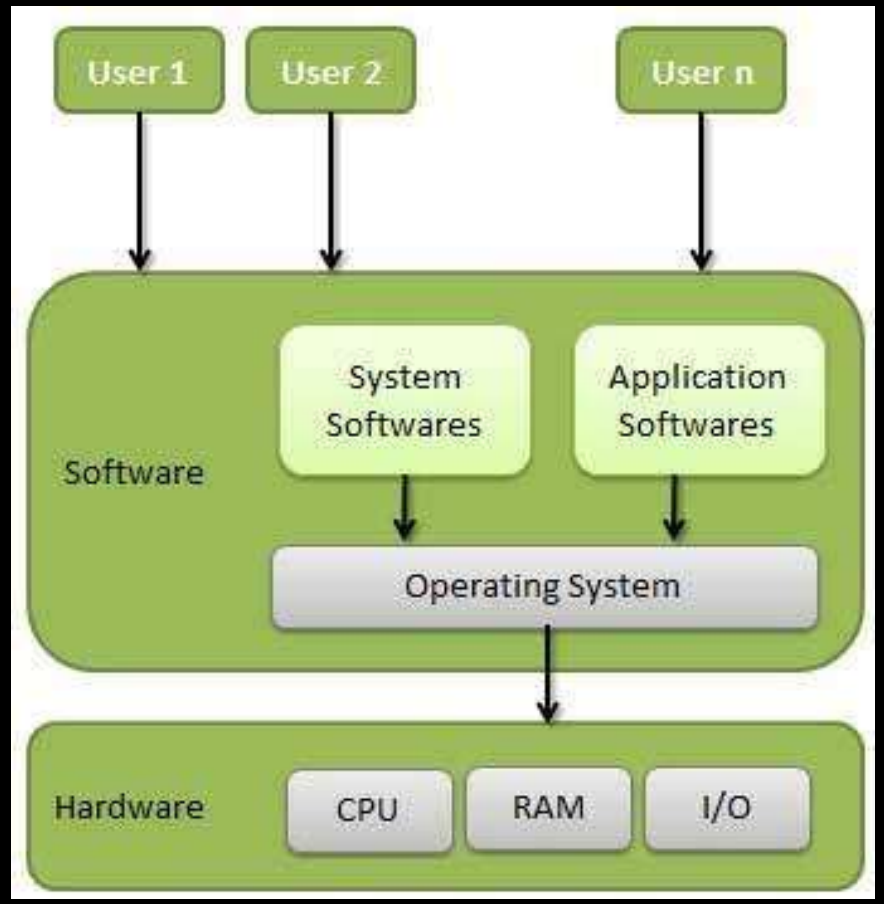
Concepts of Operating Systems

Atul Kahate

Session 1: Introduction to Operating Systems

Introduction to OS Concepts

- An **Operating System (OS)** is a program that acts as an intermediary between a user of a computer and the computer hardware.
- Operating system goals:
  - Execute user programs and make solving user problems easier.
  - Make the computer system convenient to use.
  - Use the computer hardware in an efficient manner.



Operating System versus Application Software

Linux, Windows, Mac

An **operating system** is like the entry point into a computer: This is where the computer *wakes up*.

The operating system is responsible for handling all the aspects pertaining to user programs, process management, memory management, disk management, working with the CPU, etc.

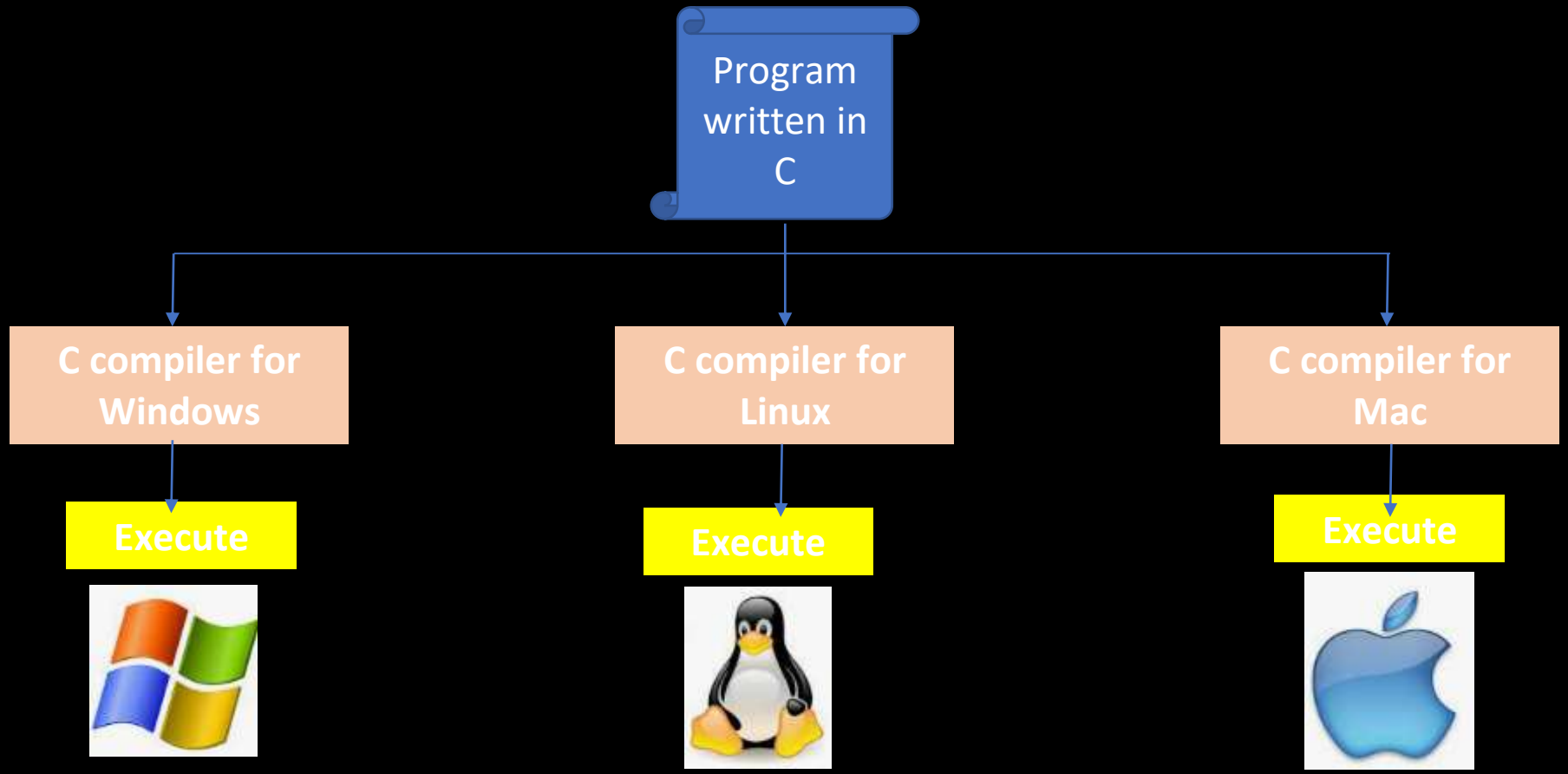
Word, PowerPoint, Firefox

Application software makes use of an operating system for performing various tasks.

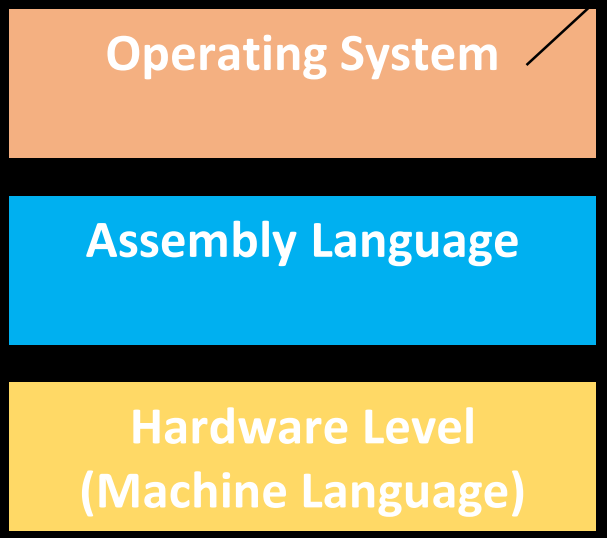
An application software allows the user to create presentations, play games, browse the Internet, etc.

An application software is *specific*, whereas an operating system is *generic*

Why are High-level Languages Portable?



Why is OS Hardware-dependent?



We have two major choices when writing an OS:

