

CSET213 – Linux and Shell Programming

Course Type - Core L-T-P Format 2-0-4 Credits – 4

COURSE SUMMARY

This course describes the essential ideas behind the open-source operating system approach to programming. Knowledge of Linux and shell script helps to understand the backbone of cybersecurity. This course involves basic Linux commands, Shell scripting, File structure and management, Processes, Inter-process communication, Socket programming, and security.

COURSE-SPECIFIC LEARNING OUTCOMES (CO)

CO1: To articulate Linux commands that are used to manipulate system operations at an admin level.

CO2: To write Shell Programming using Linux commands.

CO3: To design and write applications to manipulate internal kernel-level Linux File systems.

Detailed Syllabus

Module 1 (Contact hours: 8) ✓

Linux and Linux utilities, Architecture of Linux, features of Linux, Introduction to vi editor, Linux commands, File handling utilities, security by file permissions, process utilities, disk utilities, Networking commands, Text processing utilities, backup utilities, User management.

Module 2 (Contact hours: 8) ✓ bash

Shells need and types, Derived Operators, Linux session, Standard streams, Redirection, Pipes, Tee command, command execution, command-line editing, Quotes, command substitution, job control, aliases, variables, shell/environment customization, Filters, and pipes, File operations.

Module 3 (Contact hours: 12) ✓

Grep Operation, Grep Commands, Grep Address, Grep Application, Sed Scripts, operation, Unix file structure, File descriptors, System calls and device drivers, File management, File structures, System calls for file management, Directory API, Process and Process Structure, Process table, Viewing processes, System processes, Process scheduling, zombie processes, orphan process, Fork and its operation, Signals functions, unreliable signals, interrupted system calls, Signal sets, File locking, Threats and Vulnerabilities analysis of Linux- direct, indirect, veiled, conditional, Security Measures in Linux-SSH key pair, Scan Log files , Close Hidden ports, Linux Malwares- Botnets, Ransomware, Rootkits, Socket, Socket communications, UDP, TCP, AWK, Shell Scripting and Security- Password Tester, Permissions and Access Control Lists, Shell Scripting for DevOps- Using environment variables, Bash Script

STUDIO WORK / LABORATORY EXPERIMENTS:

Students will use LINUX / UBUNTU to gain hands-on experience on LINUX and Shell programming, Linux commands, their uses and practice, editors: vi, nano etc, Introduction to Shell, Shell basic commands, variables Shell programming environments- filters and pipe, Shell programming File handling, Grep its use and commands. Using of Grep with pipe and filters, Unix file structuring, inodes and related system calls. File handling commands and API, Network Penetration testing tools, Wireshark, Nmap, Hash cat, Process management, creation, termination and other useful commands, Process scheduling. Parent, zombie and orphan process, Process system calls. Fork, exec, wait and signal, various commands. Basics of Socket Programming via UDP socket

TEXTBOOKS/LEARNING RESOURCES:

- 1) M. Ebrahim and A Mallett, Mastering Linux Shell Scripting: A Practical Guide to Linux Command-Line, Bash Scripting, and She (2 ed.), Packt Publication, 2018. ISBN 978-1788990554.
- 2) R. Blum and C. Bresnahan, Linux Command Line and Shell Scripting Bible (3 ed.), Wiley, 2016. ISBN 978-1118983843.

REFERENCE BOOKS/LEARNING RESOURCES:

- 1) W.R. Stevens, UNIX Network Programming (3 ed.), PHI Publications, 2017. ISBN 978-8120307490.

Lecture wise Plan

Lect. No.	Content Planned	Content Practiced
1.	Course structure/handout Assessment mechanism (15) Linux and Linux Utilities (35)	Done As given
2.	Architecture of Linux (20) features of Linux (15) VI editor (15)	Done As given
3.	File handling utilities (15) security by file permissions (10) process utilities (15)	Filesystem, Linux Security
4.	disk utilities (20) Networking commands (25)	As given
5.	Text processing utilities (25) backup utilities (20)	As given
6.	User Management (45)	+ group management
7.	Linux session (15) Standard streams (15) Redirection (10) Pipes (10)	done
8.	Assessment /Buffer Lecture*	
9.	Shell/environment customization (30) Filters and pipes (15)	done
10.	File operations (45)	done
11.	Grep Operation (20) Grep Commands (30)	done
12.	Case study (50)	
13.	Start-ups in Linux and Shell programming (50)	
14.	Unix file structure (45)	done
15.	File descriptors (20) System calls and device drivers (30)	done
16.	Assessment/ Buffer Lecture	
17.	File management (15) File structures (30)	done
18.	Process and Process Structure (10) Process table (20) Viewing processes (10) System processes (10)	done
19.	Process scheduling (40) zombie processes, orphan process (10)	done
20.	Fork and its operation (45)	done
21.	Signals functions (20) unreliable signals (15) interrupted system calls (15)	Done
22.	Signal sets (25) File locking (25)	Done
23.	Threats and Vulnerabilities analysis of Linux (10) Direct (10) Indirect (10) Veiled (10) Conditional (10)	Done

24.	Security Measures in Linux (5) SSH key pair (15) Scan Log files (15) Close Hidden ports (15)	
25.	Linux Malwares (10) Botnets (15) Ransomware (10) Rootkits (10)	
26.	Trojans, Viruses, Worms (10) Shell Scripting and Security (5) Password Tester (15) Permissions and Access Control Lists (20)	
27.	Shell Scripting for DevOps (15) Using environment variables (15) Bash Script (20)	
28.	End Term Assessment/Buffer	

Lab wise Plan

Lab No.	Planned Lab Content	Practiced
1.	Installation of Linux and its various distros	Lab Assignment 1: Installation of Ubuntu as dual boot
2.	Linux commands- commands and editor	Lab Assignment 2
3.	Linux Commands-User Management	Lab Assignment 3
4.	Linux commands-Networking Commands	Lab Assignment 4
5.	Introduction to Shell, Shell basic commands, variables	Lab Assignment 5
6.	Shell Programming operators	Lab Assignment 6
7.	Shell programming loops	Lab Assignment 7
8.	Shell programming environments- filters and pipe	Lab Assignment 8
9.	Shell programming File handling	Lab Assignment 9
10.	Shell Functions	Lab Assignment 10
11.	AWK commands and its application	Lab Assignment 11
12.	Grep its use and commands. Using of Grep with pipe and filters	Lab Assignment 12
13.	Unix file structuring, inodes and related system calls.	Lab Assignment 13
14.	Mid-Term Lab Assessment/Buffer	
15.	Mid-Term Lab Assessment/Buffer	
16.	File handling commands and API	Lab Assignment 14
17.	Network Penetration testing tools, Wireshark, Nmap, Hash cat	Lab Assignment 15
18.	Threats and Vulnerabilities analysis of Linux	Lab Assignment 19
19.	Process management, creation, termination and other useful commands, Process scheduling. Parent, zombie, and orphan process	Lab Assignment 16
20.	Process system calls. Fork, exec	Lab Assignment 17

21.	Wait and signal	Lab Assignment 18
22.	Inter Process communication via pipes and shared memory, various commands.	Lab Assignment 20
23.	Socket Programming via UDP socket	Lab Assignment 21
24.	Shell Scripting and Security	Lab Assignment 22
25.	Shell Scripting for DevOps	Lab Assignment 23
26.	Buffer Lab	
27.	End Term Lab Assessment/Buffer	
28.	End Term Lab Assessment /Buffer	

Assessment Component

S. No	Component	Marks
1.	Continuous Lab Assessment	20
2.	Linux Kernel-based Project/Hackathon	20
3.	Certification	10
4.	End Term	35
5.	Mid Term	15

MOOCs

S. No	Course Name	Platform
1.	Linux - Learn App Development using Linux	Infosys Springboard: https://rb.gy/gqzifd
2.	Kali Linux - The Ultimate Kali Linux and Penetration Testing Training	Infosys Springboard: https://rb.gy/gqzifd
3.	Operating Systems and You: Becoming a Power User	Coursera https://www.coursera.org/learn/os-power-user
4.	Open Source Software Development, Linux and Git Specialization	Coursera

To be Filled each Semester

- 1) Probable Industry Talks:

S. No	Talk Title	Resorce Person	Company
1	Talk-1 (After 2 nd Module)	TBA	TBA
2	Talk-2 (After 3 rd Module)	TBA	TBA

- 2) Relevant MOOC Courses being Referred: Yes
 3) Probable Case Studies: Yes
 4) Advanced Research Topics: No
 5) Start-ups to be discussed: Yes
 6) Assessment Components Details: Yes
 7) Software required : Ubuntu OS, Kali, Docker
 8) Hardware required: A machine with Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz OR similar, 8GB RAM, 500GB HDD
 9) Industry/certificate mapping: Yes

Linux and Shell Programming

Cyber Security Specialization Core-I



Dr Vimal Kumar (Course Instructor),
Assistant r. Professor-SCSET
Bennett University Greater Noida

Outline

Objectives of the Course

Course Outcomes

Course Structure & Credits

Book Resources

Assessment Components

Industrial Certifications

What you will loose for not attending the classes/Labs?

Q & A

Objective

Essential ideas behind the open-source operating system approach

Programming. knowledge of Linux and shell script

Understand the backbone of cybersecurity

Course involves

- *Basic Linux commands*
- *Shell scripting*
- *File structure and management*
- *Processes*
- *Inter-process communication*
- *Socket programming*
- *Security*

COURSE OUTCOMES (CO)

1

CO1: To articulate Linux commands that are used to manipulate system operations at an admin level.

2

CO2: To write Shell Programming using Linux commands.

3

CO3: To design and write applications to manipulate internal kernel-level Linux File systems.

Course Structure & Credits

L	T	P	C
2	0	4	4



Theory Modules: 03



Theory Lectures: 28 Hours



Lab Sessions: 56 Hours (28+28 Self Lab)



Continuous Lab Assessments: 20



Marks per Lab: 02



The Linux Kernel Project: 27/11/2024 (Tentative)



Probable Industry Talks: 01

Resources will be available on LMS

Text & Reference Books

Textbooks

- M. Ebrahim and A Mallett, Mastering Linux Shell Scripting: A Practical Guide to Linux Command-Line, Bash Scripting, and She (2 ed.), Packt Publication, 2018. ISBN 978-1788990554.
- R. Blum and C. Bresnahan, Linux Command Line and Shell Scripting Bible (3 ed.), Wiley, 2016. ISBN 978-1118983843.

Reference Books

- W.R. Stevens, UNIX Network Programming (3 ed.), PHI Publications, 2017. ISBN 978-8120307490.

Assessment

S. No.	Component	Marks	Date of Evaluation (Tentative)	
1	Mid Term	10	As per schedule notified by CoE	
2	End Term Examination	40	As per schedule notified by CoE	
3	MOOC Certification from the notified list	10	27/11/2024	
4	Lab Continuous Evaluation	20	As per timetable. 10 Best Labs will be considered, 2 marks of each Lab	
5	Linux Kernel Project/ Hackathon	20	Evaluation-1 (5 Marks) 4 th Week of August 24	Hackathon (15 Marks) 29/11/2024
Total		100		

What you will loose if not attended the classes/Labs?

Installation and administration of Linux OS

Articulation of Linux commands that are used to manipulate system operations at an admin level

Write Shell Programming using Linux commands

Design and write applications using shell script to manipulate internal kernel-level Linux File systems and security scripts.

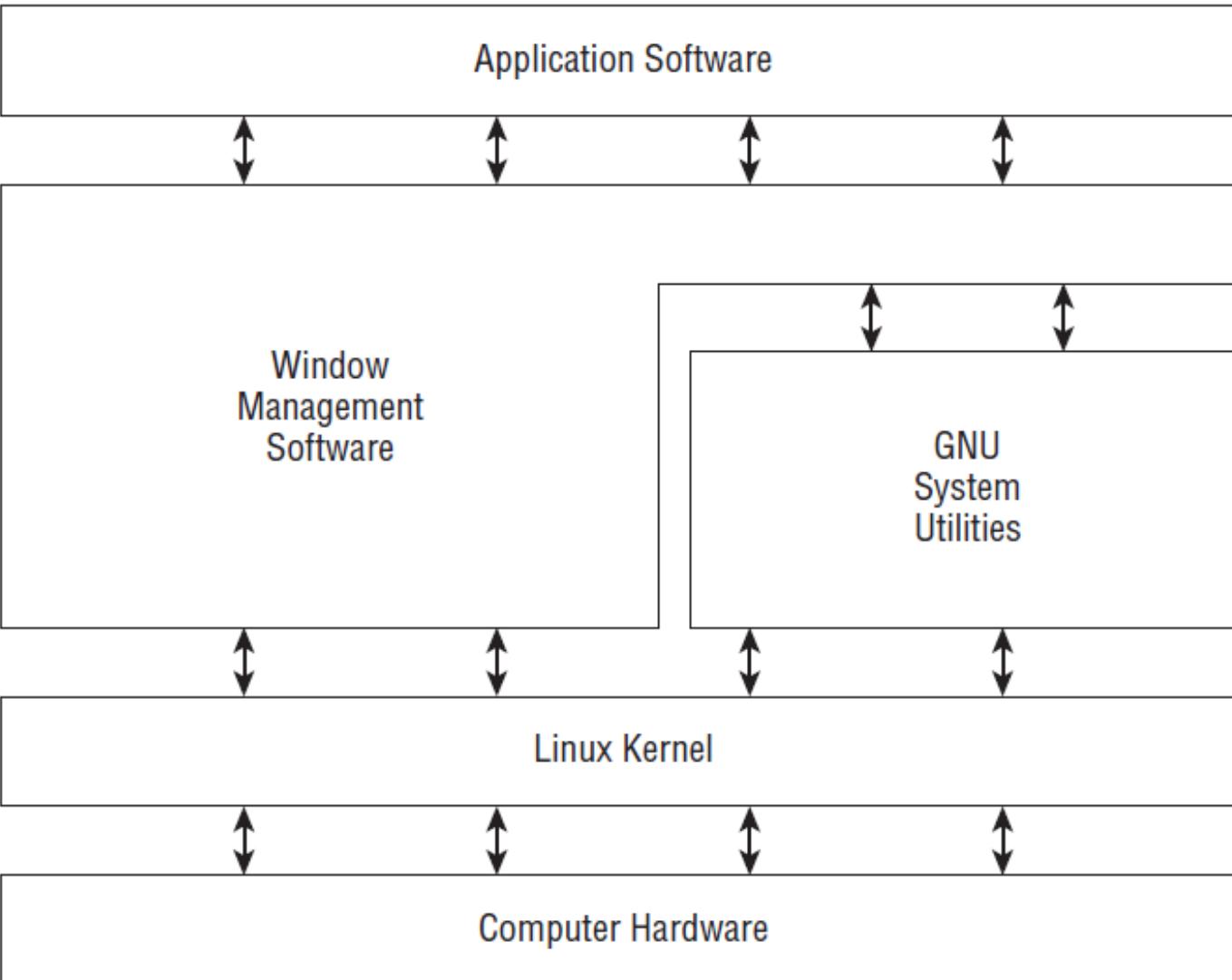
May loose Theory/lab marks due to continuous lab evaluation/missing classes

What is Linux?

- An Open source OS, written by *Linus Torvalds*.
- Four parts of Linux
 - Kernel
 - Utilities
 - Graphical Desktop
 - Applications Software



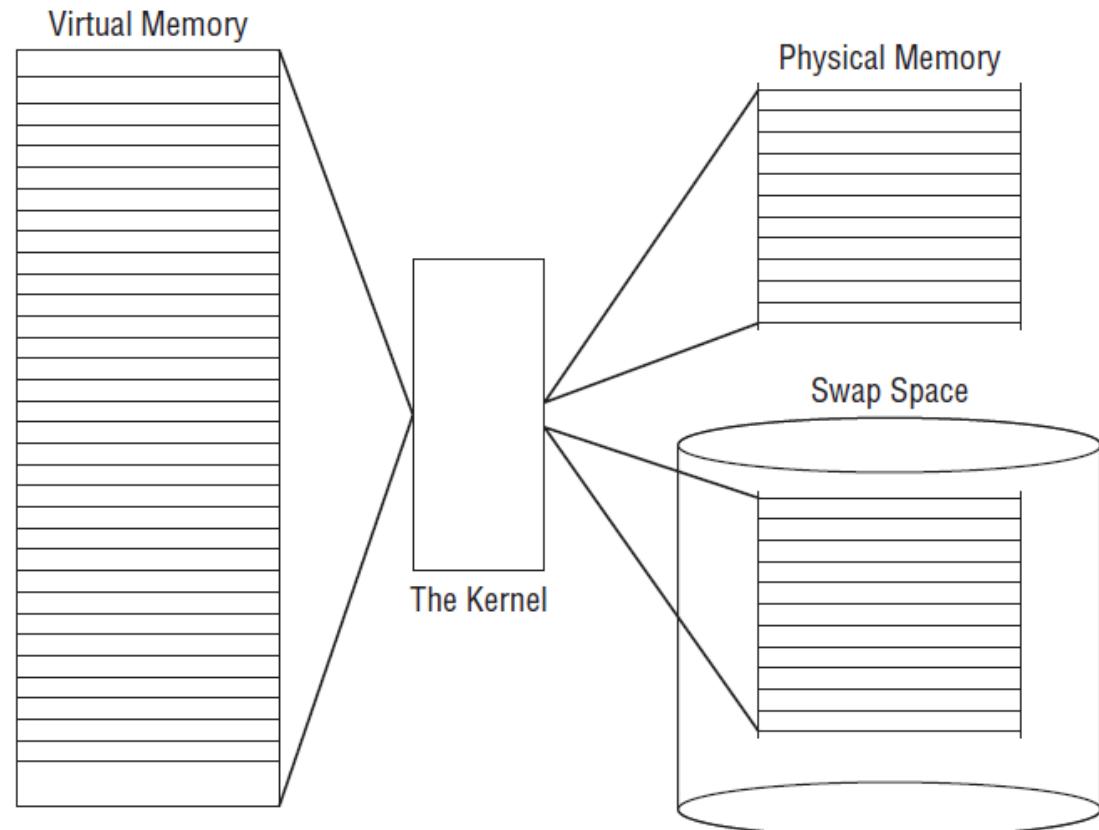
The Linux system



System Memory Management

- Virtual Memory using swap space
- Memory locations grouped into blocks are called pages
- Kernel keeps track of used pages through page tables
- Swap-out
- Swap-in

The Linux system memory map



Software Program Management

- Running program is a process, a foreground or a background process
- Kernel controls the processes
- init process---first process created by kernel
- When the kernel starts, it loads the init process into virtual memory
- Some Linux implementations contain a table of processes to start automatically on bootup.
- On Linux systems, this table is usually located in the special file /etc/inittabs.

Software Program Management

- The popular Ubuntu Linux distribution utilize the /etc/init.d folder, which contains scripts for starting and stopping individual applications at boot time.
 - The scripts are started via entries under the /etc/rcX.d folders, where X is a run level.
 - The init utilizes run levels. There are 5 init run levels in the Linux.
-
- At run level 1&2, only the basic system processes are started, along with one console terminal process. This is called single-user (admin) mode.
 - At run level 3&4, most application software, such as network support software, is started.
 - At level 5, the GUI X-windows is started

Hardware Management

- Any device that Linux must communicate with needs device driver code to be inserted into kernel
- Driver code allows the kernel to pass data back and forth to the device
- Two methods are used to insert device driver code into kernel
 - Drivers compiled into the kernel
 - **Drivers' modules are added to the kernel**
- The Linux system identifies hardware devices as special files, called device files.
- There are three classifications of device files:
 - **Character:** Character device files are for devices that can only handle data one character at a time.
Examples: modems, and terminals
 - **Block:** Block files are for devices that can handle data in large blocks at a time, such as disk drives
 - **Network:** The network file types are used for devices that use packets to send and receive data.
Examples are network cards and a special loopback device

Hardware Management

Linux creates special files, called **nodes**, for each device on the system.

All communication with the device is performed through the device **node**.

Each **node** has a unique number pair that identifies it to the Linux kernel.

The number pair includes a major and a minor device number.

Similar devices are grouped into the same major device number.

The minor device number is used to identify a specific device within the major device group.

File System Management

- The Linux kernel can support different types of filesystems to read and write data to and from hard drives.
- Table 1-1 lists the standard filesystems that a Linux system can use to read and write data.

TABLE 1-1 Linux Filesystems

Filesystem	Description
ext	Linux Extended filesystem — the original Linux filesystem
ext2	Second extended filesystem, provided advanced features over ext
ext3	Third extended filesystem, supports journaling
ext4	Fourth extended filesystem, supports advanced journaling
hpfs	OS/2 high-performance filesystem
jfs	IBM's journaling filesystem
iso9660	ISO 9660 filesystem (CD-ROMs)
minix	MINIX filesystem
msdos	Microsoft FAT16
ncp	Netware filesystem
nfs	Network File System
ntfs	Support for Microsoft NT filesystem
proc	Access to system information
ReiserFS	Advanced Linux filesystem for better performance and disk recovery
smb	Samba SMB filesystem for network access
sysv	Older Unix filesystem
ufs	BSD filesystem
umsdos	Unix-like filesystem that resides on top of msdos
vfat	Windows 95 filesystem (FAT32)
XFS	High-performance 64-bit journaling filesystem

The GNU Utilities

The core bundle of utilities supplied for Linux systems is called the *coreutils* package.

The GNU coreutils package consists of three parts:

- Utilities for handling files
- Utilities for manipulating text
- Utilities for managing processes

The Shell Utility

Shell provides a way for users to start programs, manage files on the filesystem, and manage processes running on the Linux system.

The core of the shell is the command prompt.

The command prompt is the interactive part of the shell.

It allows you to enter text commands, and then it interprets the commands and executes them in the kernel.

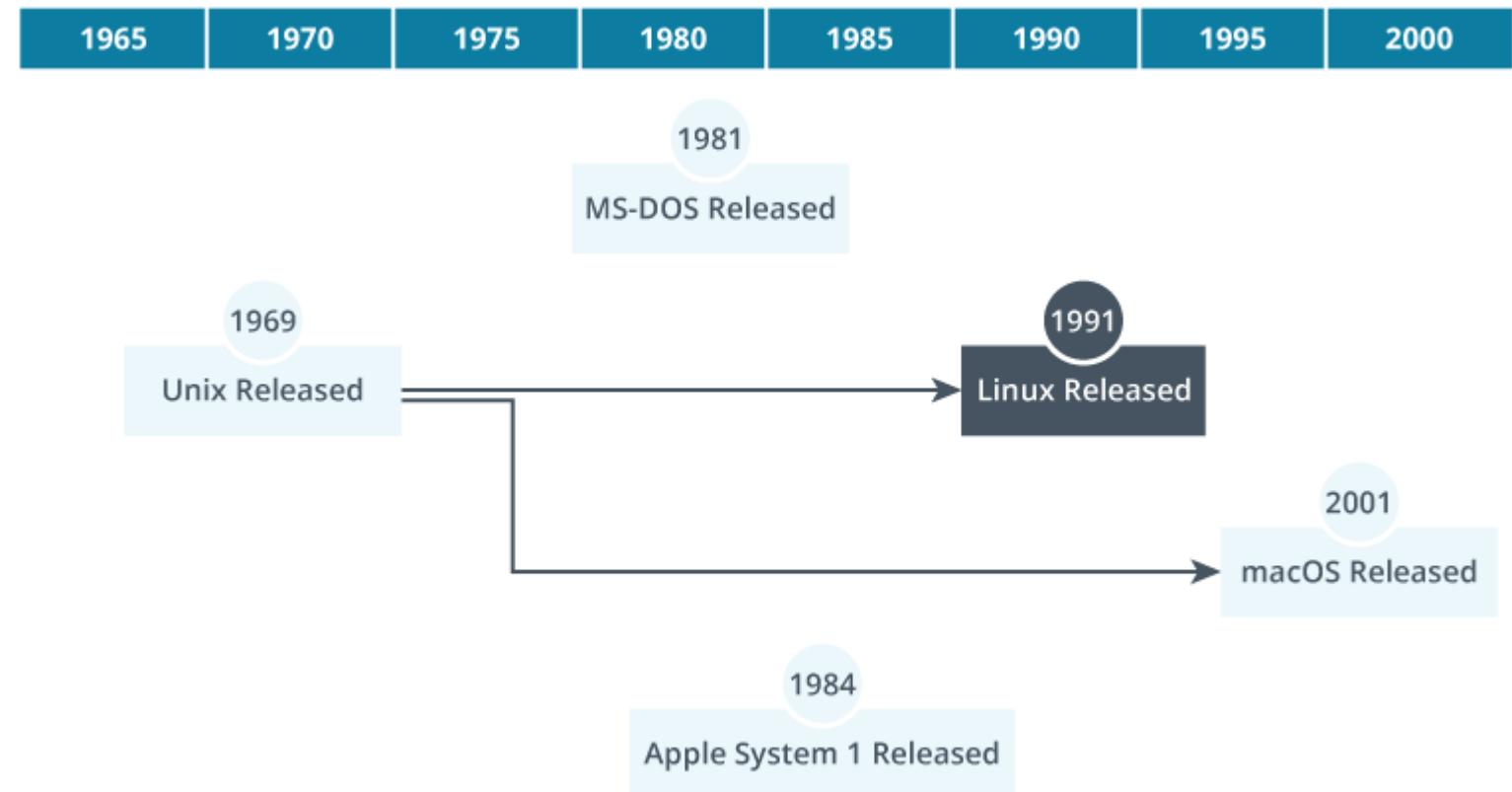
You can also group shell commands into files to execute as a program. Those files are called *shell scripts*.

Table 2 shows the Linux Shells

TABLE 1-2 Linux Shells

Shell	Description
ash	A simple, lightweight shell that runs in low-memory environments but has full compatibility with the bash shell
korn	A programming shell compatible with the Bourne shell but supporting advanced programming features like associative arrays and floating-point arithmetic
tcsh	A shell that incorporates elements from the C programming language into shell scripts
zsh	An advanced shell that incorporates features from bash, tcsh, and korn, providing advanced programming features, shared history files, and themed prompts

Timeline





Thanks

Q & A

Linux Architecture, Features & Vi Editor



Dr. Vimal Kr Baghel (Course Instructor), Assistant Professor
School of Computer Science Engineering & Technology (SCSET)
Bennett University Greater Noida

Outline

Operating System

Linux features and components

Linux Architecture

Shell

Vi Editor

Q & A

Operating System (OS)



What is an OS?



Why OS?



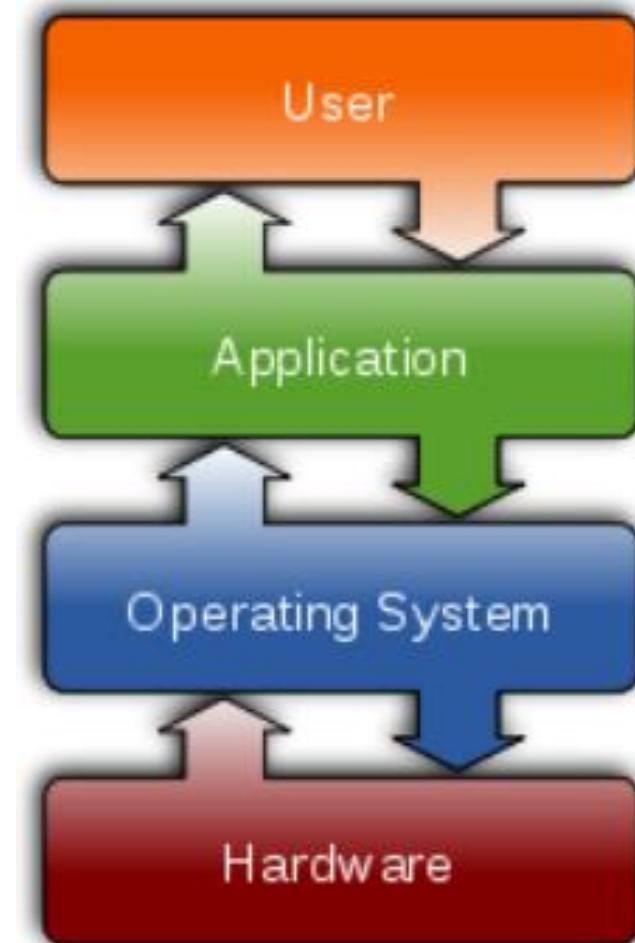
How it works?



Is there only one OS in a machine?

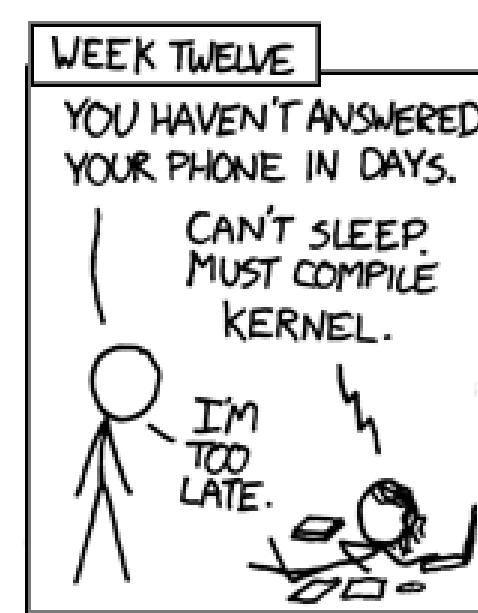
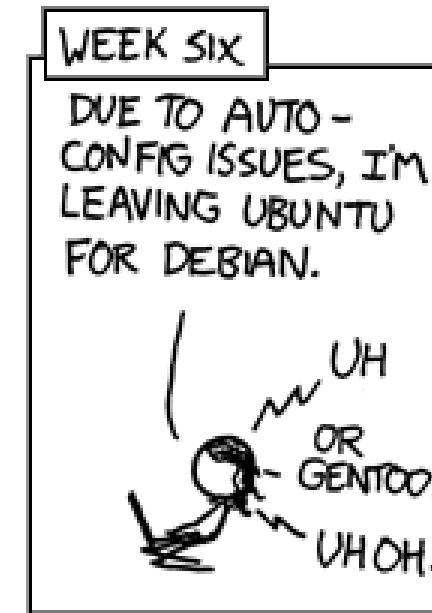
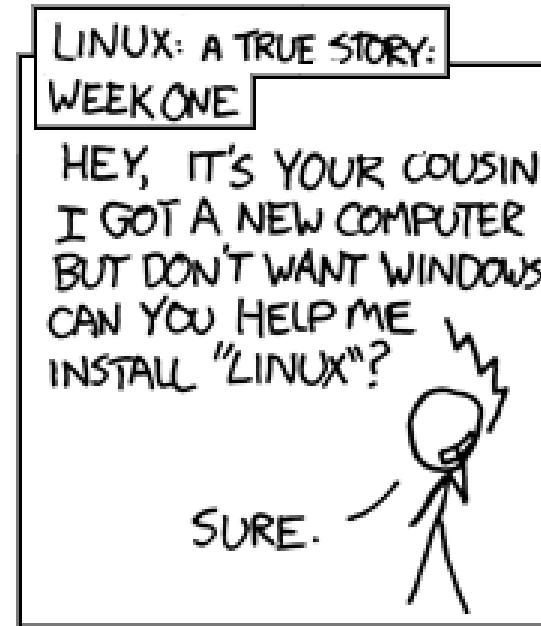


Types of OS?



On to Linux

- Courtesy XKCD.com



PARENTS: TALK TO YOUR
KIDS ABOUT LINUX..
BEFORE SOMEBODY ELSE DOES.

Linux

Linux: A kernel for a Unix-like operating system.

- commonly seen/used today in servers, mobile/embedded devices, ...

GNU: A "free software" implementation of many Unix-like tools

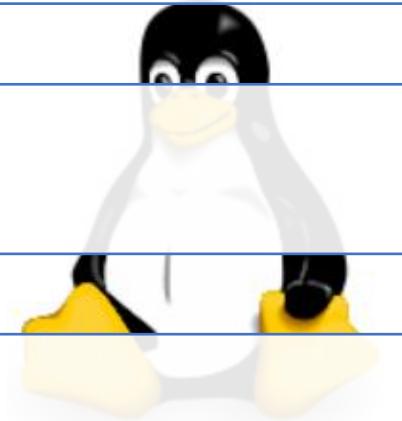
- many GNU tools are distributed with the Linux kernel

distribution: A pre-packaged set of Linux software.

- examples: Ubuntu, Fedora

key features of Linux:

- **open-source software:** source can be downloaded
- free to use
- constantly being improved/updated by the community



Linux Operating System

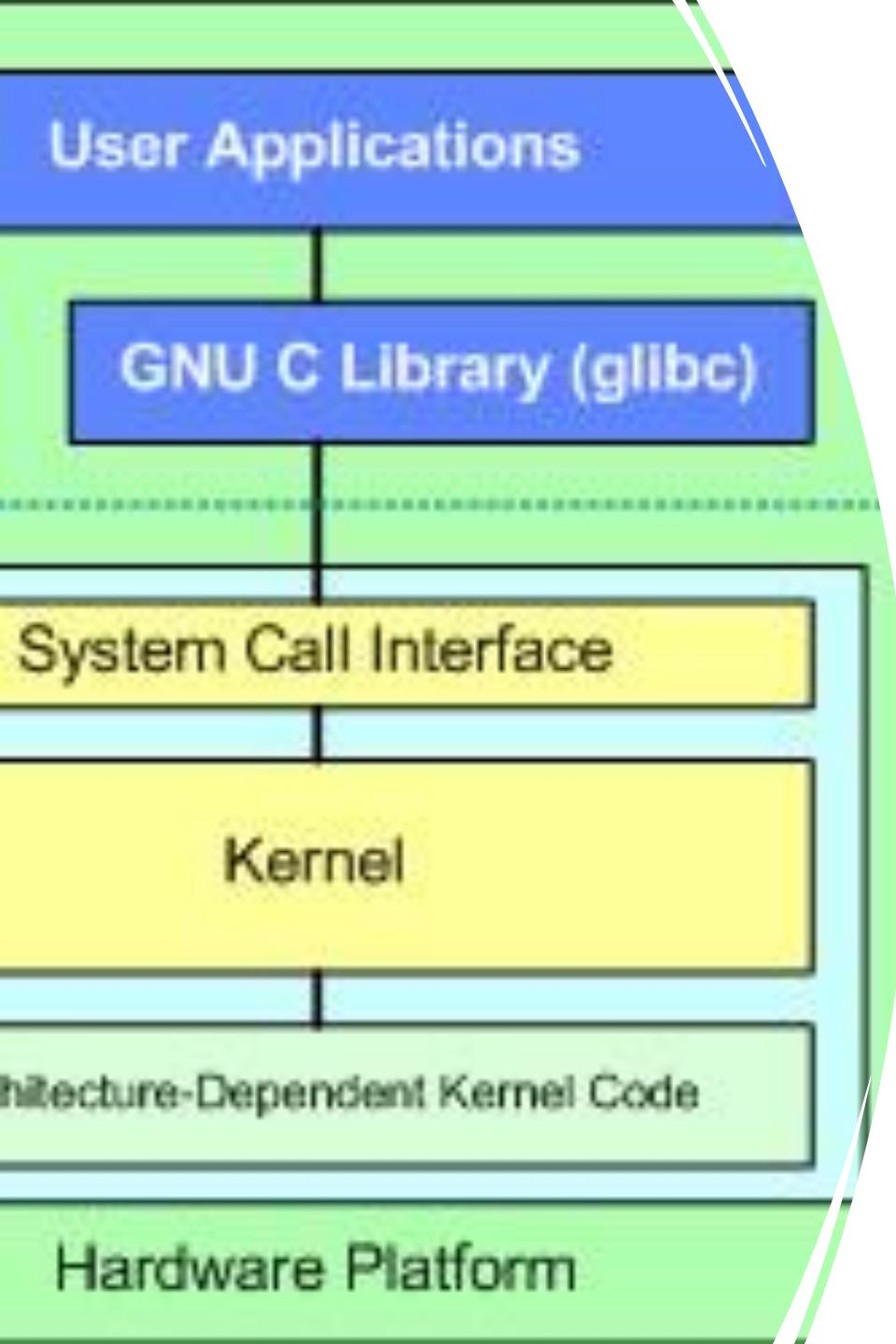
- Minix, the first open-source operating system, written by Andrew S. Tanenbaum in C, about 12000 lines of code.
- 1991, first Linux kernel written in C by **Linus Torvalds**, University of Helsinki, Finland.
- It was developed with the contribution of many programmers around the world.
- It is functionally like Unix (a clone).
- 1993 – FreeBSD 1.0 (Berkeley Unix), 1994 – RedHat Linux is introduced.
- 1999 – Linux available for PowerPC (Apple)
- Now – adopted by many companies and most universities, third world countries.
- Standard for parallel and high-performance computing (clusters).
- Available for most computers, including PDA, supports graphical user interfaces, networking, and has many applications.

Features of Linux?

- It's free!
 - the source code is also available, and anybody can write their own Linux if they include the source code in the distribution.
- Most users consider it a more stable and reliable OS than Windows.
- It's an alternative to Microsoft's dominance of the software market.
- It is multi-tasking, multi-user, and good support of multiple CPUs.
- Many utilities and APIs are now included in most distributions, like the g++ compiler, OpenGL, MPI, pthreads, etc.
- Mac OS now has an integrated shell and can run X11, Linux-specific applications.

Components of Linux

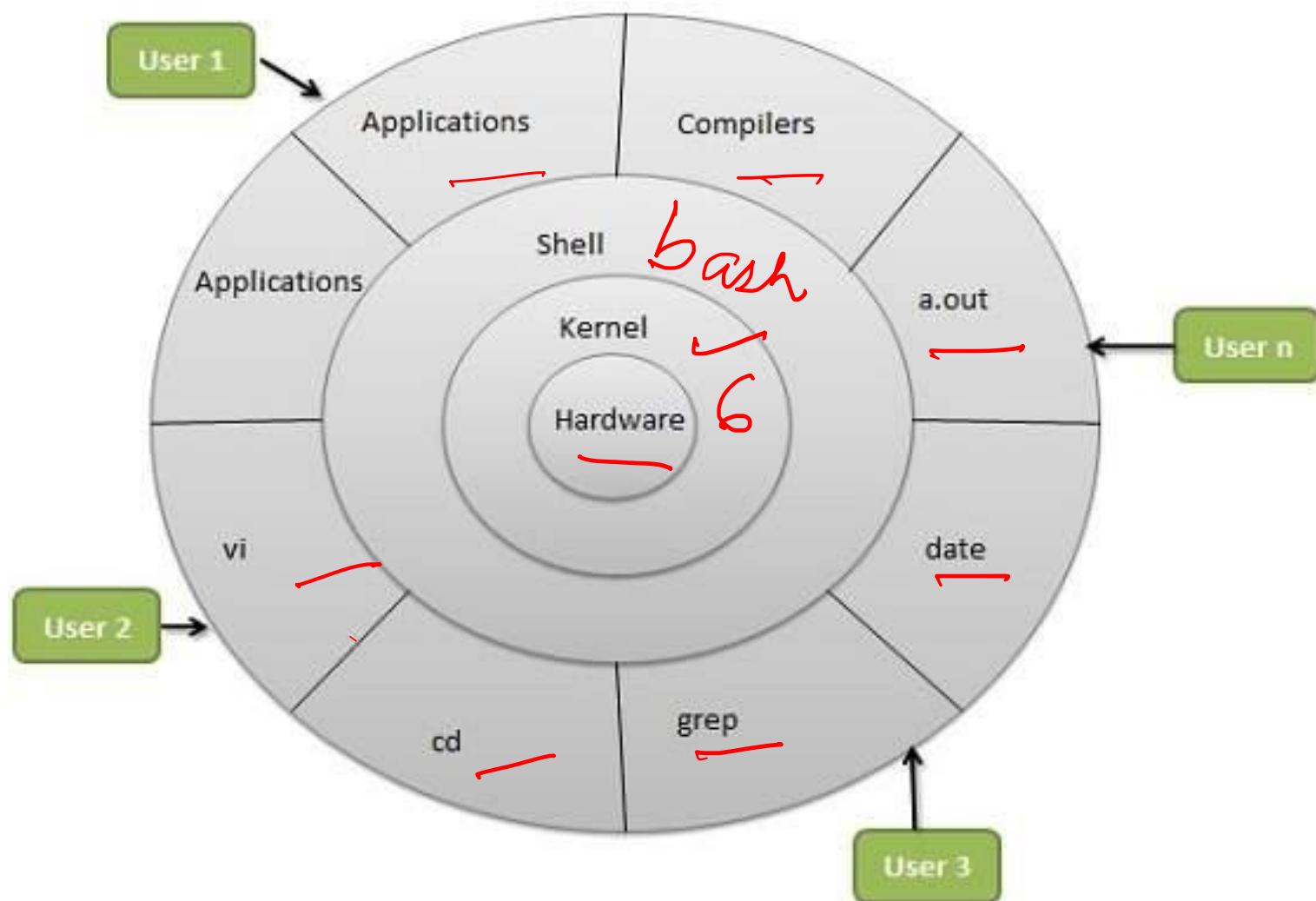
- The *kernel* – the core of the OS that controls the resources.
- A hierarchical file system (*FHS*)
- *Shells* – applications that interpret the commands from the user. They are active in the textual mode or terminal mode. Shells can also execute script files. Examples: bash, tcsh, zsh, sh, etc.
- *Graphical interfaces* – the X window system. Desktop interfaces: Gnome, KDE, fvwm, etc.
- *Specific libraries*: X11, gtk-glib-gnome, Qte, etc.



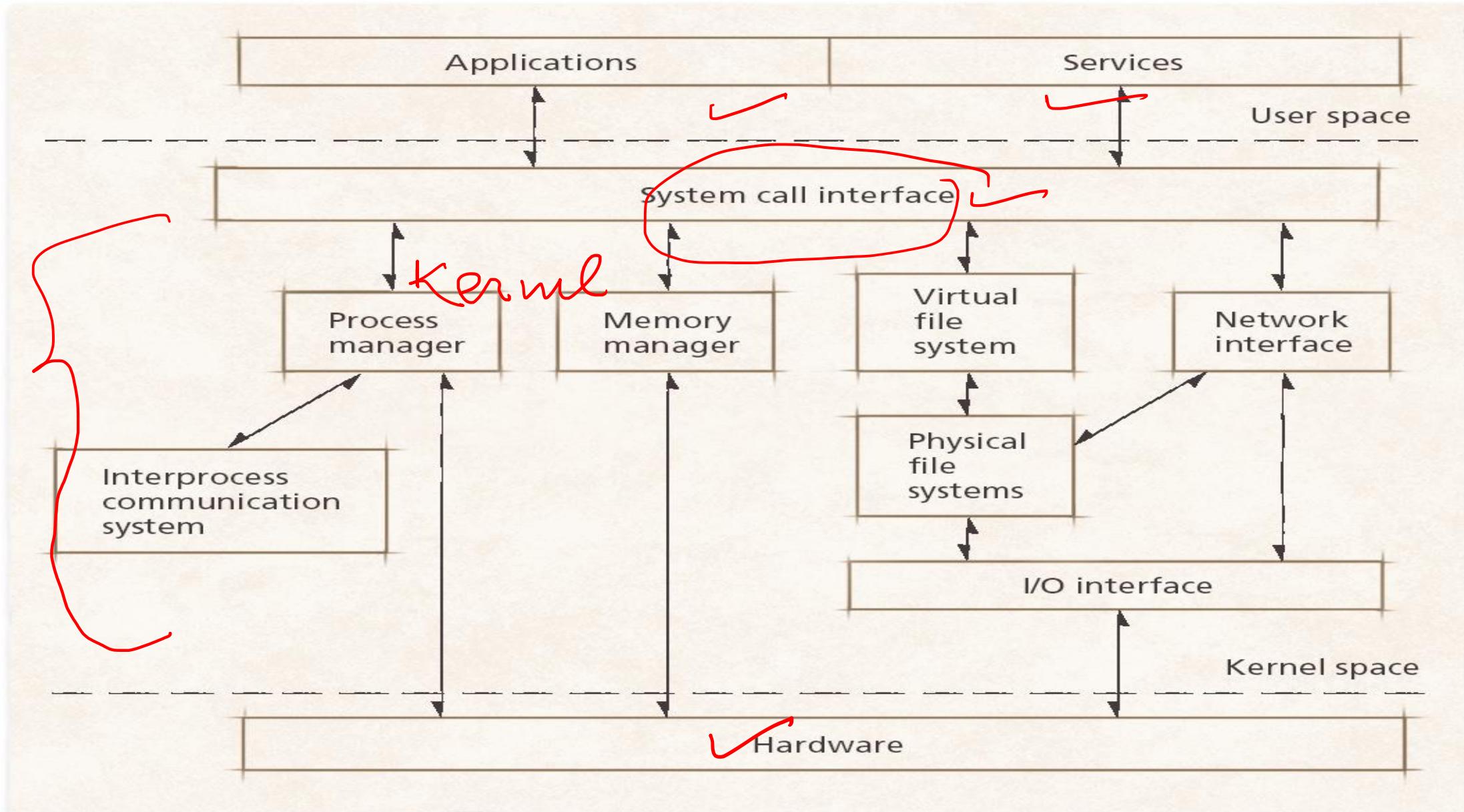
Linux Architecture

- Monolithic kernel
 - Contains modular components
- UNIX-based operating system
- Six primary subsystems:
 - Process management
 - Inter-process communication
 - Memory management
 - File system management
 - VFS: provides a single interface to multiple file systems
 - I/O management
 - Networking

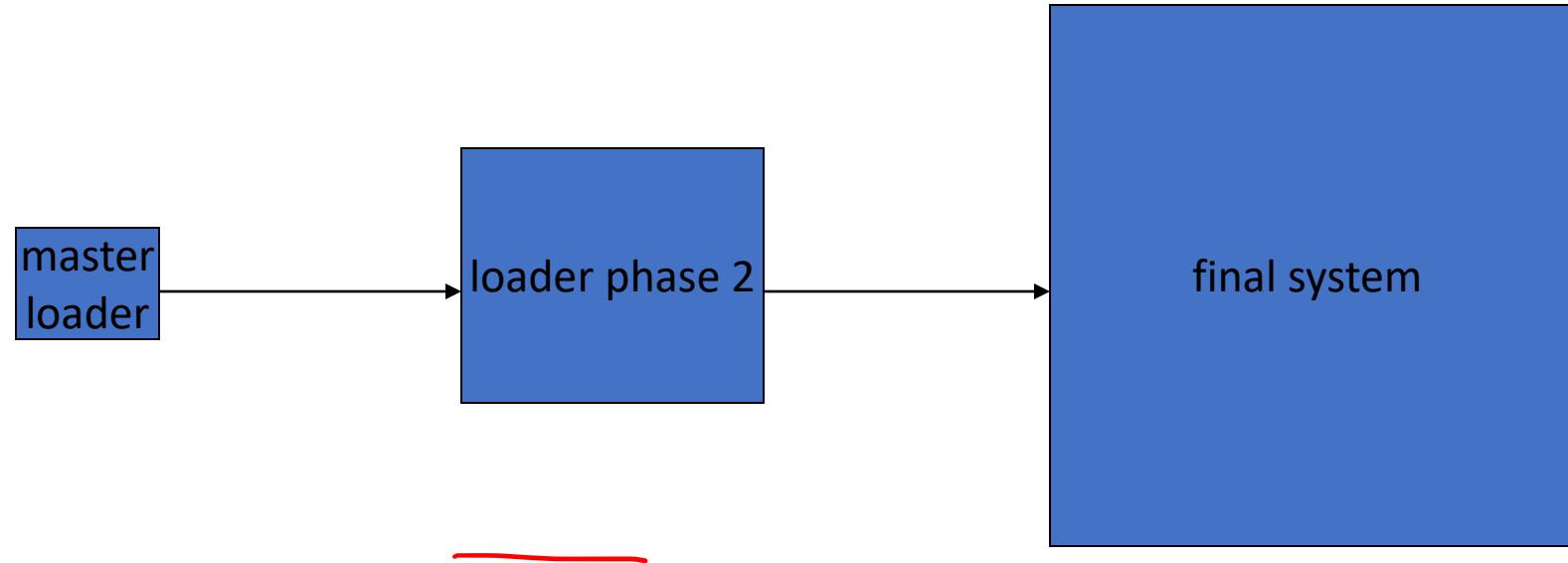
Linux Architecture



Linux Kernel Architecture

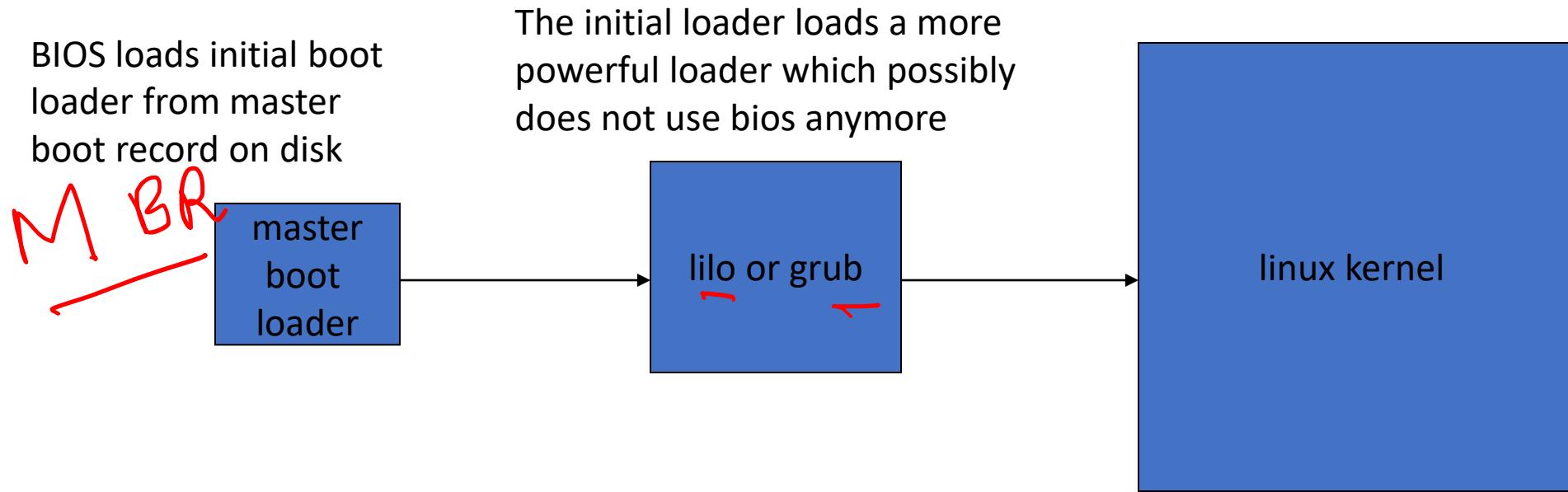


Bootstrapping



- bootstrapping is the process of starting a large and complicated system via a number of very small steps.
- A characteristic feature of bootstrapping is that the wonderful and powerful functions of a large system cannot be used to start the system itself – they are simply not yet available because the system is not running.

Loading Linux (1)



- Linux loaders are lilo or grub which are both found under /boot.
- The difference is that lilo knows exactly at which block of the partition the linux kernel starts and how big it is.
- It does NOT understand the linux filesystem and takes the information about the kernel from when the kernel was installed under /boot and lilo was re-run.
- Grub understands the filesystem and can locate the kernel within.

Loading Linux (2): System Check and Autoconfiguration

- ✓ During system start the following functions are performed:
 - determine CPU type, RAM etc.
 - stop interrupts and configure memory management and kernel stack
 - Initialize rest of kernel (buffers, debug etc.)
 - Start autoconfiguration of devices from configuration files and via probing hardware addresses.

- ✓ Probing is done via device driver routines.
- ✓ It means that certain memory locations are checked for the presence of a device.
- ✓ The system catches errors and then assumes that there is nothing mapped to this location.
- ✓ During regular operation, such errors would cause a kernel panic.

Loading Linux (3): Start processes

- ✓ ~~init~~: the first process and the only one started by the kernel itself. Starts other processes e.g. User is waiting for logins from terminals
- ✓ Swapper and other system processes (yes, the kernel depends on processes running in user mode)

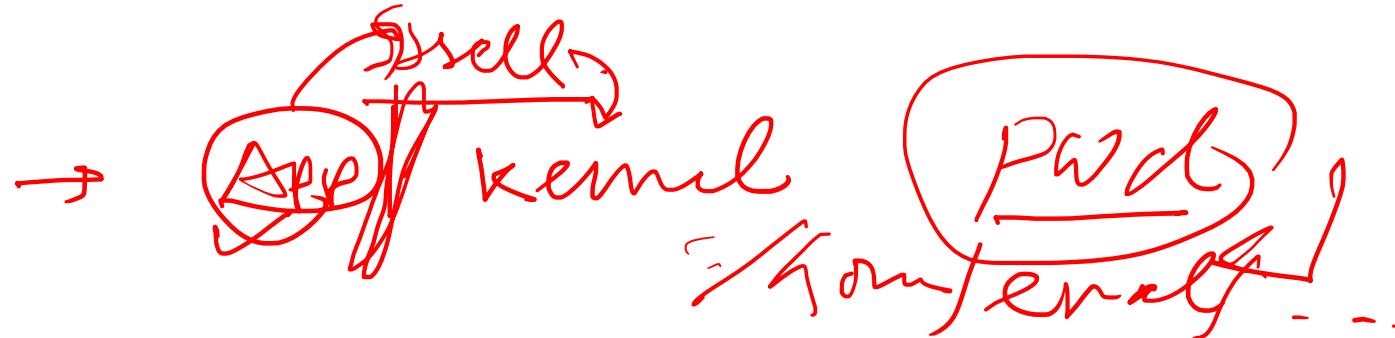
✓ Init is the parent of all processes. Killing it usually causes an immediate shutdown of the whole system.

Loading Linux (4): Go to runlevel

- ✓ System configuration scripts under /etc/rc.d/ are executed (shell scripts)
- ✓ Depending on the configured **runlevel** the system either boots into single-user mode or multi-user mode with or without networking and with or without X Window system. (The runlevel can be specified at kernel load-time)

- ✓ Shell scripts basically initialize the whole system once the kernel itself is running.

Shell



Shell is the user interface to the operating system



Functionality:

- ✓ Manage files (wildcards, I/O redirection)
- ✓ Manage processes (build pipelines, multitasking)



Most popular shells:

- The Bourne shell (sh)
- The Korn shell (ksh)
- The C shell (csh)
- The Bourne Again shell (bash)

Shell

- **The Bourne shell /bin/sh** (S. R. Bourne, 1977)
 - powerful syntactical language
 - strong in controlling input and output
 - expression matching facilities
 - ⌚ interactive use: the use of shell functions.
- **The C-shell /bin/csh** (Bill Joy, UCB, 1978)
 - ⌚new concepts: job control and aliasing
 - ⌚much better for interactive use
 - ⌚different input language:
 - out went the good control of input and output
 - too buggy to produce robust shell scripts

Shell

/bin/tsch (Ken Greer, Carnegie Mellon University, late 1970s)

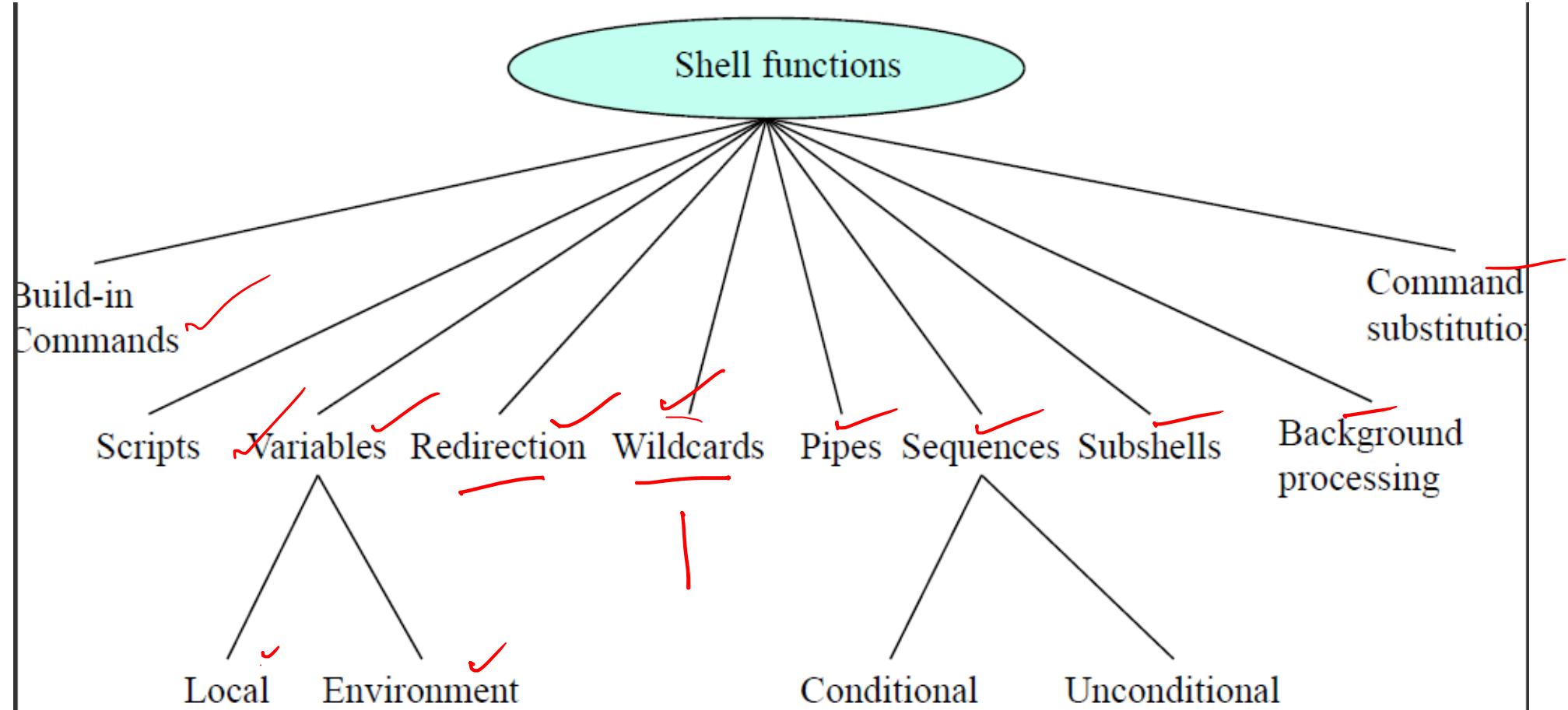
- User –oriented command line editing
- Out most of the bugs

Korn shell /bin/ksh (David Korn, AT&T, early 1980s)

- Bourne shell language
- C shell's features for interactive work
- You had to pay AT&T for it!

GNU project: a free shell=> **/bin/bash (the Bourne again shell)**

Core Shell Functionality



Selecting a Shell

```
[vimal@baghel ~]$ echo $SHELL
```

/bin/bash

```
[vimal@baghel ~]$ bash
```

```
[vimal@baghel ~]$ exit
```

exit

```
[vimal@baghel ~]$ ksh
```

\$ exit

```
[vimal@baghel ~]$ sh
```

sh-3.2\$ exit

exit

```
[vimal@baghel ~]$ csh
```

```
[vimal@baghel ~]$ exit
```

exit

\$Path

Changing shell

- To change your default shell use the ***chsh*** utility which requires full pathname of the new shell

```
[c33235@snowball ~]$ chsh
Changing shell for c33235.
Password: ✓
New shell [/bin/bash]: /bin/tcsh
Shell changed.
[c33235@snowball ~]$ █
```

What is vi ?

- The *visual editor* on the Unix.
- Before vi the primary editor used on Unix was the line editor
 - User was able to see/edit only one line of the text at a time
- The vi editor is not a text formatter (like MS Word, Word Perfect, etc.)
 - you cannot set margins
 - center headings
 - Etc...

Characteristics of vi

The vi editor is:

- a very powerful
- but at the same time it is cryptic
- It is hard to learn, specially for windows users

The best way to learn vi commands is to use them

So Practice...

Vim equals Vi

- The current iteration of **vi** for Linux is called **vim**
 - **Vi Improved**
 - <http://www.vim.org>



Starting vi

Type **vi <filename>** at the shell prompt

After pressing enter the command prompt disappears and you see tilde(~) characters on all the lines

These tilde characters indicate that the line is blank

Vi modes

There are two modes in vi

- Command mode
- Input mode

When you start vi by default it is in command mode

You enter the input mode through various commands

You exit the input mode by pressing the Esc key to get back to the command mode

How to exit from vi

First go to command mode

press **Esc** There is no harm in pressing **Esc** even if you are in command mode.
Your terminal will just beep and/or or flash if you press **Esc** in command mode



There are different ways to exit when you are in the command mode

How to exit from vi (comand mode)

:q <enter> is to exit, if you have not made any changes to the file

:q! <enter> is the forced quit, it will discard the changes and quit

:wq <enter> is for save and Exit

:x <enter> is same as above command

ZZ is for save and Exit (Note this command is uppercase)

The ! Character forces over writes, etc. **:wq!**

Moving Around

- You can move around only when you are in the command mode
- Arrow keys usually works(but may not)
- The standard keys for moving cursor are:
 - **h** - for left
 - **l** - for right
 - **j** - for down
 - **k** - for up

Moving Around

w - to move one word forward

b - to move one word backward

\$ - takes you to the end of line

<enter> takes the cursor to the beginning of next line

Moving Around

- - (minus) moves the cursor to the first character in the current line

H - takes the cursor to the beginning of the current screen(Home position)

L - moves to the Lower last line

M - moves to the middle line on the current screen

Moving Around

f - (find) is used to move cursor to a particular character on the current line

- For example, **fa** moves the cursor from the current position to next occurrence of ‘a’

F - finds in the reverse direction

Moving Around

) - moves cursor to the next sentence

} - move the cursor to the beginning of next paragraph

(- moves the cursor backward to the beginning of the current sentence

{ - moves the cursor backward to the beginning of the current paragraph

% - moves the cursor to the matching parentheses

Moving Around

Control-d scrolls
the screen down
(half screen)

Control-u scrolls
the screen up (half
screen)

Control-f scrolls
the screen forward
(full screen)

Control-b scrolls
the screen
backward (full
screen).

Entering text

- To enter the text in vi you should first switch to input mode
 - To switch to input mode there are several different commands
 - **a** - Append mode places the insertion point after the current character
 - **i** - Insert mode places the insertion point before the current character

Entering text

I - places the insertion point at the beginning of current line

o - is for open mode and places the insertion point after the current line

O - places the insertion point before the current line

R - starts the replace(overwrite) mode

Editing text

x - deletes the current character

d - is the delete command but pressing only d will not delete anything you need to press a second key

- **dw** - deletes to end of word
- **dd** - deletes the current line
- **d0** - deletes to beginning of line

There are many more keys to be used with delete command

The change command

c - this command deletes the text specified and changes the vi to input mode.
Once finished typing you should press <Esc> to go back to command mode

cw - Change to end of word

cc - Change the current line

There are many more options

Structure of vi command

The vi commands can be used followed by a number such as

- **n<command key(s)>**
- For example **dd** deletes a line **5dd** will delete five lines.

This applies to almost all vi commands

This how you can accidentally insert a number of characters into your document

Undo and repeat command

u - undo the changes made by editing commands

. (dot or period) repeats the last edit command

Copy, cut and paste in vi

yy - (yank) copy current line to buffer

n_{yy} - Where **n** is number of lines

p - Paste the yanked lines from buffer to the line below

P - Paste the yanked lines from buffer to the line above

(the paste commands will also work after the **dd** or **n_{dd}** command)

Creating a shell script using vi

Create a directory call **class**

Change into **class**

vi myscript.sh

inside the file enter following commands

- clear
- echo "=====
- echo "Hello World"
- echo "=====
- sleep 3
- clear
- echo Host is \$HOSTNAME
- echo User is \$USER

Creating a shell script using vi

- Save the file
- Change the permissions on myscript.sh
 - chmod 700 myscript.sh <enter>**
- Now execute myscript.sh
 - myscript.sh <enter>**
- Did the script run?
- Why not?
 - Hint, think about absolute vs relative path
 - Type **echo \$PATH** to see your PATH variable
 - Try this **./myscript.sh <enter>**
 - The **./** mean right here in this directory!



