieve this inputted name and store to variable as follows na=`cat /tmp/input.$$`. For inputbox exit status is as follows

Exit Status for Inputbox	Meaning
0	Command is successful
1	Cancel button is pressed by user
255	Escape key is pressed by user

Now we will write script to create menus using dialog utility, following are menu items
Date/time
Calendar
Editor
and action for each menu-item is follows

```
------------------------------------------------
MENU-ITEM          ACTION
------------------------------------------------
Date/time          Show current date/time
Calendar           Show calendar
Editor             Start vi Editor
------------------------------------------------
```

Create script as follows

```
$ cat > smenu
#
#How to create small menu using dialog
#
dialog --backtitle "Linux Shell Script Tutorial " --title "Main\
Menu" --menu "Move using [UP] [DOWN],[Enter] to\
```

```
Select" 15 50 3\
Date/time     "Shows Date and Time"\
Calendar      "To see calendar "\
Editor        "To start vi editor " 2>/tmp/menuitem.$$

menuitem=`cat /tmp/menuitem.$$`

opt=$?

case $menuitem in
   Date/time) date;;
   Calendar) cal;;
   Editor) vi;;
esac

rm -f /tmp/menuitem.$$
```

Save it and run as

```
$ chmod +x smenu
$ ./smenu
```

Here --menu option is used of dialog utility to create menus, menu option take

--menu options	Meaning
"Move using [UP] [DOWN],[Enter] to Select"	This is text show before menu
15	Height of box
50	Width of box
3	Height of menu
Date/time "Shows Date and Time"	First menu item called as *tag1* (i.e. **Date/time**) and description for menu item called as *item1* (i.e. **"Shows Date and Time")**
Calendar "To see calendar "	First menu item called as *tag2* (i.e. **Calendar**) and description for menu item called as *item2* (i.e. **"To see calendar")**
Editor "To start vi editor "	First menu item called as *tag3* (i.e. **Editor**) and description for menu item called as *item3* (i.e.**"To start vi editor")**
2>/tmp/menuitem.$$	Send sleeted menu item (tag) to this temporary file

After creating menus, user selects menu-item by pressing enter key the selected choice is redirected to temporary file, Next this menu-item is retrieved from temporary file and following case statement compare the menu-item and takes appropriate step according to selected menu item. As you see, dialog utility allows more powerful user interaction then the older read and echo statement. The only problem with dialog utility is it work slowly.

trap command

Now consider following script

```
$ cat > testsign
ls -R /
```

Save and run it as

```
$ chmod +x testsign
$ ./testsign
```

Now if you press ctrl + c , while running this script, script get terminated. The ctrl + c here work as signal, When such signal occurs its send to all process currently running in your system. Now consider following shell script

```
$ cat > testsign1
#
# Why to trap signal, version 1
#
Take_input1()
{
 recno=0
 clear
 echo "Appointment Note keeper Application for Linux"
 echo -n "Enter your database file name : "
 read filename

if [ ! -f $filename ]; then
  echo "Sorry, $filename does not exit, Creating $filename database"
  echo "Appointment Note keeper Application database file" > $filename
fi

echo "Data entry start data: `date`" >/tmp/input0.$$
#
# Set a infinite loop
#
while :
do
    echo -n "Appointment Title:"
    read na
    echo -n "Appoint time :"
    read ti
    echo -n "Any Remark :"
    read remark
    echo -n "Is data okay (y/n) ?"
    read ans

 if [ $ans = y -o $ans = Y ]; then
    recno=`expr $recno + 1`
    echo "$recno. $na $ti $remark" >> /tmp/input0.$$
 fi

   echo -n "Add next appointment (y/n)?"
    read isnext

  if [ $isnext = n -o $isnext = N ]; then
   cat /tmp/input0.$$ >> $filename
   rm -f /tmp/input0.$$
   return  # terminate loop
 fi
 done
}

#
#
# Call our user define function : Take_input1
#
Take_input1
```

Save it and run as

```
$ chmod +x testsign1
$ ./testsign1
```

It first ask you main database file where all appointment of that day is stored, if no such database file found, file is created, after that it open one temporary file in /tmp directory, and puts today's date in that

file. Then one infinite loop begins, which ask appointment title, time and remark, if this information is correct its written to temporary file, After that script ask user , whether he/she wants add next appointment record, if yes then next record is added , otherwise all records are copied from temporary file to database file and then loop will be terminated. You can view your database file by using cat command. Now problem is that while running this script, if you press CTRL + C, your shell script gets terminated and temporary file are left in /tmp directory. For e.g. try as follows

```
$./testsign1
```

After given database file name and after adding at least one appointment record to temporary file press CTRL+C, Our script get terminated, and it left temporary file in /tmp directory, you can check this by giving command as follows

```
$ ls /tmp/input*
```

Our script needs to detect when such signal (event) occurs, To achieve this we have to first detect Signal using trap command

Syntax: *trap {commands} {signal number list}*

Signal Number	When occurs
0	shell exit
1	hangup
2	interrupt (CTRL+C)
3	quit
9	kill (cannot be caught)

To catch this signal in above script, put *trap* statement before calling Take_input1 function as `trap del_file 2 .`, Here trap command called del_file() when 2 number interrupt (i.e.CTRL+C) occurs. Open above script in editor and modify it so that at the end it will look like as follows

```
$ vi testsign1
```

or

```
$ mcedit testsign1
#
# signal is trapped to delete temporary file , version 2
#
del_file()
{
   echo "* * * CTRL + C Trap Occurs (removing temporary file)* * *"
   rm -f /tmp/input0.$$
   exit 1
}

Take_input1()
{
 recno=0
 clear
 echo "Appointment Note keeper Application for Linux"
 echo -n "Enter your database file name : "
 read filename

if [ ! -f $filename ]; then
   echo "Sorry, $filename does not exit, Creating $filename database"
   echo "Appointment Note keeper Application database file" > $filename
fi

echo "Data entry start data: `date`" >/tmp/input0.$$
#
# Set a infinite loop
#
while :
do
```

```
    echo -n "Appointment Title:"
    read na
    echo -n "Appoint time :"
    read ti
    echo -n "Any Remark :"
    read remark
    echo -n "Is data okay (y/n) ?"
    read ans

 if [ $ans = y -o $ans = Y ]; then
    recno=`expr $recno + 1`
    echo "$recno. $na $ti $remark" >> /tmp/input0.$$
 fi

   echo -n "Add next appointment (y/n)?"
    read isnext

  if [ $isnext = n -o $isnext = N ]; then
   cat /tmp/input0.$$ >> $filename
   rm -f /tmp/input0.$$
   return  # terminate loop
 fi
 done
}

#
# Set trap to for CTRL+C interrupt,
# When occurs it first it calls del_file() and then exit
#
trap  del_file  2

#
# Call our user define function : Take_input1
#
Take_input1
```

Now save it run the program as
$ ./testsign1
After given database file name and after giving appointment title press CTRL+C, Here we have already captured this CTRL + C signal (interrupt), so first our function del_file() is called, in which it gives message as "** * CTRL + C Trap Occurs (removing temporary file)* * * "* and then it remove our temporary file and then exit with exit status 1. Now check /tmp directory as follows
$ ls /tmp/input*
Now Shell will report no such temporary file exit.

getopts command

This command is used to check valid command line argument passed to script. Usually used in while loop.
Syntax: *getopts {optsring} {variable1}*

getopts is used by shell to parse command line argument. optstring contains the option letters to be recognized; if a letter is followed by a colon, the option is expected to have an argument, which should be separated from it by white space. Each time it is invoked, getopts places the next option in the shell variable variable1, When an option requires an argument, getopts places that argument into the variable OPTARG. On errors getopts diagnostic messages are printed when illegal options or missing option arguments are encountered. If an illegal option is seen, getopts places ? into variable1. For e.g. We have script called ani which has syntax as
ani -n -a -s -w -d
Options: These are optional argument
-n name of animal

```
        -a age of animal
        -s sex of animal
        -w weight of animal
        -d demo values (if any of the above options are used
           their values are not taken)
$ cat > ani
#
#  Usage: ani   -n -a -s -w -d
#

#
# help_ani() To print help
#
help_ani()
{
   echo "Usage: $0  -n -a -s -w -d"
   echo "Options: These are optional argument"
   echo "     -n name of animal"
   echo "     -a age of animal"
   echo "     -s sex of animal "
   echo "     -w weight of animal"
   echo "     -d demo values (if any of the above options are used "
   echo "       their values are not taken)"
   exit 1
}
#
#Start main procedure
#

#
#Set default value for variable
#
isdef=0

na=Moti
age="2 Months"
sex=Male
weight=3Kg

#
#if no argument
#
if [ $# -lt 1 ]; then
   help_ani
fi

while getopts n:a:s:w:d  opt
do
     case "$opt" in
     n) na="$OPTARG";;
      a) age="$OPTARG";;
     s) sex="$OPTARG";;
     w) weight="$OPTARG";;
     d) isdef=1;;
    \?) help_ani;;
   esac
done
```

```
if [ $isdef -eq 0 ]
then
    echo "Animal Name: $na, Age: $age, Sex: $sex, Weight: $weight (user define mode)"
else
   na="Pluto Dog"
   age=3
   sex=Male
   weight=20kg
   echo "Animal Name: $na, Age: $age, Sex: $sex, Weight: $weight (demo mode)"
fi
```

Save it and run as follows

```
$ chmod +x ani
$ ani -n Lassie -a 4 -s Female -w 20Kg
$ ani -a 4 -s Female -n Lassie -w 20Kg
$ ani -n Lassie -s Female -w 20Kg -a 4
$ ani -w 20Kg -s Female -n Lassie -a 4
$ ani -w 20Kg -s Female
$ ani -n Lassie -a 4
$ ani -n Lassie
$ ani -a 2
```

See because of getopts, we can pass command line argument in different style. Following are invalid options for ani script

```
$ ani -nLassie -a4 -sFemal -w20Kg
```

Here no space between option and their value.

```
$ ani -nLassie-a4-sFemal-w20Kg
$ ani -n Lassie -a 4 -s Female -w 20Kg -c Mammal
```

Here -c is not one of the options.