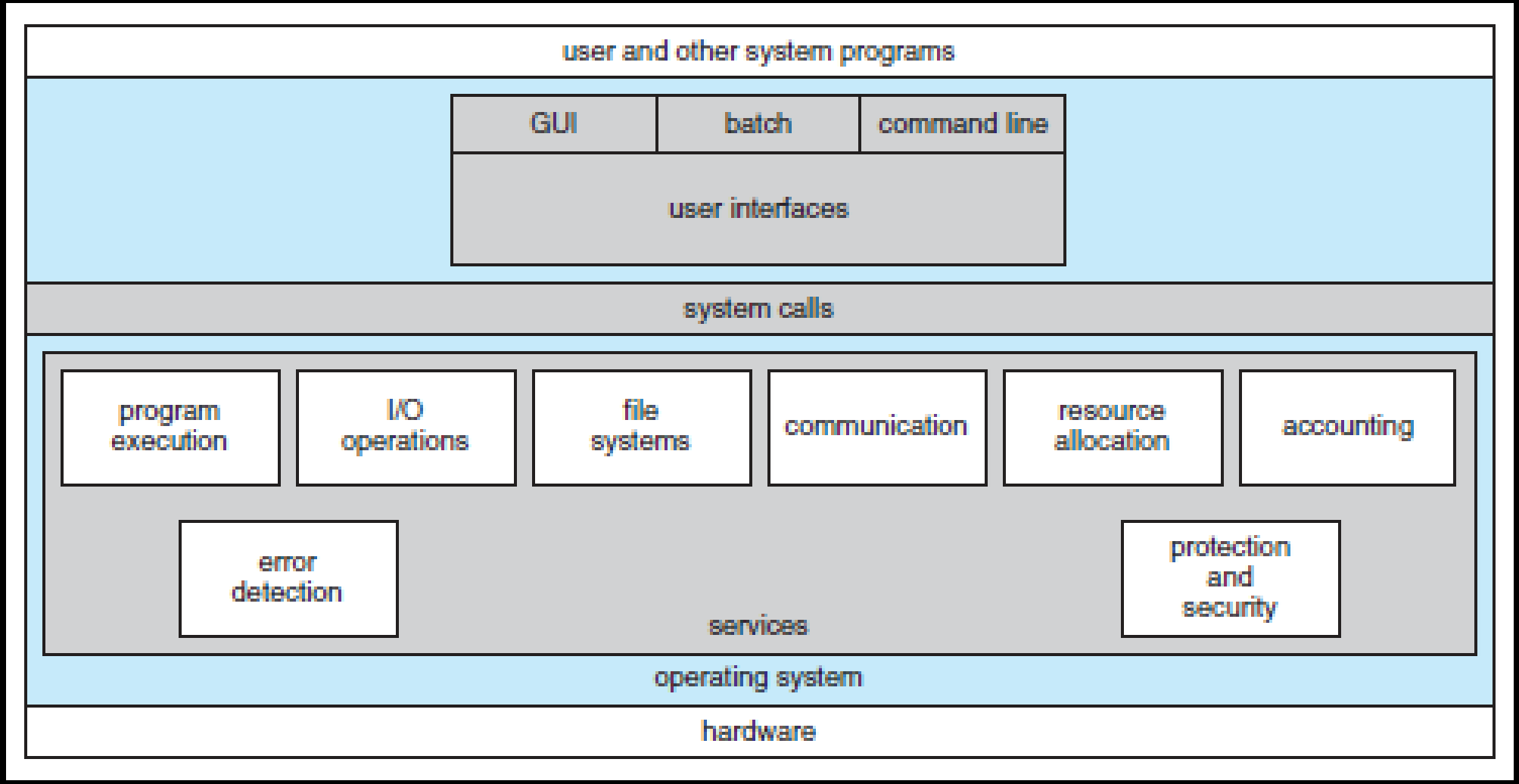
Choice 1: Write the OS in the assembly language of the chosen hardware

Hardware-dependent

Choice 2: Write the OS in a high-level language (such as C or C++ or Java)

Hardware-independent

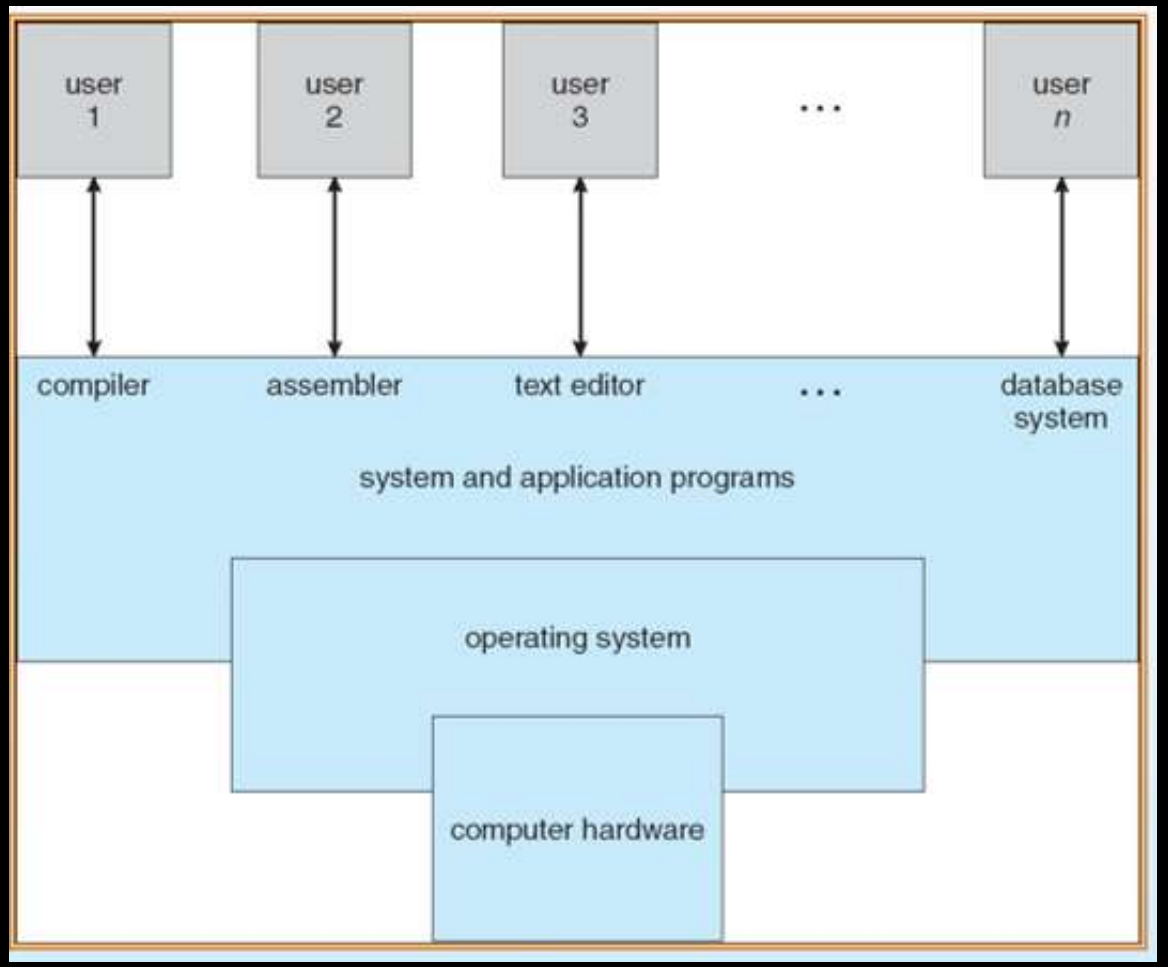
Different Components of an OS



Computer System Structure – 4 Components

- **Hardware** – Provides basic computing resources
  - CPU, memory, I/O devices
- **Operating system** - Controls and coordinates use of hardware among various applications and users
- **Application programs** – Define the ways in which the system resources are used to solve the computing problems of the users
  - Word processors, compilers, web browsers, database systems, video games
- **Users** - People, machines, other computers
- *See next slide for a diagram*

Computer System – Four Components



Examples of OS

- **Mobile OS**: Android, iOS
- **Embedded Systems OS**: Embedded Linux, QNX, VxWorks – Generally work on microwave ovens, TV, DVD, MP3 players, etc – Embedded in ROM
- **Real-time OS**: eCos - Deadline-based work, Controlling machines, Welding robots, etc
- **Desktop OS**: Windows, Linux, Mac – General-purpose OS
- **Server machine OS**: Windows server/Linux server – Need to handle client-requests, load, security

Main Functions of Operating Systems

Provide abstraction to user programs

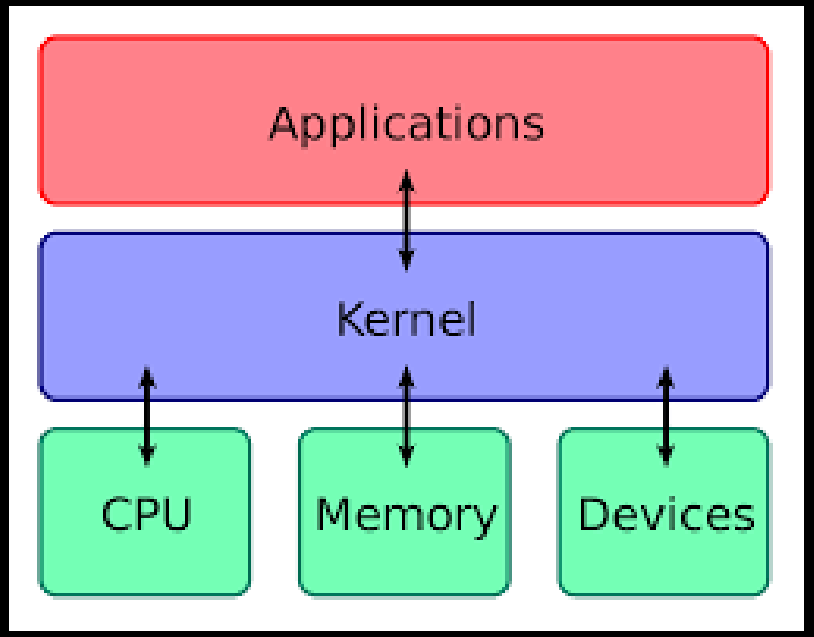
A modern computer consists of one or more processors, some main memory, disks, printers, a keyboard, a mouse, a display, network interfaces, and various other input/output devices. If every application programmer had to understand how all these things work in detail, no code would ever get written.

Managing computer resources

Managing these resources in itself is a huge task!