Thanks

Q & A

Linux File System



Dr. Vimal Kr Baghel (Course Instructor), Assistant Professor
School of Computer Science Engineering & Technology (SCSET)
Bennett University Greater Noida

Outline

Introduction to filesystem

Inode (index node)

Filesystem utilities

Q & A

Understanding the basic Linux filesystems

- Filesystem is used to store files and folders on a storage device
- Most Linux distros provide a default filesystem for us at installation time
- Each filesystem implements the **virtual directory structure** on storage devices

Understanding the basic Linux filesystems

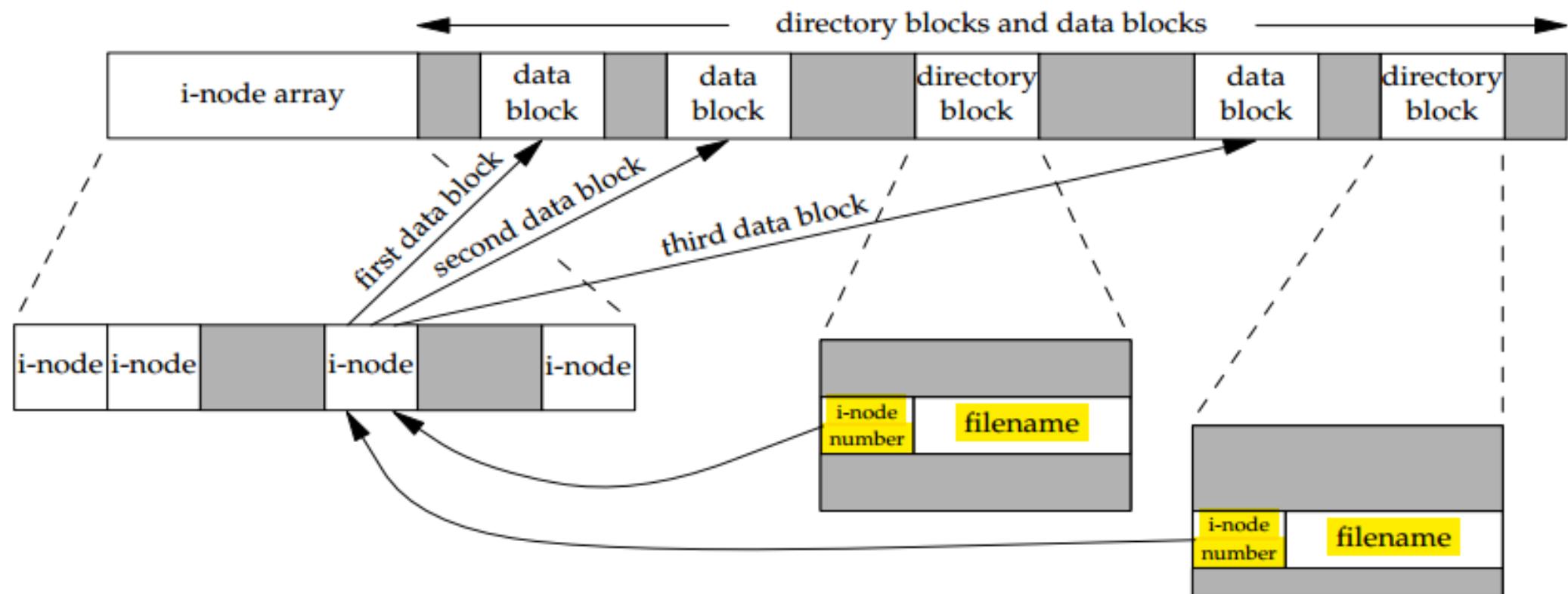
- The original filesystem introduced with the Linux is called the **extended filesystem (ext)**.
- Ext uses virtual directories to handle physical devices and storing data in **fixed-length blocks on the physical devices**.
- The ext filesystem uses ***inodes*** to track information about the files stored in the virtual directory.

inodes

- The inode system creates a separate table on each physical device, called the **inode table, to store file information.**
- Each stored file in the virtual directory has **an entry in the inode table.**
- The extended part of the name comes from the additional data that it tracks on each file, which consists of:
 - The filename
 - File Size (upto 2GB in ext & upto 3TB to 32 TB in ext2)
 - The owner of the file & the group the file belongs to
 - Access permissions for the file
 - Pointers to each disk block that contains data from the file

Inode

- The internal file representation is done through *inode*. *Inode* contains the description such as:
 - disk layout of file data,
 - Owner* of the file,
 - File access permissions*, and
 - Access times

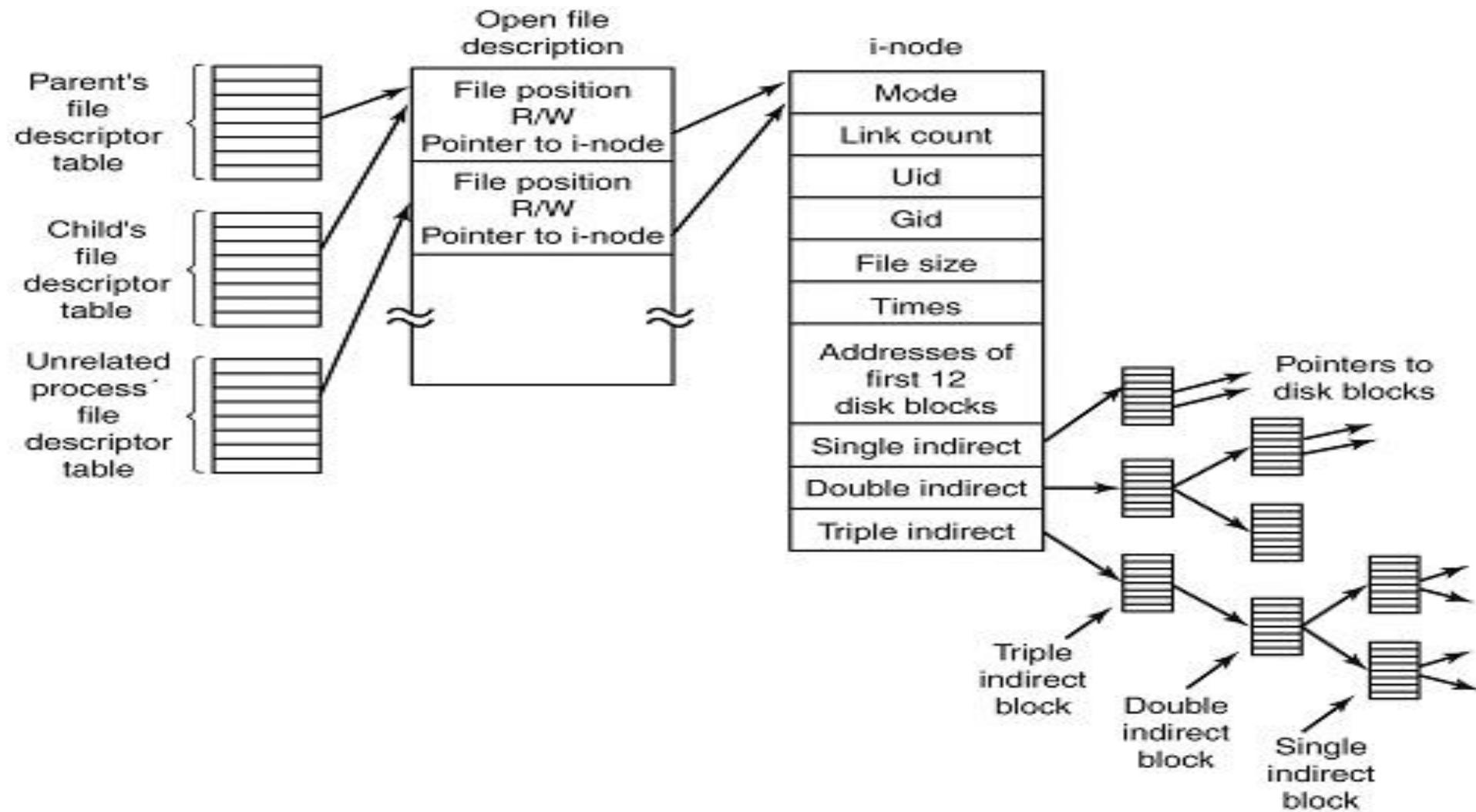


inode

- Each file has one inode, and can have multiple names (link)
- All names map onto its single inode
- Example
 - Open("/home/vimal/e1.sh", 1)
 - Kernel assigns an unused inode

```
$ ls -i *data_file
296890 data_file 296891
sl_data_file
$
```

Inode structure



A video on Inode Structure



Inode Structure



Understanding journaling filesystems

- *Journaling filesystems* provide a new level of safety to the Linux filesystem.
- Instead of writing data directly to the storage device and then updating the inode table, *journaling filesystems write file changes into a temporary file (*journal*) first.*
- After data is successfully written to the storage device and the inode table, **the journal entry is deleted.**
- In case of crash/power outage before the data can be written to the storage device, the journaling filesystem just reads through the journal file and processes any uncommitted data left over.

Journaling Methods

Method	Description
Data mode	Both inode and file data are journaled. Low risk of losing data, but poor performance.
Ordered mode	Only inode data is written to the journal, but not removed until file data is successfully written. Good compromise between performance and safety.
Writeback mode	Only inode data is written to the journal, no control over when the file data is written. Higher risk of losing data, but still better than not using journaling.

Working with Linux filesystem

- Creating partitions

- **\$ fdisk /dev/sdb**
- Unable to open /dev/sdb
- \$
- **\$ sudo fdisk /dev/sdb**
- [sudo] password for vimal:
- Device contains neither a valid DOS partition table,
- nor Sun, SGI or OSF disklabel
- Building a new DOS disklabel with disk identifier 0xd3f759b5.
- Changes will remain in memory only
- until you decide to write them.
- After that, of course, the previous content won't be recoverable.
- Warning: invalid flag 0x0000 of partition table 4 will
- be corrected by w(rite)
- [...]
- Command (m for help) :



Thanks

Q & A

File Handling Utilities



Dr. Vimal Kr Baghel (Course Instructor), Assistant Professor
School of Computer Science Engineering & Technology (SCSET)
Bennett University Greater Noida

Outline



Operations for filesystem management



Creating partitions using fdisk



Formatting the partitions



Mounting the Filesystem



Checking and repairing the partitions



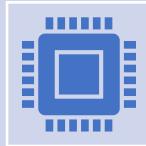
Q & A

Operations for Filesystem Management

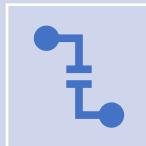


Disk is *partitioned* to have a filesystem on it.

The partition can be an entire disk or its subset.



After partitioning, the partitions are *formatted* so that Linux can use it



Checking and repairing a corrupted filesystem due to power loss/application lookup and system conflict

Disk Partitioning using fdisk

- Creating disk partitions using fdisk utility

- **\$ fdisk /dev/sdb**

- Unable to open /dev/sdb
- \$

- **\$ sudo fdisk /dev/sdb**

- sudo] password for vimal:
- Device contains neither a valid DOS partition table,
- nor Sun, SGI or OSF disklabel
- Building a new DOS disklabel with disk identifier 0xd3f759b5.
- Changes will remain in memory only
- until you decide to write them.
- After that, of course, the previous content won't be recoverable.
- Warning: invalid flag 0x0000 of partition table 4 will be corrected by w(rite)
- [...]
- Command (m for help):

TABLE 8-2 The fdisk Commands

Command	Description
a	Toggles a flag indicating if the partition is bootable
b	Edits the disklabel used by BSD Unix systems
c	Toggles the DOS compatibility flag
d	Deletes the partition
l	Lists the available partition types
m	Displays the command options
n	Adds a new partition
o	Creates a DOS partition table
p	Displays the current partition table
q	Quits without saving changes
s	Creates a new disklabel for Sun Unix systems
t	Changes the partition system ID
u	Changes the storage units used
v	Verifies the partition table
w	Writes the partition table to the disk
x	Advanced functions

Creating a Filesystem



Before you can store data on the partition, you must *format* it with a filesystem so Linux can use it.



Each filesystem type uses its *own command line program to format partitions*.

Creating a filesystem

- Formatting of filesystem partitions

TABLE 8-3 Command Line Programs to Create Filesystems

Utility	Purpose	
<code>mkefs</code>	Creates an ext filesystem	
<code>mke2fs</code>	Creates an ext2 filesystem	
<code>mkfs.ext3</code>	Creates an ext3 filesystem	\$ type mkfs.ext4 mkfs.ext4 is /sbin/mkfs.ext4
<code>mkfs.ext4</code>	Creates an ext4 filesystem	\$ \$ type mkfs.btrfs -bash: type: mkfs.btrfs: not found
<code>mkreiserfs</code>	Creates a ReiserFS filesystem	\$
<code>jfs_mkfs</code>	Creates a JFS filesystem	
<code>mkfs.xfs</code>	Creates an XFS filesystem	
<code>mkfs.zfs</code>	Creates a ZFS filesystem	
<code>mkfs.btrfs</code>	Creates a Btrfs filesystem	

Mounting the Filesystem



After you create the filesystem for a partition, the next step is to mount it on a virtual directory mount point so you can store data in the new filesystem.



You can mount the new filesystem anywhere in your virtual directory where you need the extra space.

```
$ ls /mnt
$ 
$ sudo mkdir /mnt/my_partition
$ 
$ ls -al /mnt/my_partition/
$ 
$ ls -dF /mnt/my_partition
/mnt/my_partition/
$ 
$ sudo mount -t ext4 /dev/sdb1 /mnt/my_partition
$ 
$ ls -al /mnt/my_partition/
total 24
drwxr-xr-x. 3 root root 4096 Jun 11 09:53 .
drwxr-xr-x. 3 root root 4096 Jun 11 09:58 ..
drwx----- 2 root root 16384 Jun 11 09:53 lost+found
$
```

Checking and Repairing a Filesystem

The fsck command is used to check and repair most Linux filesystem types- ext, ext2, ext3, ext4, Reiser4, JFS, and XFS.

The format of the command is:

- `fsck options filesystem`

Filesystems can be referenced using either the device name, the mount point in the virtual directory, or a special Linux UUID value assigned to the filesystem.

The fsck command uses the /etc/fstab file to automatically determine the filesystem on a storage device that's normally mounted on the system.

If the storage device isn't normally mounted (such as if you just created a filesystem on a new storage device), you need to use the -t command line option to specify the filesystem type.

Checking and Repairing a Filesystem

TABLE 8-4 The fsck Command Line Options

Option	Description
-a	Automatically repairs the filesystem if errors are detected
-A	Checks all the filesystems listed in the /etc/fstab file
-C	Displays a progress bar for filesystems that support that feature (only ext2 and ext3)
-N	Doesn't run the check, only displays what checks would be performed
-r	Prompts to fix if errors found
-R	Skips the root filesystem if using the -A option
-s	If checking multiple filesystems, performs the checks one at a time
-t	Specifies the filesystem type to check
-T	Doesn't show the header information when starting
-V	Produces verbose output during the checks
-y	Automatically repairs the filesystem if errors detected



Thanks

Q & A

Understanding Linux Security



Dr. Vimal Kr Baghel (Course Instructor), Assistant Professor
School of Computer Science Engineering & Technology (SCSET)
Bennett University Greater Noida

Outline



Linux Security



Password file



Shadow file



Q & A

Understanding Linux File Permissions

- We need mechanism to protect files against unauthorized access ?
- The Linux system follows the Unix method of file permissions, allowing individual users and groups access to files based on a set of security settings for each file and directory.

Linux Security

The core of the Linux security system is the *user account*

The permissions are based on *user account*, and are tracked with *numeric UID*

Login name of *8 characters* or less

The Linux system uses special files and utilities to track and manage user accounts on the system to understand how to use them when working with file permissions.

How Linux handles user accounts?

The /etc/passwd file

To match the login name to a corresponding UID value.

UID is 0 for root

System user account

Linux reserves UIDs below 500 for system accounts.

First UID starts from 501 usually

The /etc/passwd file



Every service that runs in background on a Linux server has its own user account to log in with. **Why?**



The /etc/passwd file contains much more than just the login name and UID for the user.



The /etc/passwd file is a standard text file.



We can use any text editor to manually perform user management functions such as adding, modifying, or removing user accounts directly in the /etc/passwd file.

Field in /etc/passwd

- The fields of the /etc/passwd file contain the following information:
 - The login username
 - The password for the user
 - The numerical UID of the user account
 - The numerical group ID (GID) of the user account
 - A text description of the user account (called the comment field)
 - The location of the HOME directory for the user
 - The default shell for the user

The /etc/passwd file

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/sbin/nologin
daemon:x:2:2:daemon:/sbin:/sbin/nologin
adm:x:3:4:adm:/var/adm:/sbin/nologin
lp:x:4:7:lp:/var/spool/lpd:/sbin/nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
news:x:9:13:news:/etc/news:
uucp:x:10:14:uucp:/var/spool/uucp:/sbin/nologin
operator:x:11:0:operator:/root:/sbin/nologin
games:x:12:100:games:/usr/games:/sbin/nologin
gopher:x:13:30:gopher:/var/gopher:/sbin/nologin
ftp:x:14:50:FTP User:/var/ftp:/sbin/nologin
nobody:x:99:99:Nobody:/:/sbin/nologin
rpm:x:37:37::/var/lib/rpm:/sbin/nologin
vcsa:x:69:69:virtual console memory owner:/dev:/sbin/nologin
mailnull:x:47:47::/var/spool/mqueue:/sbin/nologin
smmsp:x:51:51::/var/spool/mqueue:/sbin/nologin
apache:x:48:48:Apache:/var/www:/sbin/nologin
rpc:x:32:32:Rpcbind Daemon:/var/lib/rpcbind:/sbin/nologin
ntp:x:38:38::/etc/ntp:/sbin/nologin
nscd:x:28:28:NSCD Daemon:/:/sbin/nologin
tcpdump:x:72:72::/sbin/nologin
dbus:x:81:81:System message bus:/:/sbin/nologin
avahi:x:70:70:Avahi daemon:/:/sbin/nologin
hsqldb:x:96:96::/var/lib/hsqldb:/sbin/nologin
sshd:x:74:74:Privilege-separated SSH:/var/empty/sshd:/sbin/nologin
rpcuser:x:29:29:RPC Service User:/var/lib/nfs:/sbin/nologin
nfsnobody:x:65534:65534:Anonymous NFS User:/var/lib/nfs:/sbin/nologin
haldaemon:x:68:68:HAL daemon:/:/sbin/nologin
xfs:x:43:43:X Font Server:/etc/X11/fs:/sbin/nologin
gdm:x:42:42::/var/gdm:/sbin/nologin
rich:x:500:500:Rich Blum:/home/rich:/bin/bash
mama:x:501:501:Mama:/home/mama:/bin/bash
katie:x:502:502:katie:/home/katie:/bin/bash
jessica:x:503:503:Jessica:/home/jessica:/bin/bash
mysql:x:27:27:MySQL Server:/var/lib/mysql:/bin/bash
$
```

/etc/shadow

- Most Linux systems hold user passwords in a separate file
- Only root can use it
- The /etc/shadow file contains one record for each user account on the system.
- A record looks like this:
 - rich:\$1\$.FfcK0ns\$f1UgiyHQ25wrB/hykCn020:11627:0:99999:7:::

Fields in /etc/shadow file

- There are nine fields in each /etc/shadow file record:
 - The login name corresponding to the login name in the /etc/passwd file
 - The encrypted password
 - The number of days since January 1, 1970, that the password was last changed
 - The minimum number of days before the password can be changed
 - The number of days before the password must be changed
 - The number of days before password expiration that the user is warned to change the password
 - The number of days after a password expires before the account will be disabled
 - The date (stored as the number of days since January 1, 1970) since the user account was disabled
 - A field reserved for future use



Thanks

Q & A



BENNETT
UNIVERSITY
THE TIMES GROUP

Security by File Permissions in Linux

Dr. Vimal Baghel, Assistant Professor, SCSET, BU

Outline

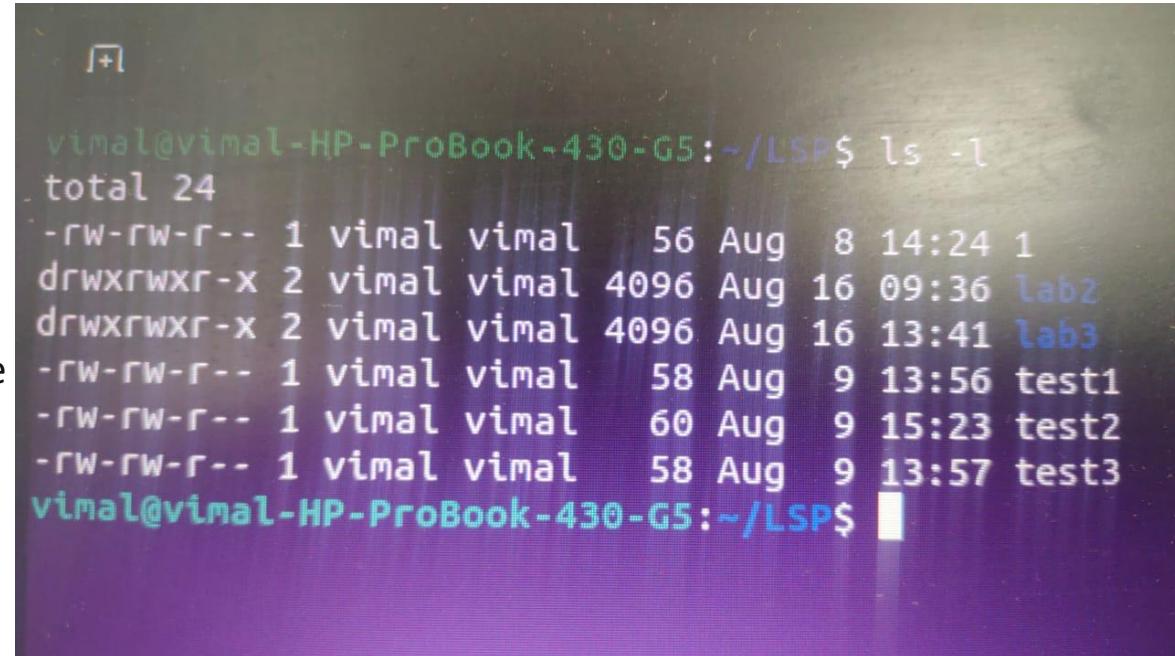
- Understanding Linux file permissions
- How do you view Linux file permissions?
- How do you read file permissions?
- What are octal values?
- What do Linux file permissions actually do?
- How do directory permissions work?
- How do you modify Linux file permissions?
- What are special file permissions?
- Q & A

Understanding Linux file Permissions

- File permissions are core to the security model used by Linux systems.
 - determine who can access files and directories on a system and how.
- **Permission Groups**
 - Owner (u)
 - Group (g)
 - Others (o)
 - All Users (a)
- **Permission Types**
 - Read (r)
 - Write (w)
 - Execute (x)

Viewing the Permissions

- \$ ls -l, output is in the format
 - **_rwxrwxrwx. 1 owner:group**
 - File type: -
 - Permission settings: rw-r--r--
 - Extended attributes: dot (.)
 - Numeric value (e.g.) 1: Number of hard links to the file
 - User owner: vimal
 - Group owner: vimal
- What are octal values?
 - r = 4, w = 2, x = 1



```
vimal@vimal-HP-ProBook-430-G5:~/LSP$ ls -l
total 24
-rw-rw-r-- 1 vimal vimal 56 Aug  8 14:24 1
drwxrwxr-x 2 vimal vimal 4096 Aug 16 09:36 lab2
drwxrwxr-x 2 vimal vimal 4096 Aug 16 13:41 lab3
-rw-rw-r-- 1 vimal vimal 58 Aug  9 13:56 test1
-rw-rw-r-- 1 vimal vimal 60 Aug  9 15:23 test2
-rw-rw-r-- 1 vimal vimal 58 Aug  9 13:57 test3
vimal@vimal-HP-ProBook-430-G5:~/LSP$
```

What happens when we interact with a file?

- When the system is looking at a file's permissions to determine what information to provide you when you interact with a file, it runs through a series of checks:
 1. It first checks to see whether you are the user that owns the file.
 2. If you are not the user that owns the file, next your group membership is validated to see whether you belong to the group that matches the group owner of the file
 3. ~~"Others"~~ permissions are applied when the account interacting with the file is neither the user owner nor in the group that owns the files.

The three fields are mutually exclusive: You can not be covered under more than one of the fields of permission settings on a file.

Principle of Least Privilege

Only grant users, applications, and processes as much access as they need, and no more!

- **Proper example:** UserA needs to be able to read file1, so they are granted the read permission but not the write permission.
- **Improper example:** UserA needs to be able to read file2, but they are granted read, write, and execute permissions and could potentially change the file.
- **Proper example:** UserA needs to perform standard system activities, such as creating files associated with their own job, and does not need to perform system administrative tasks, so the user logs on with a standard user account.
- **Improper example:** UserA needs to perform standard system activities, such as creating files associated with their own job, and does not need to perform system administrative tasks, so the user logs on with the root user account.

Recognizing Access Levels

Files

- **Read:** The ability to access and view the contents of a file.
- **Write:** The ability to save changes to a file.
- **Execute:** The ability to run a script, program, or other software file.

Directories

- **Read:** The ability to list the contents of a directory.
- **Write:** The ability to create, rename, and delete files in a directory. Requires the execute attribute to also be set.
- **Execute:** The ability to access a directory, execute a file from that directory, or perform a task on that directory.

Recognizing Access Identities



User (u)

- Single account designated as the owner.
- By default, the resource creator.

Group (g)

- Single group associated with the file or directory.

Others (o)

- All users and groups who are not the designated user (owner) or associated group.

Display Permissions

```
student@ubuntu20:/projects$ ls -l
total 4
drwxr-xr-x 2 root      root          4096 Dec  9 11:11 old-projects
-rwxrw-r-- 1 root      root          0 Dec  9 11:11 project1.txt
-rw-rw---- 1 student5  2022projects  0 Dec  9 11:11 project2.txt
-r----- 1 student9  2022projects  0 Dec  9 11:11 project3.txt
student@ubuntu20:/projects$
```

When Permissions Are Important?

- ***home directories***- drwx_____ (700)
 - \$ chmod 700 /home/user1.
- ***bootloader configuration files***- If you decide to implement password to boot specific operating systems then you will want to remove read and write permissions from the configuration file from all users but root. To do you can change the permissions of the file to 700.
- ***system and daemon configuration files***- It is very important to restrict rights to system and daemon configuration files to restrict users from editing the contents, it may not be advisable to restrict read permissions, but restricting write permissions is a must. (644).
- ***firewall scripts*** – It may not always be necessary to block all users from reading the firewall file, but it is advisable to restrict the users from writing to the file. (700)

Key Demonstration: Display Permissions



1. Sign in to at least one system (either RH or Debian-based) and then demonstrate how to display permissions. Explain the output.
2. Be in the logged-in user's home directory.
3. If necessary, create two-three text files by using the touch command: `touch fileA fileB`
4. Run `ls -l` to display permissions
5. Explain each permission field for the user, group, and others
6. Identify the owner and group field

Advanced Permissions

- **_** – no special permissions
- **d** – directory
- **I** – The file or directory is a symbolic link
- **s** – This indicated the setuid/setgid permissions.
- **t** – This indicates the sticky bit permissions.

Interpret Permissions Strings (slide 1/4)

First field

- File (-) or Directory (d)

```
student@ubuntu20:/projects$ ls -l
total 4
drwxr-xr-x 2 root      root          4096 Dec  9 11:11 old-projects
-rwxrw-r-- 1 root      root          0 Dec  9 11:11 project1.txt
-rw-rw---- 1 student5  2022projects   0 Dec  9 11:11 project2.txt
-r----- 1 student9  2022projects   0 Dec  9 11:11 project3.txt
student@ubuntu20:/projects$
```

The output of the 'ls -l' command shows the following file listing:

- The first column shows the file type and permissions:
 - 'd' indicates a directory ('old-projects').
 - '-' indicates a regular file ('project1.txt', 'project2.txt', 'project3.txt').
- The second column shows the number of links (2 for directories, 1 for files).
- The third and fourth columns show the owner and group names.
- The fifth column shows the file size (4096 for the directory, 0 for files).
- The sixth column shows the date the file was last modified.
- The seventh column shows the time the file was last modified.
- The eighth column shows the file name ('old-projects', 'project1.txt', 'project2.txt', 'project3.txt').

Annotations with red arrows highlight the first field (permissions) for the directory and the first field (permissions) for the first file ('project1.txt').

Interpret Permissions Strings (slide 2/4)

Second, Third, and Fourth Fields

- Permissions for the User identity
- Read (r), Write (w), Execute (x), or none (-)

```
student@ubuntu20:/projects$ ls -l
```

```
total 4
```

drwxr-xr-x	2	root	root	4096	Dec 9	11:11	old-projects
-rwxrw-r--	1	root	root	0	Dec 9	11:11	project1.txt
-rw-rw----	1	student5	2022projects	0	Dec 9	11:11	project2.txt
-r-----	1	student9	2022projects	0	Dec 9	11:11	project3.txt

```
student@ubuntu20:/projects$
```

Interpret Permissions Strings (slide 3/4)

Fifth, Sixth, and Seventh Fields

- Permissions for the Group identity
- Read (r), Write (w), Execute (x), or none (-)

```
student@ubuntu20:/projects$ ls -l
```

```
total 4
```

drwxr-xr-x	2	root	root	4096	Dec 9	11:11	old-projects
-rwxrw-r--	1	root	root	0	Dec 9	11:11	project1.txt
-rw-rw----	1	student5	2022projects	0	Dec 9	11:11	project2.txt
-r-----	1	student9	2022projects	0	Dec 9	11:11	project3.txt

```
student@ubuntu20:/projects$
```

Interpret Permissions Strings (slide 4/4)

Eighth, Ninth, and Tenth Fields

- Permissions for the Others identity
- Read (r), Write (w), Execute (x), or none (-)

```
student@ubuntu20:/projects$ ls -l
total 4
drwxr-xr-x 2 root      root          4096 Dec  9 11:11 old-projects
-rwxrw-r-- 1 root      root          0 Dec  9 11:11 project1.txt
-rw-rw---- 1 student5  2022projects  0 Dec  9 11:11 project2.txt
-r----- 1 student9  2022projects  0 Dec  9 11:11 project3.txt
student@ubuntu20:/projects$
```

Set Permissions with Absolute and Symbolic Modes

Absolute mode

- Uses octal values
- Ex: chmod 764 file1

Set



Note that you must be able to interpret both modes!

GT

Symbolic mode

- Uses characters
- Ex: chmod u+r file1

u + r

? ?

Absolute Mode

Octal Values

- Read = 4
- Write = 2
- Execute = 1
- Displayed with three fields,
with permissions values added.

Example

- user has rwx, or $4+2+1 = 7$
- group has r-x, or $4+0+1 = 5$
- other has ---, or $0+0+0 = 0$
- Permissions: 750 for ugo
- Syntax to set this access level:
 - `chmod 750 file1`

Symbolic Mode

Characters:

- Read = r, write = w, execute = x
- User = u, group = g, others = o

Operators:

- + grants a permission,
- - removes a permission
- = sets a permission

• Examples:

- Give r to o for file1:
`chmod o+r file1`
- Remove r for o for file1:
`chmod o-r file`
- Give rw to go for file1:
`chmod go+rw file1`

Key Demonstration: Absolute and Symbolic Modes (slide 1)



Use `chmod` to demonstrate absolute and symbolic modes. Display the permissions changes using `ls -l` and interpret the new permissions.

1. Be in the logged-in user's home directory
2. Create a directory in the logged-in user's home folder named Permissions: `mkdir Permissions`
3. Change into the Permissions directory: `cd Permissions`
4. Create dirA and file1: `mkdir dirA` then run `touch file1`
5. Use `ls -l` to display current permissions

(continued on next slide)



(continued from previous slide)

6. Use absolute mode to see rwx for ugo on file1: chmod 777 file1
7. Display the new permissions: ls -l
8. Use absolute mode to set rwx for the user and group, and no access for others on dirA: chmod 770 dirA
9. Display the new permissions: ls -l

Use the chown Command

Example 1: Change the owner but not the group:

- `chown newowner filename`

Example 2: Change the owner and the group:

- `chown newowner:newgroup filename`

Example 3: Change the group but not the owner:

- `chown :newgroup filename`

Modifying the Permissions

- Let file1 has the permissions as `_rw_rw_rw`. We want to remove the read and write permissions from the all-users: `$ chmod a-rw file1`
- To add the permissions again: `$ chmod a+rw file1`
- Using Binary References to Set permissions: `$ chmod 640 file1`
- *Other Ways:*
 - `$ chmod ug+rwx example.txt`
 - `$ chmod o+r example2.txt`

Owners and Groups

- Syntax: **\$ chown owner:group filename,**
 - *Example: \$ chown user1:family file1.*
- the **chgrp** command can be used to change the group ownership of a file.

Key Demonstration: Using chown and chgrp



Sign in to at least one system (either RH or Debian-based) and then use `chown` to demonstrate changing owners, groups, and both on a file. Demonstrate the `chgrp` command, too. Use `ls -l` to display the changes.

1. Be in the logged-in user's home directory.
2. If necessary, create two or three text files by using the touch command: `touch fileA fileB`
3. If necessary, create a user account and a group for demonstration purposes: `useradd {USERA}` and then run `groupadd {GROUP1}`
4. Demonstrate the use of `chown` by changing the associated file owner and group: `chown {USERA}:{GROUP1} fileA`
5. Demonstrate the use of `chgrp` by changing the associated group: `chgrp {GROUP1} fileB`

File Attributes

- Apart from the file mode bits that control **user and group** read, write and execute permissions, several **file systems** support file attributes that enable further customization of allowable file operations.
- The **e2fsprogs** package contains the programs **lsattr(1)** and **chattr(1)** that list and change a file's attributes, respectively.
- These are a few useful attributes. Not all filesystems support every attribute.
 - **a - append only:** File can only be opened for appending.
 - **c - compressed:** Enable filesystem-level compression for the file.
 - **e – extent:** the file is using extents for mapping the blocks on disk.
 - **i - immutable:** Cannot be modified, deleted, renamed, linked to. Can only be set by root.
 - **j - data journaling:** Use the **journal** for file data writes as well as metadata.
 - **m - no compression:** Disable filesystem-level compression for the file.
 - **A - no atime update:** The file's atime will not be modified.
 - **C - no copy on write:** Disable copy-on-write, for filesystems that support it.

Use Attribute Management Commands (slide 1)

- To display attributes for file1:

```
lsattr file1
```

```
student@ubuntu20:~/Documents$ lsattr file2
-----i-----e----- file2
student@ubuntu20:~/Documents$
```

Use Attribute Management Commands (slide 2)

- To set the immutable flag attribute for file1:

```
chattr +i file1
```

```
student@ubuntu20:~/Documents$ sudo chattr +i file3
student@ubuntu20:~/Documents$ lsattr file3
-----i-----e---- file3
student@ubuntu20:~/Documents$
```

Troubleshooting Permissions

Is the user a member of the sales group?

- Confirm the user's membership in sales with the group or id commands
- Confirm the permissions applied to sales for the file by using ls -l

Are the permissions set correctly?

- Display permissions with ls -l
- Check permissions of file
- Are permissions set recursively from parent directory?
- Use su to test access
- Reapply permissions, being careful of your absolute or symbolic mode syntax



Review Activity: Standard Linux Permissions

1. How does the principle of least privilege help mitigate threats and mistakes?
2. What octal value is used in absolute mode to set permissions at all access for all identities?
3. Write the command by using symbolic mode that removes the read permission from others for fileA without impacting other permissions.
4. Interpret the results of the following command: chown -R USERA:sales dirA

Understand SUID and SGID

SUID

- Allows the specified user to have similar permissions to the owner without being the owner.
- Files or commands can be executed as the owner.

SGID

- Allows the group associated with a directory to be inherited by files created in the directory.

What are special file permissions?

- The **setuid/setgid** permissions are used **to tell the system to run an executable as the owner with the owner's permissions.**
 - Be careful using setuid/setgid bits in permissions. If you incorrectly assign permissions to a file owned by root with the setuid/setgid bit set, then you can open your system to intrusion.
- You can only assign the setuid/setgid bit by explicitly defining permissions. The character for the setuid/setgid bit is **s**.
- So do set the setuid/setgid bit on file2.sh you would issue the command \$ **chmod g+s file2.sh**.

Key Demonstration: Configure with SGID



Sign in to at least one system (either RH or Debian-based) and then configure a directory with SGID, allowing files created in this directory by different users retain the directory's group association.

1. Be in the logged-in user's home directory.
2. If necessary, create a directory at the root of the filesystem named /projects, create one standard user named **USERA**, and one group named **MANAGERS**
3. Display the new directory's permissions: `ls -l /`
4. Set the SGID permission on the new /projects directory for the group: `chmod g+s /projects`
5. Use `su - USERA`, create a file named test in the /projects directory, and then confirm the group association was inherited: `ls -l /projects`

Understand the Sticky Bit



- Protects files within a directory.
 - Only the file owner or root can delete the file.
 - Without the Sticky Bit, any user with write or execute could potentially delete the file.

Sticky Bit Special Permissions

- The "sticky bit" is a directory-level special permission that restricts file deletion, meaning only the file owner can remove a file within the directory.
 - The sticky bit can be very useful in shared environment because when it has been assigned to the permissions on a directory it sets it so only file owner can rename or delete the said file.
- You can only assign the sticky bit by explicitly defining permissions. The character for the sticky bit is **t**.
 - To set the sticky bit on a directory named dir1 you would issue the command \$ **chmod +t dir1**.

Troubleshoot Special Permissions Access



- Troubleshoot with `ls -l` and `chmod`.
- Confirm the SUID permission is set correctly for executable files.
- Confirm the SGID permission is set correctly for directories to permit files created in the directory to inherit the group association.
- Confirm the sticky bit permission is set correctly.



Review Activity: Special Linux Permissions

1. How would SGID benefit users when set on the /projects directory where multiple users are members of the associated group and need access to each other's files in the directory?
2. Why might a sysadmin set the sticky bit on a configuration file?

Understand Access Control List (ACL) Concepts

- Standard permissions are limited to one user, one group, and all others.
 - Cannot grant different access levels to two different users.
- ACLs permit multiple users to be given multiple levels of access.
- ACLs permit multiple groups to be given multiple levels of access.

Display ACL Entries

- Use the getfacl command to display ACL entries.

```
student@ubuntu20:/projects$ getfacl project1.txt
# file: project1.txt
# owner: student5
# group: 2022projects
user::rw-
user:student9:r--
group::r--
mask::r--
other::r--

student@ubuntu20:/projects$
```

Configure ACL Entries

- Use the **setfacl** command to configure ACL entries.
- To set an ACL entry for userA with rwx access:

```
setfacl -m u:userA:rwx fileA
```

- To set an ACL entry for groupA with rwx access:

```
setfacl -m g:groupA:rwx fileA
```

- To set an ACL entry userA with rwx access and group sales with rw access:

```
setfacl -m u:userA:rwx,g:sales:rw fileA
```

- To remove an ACL entry for userA for fileA:

```
setfacl -x u:userA fileA
```

```
student@ubuntu20:/projects$ sudo setfacl -m u:student9:r project2.txt
student@ubuntu20:/projects$ getfacl project2.txt
# file: project2.txt
# owner: root
# group: root
user::rw-
user:student9:r--
group::r--
mask::r--
other::r--
```

student@ubuntu20:/projects\$ █

Key Demonstration: Configure ACLs



Sign in to at least one system (either RH or Debian-based) and then configure ACLs on a file, demonstrating multiple users with multiple levels of access. Begin at the logged-in user's home directory.

1. If necessary, create an additional user named USERZ: `useradd USERZ`
2. Create a directory named acl-demo: `mkdir acl-demo`
3. Change to the acl-demo directory: `cd acl-demo`
4. Create fileA: `touch fileA`
5. Display default ACL settings: `getfacl fileA`
6. Configure USERZ with read-only access: `setfacl SYNTAX`

Troubleshoot ACLs

- Display ACL entries by using getfacl.
- Configure ACL entries by using setfacl.
- Confirm user identity.
- Confirm group membership.



Review Activity: ACL Configuration

1. Explain the benefit offered by ACLs compared to standard Linux permissions.
2. What commands are used to set ACL entries for USERA with rwx and USERB with r-- for fileA?
3. Does the ACL structure replace standard permissions?

Summary

- Understand the principle of least privilege, which enforces the idea that users should be given as little access to resources as necessary for them to do their jobs, with no additional unneeded access.
- Recognize access levels and identities.
- Absolute mode and symbolic mode provide the same information in different ways. Absolute mode displays in octal numerals, while symbolic mode displays information using operators.
- The immutable flag is an attribute of a file or directory that prevents it from being modified, even by the root user.



Thanks
Q & A

User Management: Administering Users and Groups

(Customize User Accounts & Resource Access)



A close-up photograph of a sailboat's winch and rope mechanism. The winch is made of polished metal and is wrapped with white rope featuring blue stripes. The background shows the bright blue ocean under a clear sky. The lighting creates strong highlights on the metallic surfaces of the winch.

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Outline

- Types of Users on Linux
- User Configuration Files
- Contents of /etc/passwd & /etc/shadow
- User Management Commands
- User Configuration Commands
- User Account Modification Utilities
- Q & A

Objectives

- Manage user accounts.
- Manage group accounts.
- Configure privilege escalation.
- Troubleshoot user and group issues.

Manage User Accounts

Types of Accounts on Linux

- There are three types of accounts on Linux systems:
 - root,
 - standard user, and
 - service.



USER CONFIGURATION FILES

- **USER ACCOUNT STORAGE**

- `/etc/passwd` file stores the actual user account and maintains various settings related to accounts.
- `/etc/shadow` file stores password information for the accounts.
- `/etc/profile` to set system-wide environment variables and startup programs for new user shells.
- `/etc/bashrc` to establish system-wide functions and aliases for new user shells.

Contents of /etc/passwd & /etc/shadow

/etc/passwd

Field	Content
User Name	The name the user logs into the system with
Password	User password represented as an x; the actual password is stored elsewhere
User ID	Unique number representing the user to the system
Group ID	Unique number representing the user's primary group
Comment	Typically displays the user's full name
Home directory	Absolute path to the user's home directory
Login shell	Absolute path to the user's default shell (usually /bin/bash)

/etc/shadow

Field	Content
User name	The name the user logs into the system with
Password	Hash value of the user's password
Days since last password change	Number of days since the last password change; counted from January 1, 1970
Days before password may be changed	Minimum changeable period, typically set at 1 day
Days before password must be changed	Maximum number of days since the last password change before the password must be changed again; a value of 99999 means the password never needs to be changed, but often set at 90 days
Days until the user is warned to change password	Days before the date the password must be changed that the warning is issued, often set to 7 days
Days after password expires that the account is disabled	Number of days after the password expires until the account is disabled; should be immediate
Days until account expires	Number of days until the account expires and cannot be used
Unused field	Reserved for potential future use

User Configuration Files



/etc/password file

```
student@ubuntu20:~$ tail /etc/passwd | grep student
student:x:1000:1000:student,,,,:/home/student:/bin/bash
student@ubuntu20:~$ █
```

User Configuration Files

/etc/shadow file

```
student@ubuntu20:~$ sudo tail /etc/shadow | grep student
student:$6$XYM8.t73X57Xq/NH$IN5RCTXNyaf4RE4yn5.4Tf464W0AR
IQWRGt/UW.U92/qAK2TqjVj5V9WdmUSQSWoMqffFXljRGyflfUxDxeeCf0
:18942:0:99999:7:::
student@ubuntu20:~$ █
```

Account Management Commands

- useradd – create user accounts in the /etc/passwd and /etc/shadow files
- usermod – modify existing user accounts
- userdel – delete existing user accounts

```
student@ubuntu20:~$ sudo adduser student12
Adding user `student12' ...
Adding new group `student12' (1004) ...
Adding new user `student12' (1004) with group `student12' ...
Creating home directory `/home/student12' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for student12
Enter the new value, or press ENTER for the default
  Full Name []: Student Twelve
  Room Number []:
  Work Phone []:
  Home Phone []:
  Other []:
Is the information correct? [Y/n] Y
student@ubuntu20:~$ █
```

Options	Purpose
-c	Set the comment value, usually the user's full name
-e	Set an expiration date for the user account, format YYYY-MM-DD
-m	Create a user home directory in /home
-s	Set a default shell for the user
-u	Set a specific user ID value
-D	Display the default settings

The useradd Command

- Options:
 - -c comment (often used for full name)
 - -e expire
 - -D display default settings

```
student@ubuntu20:~$ sudo useradd student10
student@ubuntu20:~$ sudo usermod -c "Student Ten" student10
student@ubuntu20:~$ tail /etc/passwd | grep student10
student10:x:1002:1002:Student Ten:/home/student10:/bin/sh
student@ubuntu20:~$ sudo userdel student10
student@ubuntu20:~$ █
```

The *passwd* Command

\$ passwd username

Options	Purpose
-d	Delete a password and disable the account
-e	Immediately expire a password, forcing a password change by the user
-l	Lock the account (for example, during a leave of absence)
-u	Unlock a locked account

Key Demonstration: Create User and Set Password



Sign in to at least one system (either RH or Debian-based) and then walk through the process of creating a user and setting a password. Create a few more users with different options.

1. Display the contents of `/etc/login.defs`. [Configuration control definition for login package]
2. Create a user with `useradd`.
3. Create a user with `useradd` and define a non-default home directory.
4. Create a user with `useradd` and define a non-Bash shell.

(continued on next slide)

Key Demonstration: Create User and Set Password

(continued from previous slide)

- 
- 5. Set a password for each new user by using the `passwd` command.
 - 6. Create a user with `adduser`, pointing out the options available during the process and showing how a password is set.
 - 7. Display the contents of the `/etc/passwd` file to show the new users.
 - 8. Display the contents of the `/etc/shadow` file to show the hashed passwords.

Modify and Delete User Accounts

- usermod
- userdel

```
student@ubuntu20:~$ sudo useradd student10
student@ubuntu20:~$ sudo usermod -c "Student Ten" student10
student@ubuntu20:~$ tail /etc/passwd | grep student10
student10:x:1002:1002:Student Ten:/home/student10:/bin/sh
student@ubuntu20:~$ sudo userdel student10      1
student@ubuntu20:~$ █
```

USER ACCOUNT MODIFICATION UTILITIES

- usermod provides options for changing most of the fields in the /etc/passwd file.

Command	Description
usermod	Edits user account fields, as well as specifying primary and secondary group membership
passwd	Changes the password for an existing user
chpasswd	Reads a file of login name and password pairs, and updates the passwords
chage	Changes the password's expiration date
chfn	Changes the user account's comment information
chsh	Changes the user account's default shell

The chage Command

\$chage -1

Option	Purpose
-l	Display the current values
-M	Specify the maximum number of days between password changes
-m	Specify the minimum number of days between password changes
-W	Specify the number of warning days before a password expires
-E	Lock an account after a specified date

Key Demonstration: Account Configuration Commands



Sign in to at least one system (either RH or Debian-based), then display output of any of the following:

1. whoami
2. w
3. who
4. id
5. ./etc/login.defs file
6. Password configurations with chage

\$w & \$who display all current logins on the system, including those that might have remote terminal connections.

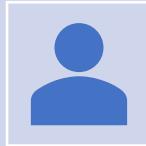


Review Activity: User Account Management

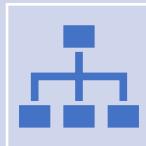
1. Why are user passwords stored in the /etc/shadow file and not the /etc/passwd file?
2. What is the purpose of the /etc/skel directory?
3. Why might an administrator change a user's default shell?

Group Management

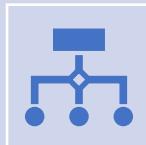
GROUP CONFIGURATION FILES



Easier to grant permissions to a resource to a single group with five members than it is to individually grant access to each user account.



Groups are a standard administrative tool for controlling access to resources.



/etc/group stores the group configuration files

Group Configuration Files

/etc/group

```
student@ubuntu20:~$ tail /etc/group
pulse-access:x:129:
gdm:x:130:
lxd:x:131:student
student:x:1000:
sambashare:x:132:student
systemd-coredump:x:999:
student999:x:1001:
student10:x:1002:
abc:x:1003:
student12:x:1004:
student@ubuntu20:~$ █
```

Group Management Commands

Group Management Command	Purpose
groupadd	create a group in the /etc/group files
groupmod	Modify an existing group
groupdel	Remove an existing group

Key Demonstration: Group Management



Sign in to at least one system (either RH or Debian-based), then create several groups and display the /etc/group file contents. The focus of this demo is group management. Adding users to the group is in a later demonstration.

1. Sign in
2. Create a new group named sales - `groupadd sales`
3. Create a new group named marketing - `groupadd marketing`
4. Display the contents of /etc/group to show the two new groups - `tail /etc/group`

(continued on next slide)

Key Demonstration: Group Management

(continued from previous slide)

- 
5. Modify the marketing group by changing its name to publicity -
`groupmod -n publicity marketing`
 6. Display the contents of `/etc/group` to show the renamed group -
`tail /etc/group`
 7. Delete the sales group -
`groupdel sales`
 8. Display the contents of `/etc/group` to show the sales group no longer exists

Add Users to a Group

- Use the **usermod** command covered earlier to add a user to an existing group.
 - **usermod –aG sales USERNAME**

Option	Purpose
-a	Append the user to the group, and maintain any existing group memberships
-G	Specify a group to which the user will be added

```
student@ubuntu20:~$ sudo useradd student9
student@ubuntu20:~$ sudo groupadd sales
student@ubuntu20:~$ sudo usermod -aG sales student9
student@ubuntu20:~$ sudo tail /etc/group | grep sales
sales:x:1006:student9
student@ubuntu20:~$
```

\$ groupmod -n publicity marketing

Key Demonstration: Add Members to Groups



Sign in to at least one system (either RH or Debian-based), then create a group and add members to it. The focus of this demo is adding users to groups; creating a group was covered in a previous demonstration.

1. Sign in
2. Create a group named Labs - `groupadd Labs`
3. Display the contents of `/etc/group` to show there are no members listed for the Labs group

(continued on next slide)

Key Demonstration: Add Members to Groups



(continued from previous slide)

4. Add USER to the Labs group - usermod -aG Labs USER
5. Display the contents of /etc/group to show that USER is a member of Labs
6. Display information about the USER account to show group membership - id USER



Review Activity: Group Account Management



1. Suggest at least two ways to display group membership information.
2. What command adds a user to a group?
3. What is the result if an administrator forgets to add the -a option when adding a user to a group?
4. Why might a user be a member of multiple groups?

Configure Privilege Escalation

Root Users

- Do not log on as the root user
- Many distributions disable the root account
- Use su or sudo to elevate privileges, or “get root”
- Delegate tasks by configuring the /etc/sudoers file

Elevate Privileges with su Command

- `su root` – switches to the root user in the original user's context.
- `su - root` – switches to the root user in the root user's context.
- You must know the password for the account you're switching to (unless you are root).

Elevate Privileges with sudo Command

To create a user account usingn sudo:

- sudo useradd {user- name}

```
student@ubuntu20:~$ sudo tail /etc/sudoers

# Members of the admin group may gain root privileges
%admin  ALL=(ALL)  ALL

# Allow members of group sudo to execute any command
%sudo   ALL=(ALL:ALL)  ALL
33

# See sudoers(5) for more information on "#include" directives:

#include dir /etc/sudoers.d
student@ubuntu20:~$ █
```

Configuration Examples for /etc/sudoers

Example1

To grant full administrative privileges to a user, type username

ALL=(ALL:ALL) ALL

- The user will be prompted for their password. Be very careful with this level of delegation!

Example 2

To delegate the ability to execute these shutdown commands without entering a password, type SOMEUSER ALL=(ALL) NOPASSWD:
SHUTDOWN_CMDS

- Assumes that SHUTDOWN_CMDS is aliased to all related options for the shutdown command

Key Demonstration: Elevate Privileges with sudo



Instructor - sign in to at least one system (either RH or Debian-based), then demonstrate the process of adding a user to the `sudoers` file and delegating the ability to issue the shutdown command to the system.

1. Log in
2. Get root privileges `su - root`
3. Select a user to delegate authority to, or create a new user with `useradd`

(continued on next slide)

Key Demonstration: Elevate Privileges with sudo

(continued from previous slide)

- 
4. Open the /etc/sudoers file for editing with vi sudo
 5. At the bottom of the file, add the following line:

```
SOMEUSER ALL=(ALL) NOPASSWD: SHUTDOWN_CMDS
```
 6. Save changes and exit
 7. *(Optional)* Switch to the delegated user and issue the shutdown -h now command

Alternative delegation method to sudo

- More granular control via defined rules and actions

Examples of delegated tasks:

- Software management
- System shutdown or hibernation
- Configuration of network devices
- Device access
- Mounting and unmounting filesystems on removable media

Polkit Commands

- pkexec - allows an authorized user to execute an action
- pkaction - display details about an action
- pkcheck - display whether a process is authorized
- pktyagent - provides a text-based authentication agent

```
student@ubuntu20:~$ sudo pkexec useradd student5
student@ubuntu20:~$ tail /etc/passwd | grep student5
student5:x:1007:1008::/home/student5:/bin/sh
student@ubuntu20:~$
```

Troubleshoot Privilege Escalation Issues



- User has switched user identities, but variables and other profile settings are not present.
- User fails to switch identities when using the `su` command.
- Sudo does not function as expected.
- Cannot exercise administrative privileges.
- User cannot run a command, even when the command is preceded by `sudo`.



Review Activity: Privilege Escalation

1. A developer at your organization needs the ability to reboot a test server, but their account's standard privileges do not permit this. The developer requests the system's root user password in order to use su to reboot the server. Is there a more secure option that aligns with the principle of least privilege?
2. How are the su root and su - root commands different?
3. You must delegate the shutdown -h privilege to SOMEUSER. What tool is used to modify the /etc/sudoers file, and what line must be added to that file?
4. Whose password must be entered with sudo? Whose password must be entered with su?

Troubleshoot User and Group Issues

Troubleshooting User Management Issues

- Only authorized users can manage groups
 - root
 - Users delegated the privileges with sudo
- Does the group exist?
 - Check etc/passwd or etc/group files to confirm
 - Halt active user processes with sudo killall -u {username}

User Login Attempt Failures



1. Confirm the user has an account on the system by displaying the contents of `/etc/passwd`. If necessary, create an account for the user by using the `useradd` command.
2. If the account exists, confirm that a password is set. Display the contents of `/etc/shadow` and verify a hashed password exists. Use the `passwd` command to set a password if one did not exist.
3. If the account exists and a password is set, the user may have forgotten the correct password. Reset the password with the `passwd` command.
4. If the account exists and a password is set, the password may be expired. Reset the password by using the `passwd` command.
5. If the account exists and a password is set, the account may be locked. Unlock the account by using the `chage` command.

Reviewing the Login Process

1. The operating system boots and the kernel is loaded. Assume the system boots to the CLI. An authentication prompt is displayed.
2. The user enters a name and password combination. These are checked against the /etc/passwd and /etc/shadow files. Settings such as expired passwords and locked accounts are checked for at this point.
3. System and user profile files are processed, and the user is presented with an authenticated and customized environment.

Using User Login Commands

- lastlog – displays recent login information
- last – pulls login history information from /var/log/wtmp
- w – displays current logins to the system, including idle time
- who – displays current logins to the system

```
student@ubuntu20:/etc/polkit-1$ last
student :1          :1                          Thu Dec  9 08:34  still logged in
reboot  system boot  5.8.0-43-generic   Thu Dec  9 08:27  still running
student :1          :1                          Thu Nov 11 14:45 - 14:47  (00:01)
reboot  system boot  5.8.0-43-generic   Thu Nov 11 14:44 - 14:48  (00:03)

wtmp begins Thu Nov 11 14:44:38 2021
student@ubuntu20:/etc/polkit-1$
```

Key Demonstration: User Login Commands



Sign in to at least one system (either RH or Debian-based), then run the following commands and discuss the output. Note that the output can vary from system to system.

1. Run the `last` command.
2. Run the `lastlog` command.
3. Run the `w` command.
4. Run the `who` command and compare the results to the output from the `w` command.



Review Activity: User and Group Troubleshooting

1. List at least three scenarios where you might need records of who logged in to a Linux system.
2. Another administrator asks you to explain the value of editing the /etc/sudoer's file with visudo rather than a traditional text editor. What is your response?
3. List at least three reasons a user account might be locked.
4. During a security audit it is discovered that a user does not have a password set. When you check the /etc/passwd file, the password field is properly populated with the x character. What file would actually display whether a password has been set for the user?
5. A user places sudo before a command, but the command still fails to run. What might be the cause?
6. An administrator asks you how to delegate Linux administrative privileges to a specific user. What group is used for such delegation?



Thanks
Q & A

Linux Networking Commands

Dr. Vimal Baghel, Assistant Professor
SCSET, BU

Outline

- Understanding Network Devices
- Configuring NIC IP address
- Configuring Networking with Command-line Utilities
- Important Files
- Tools and Network Performance Analysis
- Commands for Connectivity, ARP, Routing, Switching, VLAN, NAT Firewall
- Q & A

Understanding Network Devices in Linux

- Linux networking devices
 - Not shown in /dev directory
 - Do not “exist” on system until appropriate device driver installed in kernel
- Networking device
 - Named channel over which network traffic can pass
- Device drivers for networking are kernel modules

Understanding Network Devices in Linux (continued)

- Kernel modules can be loaded or unloaded while Linux is running
- /dev/eth0
 - First Ethernet card installed on system
- Media Access Control (MAC) address
 - Unique address assigned by Ethernet card manufacturer

66.61.81.19

192.168.1.100

Understanding Network Devices in Linux (continued)

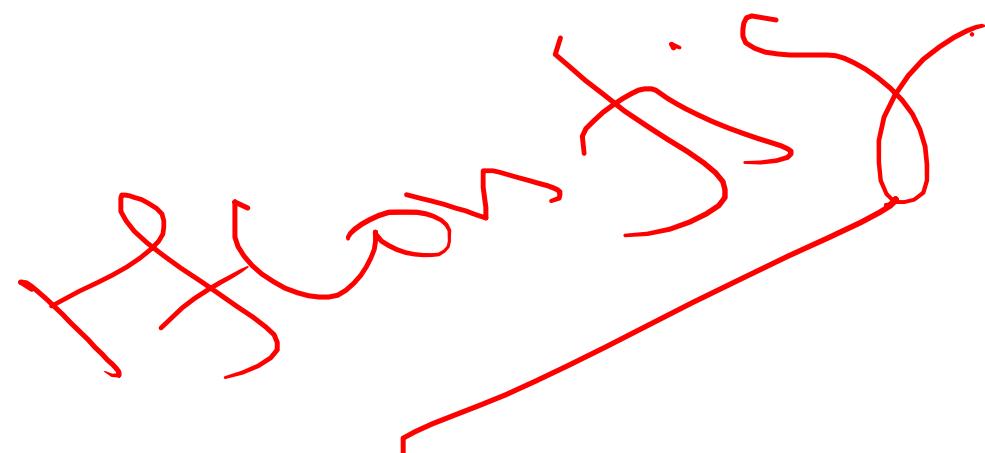
- To obtain MAC address
 - Host (switch) broadcasts message to entire network segment using Address Resolution Protocol (ARP)
 - Host with IP address responds directly to computer that sent ARP request with MAC address
 - Source host stores MAC address and IP address

Understanding Network Devices in Linux (continued)

- arp command
 - Display ARP cache
 - Mapping of IP addresses to hardware addresses
 - Used mainly for troubleshooting network connectivity
 - Refreshed frequently

Configuration NIC IP address

- NIC: Network Interface Card
- Use “ipconfig” command to determine IP address, interface devices, and change NIC configuration
- Any device use symbol to determine
 - eth0: Ethernet device number 0
 - eth1: ethernet device number 1
 - lo : local loopback device
 - Wlan0 : Wireless lan 0



Determining NIC IP Address

```
[root@tmp]# ifconfig -a
```

```
eth0 Link encap:Ethernet HWaddr 00:08:C7:10:74:A8
```

```
BROADCAST MULTICAST MTU:1500 Metric:1
```

```
RX packets:0 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
```

```
collisions:0 txqueuelen:100
```

```
RX bytes:0 (0.0 b) TX bytes:0 (0.0 b)
```

```
Interrupt:11 Base address:0x1820
```

```
lo Link encap:Local Loopback
```

```
inet addr:127.0.0.1 Mask:255.0.0.0
```

```
UP LOOPBACK RUNNING MTU:16436 Metric:1
```

```
RX packets:787 errors:0 dropped:0 overruns:0 frame:0
```

```
TX packets:787 errors:0 dropped:0 overruns:0 carrier:0
```

```
collisions:0 txqueuelen:0
```

```
RX bytes:82644 (80.7 Kb) TX bytes:82644 (80.7 Kb)
```

Changing IP Address

- We could give this eth0 interface an IP address using the **ifconfig** command.

```
[root@tmp]# ifconfig eth0 10.0.0.1 netmask 255.255.255.0 up
```

- The "up" at the end of the command activates the interface.
- To make this permanent at each boot up time, **add this command in /etc/rc.local file which is run at the end of every reboot.**

Permanent IP configuration

- Fedora Linux also makes life a little easier with interface configuration files located in the /etc/sysconfig/network-scripts directory.
- **Interface eth0 has a file called ifcfg-eth0, eth1 uses ifcfg-eth1, and so on.**
- Admin can place your IP address information in these files

File formats for network-scripts

```
root@network-scripts]# less ifcfg-eth0
```

```
DEVICE=eth0
IPADDR=192.168.1.100
NETMASK=255.255.255.0
```

```
BOOTPROTO=static
ONBOOT=yes
#
# The following settings are optional
```

```
#  
BROADCAST=192.168.1.255  
NETWORK=192.168.1.0  
[root@network-scripts]#
```

Getting the IP Address Using DHCP



```
[root@tmp]# cd /etc/sysconfig/network-scripts
```

```
[root@network-scripts]# less ifcfg-eth0
```

```
DEVICE=eth0  
BOOTPROTO=dhcp  
ONBOOT=yes
```

```
[root@network-scripts]#
```

Activate config change

- After change, the values in the configuration files for the NIC you must deactivate and activate it for the modifications to take effect.
- The ifdown and ifup commands can be used to do this:

```
[root@network-scripts]# ifdown eth0
[root@network-scripts]# ifup eth0
```

Multiple IP Addresses on a Single NIC(1)

```
[root@tmp]# ifconfig -a
```

wlan0 Link encap:Ethernet HWaddr 00:06:25:09:6A:B5
inet addr:192.168.1.100 Bcast:192.168.1.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

RX packets:47379 errors:0 dropped:0 overruns:0 frame:0

TX packets:107900 errors:0 dropped:0 overruns:0 carrier:0

collisions:0 txqueuelen:100

RX bytes:4676853 (4.4 Mb) TX bytes:43209032 (41.2 Mb)

Interrupt:11 Memory:c887a000-c887b000

wlan0.0 Link encap:Ethernet HWaddr 00:06:25:09:6A:B5

inet addr:192.168.1.99 Bcast:192.168.1.255 Mask:255.255.255.0

UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1

Interrupt:11 Memory:c887a000-c887b000

Multiple IP Addresses on a Single NIC(2)

- In the previous slide, there were two wireless interfaces: wlan0 and wlan0:0.
- Interface wlan0:0 is a child interface of wlan0, a virtual subinterface (an IP alias.)
- IP aliasing is one of the most common ways of creating multiple IP addresses associated with a single NIC.
- Aliases have the name format parent-interface-name:X, where X is the sub-interface number of your choice.

The process for creating an IP alias

- First ensure the parent real interface exists
- Verify that no other IP aliases with the same name exists with the name you plan to use. In this we want to create interface wlan0:0.
- Create the virtual interface with the ifconfig command

```
[root@tmp]# ifconfig wlan0:0 192.168.1.99 netmask 255.255.255.0 up
```

- Shutting down the main interface also shuts down all its aliases too. Aliases can be shutdown independently of other interfaces

The process for creating an IP alias

- Admin should also create a `/etc/sysconfig/network-scripts/ifcfg-wlan0:0` file
- so that the aliases will all be managed automatically with the ifup and ifdown commands

```
DEVICE=wlan0:0
ONBOOT=yes
BOOTPROTO=static
IPADDR=192.168.1.99
NETMASK=255.255.255.0
```

- The commands to activate and deactivate the alias interface would therefore be:

```
[root@tmp]# ifup wlan0:0
[root@tmp]# ifdown wlan0:0
```

How to View Current Routing Table

- The `netstat -nr` command will provide the contents of the routing table.
- Networks with a gateway of 0.0.0.0 are usually directly connected to the interface.
- No gateway is needed to reach your own directly connected interface, so a gateway address of 0.0.0.0 seems appropriate.
- The route with a destination address of 0.0.0.0 is your default gateway

#natstat –nr command

```
[root@tmp]# netstat -nr
```

Kernel IP routing table

Destination	Gateway	Genmask	Flags	MSS	Window	irtt	Iface
172.16.68.64	172.16.69.193	255.255.255.224	UG	40	0	0	eth1
172.16.11.96	172.16.69.193	255.255.255.224	UG	40	0	0	eth1
172.16.68.32	172.16.69.193	255.255.255.224	UG	40	0	0	eth1
172.16.67.0	172.16.67.135	255.255.255.224	UG	40		0	0 eth0
172.16.69.192	0.0.0.0	255.255.255.192	U	40	0	0	eth1
172.16.67.128	0.0.0.0	255.255.255.128	U	40	0	0	eth0
172.160.0	172.16.67.135	255.255.0.0	UG	40	0	0	eth0
172.16.0.0	172.16.67.131	255.240.0.0	UG	40	0	0	eth0
127.0.0.0	0.0.0.0	255.0.0.0	U	40	0	0	lo
0.0.0.0	172.16.69.193	0.0.0.0	UG	40	0	0	eth1

```
[root@tmp]#
```

How to Change Default Gateway

```
[root@tmp]# route add default gw 192.168.1.1 wlan0
```

- In this case, make sure that the router/firewall with IP address 192.168.1.1 is connected to the same network as interface wlan0
- Once done, we'll need to update “/etc/sysconfig/network” file to reflect the change. This file is used to configure your default gateway each time Linux boots.

```
NETWORKING=yes
HOSTNAME=bigboy
GATEWAY=192.168.1.1
```

How to Delete a Route

```
[root@tmp]# route del -net 10.0.0.0 netmask 255.0.0.0 gw 192.168.1.254 wlan0
```

Linux router

- Router/firewall appliances that provide basic Internet connectivity for a small office or home network are becoming more affordable every day
- when budgets are tight you might want to consider modifying an existing Linux server to be a router

Configuring IP Forwarding

- For your Linux server to become a router, you have to enable packet forwarding.
- In simple terms packet forwarding enables packets to flow through the Linux server from one network to another.
- The Linux kernel configuration parameter to activate this is named net.ipv4.ip_forward and can be found in the file /etc/sysctl.conf.
- Remove the "#" from the line related to packet forwarding.

/etc/sysctl.conf changing

Before: # Disables packet forwarding

```
net.ipv4.ip_forward=0
```

After: # Enables packet forwarding

```
net.ipv4.ip_forward=1
```

- To activate the feature immediately you must force Linux to read the /etc/sysctl.conf file with the sysctl command using the -p switch

```
[root@tmp]# sysctl -p
```

Configuring /etc/hosts File

- The **/etc/hosts** file is just a list of IP addresses and their corresponding server names.
- Your server will typically check this file before referencing DNS. If the name is found with a corresponding IP address, then DNS won't be queried at all.
- Unfortunately, if the IP address for that host changes, you also must also update the file. This may not be much of a concern for a single server but can become laborious if it must be done companywide.
- Use a centralized DNS server to handle most of the rest.
- Sometimes we might not be the one managing the DNS server, and in such cases, it may be easier to add a quick **/etc/hosts** file entry till the centralized change can be made.

/etc/hosts

```
192.168.1.101 smallfry
```

- You can also add aliases to the end of the line which enable you to refer to the server using other names.
- Here we have set it up so that smallfry can also be accessed using the names tiny and littleguy.

```
192.168.1.101 smallfry tiny littleguy
```

/etc/hosts

- You should never have an IP address more than once in this file because Linux will use only the values in the first entry it finds.