More examples of Shell Script (Exercise for You :-)

First try to write this shell script, as exercise, if any problem or for sample answer to this Shell script open the shell script file supplied with this tutorial.

Q.1. How to write shell script that will add two nos, which are supplied as command line argument, and if this two nos are not given show error and its usage
Answer: See Q1 shell Script.

Q.2.Write Script to find out biggest number from given three nos. Nos are supplies as command line argument. Print error if sufficient arguments are not supplied.
Answer: See Q2 shell Script.

Q.3.Write script to print nos as 5,4,3,2,1 using while loop.
Answer: See Q3 shell Script.

Q.4. Write Script, using case statement to perform basic math operation as follows

```
+ addition
- subtraction
x multiplication
/ division
```

The name of script must be 'q4' which works as follows
$ ./q4 20 / 3, Also check for sufficient command line arguments
Answer: See Q4 shell Script.

Q.5.Write Script to see current date, time, username, and current directory
Answer: See Q5 shell Script.

Q.6.Write script to print given number in reverse order, for eg. If no is 123 it must print as 321.

Answer: See Q6 shell Script.

Q.7.Write script to print given numbers sum of all digit, For eg. If no is 123 it's sum of all digit will be 1+2+3 = 6.
Answer: See Q7 shell Script.

Q.8.How to perform real number (number with decimal point) calculation in Linux
Answer: Use Linux's bc command

Q.9.How to calculate 5.12 + 2.5 real number calculation at $ prompt in Shell ?
Answer: Use command as , $ echo 5.12 + 2.5 | bc , here we are giving echo commands output to bc to calculate the 5.12 + 2.5

Q.10.How to perform real number calculation in shell script and store result to third variable , lets say a=5.66, b=8.67, c=a+b?
Answer: See Q10 shell Script.

Q.11.Write script to determine whether given file exist or not, file name is supplied as command line argument, also check for sufficient number of command line argument
Answer: See Q11 shell Script.

Q.12.Write script to determine whether given command line argument ($1) contains "*" symbol or not, if $1 does not contains "*" symbol add it to $1, otherwise show message "Symbol is not required". For e.g. If we called this script Q12 then after giving ,
$ Q12 /bin
Here $1 is /bin, it should check whether "*" symbol is present or not if not it should print Required i.e. /bin/*, and if symbol present then Symbol is not required must be printed. Test your script as
$ Q12 /bin
$ Q12 /bin/*
Answer: See Q12 shell Script

Q.13. Write script to print contains of file from given line number to next given number of lines. For e.g. If we called this script as Q13 and run as
$ Q13 5 5 myf , Here print contains of 'myf' file from line number 5 to next 5 line of that file.
Answer: See Q13 shell Script

Q.14. Write script to implement getopts statement, your script should understand following command line argument called this script Q14,
Q14 -c -d -m -e
Where options work as
-c clear the screen
-d show list of files in current working directory
-m start mc (midnight commander shell) , if installed
-e { editor } start this { editor } if installed
Answer: See Q14 shell Script

Q.15. Write script called sayHello, put this script into your startup file called .bash_profile, the script should run as soon as you logon to system, and it print any one of the following message in infobox using dialog utility, if installed in your system, If dialog utility is not installed then use echo statement to print message : -
Good Morning
Good Afternoon
Good Evening , according to system time.
Answer: See Q15 shell Script

Q.16. How to write script, that will print, Message "Hello World" , in Bold and Blink effect, and in different colors like red, brown etc using echo command.
Answer: See Q16 shell Script

Q.17. Write script to implement background process that will continually print current time in upper right

corner of the screen , while user can do his/her normal job at $ prompt.
Answer: See Q17 shell Script.

Q.18. Write shell script to implement menus using dialog utility. Menu-items and action according to select menu-item is as follows

Menu-Item	Purpose	Action for Menu-Item
Date/time	To see current date time	Date and time must be shown using infobox of dialog utility
Calendar	To see current calendar	Calendar must be shown using infobox of dialog utility
Delete	To delete selected file	First ask user name of directory where all files are present, if no name of directory given assumes current directory, then show all files only of that directory, Files must be shown on screen using menus of dialog utility, let the user select the file, then ask the confirmation to user whether he/she wants to delete selected file, if answer is yes then delete the file , report errors if any while deleting file to user.
Exit	To Exit this shell script	Exit/Stops the menu driven program i.e. this script

Note: Create function for all action for e.g. To show date/time on screen create function show_datetime().
Answer: See Q18 shell Script.

Q.19. Write shell script to show various system configuration like
1) Currently logged user and his logname
2) Your current shell
3) Your home directory
4) Your operating system type
5) Your current path setting
6) Your current working directory
7) Show Currently logged number of users
8) About your os and version ,release number , kernel version
9) Show all available shells
10) Show mouse settings
11) Show computer cpu information like processor type, speed etc
12) Show memory information
13) Show hard disk information like size of hard-disk, cache memory, model etc
14) File system (Mounted)
Answer: See Q19 shell Script.

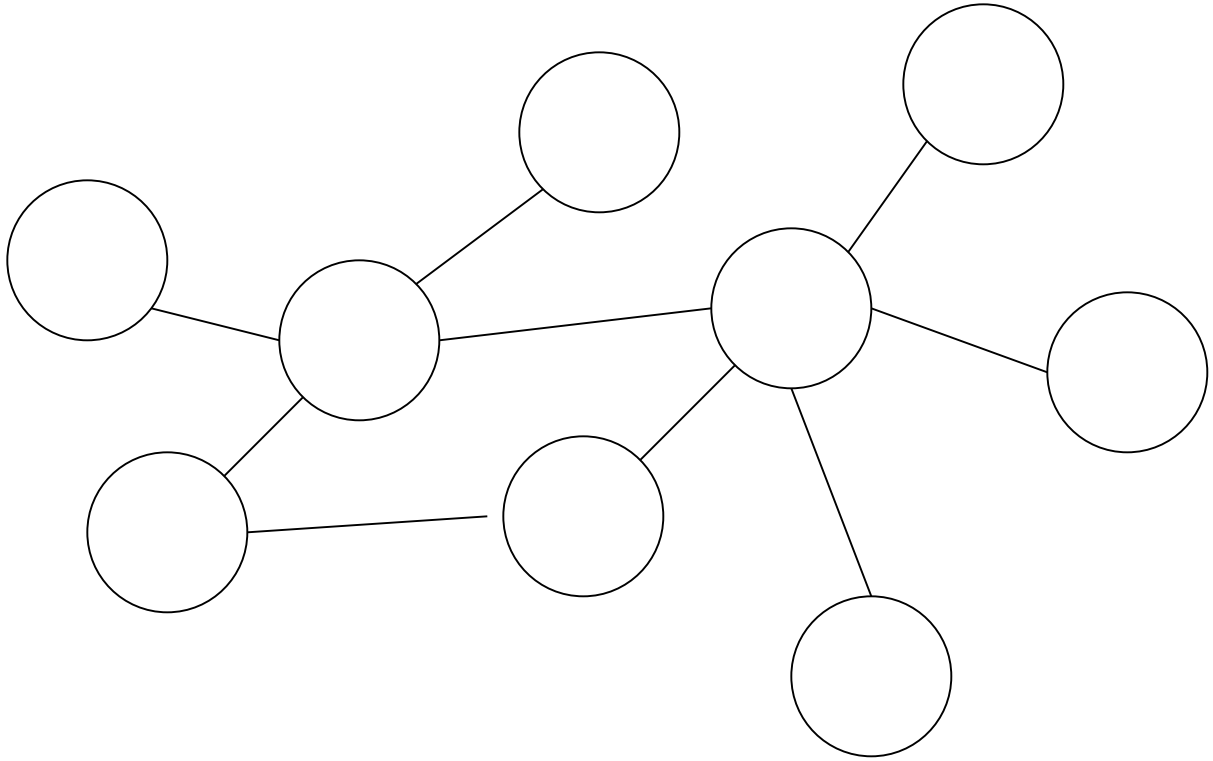
That's all!, Thanks for reading, This tutorial ends here, but Linux programming environment is big, rich and productive (As you see from above shell script exercise), you should continue to read more advance topics and books. If you have any suggestion or new ideas or problem with this tutorial, please feel free to contact me.
My e-mail is gite-vivek@usa.net.