Kernel

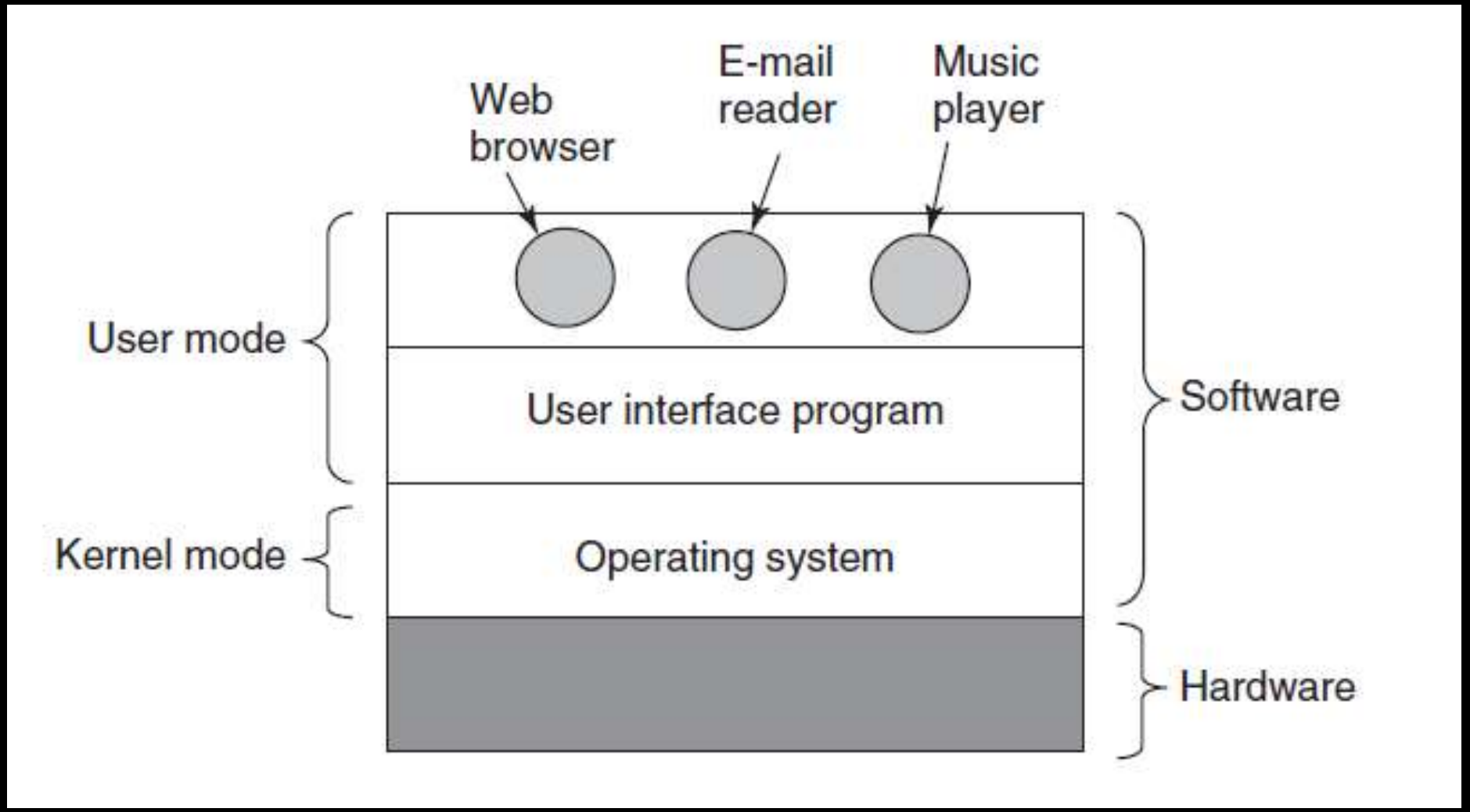
- “The one program running at all times on the computer” is the **kernel**.
- Everything else is either a **system program** (ships with the operating system) or an **application program**.



User Mode and Kernel Mode (Diagram on Next Slide)

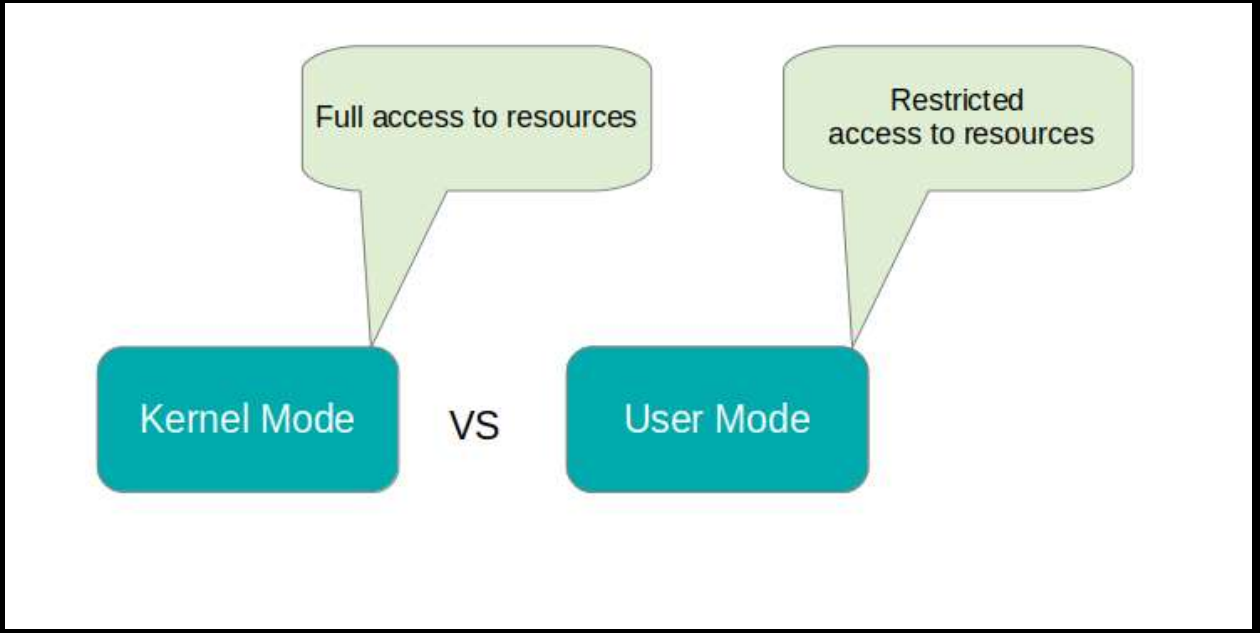
- Most computers have two modes of operation: **kernel mode** and **user mode**.
- The operating system runs in kernel mode (also called **supervisor mode**).
  - In this mode it has complete access to all the hardware and can execute any instruction the machine is capable of executing.
- The rest of the software runs in user mode, in which only a subset of the machine instructions is available.
  - In particular, those instructions that affect control of the machine or do I/O (Input/Output) are forbidden to user-mode programs.

User Mode and Kernel Mode



User Mode and Kernel Mode

- **User mode** has restricted access to resources. Example: Text editor, Media player. If a resource such as disk or file is needed, a signal is sent to the CPU via an interrupt to switch to the kernel mode.
- **Kernel mode** (also called as **supervisor mode**) has full access to memory, I/O, and other resources. Here, CPU has full access to the hardware and can execute any instruction.



User Space and Kernel Space

User space

Area of memory where all the user mode applications execute.

This memory can be "swapped out" when necessary.

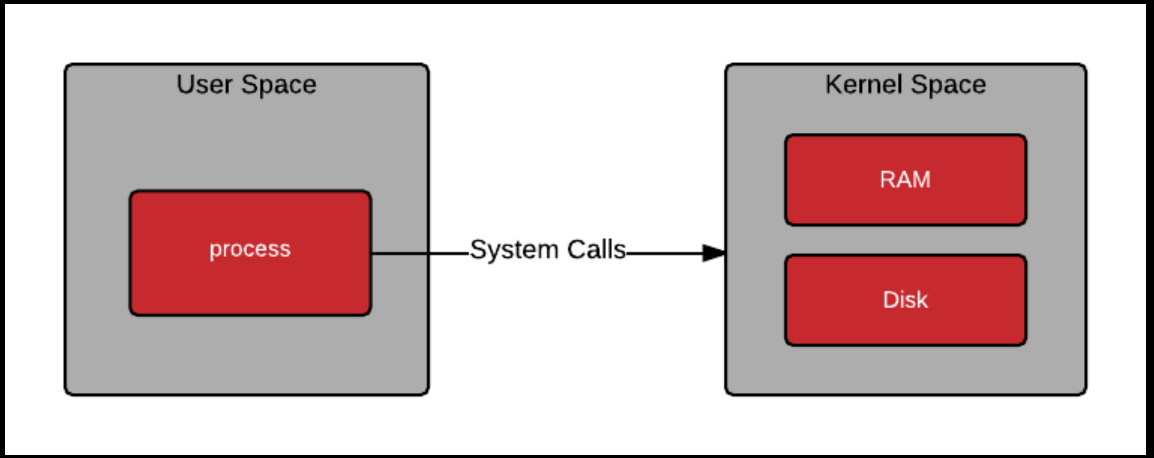
Kernel space

Area of memory where all the kernel mode applications execute.

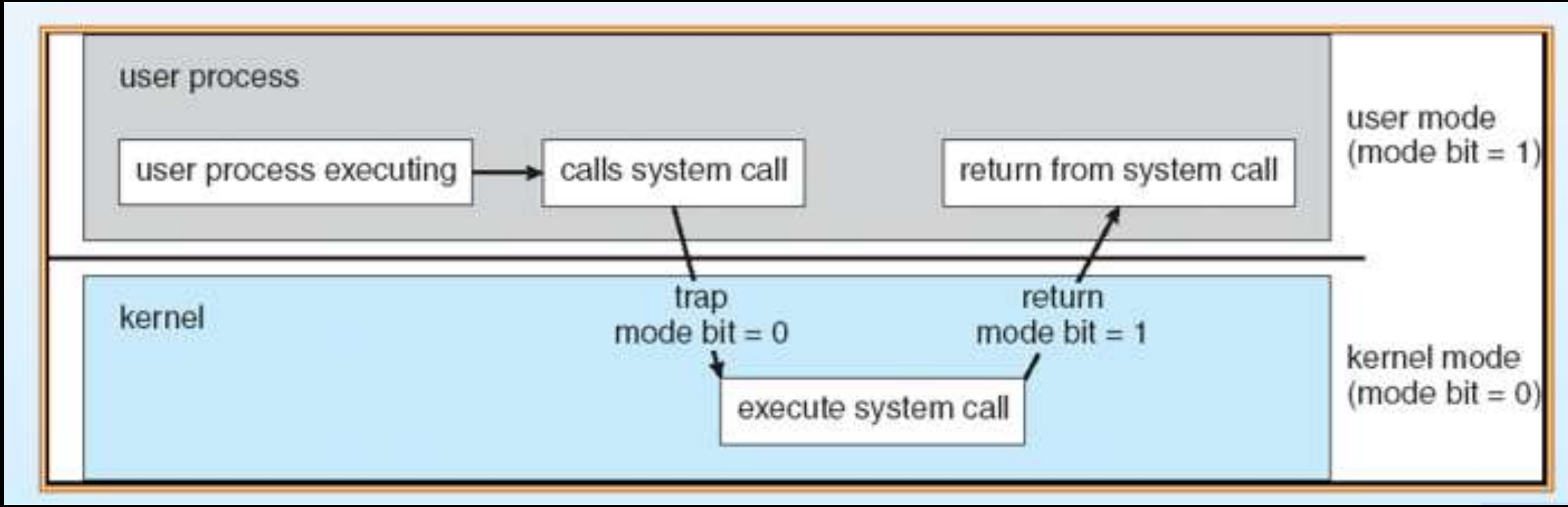
Strictly reserved for kernel background processes and has full hardware access.