```
192.168.1.101 smallfry # (Wrong)
192.168.1.101 tiny   # (Wrong)
192.168.1.101 littleguy # (Wrong)
```

Using ping to Test Network Connectivity

- The Linux ping command will send continuous pings, once a second, until stopped with a Ctrl-C.
- Here is an example of a successful ping to the server bigboy at 192.168.1.100

```
[root@smallfry tmp]# ping 192.168.1.101
PING 192.168.1.101 (192.168.1.101) from 192.168.1.100 : 56(84) bytes of data.
64 bytes from 192.168.1.101: icmp_seq=1 ttl=128 time=3.95 ms
64 bytes from 192.168.1.101: icmp_seq=2 ttl=128 time=7.07 ms
64 bytes from 192.168.1.101: icmp_seq=3 ttl=128 time=4.46 ms
64 bytes from 192.168.1.101: icmp_seq=4 ttl=128 time=4.31 ms

--- 192.168.1.101 ping statistics ---
4 packets transmitted, 4 received, 0% loss, time 3026ms
rtt min/avg/max/mdev = 3.950/4.948/7.072/1.242 ms
```

```
[root@smallfry tmp]#
```

Using ping to Test Network Connectivity

- Most servers will respond to a ping query it becomes a very handy tool.
- A lack of response could be due to:
 - A server with that IP address doesn't exist
 - The server has been configured not to respond to pings
 - A firewall or router along the network path is blocking ICMP traffic
 - You have incorrect routing. Check the routes and subnet masks on both the local and remote servers and all routers in between.
 - Either the source or destination device having an incorrect IP address or subnet mask.

Configuring Networking with Command-line Utilities

- ifconfig command
 - Set up network configuration in Linux kernel
 - Parameters include:
 - Network interface
 - IP address assigned to interface
 - Network mask
 - Syntax
 - *ifconfig device ip_address netmask address broadcast address*
 - \$ ifconfig eth0

Configuring Networking with Command-line Utilities (continued)

- Packet: Unit of data that network card transmits
- Broadcast address sends packet to all computers on same part of network
- Maximum transmission unit (MTU)
 - Maximum size of packet interface supports



Configuring Networking with Command-line Utilities (continued)

- View status of interface: **ifconfig eth0**
- Stop Ethernet interface: **ifconfig eth0 down**
- Start Ethernet interface: **ifconfig eth0 up**
- Routing table tells networking software where to send packets that are not part of local network
- A real example of configuring an Ethernet card at the command line might look like this:
- **# ifconfig eth0 192.168 . 100.1 netmask 255.255.255.0 broadcast 192. 168.100.255**

Configuring Networking with Command-line Utilities (continued)

- **route command**

- View or configure routing table within kernel
- Executed at boot time when networking initialized
- Output information for addresses
 - 192.168.100.0 (eth0 IP address)
 - 127.0.0.0
 - Other

```
# route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref Use Iface
192.168.100.0   *              255.255.255.0   U      0      0    0 eth0
127.0.0.0       *              255.0.0.0     U      0      0    0 lo
default         192.168.100.5  0.0.0.0     UG     0      0    0 eth0
```

Configuring Networking with Command-line Utilities (continued)

- Route command output
 - Destination – Ref
 - Gateway – Use
 - Genmask – Iface
 - Flags
- Add route example:
 - `route add -net 192.168.100.0 netmask 255.255.255.0 dev eth0`
 - This command adds a default gateway route,
 - `# route add default gw 192.168.100.5`

Configuring Networking with Command-line Utilities (continued)

- **service** command
 - Start or stop networking
 - Relies on script /etc/rc.d/init.d/network
- /etc/sysconfig/networking/devices configuration directory
 - Contains file for each network device
 - ifcfg-eth0 file
 - Used by /etc/rc.d/init.d/network script
 - As it executes ifconfig and route commands

Changing IP Address/Other Parameters

- Change the information in `/etc/sysconfig/network-scripts/ifcfg-eth0`
- Execute this command:
 - **# service network restart**

Configuring Networking with Command-line Utilities (continued)

- **ifup** and **ifdown** scripts manage single interface, rather than all network interfaces
 - Example:
 - # ./ifup eth0
 - # ./ifdown eth0
 - Some systems have two or more physical network devices

Configuring Networking with Command-line Utilities (continued)

- IP forwarding
 - Allows packets to be passed between network interfaces
 - Required for any router
 - To enable:
 - `# echo 1 > /proc/sys/net/ipv4/ip_forward`



Thanks
Q & A

Text Processing & Backup Utilities in Linux

Dr. Vimal Baghel, Assistant Professor
SCSET, BU

Outline

- Basic Text Editing Commands: **cat, tac, echo, vi/vim/nano/gedit**
- Text Processing Utilities: **sort, uniq, comm, cmp, diff, tr, cut, paste, grep, sed, awk**
- Pipes and redirection: **|, >, >>, <, <<**
- Backup Utilities: **rsync**
- Q & A

Basic Text Editing Commands

- Navigate, edit, and format your text efficiently
 - \$ cat filename: display the contents of a file
 - \$ tac filename: display the contents of a file in reverse lines
 - \$ vi/vim filename ✓
 - \$ nano filename
 - \$ gedit filename
- Display a value of a variable
 - \$ var="Today is Friday!"
 - \$ echo \$var

Sort: sort lines of text files

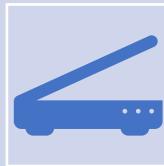
- sort alphabetically, numerically, or based on custom criteria having numbers, strings, and alphanumeric combinations.
 - \$ sort dir_list.txt
 - \$ sort -n numbers.txt
- Options
 - -R random sort
 - -r reverse the sort order
 - -o redirect sorted result to specified filename
 - -n sort numerically
 - -V version sort, aware of numbers within text
 - -h sort human readable numbers like 4K, 3M, etc
 - -k sort via key
 - -u sort uniquely
 - -b ignore leading white-spaces of a line while sorting
 - -t use SEP instead of non-blank to blank transition

~~Sort~~

uniq: report or omit repeated lines

- This command is more specific to recognizing duplicates.
- Usually requires a sorted input as the comparison is made on adjacent lines only
- **Options**
 - `-d` print only duplicate lines
 - `-c` prefix count to occurrences
 - `-u` print only unique lines
- `$ sort test_list.txt`
- `$ uniq test_list.txt`
- `$ uniq -d sorted_list.txt` print only duplicate lines
- `$ uniq -cd sorted_list.txt` print only duplicate lines and prefix the line with number of times it is repeated
- `$ uniq -u sorted_list.txt` print only unique lines, repeated lines are ignored

comm: compare two sorted files line by line

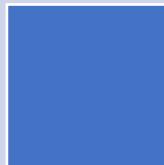


Without any options, it prints output in three columns - *lines unique to file1, line unique to file2 and lines common to both files*



Options

- 1 suppress lines unique to file1
- 2 suppress lines unique to file2
- 3 suppress lines common to both files



comm -12 sorted_file1.txt sorted_file2.txt print lines common to both files

cmp: compare two files byte by byte



Useful to compare binary files.



If the two files are same, no output is displayed (exit status 0)



If there is a difference, *it prints the first difference - line number and byte location (exit status 1)*



Option -s allows to suppress the output, useful in scripts



\$ cmp /bin/grep /bin/fgrep



/bin/grep /bin/fgrep differ: byte 25, line 1

diff: compare files line by line

```
$ diff -s test1.txt test2.txt
```

Useful to compare old and new versions of text files

All the differences are printed, which might not be desirable if files are too long

Options

- -s convey message when two files are same
- -y two column output
- -i ignore case while comparing
- -w ignore white-spaces
- -r recursively compare files between the two directories specified
- -q report if files differ, not the details of difference

tr: translate or delete characters

Options

- -d delete the specified characters
- -c complement set of characters to be replaced

tr 'a-z' 'A-Z' < file.txt

- tr a-z A-Z < test_list.txt
 - convert lowercase to uppercase
- tr -d ._ < test_list.txt
 - delete the dot and underscore characters
- tr a-z n-za-m < test_list.txt >
 - encrypted_test_list.txt Encrypt by replacing every lowercase alphabet with 13th alphabet after it
 - Same command on encrypted text will decrypt it

cut:
remove
sections
from each
line of
files

For columns operations with well defined delimiters, cut command is handy

Examples

- `ls -l | cut -d' ' -f1` first column of ls -l
 - -d option specifies delimiter character; in this case it is single space character (Default delimiter is TAB character)
 - -f option specifies which fields to print separated by commas, in this case field 1
- `cut -d':' -f1 /etc/passwd`
 - prints first column of /etc/passwd file
- `cut -d':' -f1,7 /etc/passwd`
 - prints 1st and 7th column of /etc/passwd file with character in between
- `cut -d':' --output-delimiter=' ' -f1,7 /etc/passwd`
 - use space as delimiter between 1st and 7th column while printing

paste:
merge
lines of
files

Examples

- ~~paste list1.txt list2.txt list3.txt > combined_list.txt~~
 - combines the three files column-wise into single file, the entries separated by TAB character
- `paste -d':' list1.txt list2.txt list3.txt > combined_list.txt`
 - the entries are separated by : character instead of TAB

Summary of Text Processing Utilities

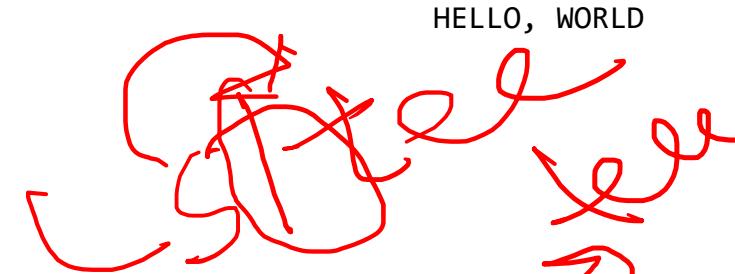
Command	Description	Examples
cat	Concatenate and display files	cat file1.txt file2.txt
sort	Sort lines of text files	sort file.txt
uniq	Remove duplicate lines from a sorted file	sort file.txt uniq
grep	Search for patterns in files	grep "pattern" file.txt
cut	Extract columns of text from files	cut -f1,3 file.txt
sed	Stream editor for filtering and transforming text	sed 's/old/new/' file.txt
awk	Pattern scanning and processing language	awk '{print \$1, \$3}' file.txt
tr	Translate or delete characters	tr 'a-z' 'A-Z' < file.txt
wc	Count lines, words, and characters in a file	wc file.txt
diff	Compare two files and show differences	diff file1.txt file2.txt
patch	Apply a diff file to a file or directory	patch file.txt patch.diff
nl	Number lines in a file	nl file.txt
head	Display the first few lines of a file	head file.txt
tail	Display the last few lines of a file	tail file.txt
tee	Redirect output to a file and to the terminal	ls tee output.txt
fmt	Format text files for printing	fmt file.txt
pr	Convert text files for printing	pr file.txt
iconv	Convert character encoding of a file	iconv -f utf-8 -t iso-8859-1 file.txt
dos2unix	Convert DOS line endings to UNIX line endings	dos2unix file.txt
rev	Reverse lines of a file	rev file.txt
fold	Wrap lines of text to a specified width	fold -w 80 file.txt
join	Join lines from two files based on a common field	join file1.txt file2.txt

Pipes & Redirection

Piping in Linux:

- Linking one command's output to another's input is known as “piping”.
- This enables you to execute complicated activities by chaining together commands.
- The vertical bar (|) is the pipe operator.
- \$ ls | wc -l

```
$ echo "Hello, World" | tr \[a-z\] [A-Z]
```



Redirection in Linux:

- Redirection is a way of changing a command's default input or output.
- This allows you to save the output of a command to a file or to read input from a file instead of the keyboard.
- The following operators are used for redirection purposes.

The output of a command can be redirected to a file using the > operator:

```
$ curl -L https://github.com/kubernetes/kubernetes/blob/master/README.md > README.md
```

The >> operator is used to write to a new file or append a command's output to the end of an already-existing file.

```
$ cat numbers.txt
one
two
```

```
$ echo "three" >> numbers.txt
```

```
$ cat numbers.txt
one
two
three
```

The < operator reads input from a file and then acts upon it.

```
$ wc -l < numbers.txt > lines.txt
```

The operator called >> redirects the errors to your desired file.

```
$ docker ps > error.txt
```

Why to backup?

- Important to prevent permanent data loss on PCs/servers.
- To know different backup tools is very important especially for **System Administrators**.
- Backup can either be done manually or configured to work automatically.
- Many backup utilities have different features that allow users to configure the
 - type of backup,
 - time of backup,
 - what to backup,
 - logging backup activities and many more

rsync

a command-line backup tool popular among Linux users especially System Administrators.

It performs incremental backups, update whole directory tree and file system, both local and remote backups, preserves file permissions, ownership, links and many more.

It also has a GUI called **Grsync** but one advantage with the rsync is that backups can be automated using scripts and cron jobs

rsync(1)

rsync(1)

NAME

rsync - a fast, versatile, remote (and local) file-copying tool

SYNOPSIS

Local: rsync [OPTION...] SRC... [DEST]

Access via remote shell:

Pull: rsync [OPTION...] [USER@]HOST:SRC... [DEST]

Push: rsync [OPTION...] SRC... [USER@]HOST:DEST

Access via rsync daemon:

Pull: rsync [OPTION...] [USER@]HOST::SRC... [DEST]

rsync [OPTION...] rsync://[USER@]HOST[:PORT]/SRC... [DEST]

Push: rsync [OPTION...] SRC... [USER@]HOST::DEST

rsync [OPTION...] SRC... rsync://[USER@]HOST[:PORT]/DEST

Usages with just one SRC arg and no DEST arg will list the source files instead of copying.

DESCRIPTION

Rsync is a fast and extraordinarily versatile file copying tool. It
Manual page rsync(1) line 1 (press h for help or q to quit)

```
# rsync options source destination
```



-v : verbose

 **-r** : copies data recursively (don't preserve timestamps and permission while transferring data.

 **-a** : archive mode allows copying files recursively and preserves symbolic links, file permissions, user & group ownerships, and timestamps.

 **-z** : compress file data.

 **-h** : human-readable, output numbers in a human-readable format.

```
$ sudo apt-get install rsync [On Debian/Ubuntu & Mint]  
$ pacman -S rsync [On Arch Linux] $ emerge sys-apps/rsync [On Gentoo]  
$ sudo yum install rsync [On Fedora/CentOS/RHEL and Rocky Linux/AlmaLinux]  
$ sudo zypper install rsync [On openSUSE]
```

Copy/Sync Files and Directory Locally

Copy/Sync a File on a Local Computer

```
[root@tecmint:/home/tecmint]# rsync -zvh backup.tar.gz /tmp/backups/
created directory /tmp/backups
backup.tar.gz
sent 224.54K bytes received 70 bytes 449.21K bytes/sec
total size is 224.40K speedup is 1.00
[root@tecmint:/home/tecmint]# _
```

Copy/Sync a Directory on Local Computer

```
[root@tecmint:~]# rsync -avzh /root/rpmpkgs /tmp/backups/
sending incremental file list
rpmpkgs/
rpmpkgs/httpd-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/mod_ssl-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/nagios-4.4.6-4.el8.x86_64.rpm
rpmpkgs/nagios-plugins-2.3.3-5.el8.x86_64.rpm

sent 3.47M bytes received 96 bytes 2.32M bytes/sec
total size is 3.74M speedup is 1.08
[root@tecmint:~]# _
```

Copy/Sync Files and Directory to or From a Server

Copy a Directory from Local Server to a Remote Server

```
[root@tecmint:~]# rsync -avzh /root/rpmpkgs root@192.168.0.141:/root/
The authenticity of host '192.168.0.141 (192.168.0.141)' can't be established.
ED25519 key fingerprint is SHA256:bH2tiWQn4S5o6qmZhmtXcBR0V5TU5H4t2C42QDEMx1c.
This key is not known by any other names
Are you sure you want to continue connecting (yes/no/[fingerprint])? yes
Warning: Permanently added '192.168.0.141' (ED25519) to the list of known hosts.
root@192.168.0.141's password:
sending incremental file list
rpmpkgs/
rpmpkgs/httpd-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/mod_ssl-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/nagios-4.4.6-4.el8.x86_64.rpm
rpmpkgs/nagios-plugins-2.3.3-5.el8.x86_64.rpm

sent 3.74M bytes received 96 bytes 439.88K bytes/sec
total size is 3.74M speedup is 1.00
[root@tecmint:~]# _
```

Copy/Sync Files and Directory to or From a Server

Copy/Sync a Remote Directory to a Local Machine

```
[root@tecmint:~]# rsync -avzh root@192.168.0.141:/root/rpmpkgs /tmp/myrpms
root@192.168.0.141's password:
receiving incremental file list
created directory /tmp/myrpms
rpmpkgs/
rpmpkgs/httpd-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/mod_ssl-2.4.37-40.module_el8.5.0+852+0aaafc63b.x86_64.rpm
rpmpkgs/nagios-4.4.6-4.el8.x86_64.rpm
rpmpkgs/nagios-plugins-2.3.3-5.el8.x86_64.rpm

sent 104 bytes  received 3.49M bytes  997.68K bytes/sec
total size is 3.74M  speedup is 1.07
[root@tecmint:~]# _
```



Thanks
Q & A

Introduction to Bash Scripting

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Shell Scripts (1)

- Basically, a shell script is a text file with Unix commands in it.
- Shell scripts usually begin with a `#!` and a shell name
 - For example: `#!/bin/sh`
 - If they do not, the user's current shell will be used
- Any Unix command can go in a shell script
 - Commands are executed in order or in the flow determined by control statements.
- Different shells have different control structures
 - The `#!` line is very important
 - We will write shell scripts with the Bourne again shell (`bash`)

Bash scripting features

- 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)

Steps in Writing a Shell Script

- Write a script file using vi:
 - The first line identifies the file as a **bash** script.
`#!/bin/bash` _____
 - Comments begin with a # and end at the end of the line.
- give the user (and others, if (s)he wishes) permission to execute it.
 - chmod +x filename
- Run from local dir
 - ./filename
- Run with a trace – echo commands after expansion
 - bash -x ./filename

Chmod 2 & 5 > Shw
Chmod 601 fd

Shell Scripts (2)

- Why write shell scripts?

- To avoid repetition:
 - If you do a sequence of steps with standard Unix commands over and over, why not do it all with just one command?

- To automate difficult tasks:
 - Many commands have subtle and difficult options that you don't want to figure out or remember every time.

\$ ps -p \$\$

The output is as follows:

PID	TTY	TIME	CMD
12578	pts/4	00:00:00	bash

Writing your First Bash Shell Script

- Find out where is your Bash interpreter located
 - \$ which bash
 - /usr/bin/bash
- Open our favorite text editor and create a file hello.sh
 - \$ gedit hello.sh
- Write the script as:
 - #!/usr/bin/bash
 - # declare STRING variable
 - STRING="Hello World"
 - # print variable on a screen
 - echo \$STRING

\$ chmod u+x hello.sh

Execution of your first bash script

- Make the file hello.sh executable as:

- \$ chmod +x hello.sh

- Execute your first bash script as:

- \$./hello.sh **OR** \$ bash hello.sh

- Output: Hello World

- You are done!

A Simple Example (1)

- `$ tr abcdefghijklmnopqrstuvwxyz \thequickbrownfxjmpsvalzydg < file1 > file2`
 - “encrypts” file1 into file2
- Record this command into shell script files as:
 - myencrypt

```
#!/bin/sh
tr abcdefghijklmnopqrstuvwxyz \thequickbrownfxjmpsvalzydg
```
 - mydecrypt

```
#!/bin/sh
tr thequickbrownfxjmpsvalzydg \abcdefghijklmnopqrstuvwxyz
```

A Simple Example (2)

- **chmod** the files to be executable; otherwise, you couldn't run the scripts

```
$ chmod u+x myencrypt mydecrypt
```

- Run them as normal commands:

```
$ ./myencrypt < file1 > file2  
$ ./mydecrypt < file2 > file3  
$ diff file1 file3
```

Remember: This is needed when “.” is not in the path

Bourne Shell Variables



- Remember: Bash shell variables are different from variables in csh and tcsh!

- Examples in sh:

```
PATH=$PATH:$HOME/bin
```

```
HA=$1
```

```
PHRASE="House on the hill"
```

```
export PHRASE
```

Note: no space around =

Make PHRASE an environment variable

Assigning Command Output to a Variable

- Using backquotes, we can assign the output of a command to a variable:

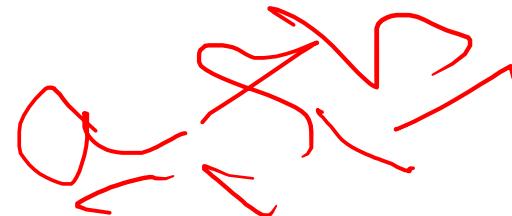
```
#!/usr/bin/bash
files=`ls`  

echo $files
```

- Very useful in numerical computation:

```
#!/usr/bin/sh
value=`expr 12345 + 54321`  

echo $value
```



Using expr for Calculations

- Variables as arguments:

```
$ count=5
```

```
$ count=`expr $count + 1`
```

```
$ echo $count
```

```
6
```

- Variables are replaced with their values by the shell!
- expr supports the following operators:
 - arithmetic operators: +,-,*,/,%
 - comparison operators: <, <=, ==, !=, >=, >
 - boolean/logical operators: &, |
 - parentheses: (,)
 - precedence is the same as C, Java

Variables

- Create a variable
 - Variablename=value (no spaces, no \$)
 - read variablename (no \$)
- Access a variable's value
 - \$variablename
- Set a variable
 - Variablename=value (no spaces, no \$ before variablename)

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 many 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

Reserved for OS, shell programming, etc. such as positional parameters \$0, \$1 ...

Variable Scope & Processes

- Variables are shared only with their own process, unless exported
 - x=Hi – define x in current process
 - sh – launch a new process
 - echo \$x – cannot see x from parent process
 - x=bye
 - <ctrl d> -- exit new process
 - echo \$x -- see x in old process did not change
 - demoShare – cannot see x
 - . demoShare – run with dot space runs in current shell
 - export x – exports the variable to make available to its children
 - demoShare – now it can see x

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.

Positional Parameter	What It References
\$0	References the name of the script
\$#	Holds the value of the number of positional parameters
\$*	Lists all the positional parameters
\$@	Same as \$*, except when enclosed in double quotes
"\$*"	Expands to a single argument (e.g., "\$1 \$2 \$3")
"\$@"	Expands to separate arguments (e.g., "\$1" "\$2""\$3")
\$1 .. \${10}	References individual positional parameters
set	Command to reset the script arguments

Positional Parameters

- **\$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.

Environment Variables

- `set | more` – shows all the environment variables that exist
- Change
 - `PS1='\u'`
 - `PATH=$PATH:/home/pe16132/bin1`
 - `IFS=':'`
 - `IFS` is **Internal Field Separator**

\$* and \$@

- \$* and \$@ can be used as part of the list in a for loop or can be used as part of it.
- When expanded \$@ and \$* are the same unless enclosed in double quotes.
 - \$* is evaluated to a single string while \$@ is evaluated to a list of separate words.

Shell Logic Structures & Control

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

Conditionals

- Conditionals are used to “test” something.
 - In Java or C, they test whether a Boolean variable is true or false.
 - In a bash script, the only thing you can test is whether a command is “successful”
- Every well-behaved command returns a [return code](#).
 - 0 if it was successful
 - Non-zero if it was unsuccessful (actually 1..255)
 - This is different from true/false conditions in C.

The if Statement

- Simple form:

```
if decision_command_1
then
    command_set_1
fi
```

- ✓ grep returns 0 if it finds something
- ✓ returns non-zero otherwise

- Example:

```
if grep unix myfile >/dev/null
then
    echo "It's there"
fi
```

- ✓ redirect to /dev/null so that "intermediate" results do not get printed

if and else

```
if grep "LINUX" myfile >/dev/null
then
    echo LINUX occurs in myfile
else
    echo No!
    echo LINUX does not occur in myfile
fi
```

if and elif

```
if grep " LINUX " myfile >/dev/null
then
    echo " LINUX occurs in file"
elif grep "DOS" myfile >/dev/null
then
    echo " LINUX does not occur, but DOS does"
else
    echo "Nobody is there"
fi
```

Use of Semicolons

- Instead of being on separate lines, statements can be separated by a semicolon (;)
 - For example:

```
if grep " LINUX " myfile; then echo "Got it"; fi
```

- This actually works anywhere in the shell.

```
$ cwd=`pwd`; cd $HOME; ls; cd $cwd
```

Use of Colon

- Sometimes it is useful to have a command which does “nothing”.
- The : (colon) command in **LINUX** does nothing

```
#!/bin/sh
if grep LINUX myfile
then
:
else
    echo "Sorry, LINUX was not found"
fi
```

Test command

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

The test Command – File Enquiry Options

-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.

- **test -f file** does **file** exist and is not a directory?
- **test -d file** does **file** exist and is a directory?
- **test -x file** does **file** exist and is executable?
- **test -s file** does **file** exist and is longer than 0 bytes?

```
#!/bin/sh
count=0
for i in *; do
    if test -x $i; then
        count=`expr $count + 1`
    fi
done
echo Total of $count files executable.
```

The test Command – String Tests

- `test -z string` is `string` of length 0?
- `test string1 = string2` does `string1` equal `string2`?
- `test string1 != string2` not equal?

Example:

```
if test -z $REMOTEHOST
then
:
else
    DISPLAY="$REMOTEHOST:0"
    export DISPLAY
fi
```

The test Command – Integer Tests

- Integers can also be compared:
 - Use -eq, -ne, -lt, -le, -gt, -ge
- For example:

```
#!/bin/sh
smallest=10000
for i in 5 8 19 8 7 3; do
    if test $i -lt $smallest; then
        smallest=$i
    fi
done
echo $smallest
```

Use of []

- The **test** program has an alias as []
 - Each bracket must be surrounded by spaces!
 - This is supposed to be a bit easier to read.
- For example:

```
#!/bin/sh
smallest=10000
for i in 5 8 19 8 7 3; do
    if [ $i -lt $smallest ] ; then
        smallest=$i
    fi
done
echo $smallest
```

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 cond2 -o cond3 ...
```

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

Examples:

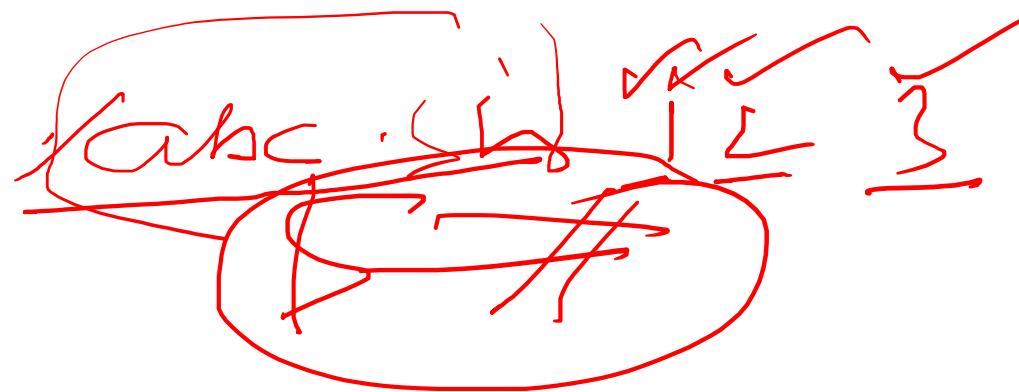
```
if date | grep "Fri" && `date +'%H'` -gt 17
then
    echo "It's Friday, it's home time!!!"
fi
```

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

Decision Logic

- Simple Example1

```
#!/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
```



Simple Example2:

```
#!/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
```

for Loops

- for loops allow the repetition of a command for a specific set of values

- Syntax:

```
for var in value1 value2 ...
do
    command_set
done
```

- command_set is executed with each value of var (value1, value2, ...) in sequence

for Loop Example (1)

```
#!/bin/bash
# timetable – print out a multiplication table
for i in 1 2 3
do
    for j in 1 2 3
    do
        value=`expr $i \* $j`
        echo -n "$value "
    done
    echo
done
```

\$ i

for Loop Example (2)

```
#!/bin/bash
# file-poke – tell us stuff about files
files='ls'
for i in $files
do
    echo -n "$i "
    grep $i $i
done
```

- Find filenames in files in current directory

for Loop Example (3)

```
#!/bin/bash
# file-poke – tell us stuff about files
for i in *; do
    echo -n "$i "
    grep $i $i
done
```

- Same as previous slide, only a little more condensed.

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:

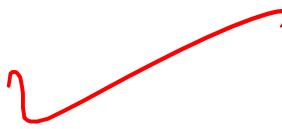
```
while this_command_execute_successfully
do
    this command
    and this command
done
```

EXAMPLE:

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

The while Loop

- While loops repeat statements as long as the next Unix command is successful.
- For example:

```
#!/bin/sh
i=1
sum=0
while [ $i -le 100 ]; do
    sum=`expr $sum + $i`  

    i=`expr $i + 1`
done
echo The sum is $sum.
```

Looping Logic: Examples

- Adding integers from 1 to 10

```
#!/bin/sh
for person in Bob Amit July Gaurav
do
    echo Hello $person
done
```

Annotations: A red bracket is drawn over the word "person" in the first line, and another red bracket is drawn over the word "done" in the fourth line.

Output:

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

```
#!/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
```

The until Loop

- Until loops repeat statements until the next Unix command is successful.
- For example:

```
#!/bin/sh
x=1
until [ $x -gt 3 ]; do
    echo x = $x
    x=`expr $x + 1`
done
```

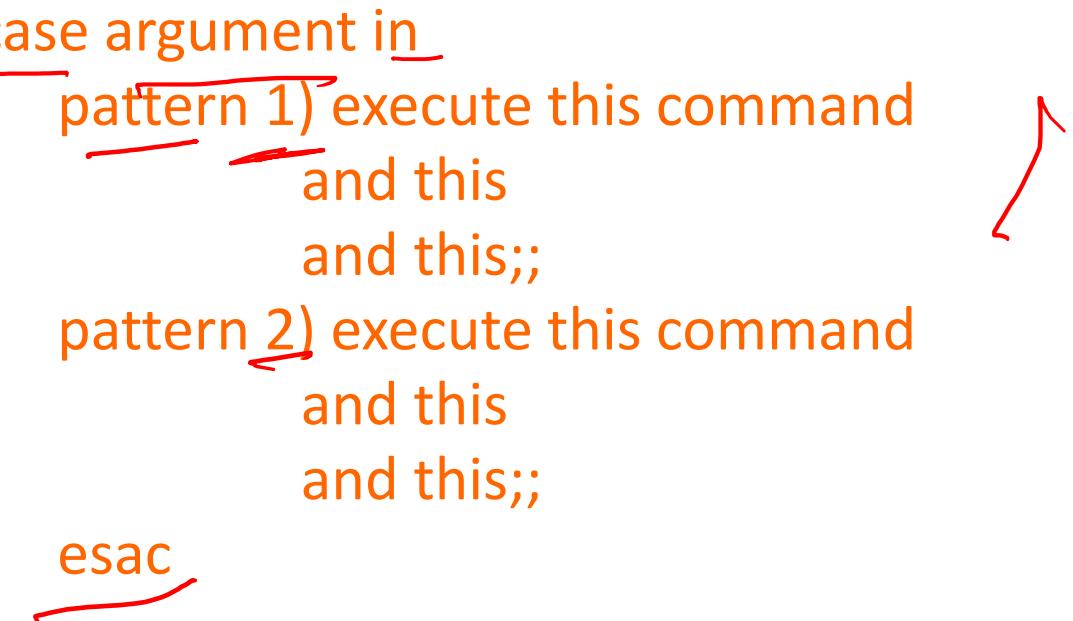
Switch/Case Logic

- The **switch logic** structure simplifies the selection of a match when you have a list of choices
- It allows our 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
```



Command Line Arguments (1)

- Shell scripts would not be very useful if we could not pass arguments to them on the command line
- Shell script arguments are “numbered” from left to right
 - `$1` - first argument after command
 - `$2` - second argument after command
 - ... up to `$9`
 - They are called “positional parameters”.

Command Line Arguments (2)

- Example: get a particular line of a file

- Write a command with the format:

`getlineno linenumber filename`

`#!/bin/sh`

`head -$1 $2 | tail -1`

- Other variables related to arguments:

- `$0` name of the command running

- `$*` All the arguments (even if there are more than 9)

- `$#` the number of arguments

Command Line Arguments (3)

- Example: print the oldest files in a directory

```
#!/bin/sh
# oldest -- examine the oldest parts of a directory
HOWMANY=$1
shift
ls -lt $* | tail +2 | tail $HOWMANY
```

- The **shift** command shifts all the arguments to the left
 - \$1 = \$2, \$2 = \$3, \$3 = \$4, ...
 - \$1 is lost (but we have saved it in \$SHOWMANY)
 - The value of \$# is changed (# - 1)
 - **useful when there are more than 9 arguments**
- The “**tail +2**” command removes the first line.

More on Bourne Shell Variables (1)

- There are three basic types of variables in a shell script:
 - Positional variables ...
 - \$1, \$2, \$3, ..., \$9
 - Keyword variables ...
 - Like \$PATH, \$HOWMANY, and anything else we may define.
 - Special variables ...

More on Bourne Shell Variables (2)

- Special variables:
 - \$*, \$# -- all the arguments, the number of the arguments
 - \$\$ -- the process id of the current shell
 - \$? -- return value of last foreground process to finish
 - more on this one later
 - There are others you can find out about with [man sh](#)

Reading Variables From Standard Input (1)

- The `read` command reads one line of input from the terminal and assigns it to variables give as arguments
- Syntax: `read var1 var2 var3 ...`
 - Action: reads a line of input from standard input
 - Assign first word to `var1`, second word to `var2`, ...
 - The last variable gets any excess words on the line.

Reading Variables from Standard Input (2)

- Example:

```
% read X Y Z
```

Here are some words as input

```
% echo $X
```

Here

```
% echo $Y
```

are

```
% echo $Z
```

some words as input

The case Statement

- The case statement supports multiway branching based on the value of a single string.
- General form:

```
case string in
  pattern1)
    command_set_11
  ;;
  pattern2)
    command_set_2
  ;;
  ...
esac
```

case Example

```
#!/bin/sh
echo -n 'Choose command [1-4] > '
read reply
echo
case $reply in
    "1")
        date
        ;;
    "2"|"3")
        pwd
        ;;
    "4")
        ls
        ;;
    *)
        echo Illegal choice!
        ;;
esac
```

Use the pipe symbol “|” as a logical or between several choices.

Provide a default case when no other cases are matched.

Redirection in bash Shell Scripts (1)

- Standard input is redirected the same (<).
- Standard output can be redirected the same (>).
 - Can also be directed using the notation 1>
 - For example: `cat x 1> ls.txt` (only stdout)
- Standard error is redirected using the notation 2>
 - For example: `cat x y 1> stdout.txt 2> stderr.txt`
- Standard output and standard error can be redirected to the same file using the notation 2>&1
 - For example: `cat x y > xy.txt 2>&1`
- Standard output and standard error can be piped to the same command using similar notation
 - For example: `cat x y 2>&1 | grep text`

Redirection in bash Shell Scripts (2)

- Shell scripts can also supply standard input to commands from text embedded in the script itself.
- General form: `command << word`
 - Standard input for `command` follows this line up to, but not including, the line beginning with `word`.
- Example:

```
#!/bin/sh
```

```
grep 'hello' << EOF
```

```
This is some sample text.
```

```
Here is a line with hello in it
```

```
Here is another line with he'o.
```

```
No more lines with that word.
```

```
EOF
```

Only these two lines will be matched and displayed.

A Shell Script Example (1)

- Suppose we have a file called `marks.txt` containing the following student grades:

091286899 90 H. White

197920499 80 J. Brown

899268899 75 A. Green

.....

- We want to calculate some statistics on the grades in this file.

A Shell Script Example (2)

```
#!/bin/sh
sum=0; countfail=0; count=0;
while read studentnum grade name; do
    sum=`expr $sum + $grade`
    count=`expr $count + 1`
    if [ $grade -lt 50 ]; then
        countfail=`expr $countfail + 1`
    fi
done
echo The average is `expr $sum / $count`.
echo $countfail students failed.
```

A Shell Script Example (3)

- Suppose the previous shell script was saved in a file called **statistics**.
- How could we execute it?
- As usual, in several ways ...
 - % cat marks.txt | statistics
 - % statistics < marks.txt
- We could also just execute **statistics** and provide marks through standard input.

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`

Array/list

- To create lists (array) – round bracket

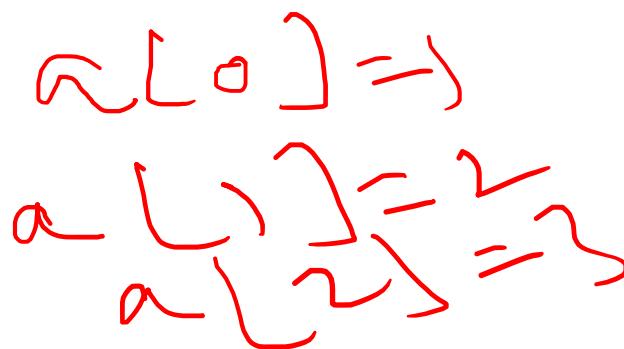
\$ set Y = (LSP 123 CSET213)

- 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 ✓

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`"

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

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-burts-ebooks.htm>
- **Shell Script in C Shell**
<http://www.grymoire.com/Unix/CshTop10.txt>
- **Linux Shell Scripting Tutorial**
<http://www.freeos.com/guides/lstt/>
- **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>





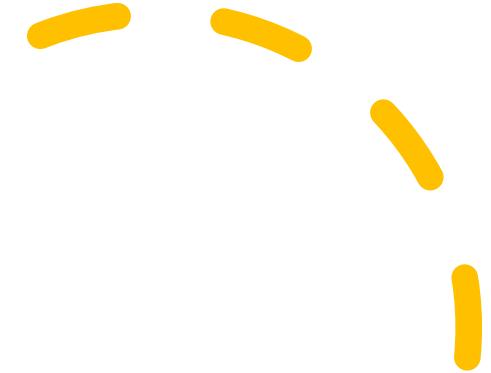
Thanks
Q & A

Linux Session, Standard Streams, Pipes & Filters in Linux



Dr. Vimal Kr Baghel (Course Instructor), Assistant Professor
School of Computer Science Engineering & Technology (SCSET)
Bennett University Greater Noida

Outline



- Linux Session
- Standard Input Output
- Redirection & Pipes
- Filters
- Q & A

Process Groups and Sessions

- A process group (Job) is created each time a command or a pipeline of commands in a shell is executed.
- In its turn, each process group belongs to a session.
- Linux kernel provides a **two-level hierarchy** for all running processes.
- As such, a process group is a set of processes, and a session is a set of related process groups.
- Another important limitation is that a process group and its members can be members of a single session.

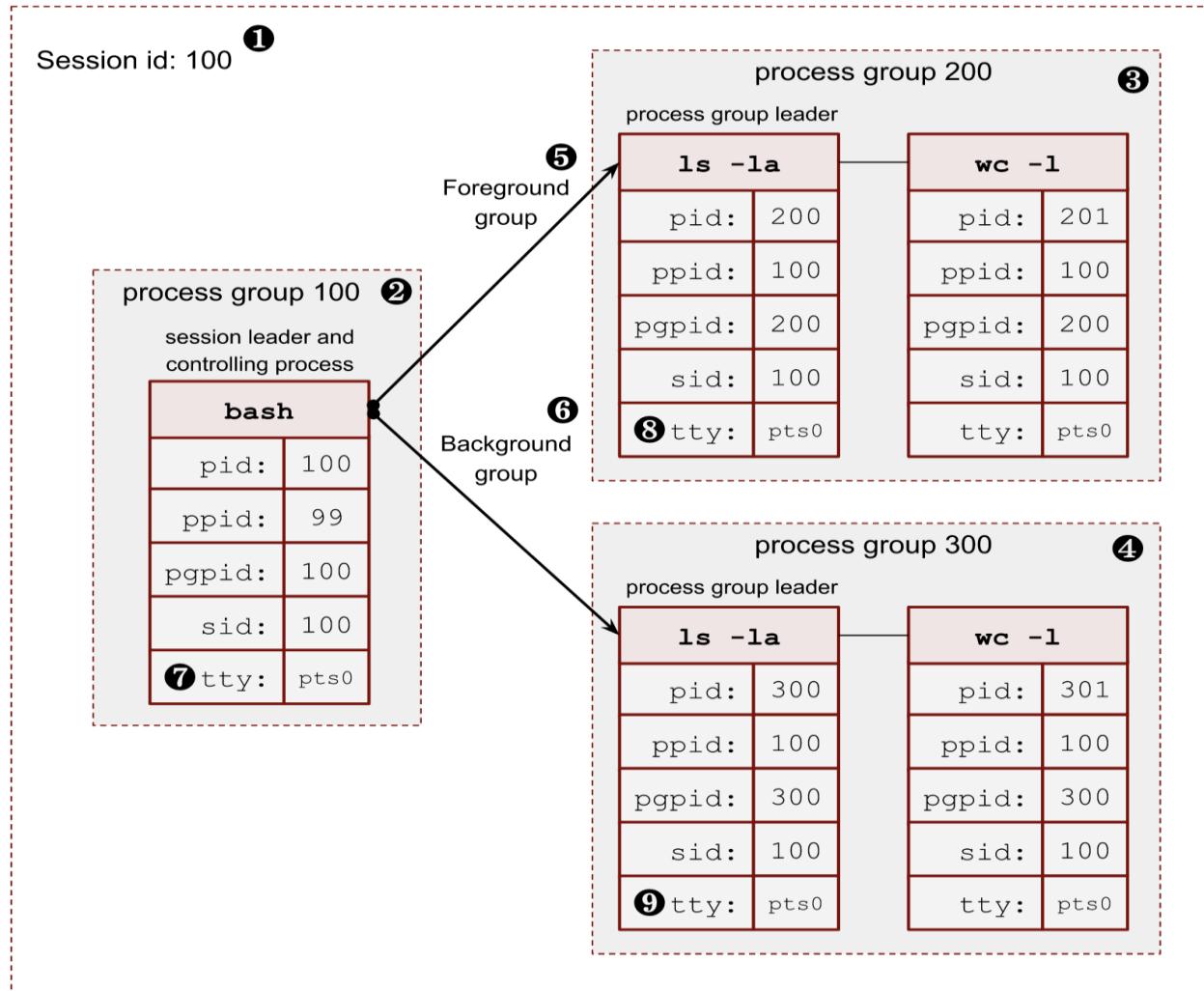
```
$ sleep 100                                # a process group with 1 process
$ cat /var/log/nginx.log | grep string | head    # a process group with 3 processes
```



More on this in wait
and signal topic....

A relationship between a session, its process groups and processes.

- ① SID is the same as the session leader process bash PID
- ② The session leader process (bash) has its own process group, where it's a leader, so PGID is the same as its PID.
- ③, ④ The session has 2 more process groups with PGIDs 200 and 300.
- ⑤, ⑥ Only one group can be a foreground for a terminal. All other process groups are background.
- ⑦, ⑧, ⑨ All members of a session share a pseudo-terminal /dev/pts/0.



Linux Sessions

- A session is a collection of process groups.
- All members of a session identify themselves by the identical SID (type: pid_t)
- As a process group, SID is also inherited from the session leader, which created the session.
- All processes in the session share a single controlling terminal
- A new process inherits its parent's session ID.
- To start a new session a process should call setsid().

Linux Sessions

- The process running this syscall begins a new session, becomes its leader, starts a new process group, and becomes its leader too.
- SID and PGID are set to the process' PID.
- That's why the process group leader can't start a new session: the process group could have members, and all these members must be in the same session.
- Basically, a new session is created in two cases:
 - When we need to log in a user with an interactive shell. A shell process becomes a session leader with a controlling terminal.
 - A daemon starts and wants to run in its own session to secure itself

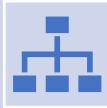
I/O in Shell



A Linux shell, such as Bash, receives input and sends **output as sequences or streams of characters.**



Each character is *independent*.



The characters are **not** organized into structured records or fixed-size blocks.



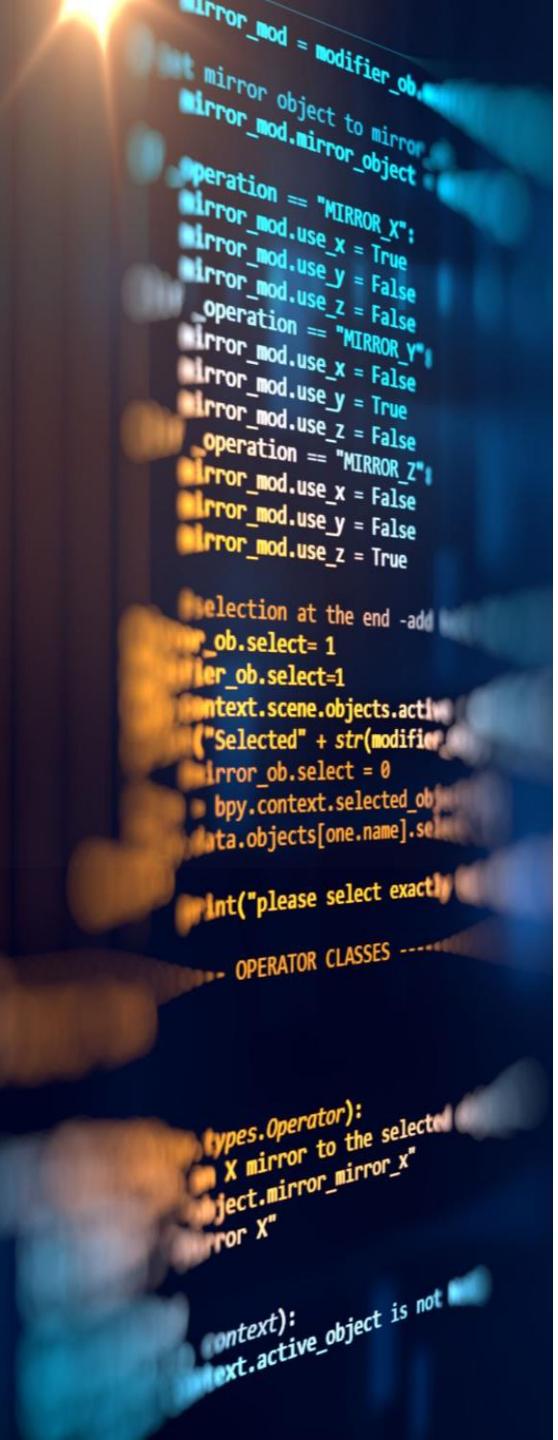
Streams are accessed using file IO techniques, **whether the actual stream of characters comes from or goes to a file, a keyboard, a window on a display, or some other IO device.**

I/O Streams

- Linux shells use 3 standard I/O streams, each of which is associated with a file descriptor (fd):
 - ***stdout*** is the *standard output stream*, which displays output from commands. It has fd 1.
 - ***stderr*** is the *standard error stream*, which displays error output from commands. It has fd 2.
 - ***stdin*** is the *standard input stream*, which provides input to commands. It has fd 0.

Redirecting standard Output

- Prepare input data in a file or save output or error information in a file.
- Redirecting output
 - There are two ways to redirect output to a file:
 - **n>**
 - **n>>**
 - Where n is a file descriptor



Redirect both standard output and standard error into a file

This is done for automated processes or background jobs so that you can review the output later.

Use **&> or &>>** to redirect both standard output and standard error to the same place

Another way of doing this is to redirect file descriptor *n* and then redirect file descriptor *m* to the same place using the construct ***m>&n or m>>&n***

The order in which outputs are redirected is important

Redirecting input

We redirect stdin from a file using the < operator

```
[vimal@baghel]$ cat text1
```

- 1 apple
- 2 pear
- 3 banana

```
[vimal@baghel]$ tr ' ' '\t'<text1
```

- 1 apple
- 2 pear
- 3 banana

Redirecting input

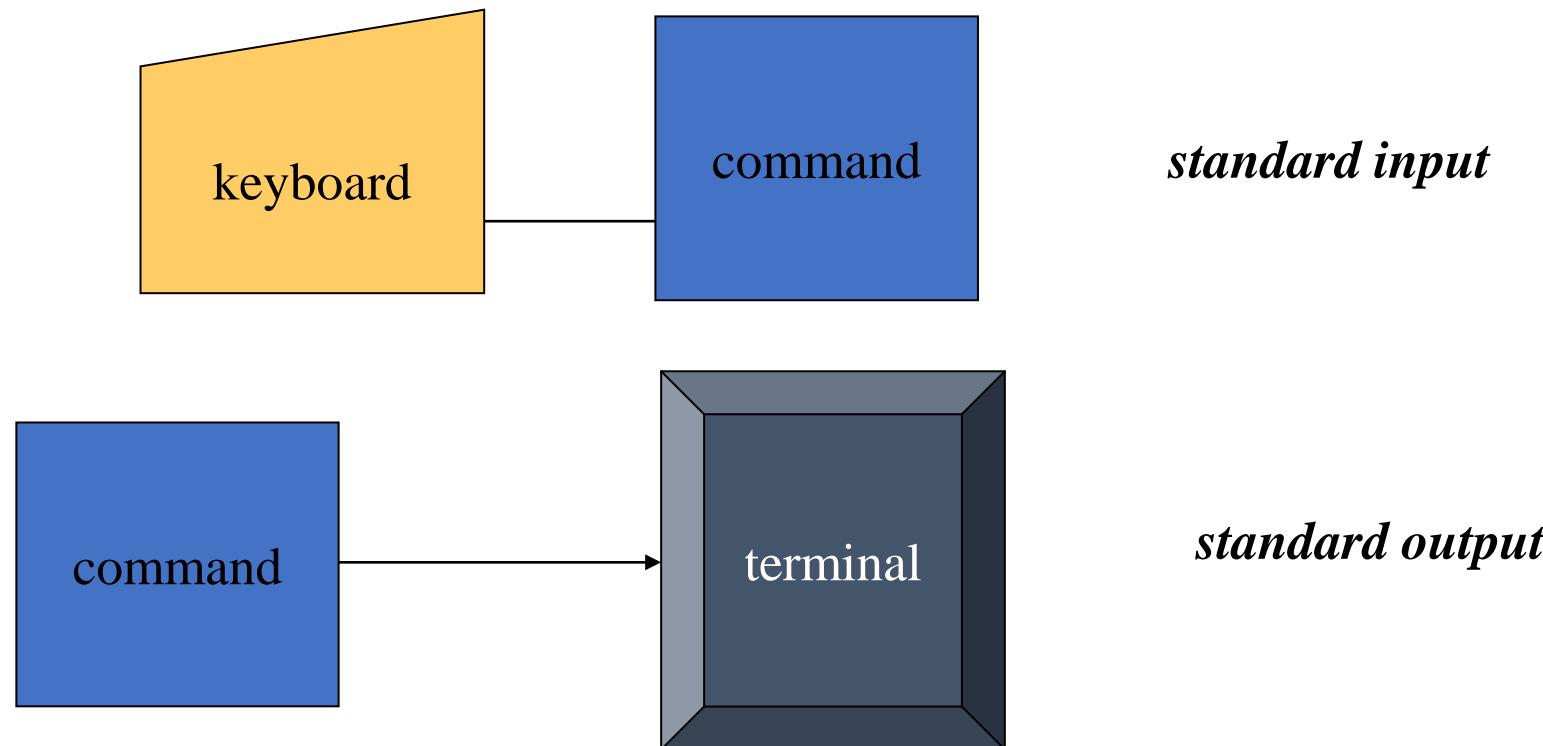
```
[vimal@baghel]$ sort -k2 <<END
```

- > 1 apple
- > 2 pear
- > 3 banana
- > END

Sorted output

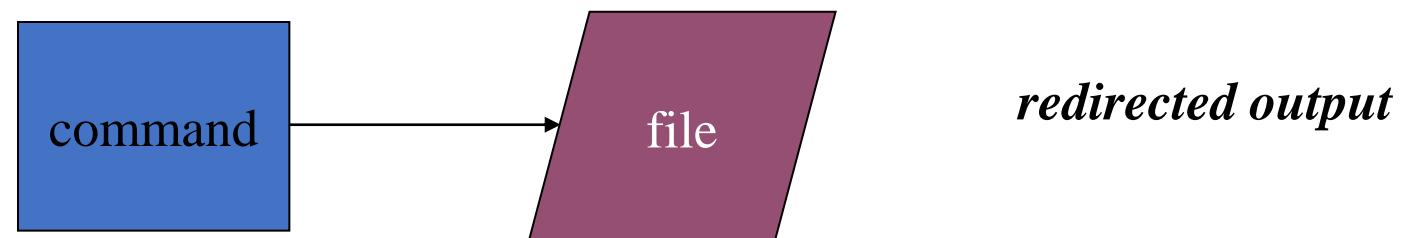
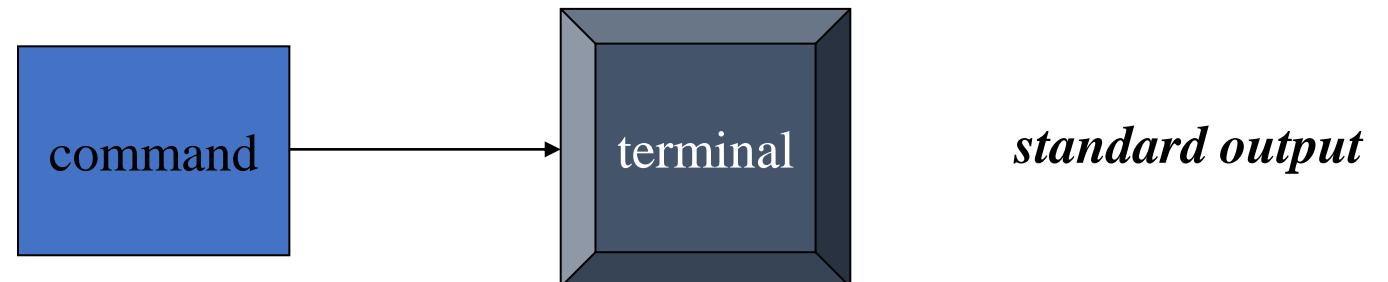
- 1 apple
- 3 banana
- 2 pear

Standard Input and Output



Redirection and Pipes

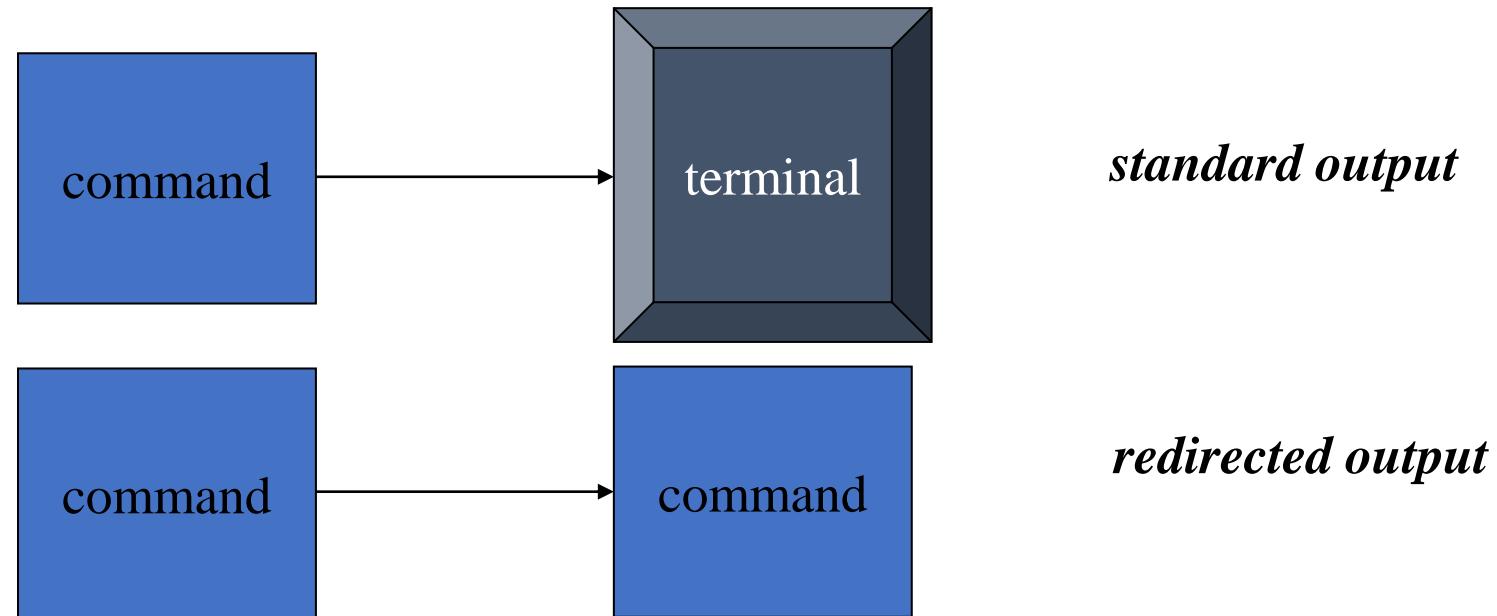
- Redirecting Standard Output to a File with: '>', '>>'



- `ls -al > mylist` (creates or overwrites file)
- `ls -al >> mylist` (appends or creates file)

Redirection and Pipes

- Redirecting Standard Output to a Command with: ‘|’



- ls -al | more (passes output to command)

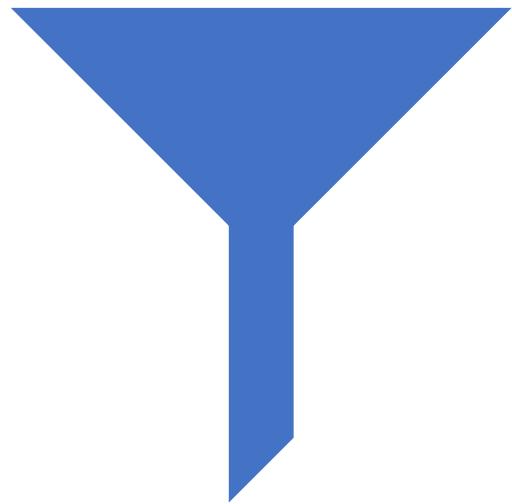
Redirection and Pipes

- Redirecting Standard Input from a File with: ‘<’



- `cat < file` (passes file to command)

Filters



- A *FILTER* is any program that reads from Standard Input and writes to Standard Output.

- grep
- uniq
- look
- spell
- sort
- wc

Filters: grep

The grep command searches for the pattern specified and writes these lines to Standard Output.

```
> grep 'Easy'  
assignments.txt
```

- **grep [-cilnvw] pattern [file...]**

Filters: uniq

The **uniq** command examines data, looking for **consecutive**, duplicate lines.

- **uniq [-cdv] [infile [outfile]]**

uniq -d , retains one copy of all lines that are duplicated

uniq -u, retains only those lines that are not duplicated.

uniq -c, counts how many times each line is found.

```
> uniq document.txt
```

Filters: look

The look command searches data in alphabetical order and will **find lines that begin with a specified pattern** (alphabetical characters only).

look [-df] pattern [file...]

>look Amer

* Access the dictionary of correctly spelled words.

- Look is not really a filter and cannot be used within a pipeline.
- *File* must be pre-sort file with –dfu options. I.e dictionary, fold, unique.

Filters: look

```
einstein.franklin.edu - PuTTY
/export/home/morris07>look Amer
Amerada
America
American
Americana
Americanism
americium
/export/home/morris07>
```



Filters: spell

- ✓ The spell command will read data and generate a list of all words that look as if they are misspelled.
- ✓ *This is a very primitive spell checker.*

```
spell [file...]  
>spell document.txt
```

Filters: sort

- The sort command sorts data (using ASCII format).

sort [-dfnru] [infile...] [-o outfile]

>sort names -o sorted_names

or

>sort names > sorted_names

Filters: sort

```
einstein.franklin.edu - PuTTY
export/home/morris07/examples>sort sorti
n

Eddie
Frank
Joey
John
John
Ralph
Sam
Sam
export/home/morris07/examples>
```

Filters: sort