Networking Fundamentals

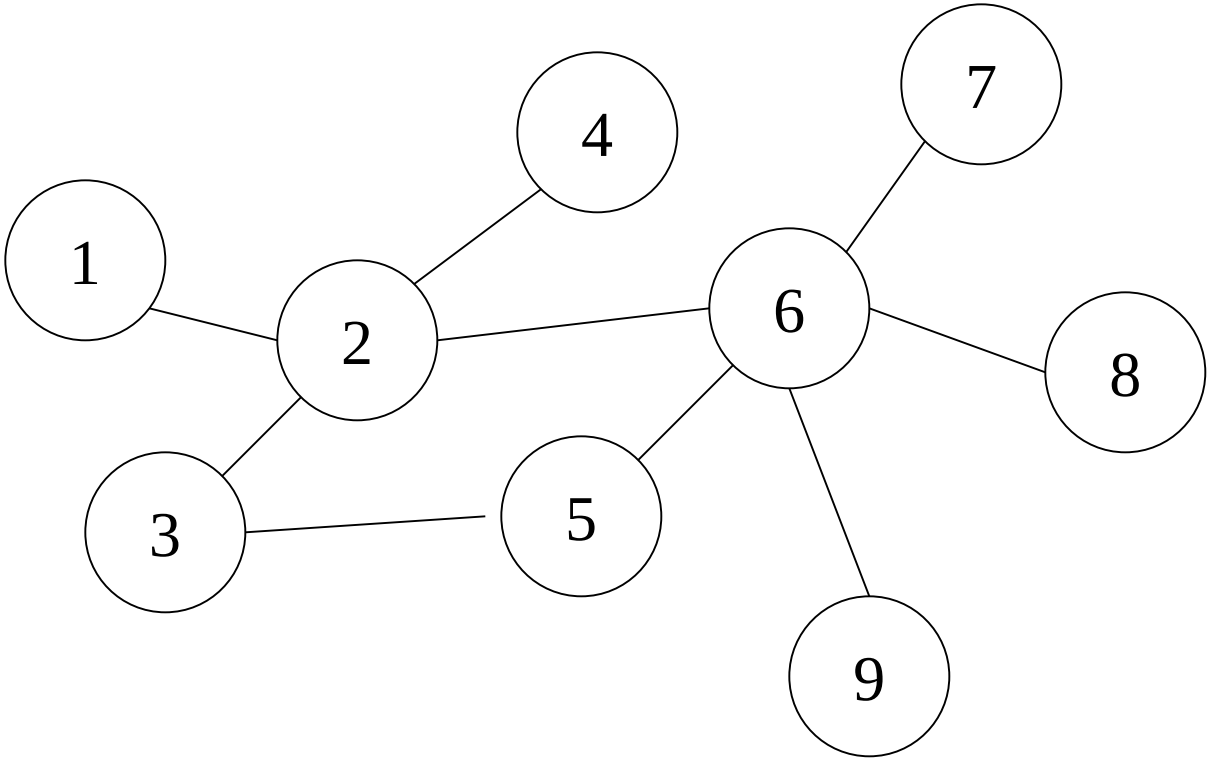
Basics

- Network – collection of nodes and links that cooperate for communication
- Nodes – computer systems
  - Internal (routers, bridges, switches)
  - Terminal (workstations)
- Links – connections for transmitting data
- Protocol – standards for formatting and interpreting data and control information

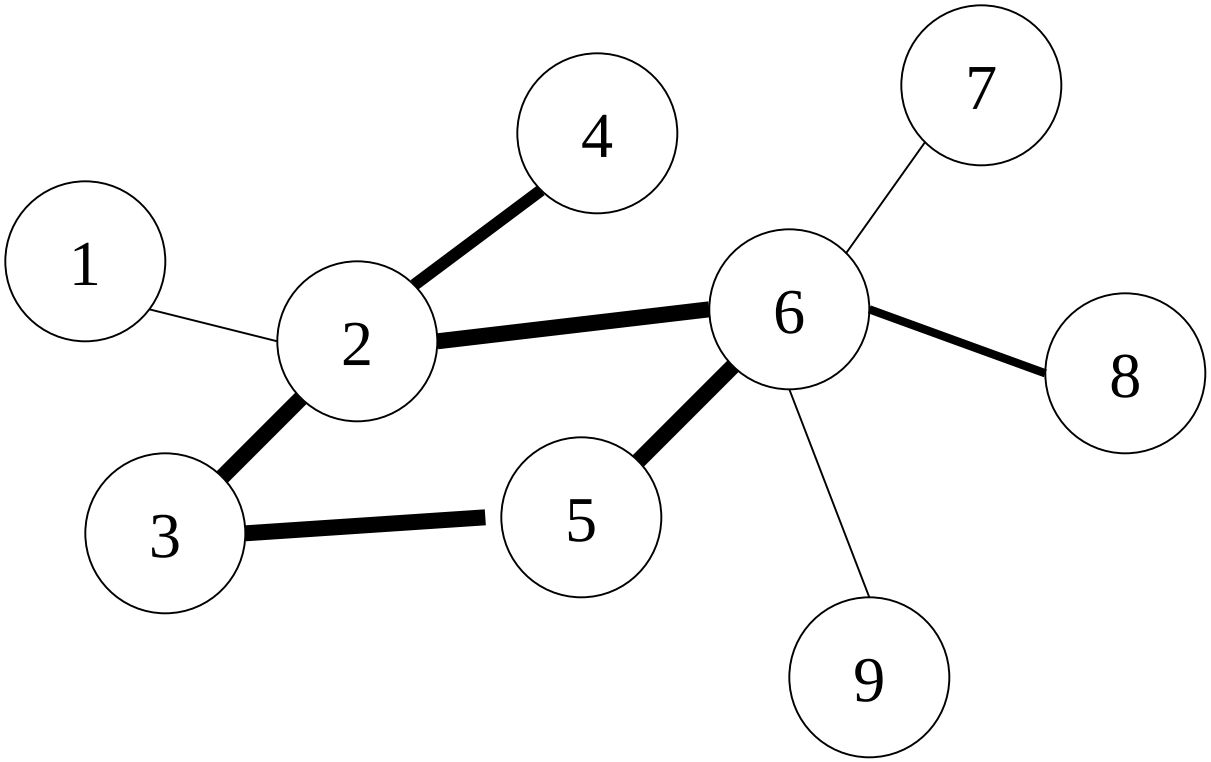
Network = {nodes and links}



Nodes have addresses



Links have bandwidths and latencies



Wires aren’t perfect

- Attenuation (resistance)
  - degrades quality of signal
- Delay (speed of light * 2/3)
  - speed of light:
    - 8ms RTT coast-to-coast
    - 8 minutes to the sun
- Noise (microwaves and such)
- Nodes aren’t perfect either
  - Unreliability is pervasive!

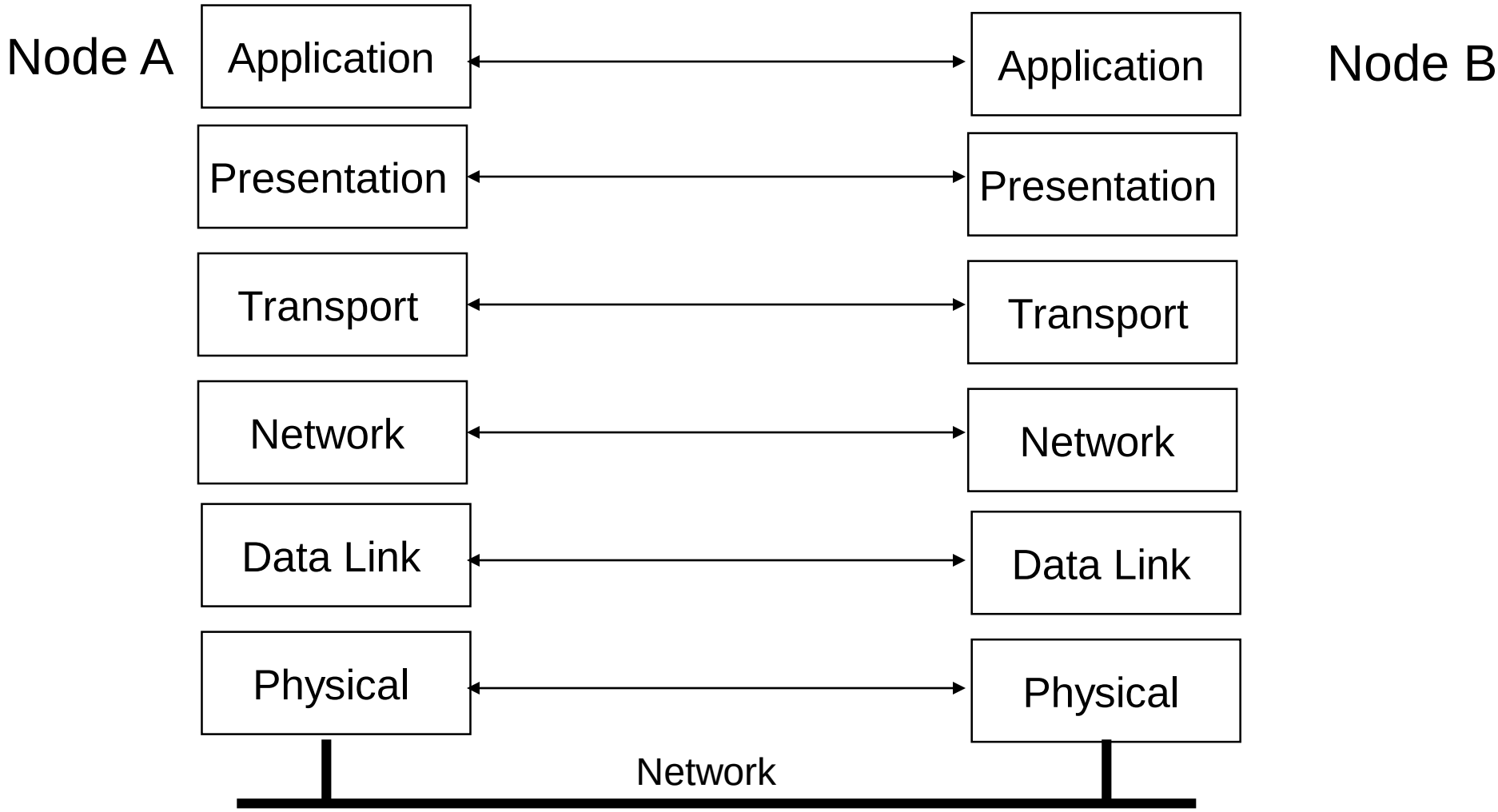
Getting Data Across (imperfect wires)