User Space and Kernel Space

- **User space** refers to all of the code in an operating system that lives outside of the kernel. Most Unix-like operating systems (including Linux) come pre-packaged with all kinds of utilities, programming languages, and graphical tools - these are user space applications.
- The **kernel space** provides abstraction for security, hardware, and internal data structures. The open() system call is commonly used to get a file handle in Python, C, Ruby and other languages. You wouldn't want your program to be able to make bit level changes to the XFS file system, so the kernel provides a system call and handles the drivers.



User Mode to Kernel Mode Transition



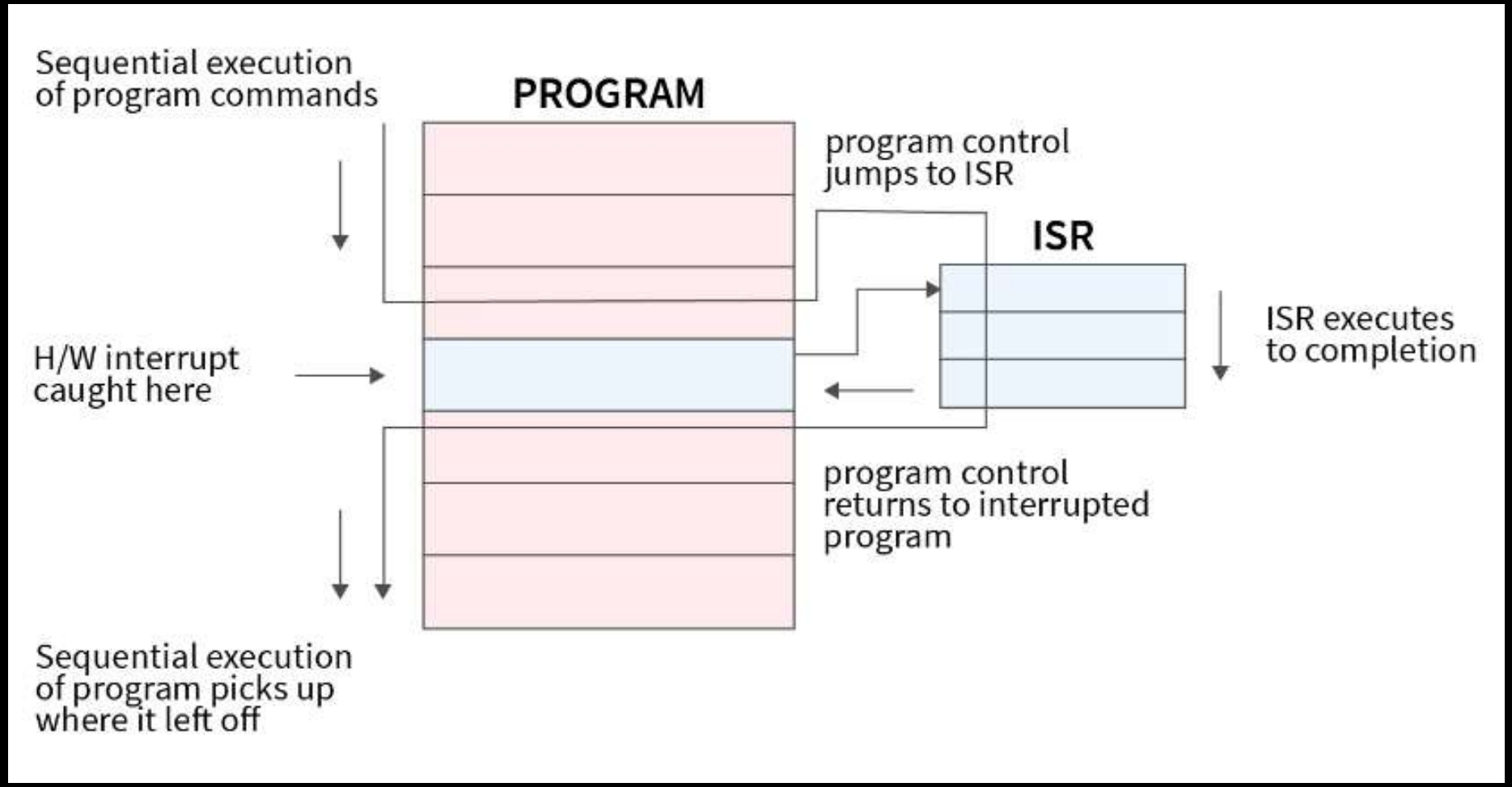
Kernel Mode versus User Mode

BASIS	KERNEL MODE	USER MODE
Permission	Unrestricted and full permissions to access the system's hardware	Restricted and limited permissions to access the system's hardware
Memory reference	It can reference to both the memory spaces	It can only reference to memory space that is dedicated to user mode
Access	Only core functionality can be allowed to operate in this very mode	User applications can access this mode for a particular system and is allowed to operate in this particular mode
System crash	Fatal and increases the complexity	Recoverable and can simply restart the session
Also Known	Privileged mode or Supervisor mode	Restricted mode

Interrupt

- When the current processing needs to be temporarily stopped for handling some event of a higher priority, an **interrupt** is generated
- Example: The operating system is reading a file and the user types something on the keyboard
  - This will generate a keyboard interrupt
  - The operating system will handle this interrupt (i.e. handle the processing related to the key) and then resume file reading

Interrupt Example



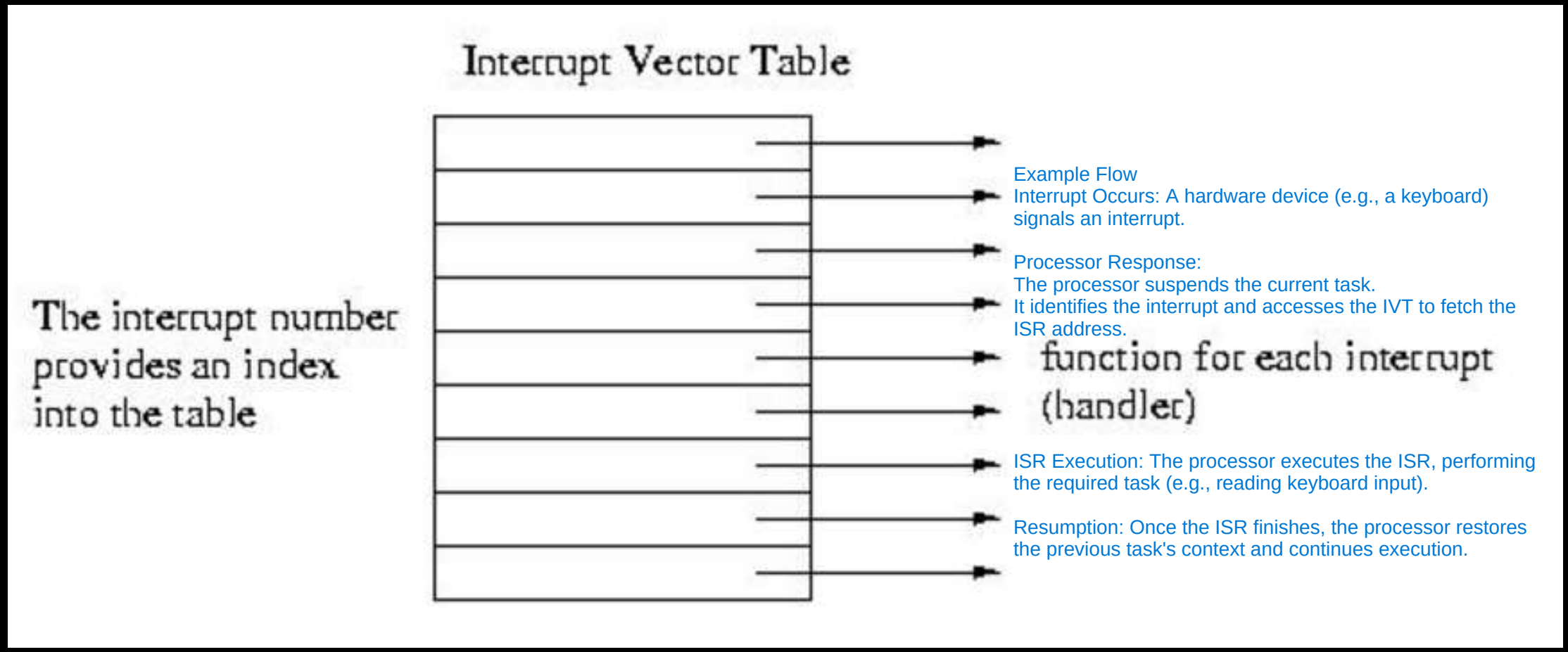
Handling interrupts

- An **interrupt** is a signal sent by hardware or software to the processor for processing
- The process that runs when an interrupt is generated is the **interrupt handler**
- The CPU saves the state of the ongoing process and shifts its attention to the interrupt generated by giving access to the interrupt handler
- This entire process is called **interrupt handling**
- The interrupt handler is also known as **Interrupt Service Routine (ISR)**
- ISR handles the request and sends it to the CPU
- When the ISR is complete, the halted process is resumed

How does the CPU know which ISR to call?

- Each type of interrupt is assigned a unique **interrupt number**
- The interrupt number and its corresponding instructions set address (or process's base address) are stored in a vector table known as **Interrupt Vector Table (IVT)**, see next slide
- Using the interrupt number and IVT, the CPU comes to know about the base address of the process that is needed to handle the respective interrupt
- Now, the control is given to that process to handle the interrupt and this process is known as **Interrupt Service Routine (ISR)**

Interrupt Vector Table



Example Flow
Interrupt Occurs: A hardware device (e.g., a keyboard) signals an interrupt.