```
einstein.franklin.edu - PuTTY
export/home/morris07/examples>sort -u so
rtin

Eddie
Frank
Joey
John
Ralph
Sam
export/home/morris07/examples>
```

Filters: wc

The wc command counts lines, words, and characters.

• **wc [-lwc] [file...]**

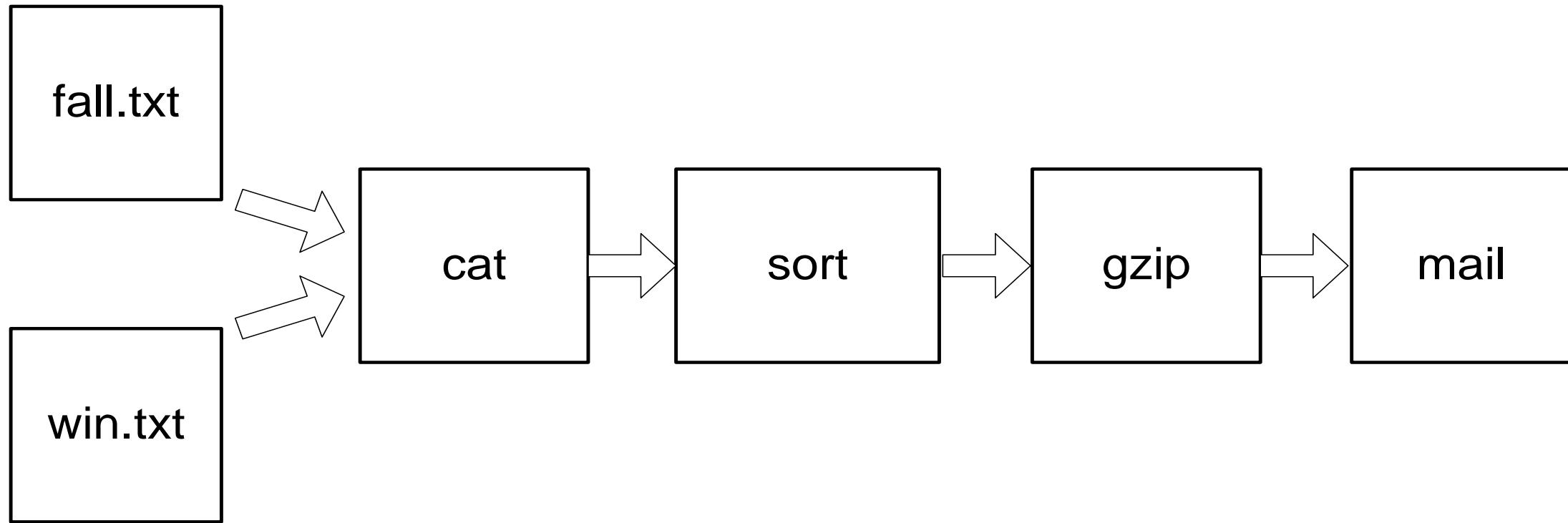
```
> wc -l  
document.txt
```

wc options

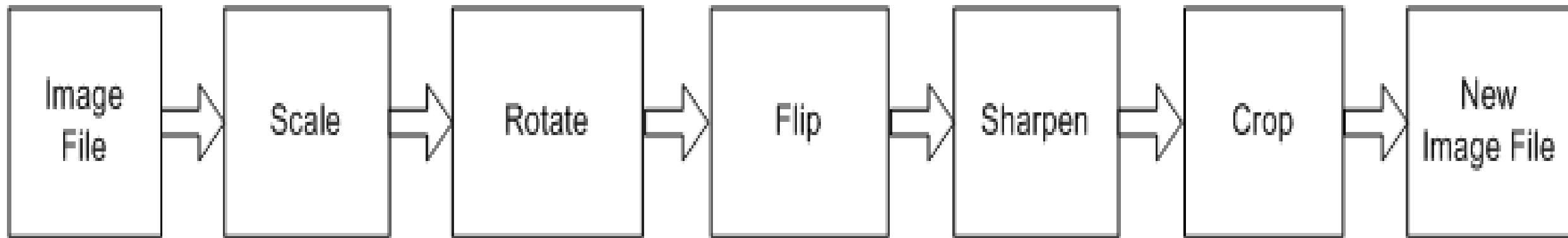
- c Count bytes.
- m Count characters.
- C Same as -m.
- l Count lines.
- w Count words delimited by white space
 characters or new line characters.

Known Uses: UNIX Command Pipelines

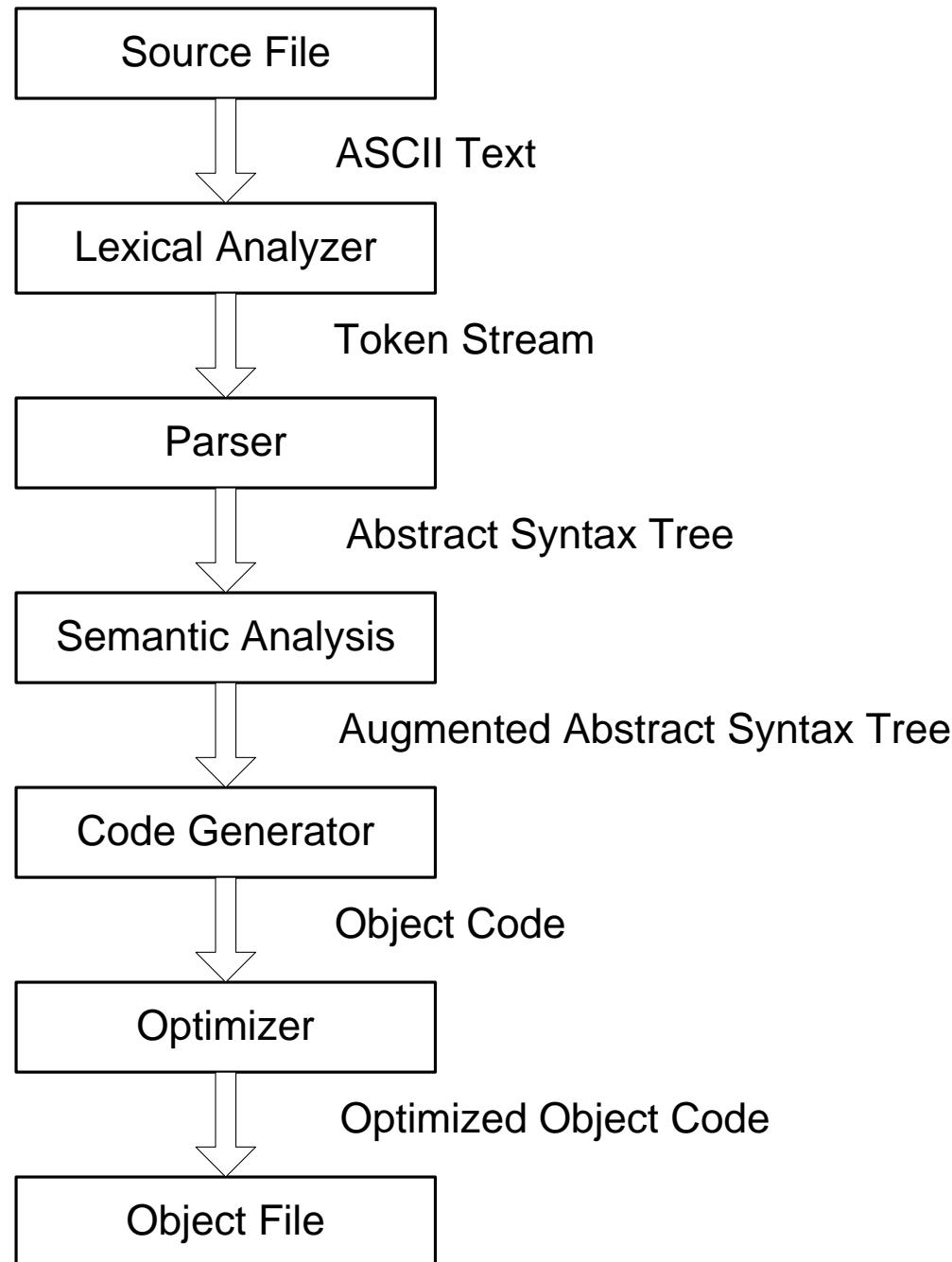
```
$ cat fall.txt win.txt | sort | gzip | mail fred@byu.edu
```



Known Uses: Image Processing



Known Uses: Compilers





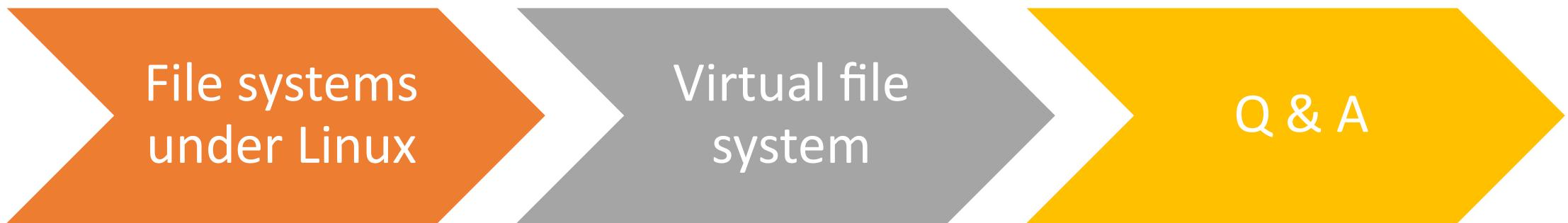
Thanks

Q & A

Linux File System & Operations

Dr. Vimal Baghel, Assistant Professor
SCSET, BU

Outline



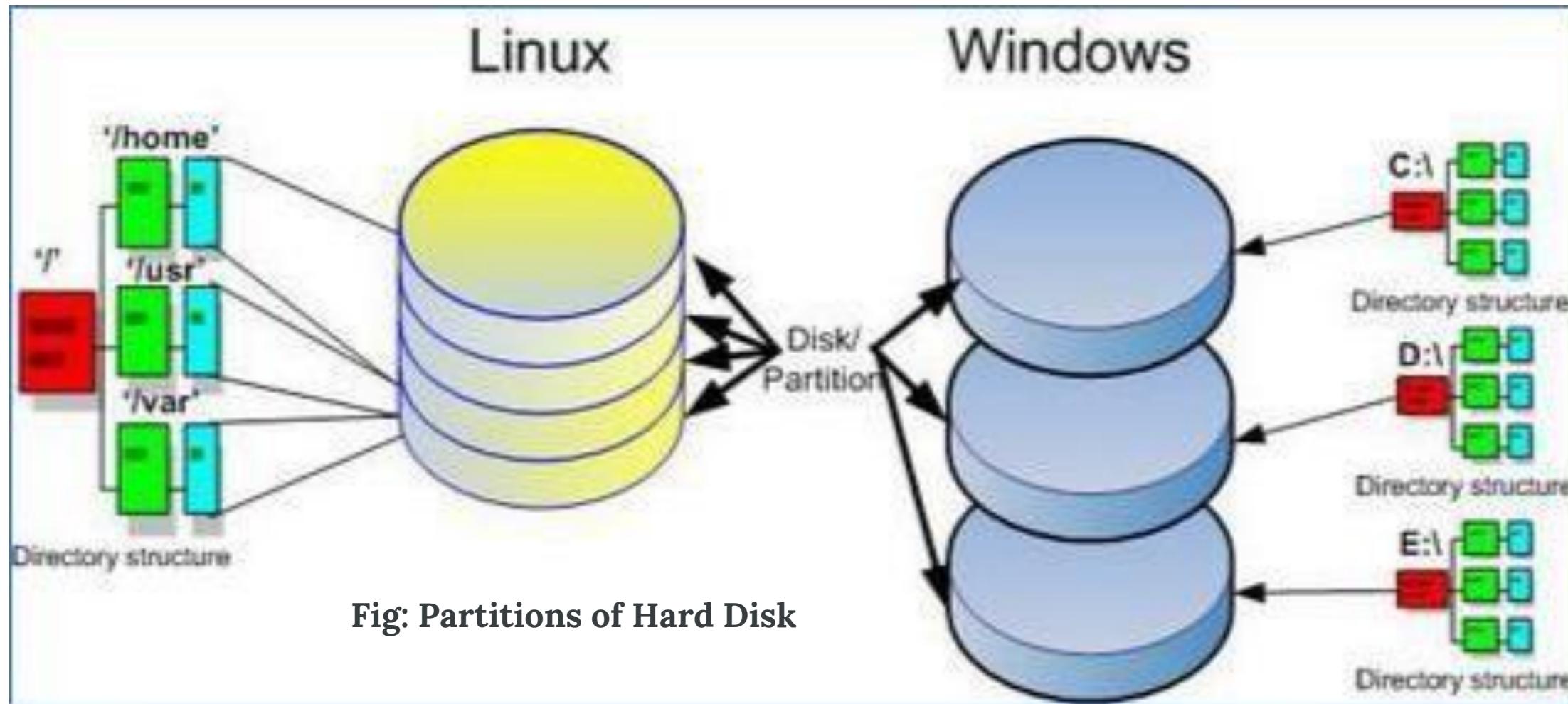
- File systems in Unix / Linux
- Symbolic links
- Mounting of file systems
- Superblock
- Inode
- Dentry object
- File object

Linux File System

- A file is an ***object that stores data, information, settings or commands*** in a computer system.
- A directory is a structure that is used ***to store files & subdir.***
- A filesystem is ***the methods and data structures that an OS uses to keep track of files on a disk or partition;*** that is, the way the files are organized on the disk.
- The file system is a data structure to store the meta-data of files.
- In Linux, the files are organized in a tree structure.

Linux File System

- A file system is a data structure that stores index of each file.
- The files reside in physical form of data blocks in the hard disk.
- The FS is responsible to map the physical & its logical location in directory structure.



file systems supported by Linux

File System	Max File Size	Max Partition Size	Notes
Fat16	2 GiB	2 GiB	Legacy
Fat32	4 GiB	8 TiB	Legacy
NTFS	2 TiB	256 TiB	(For Windows Compatibility)
ext2	2 TiB	32 TiB	Legacy
ext3	2 TiB	32 TiB	Standard linux filesystem for many years. Best choice for super-standard installation.
ext4	16 TiB	1 EiB	Modern iteration of ext3. Best choice for new installations where super-standard isn't necessary.
reiserFS	8 TiB	16 TiB	No longer well-maintained.
JFS	4PiB	32PiB	Created by IBM
XFS	8 EiB	8 EiB	Created by SGI

Linux Directory Structure

- Linux classifies the files into 3 main categories:
 - User Files – Files created and being accessed by users of the system
 - System Files – Executable files, binary files, configuration files, etc.
 - Device Files – Files corresponding to devices like sound card, graphics card, NIC, etc.

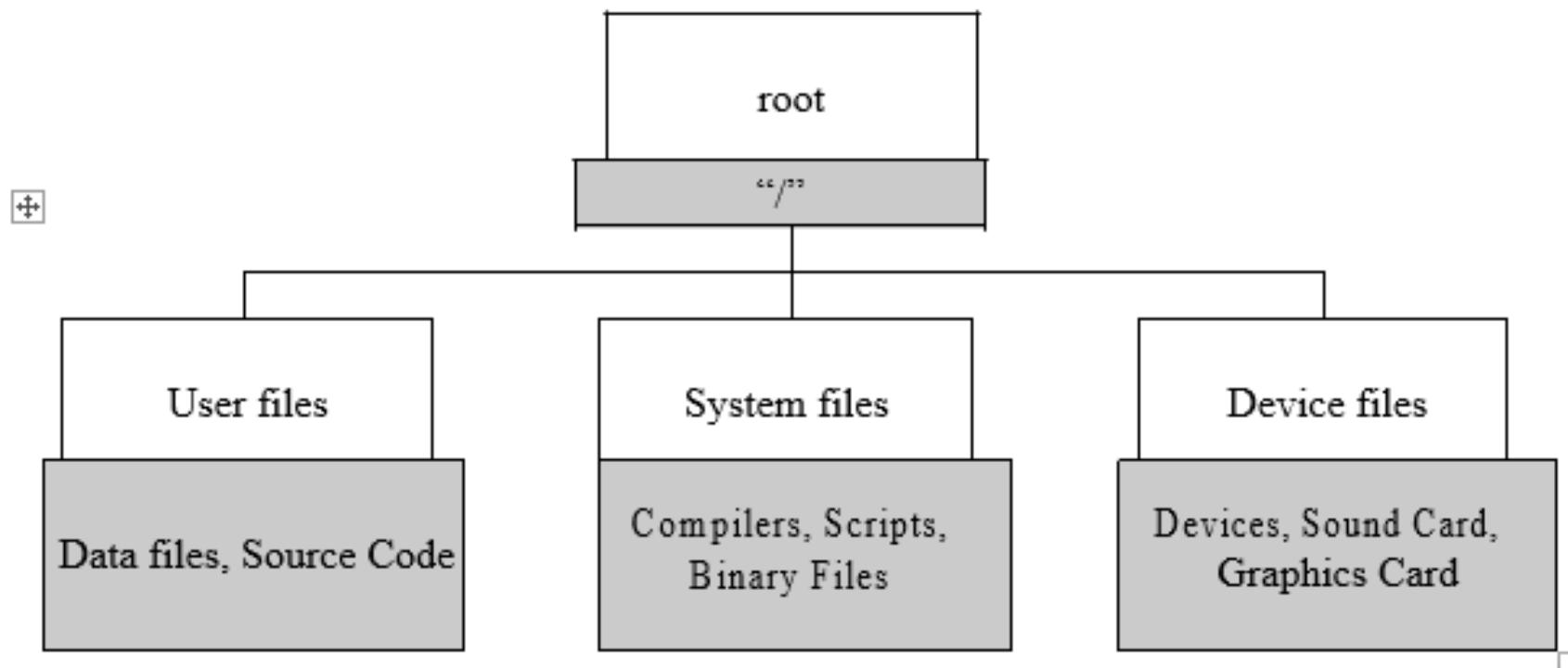


Figure 2: Linux directories & files

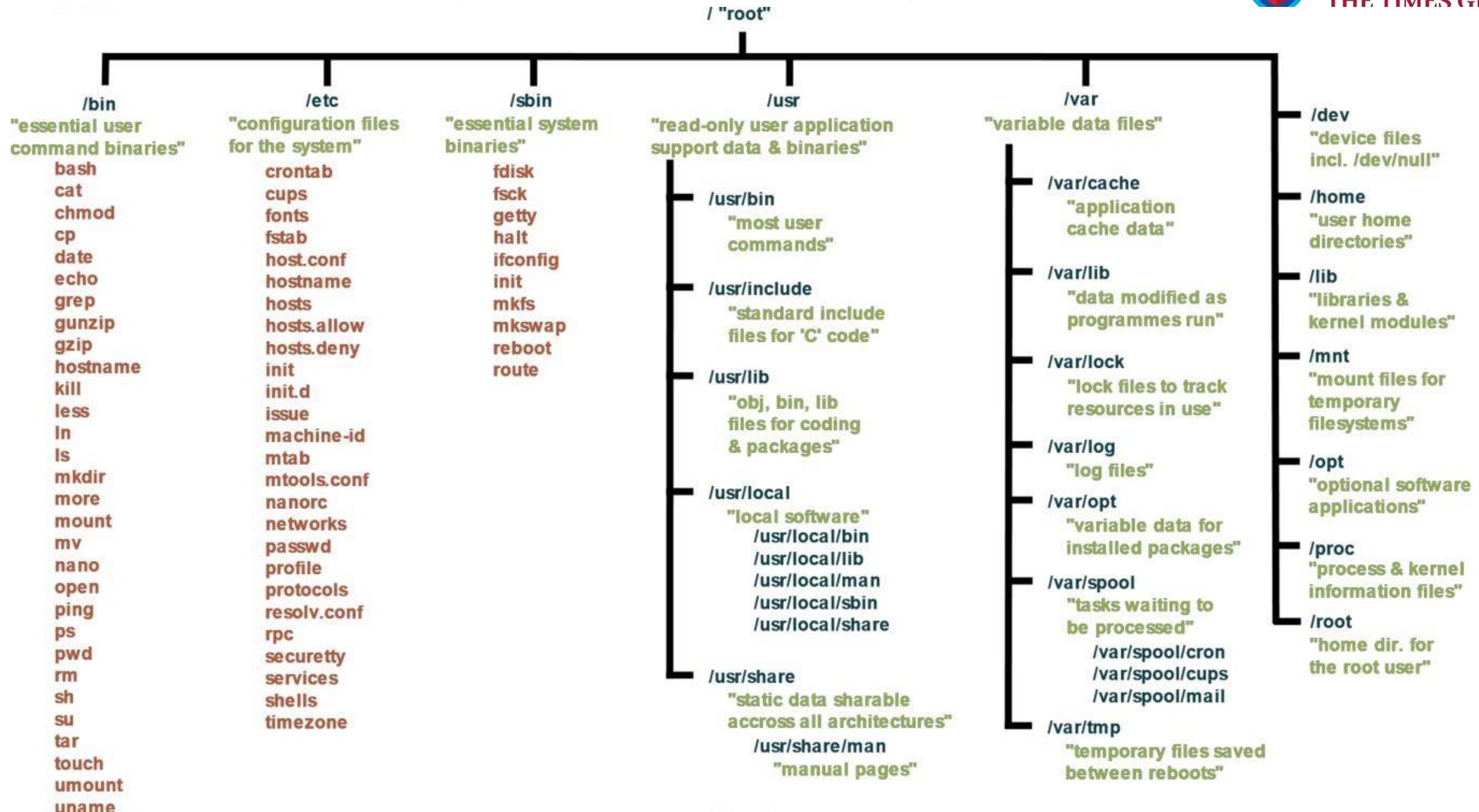
Mount Point

- A mount point is a directory in a file system where additional information is logically connected from a storage location outside the operating system's root drive and partition.

```
$ mount /dev/sda6 /home
```

```
$ umount /home
```

Linux Directory and Files Organization



Linux Files

- Linux files are classified into the following three general categories depending on the content and usage of file:
 - Regular files (-): **randomly addressable sequence of bytes**
 - Directory files (d): Contains files & other dir
 - Device files or special files: a point of interface to one of the computer's hardware devices.
 - Block files (b)
 - Character device files (c)
 - Named Pipe file (p)
 - Symbolic Link (l)
 - Socket files (s)
- ***a device file acts as a communication channel between two or more cooperating programs.***

Types of Linux Commands

External Commands:

- Commands with an independent existence in the form of a separate file.
- For example, programs for the commands such as **cat** and **ls**, exist independently in a directory called the /bin.
- When such commands are given, the shell reaches these command files with the **help of a system variable called the PATH variable** and executes them.
- Mostly Linux commands are external.

Internal/built-in Commands:

- Commands that are built into the shell.
- For example, the echo command is an internal command as its routine will be a part of the shell's routine.
- **cd** and **mkdir**, are two examples of internal commands.

Performing basic file operations

File Operations

<code>cp <i>file1 file2</i></code>	copy file1 and call it file2
<code>mv <i>file1 file2</i></code>	move or rename file1 to file2
<code>rm <i>file</i></code>	remove a file
<code>rmdir <i>directory</i></code>	remove a directory
<code>cat <i>file</i></code>	display a file
<code>more <i>file</i></code>	display a file a page at a time
<code>head <i>file</i></code>	display the first few lines of a file
<code>tail <i>file</i></code>	display the last few lines of a file
<code>less <i>file1</i></code>	Display/search a keyword in the file1
<code>grep '<i>keyword</i>' <i>file</i></code>	search a file for keywords
<code>wc <i>file</i></code>	count number of lines/words/characters in file

Advanced File Operations

- **Copying files remotely:** `scp [option] user1@host1:source user2@host2:destination`
- **Comparing files:** `diff [option] file1 file2`
- **Finding files:** `find search_path [option]`
- **Searching files according to use case:**

```
#!/bin/bash
# Filename: finding_files.sh
echo -n "Number of C/C++ header files in system: "
find / -name "*.h" 2>/dev/null | wc -l
echo -n "Number of shell script files in system: "
find / -name "*.sh" 2>/dev/null | wc -l
echo "Files owned by user who is running the script ..."
echo -n "Number of files owned by user $USER :"
find / -user $USER 2>/dev/null | wc -l
echo -n "Number of executable files in system: "
find / -executable 2>/dev/null | wc -l
```

- **Finding and deleting a file based on inode number:**

```
$ find ~/ -inum 8159146 -exec rm -i {} \;
```

Links to a file:

- A soft link or a symbolic link
- A hard link

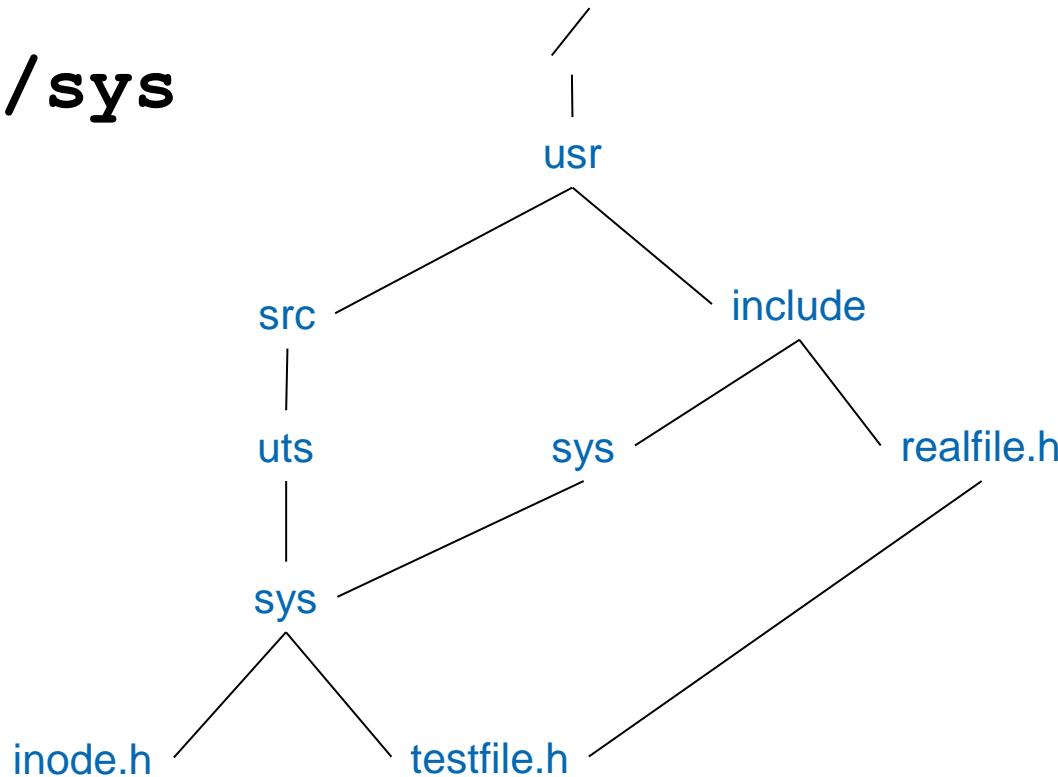
```
ln [option] target link_name
```

Soft or Symbolic Links

To know whether a file is a symbolic link or not, run `ls -l` on a file:

```
$ ls -l ~/tmp  
lrwxrwxrwx. 1 foo foo 5 Aug 23 23:31 /home/foo/tmp -> /tmp/
```

```
$ ln -s /usr/src/uts/sys ~/sys
```



Hard Links

- A hard link is a way to refer a file with different names. All such files will have the same inode number.
- To create a hard link of a file, use the `ln` command without any option.

```
$ touch file.txt
$ ls -l file.txt
-rw-rw-r--. 1 foo foo 0 Aug 24 00:13 file.txt
```

- to create a hard link of `file.txt` `$ ln file.txt hard_link_file.txt`

- To check whether a hard link is created for `file.txt`

```
$ ls -l file.txt
-rw-rw-r--. 2 foo foo 0 Aug 24 00:13 file.txt
```

```
$ ls -i file.txt hard_link_file.txt
96844 file.txt
96844 hard_link_file.txt
```

Difference between hard link and soft link

Soft link	Hard link
The inode number of the actual file and the soft link file are different.	The inode number of the actual file and the hard link file are the same.
A soft link can be created across different filesystems.	A hard link can only be created in the same filesystem.
A soft link can link to both regular files and directories.	A hard link doesn't link to directories.
Soft links are not updated if the actual file is deleted. It keeps pointing to a nonexistent file.	Hard links are always updated if the actual file is moved or deleted.

Special files

- The block device file
- The character device file
- The named pipe file
- The socket file

The block device file

- A block device file is a file that reads and writes data in block.
- Such files are useful when data needs to be written in bulk.
- Devices such as hard disk drive, USB drive, and CD-ROM are considered as block device files.
- Data is written asynchronously and, hence, other users are not blocked to perform the write operation at the same time.

The block device file

- To create a block device file, mknod is used with the option b along with providing a major and minor number.
- A major number selects which device driver is being called to perform the input and output operation.
- A minor number is used to identify subdevices:

```
$ sudo mknod block_device b 0x7 0x6 → minor number
```



major number

it is a block device file

```
$ ls -l block_device
brw-r--r--. 1 root root 7, 6 Aug 24 12:21 block_device
```



Character Device File

- A character device file is a file that reads and writes data in character-by-character fashion.
- Such devices are synchronous and only one user can do the write operation at a time.
- Devices such as keyboard, printer, and mouse are known as character device files.
- Following command will create a character special file:

```
$ sudo mknod character_device c 0X78 0X60
```

it is a character device file

```
$ ls -l character_device # viewing attribute of character_device file
c.rw-r--r--. 1 root root 120, 96 Aug 24 12:21 character_device
```

Named pipe file

- Named pipe files are used by different system processes to communicate with each other.
- Such communication is also known as inter-process communication(IPC).
- To create such a file, we use the mkfifo command:

```
$ mkfifo pipe_file # Pipe file created
$ ls pipe_file # Viewing file content
prw-rw-r--. 1 foo foo 0 Aug 24 01:41 pipe_file
```



it is a pipe file

- We can also create a named pipe using the mknod command with the p option:

```
$ mknod named_pipe_file p
$ ls -l named_pipe_file
prw-rw-r--. 1 foo foo 0 Aug 24 12:33 named_pipe_file
```

Script to send and receive a message over/from pipe file

```
#!/bin/bash
#Filename: send.sh
#Script which sends message over pipe
pipe=/tmp/named_pipe
if [[ ! -p $pipe ]]
then
mkfifo $pipe
fi
echo "Hello message from Sender">>$pipe
```

```
#!/bin/bash
#Filename: receive.sh
#Script receiving message from sender from
pipe file
pipe=/tmp/named_pipe
if [[ ! -p $pipe ]]
then
echo "Reader is not running"
fi
while read line
do
echo "Message from Sender:"
echo $line
done < $pipe
```

To execute it, run `send.sh` in a terminal and `receive.sh` in another terminal:

```
$ sh send.sh # In first terminal
$ sh receive.sh # In second terminal
Message from Sender:
Hello message from Sender
```

Socket file

- A socket file is used to pass information from one application to another.
- For example, if **Common UNIX Printing System (CUPS)** daemon is running and my printing application wants to communicate with it, then my printing application will write a request to a socket file where CUPS daemon is listening for upcoming requests.
- Once a request is written to a socket file, the daemon will serve the request:

```
$ ls -l /run/cups/cups.sock # Viewing socket file attributes
srw-rw-rw-. 1 root root 0 Aug 23 15:39 /run/cups/cups.sock
```

it is a socket file

Temporary files

- Such files are being used to keep intermediate results of running a program and they are no longer needed after the program execution is complete.

Creating a temporary file using mktemp:

```
$ mktemp  
/tmp/tmp.xEXxXYeRcF
```



- ✓ The `mktemp` command creates a temporary file and prints its name on `stdout`.
- ✓ Temporary files are created by default in the `/tmp` directory.

inode (Index node)

- Each file is represented by an Inode
- It contains
 - Owner (UID, GID)
 - Access rights
 - Time of last modification / access
 - Size
 - Type (file, directory, device, pipe, ...)
 - **Pointers to data blocks that store file's content**



Directories (file catalogues)

- Directories are handled as normal files, but are marked in Inode-- type as directory
- A directory entry contains
 - Length of the entry
 - Name (variable length up to 255 characters)
 - Inode number
- Multiple directory entries may reference the same Inode number (hard link)
- Users identify files via pathnames ("path/to/file") that are mapped to Inode numbers by the OS
- If the path starts with "/", it is absolute and is resolved up from the root directory
- Otherwise, the path is resolved relative to the current directory

Directories

Each directory contains an entry "." that represents the Inode of the current directory



The second entry ".." references parent directory



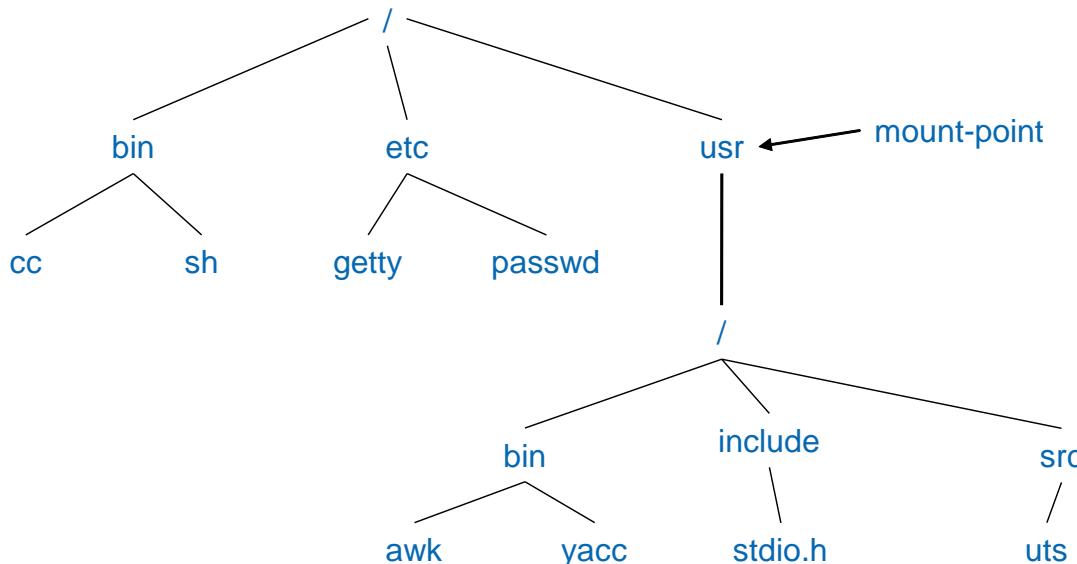
The path is resolved from left to right and the respective name is looked up in the directory



As long as the current name is not the last in the path, it has to be a directory. Otherwise, the lookup terminates with an error

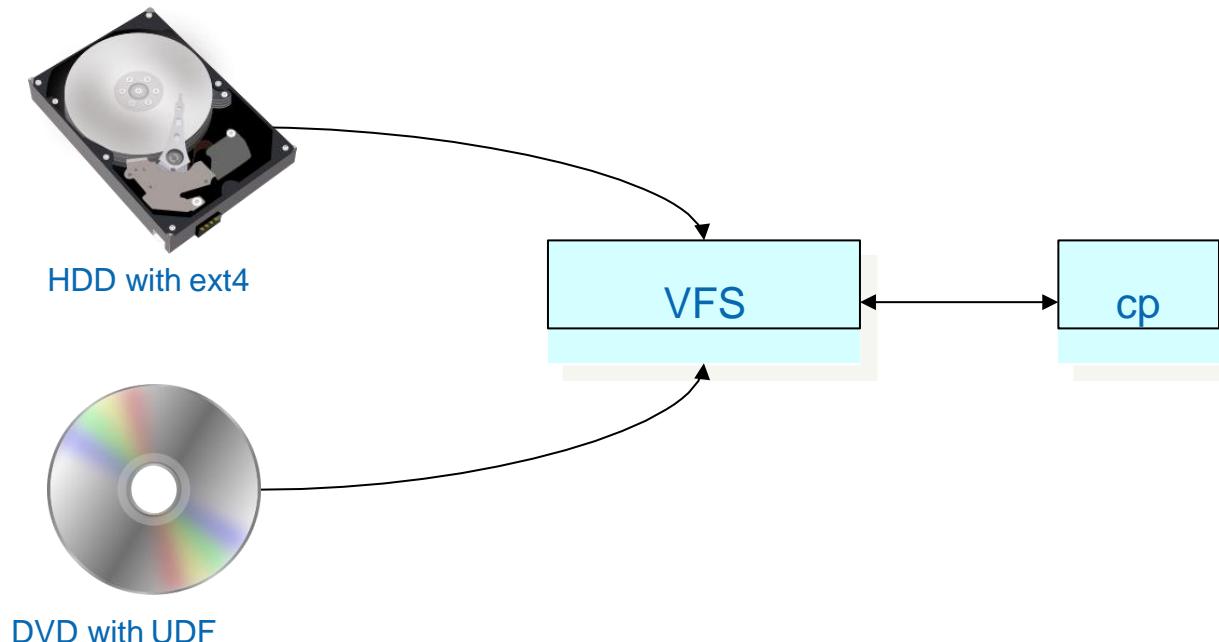
Logical and Physical File System

- A logical file system may consist of multiple physical file systems
- A file system can be hooked into any path of the virtual file system tree with the "mount" command
- Mounted file systems are managed by the OS in a "mount table" that connects paths to mount points
- This allows to identify the root Inodes of mounted file systems



Virtual File System

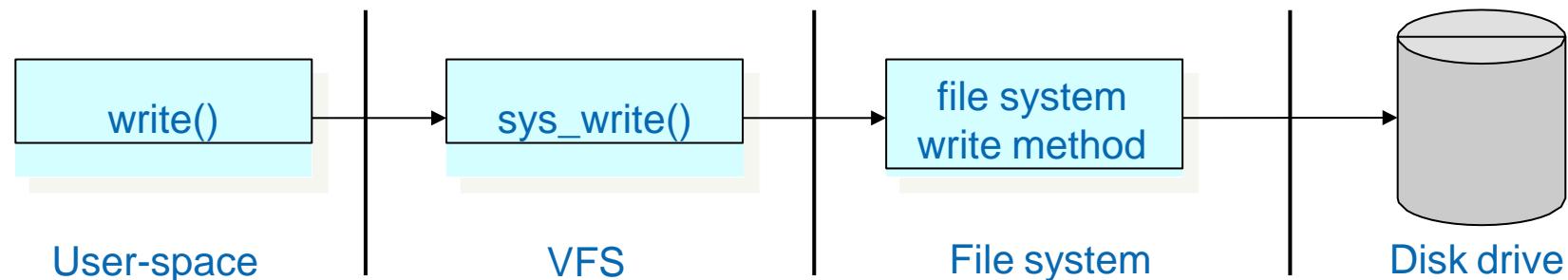
- The Virtual File System (VFS) implements a generic file system interface between the actual file system implementation (in kernel) and accessing applications to provide interoperability
- ➔➔ Applications can access different file systems on different media via a homogeneous set of UNIX system calls



Virtual File System

Example: `write(f, &buf, len);`

- Write of `len` Bytes in file with descriptor `f` from Buffer `buf` is translated into system call
- The system call is forwarded to the actual file system implementation
- The file system executes the write command



VFS Objects and Data Structures

- VFS is object oriented
- Four base objects
 - Super block: Represents specific properties of a file system
 - Inode: File description
 - Dentry: The directory entry represents a single component of a path
 - File: Representation of an open file that is associated with a process
- VFS handles directories like files
 - Dentry object represents component of a path that may be a file
 - Directories are handled like files as Inodes
 - Each object provides a set of operations

Superblock

- Each file system must provide a superblock
 - Contains properties of the file system
 - Is stored on special sectors of disk or is created dynamically (i.e. by sysfs)
 - Structure is created by alloc_super() when the file system is mounted

```
struct super_block {
    struct list_head           s_list;          /* Keep this first */
    dev_t                       s_dev;           /* search index; _not_ kdev_t */
    unsigned long                s_blocksize;
    unsigned char                 s_blocksize_bits;
    unsigned char                 s_dirt;
    unsigned long long            s_maxbytes;      /* Max file size */
    struct file_system_type     *s_type;
    struct super_operations     *s_op;
    struct dquot_operations     *dq_op;
    struct quotactl_ops         *s_qcop;
    struct export_operations     *s_export_op;
    unsigned long                  s_flags;
    unsigned long                  s_magic;
    struct dentry                  s_root;
    struct rw_semaphore           s_umount;
    struct mutex                     s_lock;
    int                           s_count;
    int                           s_syncing;
    int                           s_need_sync_fs;
    atomic_t                      s_active;
    void                          *s_security;
    **s_xattr;

    struct list_head           s_inodes;        /* all inodes */
    struct list_head           s_dirty;         /* dirty inodes */
    struct list_head           s_io;             /* parked for writeback */
    struct hlist_head           s_anon;          /* anonymous dentries for (nfs) exporting */
    struct list_head           s_files;

    struct block_device         *s_bdev;
    struct list_head           s_instances;
    struct quota_info           s_dquot;          /* Diskquota specific options */

    unsigned int                  s_prunes;        /* protected by dcache_lock */
    wait_queue_head_t           s_wait_prunes;

    int                         s_frozen;
    wait_queue_head_t           s_wait_unfrozen;
    char s_id[32];                  /* Informational name */
    void                        *s_fs_info;       /* Filesystem private info */
    /*
     * The next field is for VFS *only*. No filesystems have any business
     * even looking at it. You had been warned.
     */
    struct semaphore s_vfs_rename_sem;      /* Kludge */
    /* Granularity of c/m/atime in ns.
     * Cannot be worse than a second */
    u32                          s_time_gran;
};

};
```

Superblock Operations

- Each entry contains pointer to a function
 - File system provides implementation for the operations
 - Example:
Write superblock sb:

sb->s op->write super(sb)

```
/*
 * NOTE: write_inode, delete_inode, clear_inode, put_inode can be called
 * without the big kernel lock held in all filesystems.
 */
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void (*destroy_inode)(struct inode *);

    void (*read_inode) (struct inode *);

    void (*dirty_inode) (struct inode *);
    int (*write_inode) (struct inode *, int);
    void (*put_inode) (struct inode *);
    void (*drop_inode) (struct inode *);
    void (*delete_inode) (struct inode *);
    void (*put_super) (struct super_block *);
    void (*write_super) (struct super_block *);
    int (*sync_fs)(struct super_block *sb, int wait);
    void (*write_super_lockfs) (struct super_block *);
    void (*unlockfs) (struct super_block *);
    int (*statfs) (struct super_block *, struct kstatfs *);
    int (*remount_fs) (struct super_block *, int *, char *);
    void (*clear_inode) (struct inode *);
    void (*umount_begin) (struct super_block *);

    int (*show_options)(struct seq_file *, struct vfsmount *);

    ssize_t (*quota_read)(struct super_block *, int, char *,
        size_t, loff_t);
    ssize_t (*quota_write)(struct super_block *, int,
        const char *, size_t, loff_t);
};

};
```

Inode Object

- Contains information specific to a file
 - For typical Unix file systems, an Inode can directly be read from disk
 - Other file systems hold this information as part of or in a database
- inode has to be created by the file system
- Special Entries for non-data files
 - i.e. `i_pipe`, `i_bdev`, or `i_cdev` are reserved for pipes, block and character devices
 - Some entries are not supported by all file systems and may therefore be set to Null

```
struct inode {  
    struct hlist_node  
    struct list_head  
    struct list_head  
    struct list_head  
    unsigned long  
    atomic_t  
    umode_t  
    unsigned int  
    uid_t  
    gid_t  
    dev_t  
    loff_t  
    struct timespec  
    struct timespec  
    struct timespec  
    unsigned int  
    unsigned long  
    unsigned long  
    unsigned long  
    unsigned short  
    spinlock_t  
    struct mutex  
    struct rw_semaphore  
    struct inode_operations  
    struct file_operations  
    struct super_block  
    struct file_lock  
    struct address_space  
    struct address_space  
    struct dquot  
    struct list_head  
    struct pipe_inode_info  
    struct block_device  
    struct cdev  
    int  
    __u32  
    unsigned long  
    struct dnotify_struct  
    struct list_head  
    struct semaphore  
    unsigned long  
    unsigned long  
    unsigned int  
    atomic_t  
    void  
    union {  
        void  
    } u;  
    seqcount_t  
    i_hash;  
    i_list;  
    i_sb_list;  
    i_dentry;  
    i_ino;  
    i_count;  
    i_mode;  
    i_nlink;  
    i_uid;  
    i_gid;  
    i_rdev;  
    i_size;  
    i_atime;  
    i_mtime;  
    i_ctime;  
    i_blkbits;  
    i_blksize;  
    i_version;  
    i_blocks;  
    i_bytes;  
    i_lock;  
    i_mutex;  
    i_alloc_sem;  
    *i_op;  
    *i_fop;  
    *i_sb;  
    *i_flock;  
    *i_mapping;  
    i_data;  
    *i_dquot[MAXQUOTAS];  
    i_devices;  
    *i_pipe;  
    *i_bdev;  
    *i_cdev;  
    i_cindex;  
    i_generation;  
    i_dnotify_mask;  
    *i_dnotify;  
    inotify_watches;  
    inotify_sem;  
    i_state;  
    dirtied_when;  
    i_flags;  
    i_writecount;  
    *i_security;  
    *generic_ip;  
    i_size_seqcount;
```

Inode Operations

```
struct inode_operations {
    int (*create) (struct inode *, struct dentry *, int, struct nameidata *);
    struct dentry * (*lookup) (struct inode *, struct dentry *, struct nameidata *);
    int (*link) (struct dentry *, struct inode *, struct dentry *);
    int (*unlink) (struct inode *, struct dentry *);
    int (*symlink) (struct inode *, struct dentry *, const char *);
    int (*mkdir) (struct inode *, struct dentry *, int);
    int (*rmdir) (struct inode *, struct dentry *);
    int (*mknod) (struct inode *, struct dentry *, int, dev_t);
    int (*rename) (struct inode *, struct dentry *,
                  struct inode *, struct dentry *);
    int (*readlink) (struct dentry *, char __user *, int);
    void * (*follow_link) (struct dentry *, struct nameidata *);
    void (*put_link) (struct dentry *, struct nameidata *, void *);
    void (*truncate) (struct inode *);
    int (*permission) (struct inode *, int, struct nameidata *);
    int (*setattr) (struct dentry *, struct iattr *);
    int (*getattr) (struct vfsmount *mnt, struct dentry *, struct kstat *);
    int (*setxattr) (struct dentry *, const char *, const void *, size_t, int);
    ssize_t (*getxattr) (struct dentry *, const char *, void *, size_t);
    ssize_t (*listxattr) (struct dentry *, char *, size_t);
    int (*removexattr) (struct dentry *, const char *);
    void (*truncate_range)(struct inode *, loff_t, loff_t);
};
```

- Inode Operations describe the set of operations that are implemented by the file system and are accessed via VFS

Directories Objects

- Unix directories are handled like files
- The path /bin/vi contains the directories / and bin as well as the file vi
- Resolution of paths requires introduction of dentry objects
- Each part of a path is dentry object
- VFS creates dentry objects on the fly
- No equivalent on disk drive
- Are stored in dentry cache (handled by OS)
 - Frontend of Inode cache

```
struct dentry {  
    atomic_t d_count;  
    unsigned int d_flags; /* protected by d_lock */  
    spinlock_t d_lock; /* per dentry lock */  
    struct inode *d_inode; /* Where the name belongs to - NULL is  
    * negative */  
  
    /*  
     * The next three fields are touched by __d_lookup. Place them here  
     * so they all fit in a cache line.  
     */  
    struct hlist_node d_hash; /* lookup hash list */  
    struct dentry *d_parent; /* parent directory */  
    struct qstr d_name;  
  
    struct list_head d_lru; /* LRU list */  
    /*  
     * d_child and d_rcu can share memory  
     */  
    union {  
        struct list_head d_child; /* child of parent list */  
        struct rcu_head d_rcu;  
    } d_u;  
    struct list_head d_subdirs; /* our children */  
    struct list_head d_alias; /* inode alias list */  
    unsigned long d_time; /* used by d_revalidate */  
    struct dentry_operations *d_op;  
    struct super_block *d_sb; /* The root of the dentry tree */  
    void *d_fsdata; /* fs-specific data */  
#ifdef CONFIG_PROFILING  
    struct dcookie_struct *d_cookie; /* cookie, if any */  
#endif  
    int d_mounted;  
    unsigned char d_iname[DNAME_INLINE_LEN_MIN]; /* small names */  
};
```

File Object

- File object represents open file
- Interface to applications
- Is created as reply to `open()` system call
- Is removed on `close()`
- Different processes can open a file multiple times ➔ different file objects
- The file object is an in--memory data structure of the OS

```
struct file {
    union {
        struct list_head          fu_list;
        struct rcu_head           fu_rcuhead;
    } f_u;
    struct dentry               *f_dentry;
    struct vfsmount             *f_vfsmnt;
    struct file_operations      *f_op;
    atomic_t                   f_count;
    unsigned int                f_flags;
    mode_t                      f_mode;
    loff_t                      f_pos;
    struct fown_struct          f_owner;
    unsigned int                f_uid, f_gid;
    f_ra;
    f_version;
    *f_security;
    *private_data;
    f_ep_links;
    f_ep_lock;
    *f_mapping;
};

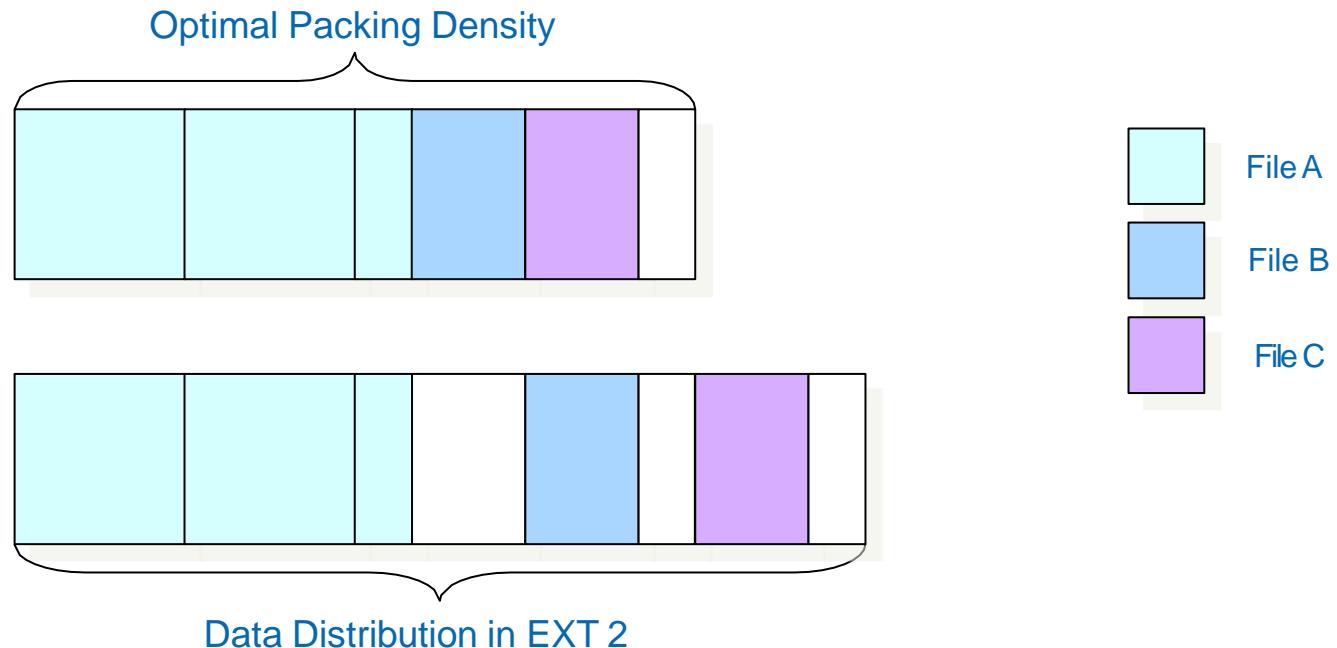
};
```

File operations

```
/*
 * NOTE:
 * read, write, poll, fsync, readyv, writev, unlocked_ioctl and compat_ioctl
 * can be called without the big kernel lock held in all filesystems.
 */
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_write) (struct kiocb *, const char __user *, size_t, loff_t *);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, struct dentry *, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*readyv) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
    ssize_t (*sendfile) (struct file *, loff_t *, size_t, read_actor_t, void *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*dir_notify)(struct file *filp, unsigned long arg);
    int (*flock) (struct file *, int, struct file_lock *);
#define HAVE_FOP_OPEN_EXEC
    int (*open_exec) (struct inode *);
};
```

Physical Architecture

- Block based devices have sectors as smallest addressable unit
- EXT2 is block based file system that partitions the hard disk into blocks (clusters) of the same size
- Blocks are used for metadata and data
- Blocks lead to internal fragmentation

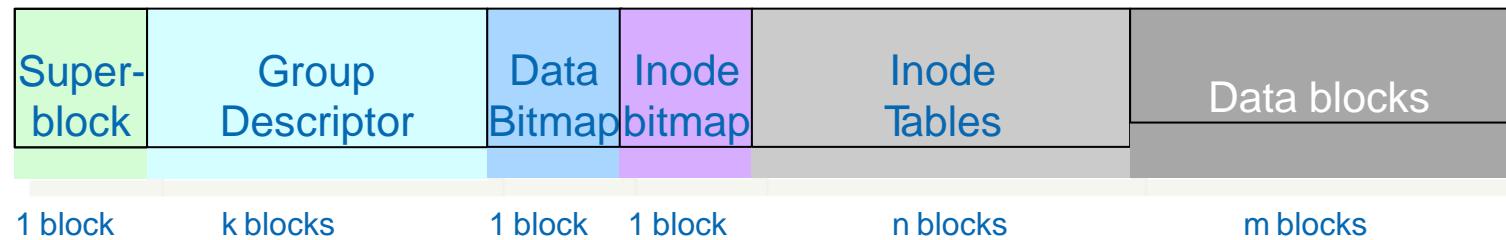


Structural Architecture of EXT 2

- EXT2 divides storage system into block groups



- Boot block is equivalent to first sector on hard disk
- Block group is basic component, which contains further file system components



Metadata

- Superblock: Central structure, which contains number of free and allocated blocks, state of the file system, used block size, ...
- Group descriptor contains the state, number of free blocks and inodes in each block group. Each block group contains group descriptor!
- Data bitmap: 1/0 allocation representation for data blocks
- Inode bitmap: 1/0 allocation representation for inode blocks
- Inode table stores all inodes for this block group

- Data blocks store user data

Data Structures

- EXT2 stores metadata in each block group
- Basic idea:
 - If a system crash corrupts the superblock, then there are enough redundant copies of it
 - Distance between metadata and data is small ➔ fewer head movements
- Implementations work differently:
 - Kernel only works with in RAM copy of the first superblock, which is written back to redundant super blocks during file system checks
 - Later versions of EXT2 include *Sparse Superblock* option, where superblocks are only stored in group 0, 1 as well in groups, which are a power of 3, 5, or 7

Group descriptor

- One copy of the descriptor for each block group in the kernel
- Block descriptor for each block group in each block group
 - ➔ Bitmaps can be accessed from everywhere
- Pointer to bitmaps with allocation information of blocks and inodes
 - ➔ Number of blocks in each block group restricted by block size
- Position of free blocks can be directly calculated from position in bitmap
- Counter for free structures

```
struct ext2_group_desc
{
    __le32 bg_block_bitmap;
    __le32 bg_inode_bitmap;
    __le32 bg_inode_table;
    __le16 bg_free_blocks_count;
    __le16 bg_free_inodes_count;
    __le16 bg_used_dirs_count;
    __le16 bg_pad;
    __le32 bg_reserved[3];
};
```

EXT2 Inodes

- `i_mode` stores access permissions for and type of file
- Several time stamps
- `i_size` and `i_blocks` store size in bytes, resp. blocks
- `i_block` contains pointer to direct and indirect block links
- `i_links_count` counts hard links

```

struct ext2_inode {
    __le16 i_mode;          /* File mode */
    __le16 i_uid;           /* Low 16 bits of Owner Uid */
    __le32 i_size;          /* Size in bytes */
    __le32 i_atime;         /* Access time */
    __le32 i_ctime;         /* Creation time */
    __le32 i_mtime;         /* Modification time */
    __le32 i_dtime;         /* Deletion Time */
    __le16 i_gid;           /* Low 16 bits of Group Id */
    __le16 i_links_count;   /* Links count */
    __le32 i_blocks;        /* Blocks count */
    __le32 i_flags;         /* File flags */

    union {
        struct {
            __le32 l_i_reserved1;
        } linux1;
        struct {
            __le32 h_i_translator;
        } hurd1;
        struct {
            __le32 m_i_reserved1;
        } masix1;
    } osd1;                  /* OS dependent 1 */

    __le32 i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
    __le32 i_generation;        /* File version (for NFS) */
    __le32 i_file_acl;         /* File ACL */
    __le32 i_dir_acl;          /* Directory ACL */
    __le32 i_faddr;             /* Fragment address */

    union {
        struct {
            __u8 l_i_frag;           /* Fragment number */
            __u8 l_i_fsize;          /* Fragment size */
            __u16 i_pad1;
            __le16 l_i_uid_high;     /* these 2 fields */
            __le16 l_i_gid_high;     /* were reserved2[0] */
            __u32 l_i_reserved2;
        } linux2;
        struct {
            .....
        } hurd2;
        struct {
            .....
        } masix2;
    } osd2;                  /* OS dependent 2 */
};

```

How does OS find an Inode?

```

static struct ext2_inode *ext2_get_inode(struct super_block *sb, ino_t ino,
                                         struct buffer_head **p)
{
    struct buffer_head * bh;
    unsigned long block_group;
    unsigned long block;
    unsigned long offset;
    struct ext2_group_desc * gdp;

    *p = NULL;
    if ((ino != EXT2_ROOT_INO && ino < EXT2_FIRST_INO(sb)) ||
        ino > le32_to_cpu(EXT2_SB(sb)->s_es->s_inodes_count))
        goto Einval;

    block_group = (ino - 1) / EXT2_INODES_PER_GROUP(sb);
    gdp = ext2_get_group_desc(sb, block_group, &bh);
    if (!gdp)
        goto Egdp;
    /*
     * Figure out the offset within the block group inode table
     */
    offset = ((ino - 1) % EXT2_INODES_PER_GROUP(sb)) * EXT2_INODE_SIZE(sb);
    block = le32_to_cpu(gdp->bg_inode_table) +
            (offset >> EXT2_BLOCK_SIZE_BITS(sb));
    if (!(bh = sb_bread(sb, block)))
        goto Eio;

    *p = bh;
    offset &= (EXT2_BLOCK_SIZE(sb) - 1);
    return (struct ext2_inode *) (bh->b_data + offset);

Einval:
    ext2_error(sb, "ext2_get_inode", "bad inode number: %lu",
               (unsigned long) ino);
    return ERR_PTR(-EINVAL);

Eio:
    ext2_error(sb, "ext2_get_inode",
               "unable to read inode block - inode=%lu, block=%lu",
               (unsigned long) ino, block);

Egdp:
    return ERR_PTR(-EIO);
}

```

Is it a valid Inode address?

In which group resides Inode

Information about the group

Offset within the group

Read data from disk / from Cache

Directory entries in EXT2

- Directories are handled as standard inodes
- `ext2_dir_entry` marks directory entry
- Inode contains associated inode number
- `name_len` stores length of directory name
 - Has to be multiple of four
 - Can be filled with /0
- `rec_len` points to next entry

```
struct ext2_dir_entry_2 {
    __le32  inode;                      /* Inode number */
    __le16  rec_len;                    /* Directory entry length */
    __u8    name_len;                  /* Name length */
    __u8    file_type;
    char   name[EXT2_NAME_LEN];        /* File name */
};
```

Directory entries in EXT2

inode	rec_len	name_len	file_type	name							
	12	1	2.	\0	\0	\0					
	12	2	2.	.	\0	\0					
	16	8	4h	a	r	d	d	i	s	k	
	32	5	7l	i	n	u	x	\0	\0	\0	
	16	6	2d	e	l	d	i	r	\0	\0	
	16	6	1s	a	m	p	l	e	\0	\0	
	16	7	2s	o	u	r	c	e	\0	\0	

Corresponds to the following directory:

```

drwxr-xr-x   3 brinkman users    4096 Dec 10 19:44 .
drwxrwxrwx  13 brinkman users   8192 Dec 10 19:44 ..
brw-r-r--   1 brinkman users    3,  0 Dec 10 19:44 harddisk
lrwxrwxrwx   1 brinkman users     14 Dec 10 19:44 linux->/usr/src/linux
-rw-r--r--   1 brinkman users     13 Dec 10 19:44 sample
drwxr-xr-x   2 brinkman users    4096 Dec 10 19:44 source

```

How does the os find a file?

Example: Opening the file `/home/user/.profile`:

- `/` is always stored in Inode 2 of the root file system
 - (Exception: Process was `chroot`'ed)
 - Open Inode 2, read data of Inode, lookup entry `home` and read its inode number
 - Open Inode for `home`, read its data, lookup entry for `user` and read its inode number
 - Open Inode for `user`, read its data, lookup entry for `.profile` and read its inode number
 - Open Inode for `.profile`, read its data, create a `struct file`
 - A pointer to the file is added to the file pointer table of the OS
- ➔ The file descriptor table of the calling process is updated with the new pointer

Allocation of data blocks

- Allocation of data blocks always necessary if the file becomes bigger
- Aim: Map successive addresses sequentially to the storage system
- Approach of `ex2_get_block()`
 - If there is a logical block directly before address of current block ➔ take next physical block
 - Else take physical block number of the block with the logical block number directly before the logical block number of the current block
 - Else take block number of first block in block group, where inode is stored
- Target block can be already occupied
 - Task of `ext2_alloc_branch()`: Allocate nearby block based on goal-block
- `ext2_alloc_block()` includes options for the preallocation of blocks
- Orlov-Allokator: typically no relationship between subdirectories in root directory ➔ if there a new subdirectory is created in the root directory, just place it somewhere



Thanks
Q & A

Shell Scripting: Metacharacters

Dr. Vimal Baghel, Assistant Professor, SCSET, BU

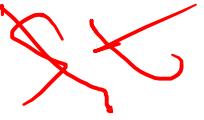
Metacharacters

- A metacharacter is a character *that has a special meaning during pattern processing.*
- We use metacharacters in *regular expressions* to define the search criteria and any text manipulations.

^ ^ ^

- ✓ All characters enclosed between single quotation marks are considered quoted and are interpreted literally by the shell.
- ✓ The special meaning of metacharacters is retained if not quoted.

' * '



Metacharacters

Metacharacters

- ✓ pipe (|),
- ✓ ampersand (&),
- ✓ semicolon (;),
- ✓ less-than sign (<),
- ✓ greater-than sign (>),
- ✓ left parenthesis ((),
- ✓ right parenthesis ()) ,

Metacharacters

- ✓ dollar sign (\$),
- ✓ backquote (`),
- ✓ backslash (\),
- ✓ right quote (‘),
- ✓ double quotation marks ("),
- ✓ newline character \n,
- ✓ space character , and tab character \t.

Types of Metacharacters

- ✓ *Search string* metacharacters
- ✓ *Replacement string* metacharacters.

Search string Metacharacters

Metacharacter Action

^

Beginning of line

^ - *AGFA*

\$

End of line

|

Or Not applicable to basic regular expressions.

[abc]

Match any character enclosed in the brackets

[^abc]

Match any character not enclosed in the brackets

[a-z]

Match the range of characters specified by the hyphen

Search string Metacharacters (classes)

- Use the character list that is specified by **cclass:alnum** = Uppercase and lowercase alphabetic characters and numbers: [A-Za-z0-9]
- **alpha** = Uppercase and lowercase alphabetic characters: [A-Za-z]
- **blank** = Whitespace and tab characters
- **cntrl** = Control characters
- **digit** = Numbers: [0-9]
- **lower** = Lowercase alphabetic characters: [a-z]
- **print** = Printable characters (the **graph** class plus whitespace)
- **punct** = Punctuation marks: !"#\$%&'()*+,-./:;<=>?@\[\]\\^_`{|}~
- **graph** = Visible characters (the **alnum** class plus the **punct** class)

Search string Metacharacters (classes)

- **space** = Whitespace characters: tab, newline, carriage-return, form-feed, and vertical-tab
- **upper** = Uppercase alphabetic characters: [A-Z]
- **xdigit** = Hexadecimal characters: [0-9a-fA-F]
- *These classes are valid for single-byte character sets.*

[=*cname*=] • Substitute the character name that is specified by *cname* with the corresponding character code.

Search string Metacharacters

- . Match any single character.
- () Group the regular expression within the parentheses.
- ? Match zero or one of the preceding expression. Not applicable to basic regular expressions.
- * Match zero, one, or many of the preceding expression.
- + Match one or many of the preceding expression. Not applicable to basic regular expressions.
- \ Use the literal meaning of the metacharacter. For basic regular expressions, treat the next character as a metacharacter.

Replacement string metacharacters

Metacharacter

Action

&

- Reference the entire matched text for string substitution.
- **For example**, the statement execute function `regex_replace('abcdefg', '[af]', '.&')` replaces 'a' with '.a.' and 'f' with '.f.' to return: '.a.bcde.f.g'.

Replacement string metacharacters

- Reference the subgroup n within the matched text, where n is an integer 0-9.
- \n
- \0 and & have identical actions.
 - \1 - \9 substitute the corresponding subgroup.

Replacement string metacharacters

Use the literal meaning of the metacharacter, **for example**, \& escapes the Ampersand symbol \ and \\ escapes the backslash. For basic regular expressions, treat the next character as a metacharacter.

Metacharacters

- Most commonly used: *
- Search the current directory for file names in which any strings occurs in the position of *

```
% echo * # same effect as  
% ls *
```

- To protect metacharacters from being interpreted: enclose them in single quotes.

```
% echo '***'  
***
```

Metacharacters (cont.)

- Or to put a backslash \ in front of each character:

```
% echo \*\\*\*
```

- Double quotes can also be used to protect metacharacters, but ...
- The shell will interpret \$, \ and `...` inside the double quotes.
- So don't use double quotes unless you intend some processing of the quoted string (to be discussed later).

Quotes

- Quotes do not have to surround the whole argument.

```
% echo x'*'y      # same as echo 'x*y'
```

x*y

- What's the difference between

```
% ls x*y
```

```
% ls 'x*y'
```



Thanks
Q & A

Regular Expressions & grep

Dr. Vimal Baghel, Assistant Professor, SCSET, BU

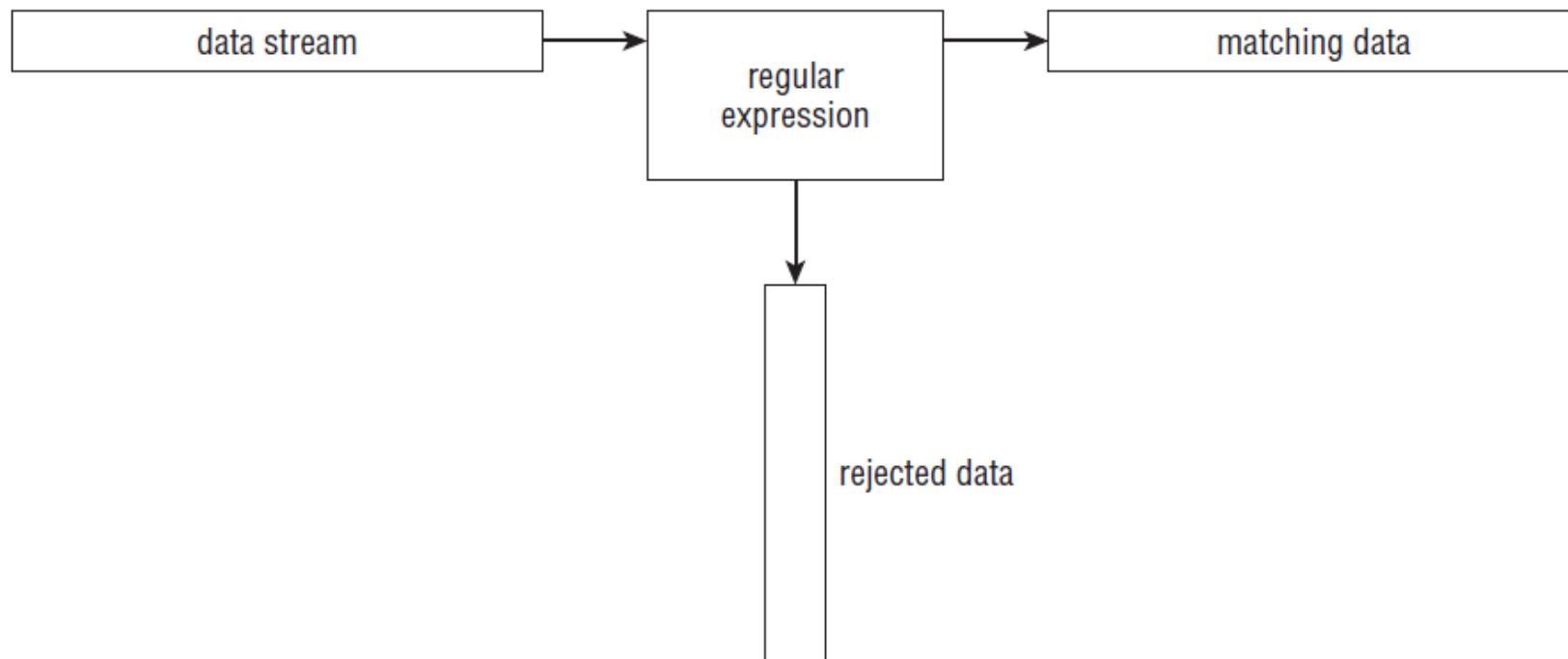
Outline

- Regular Expressions
 - Definition, and Usage
 - Character classes
 - Anchors
 - repetitions
 - Subexpressions
- grep
 - Grep family

What is Regular Expression?

- A *regular expression* is a pattern template we define that a Linux utility (sed/awk) uses to filter text.
- A Linux utility matches the regular expression pattern against data as that data flows into the utility.
 - If the data matches the pattern, it's accepted for processing.
 - If the data doesn't match the pattern, it's rejected.

Matching data against a regular expression pattern



What is regex?

- A *regular expression (regex)* is a rule that a computer can use to match characters or groups of characters within a larger body of text.
- **regex is a set of possible input strings, descended from finite automata theory**
 - For instance, using regular expressions, you could find all the instances of the word *cat* in a document, or all instances of a word that begins with *c* and ends with *t*.
- *regex is used in*
 - **vi, ed, sed, and emacs**
 - **awk, tcl, perl and Python**
 - **grep, egrep, fgrep**
 - **compilers**

Regular Expressions

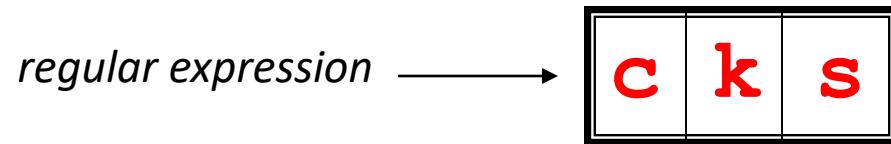
- The simplest regular expressions are a string of literal characters to match.
- The string *matches* the regular expression if it contains the substring.
- Once mastered, regular expressions provide developers with the ability to locate patterns of text in source code and documentation at design time.
- You can also apply regular expressions to text that is subject to algorithmic processing at runtime such as content in HTTP requests or event messages.

Power of Regex is more!

- Regex usage in the real world can get much more complex, and powerful.
 - **Example:** Imagine you need to code to verify contents in the body of an HTTP POST request is free of script injection attacks.
- Injected script code will always appear between <script></script> HTML tags.
- You can apply the regular expression <script>.*</script>, which matches any block of code text bracketed by <script> tags, to the HTTP request body as part of your search for script injection code.

Types of regular expressions

- A regular expression is implemented using a *regular expression engine*.
- A regular expression engine is the underlying software that interprets regular expression patterns and uses those patterns to match text.
- The Linux world has two popular regular expression engines:
 - The POSIX Basic Regular Expression (BRE) engine
 - The POSIX Extended Regular Expression (ERE) engine
- Most Linux utilities at a minimum conform to the POSIX BRE engine specifications, recognizing all the pattern symbols it defines.
- The POSIX ERE engine is often found in programming languages that rely on regular expressions for text filtering.



UNIX Tools **rocks.**

↑
match

UNIX Tools **sucks.**

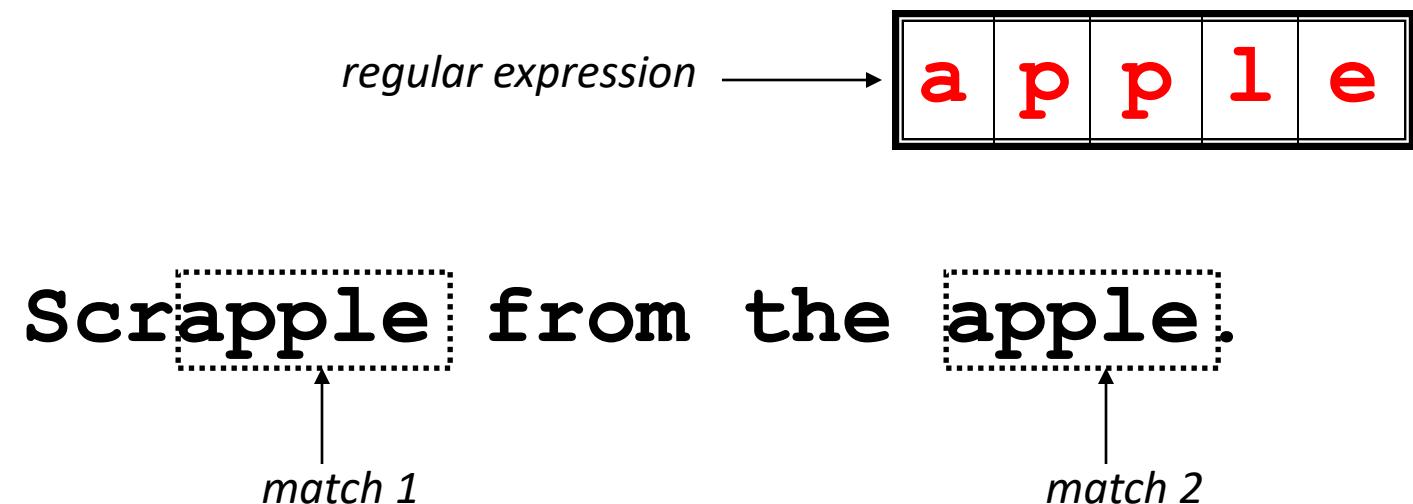
↑
match

UNIX Tools **is okay.**

no match

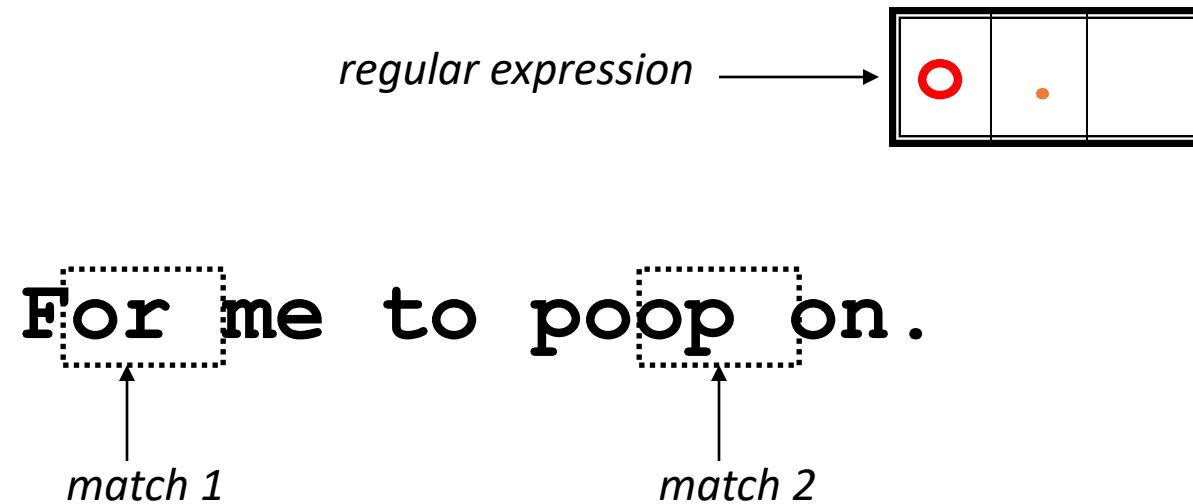
Regular Expressions

- A regular expression can match a string in more than one place.



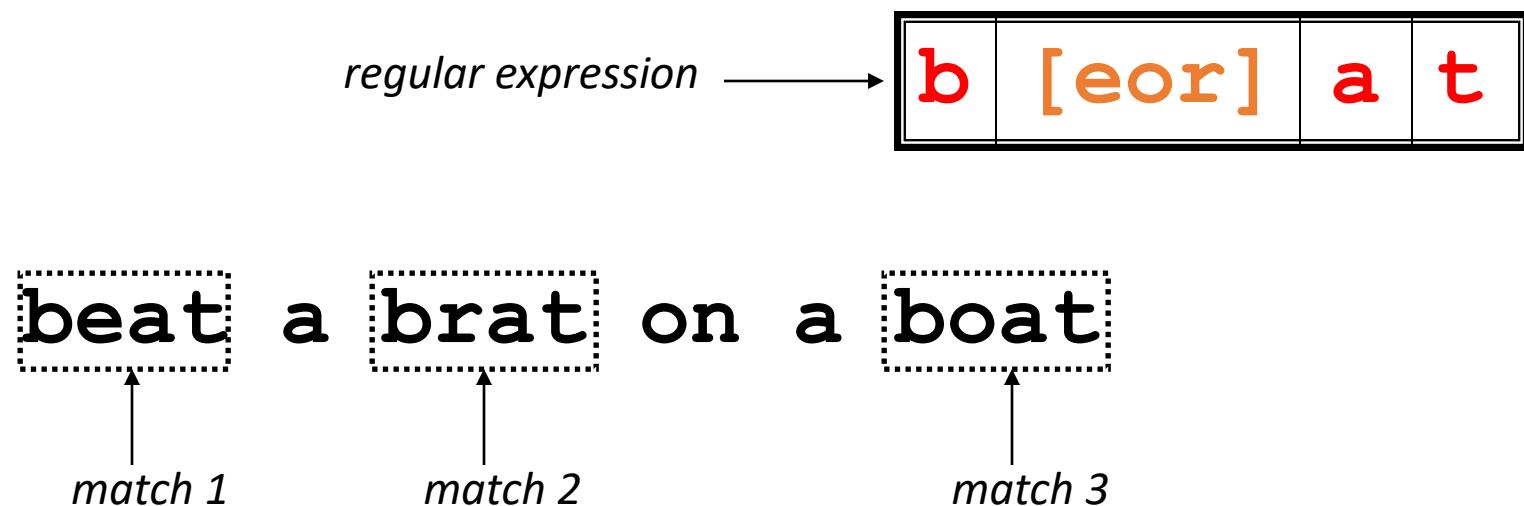
Regular Expressions

- The **.** regular expression can be used to match any character.



Character Classes

- Character classes **[]** can be used to match any specific set of characters.



Negated Character Classes

- Character classes can be negated with the `[^]` syntax.

regular expression →

b	[^eo]	a	t
---	-------	---	---

beat a brat on a boat

match

More About Character Classes

- **[aeiou]** will match any of the characters **a, e, i, o, or u**
- **[kK]orn** will match **korn** or **Korn**
- Ranges can also be specified in character classes
 - **[1-9]** is the same as **[123456789]**
 - **[abcde]** is equivalent to **[a-e]**
 - You can also combine multiple ranges
 - **[abcde123456789]** is equivalent to **[a-e1-9]**
 - Note that the **-** character has a special meaning in a character class **but only** if it is used within a range,
[-123] would match the characters **-**, **1**, **2**, or **3**

Named Character Classes

- Commonly used character classes can be referred to by name (*alpha*, *lower*, *upper*, *alnum*, *digit*, *punct*, *cntrl*)
- Syntax `[:name:]`
 - `[a-zA-Z]` `[:alpha:]`
 - `[a-zA-Z0-9]` `[:alnum:]`
 - `[45a-z]` `[45[:lower:]]`
- Important for portability across languages

Anchors

- Anchors are used to match at the beginning or end of a line (or both).
- **^** means beginning of the line
- **\$** means end of the line

regular expression



beat a brat on a boat

↑
match

regular expression



beat a brat on a **boat**

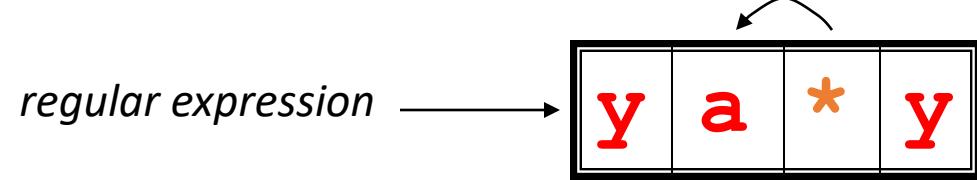
↑
match

^word\$

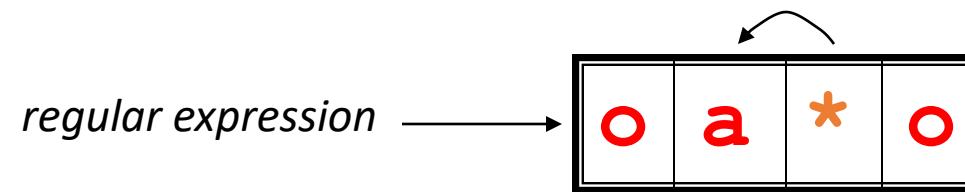
^\$

Repetition

- The ***** is used to define **zero or more** occurrences of the *single* regular expression preceding it.



I got mail, `yaaaaaaaaay!`



For me to `oop` on.

match

.*

Repetition Ranges

- Ranges can also be specified
 - $\{n, m\}$ notation can specify a range of repetitions for the immediately preceding regex
 - $\{n\}$ means exactly n occurrences
 - $\{n, \}$ means at least n occurrences
 - $\{n, m\}$ means at least n occurrences but no more than m occurrences
- Example:
 - $. \{0, \}$ same as $.^*$
 - $a \{2, \}$ same as aaa^*

Subexpressions

- If you want to group part of an expression so that $*$ applies to more than just the previous character, use $()$ notation
- Subexpressions are treated like a single character
 - a^* matches 0 or more occurrences of a
 - abc^* matches ab , abc , $abcc$, $abccc$, ...
 - $(abc)^*$ matches abc , $abcabc$, $abcababc$, ...
 - $(abc) \{2,3\}$ matches $abcabc$ or $abcababc$

Regular Expression Metacharacters Summary

Metacharacters	Description
* (Asterisk)	This matches zero or more occurrences of the previous character
+ (Plus)	This matches one or more occurrences of the previous character
?	This matches zero or one occurrence of the previous element
. (Dot)	This matches any one character
^	This matches the start of the line
\$	This matches the end of line
[...]	This matches any one character within a square bracket
[^...]	This matches any one character that is not within a square bracket
(Bar)	This matches either the left side or the right side element of
\{X\}	This matches exactly X occurrences of the previous element
\{X,\}	This matches X or more occurrences of the previous element
\{X,Y\}	This matches X to Y occurrences of the previous element
\(...\)	This groups all the elements
\<	This matches the empty string at the beginning of a word
\>	This matches the empty string at the end of a word
\	This disables the special meaning of the next character

A Website to validate your Regular Expression

- <https://regexr.com/>

Example1: Regular Expression in a Script



A script to extract the ***domain name*** from a given URL