- Split up big files into small pieces
  - the pieces are called packets
- Each packet (~1500 bytes) is sent separately
  - packets can be corrupted
    - noise, bugs
  - packets can be dropped
    - corrupted, overloaded nodes
  - packets can be reordered
    - retransmission + different paths
- Allows packets from different flows to be multiplexed along the same link

Layers

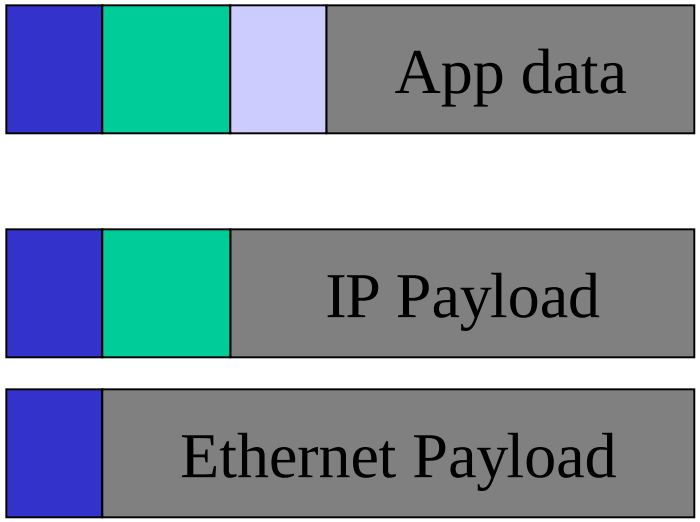
- Each layer abstracts the services of various lower layers, providing a uniform interface to higher layers.
- Each layer needs to know:
  - How to interpret a packet's payload
    - e.g., protocol numbers
  - How to use the services of a lower layer

OSI Levels


Node A
Application
Presentation
Transport
Network
Data Link
Physical
Application
Presentation
Transport
Network
Data Link
Physical
Node B
Network

Layers

OSI Reference	Reality
Application	HTTP
Presentation	
Session	
Transport	TCP
Network	IP
Data-Link	Ethernet
Physical	Twisted Pair



The Internet Protocol (IP)

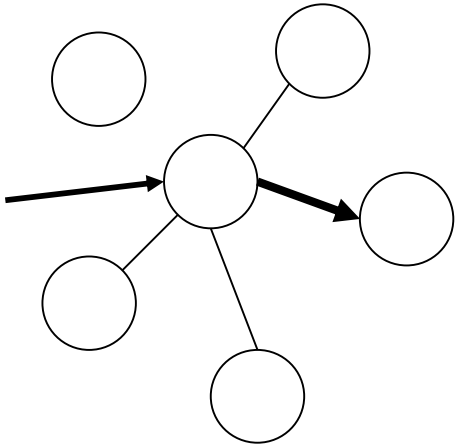
- Connects disparate networks
  - Single (hierarchical) address space
  - Single network header
- Assumes data link is unreliable
- Provides unreliable service
  - Loss: A B D E
  - Duplication: A B B C D E
  - Corruption: A Q C D E
  - Reordering: A C D B E

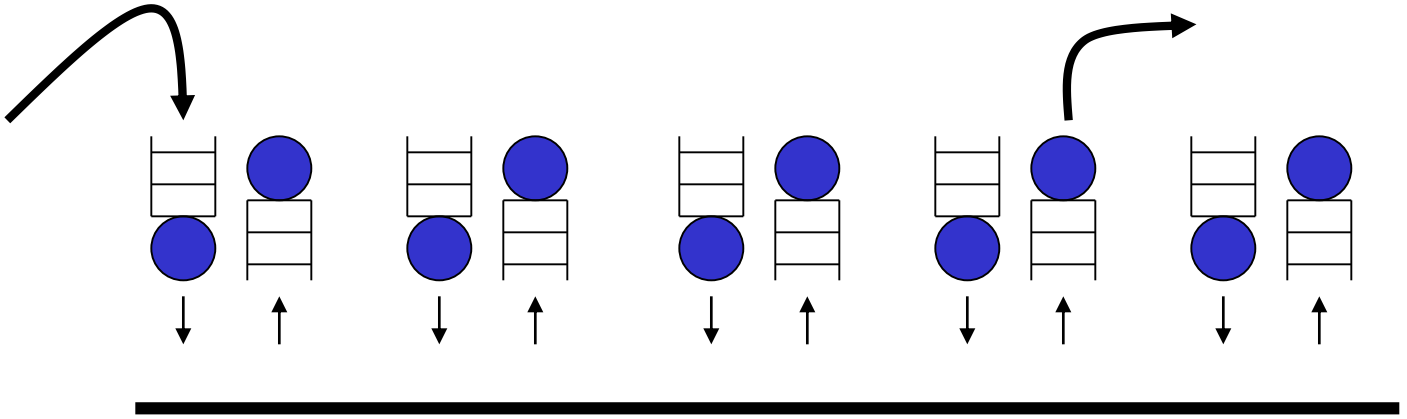
IP Addresses

- 32 bits long, split into 4 octets:
  - For example, 128.95.2.24
- Hierarchical:
  - First bits describe which network
  - Last bits describe which host on the network
- UW subnets include:
  - 128.95, 140.142…
- UW CSE subnets include:
  - 128.95.2, 128.95.4, 128.95.219…

Packet Forwarding

- Buffer incoming packets
- Decide which output link
- Buffer outgoing packets
- Send packet

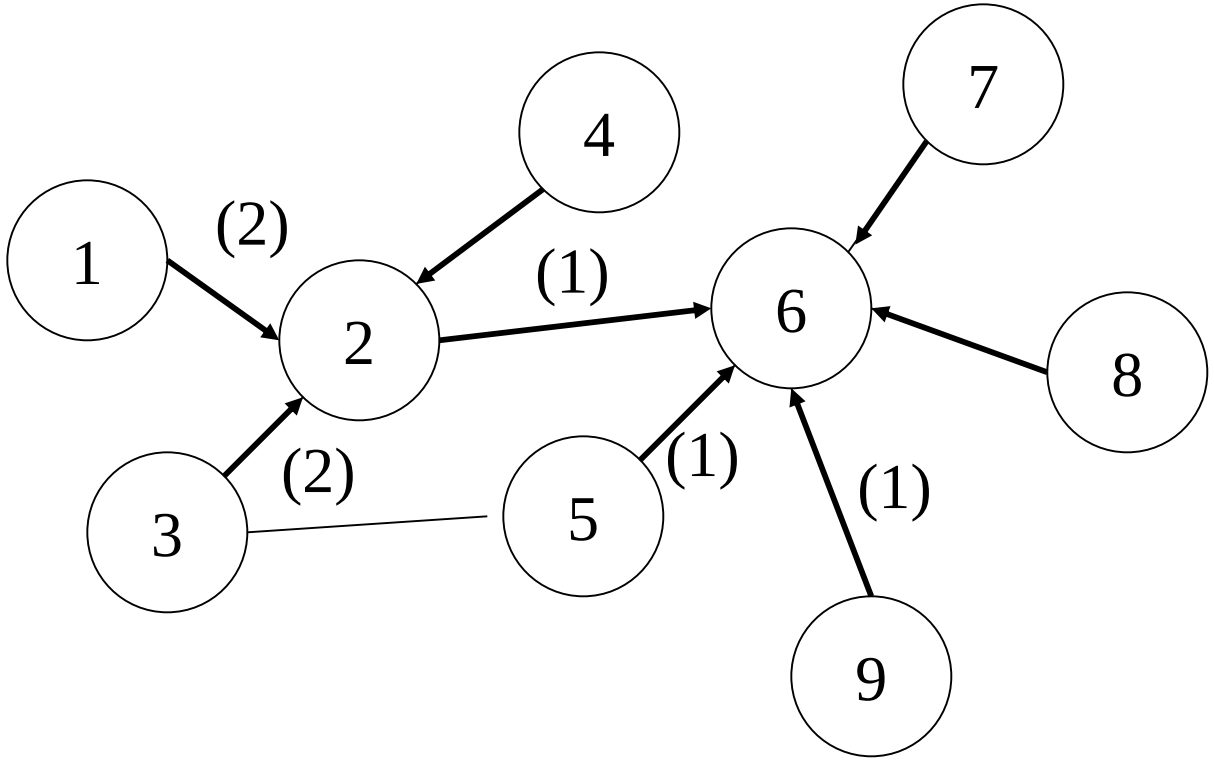




Routing

- How do nodes determine which output link to use to reach a destination?
- Distributed algorithm for converging on shortest path tree
- Nodes exchange reachability information:
  - "I can get to 128.95.2.x in 3 hops"

Shortest path tree



(x) Is the cost to get to 6. The metric (cost per link) here is 1.
Simple algorithm: 6 broadcasts “I’m alive” to neighbors.
Neighbors send “I can get to 6 in 1 hop”, etc.

Route Aggregation

- What hierarchical addressing is good for.
- UW routers can advertise 128.95.x.x
  - instead of 128.95.2.x, 128.95.3,x, …
- Other routers don’t need forwarding table entries for each host in the network.