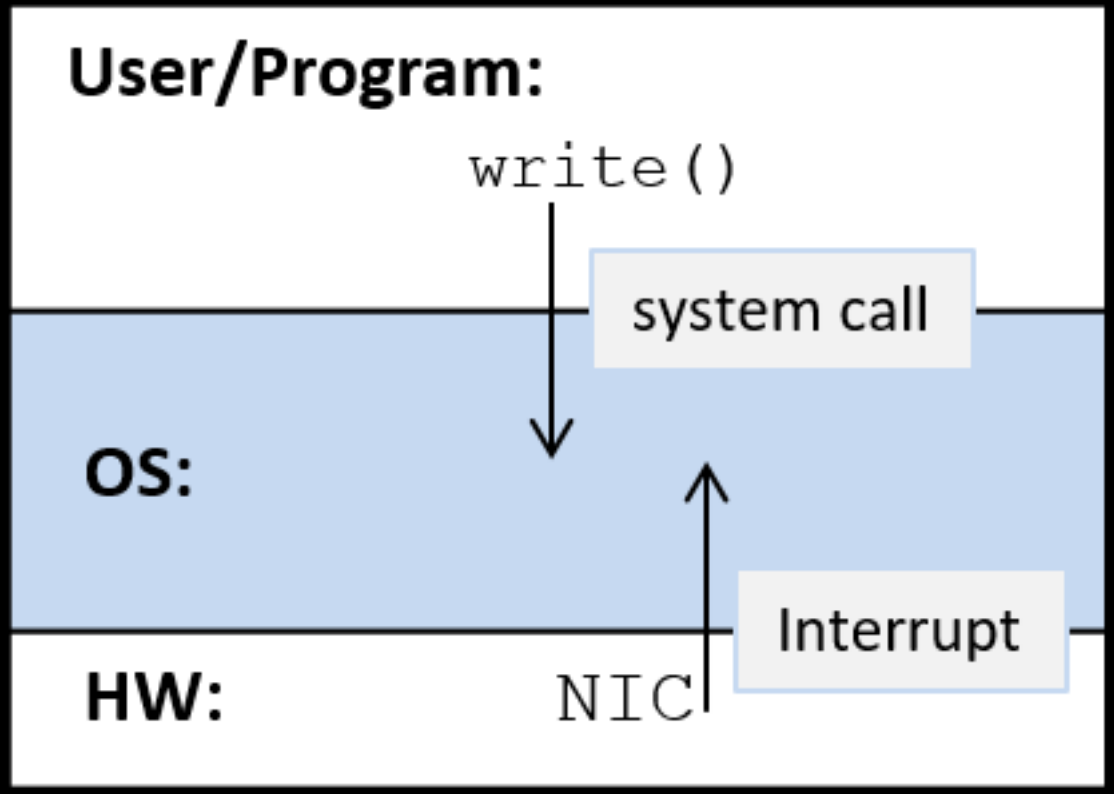
Processor Response:
The processor suspends the current task.
It identifies the interrupt and accesses the IVT to fetch the ISR address.

ISR Execution: The processor executes the ISR, performing the required task (e.g., reading keyboard input).

Resumption: Once the ISR finishes, the processor restores the previous task's context and continues execution.

Interrupts and System Calls

- Most operating systems are implemented as **interrupt-driven systems**, meaning that the OS does not run until some entity needs it to do something — the OS is woken up (*interrupted from its sleep*) to handle a request.
- For example, a **Network Interface card (NIC)** is a hardware interface between a computer and a network. When the NIC receives data over its network connection, it interrupts (or wakes up) the OS to handle the received data.
- Requests to the OS also come from user applications when they need access to **protected resources**. For example, when an application wants to write to a file, it makes a **system call** to the OS, which wakes up the OS to perform the write on its behalf.



IVT serves as a directory or map, telling the system where to find the ISR for a given interrupt.
ISR is the actual code that runs to handle the interrupt.

Session 2: Introduction to Linux

Unix

- In 1969-1970, **Kenneth Thompson**, Dennis Ritchie, and others at AT&T Bell Labs began developing a small operating system on a little-used PDP-7 computer.
- The operating system was soon christened **Unix**, a pun on an earlier operating system project called MULTICS.
- In 1972-73, Unix was rewritten in C (a visionary and unusual step), which made it **portable**

Unix Progression

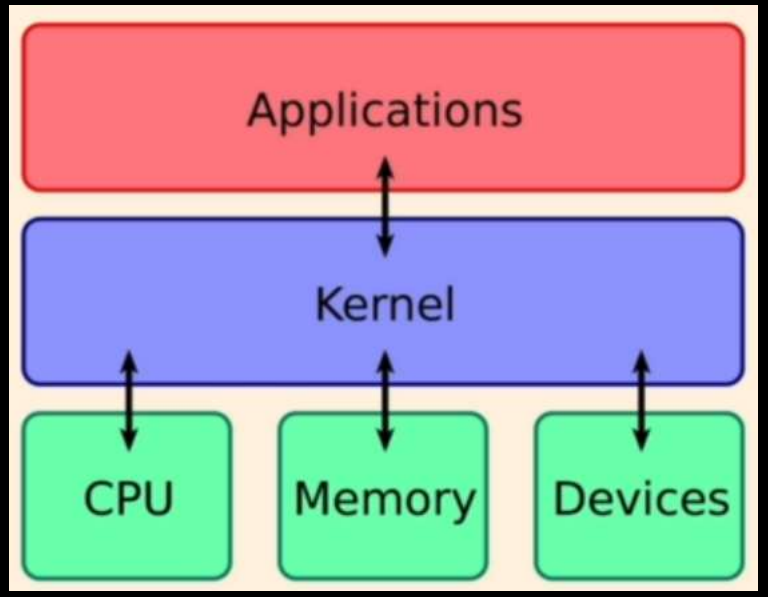
- Two dominating versions emerged
  - Berkley Software Distribution (BSD)
  - AT&T
- Many other *mixed* versions emerged later
- Most of these were proprietary and were maintained by their respective hardware vendor
  - Example: Sun Solaris
  - FreeBSD, NetBSD, OpenBSD became open source

Free Software Foundation (FSF)

- In 1984, Richard Stallman's **Free Software Foundation (FSF)** began the GNU project, a project to create a free version of Unix (See next slide)

Linux History

- A Finnish student Linus Torvalds started a personal project in 1991
- He was trying to create a new free operating system **kernel**, which could work with the FSF material and some BSD components to create a very useful operating system



Memory management, Task scheduling, …

Kernel = *Heart* of an operating system, which facilitates interactions between hardware and software

Installing Linux – Multiple Options

- Use a virtual machine and install a Linux distribution
  - VMWare or Oracle VirtualBox
- Create a dual-boot machine (Windows and Linux)
- On a Windows machine, install WSL (Simplest and light-weight)
  - https://learn.microsoft.com/en-us/windows/wsl/install

Making the screen larger

- Click on the left bottom blue button, System tools, QTerminal

Example Workflow:
Open a terminal (e.g., GNOME Terminal or Windows Terminal).

The terminal starts a shell (like Bash, Zsh, or Fish).

Type a command (e.g., ls), and the shell interprets it.

The shell executes the command via the operating system and displays the result in the terminal.

Installation …

- sudo apt-get update
- sudo apt install virtualbox-guest-x11 -y

```
atul@atul-virtualbox:~$ sudo apt install virtualbox-guest-x11 -y
[sudo] password for atul:
```

- Again click on the blue button, Leave, Reboot
- Now if we use the maximize button (Right top), we should be able to see a full window of Lubuntu

Linux Distributions (“Distros”)

- In the Linux community, different organizations have combined the available components differently.
- Each combination is called a “distribution”, and the organizations that develop distributions are called “distributors”.
- Common distributions include Red Hat, Mandrake, SuSE, Caldera, Corel, and Debian.
- There are differences between the various distributions, but all distributions are based on the same foundation: the Linux kernel and the GNU glibc libraries.

Linux Command Line

- Command line = **Shell**
- **Shell** is a program that takes keyboard commands and passes them to the operating system for execution
- Almost all Linux distributions support a shell program called **bash**
- Bash = Bourne Again Shell (because bash replaces sh, the original Unix shell program written by Steve Bourne)
- The name *shell* is given because it is the outer layer of the OS like the shell around an oyster

The Shell Prompt

- General format: *username@machinename*, followed by the current working directory, and a dollar sign