```
#!/bin/bash
url=$1
regex_pattern="^https?:\/\/([^\/]*)"
if [[ "$url" =~ $ regex_pattern ]]; then
    domain=${BASH_REMATCH[1]}
    echo "Domain name: $domain"
else
    echo "Invalid URL"
fi
```

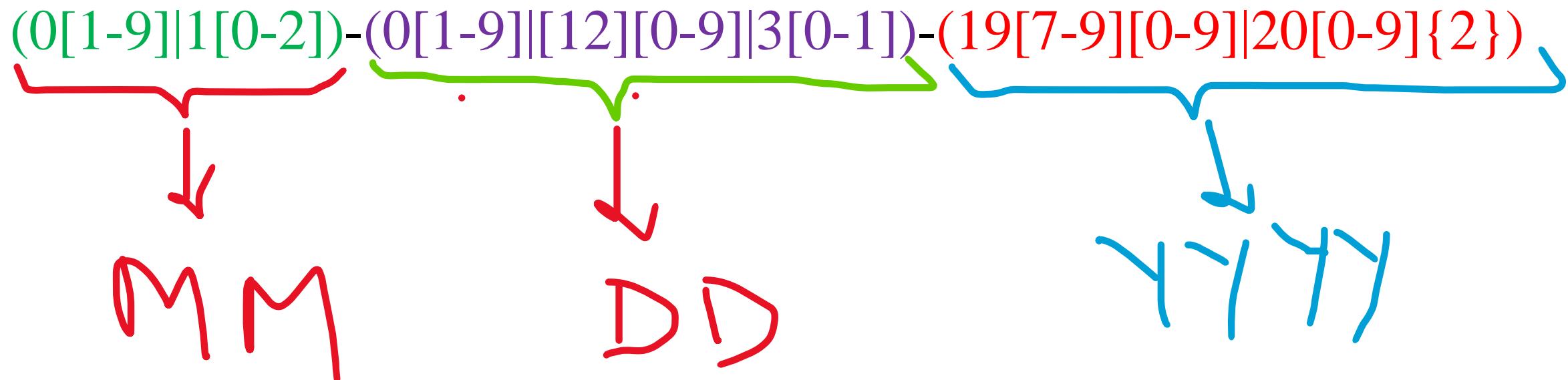
Example2: Regular Expression in a Script

- Script to check if a string is a valid email address

```
#!/bin/bash
email=$1
regex_pattern=" ^[[[:alnum:]]+@[[:alnum:]]+\.[[:alnum:]]+$ "
if [[ "$email" =~ $ regex_pattern ]]; then
    domain=${BASH_REMATCH[1]}
    echo "The email address $domain is a valid address"
else
    echo "Invalid email"
fi
```

Example3: Regular Expression in a Script

- Combining valid months, days, and years regex to form valid dates in MM-DD-YYYY format:



grep

- grep comes from the **ed** (Unix text editor) search command “**global regular expression print**” or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, *egrep* and *fgrep* that comprise the *grep* family
- *grep* is the answer to the moments where you know you want the file that contains a **specific phrase**, but you can’t remember its name

Family Differences

- **grep** - uses regular expressions for pattern matching
- **fgrep** - file grep, does not use regular expressions, only matches fixed strings but can get search strings from a file
- **egrep** - extended grep, uses a more powerful set of regular expressions but does not support backreferencing, generally the fastest member of the grep family

Syntax

- Regular expression concepts we have seen so far are common to **grep** and **egrep**.
- grep and egrep have different syntax
 - **grep**: BREs
 - **egrep**: EREs
- Major syntax differences:
 - **grep**: \(| and \)|, \{ and \}
 - **egrep**: (and), { and }

Protecting Regex Metacharacters

- Since many of the special characters used in regex also have special meaning to the shell, *it's a good idea to get in the habit of single quoting your regex*
 - This will protect any special characters from being operated on by the shell
 - If you habitually, do it, you won't have to worry about when it is necessary

Escaping Special Characters

- Even though we are single quoting our *regex* so the shell won't interpret the special characters, sometimes we still want to use an operator as itself
- To do this, we “escape” the character with a \ (backslash)
- Suppose we want to search for the character sequence ‘a*b*’
 - Unless we do something special, this will match zero or more ‘a’s followed by zero or more ‘b’s, *not what we want*
 - ‘a*b*’ will fix this - now the asterisks are treated as regular characters

Egrep: Alternation

- Regex also provides an alternation character | for matching one or another subexpression
 - **(T|Fl)an** will match ‘Tan’ or ‘Flan’
 - **^(From|Subject):** will match the From and Subject lines of a typical email message
 - It matches a beginning of line followed by either the characters ‘From’ or ‘Subject’ followed by a ‘:’
- Subexpressions are used to limit the scope of the alternation
 - **At(ten|nine)tion** then matches “Attention” or “Atninetion”, not “Atten” or “ninetion” as would happen without the parenthesis - **Atten|ninetion**

Egrep: Repetition Shorthands

- The ***** (star) has already been seen to specify zero or more occurrences of the immediately preceding character
- **+** (plus) means “one or more”
 - **abc+d** will match ‘abcd’, ‘abccd’, or ‘abcccccccd’ but will not match ‘abd’
 - Equivalent to **{1,}**

Egrep: Repetition Shorthands cont

- The ‘?’ (question mark) specifies an optional character, the single character that immediately precedes it
 - **July?** will match ‘Jul’ or ‘July’
 - Equivalent to **{0,1}**
 - Also equivalent to **(Jul|July)**
- The *****, **?**, and **+** are known as *quantifiers* because they specify the quantity of a match
- Quantifiers can also be used with subexpressions
 - **(a*c)+** will match ‘c’, ‘ac’, ‘aac’ or ‘aacaacac’ but will not match ‘a’ or a blank line

Grep: Backreferences

- Sometimes it is handy to be able to refer to a match that was made earlier in a regex
- This is done using *backreferences*
 - $\backslash n$ is the backreference specifier, where n is a number
- For example, *to find if the first word of a line is the same as the last:*
 - $^{\backslash([[:alpha:]]\{1,\})}.*\backslash1\$$
 - The $\backslash([[:alpha:]]\{1,\})$ matches 1 or more letters

Practical Regex Examples

- Variable names in C
 - `[a-zA-Z_][a-zA-Z_0-9]*`
- Dollar amount with optional cents
 - `\$[0-9]+(\.[0-9][0-9])?`
- Time of day
 - `(1[012]|1-9):[0-5][0-9] (am|pm)`
- HTML headers `<h1> <H1> <h2> ...`
 - `<[hH][1-4]>`

grep Family

- Syntax

grep [-hilnv] [-e expression] [filename]

egrep [-hilnv] [-e expression] [-f filename] [expression] [filename]

fgrep [-hilnxv] [-e string] [-f filename] [string] [filename]

- **-h** Do not display filenames
- **-i** Ignore case
- **-l** List only filenames containing matching lines
- **-n** Precede each matching line with its line number
- **-v** Negate matches
- **-x** Match whole line only (*fgrep* only)
- **-e expression** Specify expression as option
- **-f filename** Take the regular expression (*egrep*) or a list of strings (*fgrep*) from *filename*

Fun with the Dictionary

- `/usr/dict/words` contains about 25,000 words
 - `egrep hh /usr/dict/words`
 - beachhead
 - highhanded
 - withheld
 - withhold
- **egrep** as a simple spelling checker: Specify plausible alternatives you know
`egrep "n(ie|ei)ther" /usr/dict/words`
neither
- How many words have 3 a's one letter apart?
 - `egrep a.a.a /usr/dict/words | wc -l`
 - 54
 - `egrep u.u.u /usr/dict/words`
 - cumulus



Thanks
Q & A

Processes in Linux: An Overview of Linux Process Management

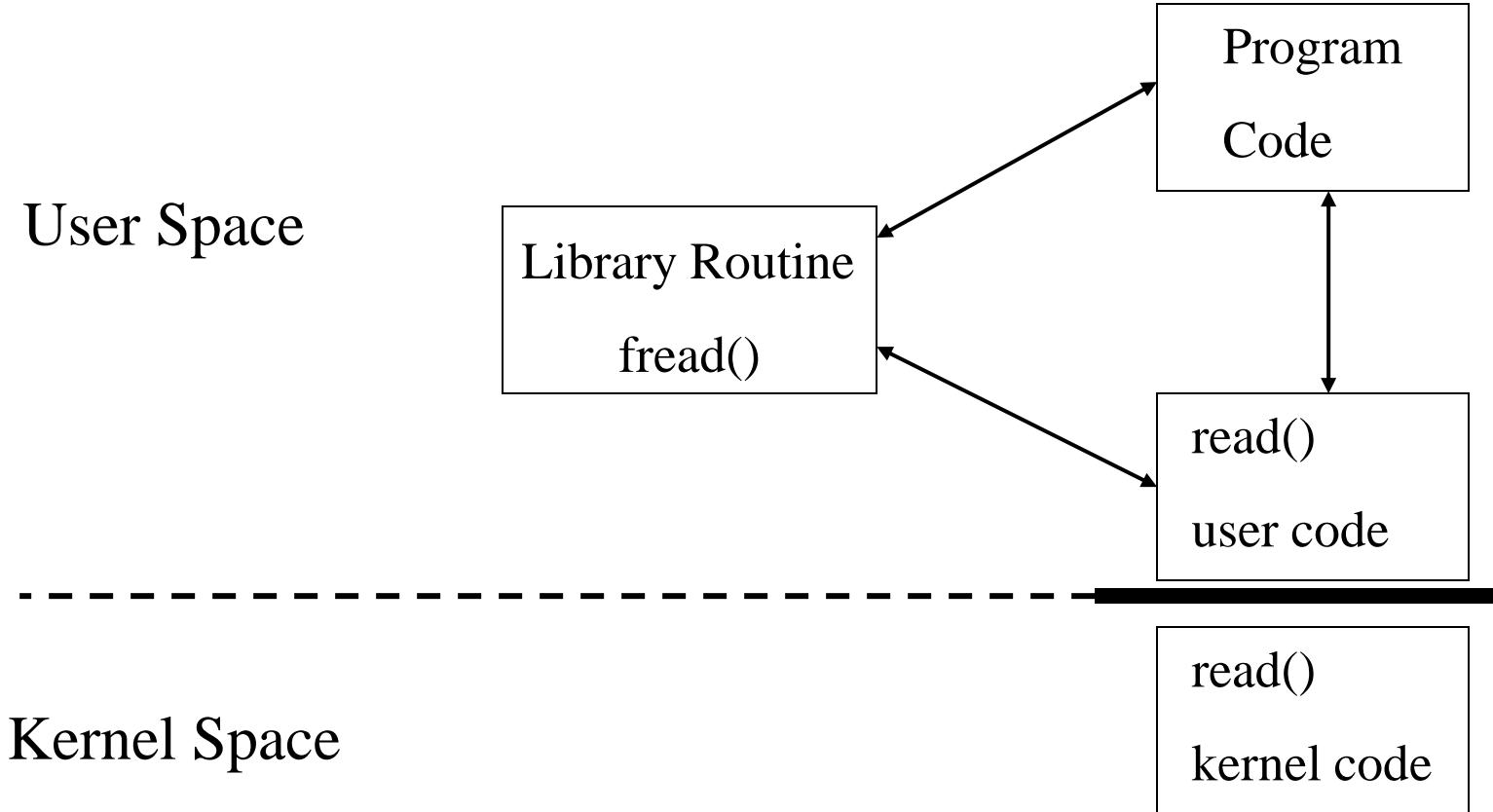


Dr. Vimal Baghel
Assistant Professor
SCSET, BU

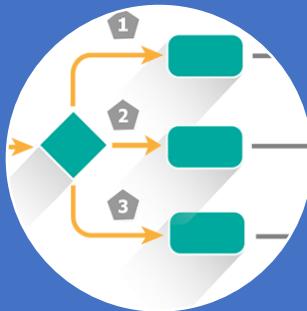
Outline

- Introduction to Linux Processes
- Processes Resources
- Q&A

User and System Space



What is a Process?:

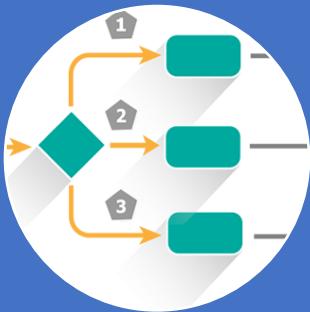


A process is a program in execution state.

It is a dynamic Entity in OS.



The Process



The process includes

- the PC,
- CPU's registers,
- the process stacks containing temporary data such as routine parameters, return addresses and saved variables.



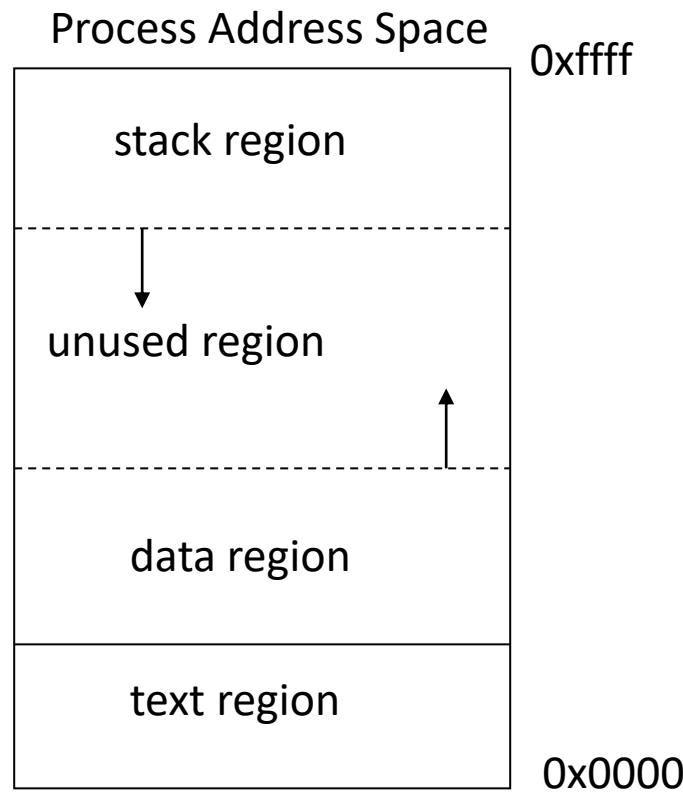
Process Description

- A process is completely defined by
 - the CPU registers
 - program counter, stack pointer, control, general purpose, etc.
 - memory regions
 - user and kernel stacks
 - code
 - heap
- To start and stop a program, all of the above must be saved or restored
 - CPU registers must be explicitly saved/restored
 - memory regions are implicitly saved/restored

Memory Regions of a Process

- Every process has 3 main regions
 - text area
 - stores the actual program code
 - static in size (usually)
 - stack area
 - stores local data
 - function parameters, local variables, return address
 - data area (heap)
 - stores program data not on the stack
 - grows dynamically per user requests

Memory Regions of a Process



Note: the stack usually grows down while the data region grows upward – the area in between is free

User vs. Kernel Stack

- Each process gets its own user stack
 - resides in user space
 - manipulated by the process itself
- In Linux, each process gets its own kernel stack
 - resides in kernel space
 - manipulated by the OS
 - used by the OS to handle system calls and interrupts that occur while the process is running

User Stack

Function: <i>printAvg</i> Return: <i>check</i> call inst Param: avg Local: none
Function: <i>check</i> Return: <i>main</i> call inst Param: grade Local: hi, low, avg
Method: <i>main</i> Return: halt Param: command line Local: grade[5], num

Kernel Stack

Function: <i>calcSector</i> Return: <i>read</i> call inst Param: avg Local: sector
Function: <i>read</i> Return: <i>user program</i> Param: block Local: sector
User program counter User stack pointer

Process Descriptor

- OS data structure that holds all necessary information for a process
 - process state
 - CPU registers
 - memory regions
 - pointers for lists (queues)
 - etc.

Process Descriptor

pointer	state
process ID number	
program counter	
registers	
memory regions	
list of open files	
.	
.	
.	

Process Descriptor

- Pointer
 - used to maintain queues that are linked lists
- State
 - current state the process is in (i.e. running)
- Process ID Number
 - identifies the current process
- Program Counter
 - needed to restart a process from where it was interrupted

Process Descriptor

- Registers
 - completely define state of process on a CPU
- Memory Limits
 - define the range of legal addresses for a process
- List of Open Files
 - pretty self explanatory

Background & Foreground Processes

- A foreground process is any process which is not continuously running and it waiting on something like user input
- A background process is something that is continually running and does not require any additional input
- Can someone name examples of each?

Moving a Process to the Background

- When executing commands on the command line, there is usually some output that is displayed on the terminal
- If you move a process to the background, the output will not be shown

Background Process Example

- Usually, when you download a file from the command line, the status is displayed on the terminal
- To move a process to the background all we will do is add an ampersand (&) at the end of the command
 - `wget http://releases.ubuntu.com/24.04.2/ubuntu-24.04.2-desktop-amd64.iso ga=2.142658160.410030815.1551071806-1676866732.1550780350 &`
- Now this will be moved to the background

Moving back to the Foreground

- To move a process back to the foreground, use the following steps:
 - Use the **jobs** command to identify the job number of the background process
 - Then use the **fg** command to bring it back with the following syntax
 - **fg [job number]**

Different Types of Processes

- There are four types of processes:
 - Running: current process that is being executed in the operating system
 - Waiting: process which is waiting for system resources to run
 - Stopped: process that is not running
 - Zombie: process whose parent processes has ended, but the child process is still in the process table

Viewing Processes

- Two commands you can use to view the process from the command line: **ps** and **top**
- To view all the processes with **ps**, use **ps -ef**

```
ubuntu@ubuntu-VirtualBox:~/labs/lab6$ ps -ef
UID      PID  PPID  C STIME TTY      TIME CMD
root      1      0  0 10:27 ?        00:00:01 /sbin/init splash
root      2      0  0 10:27 ?        00:00:00 [kthreadd]
root      4      2  0 10:27 ?        00:00:00 [kworker/0:0H]
root      6      2  0 10:27 ?        00:00:00 [mm_percpu_wq]
root      7      2  0 10:27 ?        00:00:00 [ksoftirqd/0]
root      8      2  0 10:27 ?        00:00:00 [rcu_sched]
root      9      2  0 10:27 ?        00:00:00 [rcu_bh]
root     10      2  0 10:27 ?        00:00:00 [migration/0]
root     11      2  0 10:27 ?        00:00:00 [watchdog/0]
root     12      2  0 10:27 ?        00:00:00 [cpuhp/0]
root     13      2  0 10:27 ?        00:00:00 [kdevtmpfs]
root     14      2  0 10:27 ?        00:00:00 [netns]
root     15      2  0 10:27 ?        00:00:00 [rcu_tasks_kthre]
root     16      2  0 10:27 ?        00:00:00 [kaudittd]
```

ps -ef

```
top - 10:48:42 up 21 min,  1 user,  load average: 0.03, 0.09, 0.20
Tasks: 212 total,   1 running, 180 sleeping,   0 stopped,   0 zombie
%Cpu(s): 17.0 us,  4.1 sy,  0.0 ni, 75.5 id,  3.4 wa,  0.0 hi,  0.0 si,  0.0 st
KiB Mem : 8168488 total, 5414240 free, 1656284 used, 1097964 buff/cache
KiB Swap: 1459804 total, 1459804 free,          0 used. 6165520 avail Mem

 PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
1294 ubuntu    20   0 2933144 202880 80452 S 13.5  2.5  0:20.53 gnome-shell
1122 ubuntu    20   0 501344 122496 65504 S  3.0  1.5  0:08.06 Xorg
1673 ubuntu    20   0 868420 37936 27812 S  2.0  0.5  0:01.47 gnome-terminal-
915 gdm       20   0 2903920 129028 76872 S  0.7  1.6  0:03.48 gnome-shell
1316 ubuntu    9 -11 1959040 12456 8944 S  0.7  0.2  0:00.08 pulseaudio
1325 ubuntu    20   0 361564 7892 6416 S  0.7  0.1  0:00.68 ibus-daemon
1453 ubuntu    20   0 1130700 24192 19160 S  0.7  0.3  0:00.08 gsd-media-keys
1869 ubuntu    20   0 2124492 529224 172348 S  0.7  6.5  1:00.94 Web Content
  870 root      20   0 255476 2748 2376 S  0.3  0.0  0:00.30 VBoxService
  922 root      20   0 322300 8448 7328 S  0.3  0.1  0:00.09 upowerd
1959 ubuntu    20   0 1518980 104680 80468 S  0.3  1.3  0:03.65 WebExtensions
    1 root      20   0 159948  9244 6764 S  0.0  0.1  0:01.54 systemd
    2 root      20   0      0    0    0 S  0.0  0.0  0:00.00 kthreadd
```

top

Ending a Process In Linux

- Sometimes we need to end a program or process from the command line.
- Use the following steps:
 1. Locate the process id [PID] of the process/program you want to kill
 2. Use the **kill** command with the following syntax: **kill [PID]**
 3. If the process is still running, do the following: **kill -9 [PID]**
 4. The -9 is a SIGKILL signal telling the process to terminate immediately

Ending All Process

- We can use the **killall** command to kill multiple processes at the same time
- Syntax: **killall [options] PIDs**
- Or you can use **pkill -u [username]** to kill all processes started by [username]

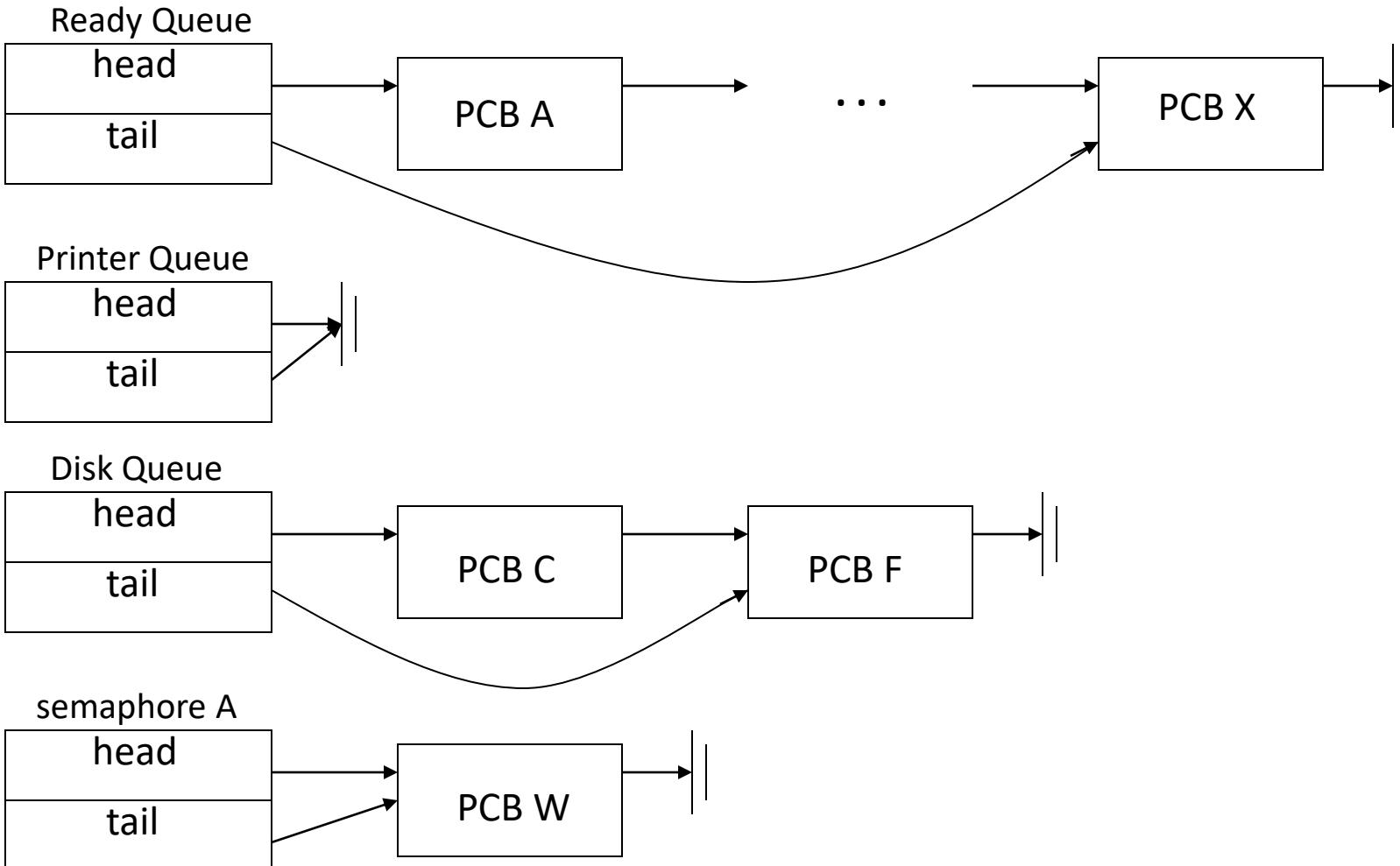
Process States

- 5 generic states for processes
 - new
 - ready
 - running
 - waiting
 - terminated (zombie)
- Many OS's combine ready and running into *runnable* state

Process Queues

- Every process belongs to some queue
 - implemented as linked list
 - use the pointer field in the process descriptor
- Ready queue
 - list of jobs that are ready to run
- Waiting queues
 - any job that is not ready to run is waiting on some event
 - I/O, semaphores, communication, etc.
 - each of these events gets its own queue

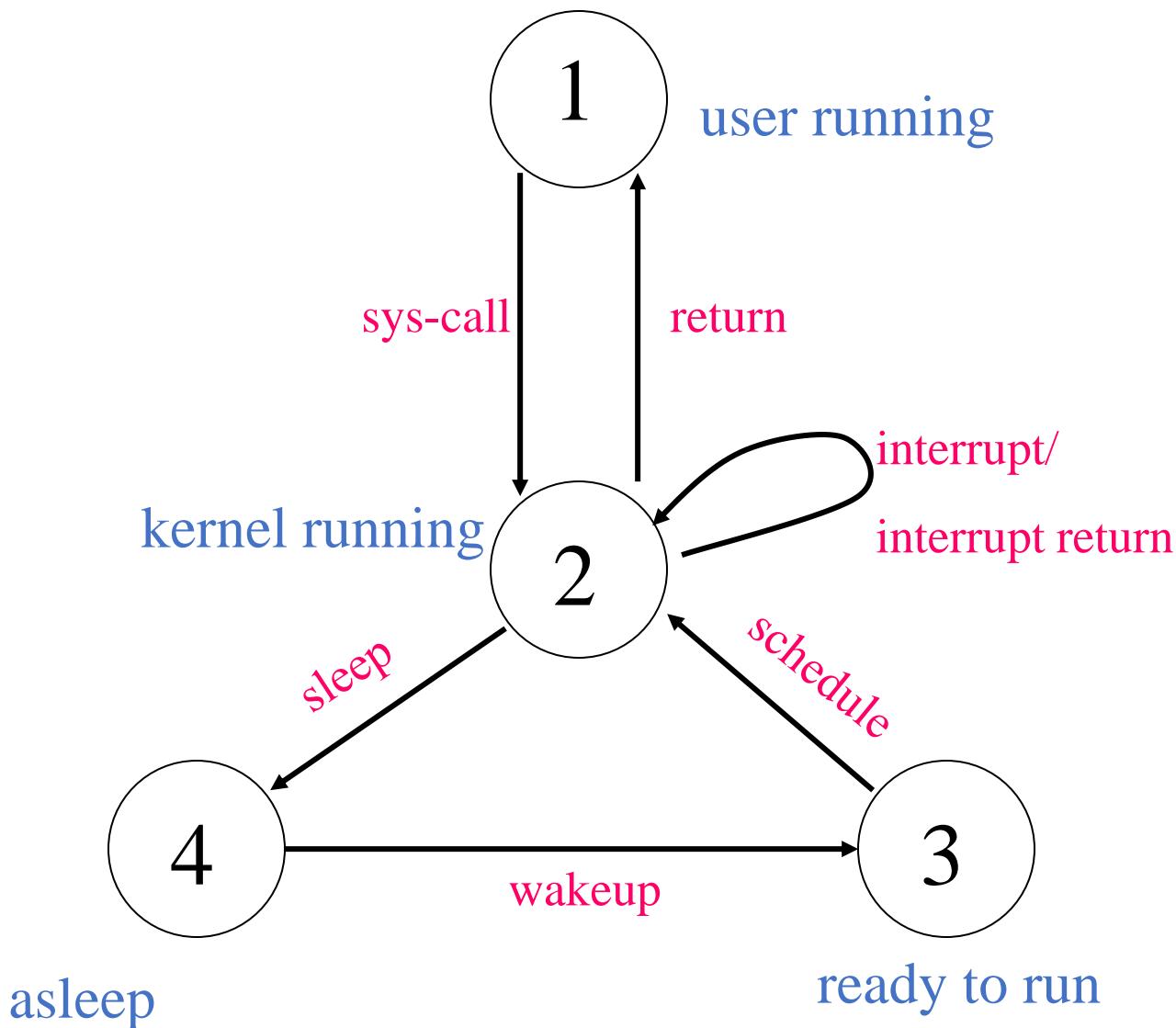
Process Queues



Process States Transitions

- A process changes *state* according to its circumstances
 - Running or ready
 - Waiting for an event or for a resource, (Interruptible or Uninterruptible)
 - Stopped by receiving a signal,
 - zombie – a halted process

Process State Transitions



How do Processes Actually Work?

- In the Linux, processes are created by a method called ***“forking”***
 - Forking is done when the OS duplicated a process
 - The original process is called the parent process
 - And the new process is the child process

Forkbombing Example

- What is forkbombing?
 - It is when you spawn multiple processes which leads to lack of system resources and other very bad things
- I will forkbomb the server to demonstrate the **pkill** command
- DO NOT TRY THIS (the Systems department will not be happy with you)

Creating Processes

- Parent process creates a child process
 - results in a *tree*
- Execution options
 - parent and child execute concurrently
 - parent waits for child to terminate
- Address space options
 - child gets its own memory
 - child gets a subset of parents memory

Creating Processes in Unix

```
void main() {  
    int pid;  
    pid = fork();  
    if(pid == 0) { // child process - start a new program  
        execlp("/bin/ls", "/home/mattmcc/", NULL);  
    }  
    else { // parent process - wait for child  
        wait(NULL);  
        exit(0);  
    }  
}
```

Creating Processes in Unix

- *fork()* system call
 - creates **exact** copy of parent
 - only thing different is return address
 - child gets 0
 - parent gets child ID
 - child may be a *heavyweight process*
 - has its own address space
 - runs concurrently with parent
 - child may be a *lightweight process*
 - shares address space with parent (and siblings)
 - still has its own execution context and runs concurrently with parent

Creating Processes in Linux

- *exec()* system call starts new program
 - needed to get child to do something new
 - remember, child is exact copy of parent
- *wait()* system call forces parent to suspend until child completes
- *exit()* system call terminates a process
 - places it into zombie state

Destroying a Process

- Multiple ways for a process to get destroyed
 - process issues and *exit()* call
 - parent process issues a *kill()* call
 - process receives a terminate signal
 - did something illegal
- On death:
 - reclaim all of process's memory regions
 - make process unrunnable
 - put the process in the *zombie state*
 - ***However, do not remove its process descriptor from the list of processes***

Zombie State

- Why keep process descriptor around?
 - parent may be waiting for child to terminate
 - via the *wait()* system call
 - parent needs to get the exit code of the child
 - this information is stored in the descriptor
 - if descriptor was destroyed immediately, this information could not be retrieved
 - after getting this information, the process descriptor can be removed
 - no more remnants of the process

init Process

- This is one of the first processes spawned by the OS
 - is an ancestor to all other processes
- Runs in the background and does clean-up
 - looks for zombie's whose parents have not issued a *wait()*
 - removes them from the system
 - looks for processes whose parents have died
 - adopts them as its own

Process Representation: task_struct

- Process resources: \$ ls -l /proc/self/
- Each Linux process is represented by a data structure: *task_struct* (*A PCB/TCB in Linux*)
- The task vector is an array of pointers to every *task_struct* in the system
- The max number of processes are limited by the size of the task vector, default is 512.

Process Representation: `task_struct`

- As the new processes are created, a new *task_struct* is allocated from system memory and added into the *task* vector.
- To make it easy to find, the current running process is pointed to by the *current* pointer
- Linux also supports real time process.

Processes and its Resources

- A process is an OS abstraction that groups together multiple resources:

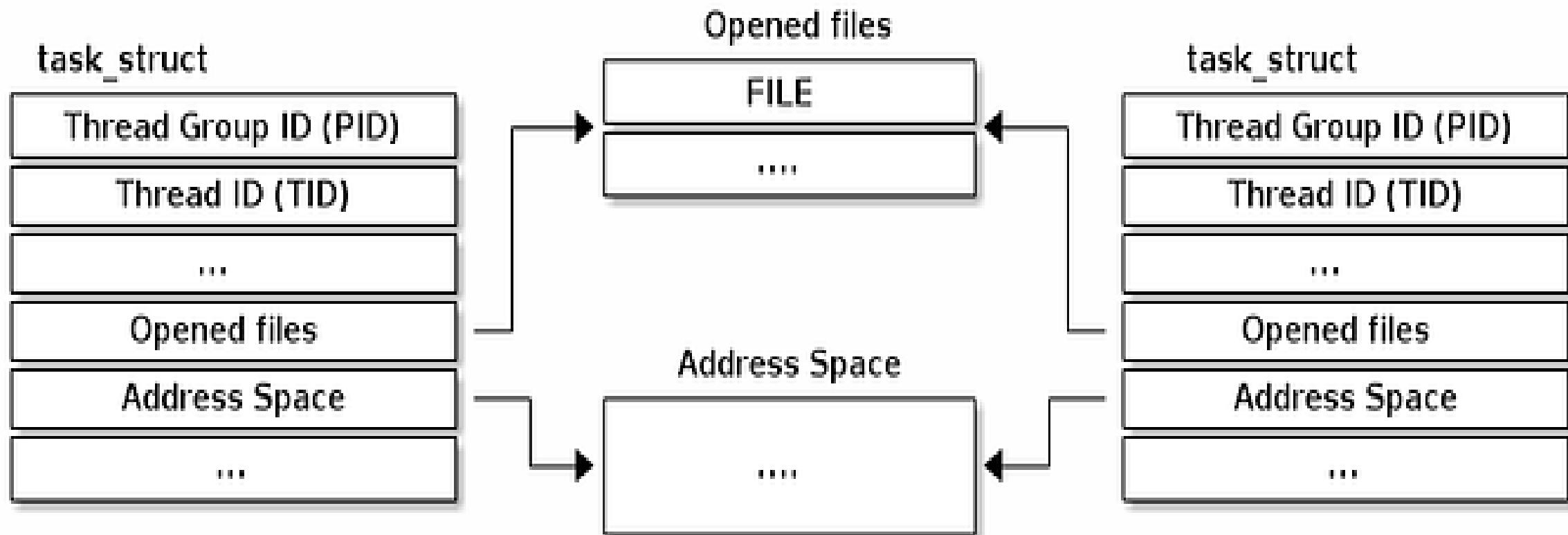
- | | | |
|--------------------|-----------------------|--|
| ✓ An address space | ✓ One or more threads | ✓ Shared memory regions |
| ✓ Opened files | | ✓ Timers |
| ✓ Sockets | | ✓ Signal handlers |
| ✓ Semaphores | | ✓ Many other resources and
status information |

All this information is grouped in the Process Control Group (PCB). In Linux this is **struct task_struct**.

Overview of process resources

- A summary of the resources a process has can be obtained from the `/proc/<pid>` directory, where `<pid>` is the process id for the process we want to look at.
- `$ ls -l /proc/pid`
- `$ ls -l /proc/self`

2 processes opening the same file and the relationship between the 2 different “task_struct”



Process representation in Linux

- Process is represented by a large structure **task_struct**.
- It contains:
 - The necessary data to represent the process
 - data for accounting and to maintain relationships with other processes (parents and children)

Managing the task Array

- The **task** array is updated every time a process is created or destroyed.
- A separate list (headed by **tarray_freelist**) keeps track of free elements in the **task** array.
 - When a process is destroyed its entry in the **task** array is added to the head of the freelist.

task_struct

- Although, **task_struct** is quite large and complex data structure, but its fields can be divided into several functional areas:
 - **State**
 - **Scheduling Information**
 - **Identifiers**
 - **IPC**
 - **Links**
 - **Times and Timers**
 - **File system**
 - **Virtual memory**
 - **Processor Specific Context**

Scheduling Info

- The scheduler needs this information in order to fairly decide which process in the system most deserves to run.

Process Identifiers

- Every process in the system has a process identifier (PID).
- The PID is not an index into the `task` vector, it is simply a number.
- Each process also has User and group identifiers, these are used to control this processes access to the files and devices in the system,

IPC

- Linux supports the classic Unix IPC mechanisms:
 - signals,
 - pipes
 - semaphores
 - the System V IPC mechanisms of shared memory, semaphores and message queues.

Process Links

- No process in Linux is independent
- Every process in the system, except the initial process has a parent process.
- New processes are not created, **they are copied, or rather *cloned* from previous processes.**
- Every *task_struct* representing a process keeps pointers to its parent process and to its siblings (those processes with the same parent process) as well as to its own child processes.

Process Links

- You can see the family relationship between the running processes in a Linux system using the `ps tree` command:



```
init(1)-+-crond(98)
|-emacs(387)
|-gpm(146)
|-inetd(110)
|-kerneld(18)
|-kflushd(2)
|-klogd(87)
|-kswappd(3)
|-login(160)---bash(192)---emacs(225)
|-lpd(121)
|-mingetty(161)
|-mingetty(162)
|-mingetty(163)
|-mingetty(164)
|-login(403)---bash(404)---ps tree(594)
|-sendmail(134)
|-syslogd(78)
`-update(166)
```

Process Links

- Additionally, all the processes in the system are held in a doubly linked list whose root is the `init processes task_struct` data structure.
- This list allows the Linux kernel to look at every process in the system.
- It needs to do this to provide support for commands such as ps or kill.

Times and Timers

- The kernel keeps track of a processes creation time as well as the CPU time that it consumes during its lifetime.
- Each clock tick, the kernel updates the amount of time in ^{jiffies} that the current process has spent in system and in user mode. Linux also supports process specific *interval* timers, processes can use system calls to set up timers to send signals to themselves when the timers expire.
- These timers can be single-shot or periodic timers.

File system

- Processes can open and close files as they wish and the processes `task_struct` contains pointers to descriptors for each open file as well as pointers to two VFS inodes.
- Each VFS inode uniquely describes a file or directory within a file system and also provides a uniform interface to the underlying file systems.
- The first is to the root of the process (its home directory) and the second is to its current or `pwd` directory.
- These two VFS inodes have their `count` fields incremented to show that one or more processes are referencing them.
- This is why you cannot delete the directory that a process has as its `pwd` directory set to, or for that matter one of its sub-directories.

Virtual memory

- Most processes have some virtual memory (kernel threads and daemons do not) and the Linux kernel must track how that virtual memory is mapped onto the system's physical memory.

Processor Specific Context

- A process could be thought of as the sum total of the system's current state.
- Whenever a process is running it is using the processor's registers, stacks and so on. This is the processes context and, when a process is suspended, all of that CPU specific context must be saved in the **task_struct** for the process.
- When a process is restarted by the scheduler its context is restored from here.



Thanks
Q & A

Fork & zombies

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

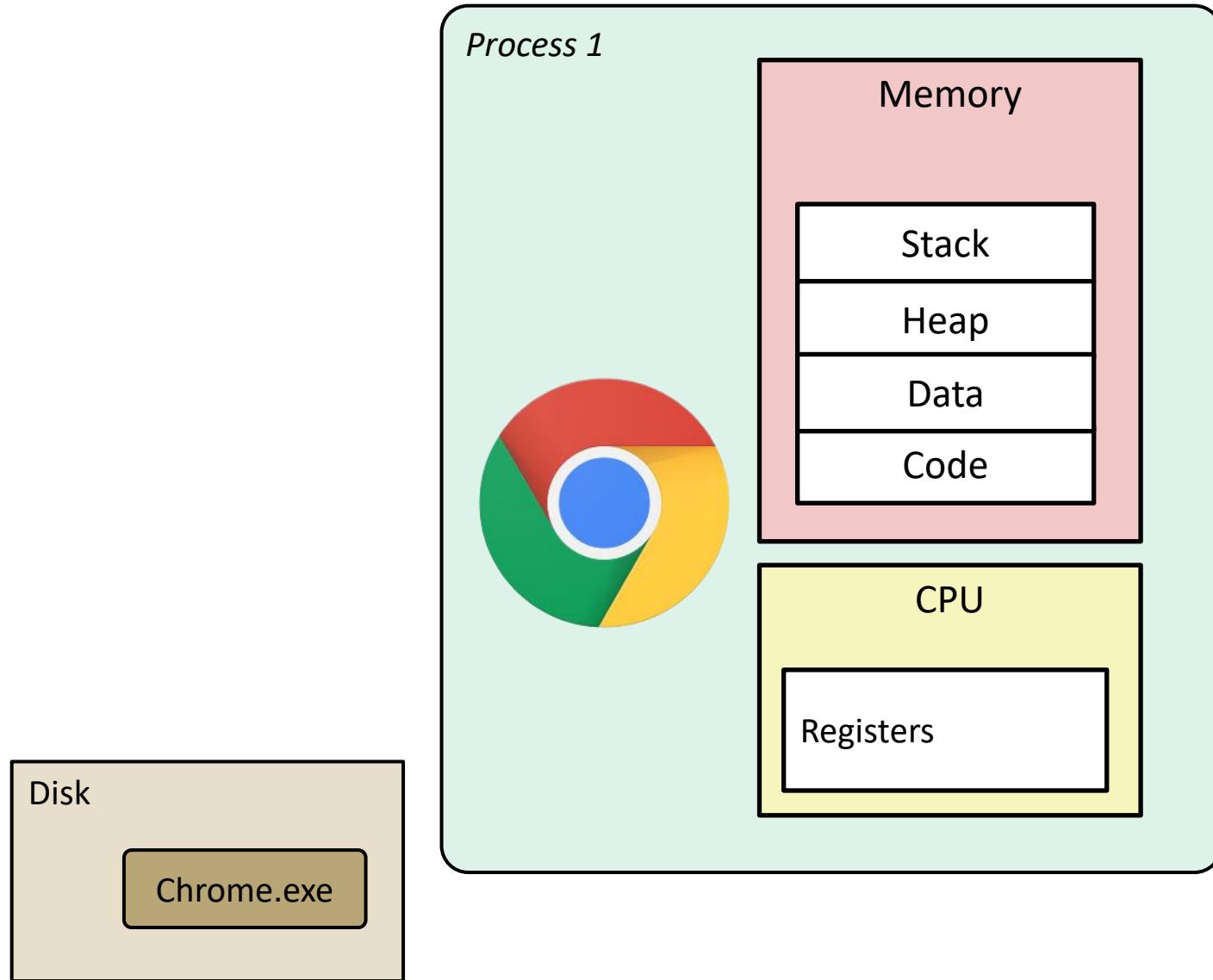


Processes

- **Processes and context switching**
- Creating new processes
 - `fork()`, `exec*`(), and `wait()`
- Zombies

What is a process?

It's an *illusion!*



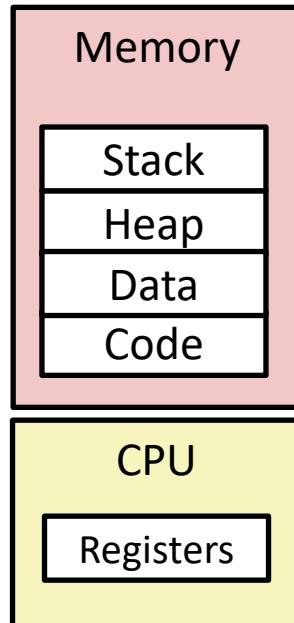
What is a process?

- Another *abstraction* in our computer system
 - Provided by the OS
 - OS uses a data structure to represent each process
 - Maintains the **interface** between the program and the underlying hardware (CPU + memory)
- What do *processes* have to do with *exceptional control flow*?
 - Exceptional control flow is the *mechanism* the OS uses to enable **multiple processes** to run on the same system
- What is the difference between:
 - A processor? A program? A process?

hardware the "blueprint" an instance

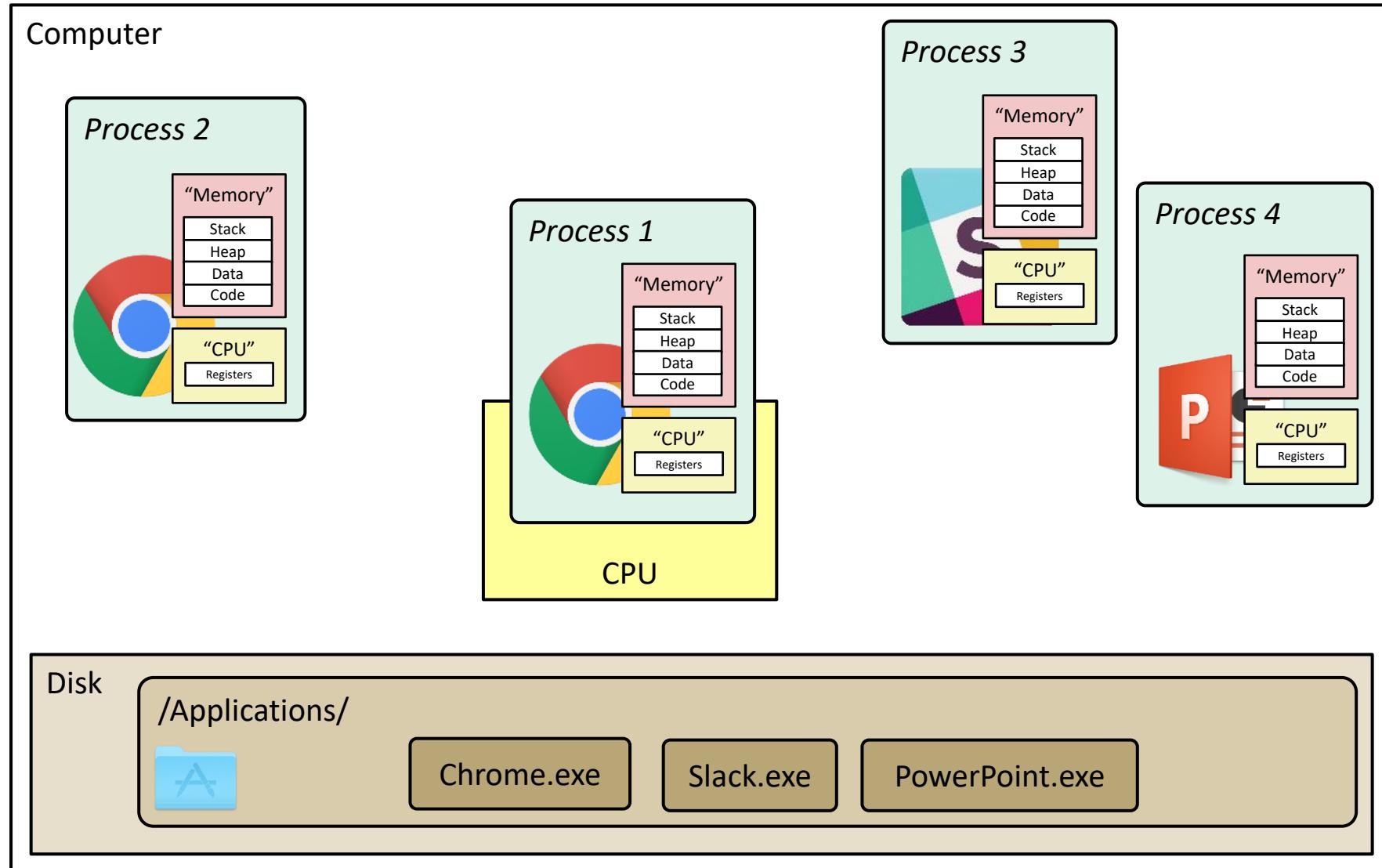
Processes

- A **process** is an instance of a running program
 - One of the most profound ideas in computer science
 - Not the same as “program” or “processor”
- Process provides each program with two key abstractions:
 - *Logical control flow*
 - Each program seems to have exclusive use of the CPU
 - Provided by kernel mechanism called **context switching**
 - *Private address space*
 - Each program seems to have exclusive use of main memory
 - Provided by kernel mechanism called **virtual memory**



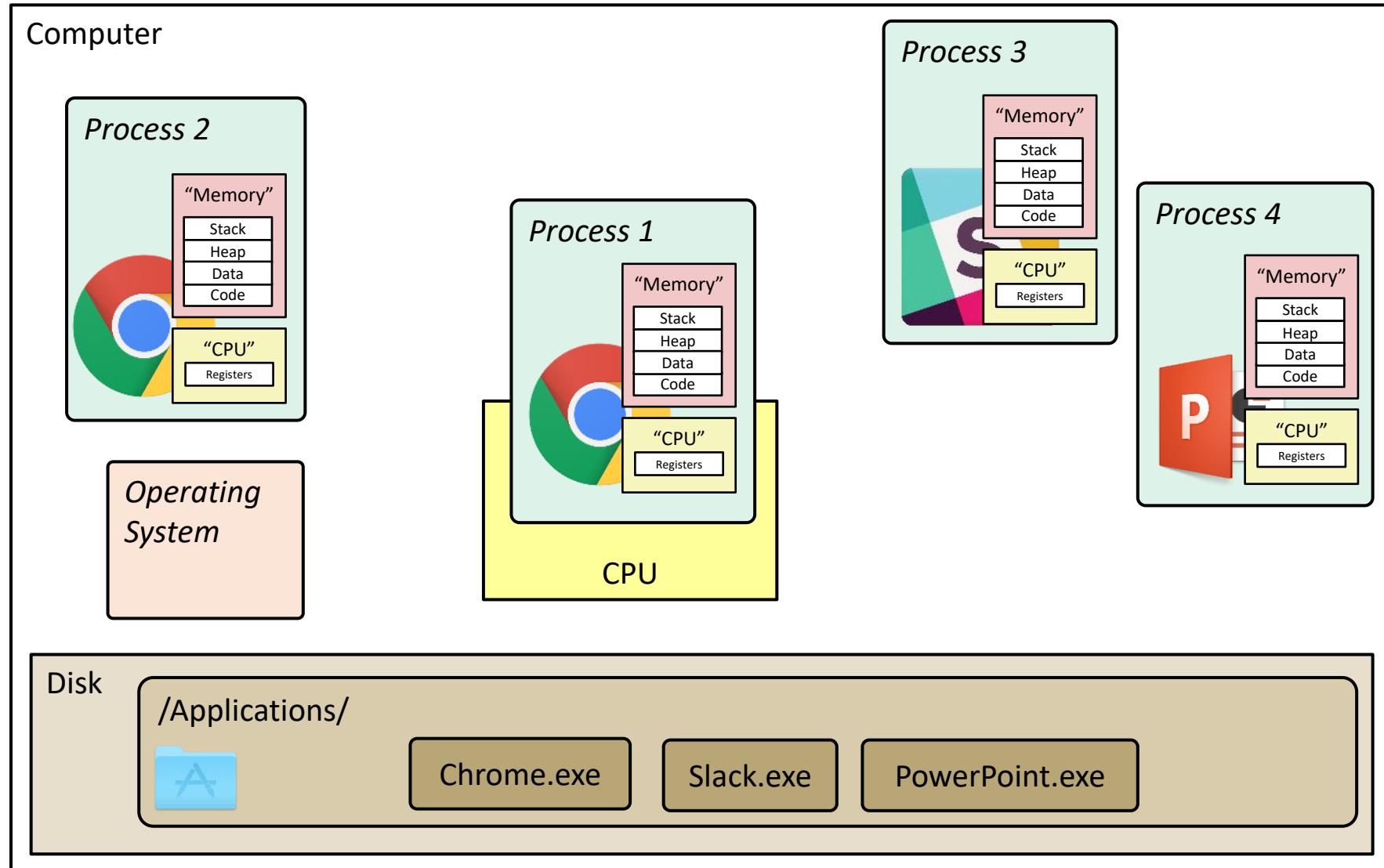
What is a process?

It's an *illusion!*



What is a process?

It's an *illusion!*

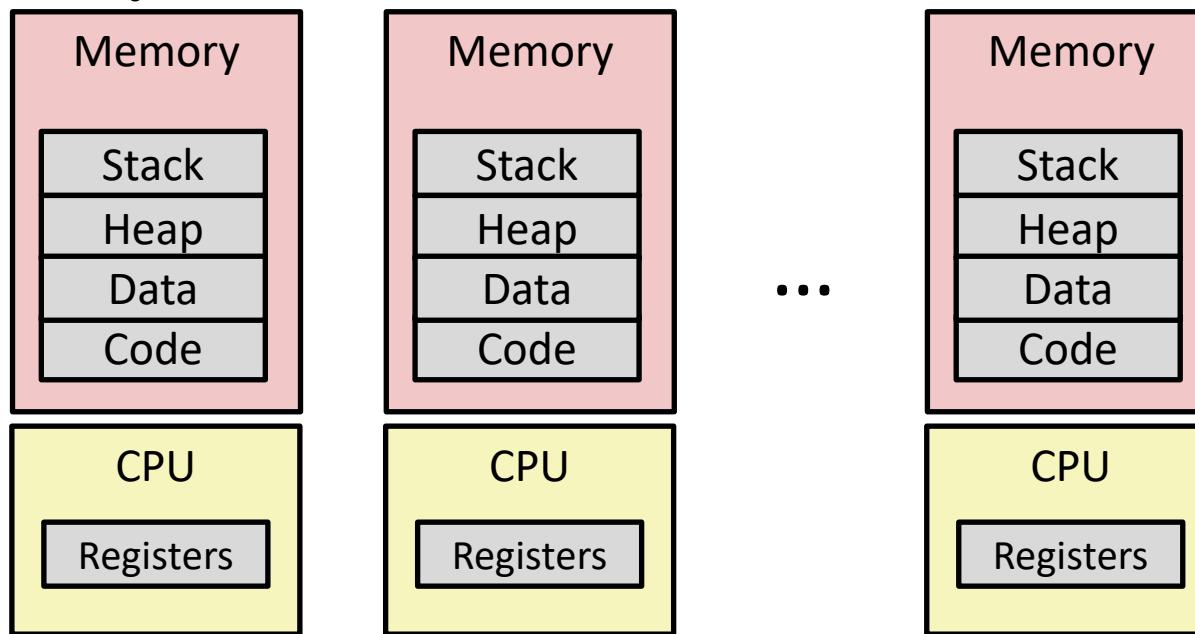


Multiprocessing on Uniprocessor: The Illusion

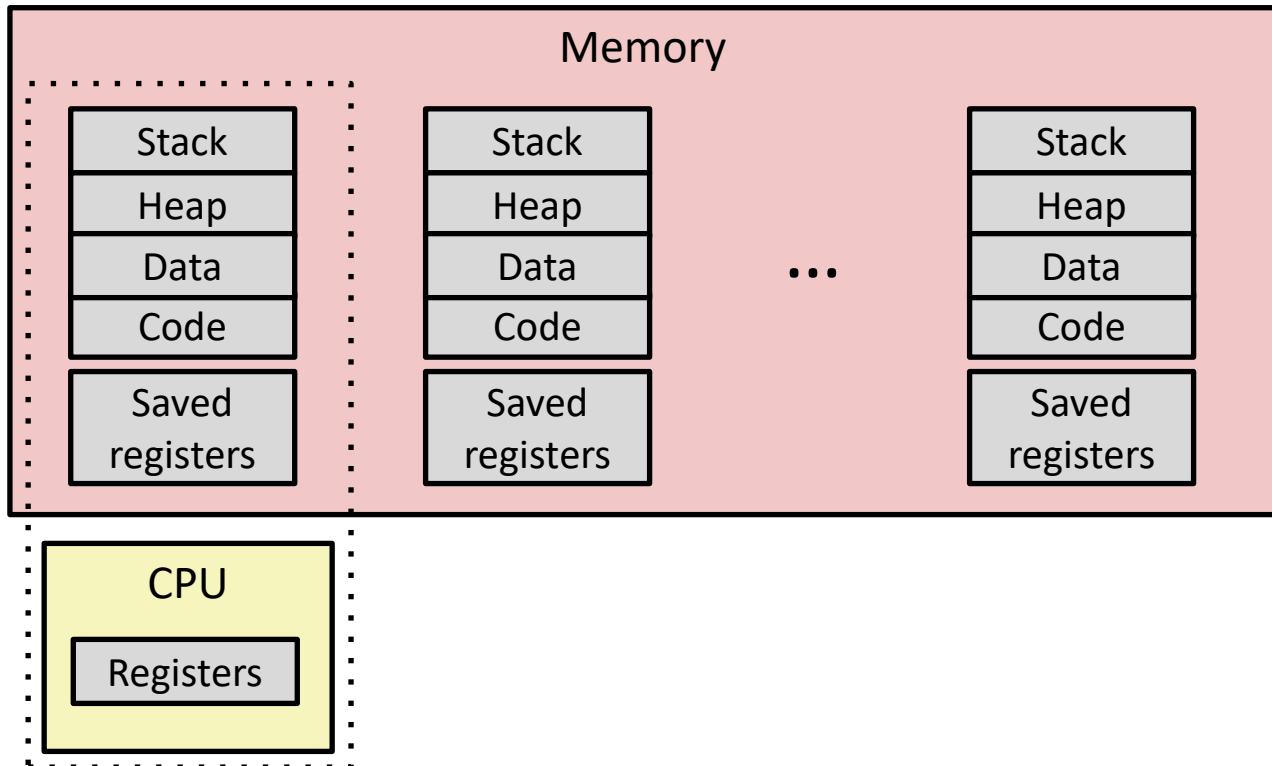
- While true multiprocessing is not possible on a uniprocessor, there are techniques that can simulate the multiprocessing:
 - ✓ **Process Scheduling:** The OS can rapidly switch between processes, giving the illusion of simultaneous execution. This is known as **context switching**.
 - ✓ **Asynchronous I/O:** When a process waits for I/O (reading from a disk), the operating system can switch to another process, making better use of CPU time.
 - ✓ **Multithreading:** Within a single process, multiple threads can be created to execute different tasks concurrently. This can improve responsiveness and resource utilization, even on a uniprocessor.
 - ✓ **Distributed Computing:** While not strictly multiprocessing, distributed computing involves using multiple computers connected by a network to work on a single task. This can simulate the behavior of a multiprocessor system.
- Computer runs many processes simultaneously

- Applications for one or more users
 - Web browsers, email clients, editors, ...
- Background tasks
 - Monitoring network & I/O devices

} user-level
} mostly kernel/OS - level

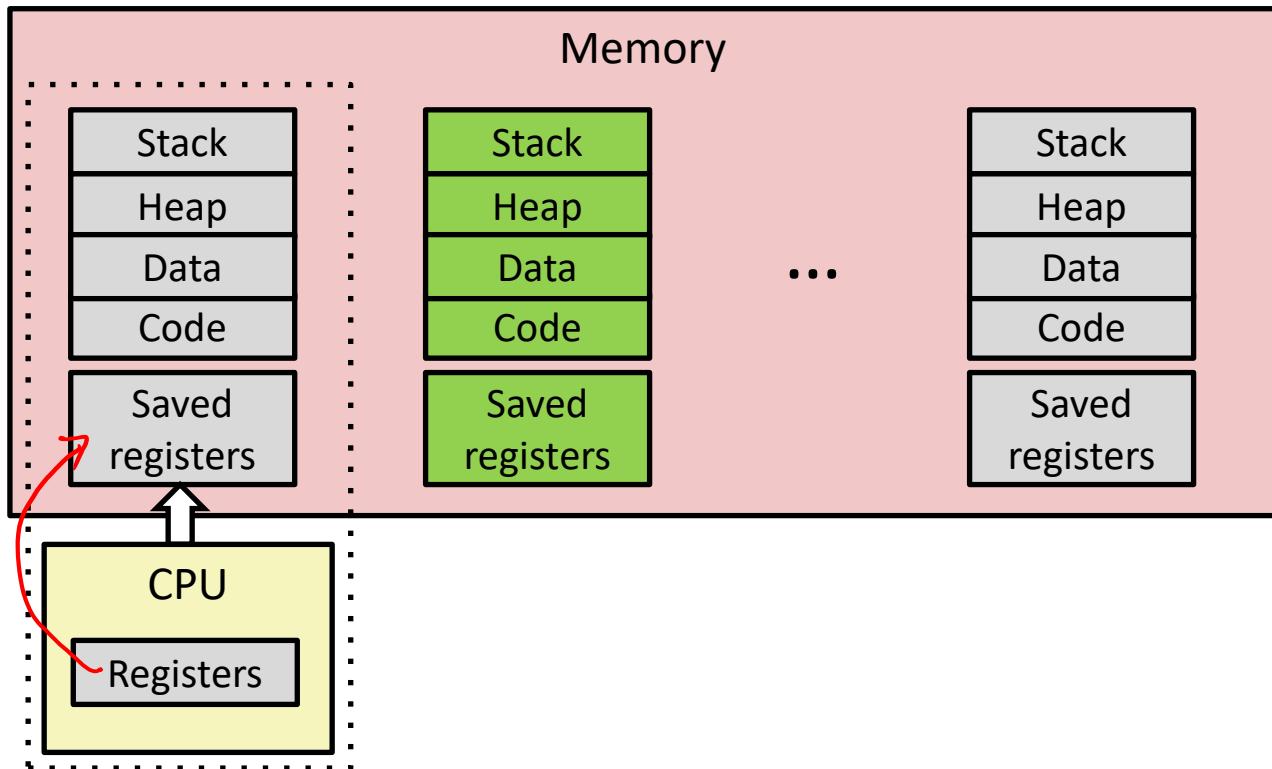


Multiprocessing: The Reality



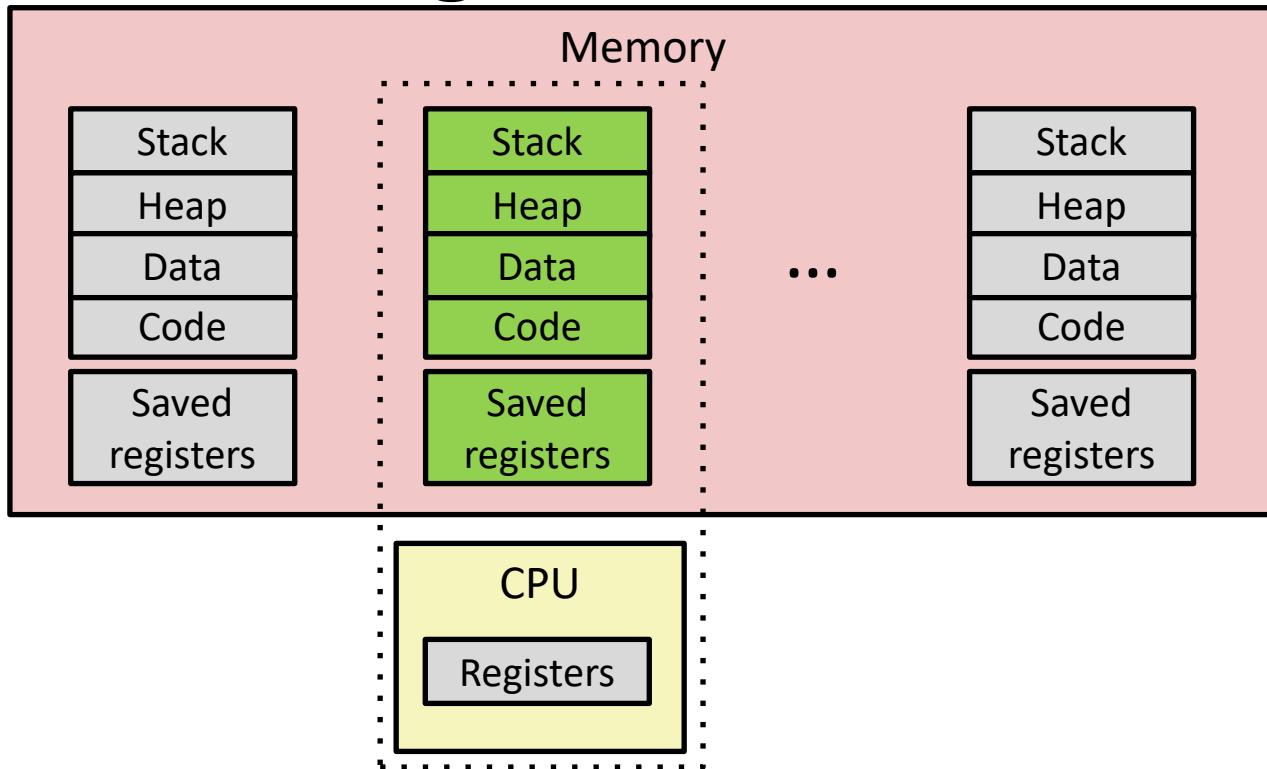
- Single processor executes multiple processes *concurrently*
 - Process executions interleaved, CPU runs *one at a time*
 - Address spaces managed by virtual memory system (later in course)
 - *Execution context* (register values, stack, ...) for other processes saved in memory

Multiprocessing



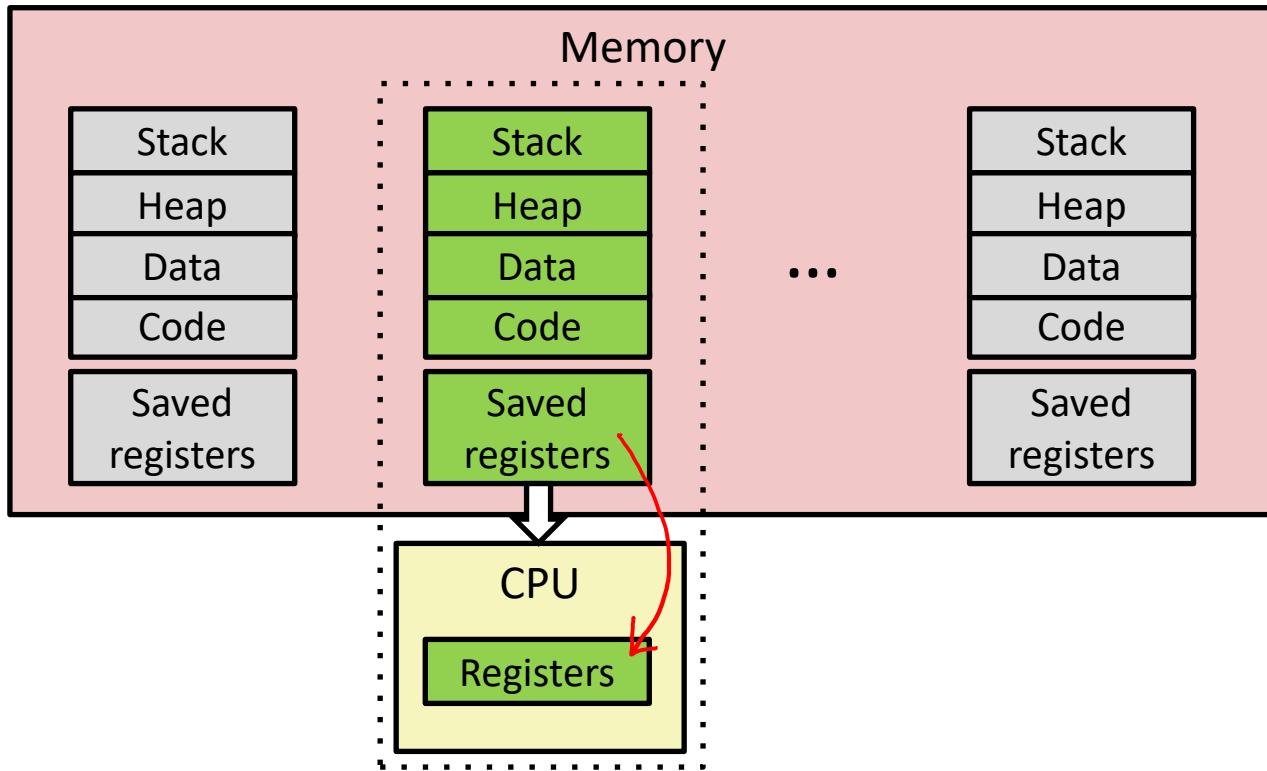
- Context switch
 - 1) Save current registers in memory

Multiprocessing



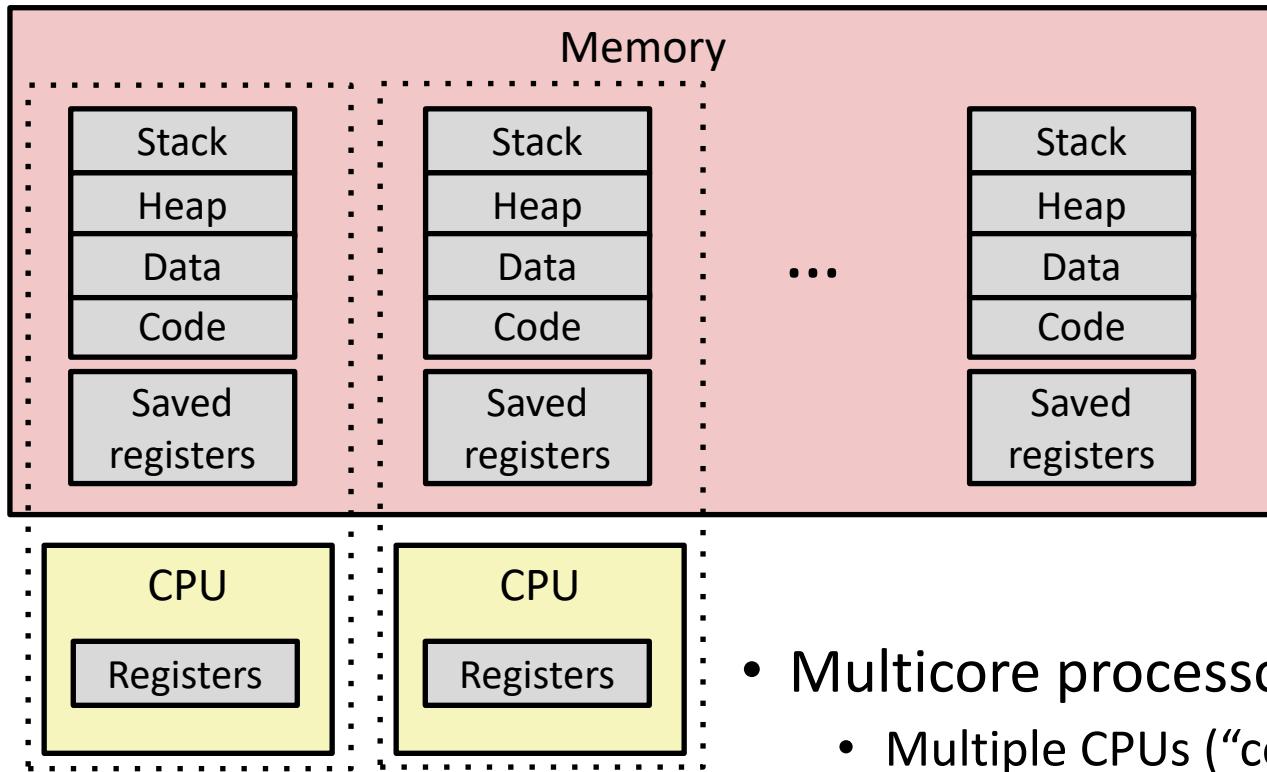
- Context switch
 - 1) Save current registers in memory
 - 2) Schedule next process for execution *(OS decides)*

Multiprocessing



- ❖ **Context switch**
- 1) Save current registers in memory
 - 2) Schedule next process for execution
 - 3) **Load saved registers and switch address space**

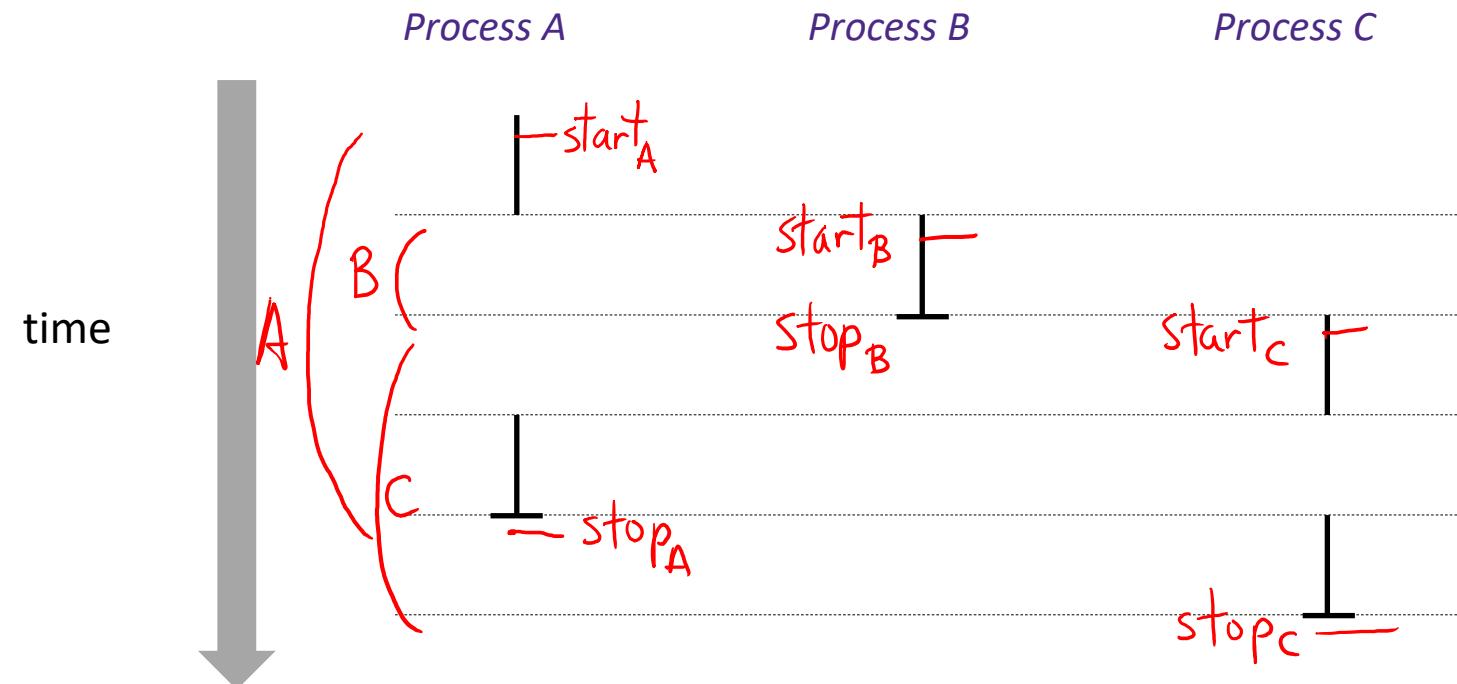
Multiprocessing: The (Modern) Reality



- Multicore processors
 - Multiple CPUs (“cores”) on single chip
 - Share main memory (and some of the caches)
 - Each can execute a separate process
 - Kernel schedules processes to cores
 - ***Still constantly swapping processes***

Concurrent Processes

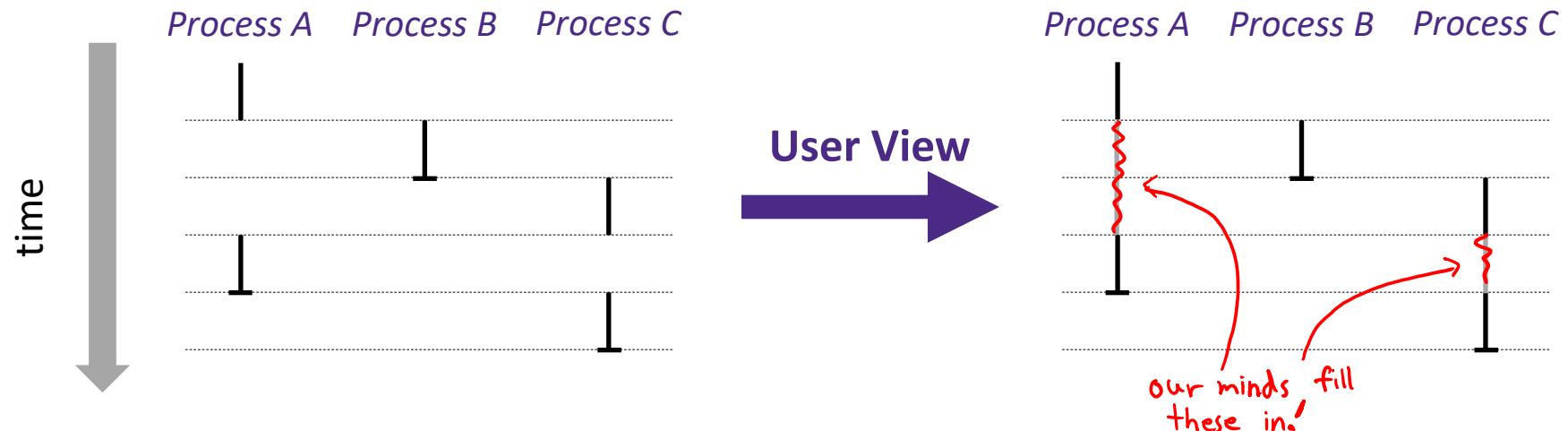
- Each process is a logical control flow
- Two processes *run concurrently* if their instruction executions overlap in time
 - Otherwise, they are *sequential*
- Example: (running on single core)
 - Concurrent: A & B, A & C
 - Sequential: B & C



User's View of Concurrency

Assume only one CPU

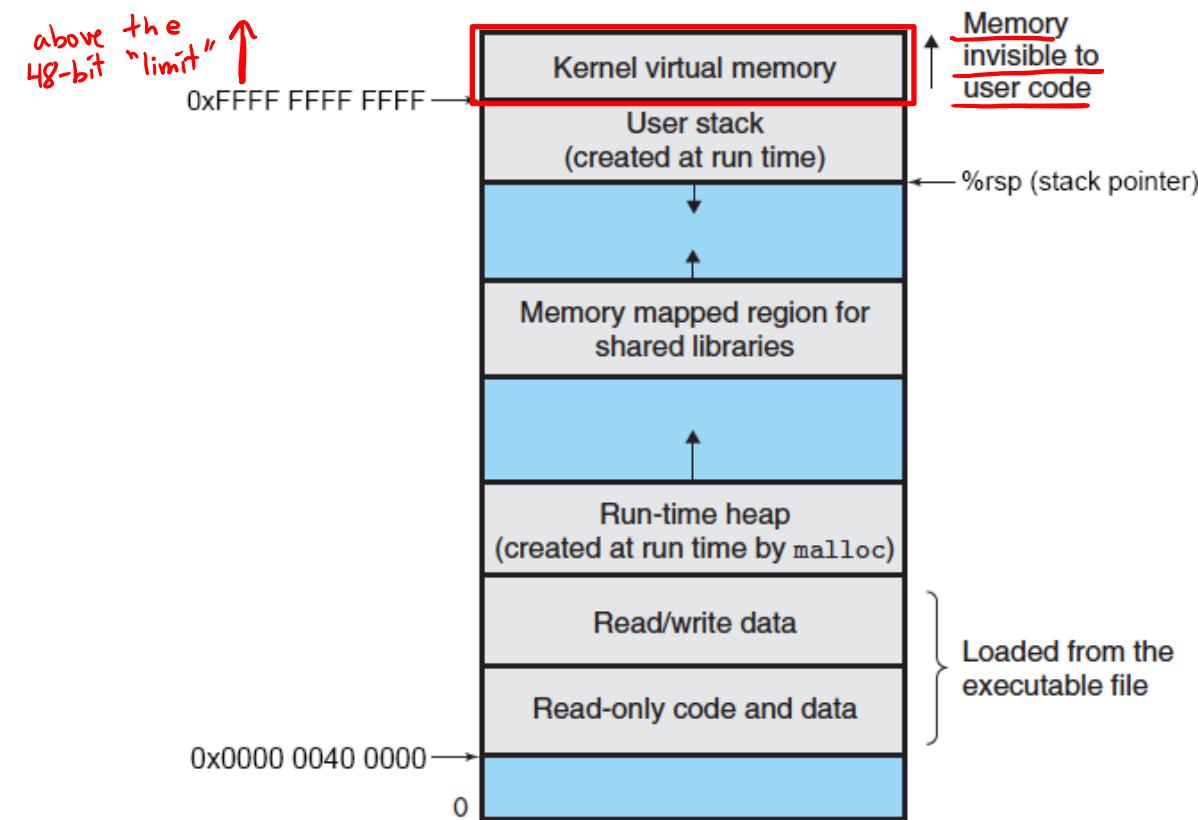
- Control flows for concurrent processes are physically disjoint in time
 - CPU only executes instructions for one process at a time
- However, the user can *think of concurrent processes as executing at the same time, in parallel*



Context Switching

Assume only one CPU

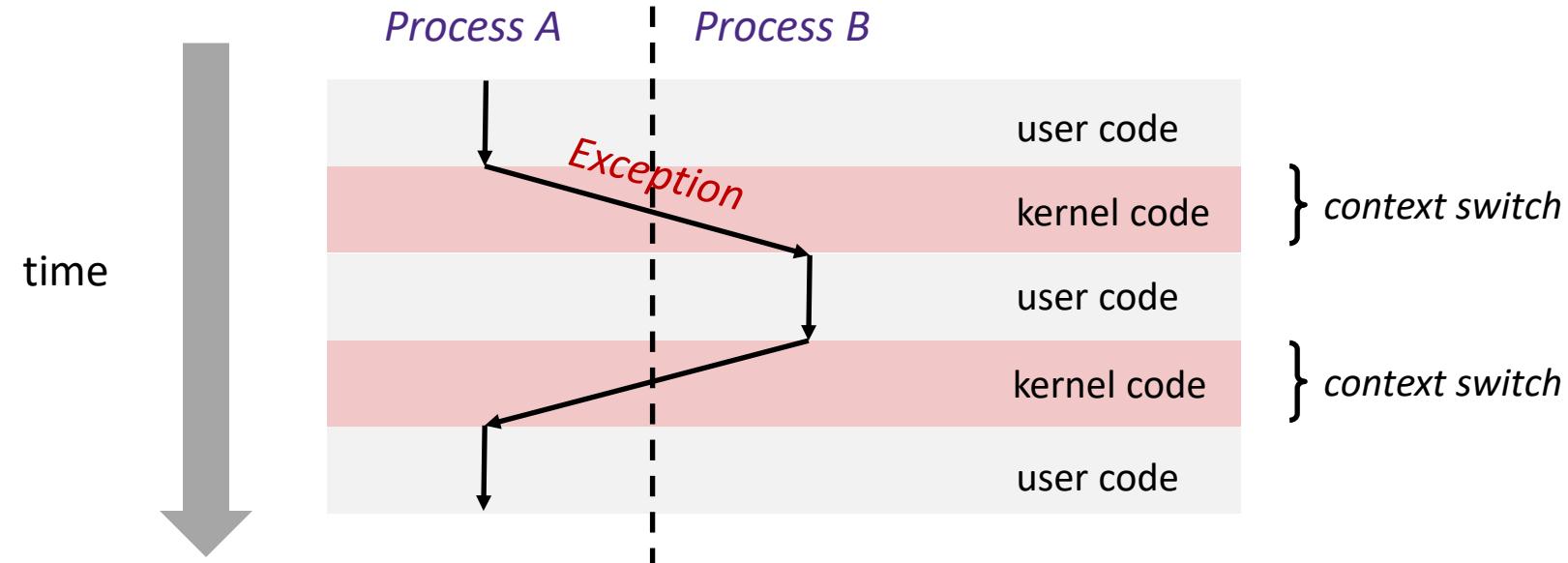
- Processes are managed by a *shared* chunk of OS code called the **kernel**
 - The kernel is not a separate process, but rather runs as part of a user process
- In x86-64 Linux:
 - Same address in each process refers to same shared memory location



Context Switching

Assume only one CPU

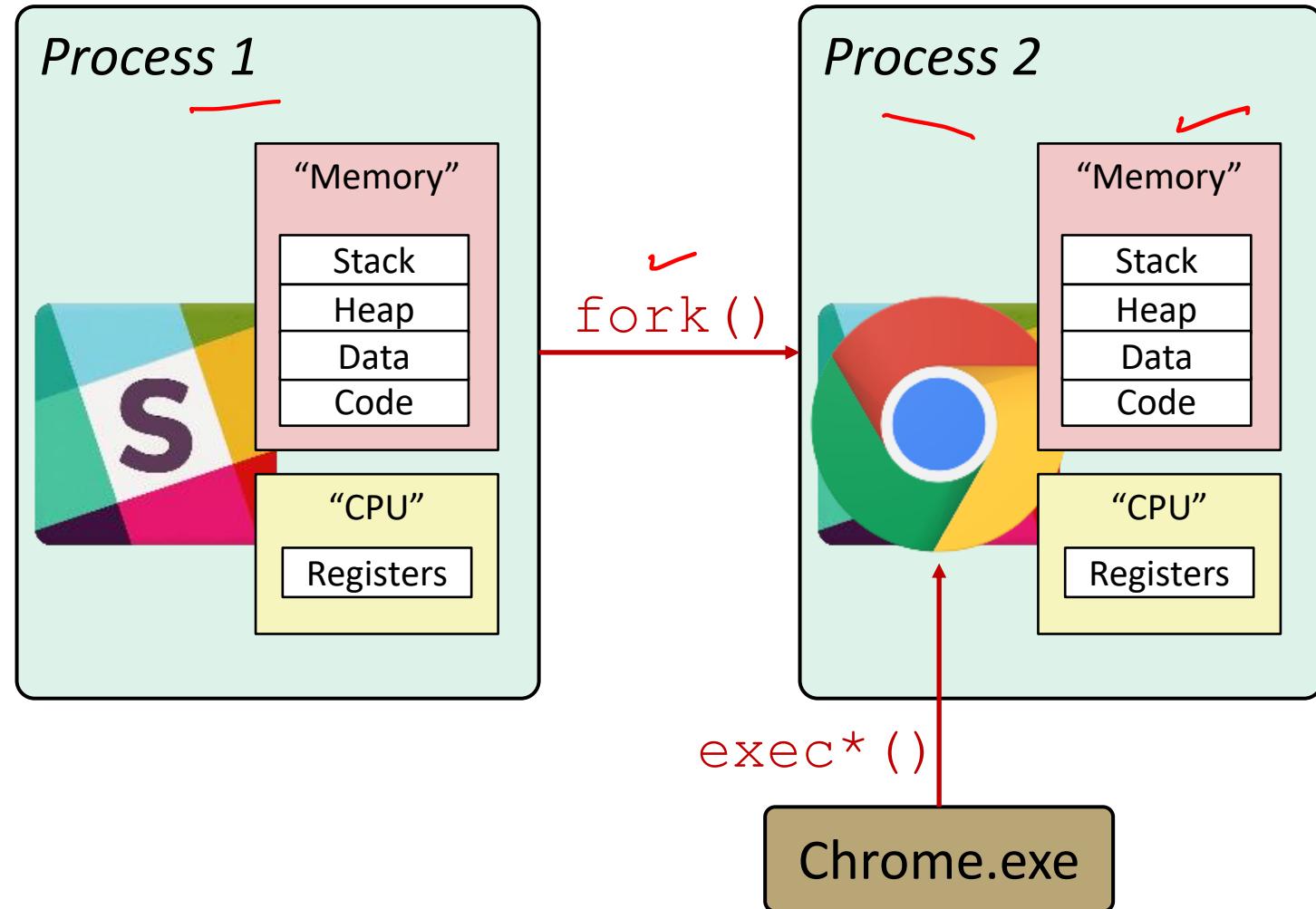
- Processes are managed by a *shared* chunk of OS code called the **kernel**
 - The kernel is not a separate process, but rather runs as part of a user process
- Context switch passes control flow from one process to another and is performed using kernel code



Processes

- Processes and context switching
- **Creating new processes**
 - `fork()`, `exec*`(()), and `wait()`
- Zombies

Creating New Processes & Programs



Creating New Processes & Programs



- fork-ex~~ec~~ model (Linux):
 - fork() creates a copy of the current process
 - exec*() replaces the current process' code and address space with the code for a different program
 - Family: execv, execl, execve, execle, execvp, execlp
 - fork() and execve() are *system calls*
- Other system calls for process management:
 - getpid()
 - exit()
 - wait(), waitpid()

fork: Creating New Processes

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```



- **pid_t fork(void)**

- Creates a new “**child**” process that is *identical* to the calling “**parent**” process, including all state (memory, registers, etc.)
- Returns 0 to the **child** process
- Returns child’s **process ID (PID)** to the **parent** process

- Child is *almost* identical to parent:

- Child gets an identical (but separate) copy of the parent’s virtual address space
 - Child has a different PID than the parent
- fork is unique (and often confusing) because it is called **once** but returns “**twice**”

Understanding fork

Process X (parent)

PID X

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

Process Y (child)

PID Y

```
pid_t pid = fork();  
if (pid == 0) {  
    printf("hello from child\n");  
} else {  
    printf("hello from parent\n");  
}
```

fork

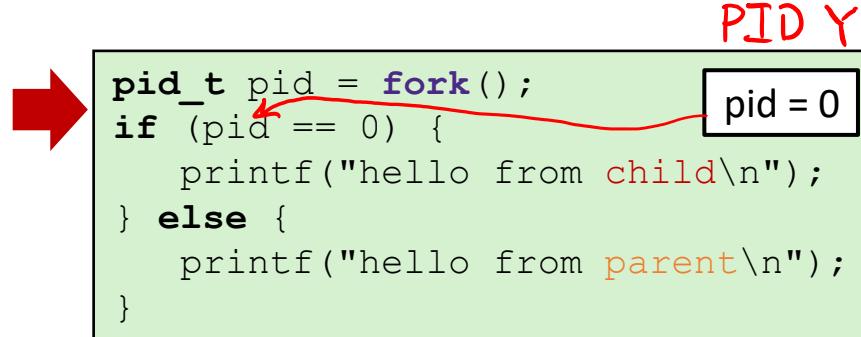
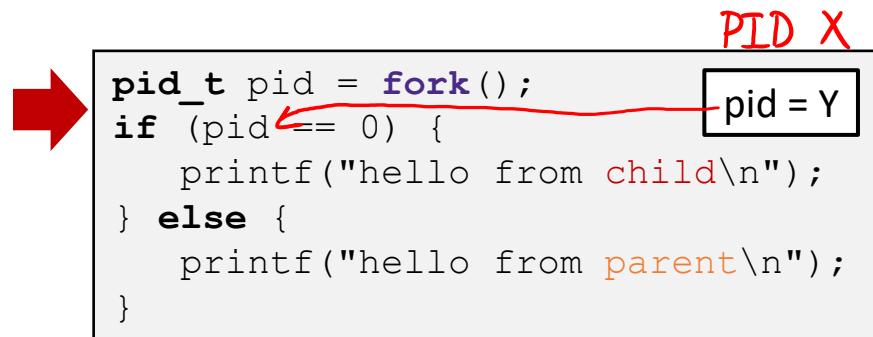
Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```



Understanding fork

Process X (parent)

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Process Y (child)

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = Y

```
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = 0

hello from parent

hello from child

*Which one appears first?
non-deterministic!*

Fork Example

```

void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) splits here
        printf("Child has x = %d\n", x++); child only
    else
        printf("Parent has x = %d\n", --x); parent only
    printf("Bye from process %d with x = %d\n", getpid(), x); both
}

```

- Both processes continue/start execution after `fork`
 - Child starts at instruction after the call to `fork` (storing into `pid`)
- Can't predict execution order of parent and child
- Both processes start with `x=1`
 - Subsequent changes to `x` are independent
- Shared open files: `stdout` is the same in both parent and child

Modeling fork with Process Graphs

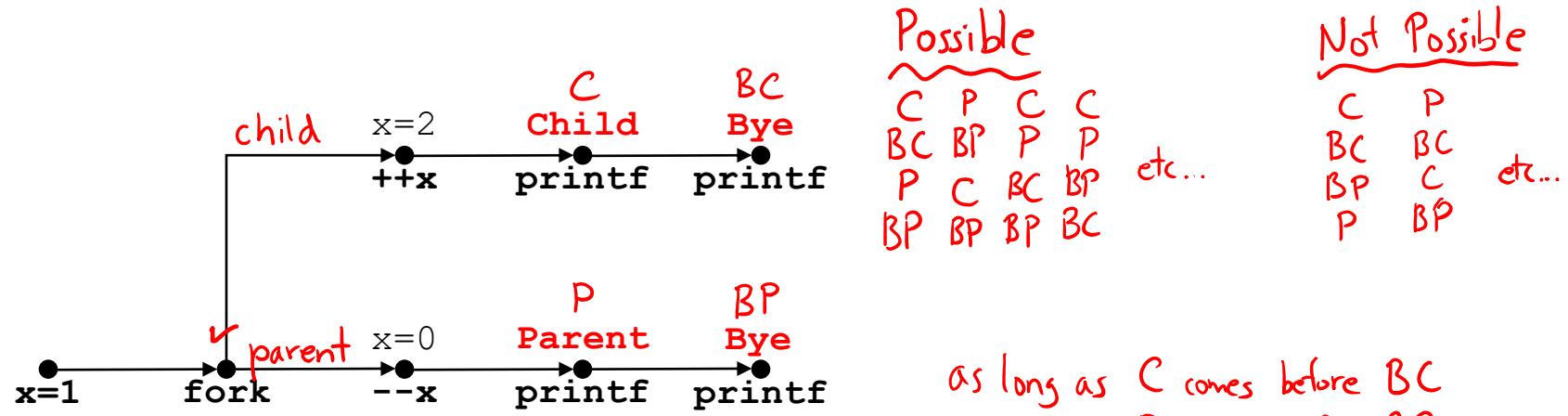
- A *process graph* is a useful tool for capturing the partial ordering of statements in a concurrent program
 - Each vertex is the execution of a statement
 - $a \rightarrow b$ means a happens before b
 - Edges can be labeled with current value of variables
 - **printf** vertices can be labeled with output
 - Each graph begins with a vertex with no in-edges
- Any *topological sort* of the graph corresponds to a feasible total ordering
 - Total ordering of vertices where all edges point from left to right

Fork Example: Possible Output

```

void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
    printf("Bye from process %d with x = %d\n", getpid(), x);
}

```



Fork-Exec

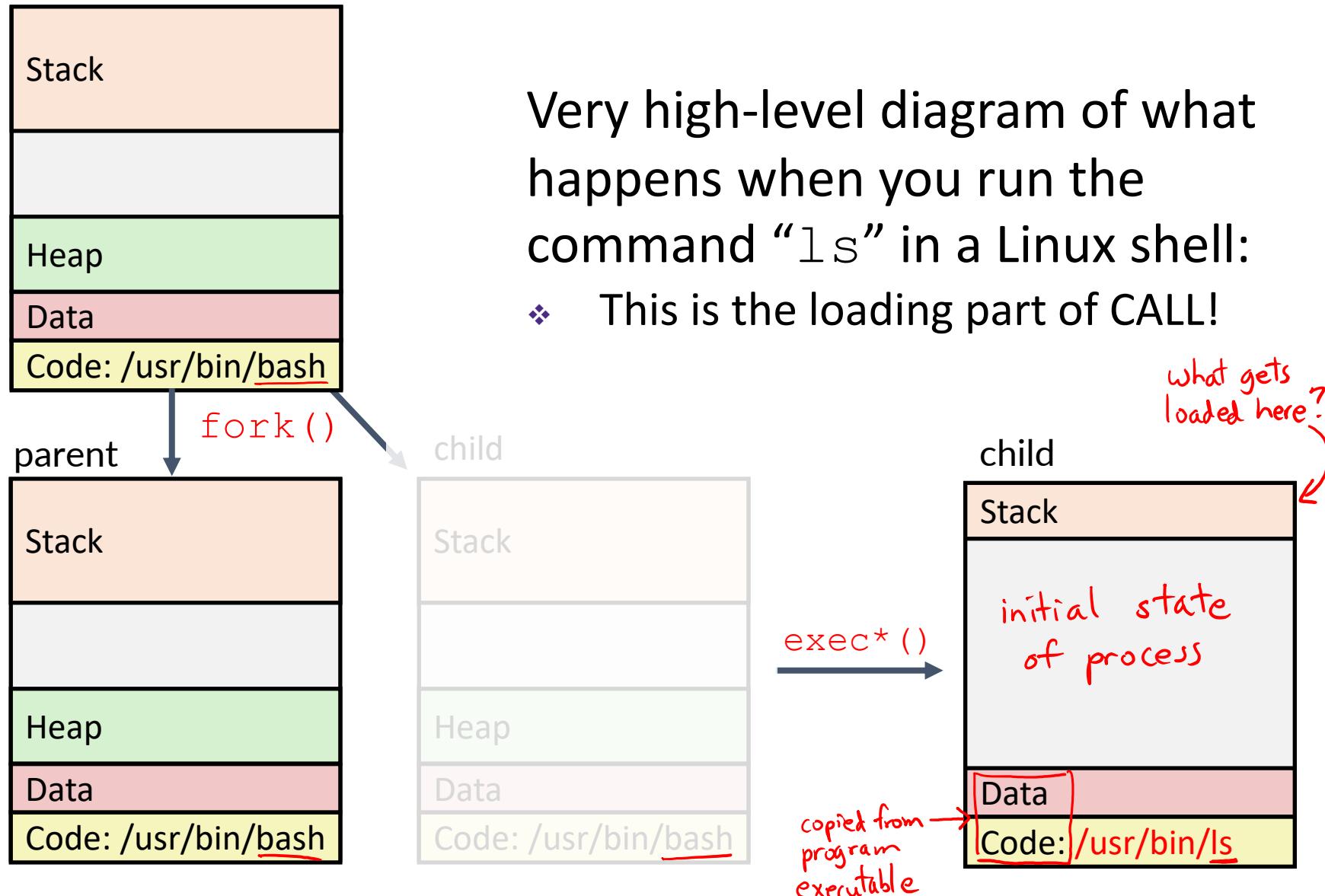
Note: the return values of `fork` and `exec*` should be checked for errors

- fork-exec model:

- `fork()` creates a copy of the current process
- `exec*()` replaces the current process' code and address space with the code for a different program
 - Whole family of `exec` calls – see **exec (3)** and **execve (2)**

```
// Example arguments: path="/usr/bin/ls",
//           argv[0]="/usr/bin/ls", argv[1]="-ahl", argv[2]=NULL
void fork_exec(char *path, char *argv[]) {
    pid_t pid = fork();
    if (pid != 0) {
        printf("Parent: created a child %d\n", pid);
    } else {
        printf("Child: about to exec a new program\n");
        execv(path, argv);
    }
    printf("This line printed by parent only!\n");
}
```

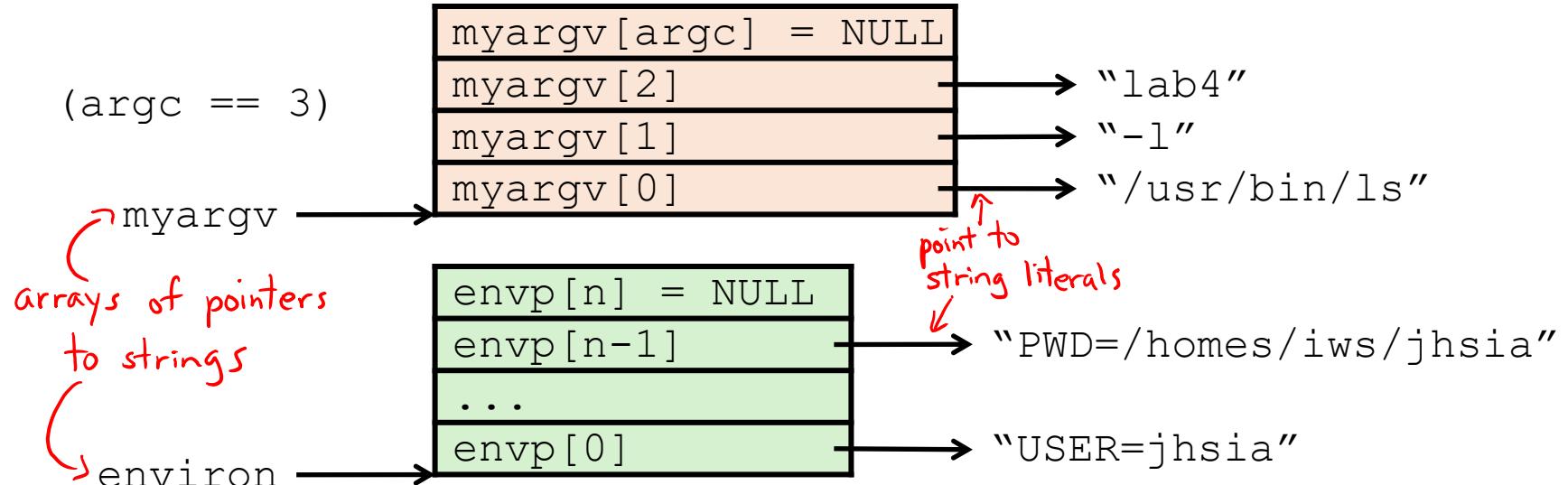
Exec-ing a new program



execve Example

```
int main(int argc, char* argv[])
    get command-line
    arguments into program
```

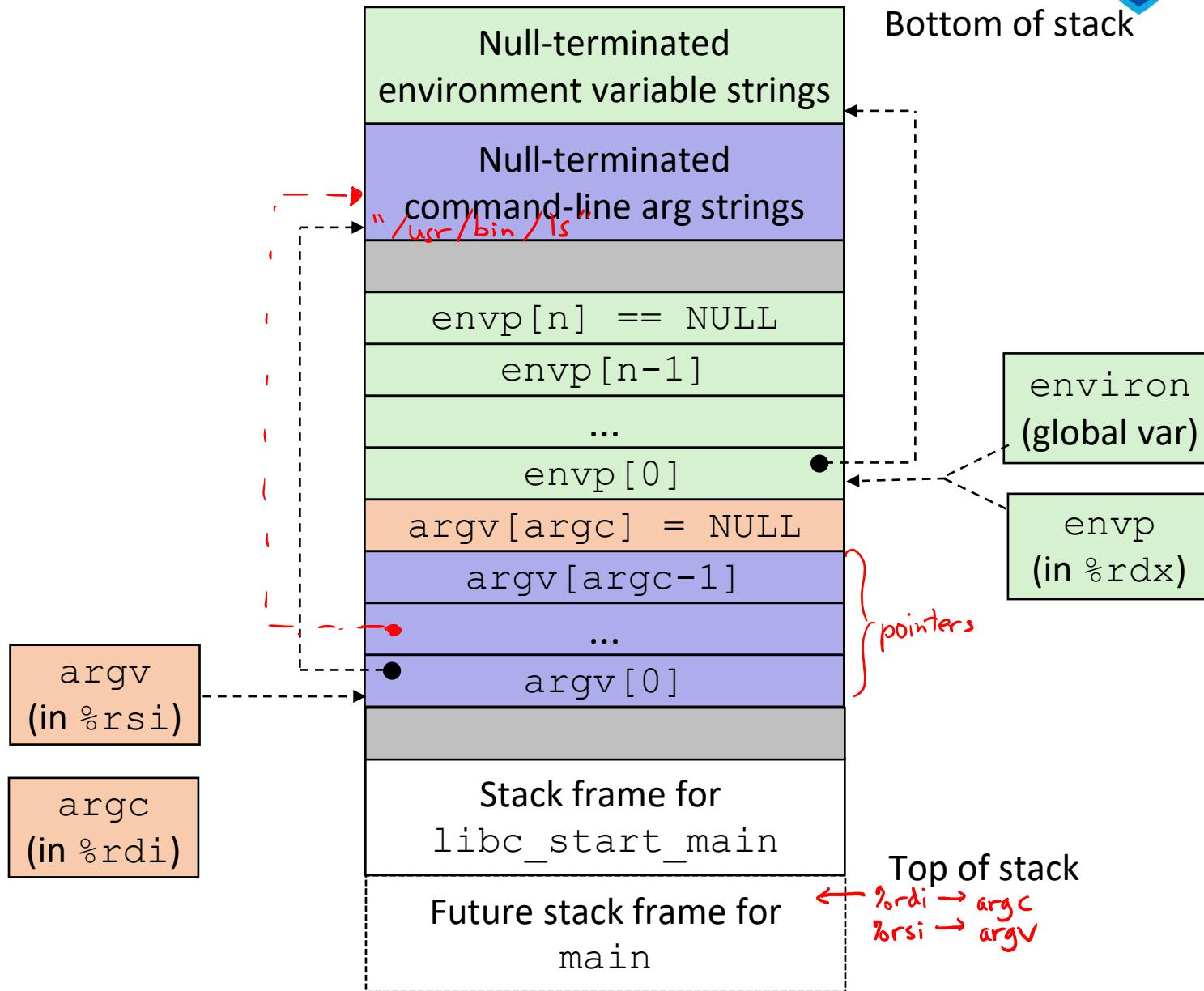
Execute "/usr/bin/ls -l lab4" in child process using current environment:



```
if ((pid = fork()) == 0) { /* Child runs program */
    if (execve(myargv[0], myargv, environ) < 0) {
        printf("%s: Command not found.\n", myargv[0]);
        exit(1);
    }
}
```

Run the printenv command in a Linux shell to see your own environment variables

Structure of the Stack when a new program starts



exit: Ending a process

- **void exit(int status)**
 - Exits a process
 - Status code: 0 is used for a normal exit, nonzero for abnormal exit

Processes

- Processes and context switching
- Creating new processes
 - `fork()`, `exec*`(), and `wait()`
- **Zombies**

Zombies

- When a process terminates, it still consumes system resources
 - Various tables maintained by OS
 - Called a “**zombie**” (a living corpse, half alive and half dead)
- *Reaping* is performed by parent on terminated child
 - Parent is given exit status information and kernel then deletes zombie child process
- What if parent doesn’t reap?
 - If any parent terminates without reaping a child, then the orphaned child will be reaped by `init` process (`pid == 1`)
 - **Note:** on more recent Linux systems, `init` has been renamed `systemd`
 - In long-running processes (e.g. shells, servers) we need *explicit* reaping

wait: Synchronizing with Children

- **int** wait(**int** *child_status)
 - Suspends current process (*i.e.* the parent) until one of its children terminates
 - Return value is the PID of the child process that terminated
 - *On successful return, the child process is reaped*
 - If child_status != NULL, then the *child_status value indicates why the child process terminated
 - Special macros for interpreting this status – see **man wait(2)**
- **Note:** If parent process has multiple children, wait will return when *any* of the children terminates
 - waitpid can be used to wait on a specific child process

wait: Synchronizing with Children

```

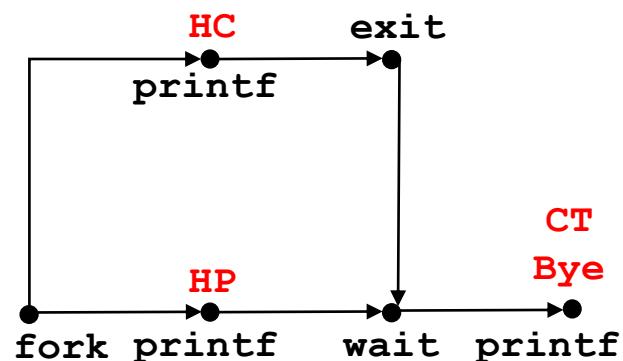
void fork_wait() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}

```

forks.c

} child
}
parent



Feasible output:

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

HP
CT
Bye
HC

Example: Zombie

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
```

```
linux> ps
  PID TTY      TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6639 tttyp9    00:00:03 forks
 6640 tttyp9    00:00:00 forks <defunct>
 6641 tttyp9    00:00:00 ps
```

parent

child

```
linux> kill 6639
[1]    Terminated
```

```
linux> ps
  PID TTY      TIME CMD
 6585 tttyp9    00:00:00 tcsh
 6642 tttyp9    00:00:00 ps
```

```
void fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    }
}
```

parent persists

forks.c

- ps shows child process as “defunct”
- Killing parent allows child to be reaped by init

Example:

Non-terminating Child

```

void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    } else {
        /* child persists */
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}

```

forks.c

```

linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID  TTY      TIME CMD
  6585 tttyp9  00:00:00 tcsh
  6676 tttyp9  00:00:06 forks
  6677 tttyp9  00:00:00 ps
linux> kill 6676
linux> ps
  PID  TTY      TIME CMD
  6585 tttyp9  00:00:00 tcsh
  6678 tttyp9  00:00:00 ps

```

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

Process Management Summary

- `fork` makes two copies of the same process (parent & child)
 - Returns different values to the two processes
- `exec*` replaces current process from file (new program)
 - Two-process program:
 - First `fork()`
 - `if (pid == 0) { /* child code */ } else { /* parent code */ }`
 - Two different programs:
 - First `fork()`
 - `if (pid == 0) { execv(...) } else { /* parent code */ }`
- `wait` or `waitpid` used to synchronize parent/child execution and to reap child process

Summary

- Processes
 - At any given time, system has multiple active processes
 - On a one-CPU system, only one can execute at a time, but each process appears to have total control of the processor
 - OS periodically “context switches” between active processes
 - Implemented using *exceptional control flow*
- Process management
 - `fork`: one call, two returns
 - `execve`: one call, usually no return
 - `wait` or `waitpid`: synchronization
 - `exit`: one call, no return



Thanks
Q & A



File Descriptors & System Calls

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Contents

- File Descriptor
- System Calls
- System call Vs Function Call
- Sequence of making system call
- Q & A

File Descriptors

- A *file descriptor* is the Unix abstraction for an open I/O stream:
 - a file, a network connection, a pipe, a terminal, etc.
- **File descriptor table:** an array of kernel objects
- The file descriptors that applications manipulate are *indexes* into this table.

File Descriptors

- Logically, a file descriptor comprises
 - a **file reference** (/home/vimal/data.txt) and
 - a **file position** (an offset into the file)
- There can be many file descriptors simultaneously open for the same file reference, each with a different position.

File Descriptors

- For disk files, the position can be explicitly changed:
 - a process can rewind and re-read part of a file or skip around.
- These files are called *seekable*.
- However, not all types of file descriptor are seekable.

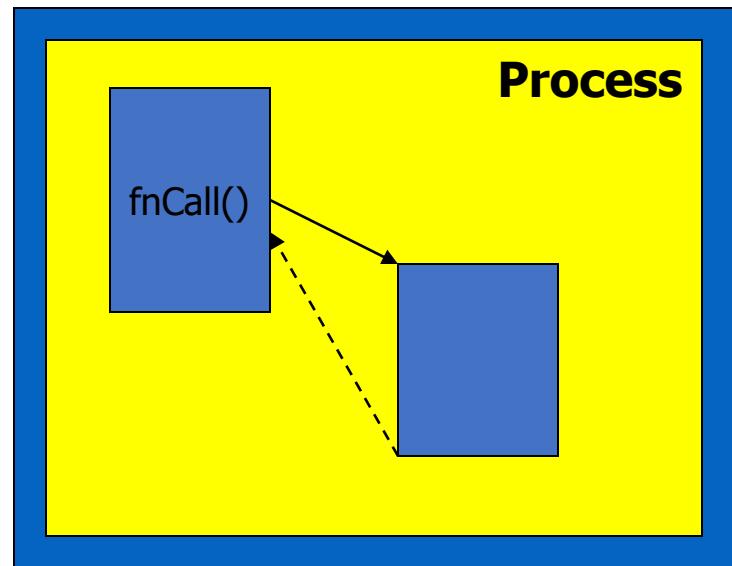
System Calls

- **System Calls**
 - A request to the OS to perform some activity
- **System calls are expensive**
 - The system needs to perform many things before executing a system call
 - The computer (hardware) saves its state
 - The OS code takes control of the CPU, privileges are updated.
 - The OS examines the call parameters
 - The OS performs the requested function
 - The OS saves its state (and call results)
 - The OS returns control of the CPU to the caller

System Calls versus Function Calls?

System Calls versus Function Calls

Function Call

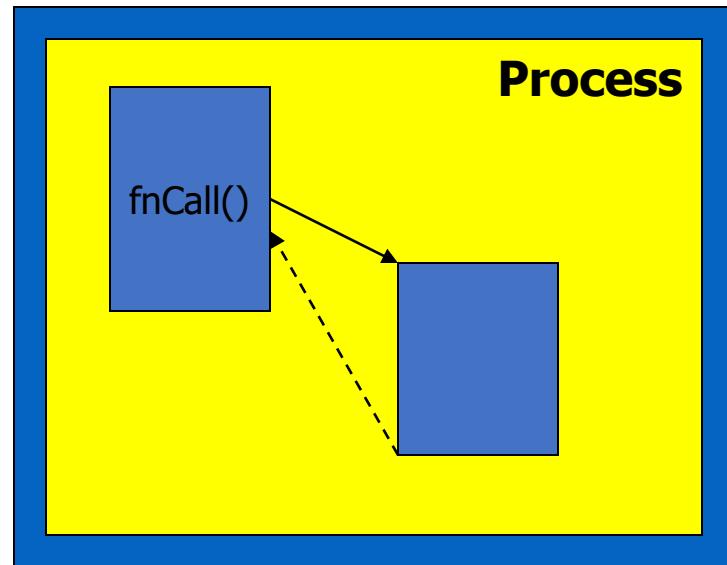


Caller and callee are in the same Process

- Same user
- Same “domain of trust”

System Calls versus Function Calls

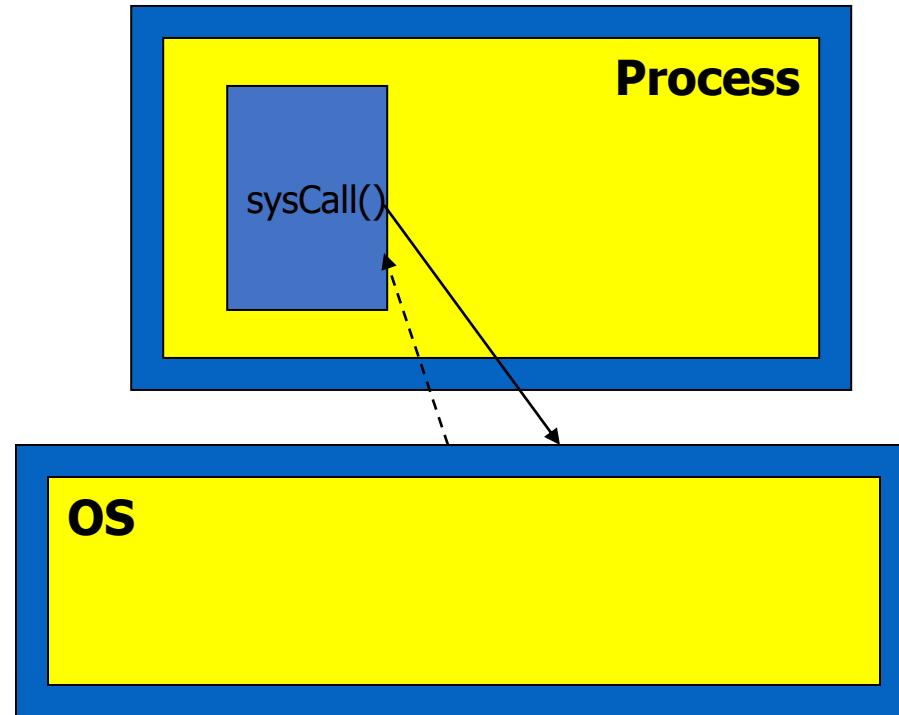
Function Call



Caller and callee are in the same Process

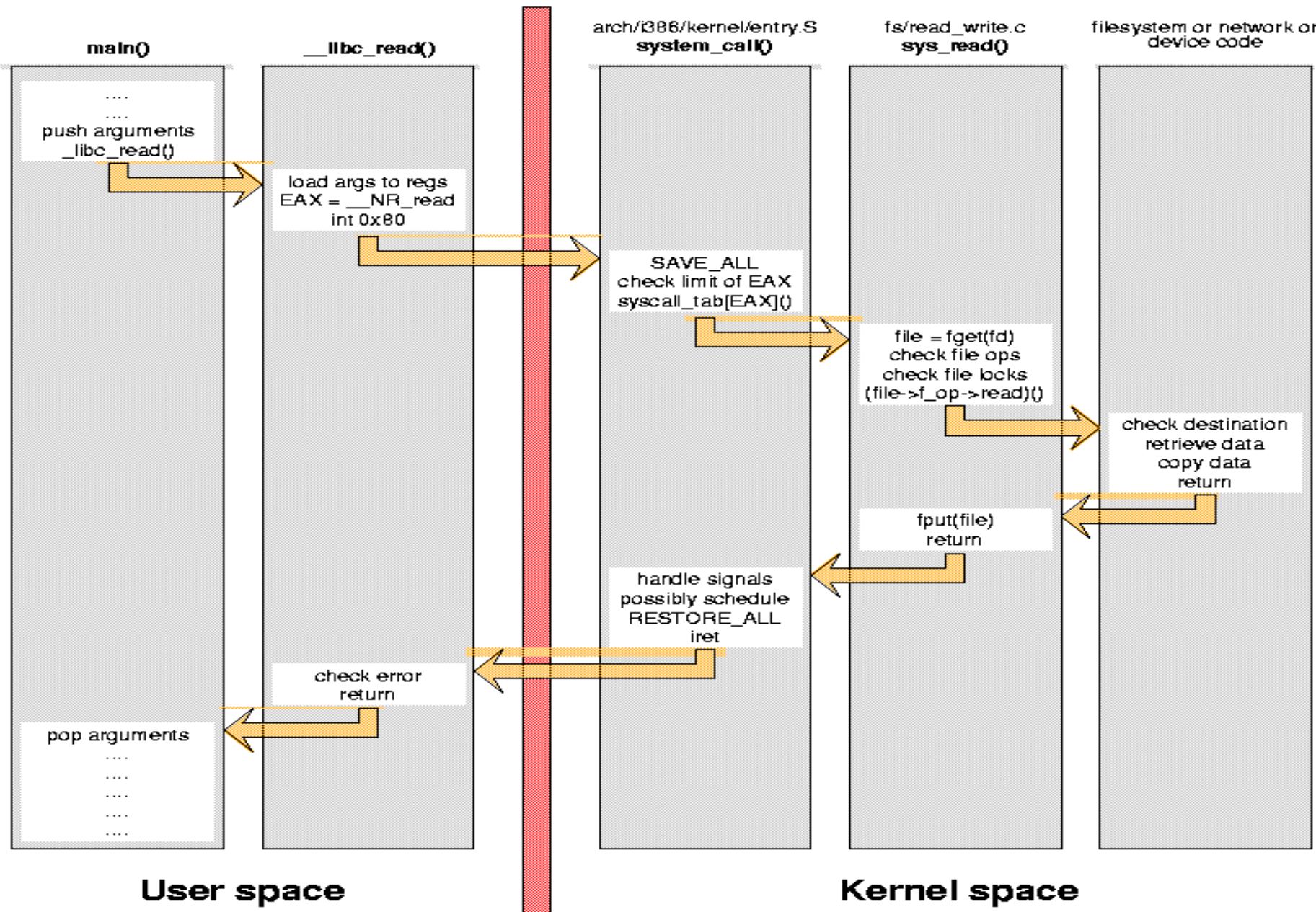
- Same user
- Same “domain of trust”

System Call



- OS is trusted; user is not.
- OS has super-privileges; user does not
- Must take measures to prevent abuse

Sequence for Making a System Call (Example: read call)



Steps for Making a System Call (Example: read call)

- A system call is implemented by a “*software interrupt*” that transfers control to kernel code; in Linux/i386 this is “interrupt 0x80”.
- The specific system call being invoked is stored in the EAX register, and its arguments are held in the other CPU registers.
- *After the switch to kernel mode, the CPU must save all of its registers and dispatch execution to the proper kernel function, after checking whether EAX is out of range.*
- The read finally performs the data transfer and all the previous steps are unwound up to the calling user function.

Cost for Making a System Call (Example:read)

- Each arrow in the figure represents a jump in CPU instruction flow, and each jump may require flushing the prefetch queue and possibly a ***“cache miss” event***.
- Transitions between user and kernel space are especially important, as they are the most expensive in processing time and prefetch behavior.

File Locking & Record Locking

- **File locking** is the capability to prevent other processes from reading/writing any part of a file
- **Record locking** is the capability to prevent other processes from reading/writing particular records.
- `fcntl(fd, cmd, arg);` system call is used to implement file/record locking
 - fd= file descriptor, cmd= type of lock (R/W), arg=lock type/byte offset,
- Kernel releases the locks as soon as the process closes the file

System calls Examples

- System calls to access the existing files:
 - open(), read(), write(), lseek(), and close()
 - getuid() //get the user ID
 - fork() //create a child process
 - exec() //executing a program => execve (2) in Linux
 - System calls to create new files:
 - Create(), mknod()
 - System calls to modify the inode or maneuver through the filesystem:
 - chdir(), chroot(), chown(), chmod(), stat(), fstat()
 - Advanced System calls :
 - pipe(), dup(), mount(), umount(), link(), unlink()
-
- Don't confuse system calls with *libc* calls
 - Differences?
 - Is printf() a system call?
 - Is rand() a system call?

System calls vs. *libc*

Each I/O system call has corresponding procedure calls from the standard I/O library.

System calls	Library calls
open	fopen
close	fclose
read	fread, getchar, scanf, fscanf, getc, fgetc, gets, fgets
write	fwrite, putchar, printf, fprintf putc, fputc, puts, fputs
lseek	fseek

Use man -s 2

Use man -s 3

Define a new system call

- Create a directory hello in the kernel source directory:
 - **mkdir hello**
- Change into this directory
 - **cd hello**
- Create a “hello.c” file in this folder and add the definition of the system call to it as given below using any text editor.

```
#include <linux/kernel.h>  asmlinkage long
sys_hello(void)
{
    printk(KERN_INFO "Hello world\n"); return 0;
}
```

Create Makefile

- Create a “Makefile” in the hello folder and add the given line to it.
 - **gedit Makefile**
- add the following line to it:-
 - **objy := hello.o**
- This is to ensure that the hello.c file is compiled and included in the kernel source code.

Add the hello directory to the kernel's Makefile

- change back into the linux-3.16 folder and open Makefile.
 - gedit Makefile
 - goto line number 842 which says :- **"core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ "**
 - change this to **"core-y += kernel/ mm/ fs/ ipc/ security/ crypto/ block/ hello/"**
- This is to tell the compiler that the source files of our new system call (sys_hello()) are present in the hello directory.

Add system call to the table

- If your system is a 64 bit system you will need to alter the **syscall_64.tbl** file else **syscall_32.tbl**.
 - **cd arch/x86/syscalls**
 - **gedit syscall_32.tbl**
- Add the following line in the end of the file :-

358 i386 hello sys_hello

358 – It is the number of the system call .

It should be one plus the number of the last system call. (it was 358 in my system). This has to be noted down to make the system call in the userspace program.

Add system call to header file

- **cd include/linux/**
- **gedit syscalls.h**
 - ✓ add the following line to the end of the file just before the #endif statement at the very bottom.
 - **asmlinkage long sys_hello(void);**
- This defines the prototype of the function of our system call.
- “**asmlinkage**” is a key word used to indicate that all parameters of the function would be available on the stack.

Now compile the linux source code according to the standard procedure.

Test the system call

- Create a “userspace.c” program in your home folder and type in the following code :-

```
#include <stdio.h>
#include <linux/kernel.h> #include
<sys/syscall.h> #include <unistd.h>

int main()
{
    long int r = syscall(358);
    printf("System call sys_hello returned %ld\n", r); return 0;
}
```

Test the system call

- Now compile this program using the following command.
 - **gcc userspace.c**
- If all goes well you will not have any errors else, rectify the errors. Now run the program using the following command.
 - **./a.out**
- You will see the following line getting printed in the terminal if all the steps were followed correctly.
 - **“System call sys_hello returned 0“.**
- Now to check the message of the kernel you can run the following command.
 - **dmesg**

Why does the OS control I/O?

Why does the OS control I/O?

- **Safety:** The computer must ensure that if my program has a bug in it, then it doesn't crash or mess up
 - the system,
 - other programs that may run at the same time or later.
- **Fairness:** Make sure other programs have a fair use of device

System Calls for I/O

- There are 5 basic system calls that Unix provides for file I/O
 - `int open(char *path, int flags [, int mode]);` (check man -s 2 open)
 - `int close(int fd);`
 - `int read(int fd, char *buf, int size);`
 - `int write(int fd, char *buf, int size);`
 - `off_t lseek(int fd, off_t offset, int whence);`
- Some library calls themselves make a system call
 - (e.g. `fopen()` calls `open()`)

Open

- **int open(char *path, int flags [, int mode])** makes a request to the OS to use a file.
 - The '**path**' argument specifies the file you would like to use
 - The '**flags**' and '**mode**' arguments specify how you would like to use it.
 - If the OS approves your request, it will return a *FD*.
 - This is a non-negative integer. Any future accesses to this file needs to provide this FD
 - If it returns -1, then you have been denied access; check the value of global variable "**errno**" to determine why (or use **perror()** to print corresponding error message).

perror() & errno

- When a system call fails, it usually returns -1 and sets the variable errno to a value describing what went wrong.
- (These values can be found in <errno.h>.) Many library functions do likewise.
- The function perror() serves to translate this error code into human-readable form.

Standard Input, Output and Error

- Now, every process in Unix starts out with three FDs predefined:
 - File descriptor 0 is standard input.
 - File descriptor 1 is standard output.
 - File descriptor 2 is standard error.
- We can read from standard input, using **read(0, ...)**, and write to standard output using **write(1, ...)** or using two **library** calls
 - **printf**
 - **scanf**

Example 1

```
#include <fcntl.h>
#include <errno.h>

main(int argc, char** argv) {
    int fd;
    fd = open("foo.txt", O_RDONLY);
    printf("%d\n", fd);
    if (fd== -1) {
        fprintf (stderr, "Error Number %d\n", errno);
        perror("Program example1_open");
    }
}
```

Other flag options are:
O_WRONLY or O_RDWR
(see man pages for more flags!)

Close()

- **int close(int fd)**

Tells the OS you are done with a FD.