Routing Reality

- Routing in the Internet connects Autonomous Systems (AS's)
  - AT&T, Sprint, UUNet, BBN…
- Shortest path, sort of… money talks.
  - actually a horrible mess
  - nobody really knows what's going on
  - get high $$ job if you are a network engineer that messes with this stuff

TCP Service Model

- Provide reliability, ordering on the unreliable, unordered IP
- Bytestream oriented: when you send data using TCP, you think about bytes, not about packets.

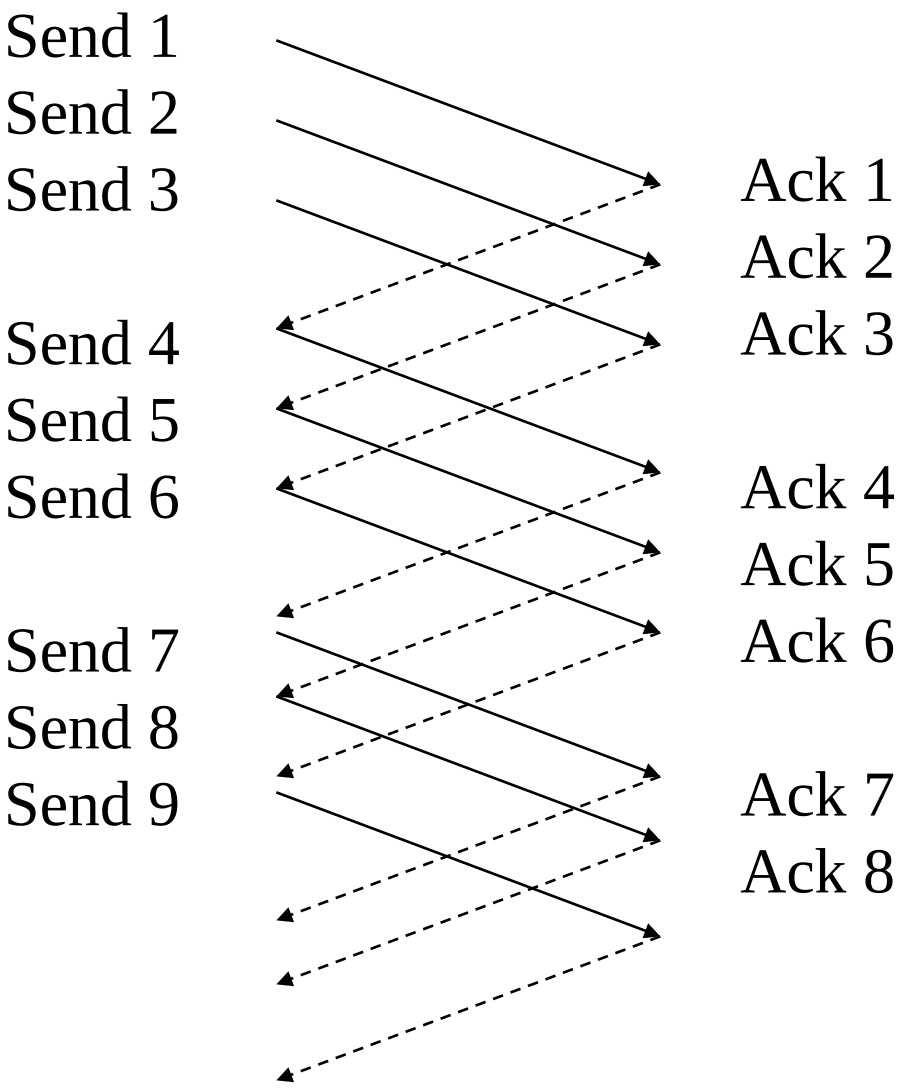
TCP Ports

- Connections are identified by the tuple:
  - IP source address
  - IP destination address
  - TCP source port
  - TCP destination port
- Allows multiple connections; multiple application protocols, between the same machines
- Well known ports for some applications: (web: 80, telnet: 23, mail:25, dns: 53)

TCP's Sliding Window

- Simple reliability:
  - Send one packet, wait for acknowledgment, then send the next…
- Better performance:
  - Keep several unacknowledged packets in the network (a window)

Sliding Window Example



- Window size = 3
- Can send up to three packets into the network at a time.
- Each packet has a sequence number for ordering

TCP's Congestion Control

- How big should the window be?
- Performance is limited by:
  - (window size) / round trip time
  - Performance of bottleneck link (modem?)
- If window is too small, performance is wasted.
- If window is too big, may overflow network buffers, causing packet loss.

Steps for a web access

- Name lookup
  - Client to local DNS server
  - Local DNS may return a cached binding, or lookup the name for itself
- TCP Connection setup
  - Client to remote IP, port 80
- Send HTTP request
  - “GET /index.html”
- Receive HTTP response
  - “blah blah blah” maybe several packets
- TCP Connection teardown

HTTP 1.1

- Incremental improvements
- "Persistent connections" allow multiple requests over the same connection
  - Web transfers are often small
  - Avoid connection setup and teardown overhead
  - TCP is better the longer you use it: it learns how fast to send to get best performance without overflowing buffers.

UNC
INFORMATION
TECHNOLOGY SERVICES

Shell Scripting

Santosh Kumar
Assistant Professor
AI&DS Department, VIIT, Pune
santosh.kumar@viit.ac.in

Contents

- Introduction
  - UNIX/LINUX and Shell
  - UNIX Commands and Utilities
  - Basic Shell Scripting Structure
- Shell Programming
  - Variable
  - Operators
  - Logic Structures
- Examples of Application in Cloud & Devops
- Hands-on Exercises

Why Shell Scripting ?

- Shell scripts can be used to prepare input files, job monitoring, and output processing.
- Useful to create own commands.
- Save lots of time on file processing.
- To automate some task of day to day life.
- System Administration part can be also automated.

Objectives & Prerequisites

- **After this workshop, you should be:**
  - Familiar with UNIX/LINUX, Borne Shell, shell variables/operators
  - Able to write simple shell scripts to illustrate programming logic
  - Able to write scripts for Cloud computing purposes
- We assume that you have/know
  - An account on the Emerald cluster
  - Basic knowledge of UNIX/LINUX and commands
  - UNIX editor e.g. vi or emacs

History of UNIX/Linux

- Unix is a command line operating system developed around 1969 in the Bell Labs
- Originally written using C
- Unix is designed so that users can extend the functionality
  - To build new tools easily and efficiently
  - To customize the shell and user interface.
  - To string together a series of Unix commands to create new functionality.
  - To create custom commands that do exactly what we want.
- Around 1990 Linus Torvalds of Helsinki University started off a freely available academic version of Unix
- Linux is the Antidote to a Microsoft dominated future

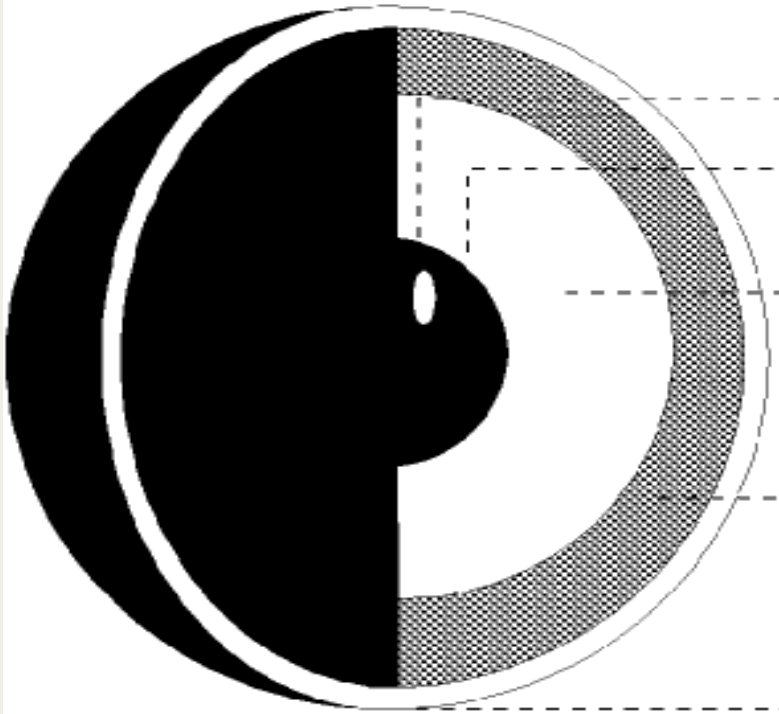
What is UNIX/Linux ?

Simply put

- Multi-Tasking O/S
- Multi-User O/S
- Available on a range of Computers

SunOS	Sun Microsystems
IRIX	Silicon Graphics
HP-UX	Hewlett Packard
AIX	IBM
Linux	….

UNIX/LINUX Architecture



CPU

KERNEL

Resource allocation

SHELL

Command
Language interpreter
Interface
to other OS parts

PROGRAMS

List Processors
Text processors
Programmer's tools
Programming languages
Networks and communications

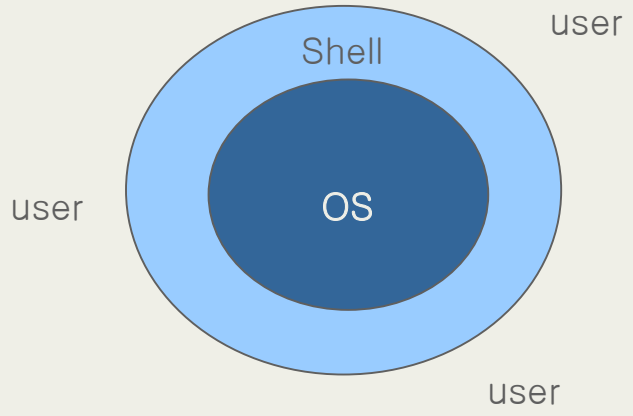
APPLICATIONS

What is a "Shell"?