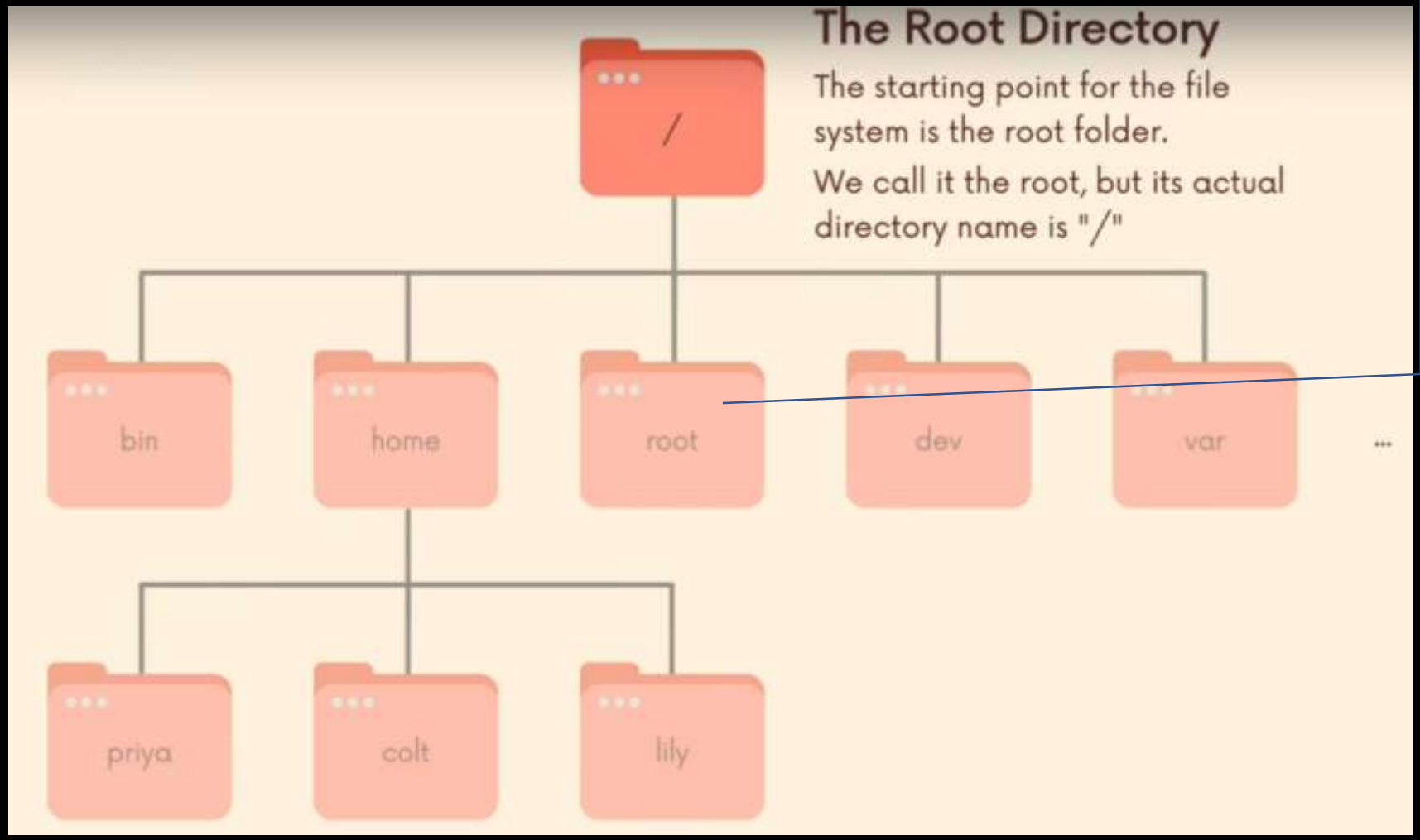
```
akahate@LAPTOP-E9C3P1LP:~$
```

- If we have a hash (#) at the end, instead of the dollar, the user has *superuser/root* privileges

Filesystem Tree

- Like Windows, a Unix-like operating system such as Linux organizes its files in what is called a **hierarchical directory structure**.
- This means that they are organized in a tree-like pattern of directories (sometimes called folders in other systems), which may contain files and other directories.
- The first directory in the filesystem is called the **root directory**.
- The root directory contains files and subdirectories, which contain more files and subdirectories, and so on.

The Root Directory



Confusion!

The real root directory is /

There is also another directory named *root* within the real root (i.e. /). But this is not the real root.

The Current Directory

- The **pwd** command (print working directory)

```
akahate@LAPTOP-E9C3P1LP:~$ pwd
/home/akahate
```

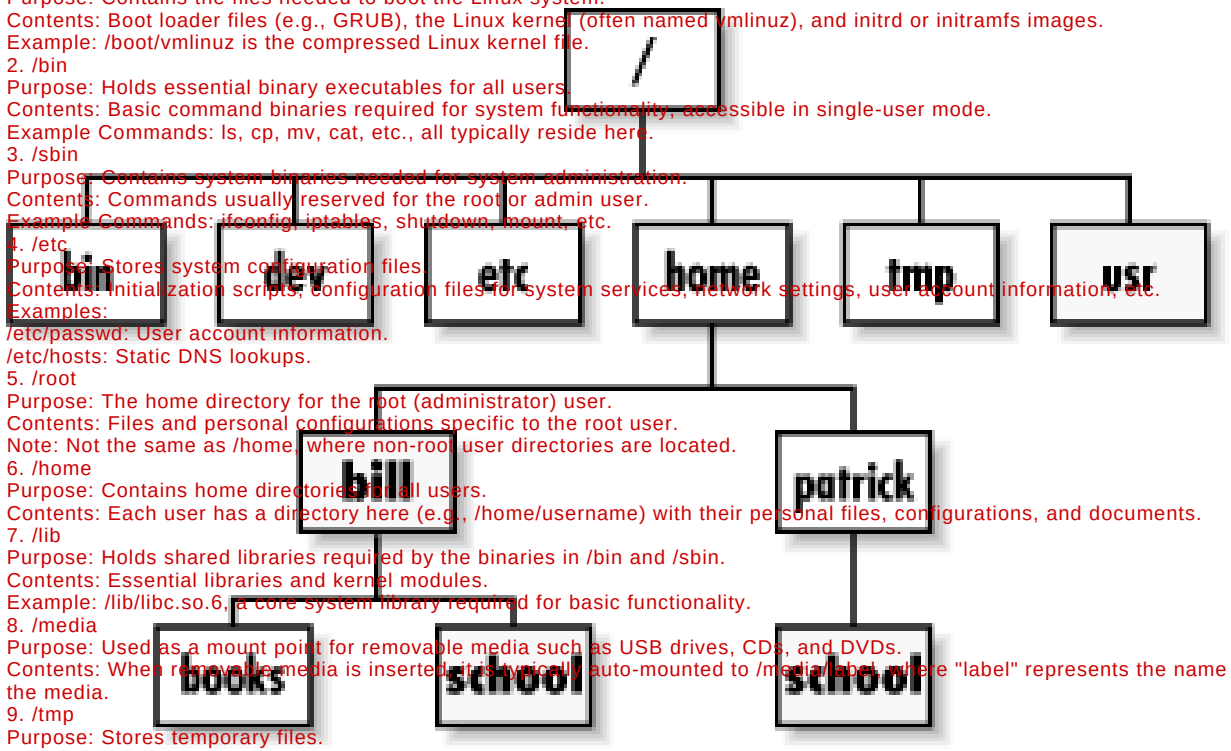
- When we first log in to our system (or start a terminal emulator session), our current working directory is set to our home directory.
- Each user account is given its own home directory, which is the only place the user is allowed to write files when operating as a regular user.

The Home Directory

- Each user on the system has a sub-directory inside *home*

boot
bin
sbin
etc
root
home
lib
media
tmp
usr
mnt
dev
proc
sys

1. /boot
Purpose: Contains the files needed to boot the Linux system.
Contents: Boot loader files (e.g., GRUB), the Linux kernel (often named vmlinuz), and initrd or initramfs images.
Example: /boot/vmlinuz is the compressed Linux kernel file.
2. /bin
Purpose: Holds essential binary executables for all users.
Contents: Basic command binaries required for system functionality, accessible in single-user mode.
Example Commands: ls, cp, mv, cat, etc., all typically reside here.
3. /sbin
Purpose: Contains system binaries needed for system administration.
Contents: Commands usually reserved for the root or admin user.
Example Commands: ifconfig, iptables, shutdown, mount, etc.
4. /etc
Purpose: Stores system configuration files.
Contents: Initialization scripts, configuration files for system services, network settings, user account information, etc.
Examples:
/etc/passwd: User account information.
/etc/hosts: Static DNS lookups.
5. /root
Purpose: The home directory for the root (administrator) user.
Contents: Files and personal configurations specific to the root user.
Note: Not the same as /home, where non-root user directories are located.
6. /home
Purpose: Contains home directories for all users.
Contents: Each user has a directory here (e.g., /home/username) with their personal files, configurations, and documents.
7. /lib
Purpose: Holds shared libraries required by the binaries in /bin and /sbin.
Contents: Essential libraries and kernel modules.
Example: /lib/libc.so.6, a core system library required for basic functionality.
8. /media
Purpose: Used as a mount point for removable media such as USB drives, CDs, and DVDs.
Contents: When removable media is inserted, it is typically auto-mounted to /media/label, where "label" represents the name of the media.
9. /tmp
Purpose: Stores temporary files.
Contents: Files that are only needed temporarily, often created by programs. This directory is cleared upon reboot.
Examples: Temporary data storage for installations or user sessions.
10. /usr
Purpose: Holds user-installed applications and utilities.
Contents: Subdirectories for user binaries (/usr/bin), libraries (/usr/lib), and documentation (/usr/share/doc).




```
atul@LAPTOP-E9C3P1LP:/$ cd home/
atul@LAPTOP-E9C3P1LP:/home$ ls
atul
atul@LAPTOP-E9C3P1LP:/home$
```

Other Directory Commands

- **ls**: To list the files and directories in the current working directory, we use the ls command
- **cd**: Change directory
  - **Absolute path names**: Begins with the root directory and follows the tree branch by branch until the path to the desired directory or file is completed.
  - For example, there is a directory on your system in which most of your system's programs are installed. The pathname of that directory is /usr/bin.

```
akahate@LAPTOP-E9C3P1LP:~$ pwd
/home/akahate
akahate@LAPTOP-E9C3P1LP:~$ cd /usr/bin
akahate@LAPTOP-E9C3P1LP:/usr/bin$ pwd
/usr/bin
```

More Directory Commands

- **mkdir**: Create a directory

```
akahate@LAPTOP-E9C3P1LP:~$ cd /home
akahate@LAPTOP-E9C3P1LP:/home$ ls
akahate  atul
akahate@LAPTOP-E9C3P1LP:/home$ cd akahate
akahate@LAPTOP-E9C3P1LP:~$ ls
akahate@LAPTOP-E9C3P1LP:~$ pwd
/home/akahate
akahate@LAPTOP-E9C3P1LP:~$ mkdir dir1
akahate@LAPTOP-E9C3P1LP:~$ ls
dir1
```

mkdir new_folder
mkdir dir1 dir2 dir3

Use the -p (parents) option. This will create all necessary parent directories if they don't already exist.

mkdir -p parent/child/grandchild

The -v (verbose) option outputs a message for each directory created, which is useful for confirming actions when creating multiple directories.

mkdir -v new_folder

You can specify permissions for the directory using the -m option, with the mode in octal format (similar to chmod):

mkdir -m 755 new_folder

ls [options] [directory_name]

Option Description
-l Displays a long listing format, showing detailed information about each file (permissions, size).
-a Lists all files, including hidden files (those starting with a .).
-h Displays file sizes in a human-readable format (e.g., KB, MB), useful with -l.
-R Recursively lists all files in subdirectories.
-t Sorts files by modification time, with the newest files listed first.
-r Reverses the order of the output.
-S Sorts files by size, with the largest files listed first.
-d

More Directory Commands

- **cp**: Copy files and directories

Command	Results
`cp file1 file2`	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created.
`cp -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`cp file1 file2 dir1`	Copy *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, all the files in *dir1* are copied into *dir2*. *dir2* must already exist.
`cp -r dir1 dir2`	Copy directory *dir1* (and its contents) to directory *dir2*. If directory *dir2* does not exist, it is created and will contain the same contents as directory *dir1*.