```
#include <fcntl.h>
main(){
    int fd1, fd2;

    if(( fd1 = open("foo.txt", O_RDONLY)) < 0){
        perror("foo.txt");
        exit(1);
    }
    if (close(fd1) < 0) {
        perror("foo.txt");
        exit(1);
    }
    printf("closed the fd's\n");
```

Why do we need to close a file?

After close, can you still use the file descriptor?

read(...)

- int **read(int fd, char *buf, int size)** tells the OS:
 - To read "**size**" bytes from the file specified by "**fd**" into the memory location pointed to by "**buf**".
 - It returns how many bytes were actually read (**why?**)
 - 0 : at end of the file
 - < size : fewer bytes are read to the buffer (**why?**)
 - == size : read the specified # of bytes
- Things to be careful about
 - buf must point to valid memory not smaller than the specified size
 - Otherwise, what could happen?
 - fd should be a valid file descriptor returned from open() to perform read operation

read(...)

- It is not an error and if the number of read bytes is smaller than the number of bytes requested
 - this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe)
 - it can also happen because read() was interrupted by a signal.

Example 2

```
#include <fcntl.h>
main(int argc, char** argv) {
    char *c;
    int fd, sz;
    c = (char *) malloc(100 * sizeof(char));

    fd = open("foo.txt", O_RDONLY);
    if (fd < 0) { perror("foo.txt"); exit(1); }

    sz = read(fd, c, 10);
    printf("called read(%d, c, 10), which read %d bytes.\n", fd, sz);
    c[sz] = '\0';
    printf("Those bytes are as follows: %s\n", c);

    close(fd);
}
```

write(...)

- **int write(int fd, char *buf, int size)** writes the bytes stored in **buf** to the file specified by **fd**
 - It returns the number of bytes written, which is usually “size” unless there is an error
- Things to be careful about
 - **buf** must be at least as long as “size”
 - The file must be open for write operations

Example 3

```
#include <fcntl.h>
main()
{
    int fd, sz;
```

```
fd = open("out3", O_RDWR | O_CREAT | O_APPEND, 0644);
```

```
if (fd < 0) { perror("out3"); exit(1); }
```

```
sz = write(fd, "cs241\n", strlen("cs241\n"));
```

```
printf("called write(%d, \"cs241\\n\", %d), which returned %d\\n",
       strlen("cs241\\n"), sz);
```

```
close(fd);
}
```

If the file does not exist,
it will be created

the file offset is set to the end of the
file before each write

lseek()

- All opened files have a "file pointer" associated with them to record the current position for the next file operation
 - When file is opened, file pointer points to the beginning of the file
 - After reading/writing m bytes, the file pointer moves m bytes forward
- **off_t lseek(int fd, off_t offset, int whence)** moves the file pointer explicitly
 - The 'whence' argument specifies how the seek is to be done
 - from the beginning of the file
 - from the current value of the pointer, or
 - from the end of the file
 - The return value is the offset of the pointer after the lseek
- How would you know to include sys/types.h and unistd.h?
 - Read "man -s 2 lseek"

Lseek() example

```
c = (char *) malloc(100 * sizeof(char));
fd = open("foo.txt", O_RDONLY);
if (fd < 0) { perror("foo.txt"); exit(1); }

sz = read(fd, c, 10);
printf("We have opened foo.txt, and called read(%d, c, 10).\n", fd);
c[sz] = '\0';
printf("Those bytes are as follows: %s\n", c);

i = lseek(fd, 0, SEEK_CUR);
printf("lseek(%d, 0, SEEK_CUR) returns the current offset = %d\n\n", fd, i);

printf("now, we seek to the beginning of the file and call read(%d, c, 10)\n", fd);
lseek(fd, 0, SEEK_SET);
sz = read(fd, c, 10);
c[sz] = '\0';
printf("The read returns the following bytes: %s\n", c);
...:
```



Thanks
Q & A

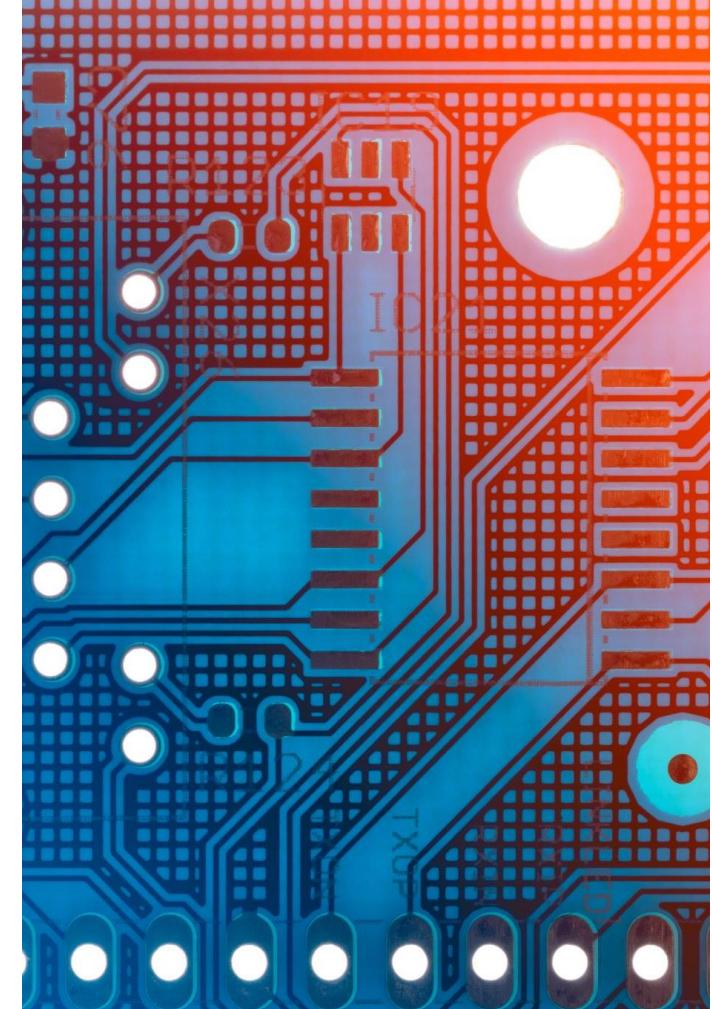
A close-up photograph of a person's hand gripping the boom of a sailboat. The hand is wearing a dark long-sleeved shirt. The background shows the blue ocean under a bright sky.

Linux Device Drivers

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Outline

- System Call Review
- Introduction to Device Drivers
- Why Device Drivers?
- Kernel & Device Drivers
- Policy Independence
- Kernel Functions
- Classes of Devices and Modules
- Security Issues
- Version Numbering & License Issues
- Q&A



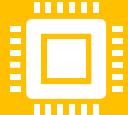
System Calls



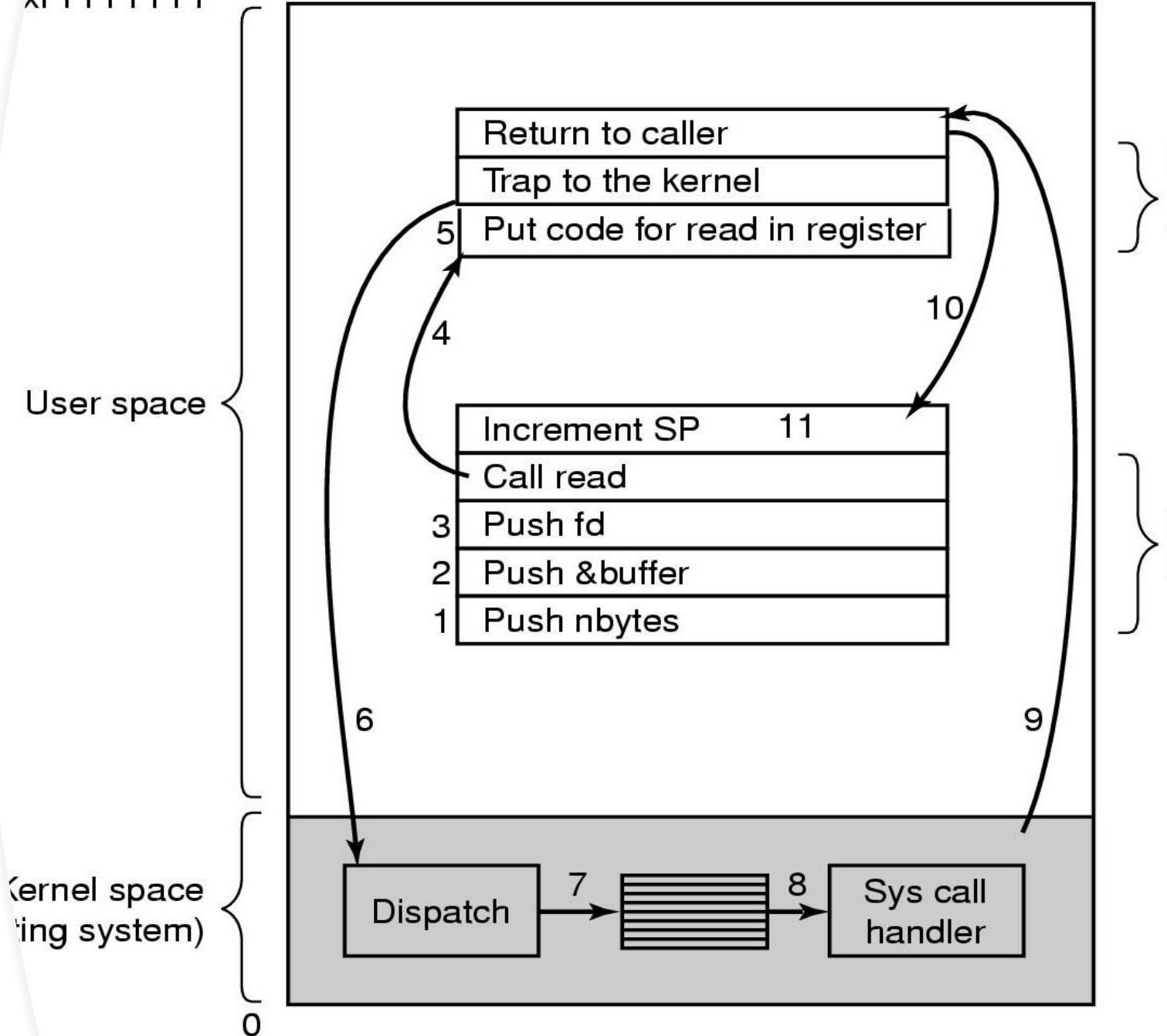
The mechanism used by an application program to request service from the operating system.



System calls often use a special machine code instruction which causes the processor to change mode (e.g. to "supervisor mode" or "protected mode").



This allows the OS to perform restricted actions such as accessing hardware devices or the memory management unit.



Steps in Making a System Call

- ✓ There are 11 steps in making the system call
- ✓ read (fd, buffer, nbytes)

Some System Calls For Process Management

Process management

Call	Description
pid = fork()	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some System Calls For File Management

File management

Call	Description
<code>fd = open(file, how, ...)</code>	Open a file for reading, writing or both
<code>s = close(fd)</code>	Close an open file
<code>n = read(fd, buffer, nbytes)</code>	Read data from a file into a buffer
<code>n = write(fd, buffer, nbytes)</code>	Write data from a buffer into a file
<code>position = lseek(fd, offset, whence)</code>	Move the file pointer
<code>s = stat(name, &buf)</code>	Get a file's status information

Some System Calls For Directory Management

Directory and file system management

Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some System Calls For Miscellaneous Tasks

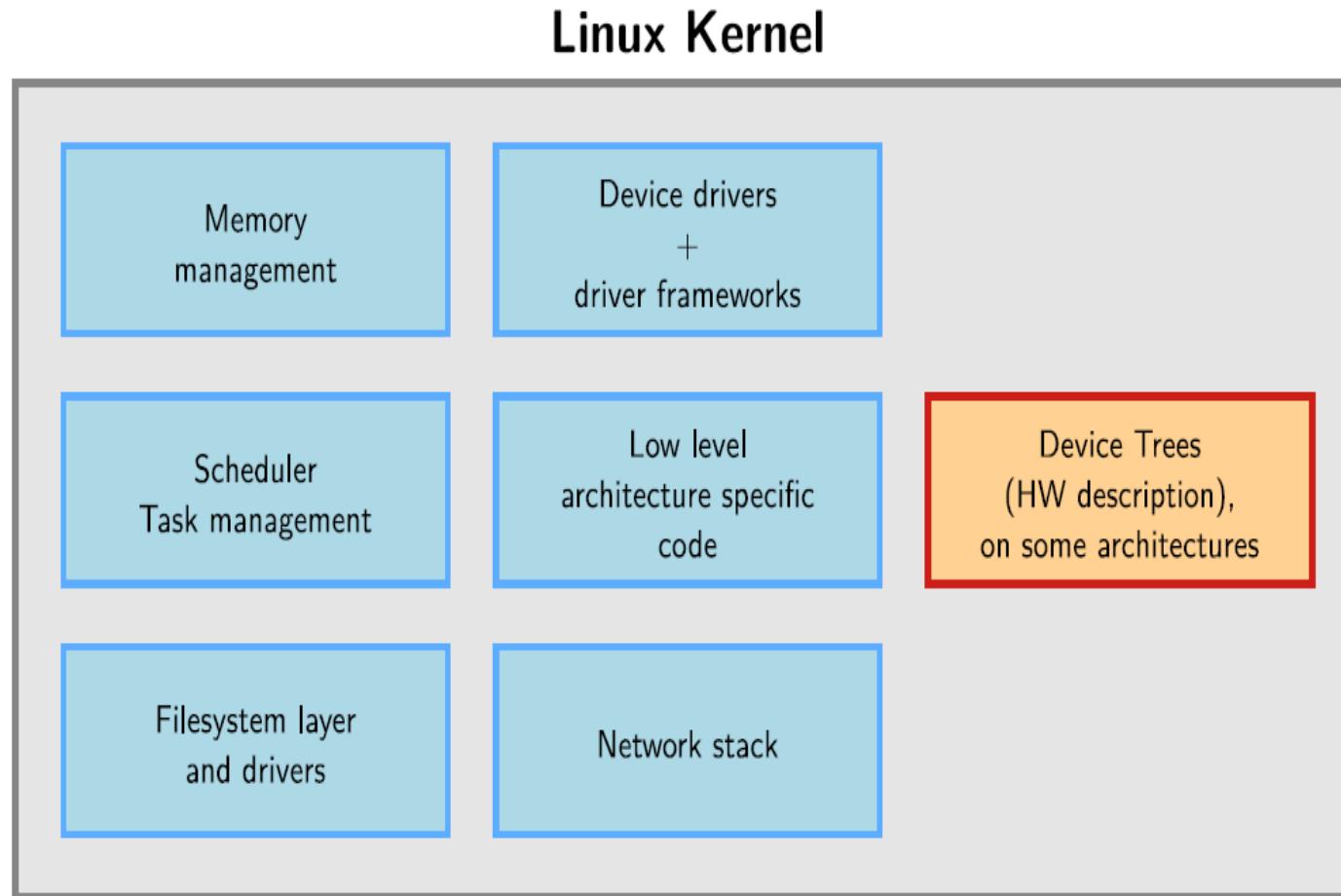
Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

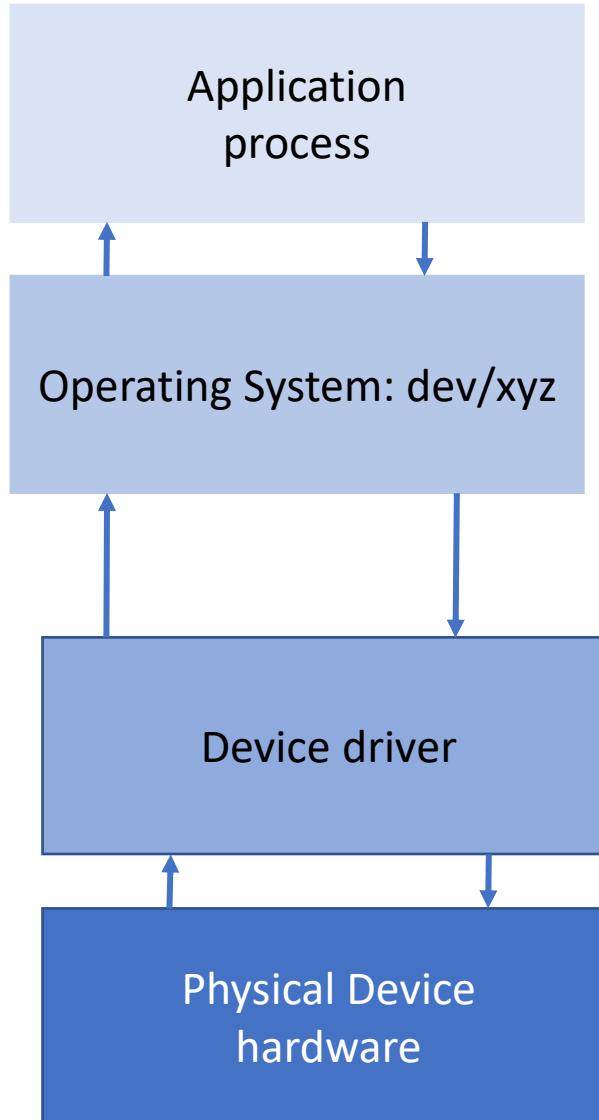
Introduction to Linux Device Drivers

- ✓ Device drivers are the “black boxes” that make a particular piece of hardware respond to a well-defined internal programming interface
- ✓ they hide completely the details of how the device works

- Device drivers
 - Software interface to hardware device
 - Use standardized calls
 - Independent of the specific driver
 - Main role
 - Map standard calls to device-specific operations
 - Can be developed separately from the rest of the kernel
 - Plugged in at runtime when needed



Why Device Drivers?

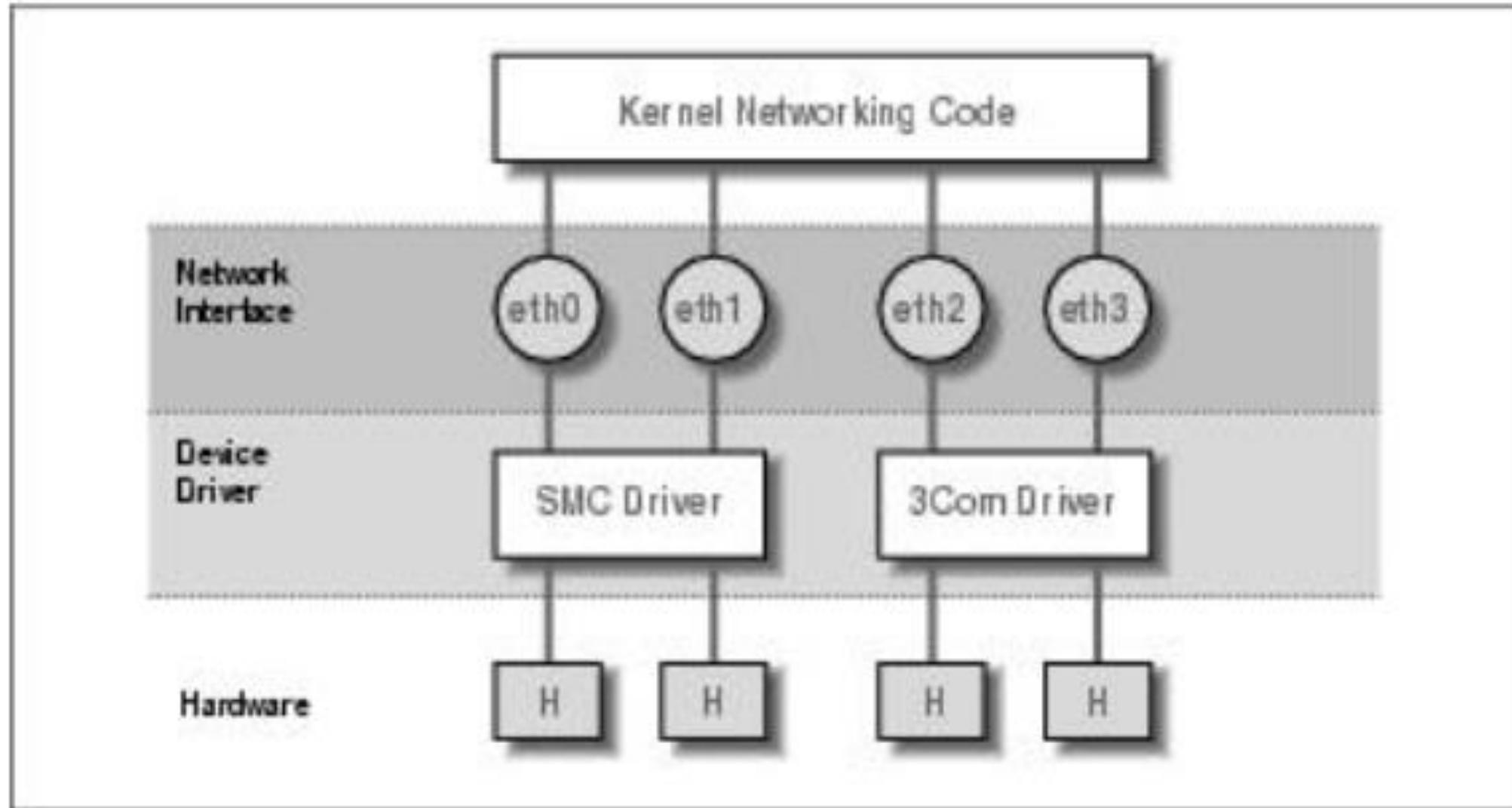


- Drivers help the hardware devices interact with the Linux Kernel.

Role of the Device Driver

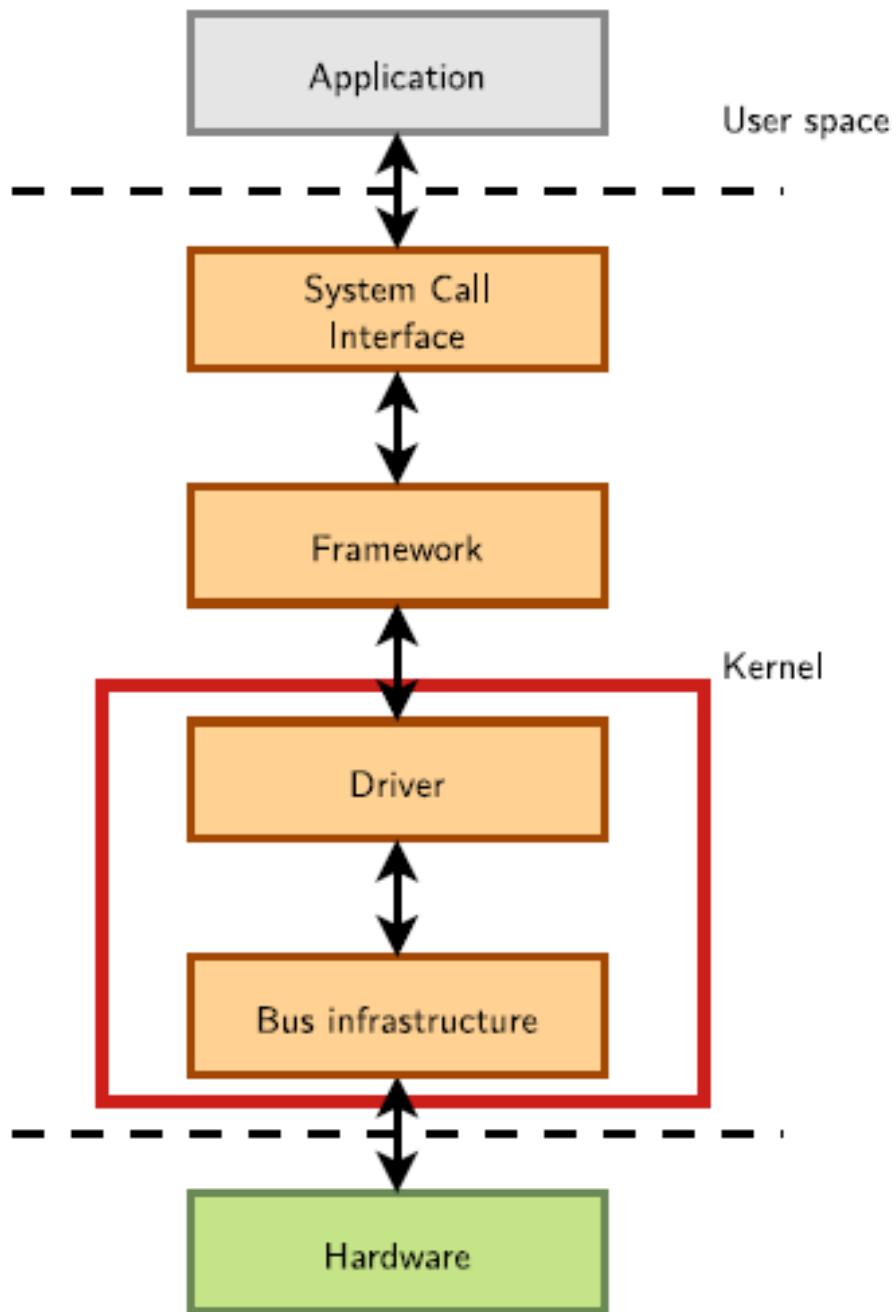
- Implements the *mechanisms* to access the hardware
 - E.g., Hard disk
 - Read/write blocks of data
 - Access as a continuous array
 - Does not force particular *policies* on the user
 - Examples
 - Who and How many access the drive
 - Whether the drive is accessed via a file system
 - Whether users may mount file systems on the drive

The relationship between drivers, interfaces, and hardware



- When booting, the kernel displays the devices it detects and the interfaces it installs.

Kernel & Device Drivers



- In Linux, a driver is always interfacing with:
 - ► a **framework** that allows the driver to expose the hardware features in a generic way.
 - ► a **bus infrastructure**, part of the device model, to detect/communicate with the hardware.

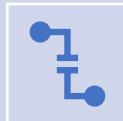
Policy-Free Implementation

- Simplifies the design
- Separation of concerns
 - Capabilities provided
 - Use of Capabilities
- Reuse
 - Different policies do not require changes to the kernel

Splitting the Kernel



Kernel handles resource requests



Process management

Creates, destroys processes

Supports communication among processes

Signals, pipes, etc.

Schedules how processes share the CPU/cores

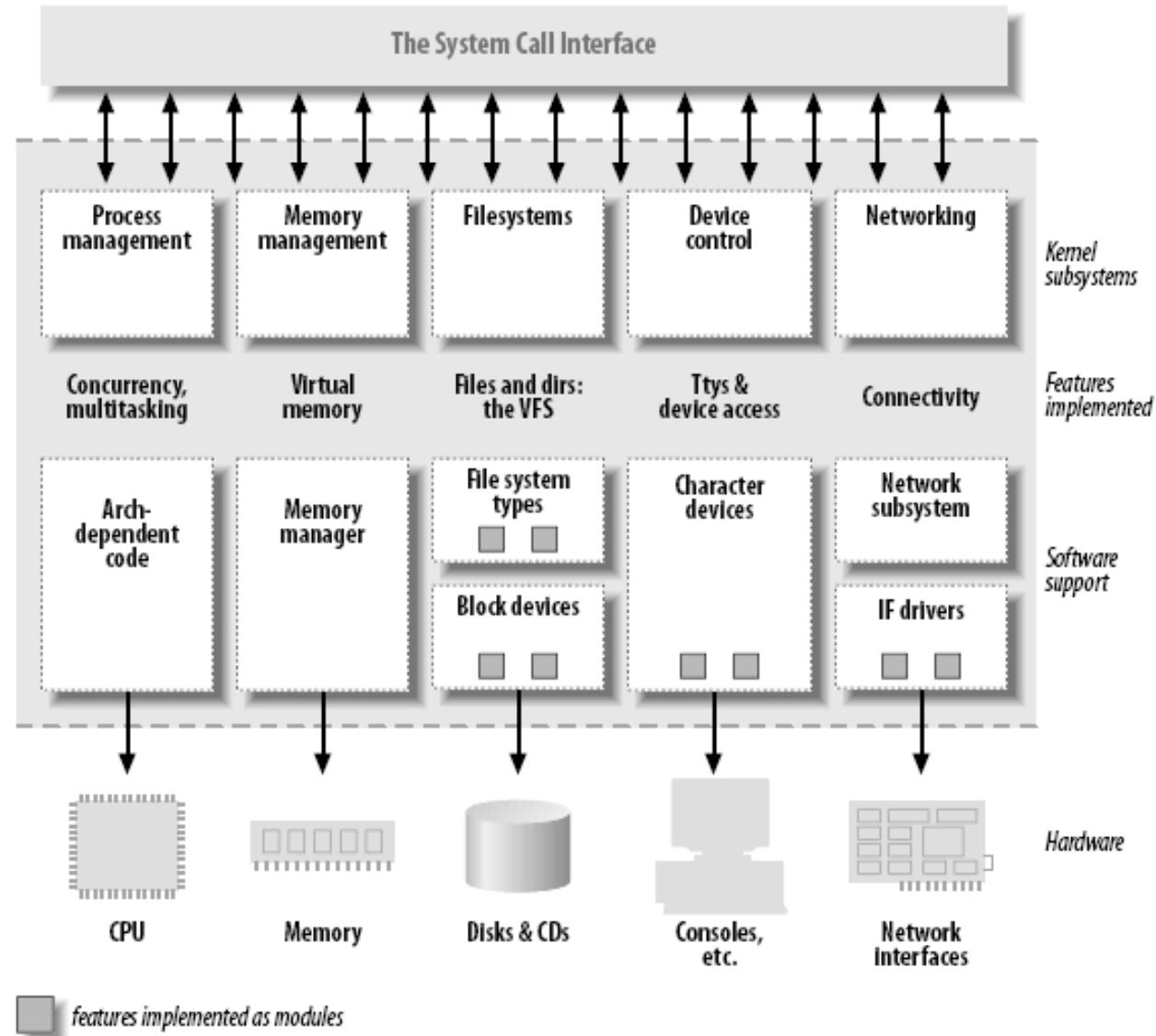


Memory management

Virtual addressing

Splitting the Kernel

- File systems
 - Everything in UNIX can be treated as a file
 - Linux supports multiple file systems
- Device control
 - System operation maps to a physical device
- Networking
 - Handles packets
 - Routing and network address resolution



Hardware Management using Device Drivers



- Any device that the Linux system must communicate with *needs driver code inserted inside the kernel code*.
- The driver code allows the kernel *to pass data back and forth to the device*, acting as a middleman between applications and the hardware.
- Two methods are used for inserting device driver code in the Linux kernel:
 - Drivers compiled in the kernel
 - Driver modules added to the kernel

Hardware Management using Device Drivers

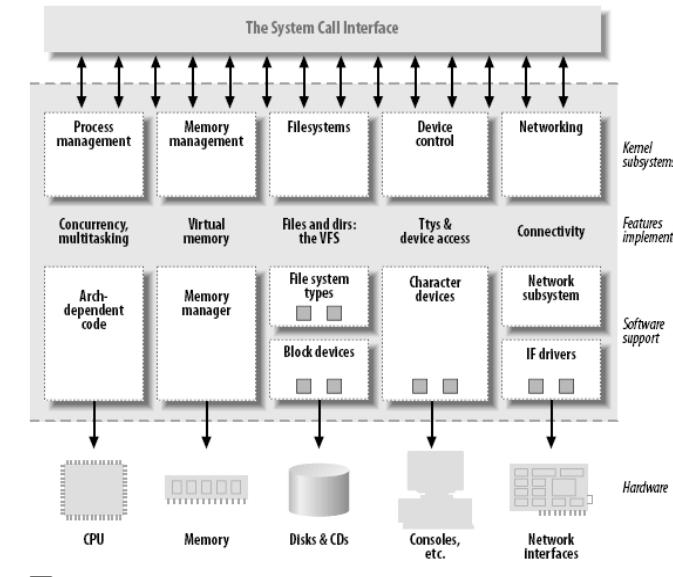
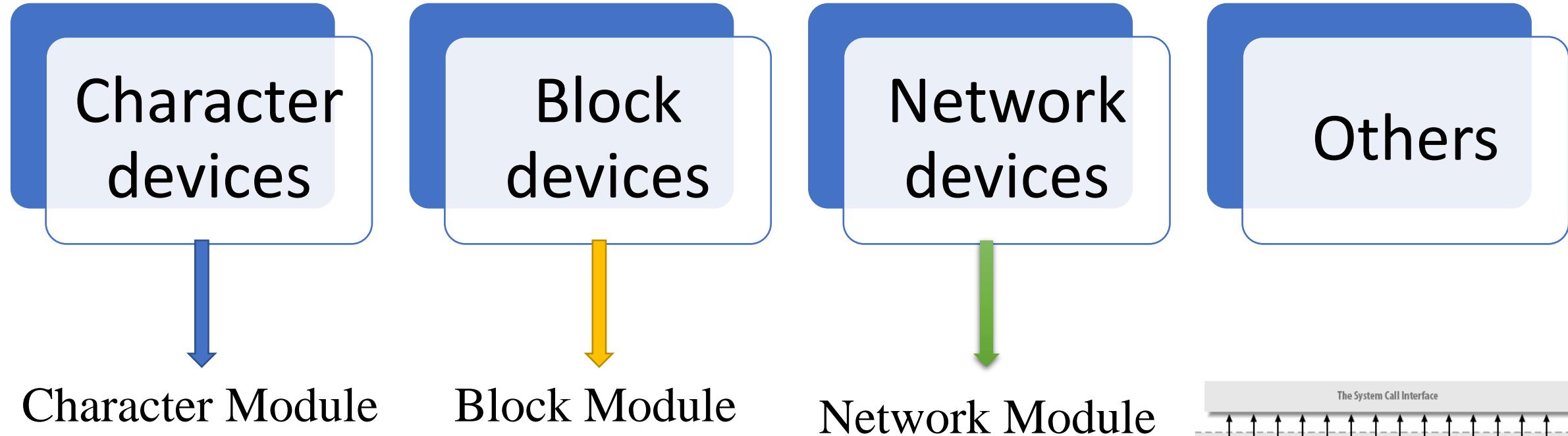
- Previously,
 - The only way to insert device driver code was to recompile the kernel.
 - Each time we added a new device to the system, We had to recompile the kernel code.
 - This process became even more inefficient as Linux kernels supported more hardware. Fortunately,
- Now,
 - Linux developers devised a better method to insert driver code into the running kernel.
 - Programmers developed ***the concept of kernel modules*** to allow you to insert driver code into a running kernel without having to recompile the kernel.
 - Also, a kernel module could be removed from the kernel when the device was finished being used.
 - This greatly simplified and expanded using hardware with Linux.

Runtime Loadable Kernel Modules



- The ability to add and remove kernel features at runtime
- Each unit of extension is called a *module*
- Use **insmod** and **modprobe** program to add a kernel module
- Use **rmmmod** program to remove a kernel module

Classes of Devices and Kernel Modules



Character Devices



Abstraction: a stream of bytes

Examples

- Text console (`/dev/console`)
- Serial ports (`/dev/ttys0`)



Usually supports **open, close, read, write system calls**



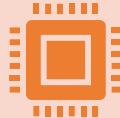
Accessed sequentially (in most cases)

Might not support file seeks



Accessed through a filesystem node

Block Devices



Abstraction: continuous array of storage blocks (e.g. sectors)

Applications can access a block device in bytes



Accessed through a file system node



A block device can host a file system.

A block device can only handle I/O operations that transfer one or more whole blocks, which are usually 512 bytes (or a larger power of two) bytes in length.

Abstraction: data
packets

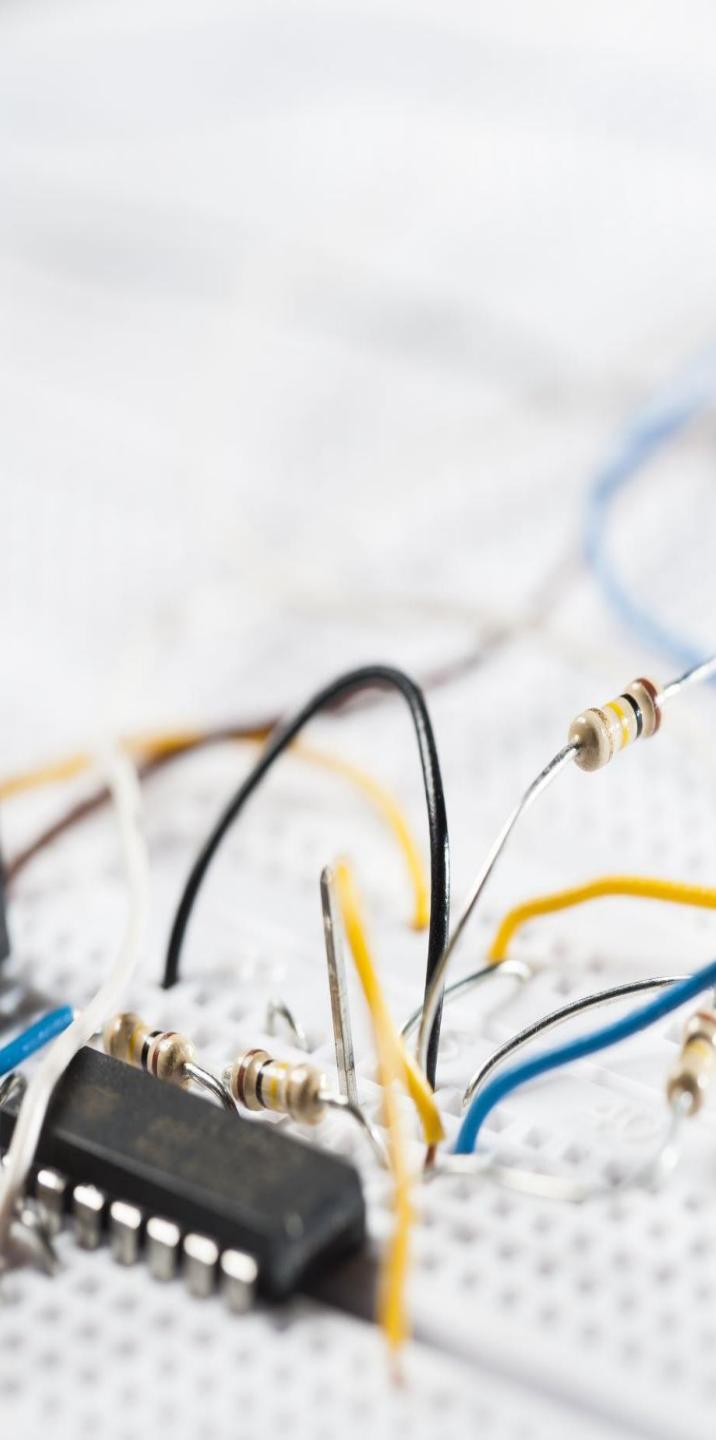
Send and receive
packets

Have unique names
(e.g., eth0)

- Do not know about individual connections
- Not in the file system
- Support protocols and streams related to packet transmission (i.e., no **read** and **write**)

Other Classes of Devices

- Examples that do not fit to previous categories:
 - USB
 - SCSI (Small Comp. System Interface)
 - FireWire
 - MTD (Memory Tech. Device)



Security Issues

- Deliberate vs. incidental damage
- Kernel modules present possibilities for both
- System does only rudimentary checks at module load time
- Relies on limiting privilege to load modules
 - And trusts the driver writers
- Driver writer must be on guard for security problems

Security Issues



- Do not define security policies
 - Provide mechanisms to enforce policies
- Be aware of operations that affect global resources
 - Setting up an interrupt line
 - Could cause another device to malfunction
 - Setting up a default block size
 - Could affect other users

Security Issues

- Buffer overrun
 - Overwriting unrelated data
- Treat input/parameters with utmost suspicion
- Uninitialized memory
 - Kernel memory should be zeroed before being made available to a user
 - Otherwise, information leakage could result
 - Passwords
- Avoid running kernels/device drivers compiled by an untrusted friend
 - Modified kernel could allow anyone to load a module



Building and Running Kernel Modules

- Setting Up Your Test System

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
    printk(KERN_ALERT "Hello, world\n");
    return 0;
}
static void hello_exit(void)
{
    printk(KERN_ALERT "Goodbye, cruel world\n");
}
module_init(hello_init);
module_exit(hello_exit);
```



- ✓ This module defines two functions, one to be invoked when the module is loaded into the kernel (*hello_init*) and one for when the module is removed (*hello_exit*).



Thanks
Q & A



Basics of Socket Programming

Dr. Vimal Baghel
Assistant Professor
SCSET, BU

Outline

- OSI and TCP/IP Models
- Services
- Encapsulation
- Socket Programming
- UDP Socket
- TCP Socket
- Q & A

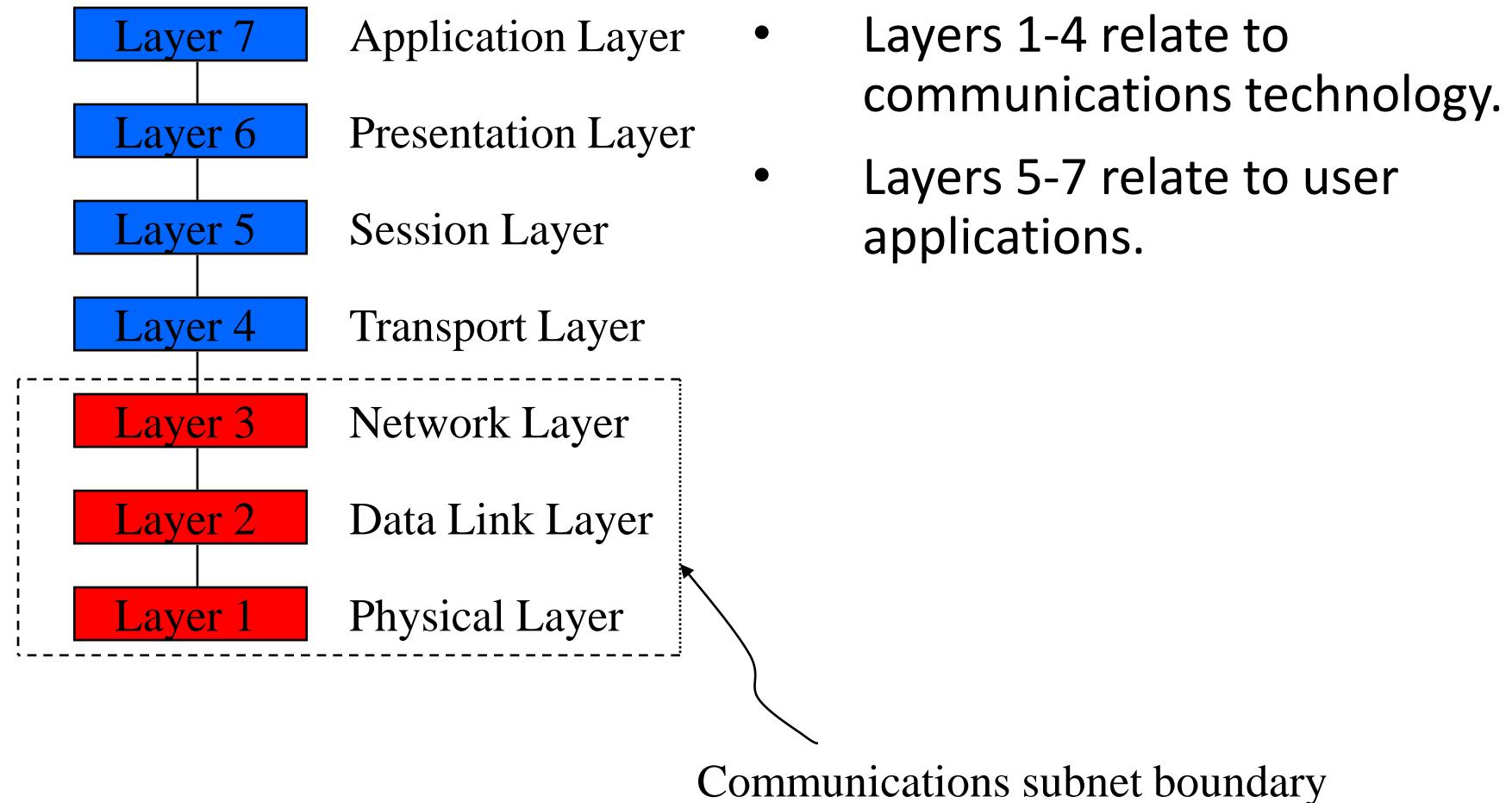
OSI Reference Model

- OSI Reference Model - internationally standardised network architecture.
- *OSI = Open Systems Interconnection:* deals with *open systems*, i.e. systems open for communications with other systems.
- Specified in ISO 7498.
- Model has 7 layers.

OSI History

- In 1978, the International Standards Organization (ISO) began to develop its OSI framework architecture.
- OSI has two major components:
 - an abstract model of networking, called the Basic Reference Model or seven-layer model,
 - a set of specific protocols.
- The concept of a 7-layer model was provided by the work of Charles Bachman, then of Honeywell.
- Various aspects of OSI design evolved from experiences with the Advanced Research Projects Agency Network (ARPANET) and the fledgling Internet.

7-Layer OSI Model



Layer 7: Application Layer

- Level at which applications access network services.
 - Represents services that directly support software applications for file transfers, database access, and electronic mail etc.

Layer 6: Presentation Layer

- Related to representation of transmitted data
 - Translates different data representations from the Application layer into uniform standard format
- Providing services for secure efficient data transmission
 - e.g. data encryption, and data compression.

Layer 5: Session Layer

- Allows two applications on different computers to establish, use, and end a session.
 - e.g. file transfer, remote login
- Establishes dialog control
 - Regulates which side transmits, plus when and how long it transmits.
- **Performs *token management* and *synchronization*.**

Layer 4: Transport Layer

- Manages transmission packets
 - Repackages long messages when necessary into small packets for transmission
 - Reassembles packets in correct order to get the original message.
- Handles error recognition and recovery.
 - Transport layer at receiving acknowledges packet delivery.
 - Resends missing packets

Layer 3: Network Layer

- Manages addressing/routing of data within the subnet
 - Addresses messages and translates logical addresses and names into physical addresses.
 - **Determines the route from the source to the destination computer**
 - Manages traffic problems, such as switching, routing, and controlling the congestion of data packets.
- **Routing can be:**
 - Based on static tables
 - determined at start of each session
 - Individually determined for each packet, reflecting the current network load.

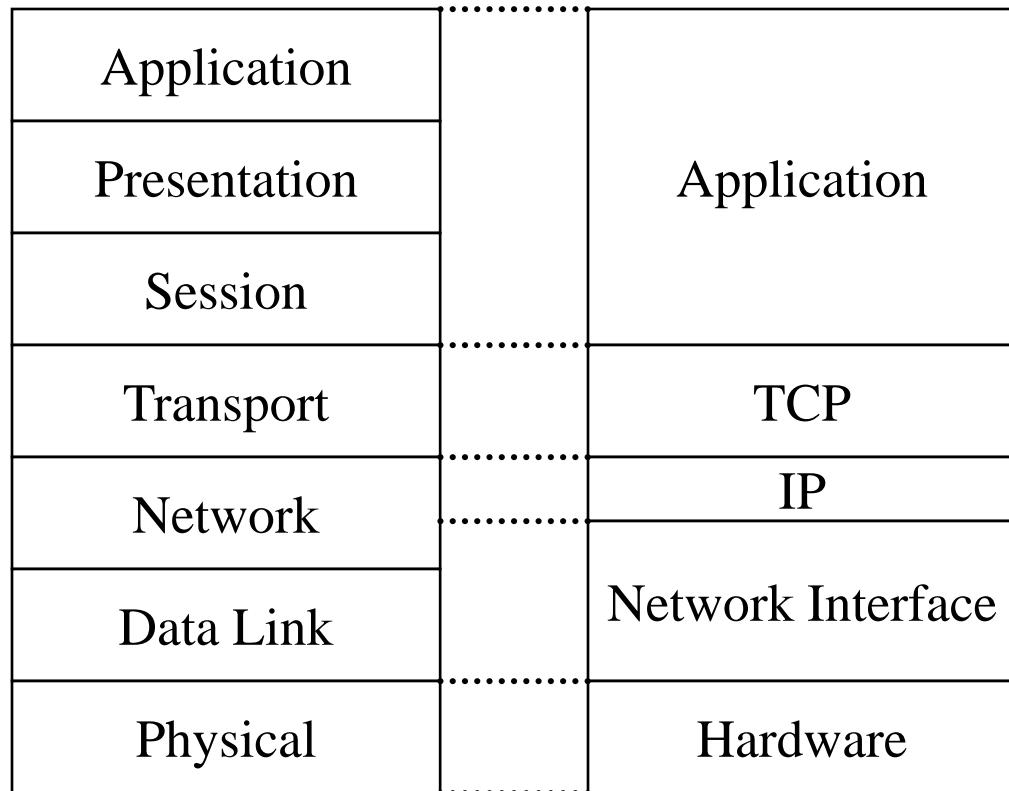
Layer 2: Data Link Layer

- Packages raw bits from the Physical layer into frames (logical, structured packets for data).
- Provides reliable transmission of frames
 - It waits for an acknowledgment from the receiving computer.
 - Retransmits frames for which acknowledgement not received

Layer 1: Physical Layer

- Transmits bits from one computer to another
- Regulates the transmission of a stream of bits over a physical medium.
- Defines how the cable is attached to the network adapter and what transmission technique is used to send data over the cable.
Deals with issues like
 - The definition of 0 and 1, e.g. how many volts represents a 1, and how long a bit lasts?
 - Whether the channel is simplex or duplex?
 - How many pins a connector has, and what the function of each pin is?

Internet Protocols vs OSI

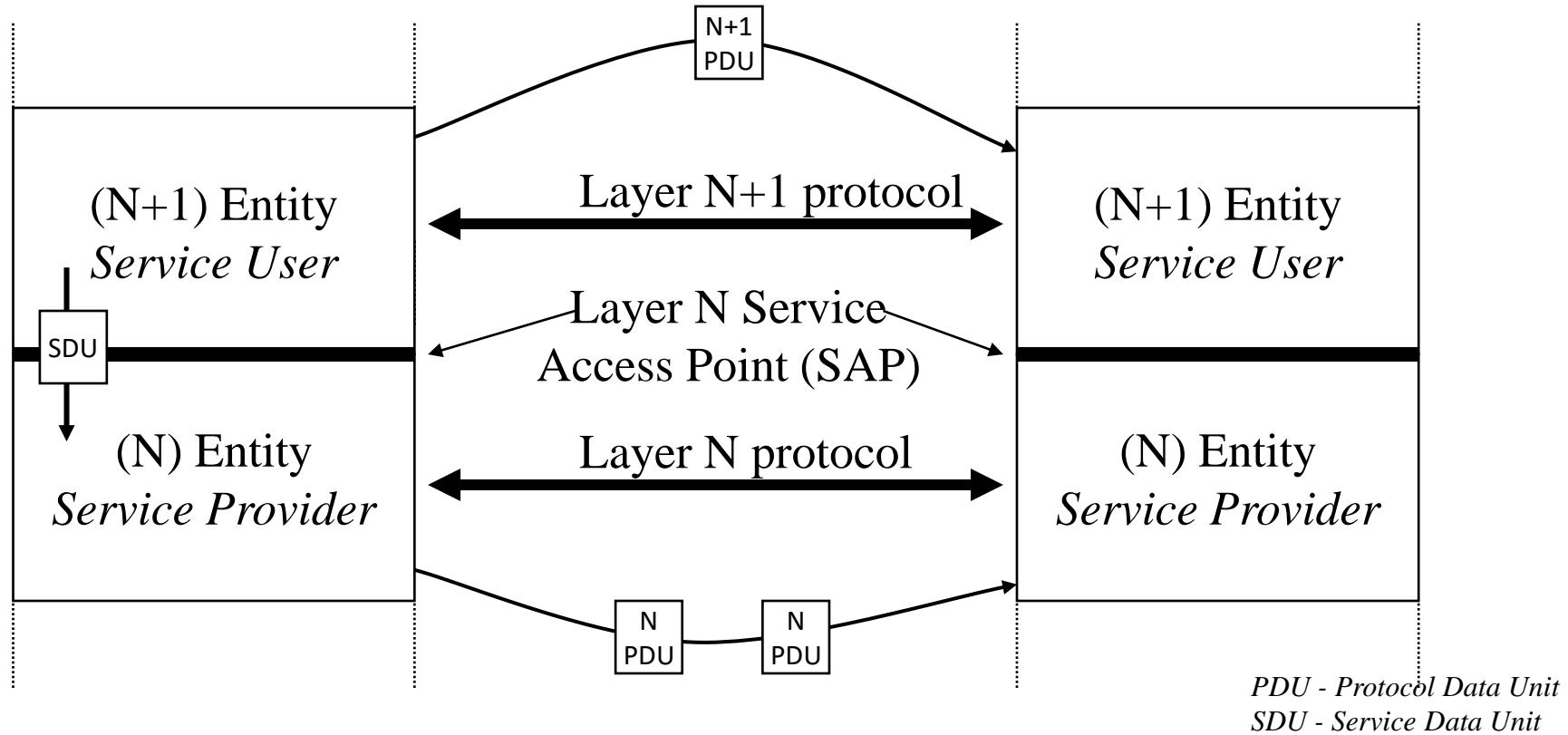


- Explicit Presentation and session layers missing in Internet Protocols
- Data Link and Network Layers redesigned

Services in the OSI Model

- In OSI model, *each layer provide services to layer above, and ‘consumes’ services provided by layer below.*
- Active elements in a layer called *entities*.
- Entities in same layer in different machines called *peer entities*.

Layering Principles



- Layer N provides service to layer N+1

Connections

- Layers can offer *connection-oriented* or *connectionless services*.
- Connection-oriented like telephone system.
- Connectionless like postal system.
- Each service has an associated *Quality-of-service* (e.g. reliable or unreliable).

Reliability

- Reliable services *never lose/corrupt data.*
- Reliable service *costs more.*
- Typical application for reliable service is file transfer.
- Typical application not needing reliable service is voice traffic.
- Not all applications need connections.

Topics

- Service = set of primitives provided by one layer to layer above.
- Service defines what layer can do (but not how it does it).
- Protocol = *set of rules governing data communication between peer entities, i.e. format and meaning of frames/packets.*
- Service/protocol decoupling very important.

TCP/IP Model

- Protocol Layers
- Services
- Encapsulation

Protocol “layers” and reference models

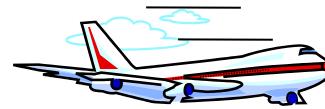
Networks are complex,
with many “pieces”:

- hosts
- routers
- links of various media
- applications
- protocols
- hardware, software

Question: is there any
hope of *organizing*
structure of network?

- and/or our *discussion*
of networks?

Example: organization of air travel



— *end-to-end transfer of person plus baggage* —→

ticket (purchase)

baggage (check)

gates (load)

runway takeoff

airplane routing

ticket (complain)

baggage (claim)

gates (unload)

runway landing

airplane routing

airplane routing

How would you *define/discuss* the *system* of airline travel?

- a series of steps, involving many services

Example: organization of air travel



layers: each layer implements a service

- via its own internal-layer actions
- relying on services provided by layer below

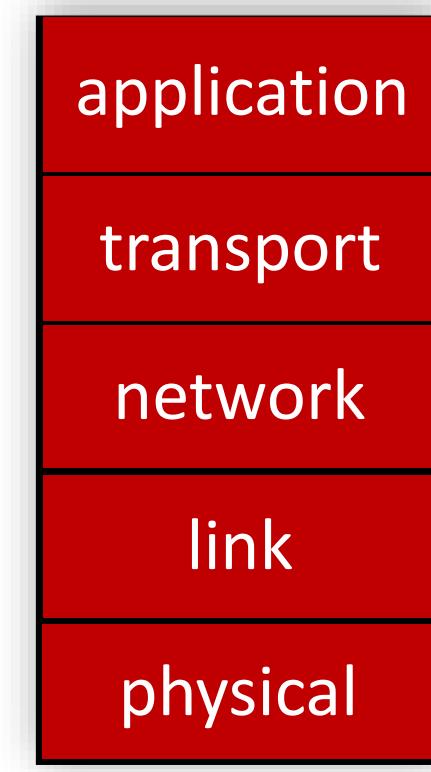
Why layering?

Approach to designing/discussing complex systems:

- explicit structure allows identification, relationship of system's pieces
 - layered *reference model* for discussion
- modularization eases maintenance, updating of system
 - change in layer's service *implementation*: transparent to rest of system
 - e.g., change in gate procedure doesn't affect rest of system

Layered Internet protocol stack

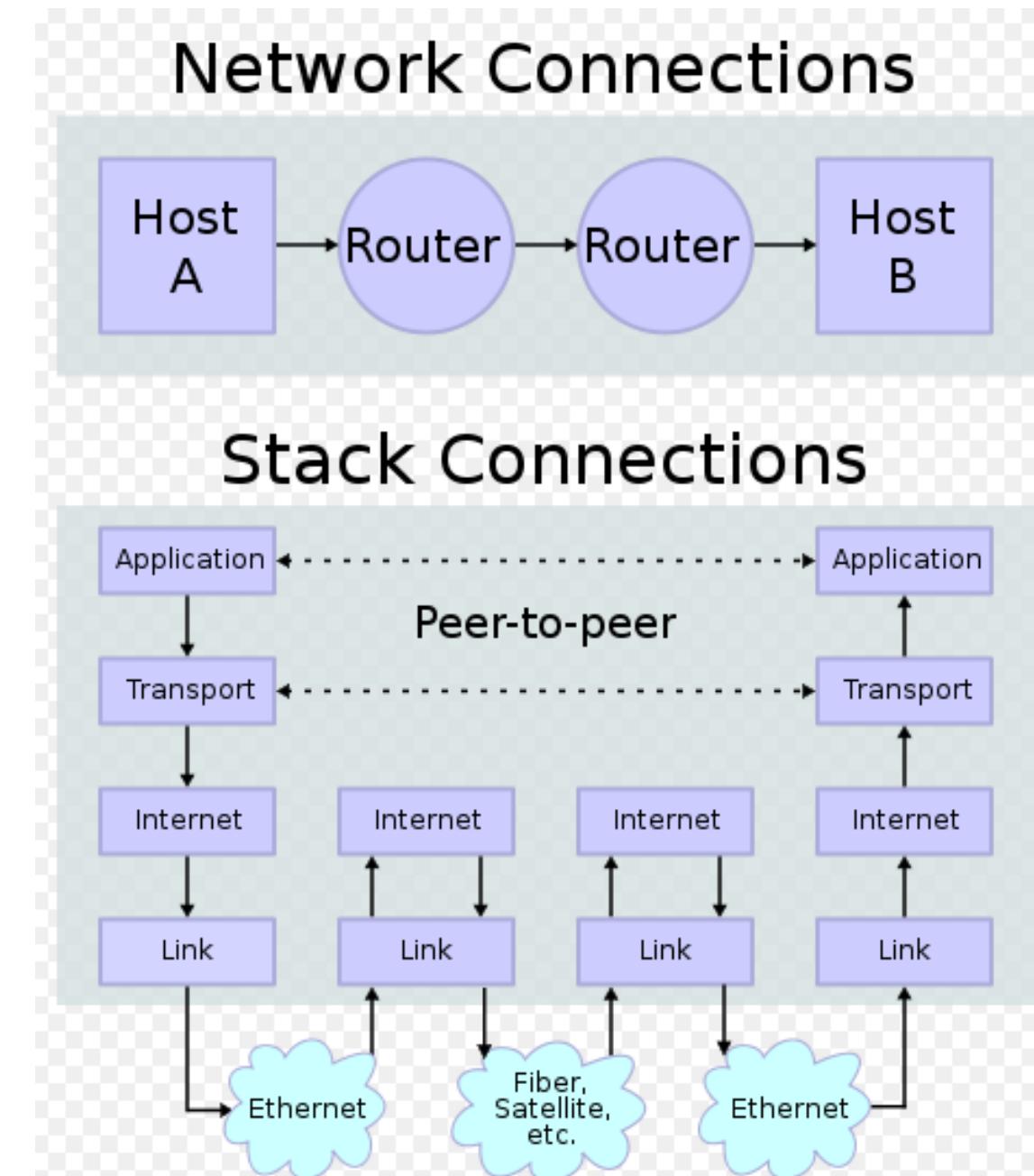
- *application*: supporting network applications
 - HTTP, IMAP, SMTP, DNS
- *transport*: process-process data transfer
 - TCP, UDP
- *network*: routing of datagrams from source to destination
 - IP, routing protocols
- *link*: data transfer between neighboring network elements
 - Ethernet, 802.11 (WiFi), PPP
- *physical*: bits “on the wire”



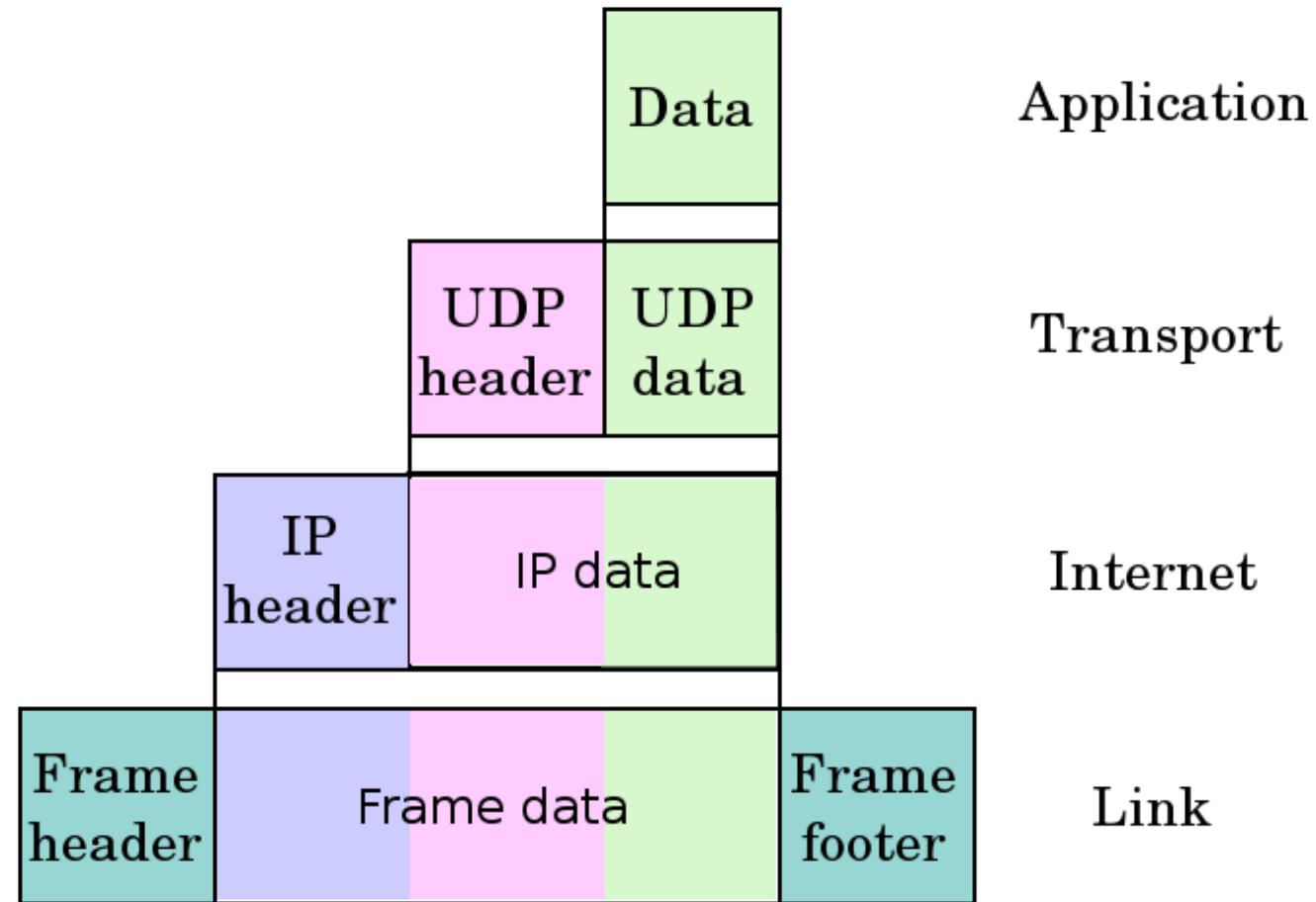
TCP/IP Layers

OSI	TCP/IP
Application Layer	Application Layer
Presentation Layer	TELNET, FTP, SMTP, POP3, SNMP, NNTP, DNS, NIS, NFS, HTTP, ...
Session Layer	
Transport Layer	Transport Layer TCP, UDP, ...
Network Layer	Internet Layer IP, ICMP, ARP, RARP, ...
Data Link Layer	Link Layer
Physical Layer	FDDI, Ethernet, ISDN, X.25, ...