- The "Shell" is simply *another program* on top of the kernel which provides a basic human-OS interface.
  - It is a command interpreter
    - Built on top of the kernel
    - Enables users to run services provided by the UNIX OS
  - In its simplest form, a series of commands in a file is a shell program that saves having to retype commands to perform common tasks.

- How to know what shell you use

```
echo $SHELL
```



UNIX Shells

- sh Bourne Shell (Original Shell) (*Steven Bourne of AT&T*)
- bash Bourne Again Shell (*GNU Improved Bourne Shell*)
- csh C-Shell (C-like Syntax)(*Bill Joy of Univ. of California*)
- ksh Korn-Shell (Bourne+some C-shell)(*David Korn of AT&T*)
- tcsh Turbo C-Shell (More User Friendly C-Shell).
- To check shell:
  - $ `echo $SHELL` (shell is a pre-defined variable)
- To switch shell:
  - $ exec shellname (e.g., `$ exec bash` or simply type `$ bash`)
  - You can switch from one shell to another by just typing the name of the shell. `exit` return you back to previous shell.

Which Shell to Use?

- sh (Bourne shell) was considered better for programming
- csh (C-Shell) was considered better for interactive work.
- tcsh and korn were improvements on c-shell and bourne shell respectively.
- bash is largely compatible with sh and also has many of the nice features of the other shells
- On many systems such as our LINUX clusters sh is symbolically linked to bash, /bin/sh -> /bin/bash
- We recommend that you use sh/bash for writing new shell scripts but learn csh/tcsh to understand existing scripts.
- Many, if not all, scientific applications require csh/tcsh environment (GUI, Graphics Utility Interface)
- **All Linux versions use the Bash shell (Bourne Again Shell) as the default shell**
  - **Bash/Bourn/ksh/sh prompt: $**
- **All UNIX system include C shell and its predecessor Bourne shell.**
  - **Csh/tcsh prompt: %**

What is Shell Script?

- A **shell script** is a script written for the shell

- Two key ingredients
  - UNIX/LINUX commands
  - Shell programming syntax

A Shell Script Example

```
#!/bin/sh

`ls -l *.log| awk '{print $8}' |sed 's/.log//g' > file_list`

cat file_list|while read each_file
do
      babel -ig03 $each_file".log" -oxyz $each_file".xyz“

      echo '# nosymmetry integral=Grid=UltraFine scf=tight rhf/6-311++g** pop=(nbo,chelpg)'>head
      echo ' ' >>head
      echo "$each_file'  opt pop nbo chelp aim charges ' >> head
      echo ' ' >>head
      echo '0 1 ' >>head

      `sed '1,2d' $each_file.xyz >junk`
      input=./$each_file".com"
      cat head  > $input
      cat junk >> $input
      echo ' '  >> $input

done
/bin/rm ./junk ./head ./file_list
```

UNIX/LINUX Commands

- File Management and Viewing
- Filesystem Mangement
- Help,Job/Process Management
- Network Management
- System Management
- User Management
- Printing and Programming
- Document Preparation
- Miscellaneous

- To understand the working of the command and possible options use (`man` command)
- Using the GNU Info System (`info`, info command)
- Listing a Description of a Program (`whatis` command)
- Many tools have a long-style option, `` `--help' ``, that outputs usage information about the tool, including the options and arguments the tool takes. Ex: *`whoami --help`*

File and Directory Management

- **cd** Change the current directory. With no arguments "cd" changes to the users home directory. (cd <directory path>)
- **chmod** Change the file permissions.

Ex: `chmod 751 myfile` : change the file permissions to rwx for owner, rx for group and x for others (x=1,r=4,w=2)

Ex: `chmod go=+r myfile` : Add read permission for the group and others (character meanings u-user, g-group, o-other, + add permission,-remove,r-read,w-write,x-exe)

Ex: `chmod +s myfile` - Setuid bit on the file which allows the program to run with user or group privileges of the file.

- **chown** Change owner.

Ex: chown <owner1> <filename> : Change ownership of a file to owner1.

- **chgrp** Change group.

Ex: chgrp <group1> <filename> : Change group of a file to group1.

- **cp** Copy a file from one location to another.

Ex: cp file1 file2 : Copy file1 to file2; Ex: cp –R dir1 dir2 : Copy dir1 to dir2

File and Directory Management

- **ls** List contents of a directory.

  Ex: ls, ls –l , ls –al, ls –ld, ls –R

- **mkdir** Make a directory.

  Ex: mkdir <directory name> : Makes a directory

  Ex *mkdir –p /www/chache/var/log* will create all the directories starting from www.

- **mv** Move or rename a file or directory.

  Ex: mv <source> <destination>

- **find** Find files (find <start directory> -name <file name> -print)

  Ex: *find /home –name readme -print*

  Search for readme starting at home and output full path, “/home" = Search starting at the home directory and proceed through all its subdirectories; "-name readme" = Search for a file named readme "-print" = Output the full path to that file

- **locate** File locating program that uses the slocate database.

  Ex: locate –u to create the database,

  locate <file/directory> to find file/directory

File and Directory Management

- **pwd** Print or list the present working directory with full path.
- **rm** Delete files (Remove files). (rm –rf <directory/file>)
- **rmdir** Remove a directory. The directory must be empty. (rmdir <directory>)
- **touch** Change file timestamps to the current time. Make the file if it doesn't exist. (touch <filename>)
- **whereis** Locate the binary and man page files for a command. (whereis <program/command>)
- **which** Show full path of commands where given commands reside. (which <command>)

File viewing and editing

- **emacs** Full screen editor.
- **pico** Simple text editor.
- **vi** Editor with a command mode and text mode. Starts in command mode.
- **gedit** GUI Text Editor
- **tail** Look at the last 10 lines of a file.

  Ex: tail –f <filename> ; Ex: tail -100 <filename>
- **head** Look at the first 10 lines of a file. (head <filename>)

File and Directory Management

File compression, backing up and restoring

- **compress** Compress data.
- **uncompress** Expand data.
- **cpio** Can store files on tapes. to/from archives.
- **gzip** - zip a file to a gz file.
- **gunzip** - unzip a gz file.
- **tar** Archives files and directories. Can store files and directories on tapes.

  Ex: tar -zcvf <destination> <files/directories> - Archive copy groups of files. tar –zxvf <compressed file> to uncompress
- **zip** – Compresses a file to a .zip file.
- **unzip** – Uncompresses a file with .zip extension.
- **cat** View a file

  Ex: cat filename
- **cmp** Compare two files.
- **cut** Remove sections from each line of files.

File and Directory Management

- **diff** Show the differences between files.

  Ex: diff file1 file2 : Find differences between file1 & file2.

- **echo** Display a line of text.
- **grep** List all files with the specified expression.
  (*grep pattern <filename/directorypath>*)

  Ex: `ls -l |grep sidbi` : List all lines with a sidbi in them.

  Ex: grep " R " : Search for R with a space on each side

- **sleep** Delay for a specified amount of time.
- **sort** Sort a file alphabetically.
- **uniq** Remove duplicate lines from a sorted file.
- **wc** Count lines, words, characters in a file. (wc –c/w/l <filename>).
- **sed** stream **ed**itor, extremely powerful!
- **awk** an extremely versatile programming language for working on files

Useful Commands in Scripting

- `grep`
  - Pattern searching
  - Example: `grep 'boo' filename`
- `sed`
  - Text editing
  - Example: `sed 's/XYZ/xyz/g' filename`
- `awk`
  - Pattern scanning and processing
  - Example: `awk '{print $4, $7}' filename`

Shell Scripting

- Start vi scriptfilename.sh with the line

  #!/bin/sh
- All other lines starting with # are comments.
  - make code readable by including comments
- Tell Unix that the script file is executable

  $ chmod u+x scriptfilename.sh

  $ chmod +x scriptfilename.sh
- Execute the shell-script

  $ ./scriptfilename.sh

My First Shell Script

```
$ vi myfirstscript.sh
    #! /bin/sh

    # The first example of a shell script
    directory=`pwd`
    echo Hello World!
    echo The date today is `date`
    echo The current directory is $directory
$ chmod +x myfirstscript.sh
$ ./myfirstscript.sh
    Hello World!
    The date today is Mon Mar 8 15:20:09 EST 2010
    The current directory is /netscr/shubin/test
```

Shell Scripts

- Text files that contain sequences of UNIX commands , created by a text editor
- No compiler required to run a shell script, because the UNIX shell acts as an interpreter when reading script files
- After you create a shell script, you simply tell the OS that the file is a program that can be executed, by using the chmod command to change the files' mode to be executable
- Shell programs run less quickly than compiled programs, because the shell must interpret each UNIX command inside the executable script file before it is executed

Commenting

- Lines starting with # are comments except the very first line where #! indicates the location of the shell that will be run to execute the script.
- On any line characters following an unquoted # are considered to be comments and ignored.
- Comments are used to;
  - Identify who wrote it and when
  - Identify input variables
  - Make code easy to read
  - Explain complex code sections
  - Version control tracking
  - Record modifications

Quote Characters

There are three different quote characters with different behaviour. These are:

“ : double quote, weak quote. If a string is enclosed in “ ” the references to variables (i.e *$variable*) are replaced by their values. Also back-quote and escape \ characters are treated specially.