More Directory Commands

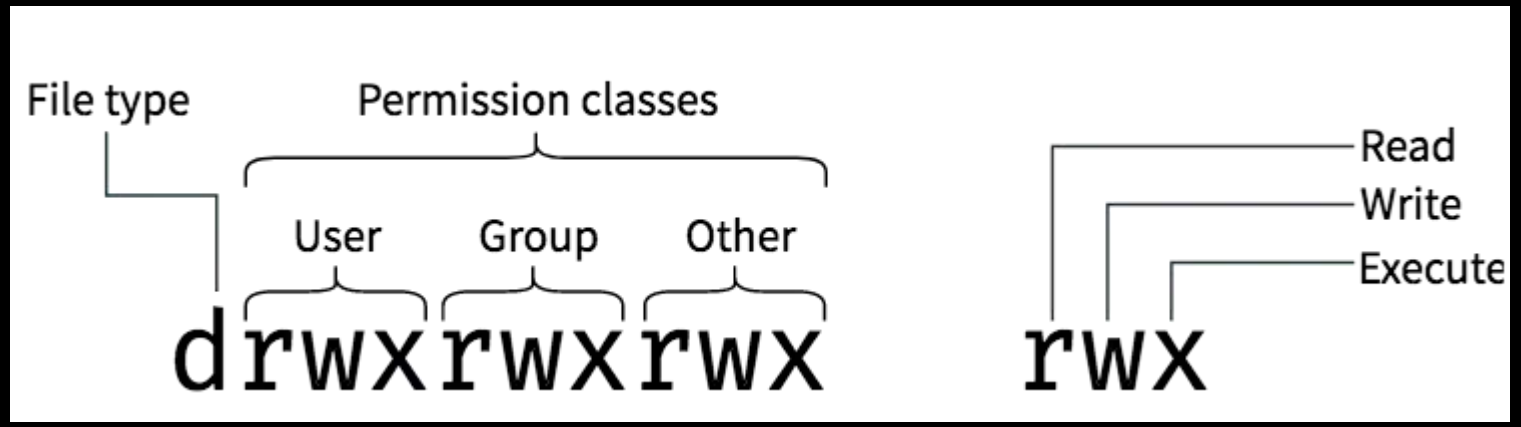
- **mv**: Move and rename files

Command	Results
`mv file1 file2`	Move *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created. **In either case, *file1* ceases to exist.**
`mv -i file1 file2`	Same as above, except that if *file2* exists, the user is prompted before it is overwritten.
`mv file1 file2 dir1`	Move *file1* and *file2* into directory *dir1*. *dir1* must already exist.
`mv dir1 dir2`	Move directory *dir1* (and its contents) into directory *dir2*. If directory *dir2* does not exist, create directory *dir2*, move the contents of directory *dir1* into *dir2*, and delete directory *dir1*.

More Directory Commands

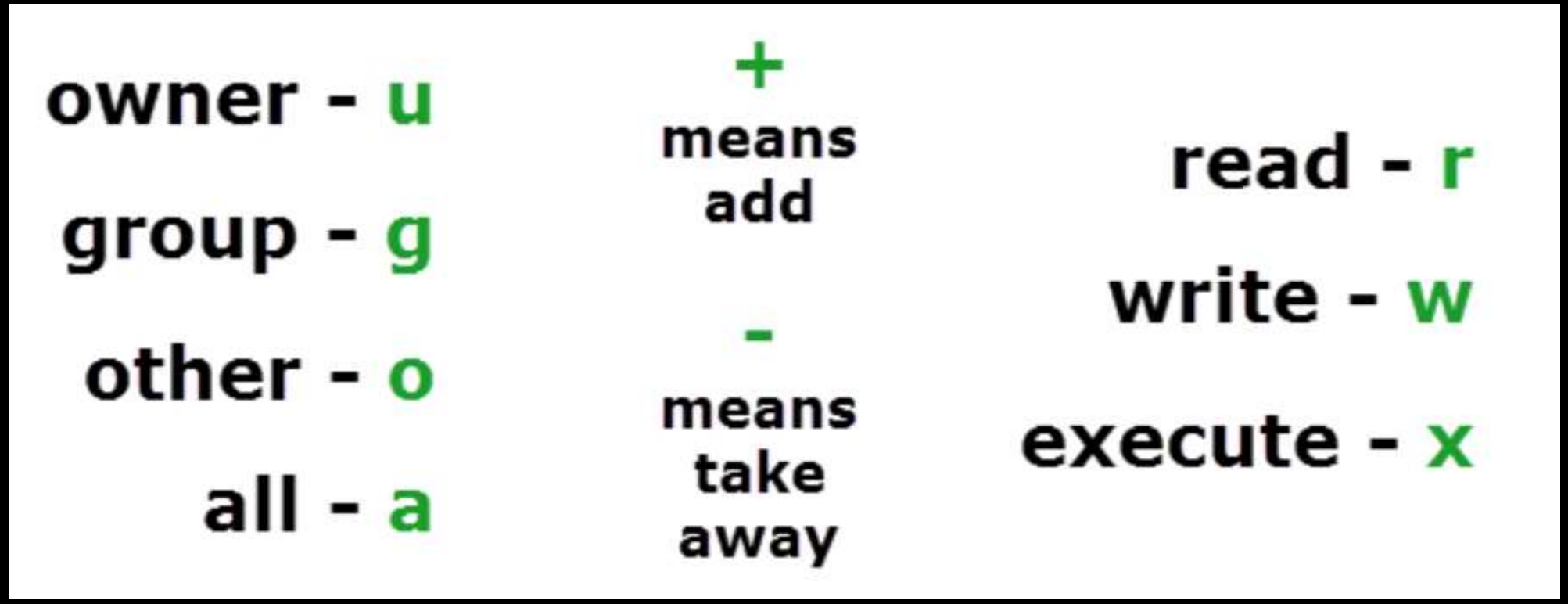
- **rm**: Delete files and directories

Option	Meaning
`-i, --interactive`	Before deleting an existing file, prompt the user for confirmation. **If this option is not specified, `rm` will silently delete files.**
`-r, --recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f, --force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v, --verbose`	Display informative messages as the deletion is performed.



File Permissions

How to Specify Who to Give Permissions To?



File Permission Examples

File Attributes	Meaning
`-rwx------`	A regular file that is readable, writable, and executable by the file's owner. No one else has any access.
`-rw-------`	A regular file that is readable and writable by the file's owner. No one else has any access.
`-rw-r--r--`	A regular file that is readable and writable by the file's owner. Members of the file's owner group may read the file. The file is world readable.
`-rwxr-xr-x`	A regular file that is readable, writable, and executable by the file's owner. The file may be read and executed by everybody else.
`-rw-rw----`	A regular file that is readable and writable by the file's owner and members of the file's owner group only.

File Permissions in Binary

#	Permission	rwx	Binary
7	read, write and execute	rwx	111
6	read and write	rw-	110
5	read and execute	r-x	101
4	read only	r--	100
3	write and execute	-wx	011
2	write only	-w-	010
1	execute only	--x	001
0	none	---	000

Understanding Numeric File Permissions

- Remember, Read permission = 4, Write permission = 2, Execute permission = 1

Requirement	Character-based permissions	Numeric representation
Give all permissions to all the users	4 + 2 + 1 for everyone	777
Give read-write permissions to all the users	4 + 2 for everyone	666
Owner should have all the permissions, and the others should have read and execute permissions	4 + 2 + 1 for the owner, 4 +1 for the others	755

Examples

- chmod 755 file1
- chmod u+x file1
- chmod 525 file1
- chmod g-w file1

What is a Shell Script?

- In the simplest terms, a **shell script** is a file containing a series of commands.
- The shell reads this file and carries out the commands as though they have been entered directly on the command line.
- The shell is distinctive, in that it is both a powerful command-line interface to the system and a scripting language interpreter.

Redirection using Pipe

- echo “5/2” | bc
- echo “scale=2; 5/2” | bc
- echo “scale=10; 5/2” | bc

Simply type bc in the terminal, and you can enter calculations line-by-line.

```
bc
5 + 3
8
10 / 4
2
```

Use echo and a pipe (|) to perform a calculation without opening an interactive session.

```
echo "5 + 3" | bc
8
```

By default, bc performs integer division. You can set the scale variable to specify the number of decimal places for floating-point division.

```
echo "scale=2; 10 / 3" | bc
3.33

echo "x=5; y=3; x*y" | bc
15

echo "2^3" | bc       # Calculates 2 raised to the power of 3
echo "scale=4; sqrt(25)" | bc
```

Redirection using >

- name=“Linux Student”
- echo $name > out.txt

echo $name: This command prints the value of the variable name (in this case, "Linux Student") to standard output (the terminal).

> out.txt: The > operator redirects this output to a file named out.txt. If out.txt already exists, it will be overwritten with the new content ("Linux Student").

After running this command, the file out.txt will contain:

Linux Student
If you want to append the content to out.txt instead of overwriting it, you can use >> instead of >.

PS Variables

- Prompt Statement Environment Variables
- PS1: Default interaction prompt
- PS2: Continuation interaction prompt
- PS3: Prompt used by the select statement inside a script

PS1: Default Interaction Prompt

This variable defines the primary shell prompt that appears when the shell is ready to accept a new command.

- Try this command: export PS1="\u@\h \w> “
  - The default prompt will be changed to show the user name, home, and working directory
- export PS1= export PS1="\u\w:\d-\t>«

\u: Username
\h: Hostname
\W: Current directory
\$: # for root, $ for normal users

- PS1="\$(date +%d.%m.%Y) > "
- export PS1

- PS1="\D{%F %T}>“

- export PS1="\u@\h [\$(date +%k:%M:%S)]> "

PS2

This is displayed when a command spans multiple lines (e.g., when a user doesn't complete the syntax, like an unmatched quote or backslash).

- When you issue a command that is incomplete, the shell will display a secondary prompt and wait for you to complete the command and hit Enter again. The default secondary prompt is > (the greater than sign), but can be changed by re-defining the PS2 shell variable.