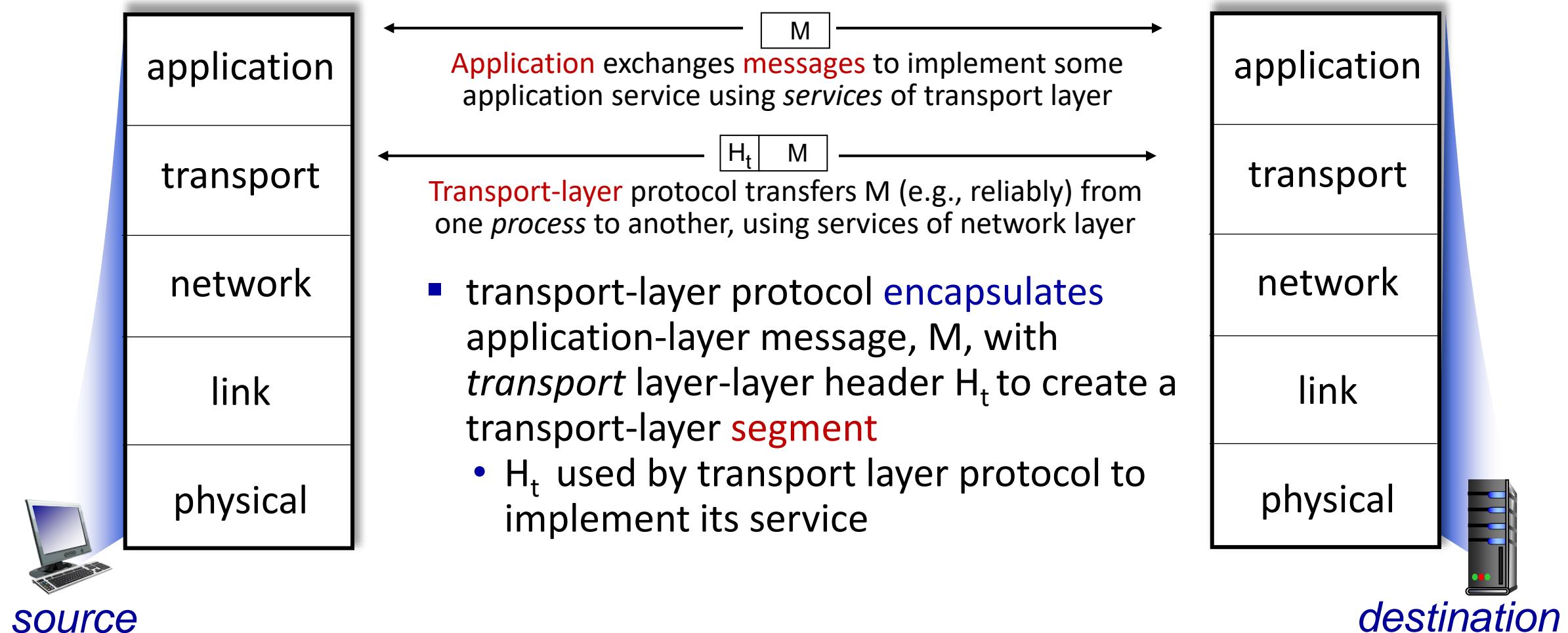
TCP/IP Stack



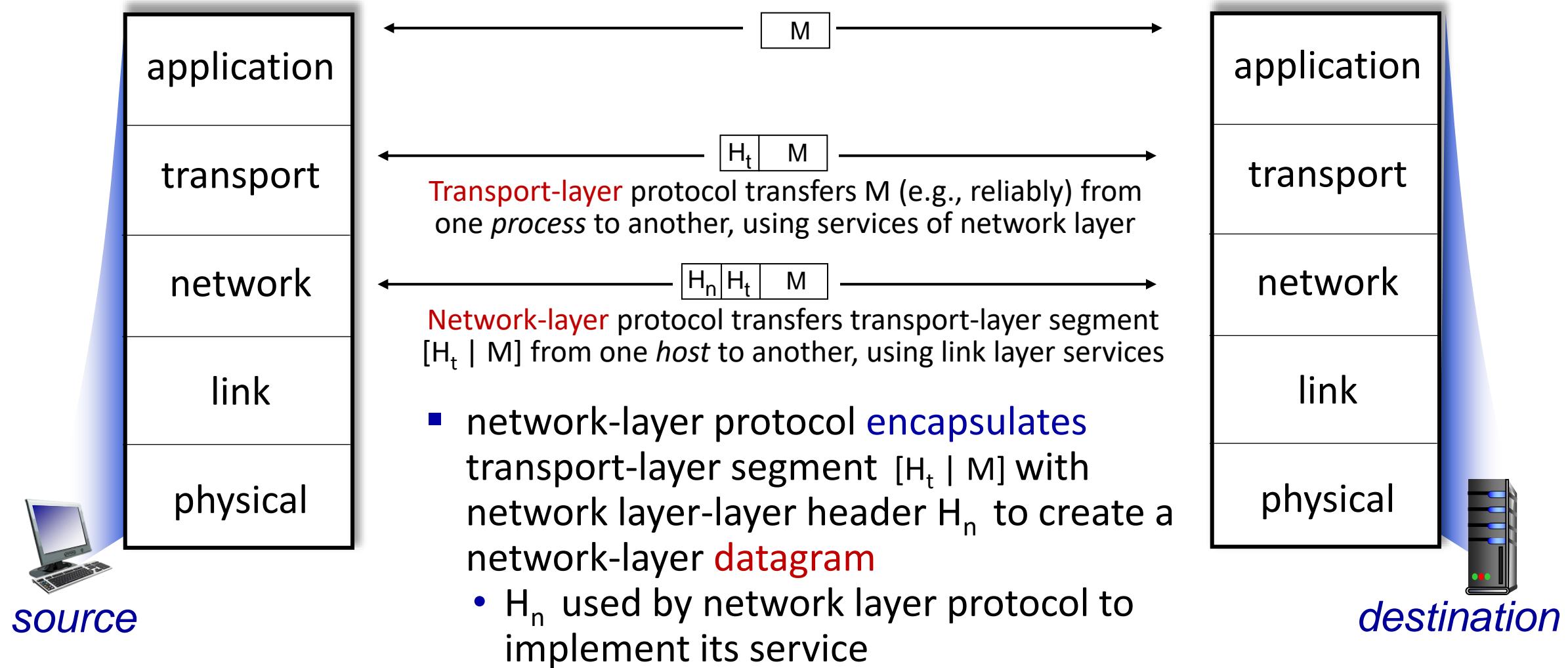
TCP/IP Encapsulation



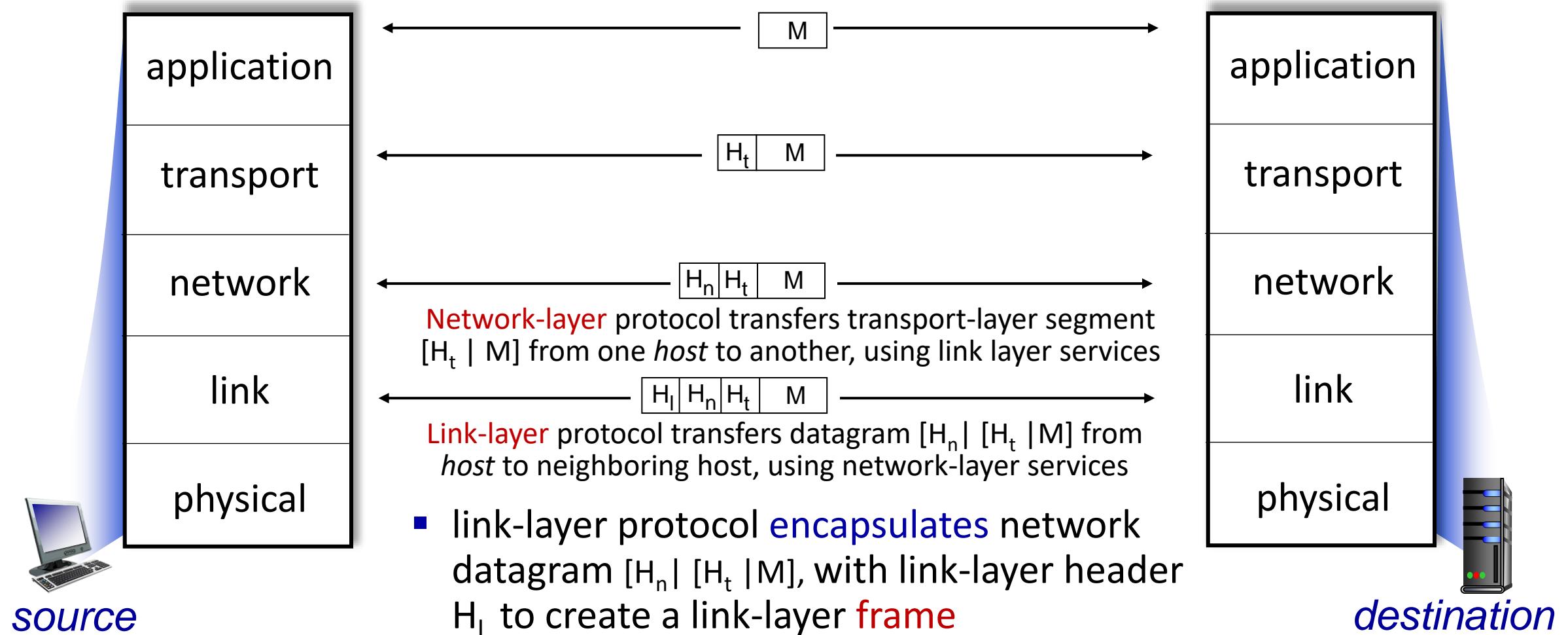
Services, Layering and Encapsulation



Services, Layering and Encapsulation

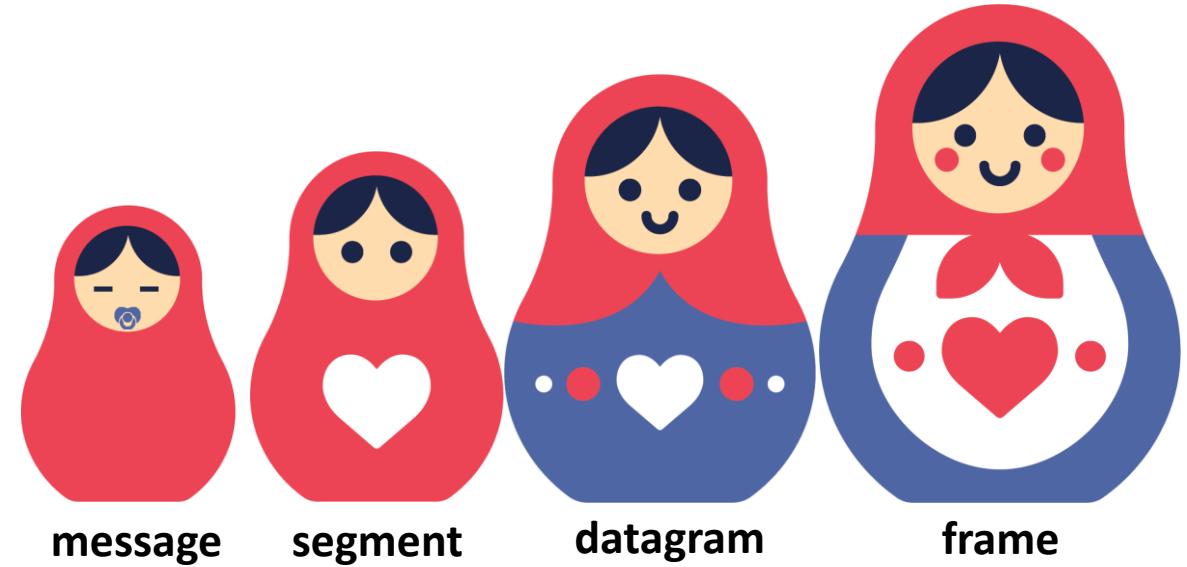


Services, Layering and Encapsulation

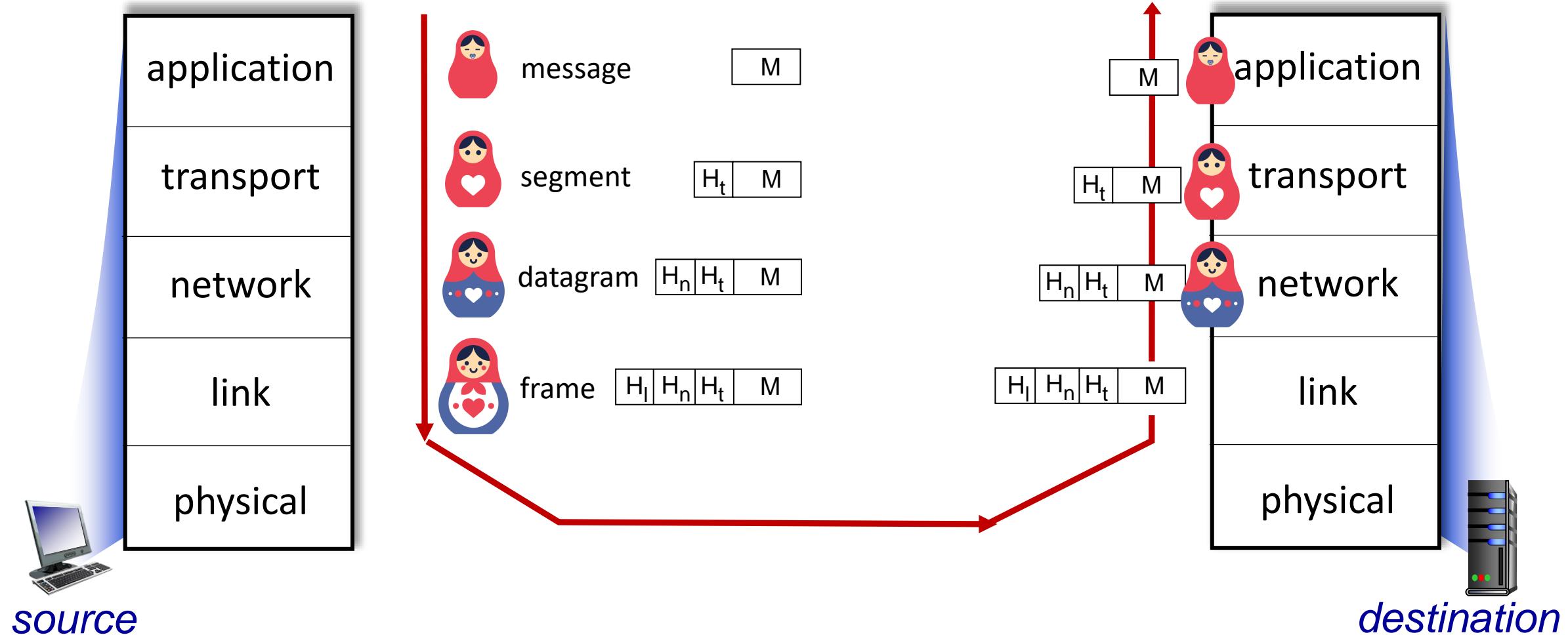


Encapsulation

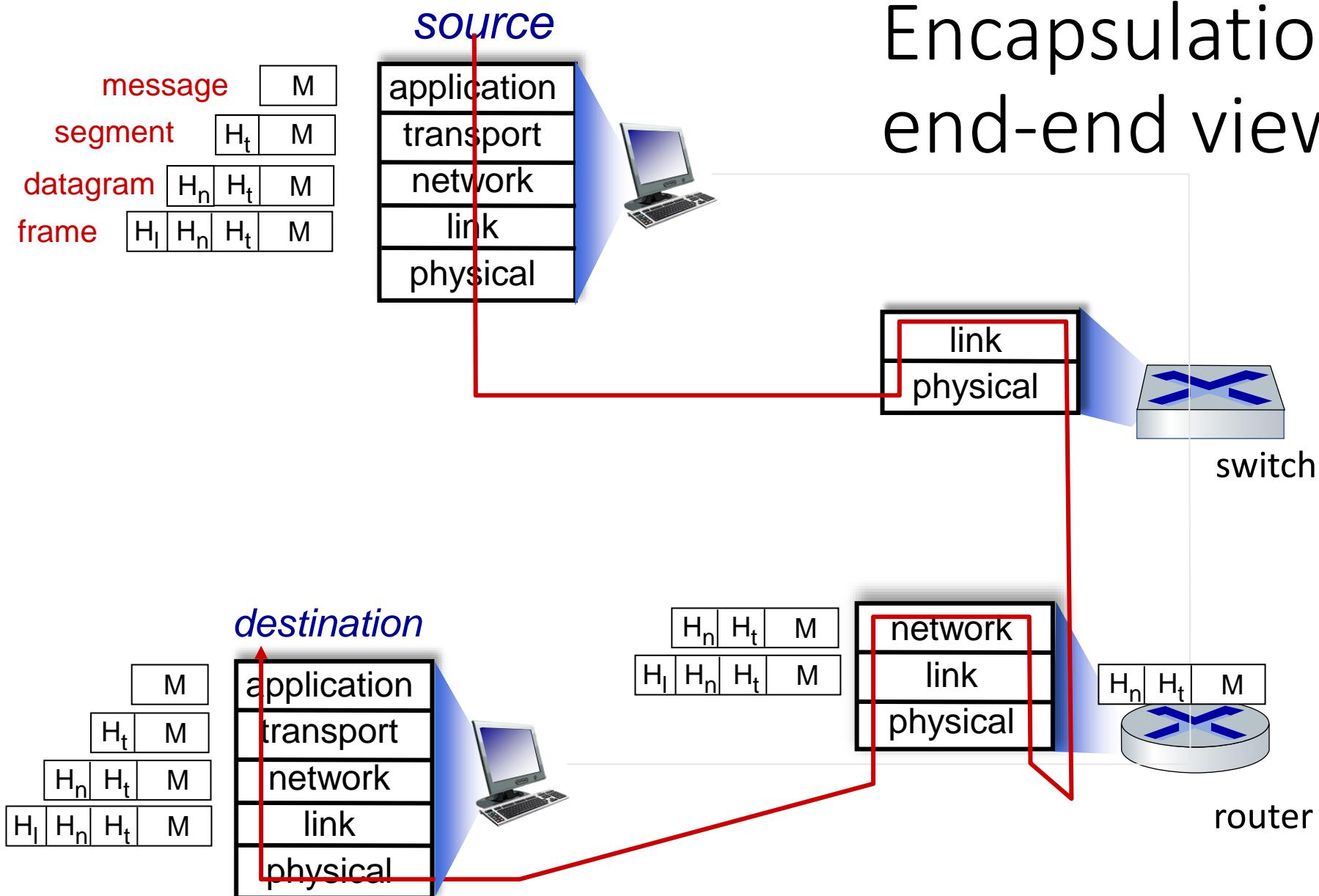
Matryoshka dolls (stacking dolls)



Services, Layering and Encapsulation



Encapsulation: an end-end view



TCP/IP Some Protocol

Layer	Protocol
<u>Application</u>	DNS , TFTP , TLS/SSL , FTP , Gopher , HTTP , IMAP , IRC , NNTP , POP3 , SIP , SMTP , SMPP , SNMP , SSH , Telnet , Echo , RTP , PNRP , rlogin , ENRP
	Routing protocols like BGP and RIP which run over TCP/UDP, may also be considered part of the Internet Layer.
<u>Transport</u>	TCP , UDP , DCCP , SCTP , IL , RUDP , RSVP
<u>Internet</u>	IP (IPv4, IPv6) , ICMP , IGMP , and ICMPv6 OSPF for IPv4 was initially considered IP layer protocol since it runs per IP-subnet, but has been placed on the Link since RFC 2740 .
<u>Link</u>	ARP , RARP , OSPF (IPv4/IPv6) , IS-IS , NDP

Application layer: overview

- Principles of network applications
- Web and HTTP
- E-mail, SMTP, IMAP
- The Domain Name System DNS
- P2P applications
- video streaming and content distribution networks
- socket programming with UDP and TCP



Application layer: overview

Our goals:

- conceptual *and* implementation aspects of application-layer protocols
 - transport-layer service models
 - client-server paradigm
 - peer-to-peer paradigm
- learn about protocols by examining popular application-layer protocols and infrastructure
 - HTTP
 - SMTP, IMAP
 - DNS
 - video streaming systems, CDNs
- programming network applications
 - socket API

Some network apps

- social networking
- Web
- text messaging
- e-mail
- multi-user network games
- streaming stored video
(YouTube, Hulu, Netflix)
- P2P file sharing
- voice over IP (e.g., Skype)
- real-time video conferencing
(e.g., Zoom)
- Internet search
- remote login
- ...

Q: your favorites?

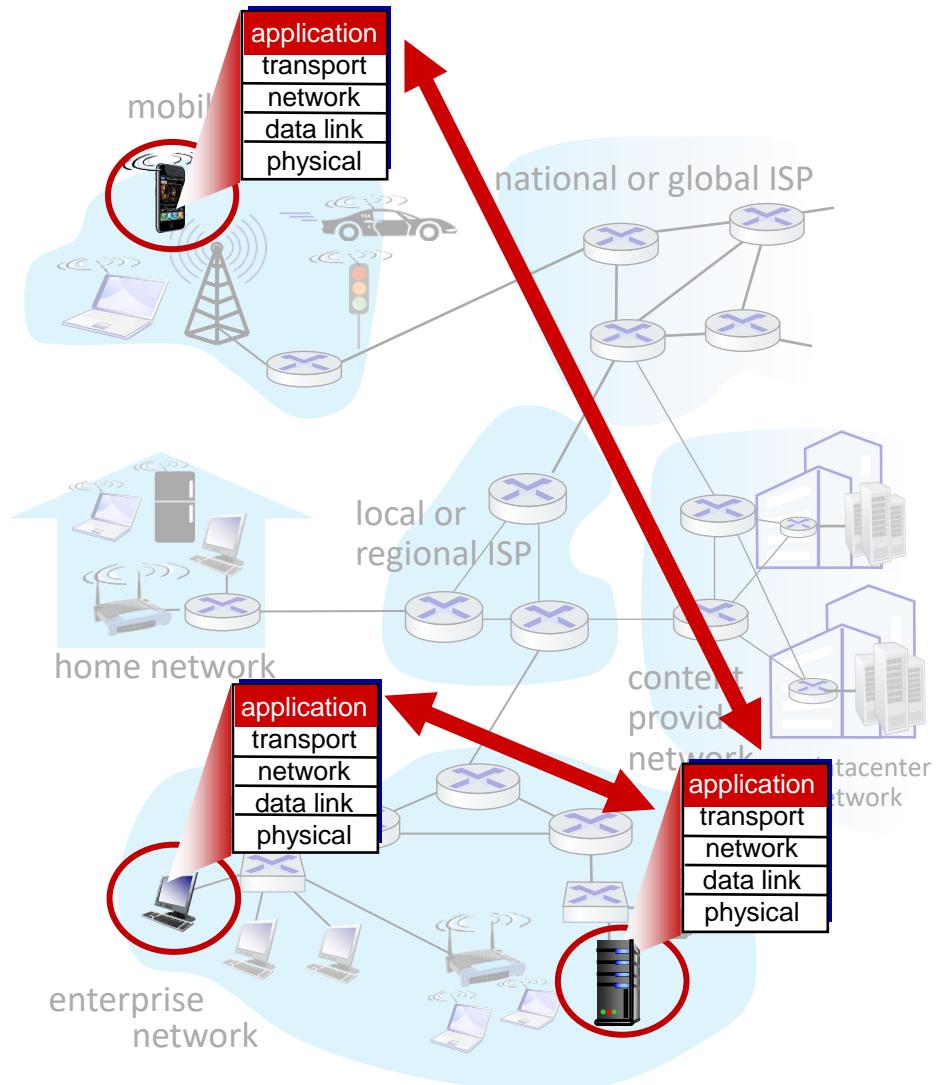
Creating a network app

write programs that:

- run on (different) end systems
- communicate over network
- e.g., web server software
communicates with browser software

**no need to write software for
network-core devices**

- network-core devices do not run user applications
- applications on end systems allows for rapid app development, propagation



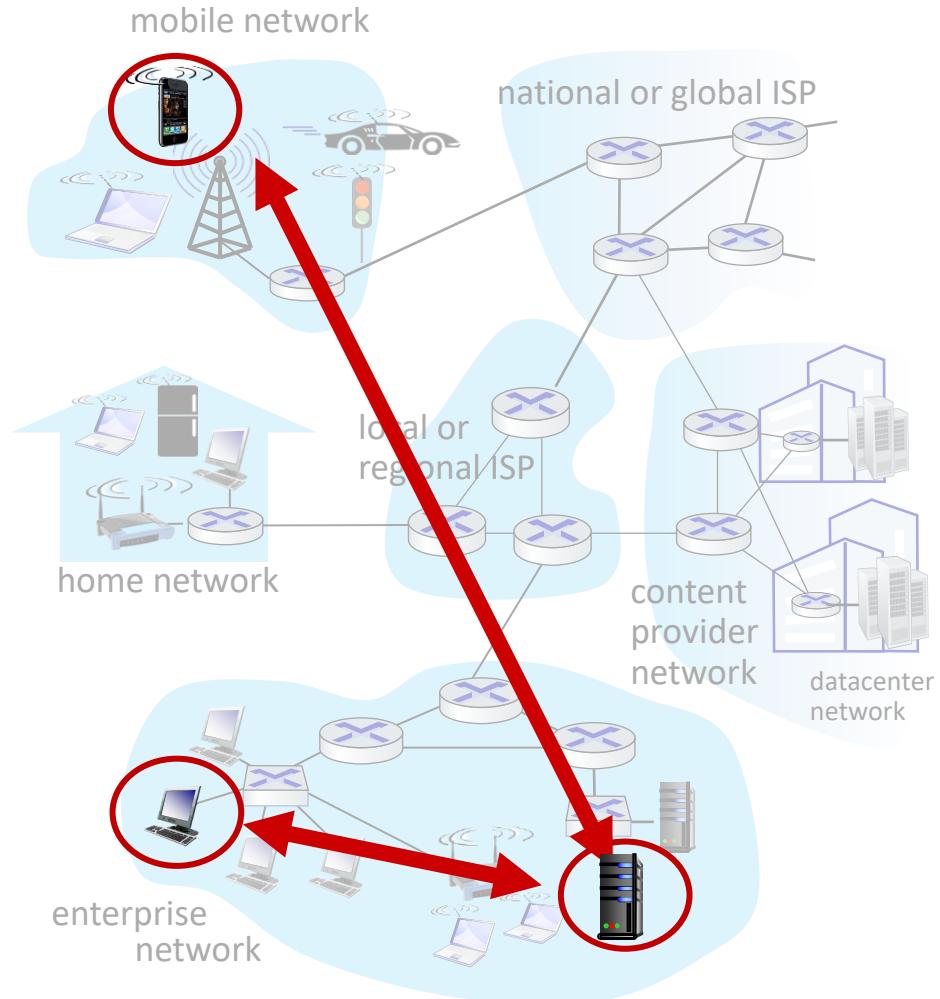
Client-server paradigm

server:

- always-on host
- permanent IP address
- often in data centers, for scaling

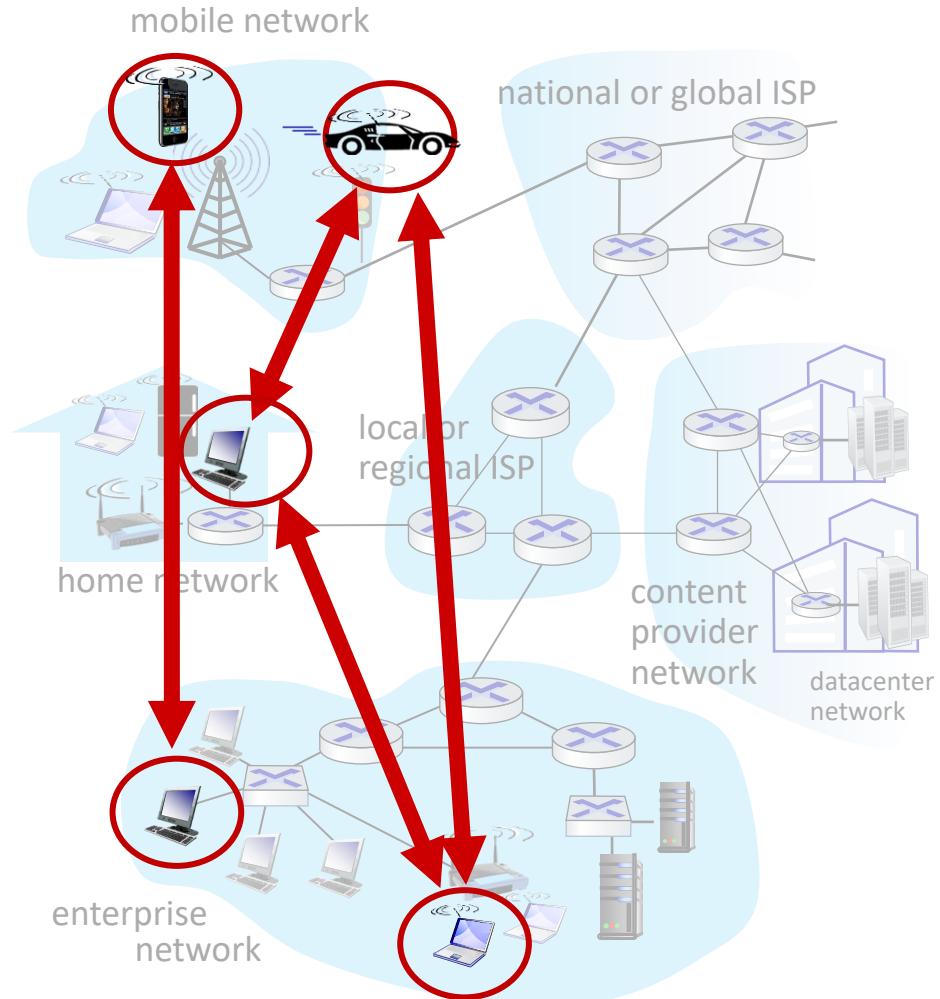
clients:

- contact, communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do *not* communicate directly with each other
- examples: HTTP, IMAP, FTP



Peer-peer architecture

- no always-on server
- arbitrary end systems directly communicate
- peers request service from other peers, provide service in return to other peers
 - *self scalability* – new peers bring new service capacity, as well as new service demands
- peers are intermittently connected and change IP addresses
 - complex management
- example: P2P file sharing [BitTorrent]



Processes communicating

- process*: program running within a host
- within same host, two processes communicate using **inter-process communication** (defined by OS)
 - processes in different hosts communicate by exchanging **messages**

clients, servers

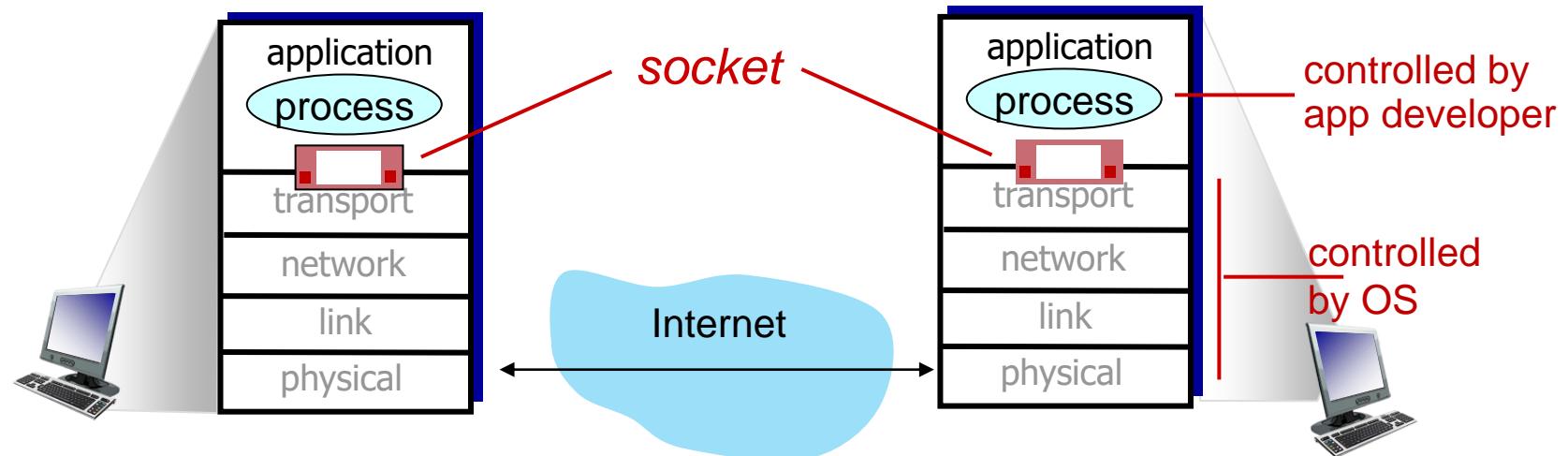
client process: process that initiates communication

server process: process that waits to be contacted

- note: applications with P2P architectures have client processes & server processes

Sockets

- process sends/receives messages to/from its **socket**
- socket analogous to door
 - sending process shoves message out door
 - sending process relies on transport infrastructure on other side of door to deliver message to socket at receiving process
 - two sockets involved: one on each side



Addressing processes

- to receive messages, process must have *identifier*
- host device has unique 32-bit IP address
- Q: does IP address of host on which process runs suffice for identifying the process?
 - A: no, *many* processes can be running on same host
- *identifier* includes both **IP address** and **port numbers** associated with process on host.
- example port numbers:
 - HTTP server: 80
 - mail server: 25
- to send HTTP message to gaia.cs.umass.edu web server:
 - **IP address:** 128.119.245.12
 - **port number:** 80
- more shortly...

An application-layer protocol defines:

- types of messages exchanged,
 - e.g., request, response
- message syntax:
 - what fields in messages & how fields are delineated
- message semantics
 - meaning of information in fields
- rules for when and how processes send & respond to messages

open protocols:

- defined in RFCs, everyone has access to protocol definition
- allows for interoperability
- e.g., HTTP, SMTP

proprietary protocols:

- e.g., Skype, Zoom

What transport service does an app need?

data integrity

- some apps (e.g., file transfer, web transactions) require 100% reliable data transfer
- other apps (e.g., audio) can tolerate some loss

throughput

- some apps (e.g., multimedia) require minimum amount of throughput to be “effective”
- other apps (“elastic apps”) make use of whatever throughput they get

timing

- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”

security

- encryption, data integrity, ...

Transport service requirements: common apps

application	data loss	throughput	time sensitive?
file transfer/download	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	no loss	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kbps-1Mbps video:10Kbps-5Mbps	yes, 10's msec
streaming audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	Kbps+	yes, 10's msec
text messaging	no loss	elastic	yes and no

Internet transport protocols services

TCP service:

- ***reliable transport*** between sending and receiving process
- ***flow control***: sender won't overwhelm receiver
- ***congestion control***: throttle sender when network overloaded
- ***connection-oriented***: setup required between client and server processes
- ***does not provide***: timing, minimum throughput guarantee, security

UDP service:

- ***unreliable data transfer*** between sending and receiving process
- ***does not provide***: reliability, flow control, congestion control, timing, throughput guarantee, security, or connection setup.

Q: why bother? *Why* is there a UDP?

Internet applications, and transport protocols

application	application layer protocol	transport protocol
file transfer/download	FTP [RFC 959]	TCP
e-mail	SMTP [RFC 5321]	TCP
Web documents	HTTP [RFC 7230, 9110]	TCP
Internet telephony	SIP [RFC 3261], RTP [RFC 3550], or proprietary	TCP or UDP
streaming audio/video	HTTP [RFC 7230], DASH	TCP
interactive games	WOW, FPS (proprietary)	UDP or TCP

Securing TCP

Vanilla TCP & UDP sockets:

- no encryption
- cleartext passwords sent into socket traverse Internet in cleartext (!)

Transport Layer Security (TLS)

- provides encrypted TCP connections
- data integrity
- end-point authentication

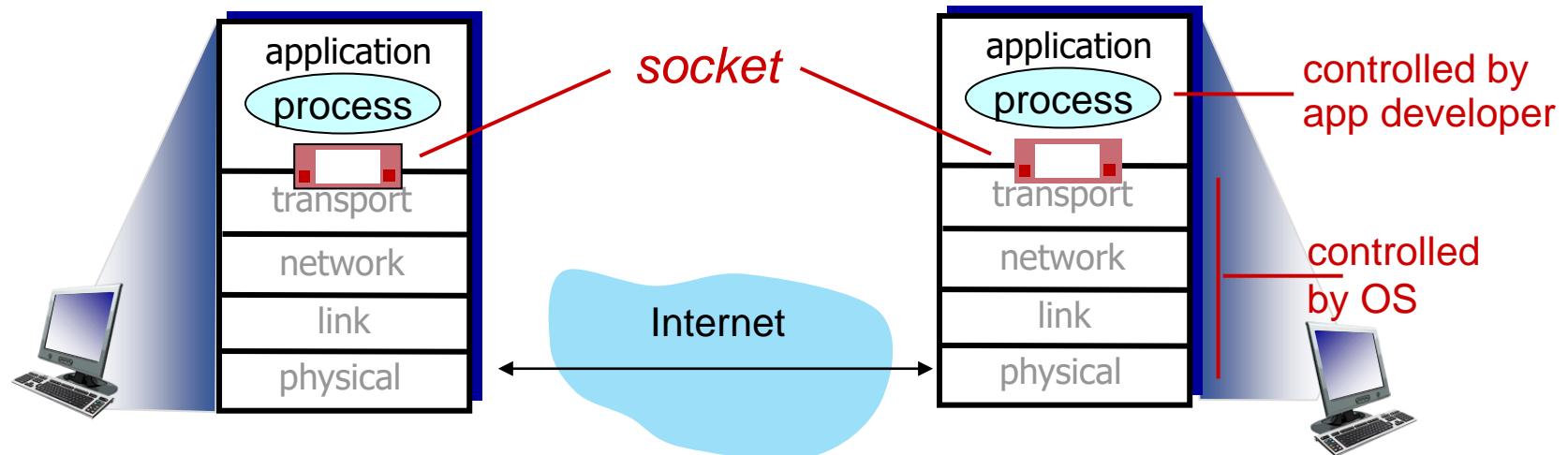
TLS implemented in application layer

- apps use TLS libraries, that use TCP in turn
- cleartext sent into “socket” traverse Internet *encrypted*
- more: Chapter 8

Socket programming

goal: learn how to build client/server applications that communicate using sockets

socket: door between application process and end-end-transport protocol



Socket programming

Two socket types for two transport services:

- *UDP*: unreliable datagram
- *TCP*: reliable, byte stream-oriented

Application Example:

1. client reads a line of characters (data) from its keyboard and sends data to server
2. server receives the data and converts characters to uppercase
3. server sends modified data to client
4. client receives modified data and displays line on its screen

Socket programming with UDP

UDP: no “connection” between client and server:

- no handshaking before sending data
- sender explicitly attaches IP destination address and port # to each packet
- receiver extracts sender IP address and port# from received packet

UDP: transmitted data may be lost or received out-of-order

Application viewpoint:

- UDP provides *unreliable* transfer of groups of bytes (“datagrams”) between client and server processes

Client/server socket interaction: UDP



server (running on serverIP)

```
create socket, port= x:  
serverSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
read datagram from  
serverSocket
```

```
write reply to  
serverSocket  
specifying  
client address,  
port number
```



client

```
create socket:  
clientSocket =  
socket(AF_INET,SOCK_DGRAM)
```

```
Create datagram with serverIP address  
And port=x; send datagram via  
clientSocket
```

```
read datagram from  
clientSocket  
close  
clientSocket
```

Example app: UDP client

Python UDPCClient

```
include Python's socket library → from socket import *
serverName = 'hostname'
serverPort = 12000
create UDP socket → clientSocket = socket(AF_INET,
                                             SOCK_DGRAM)
get user keyboard input → message = input('Input lowercase sentence:')
attach server name, port to message; send into socket → clientSocket.sendto(message.encode(),
                           (serverName, serverPort))
read reply data (bytes) from socket → modifiedMessage, serverAddress =
                                         clientSocket.recvfrom(2048)
print out received string and close socket → print(modifiedMessage.decode())
                                                clientSocket.close()
```

Example app: UDP server

Python UDPServer

```
from socket import *
serverPort = 12000
create UDP socket → serverSocket = socket(AF_INET, SOCK_DGRAM)
bind socket to local port number 12000 → serverSocket.bind(('', serverPort))
                                         print('The server is ready to receive')
loop forever → while True:
Read from UDP socket into message, getting →   message, clientAddress = serverSocket.recvfrom(2048)
client's address (client IP and port)           modifiedMessage = message.decode().upper()
                                               serverSocket.sendto(modifiedMessage.encode(),
send upper case string back to this client →   clientAddress)
```

Socket programming with TCP

Client must contact server

- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:

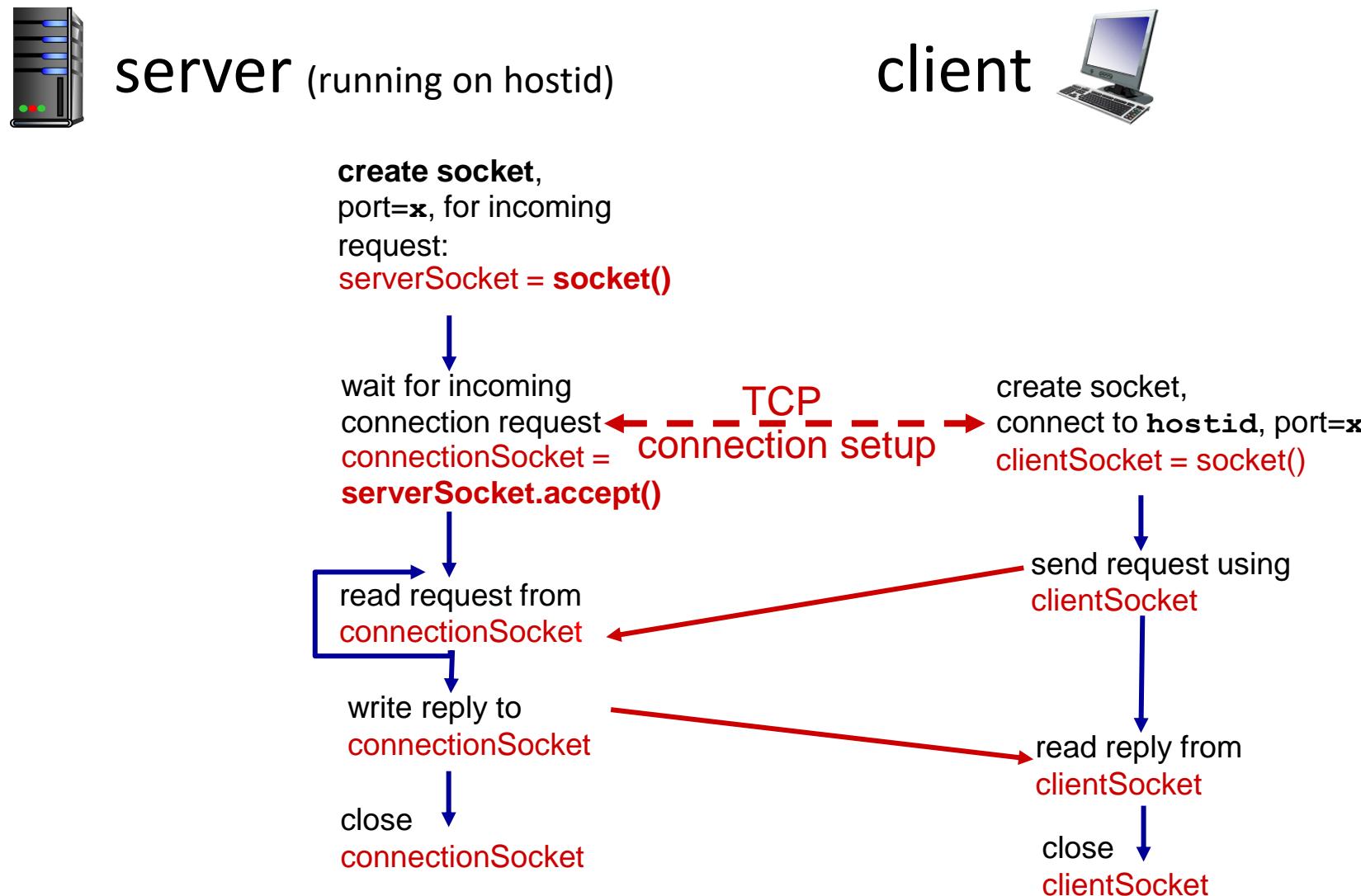
- Creating TCP socket, specifying IP address, port number of server process
- *when client creates socket*: client TCP establishes connection to server TCP

- when contacted by client, *server TCP creates new socket* for server process to communicate with that particular client
 - allows server to talk with multiple clients
 - client source port # and IP address used to distinguish clients

Application viewpoint

TCP provides reliable, in-order byte-stream transfer ("pipe") between client and server processes

Client/server socket interaction: TCP



Example app: TCP client

Python TCPClient

```
from socket import *
serverName = 'servername'
serverPort = 12000
clientSocket = socket(AF_INET, SOCK_STREAM)
clientSocket.connect((serverName,serverPort))
sentence = input('Input lowercase sentence:')
clientSocket.send(sentence.encode())
modifiedSentence = clientSocket.recv(1024)
print ('From Server:', modifiedSentence.decode())
clientSocket.close()
```

create TCP socket for server,
remote port 12000 → clientSocket = socket(AF_INET, SOCK_STREAM)

No need to attach server name, port → clientSocket.connect((serverName,serverPort))

Example app: TCP server

Python TCP Server

```

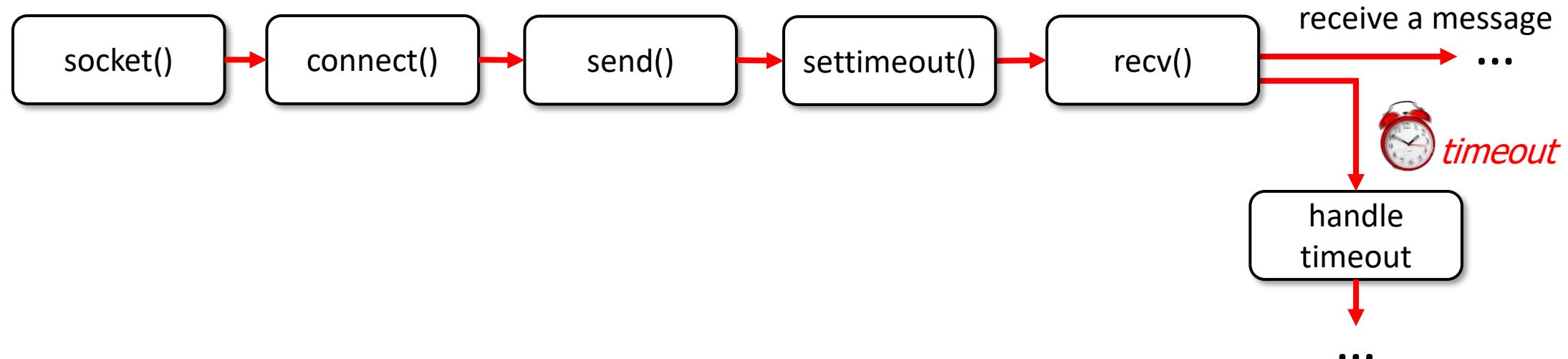
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
print('The server is ready to receive')
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.
                           encode())
    connectionSocket.close()

```

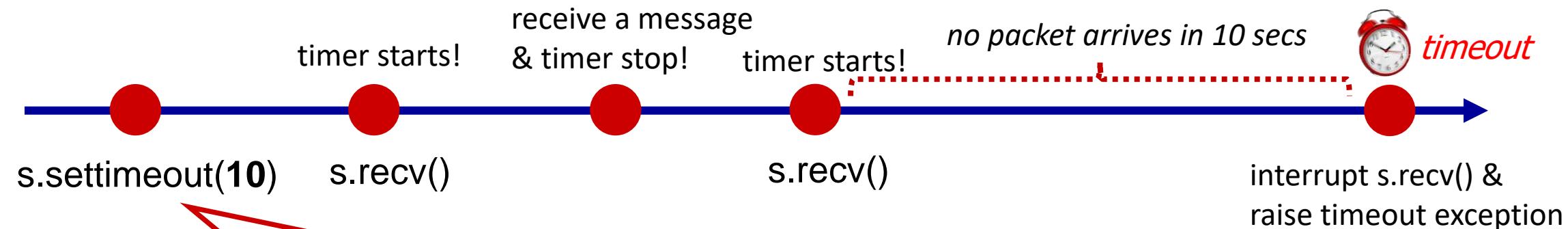
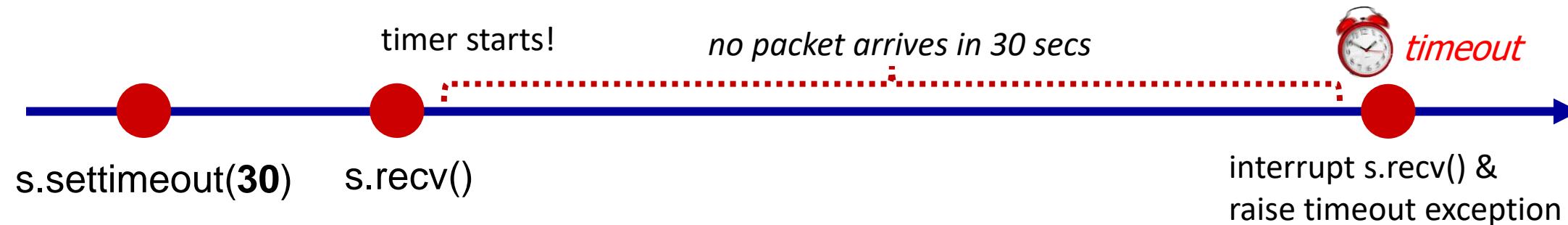
create TCP welcoming socket →
 server begins listening for incoming TCP requests →
 loop forever →
 server waits on accept() for incoming requests, new socket created on return →
 read bytes from socket (but not address as in UDP) →
 close connection to this client (but *not* welcoming socket) →

Socket programming: waiting for multiple events

- sometimes a program must **wait for one of several events** to happen, e.g.,:
 - wait for either (i) a reply from another end of the socket, or (ii) timeout: **timer**
 - wait for replies from several different open sockets: **select()**, multithreading
- timeouts are used extensively in networking
- using timeouts with Python socket:



How Python socket.settimeout() works?



Set a timeout on all future socket operations of that specific socket!

Python try-except block

Execute a block of code, and handle “exceptions” that may occur when executing that block of code

try:

 <do something>

except <exception>:

 <handle the exception>

Executing this **try code block** may cause exception(s) to catch. If an exception is raised, execution jumps directly into **except code block**

this **except code block** is only executed *if an <exception> occurred* in the **try code block** (note: except block is *required* with a try block)

Socket programming: socket timeouts

Toy Example:



- A shepherd boy tends his master's sheep.
- If he sees a wolf, he can send a message to villagers for help using a TCP socket.
- The boy found it fun to connect to the server without sending any messages. But the villagers don't think so.
- And they decided that if the boy connects to the server and doesn't send the wolf location **within 10 seconds for three times**, they will **stop listening** to him forever and ever.

set a 10-seconds timeout on all future socket operations



```
connectionSocket, addr = serverSocket.accept()
connectionSocket.settimeout(10)
```

try:

```
wolf_location = connectionSocket.recv(1024).decode()
send_hunter(wolf_location) # a villager function
connectionSocket.send('hunter sent')
```

except timeout:

```
counter += 1
```

```
connectionSocket.close()
```

timer starts when `recv()` is called and will raise timeout exception if there is no message within 10 seconds.



catch socket timeout exception



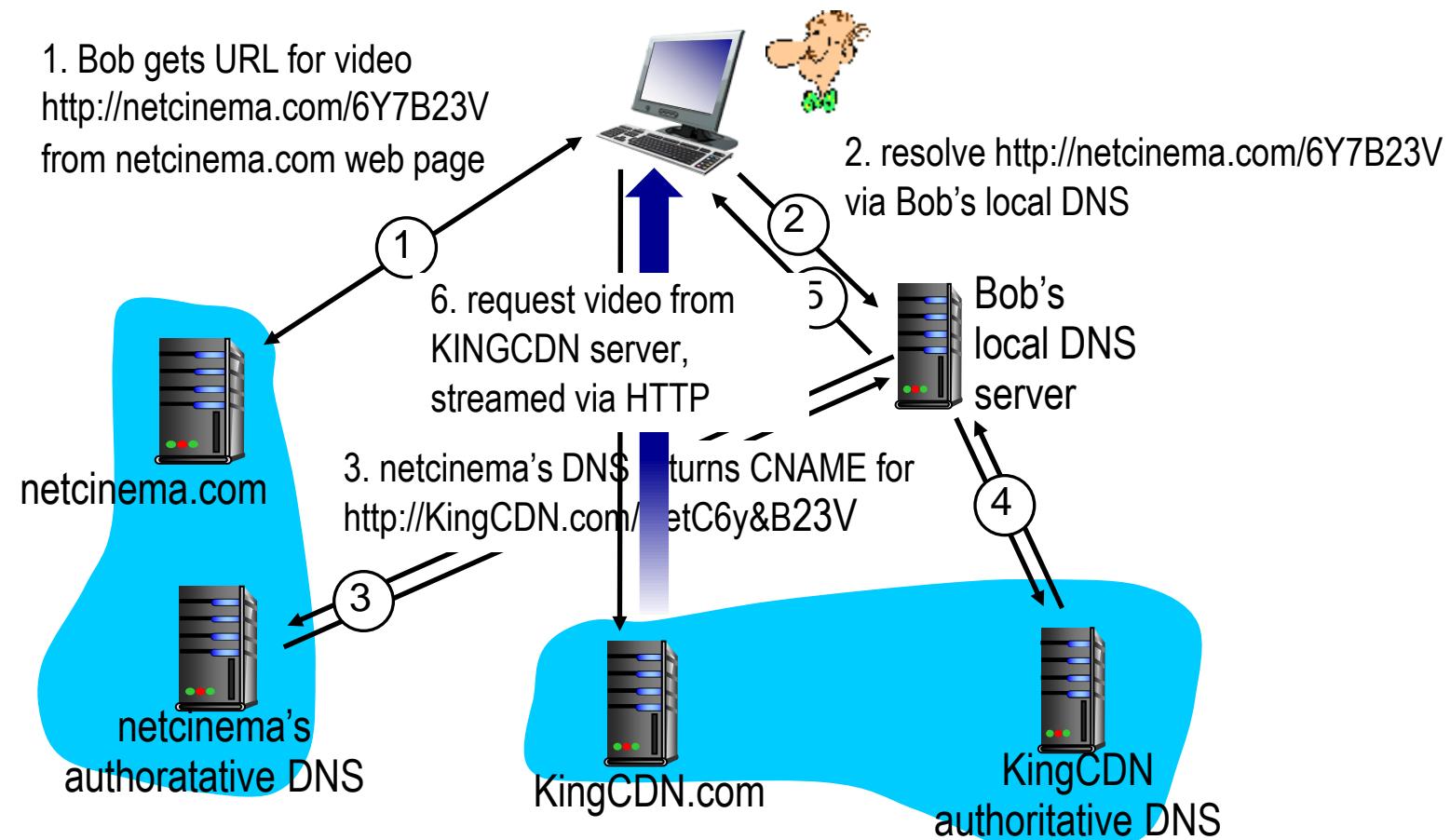
Python TCPServer (Villagers)

```
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(("",serverPort))
serverSocket.listen(1)
counter = 0
while counter < 3:
    connectionSocket, addr = serverSocket.accept()
    connectionSocket.settimeout(10)
    try:
        wolf_location = connectionSocket.recv(1024).decode()
        send_hunter(wolf_location) # a villager function
        connectionSocket.send('hunter sent')
    except timeout:
        counter += 1
    connectionSocket.close()
```

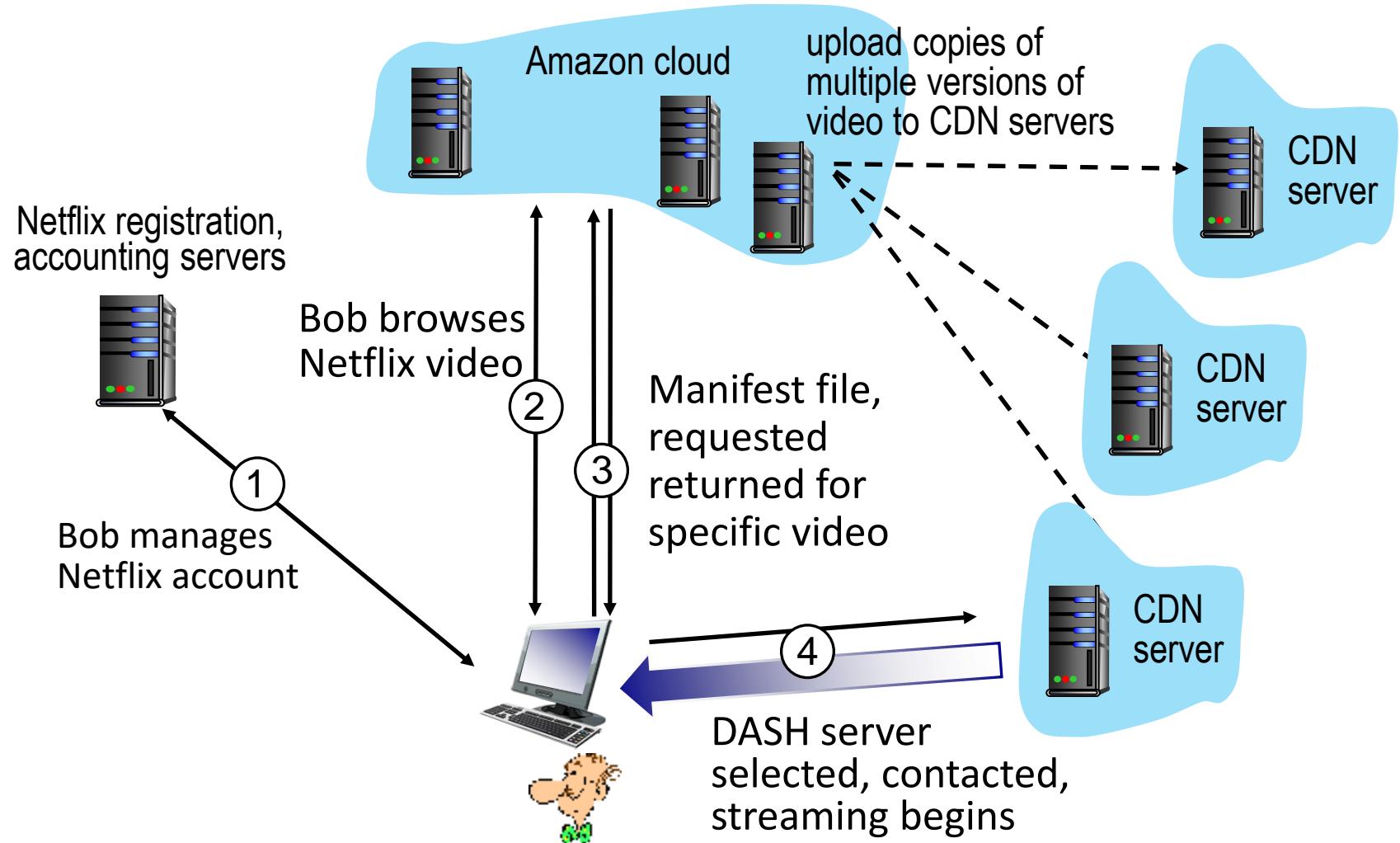
CDN content access: a closer look

Bob (client) requests video <http://netcinema.com/6Y7B23V>

- video stored in CDN at <http://KingCDN.com/NetC6y&B23V>



Case study: Netflix





Thanks
Q & A

Threats and Vulnerability Analysis in Linux

Types of Vulnerabilities

1. Direct
2. Indirect
3. Veiled
4. Conditional

Direct Vulnerabilities

- Direct vulnerabilities are **immediate flaws or weaknesses** that attackers can exploit without intermediaries.
- They often stem from **poor configurations or weak access controls**.
- **Example:** A **weak root password** on a Linux system provides direct access if someone tries a brute-force attack.
- **Other Examples:**
 - **Unpatched software:** Known vulnerabilities in applications or services (e.g., SSH, Apache).
 - **Exposed Ports:** Open services (e.g., FTP on port 21) accessible to everyone without restrictions.

Indirect Vulnerabilities

- Indirect vulnerabilities involve an attacker gaining access through **intermediaries** or compromised third-party software.
- **Example:** An attacker compromises a **Linux web server** and uses it to access internal network systems (e.g., databases or other Linux servers).
- **Other Examples:**
 - **Man-in-the-Middle (MitM) Attacks:** Intercepting traffic to gather information or inject malicious code.
 - **Dependency Exploits:** Vulnerabilities in packages or dependencies installed on the Linux system (e.g., a bug in Python libraries).

Veiled Vulnerabilities

- Veiled vulnerabilities are **hidden flaws** that are difficult to detect and usually embedded in malware.
- **Example:** A rootkit that modifies essential system commands (e.g., ps or top) to hide its processes, making it difficult to detect.
- **Note:** A rootkit is a type of malware that allows cybercriminals to gain access to a computer system and perform malicious activities without being detected:
- **Other Examples:**
 - **Modified Kernel Modules:** Attackers inject code into kernel modules to avoid detection.
 - **Trojanized Programs:** Attackers replace binaries (e.g., netstat) with malicious versions that hide malicious network connections.

Practical Example

- Suppose you suspect your system might have been compromised. You run the netstat command to check for unusual connections:

Normal Output:

```
bash Copy code
$ netstat -tuln
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 192.168.1.10:22          0.0.0.0:*
tcp      0      0 192.168.1.10:80          0.0.0.0:*
```

Trojanized Output:

```
bash Copy code
$ netstat -tuln
Proto Recv-Q Send-Q Local Address          Foreign Address        State
tcp      0      0 192.168.1.10:22          0.0.0.0:*
```

How to Detect Trojanized Programs:

- **Verify Checksums:** Use tools like sha256sum or md5sum to compare the checksums of binaries against known good versions.

```
bash
sha256sum /bin/netstat
```



- **Use Trusted Sources:** Always download software and updates from verified and secure sources.
- **Check File Integrity:** Use tools like rpm -Va (for RPM-based systems) or debsums (for Debian-based systems).
- **Tripwire or AIDE:** These are file integrity monitoring tools that can alert you to unauthorized changes in system files.

Conditional Vulnerabilities

- Conditional vulnerabilities are **exploitable only under specific configurations or circumstances.**
- **Example:** A vulnerability that only affects a particular version of OpenSSL when a specific setting is enabled (e.g., Heartbleed vulnerability).
- **Other Examples:**
 - **Kernel-Specific Exploits:** Vulnerabilities affecting only certain kernel versions.
 - **Service-Specific Configurations:** Misconfigurations in SSH, FTP, or database servers that expose sensitive data.

Heartbleed Vulnerability

- Heartbleed was a serious security flaw in **OpenSSL versions 1.0.1 through 1.0.1f**.
- It exploited a feature called the **Heartbeat extension**, which is used to keep SSL/TLS connections **alive**.
- The vulnerability allowed attackers to **read the memory** of the server, potentially exposing sensitive information like **usernames, passwords, and encryption keys**.

Key Characteristics of the Heartbleed Vulnerability:

- **Version-Specific:**
 - It affected only **OpenSSL versions 1.0.1 to 1.0.1f**.
 - Versions before 1.0.1 or after 1.0.1f were not vulnerable.
- **Condition-Specific:**
 - The server or client needed to have the **Heartbeat extension enabled** for the vulnerability to be exploited.
 - If the Heartbeat feature was disabled, the vulnerability could not be triggered.

Other Examples of Condition-Specific Vulnerabilities:

- **Shellshock** (CVE-2014-6271):
 - Affected only specific versions of the **Bash shell**.
 - It could be exploited only if the system used **Bash** as its command interpreter and allowed remote code execution.
- **Spectre and Meltdown**:
 - These vulnerabilities affected specific **CPU architectures** and required **speculative execution** to be enabled.

Key Takeaways for Cybersecurity Engineers

- **Version-Specific:** Always ensure that your software is up to date. Vulnerabilities often impact only certain versions, so applying patches can protect you from known issues.
- **Configuration Matters:** Disabling unused or unnecessary features can reduce your attack surface. For example, if you don't need the Heartbeat feature, disabling it would have protected against Heartbleed.
- **Continuous Monitoring:** Use vulnerability scanners and tools like **Nessus** or **OpenVAS** to identify version-specific vulnerabilities in your system.

Security Measures in Linux

- To maintain system security, several measures and best practices can be employed:
 1. SSH Key Pair for Secure Authentication
 2. Scanning Log Files
 3. Closing Hidden Ports

SSH Key Pair for Secure Authentication

- SSH keys provide a **more secure alternative** to password-based authentication.
 - **How it Works:** Generates a pair of cryptographic keys—a **public key** (stored on the server) and a **private key** (kept on the user's local machine).
 - Commands:

```
bash
```

 Copy code

```
ssh-keygen -t rsa -b 4096 -c "user@example.com" # Generate SSH key ✓  
ssh-copy-id user@server_ip # Copy public key to server
```

SSH Key Pair for Secure Authentication

- Command 1: `ssh-keygen -t rsa -b 4096 -C user@example.com`
 - This command generates a new SSH key pair with specific parameters.
 - **ssh-keygen**: This is the command-line tool for generating SSH keys. It creates both a **private key** and a **public key**.
 - **-t rsa**: This specifies the type of encryption algorithm to use. Here, **rsa** means it ~~will use the RSA algorithm~~ (Rivest-Shamir-Adleman), which is widely used for secure data transmission.
 - **-b 4096**: This option defines the **number of bits in the key**. A key size of 4096 bits is ~~quite strong~~ and more secure than the default 2048 bits. The larger the key size, the more difficult it is to crack, but it also requires more processing power.
 - **-C "user@example.com"**: This is a **comment or label** for the key, often set to your email or username to help identify the key. This comment is appended to the public key file, making it easier to manage keys by associating them with specific users.

SSH Key Pair for Secure Authentication

- Command 2: `ssh-copy-id user@server_ip` ✓
 - This command copies the **public key** to the remote server, allowing you to log in securely without a password.
 - **ssh-copy-id**: This command securely installs the generated public key (`id_rsa.pub`) on the server so you can use **passwordless SSH authentication**.
 - **user@server_ip**: This specifies the **username and IP address (or hostname)** of the remote server where the key will be copied. For example, if your username is `debian` and your server IP is `192.168.1.5`, you would enter `debian@192.168.1.5`. This setup ensures that the remote server recognizes your private key and allows access without a password.

Scanning Log Files

- Logs provide **valuable insights** into potential security issues. By analyzing log files, students can detect anomalies, failed login attempts, and other suspicious activities.
- **Key Log Files:**
- `/var/log/auth.log` - Authentication logs (failed and successful login attempts). 
- `/var/log/syslog` - General system events. 

Identifying and Closing Hidden Ports

- Unused or hidden ports can expose services to potential threats. Identifying and closing these ports helps in **minimizing the attack surface**.
- Identifying Open Ports:**

```
sudo ss -tuln          # List services listening on ports  
sudo nmap -sT localhost # Scan for open ports on localhost
```

- Closing Unused Ports:**

```
sudo systemctl stop service_name      # Stop unused services  
sudo systemctl disable service_name    # Disable service from starting at boot  
sudo iptables -A INPUT -p tcp --dport 8080 -j DROP # Block specific port using iptables
```

Identifying Open Ports - Command 1: sudo ss -tuln

- This command uses the `ss` tool to list active network connections and sockets. Each parameter specifies the type of information to display.
 - `sudo`: Runs the command with root privileges, necessary to view information about services running on privileged ports (ports below 1024).
 - `ss`: A tool to display socket statistics, similar to `netstat`, but faster and more modern.
 - `-t`: Shows **TCP connections** only. TCP (Transmission Control Protocol) is a connection-oriented protocol, commonly used for data transmission over the internet.
 - `-u`: Shows **UDP connections** only. UDP (User Datagram Protocol) is a connectionless protocol, often used for streaming or real-time applications.
 - `-l`: Displays only **listening sockets**. Listening sockets are those that are open and waiting for incoming connections on specified ports.
 - `-n`: Shows addresses and port numbers in **numeric format** rather than resolving them into hostnames and service names. This makes the output faster to generate and easier to read when looking for specific port numbers.
- **Overall Meaning:** `sudo ss -tuln` lists all active listening TCP and UDP sockets (ports) on the system, showing them in a numeric format.

Identifying Open Ports - Command 2: sudo nmap -sT localhost

- This command uses nmap to scan for open ports on the local machine.
 - **sudo**: Runs the command with root privileges, which may be necessary for a more complete scan of certain ports and services.
 - **nmap**: Network Mapper, a powerful tool for network discovery and security auditing. It can identify open ports, running services, and operating systems.
 - **-sT**: Specifies a **TCP connect scan**. This type of scan establishes a full TCP connection to each scanned port, identifying whether each port is open, closed, or filtered. It's the most reliable but can be slower since it fully establishes and tears down the TCP connection.
 - **localhost**: Specifies the **target to scan**, in this case, localhost (the current machine).
- **Overall Meaning:** sudo nmap -sT localhost performs a TCP connect scan on the local machine to identify which TCP ports are open and listening.

Closing Hidden Ports-Command 1: sudo systemctl stop service_name

- This command stops a service currently running on the system.
 - **sudo**: Runs the command with root privileges, which are needed to manage system services.
 - **systemctl**: This is the command used to **control systemd** and manage services in modern Linux distributions.
 - **stop**: Tells `systemctl` to **stop** a specific service.
 - **service_name**: This is a placeholder for the name of the service you want to stop (e.g., `apache2` for Apache server, `nginx` for NGINX server).
- **Overall Meaning:** This command stops a currently running service, which releases system resources and closes any associated network connections.

Closing Hidden Ports-Command 2: sudo systemctl disable service_name

- This command prevents a service from starting automatically at system boot.
 - **sudo**: Runs the command with root privileges, necessary to change the startup configuration of system services.
 - **systemctl**: Manages services and system settings with systemd.
 - **disable**: Tells systemctl to **disable** the specified service from starting on boot.
 - **service_name**: The name of the service you wish to disable (e.g., apache2, nginx).
- **Overall Meaning:** This command stops the specified service from launching automatically the next time the system boots, which can improve boot time and reduce system load.

Closing Hidden Ports-Command 3: sudo iptables -A INPUT -p tcp --dport 8080 -j DROP

- This command uses **iptables** to block incoming connections on a specific port, enhancing system security by restricting access.
 - **sudo**: Runs the command with root privileges, necessary to alter firewall rules.
 - **iptables**: A command-line firewall utility for managing network traffic rules in Linux.
 - **-A INPUT**: Adds (-A) a rule to the INPUT chain, which handles incoming connections. This tells the system to evaluate incoming packets against this rule.
 - **-p tcp**: Specifies the protocol to be matched by the rule, in this case, **TCP**. This is common for applications requiring reliable connections, like web servers.
 - **--dport 8080**: Specifies the **destination port** number (8080 in this example) for the rule. This is the port we want to block.
 - **-j DROP**: Defines the **action** to be taken on matching packets, which is to **DROP** them. Dropping a packet means it is discarded silently without any response, effectively blocking incoming connections on this port.
- **Overall Meaning:** This command blocks all incoming TCP connections on port 8080, which can prevent unwanted access to services listening on that port.

Linux Malware: Types and Characteristics

- Linux is not immune to malware; here are three significant types:
 1. Botnets
 2. Ransomware
 3. Rootkits

Botnets

- Botnets consist of **infected devices** that an attacker can remotely control to perform tasks like DDoS attacks, spam, or data theft.
- **Example:** Mirai Botnet - targets Linux-based IoT devices by scanning for weak Telnet passwords.
- **Detection and Prevention:**
 - **Disable Telnet** and use SSH with key-based authentication.
 - **Monitor outbound traffic** for unusual activity patterns.
 - **Use Intrusion Detection Systems (IDS)** to detect unusual network activity.

Ransomware

- Linux ransomware is less common but can encrypt files or services, demanding a ransom to restore access.
- **Example:** Erebus Ransomware - targeted Linux servers in South Korea, encrypting files and demanding Bitcoin for decryption.
- **Protection Measures:**
 - Regular **backups** of important data.
 - Use **read-only mounts** for critical directories.
 - Implement **file integrity monitoring** tools to detect unauthorized modifications.

Rootkits

- Rootkits are highly sophisticated malware that gives attackers **persistent root-level access** and hides their presence.
- **Example:** Linux.Lady - a cryptocurrency mining rootkit that stays hidden in infected systems.
- **Detection and Prevention:**
 - Use **rootkit detection tools** like chkrootkit or rkhunter.
 - Enable **kernel hardening** features, such as disabling module loading after boot.
 - Limit **root privileges** and use **Least Privilege Principles** for processes.

Summary

Concept	Explanation	Example/Command
Direct Vulnerability	Immediate flaws (e.g., weak root password)	Exposed SSH port, unpatched service
Indirect Vulnerability	Through intermediaries (e.g., MitM attacks)	Man-in-the-Middle (MitM)
Veiled Vulnerability	Hidden malware, hard to detect	Rootkits modifying system command
Conditional Vulnerability	Exploitable only in specific configurations	Heartbleed affecting OpenSSL
SSH Key Pair	Secure remote access without passwords	<code>ssh-keygen</code> , <code>ssh-copy-id</code>
Log File Scanning	Detect suspicious activities via logs	<code>grep "Failed password"</code>
Close Hidden Ports	Minimize attack surface by closing ports	<code>iptables -A INPUT --dport 8080 DROP</code>
Botnet	Infected devices for coordinated attacks	Mirai Botnet - DDoS and spam
Ransomware	Encrypts files, demands ransom	Erebus targeting Linux servers
Rootkit	Hidden malware for persistent access	<code>chkrootkit</code> , <code>rkhunter</code>