‘ : single quote, strong quote. Everything inside single quotes are taken literally, nothing is treated as special.

` : back quote. A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example: `echo “Today is:” `date``

Echo

Echo command is well appreciated when trying to debug scripts.

Syntax : `echo {options}  string`

Options: `-e` : expand \ (back-slash) special characters

`-n` : do not output a new-line at the end.

String can be a “weakly quoted” or a ‘strongly quoted’ string. In the weakly quoted strings the references to variables are replaced by the value of those variables before the output.

As well as the variables some special backslash_escaped symbols are expanded during the output. If such expansions are required the -e option must be used.

User Input During Shell Script Execution

- As shown on the hello script input from the standard input location is done via the read command.
- Example

```
echo "Please enter three filenames:”
read  filea fileb filec
echo “These files are used:$filea  $fileb  $filec”
```

- Each read statement reads an entire line. In the above example if there are less than 3 items in the response the trailing variables will be set to blank ‘ ‘.
- Three items are separated by one space.

Hello script exercise continued…

- The following script asks the user to enter his name and displays a personalised hello.

```
#!/bin/sh
echo "Who am I talking to?"
read user_name
echo "Hello $user_name"
```

- Try replacing " with ' in the last line to see what happens.

Debugging your shell scripts

- Generous use of the echo command will help.

- Run script with the -x parameter.

  E.g. sh -x ./myscript

  or set -o xtrace before running the script.

- These options can be added to the first line of the script where the shell is defined.

  e.g. #!/bin/sh -xv

Shell Programming

- **Programming features of the UNIX/LINUX shell:**
  - ***Shell variables***: Your scripts often need to keep values in memory for later use. Shell variables are symbolic names that can access values stored in memory
  - ***Operators***: Shell scripts support many operators, including those for performing mathematical operations
  - ***Logic structures***: Shell scripts support **sequential logic** (for performing a series of commands), **decision logic** (for branching from one point in a script to another), **looping logic** (for repeating a command several times), and **case logic** (for choosing an action from several possible alternatives)

Variables

- **Variables are symbolic names that represent values stored in memory**
- **Three different types of variables**
  - Global Variables: **Environment and configuration variables, capitalized, such as HOME, PATH, SHELL, USERNAME, and PWD.**

    When you login, there will be a large number of global System variables that are already defined. These can be freely referenced and used in your shell scripts.
  - Local Variables

    Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
  - Special Variables

    Reversed for OS, shell programming, etc. such as positional parameters $0, $1 …

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1 & PS2	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Referencing Variables

Variable contents are accessed using '$':

e.g. $ echo $HOME

$ echo $SHELL

To see a list of your environment variables:

```
$ printenv
```

or:

```
$ printenv | more
```

Defining Local Variables

- As in any other programming language, variables can be defined and used in shell scripts.
- Unlike other programming languages, variables in Shell Scripts are not typed.
- Examples :

```
a=1234  # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but be the string '1234+1'
b=`expr $a + 1 `  will perform arithmetic so b is 1235 now.
```

Note : +,-,/,*,**, % operators are available.

```
b=abcde # b is string
b='abcde' # same as above but much safer.
b=abc  def # will not work unless 'quoted'
b='abc def'  # i.e. this will work.
```

IMPORTANT NOTE: DO NOT LEAVE SPACES AROUND THE =

Referencing variables --curly bracket

- Having defined a variable, its contents can be referenced by the $ symbol. E.g. `${variable}` or simply `$variable`. When ambiguity exists $variable will not work. Use `${ }` the rigorous form to be on the safe side.
- Example:

```
a='abc'
b=${a}def # this would not have worked without the{ } as
          #it would try to access a variable named adef
```

Variable List/Arrary

- To create lists (array) – round bracket
  $ set Y = (UNL 123 CS251)

- To set a list element – square bracket
  $ set Y[2] = HUSKER

- To view a list element:
  $ echo $Y[2]

- Example:

```
#!/bin/sh
a=(1 2 3)
echo ${a[*]}
echo ${a[0]}
```

Results:

```
1 2 3
1
```

Positional Parameters

- When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed.
- $0 This variable that contains the name of the script
- $1, $2, ….. $n 1st, 2nd 3rd command line parameter
- $# Number of command line parameters
- $$ process ID of the shell
- $@ same as $* but as a list one at a time (see for loops later)
- $? Return code 'exit code' of the last command
- `Shift` command: This shell command shifts the positional parameters by one towards the beginning and drops $1 from the list. After a shift $2 becomes $1 , and so on … It is a useful command for processing the input parameters one at a time.

Example:

Invoke : `./myscript one two buckle my shoe`

During the execution of `myscript` variables $1 $2 $3 $4 and $5 will contain the values *one, two, buckle, my, shoe* respectively.

Variables

- vi myinputs.sh

```
#! /bin/sh
echo Total number of inputs: $#
echo First input: $1
echo Second input: $2
```

- chmod u+x myinputs.sh

- myinputs.sh HUSKER UNL CSE

```
Total number of inputs: 3
First input: HUSKER
Second input: UNL
```

Shell Programming

- **programming features of the UNIX shell:**
  - ***Shell variables***
  - ***Operators***
  - ***Logic structures***

Shell Operators

- **The Bash/Bourne/ksh shell operators are divided into three groups: defining and evaluating operators, arithmetic operators, and redirecting and piping operators**

Defining and Evaluating

- A shell variable take on the generalized form variable=value (except in the C shell).

```
$ set x=37; echo $x
37
$ unset x; echo $x
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Pipes & Redirecting

- **Piping: An important early development in Unix , a way to pass the output of one tool to the input of another.**

```
$ who | wc -l
```

By combining these two tools, giving the wc command the output of who, you can build a new command to list the number of users currently on the system

- **Redirecting via angle brackets:** Redirecting input and output follows a similar principle to that of piping except that redirects work with files, not commands.

```
tr '[a-z]' '[A-Z]' < $in_file > $out_file
```

The command must come first, the *in_file* is directed in by the less_than sign (<) and the *out_file* is pointed at by the greater_than sign (>).

Arithmetic Operators

- **expr supports the following operators:**
  - **arithmetic operators: +,-,*,/,%**
  - **comparison operators: <, <=, ==, !=, >=, >**
  - **boolean/logical operators: &, |**
  - **parentheses: (,)**
  - **precedence is the same as C, Java**

Arithmetic Operators

- **vi math.sh**

```
#!/bin/sh
count=5
count=`expr $count + 1 `
echo $count
```

- **chmod u+x math.sh**
- **math.sh**

6

Arithmetic Operators

- **vi real.sh**

```
#!/bin/sh
a=5.48
b=10.32
c=`echo "scale=2; $a + $b" |bc`
echo $c
```

- **chmod u+x real.sh**
- **./real.sh**

```
15.80
```

Arithmetic operations in shell scripts

var++ ,var-- , ++var , --var	post/pre increment/decrement
+ , -	add subtract
* , / , %	multiply/divide, remainder
**	power of
! , ~	logical/bitwise negation
& , |	bitwise AND, OR
&& ||	logical AND, OR

Shell Programming

- **programming features of the UNIX shell:**
  - ***Shell variables***
  - ***Operators***
  - ***Logic structures***

Shell Logic Structures

The four basic logic structures needed for program development are:

- **Sequential logic:** to execute commands in the order in which they appear in the program
- **Decision logic:** to execute commands only if a certain condition is satisfied
- **Looping logic:** to repeat a series of commands for a given number of times
- **Case logic:** to replace "if then/else if/else" statements when making numerous comparisons

Conditional Statements (if constructs)

The most general form of the if construct is;

```
if command executes successfully
then
        execute command
elif this command executes successfully
then
        execute this command
        and execute this command
else
        execute default command
fi
```

However- elif and/or else clause can be omitted.

Examples

SIMPLE EXAMPLE:

```
if date | grep “Fri”
then
        echo “It’s Friday!”
fi
```

FULL EXAMPLE:

```
if  [ “$1”  ==  “Monday”  ]
then
        echo “The typed argument is Monday.”
elif [ “$1” == “Tuesday” ]
 then
        echo “Typed argument is Tuesday”
 else
        echo “Typed argument is neither Monday nor Tuesday”
fi
```

Note: = or == will both work in the test but == is better for readability.

Tests

String and numeric comparisons used with test or [[ ]] which is an alias for test and also [ ] which is another acceptable syntax

string1 = string2	True if strings are identical
String1 == string2	...ditto....
string1 !=string2	True if strings are not identical
string	Return 0 exit status (=true) if string is not null
-n string	Return 0 exit status (=true) if string is not null
-z string	Return 0 exit status (=true) if string is null

int1 -eq int2	Test identity
int1 -ne int2	Test inequality
int1 -lt int2	Less than
int1 -gt int2	Greater than
int1 -le int2	Less than or equal
int1 -ge int2	Greater than or equal

Combining tests with logical operators || (or) and && (and)

Syntax: if cond1 && cond2 || cond3 …

An alternative form is to use a compound statement using the –a and –o keywords, i.e.

if cond1 –a cond22 –o cond3 …

Where cond1,2,3 .. Are either commands returning a a value or test conditions of the form [ ] or test …

Examples:

```
if  date | grep “Fri”  &&  `date +’%H’` -gt 17
then
    echo “It’s Friday, it’s home time!!!”
fi

if [ “$a” –lt 0 –o “$a” –gt 100 ]     # note the spaces around ] and [
then
    echo “ limits exceeded”
fi
```

File enquiry operations

-d file	Test if file is a directory
-f file	Test if file is not a directory
-s file	Test if the file has non zero length
-r file	Test if the file is readable
-w file	Test if the file is writable
-x file	Test if the file is executable
-o file	Test if the file is owned by the user
-e file	Test if the file exists
-z file	Test if the file has zero length

All these conditions return true if satisfied and false otherwise.

Decision Logic

- **A simple example**

```
#!/bin/sh
if [ "$#" -ne 2 ] then
        echo $0 needs two parameters!
        echo You are inputting $# parameters.
 else
        par1=$1
        par2=$2
 fi
 echo $par1
 echo $par2
```

Decision Logic

Another example:

```
#! /bin/sh
#  number is positive, zero or negative
echo –e "enter a number:\c"
read number
if [ “$number” -lt 0 ]
then
      echo "negative"
elif [ “$number” -eq 0 ]
then
      echo zero
else
       echo positive
fi
```

Loops

Loop is a block of code that is repeated a number of times.

The repeating is performed either a pre-determined number of times determined by a list of items in the loop count (for loops) or until a particular condition is satisfied (while and until loops)

To provide flexibility to the loop constructs there are also two statements namely break and continue are provided.

for loops

Syntax:

```
for  arg in list
do
    command(s)
      ...
done
```

Where the value of the variable ***arg*** is set to the values provided in the list one at a time and the block of statements executed. This is repeated until the list is exhausted.

Example:

```
for i in 3 2 5 7
do
      echo " $i times 5 is  $((  $i  * 5 ))  "
done
```

The while Loop

- A different pattern for looping is created using the **while** statement
- The **while statement** best illustrates how to set up a loop to test repeatedly for a matching condition
- The while loop tests an expression in a manner similar to the if statement
- As long as the statement inside the brackets is true, the statements inside the do and done statements repeat

while loops

Syntax:

```
while this_command_execute_successfully
 do
        this command
        and this command
 done
```

EXAMPLE:

```
while test "$i" -gt 0      # can also be  while  [ $i > 0 ]
do
        i=`expr $i - 1`
done
```

Looping Logic

- **Example:**

```
#!/bin/sh
for person in Bob Susan Joe Gerry
do
   echo Hello $person
done
```

Output:

```
Hello Bob
Hello Susan
Hello Joe
Hello Gerry
```

- **Adding integers from 1 to 10**

```
#!/bin/sh
i=1
sum=0
while [ "$i" -le 10 ]
  do
  echo Adding $i into the sum.
  sum=`expr $sum + $i `
          i=`expr $i + 1 `
  done
echo The sum is $sum.
```

until loops

The syntax and usage is almost identical to the while-loops.

Except that the block is executed until the test condition is satisfied, which is the opposite of the effect of test condition in while loops.

Note: You can think of *until* as equivalent to *not_while*

Syntax:

```
until test
do
   commands ….
done
```

Switch/Case Logic

- **The switch logic structure simplifies the selection of a match when you have a list of choices**
- **It allows your program to perform one of many actions, depending upon the value of a variable**

Case statements

The case structure compares a string ‘usually contained in a variable’ to one or more patterns and executes a block of code associated with the matching pattern. Matching-tests start with the first pattern and the subsequent patterns are tested only if no match is not found so far.

```
case argument in
    pattern 1) execute this command
                    and this
                    and this;;
    pattern 2) execute this command
                    and this
                    and this;;
    esac
```

Functions

- Functions are a way of grouping together commands so that they can later be executed via a single reference to their name. If the same set of instructions have to be repeated in more than one part of the code, this will save a lot of coding and also reduce possibility of typing errors.

  SYNTAX:

  functionname()
  {
  block of commands
  }

```
#!/bin/sh
  sum() {
    x=`expr $1 + $2`
    echo $x
    }

sum 5 3
echo "The sum of 4 and 7 is `sum 4 7`"
```

Take-Home Message

- Shell script is a high-level language that must be converted into a low-level (machine) language by UNIX Shell before the computer can execute it
- UNIX shell scripts, created with the vi or other text editor, contain two key ingredients: a selection of UNIX commands glued together by Shell programming syntax
- UNIX/Linux shells are derived from the UNIX Bourne, Korn, and C/TCSH shells
- UNIX keeps three types of variables:
  - Configuration; environmental; local
- The shell supports numerous operators, including many for performing arithmetic operations
- The logic structures supported by the shell are sequential, decision, looping, and case

To Script or Not to Script

- Pros
  - File processing
  - Glue together compelling, customized testing utilities
  - Create powerful, tailor-made manufacturing tools
  - Cross-platform support
  - Custom testing and debugging
- Cons
  - Performance slowdown
  - Accurate scientific computing

Shell Scripting Examples

- Input file preparation
- Job submission
- Job monitoring
- Results processing

Input file preparation

```
#!/bin/sh

`ls -l *.log| awk '{print $8}' |sed 's/.log//g' > file_list`

cat file_list|while read each_file
do
      babel -ig03 $each_file".log" -oxyz $each_file".xyz“

      echo '# nosymmetry integral=Grid=UltraFine scf=tight rhf/6-311++g** pop=(nbo,chelpg)'>head
      echo ' ' >>head
      echo "$each_file'  opt pop nbo chelp aim charges ' >> head
      echo ' ' >>head
      echo '0 1 ' >>head

      `sed '1,2d' $each_file.xyz >junk`
      input=./$each_file".com"
      cat head  > $input
      cat junk >> $input
      echo ' '  >> $input

done
/bin/rm ./junk ./head ./file_list
```

LSF Job Submission

```
$ vi submission.sh
    #!/bin/sh -f

    #BSUB -q week
    #BSUB -n 4
    #BSUB -o output
    #BSUB -J job_type
    #BSUB -R "RH5 span[ptile=4]"
    #BSUB -a mpichp4

    mpirun.lsf ./executable.exe

    exit
$chmod +x submission.sh
$bsub < submission.sh
```

Results Processing

```
#!/bin/sh
`ls -l *.out| awk '{print $8}'|sed 's/.out//g'  > file_list`
cat file_list|while read each_file
do
    file1=./$each_file".out"
    Ts=`grep 'Kinetic energy =' $file1 |tail -n 1|awk '{print $4}' `
    Tw=`grep 'Total Steric Energy:' $file1 |tail -n 1|awk '{print $4}' `
    TsVne=`grep 'One electron energy =' $file1 |tail -n 1|awk '{print $5}' `
    Vnn=`grep 'Nuclear repulsion energy' $file1 |tail -n 1|awk '{print $5}' `
    J=`grep 'Coulomb energy =' $file1 |tail -n 1|awk '{print $4}' `
    Ex=`grep 'Exchange energy =' $file1 |tail -n 1|awk '{print $4}' `
    Ec=`grep 'Correlation energy =' $file1 |tail -n 1|awk '{print $4}' `
    Etot=`grep 'Total DFT energy =' $file1 |tail -n 1|awk '{print $5}' `
    HOMO=`grep 'Vector' $file1 | grep 'Occ=2.00'|tail -n 1|cut -c35-47|sed 's/D/E/g' `
    orb=`grep 'Vector' $file1 | grep 'Occ=2.00'|tail -n 1|awk '{print $2}' `
    orb=`expr $orb + 1 `
    LUMO=`grep 'Vector' $file1 |grep 'Occ=0.00'|grep ' '$orb' ' |tail -n 1|cut -c35-47|sed
's/D/E/g`
    echo $each_file $Etot $Ts $Tw $TsVne $J $Vnn $Ex $Ec $HOMO $LUMO $steric >>out
done
/bin/rm file_list
```

Reference Books

- **Class Shell Scripting**
  http://oreilly.com/catalog/9780596005955/
- **LINUX Shell Scripting With Bash**
  http://ebooks.ebookmall.com/title/linux-shell-scripting-with-bash-burtch-ebooks.htm
- **Shell Script in C Shell**
  http://www.grymoire.com/Unix/CshTop10.txt
- **Linux Shell Scripting Tutorial**
  http://www.freeos.com/guides/lsst/
- **Bash Shell Programming in Linux**
  http://www.arachnoid.com/linux/shell_programming.html
- **Advanced Bash-Scripting Guide**
  http://tldp.org/LDP/abs/html/
- **Unix Shell Programming**
  http://ebooks.ebookmall.com/title/unix-shell-programming-kochan-wood-ebooks.htm

Questions & Comments

Please direct comments/questions about research computing to

E-mail: research@unc.edu

Please direct comments/questions pertaining to this presentation to

E-Mail: shubin@email.unc.edu

The PPT file of this presentation is available here:
http://its2.unc.edu/divisions/rc/training/scientific/short_courses/Shell_Scripting.ppt

UNC
INFORMATION
TECHNOLOGY SERVICES

Hands-on Exercises

1. The simplest Hello World shell script – Echo command
2. Summation of two integers – If block
3. Summation of two real numbers – bc (basic calculator) command
4. Script to find out the biggest number in 3 numbers – If –elif block
5. Operation (summation, subtraction, multiplication and division) of two numbers – Switch
6. Script to reverse a given number – While block
7. A more complicated greeting shell script
8. Sort the given five numbers in ascending order (using array) – Do loop and array
9. Calculating average of given numbers on command line arguments – Do loop
10. Calculating factorial of a given number – While block
11. An application in research computing – Combining all above
12. Optional: Write own shell scripts for your own purposes if time permits

The PPT/WORD format of this presentation is available here:
http://its2.unc.edu/divisions/rc/training/scientific/
/afs/isis/depts/its/public_html/divisions/rc/training/scientific/short_courses/