- Before using PS2 …
- echo “this is a
- > test”

PS2

- After using PS2 …
- PS2=“continue …”
- echo “this is a
- continue... test”

```
atul@LAPTOP-E9C3P1LP [14:26:28]> echo "this is a
> test"
this is a
test
atul@LAPTOP-E9C3P1LP [14:26:34]> PS2="continue ..."
atul@LAPTOP-E9C3P1LP [14:26:47]> echo "this is a
continue ...test"
this is a
test
atul@LAPTOP-E9C3P1LP [14:26:55]>
```

PS3: Used in scripts for the prompt displayed during a select command.
PS4: The prompt displayed when debugging a shell script (using set -x).

How to Set System Variables
Temporarily (for current session): Use the export command in the terminal:
export PS1="[\u@\h \W]\$ "

Permanently (for all sessions): Add the export command to the shell configuration file (~/.bashrc, ~/.zshrc, or ~/.bash_profile depending on your shell):
echo 'export PS1="[\u@\h \W]\$ "' >> ~/.bashrc
source ~/.bashrc

The select Command and the PS3 (Prompt for Select command) Variable

- select_example_ps3.sh
- #!/bin/bash

Called ‘Shebang’ or ‘Shabang’ operator, because:
means sharp (as in C#)
! In comics is used such as Bang!

- PS3="What is the day of the week?: "

The PS3 variable in Linux/Unix is used specifically in shell scripts to customize the prompt displayed during the select command. The select command is a built-in shell feature that generates a menu-like interface for the user to choose options.

- select day in mon tue wed thu fri sat sun;
- do
- echo "You selected $day"
- break
- done

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./select_example_ps3.sh
1) mon
2) tue
3) wed
4) thu
5) fri
6) sat
7) sun
What is the day of the week?: 2
You selected tue
```

Variables

- A variable allows us to store useful data under convenient names
  - User-defined variables
  - Shell variables

User-defined Variables

- Example
  - INSTITUTE=“ACTS”
  - echo “Hello $INSTITUTE”

Variable Examples – Note the Difference

- #!/bin/bash
- COUNT=5
- echo “count = $COUNT”
- echo We have $COUNT oranges

Variable Assignment:

No spaces are allowed around the = in Bash.
Correct: COUNT=5
Incorrect: COUNT = 5

Quotes:
Use standard quotes (") instead of smart quotes (“ ”) to avoid syntax errors.

Printing Variables:
Use $ to reference the variable's value.
Example: echo "The value is $COUNT"

- #!/bin/bash
- COUNT=5
- COUNT = 6
- echo “count = $COUNT”
- echo We have $COUNT oranges

paces around the = are not allowed in variable assignment in Bash. The shell will interpret this line as a command COUNT with arguments = and 6, causing an error.

Variable Examples – Strings and Spaces

- #!/bin/bash
- MESSAGE=Hello
- echo “I want to say: $MESSAGE”

- #!/bin/bash
- MESSAGE=Hello there
- echo “I want to say: $MESSAGE”

Problem: The string Hello there contains a space, but it is not enclosed in quotes. Bash treats Hello and there as separate arguments, resulting in an error.

- #!/bin/bash
- MESSAGE=“Hello there”
- echo “I want to say: $MESSAGE”

Reading Multiple Inputs in User Variables

- read_example.sh

- #!/bin/bash

- read name age city

- echo $name
- echo $age
- echo $city

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./read_example.sh
test 30 pune
test
30
pune
```

Reading Multiple Inputs in User Variables – Better Formatted

- read_example_modified.sh
- #!/bin/bash

- read name age city

- echo "My name is: $name"
- echo "My age is: $age"
- echo "My city is: $city"

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./read_example_modified.sh
test 30 pune
My name is: test
My age is: 30
My city is: pune
```

Using echo with read

- #!/bin/bash

- echo –n “Enter your age:”
- read AGE
- echo “Your age is $AGE”

Adding Prompt to the read Command

- read_example_prompt.sh
- #!/bin/bash

- read -p "Enter your name: " name
- read -p "Enter your age: " age
- read -p "Enter your city: " city

- echo "My name is: $name"
- echo "My age is: $age"
- echo "My city is: $city"

Other useful options
-t: Timeout in seconds
-s: Secret

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./read_example_prompt.sh
Enter your name: test
Enter your age: 30
Enter your city: pune
My name is: test
My age is: 30
My city is: pune
```

Session 3: Shell Programming

Conditions – if Statement

- #!/bin/sh
- a=10
- b=20
- if [$a -eq $b]
- then
- echo “a is equal to b”
- fi
- if [$a -ne $b]
- then
- echo “a is not equal to b”
- fi

Same Example using if-else

- #!/bin/sh

- a=10
- b=20

- if [$a -eq $b]
- then
- echo “a is equal to b”
- else
- echo “a is not equal to b”
- fi

Check If a Number is Even or Odd

- #!/bin/bash
- read -p "Enter a number : " n
- rem=$(($n % 2))
- if [$rem -eq 0]
- then
- echo "$n is even number"
- else
- echo "$n is odd number"
- fi

if-elif-else

- #!/bin/bash

- echo "Enter a Number"
- read num

- if [$num -lt 0]
- then
- 	echo "Negative"
- elif [$num -gt 0]
- then
- 	echo "Positive"
- else
- 	echo "Neither Positive Nor Negative"
- fi

Max of three numbers

```
#!/bin/bash
echo "Enter Num1"
read num1
echo "Enter Num2"
read num2
echo "Enter Num3"
read num3
# To accept multiple inputs: read num1 num2 num3

if [ $num1 -gt $num2 ] && [ $num1 -gt $num3 ]
then
    echo $num1
elif [ $num2 -gt $num1 ] && [ $num2 -gt $num3 ]
then
    echo $num2
else
    echo $num3
fi
```

String Comparison

```
#!/bin/bash

# Shell script to check whether two
# input strings is equal

# take user input
echo "Enter string a : "
read a

echo "Enter string b : "
read b

# check equality of input
if [ "$a" = "$b" ]; then

    # if equal print equal
    echo "Strings are equal."
else
     # if not equal
    echo "Strings are not equal."
fi
```

Simple Calculator

```
#!/bin/bash

echo "A Simple Integer Calculator ..."

read -p "Enter the first number: " num1
read -p "Enter the first number: " num2
read -p "Enter the mathematical operator (+ - * / %): " operator

echo -n "You entered $num1 $operator $num2 and its result is "

if [ "$operator" = "+" ]; then
                result=$(( num1 + num2 ))
elif [ "$operator" = "-" ]; then
                result=$(( num1 - num2 ))
elif [ "$operator" = "*" ]; then
                result=$(( num1 * num2 ))
elif [ "$operator" = "/" ]; then
                result=$(( num1 / num2 ))
elif [ "$operator" = "%" ]; then
                result=$(( num1 % num2 ))
else
                result="undefined"
fi
```

Salary Calculator

- Write a shell script that computes the gross salary of an employee according to the following rules:
  - If basic salary is < 1500 then HRA = 10% of the basic and DA = 90% of the basic.
  - If basic salary is >= 1500 then HRA = Rs 500 and DA = 98% of the basic.
- The basic salary is entered interactively through the keyboard.
- Gross salary = Basic + Additional salary components, as appropriate.

salary.sh

- echo "Gross salary calculator ..."

- read -p "Enter Basic salary: " bsal

- if [$bsal -lt 1500]; then
- gsal=$((bsal+((bsal/100)*10)+(bsal/100)*90))

- else
- gsal=$(((bsal+500)+(bsal/100)*98))
- fi

- echo "Gross salary: $gsal"

Calculate Average using Bash Calculator

- #!/bin/bash

- read -p "Enter marks in subject 1: " marks1
- read -p "Enter marks in subject 2: " marks2
- read -p "Enter marks in subject 3: " marks3

- total=$(($marks1 + $marks2 + $marks3))

- echo "Total marks: $total"

- avg=$(echo "scale=4; $total / 3" | bc)

- echo "Average marks: $avg"

for Loop

- #!/bin/bash
- # basic for command
- for test in Alabama Alaska Arizona Arkansas California Colorado
- do
- echo The next state is $test
- done

C-Style for Loop

- #!/bin/bash
- # testing the C-style for loop – double brackets are needed
- for ((i=1; i <= 10; i++))
- do
- echo "The next number is $i"
- done

while Command

- #!/bin/bash
- # while command test
- var1=10
- while [$var1 -gt 0]
- do
- echo $var1
- var1=$[$var1 - 1]
- done

Table of a Number

- #!/bin/bash
- read -p "Enter The Number for which you want to Print Table: " n

- i=1
- while [$i -ne 11]; do
- table=$((i * n))
- echo "$table"
- i=$((i + 1))
- done

until Example

- #!/bin/bash
- # using the until command
- var1=100
- until [$var1 -eq 0]
- do
- echo $var1
- var1=$[$var1 - 25]
- done

break Example

- #!/bin/bash
- # breaking out of a for loop
- for var1 in 1 2 3 4 5 6 7 8 9 10
- do
- if [$var1 -eq 5]
- then
- break
- fi
- echo "Iteration number: $var1"
- done
- echo "The for loop is completed"

grep – Searching through text

- grep searches text files for the occurrence of a specified **regular expression** and outputs any line containing a match to standard output.
- Has many options

grep PATTERN file
e.g. grep “tiger” animals.txt

Option	Description
-i	Ignore case. Do not distinguish between upper- and lowercase characters. May also be specified --ignore-case.
-v	Invert match. Normally, grep prints lines that contain a match. This option causes grep to print every line that does not contain a match. May also be specified --invert-match.
-c	Print the number of matches (or non-matches if the -v option is also specified) instead of the lines themselves. May also be specified --count.
-l	Print the name of each file that contains a match instead of the lines themselves. May also be specified --files-with-matches.
-L	Like the -l option, but print only the names of files that do not contain matches. May also be specified --files-without-match.
-n	Prefix each matching line with the number of the line within the file. May also be specified --line-number.
-h	For multifile searches, suppress the output of filenames. May also be specified --no-filename.

Another Example

- Create a test_grep directory in home/<user name> directory (e.g. /home/atul/test_grep
- Create a file GreatGatsby.txt in home/atul/test_grep directory (Available at https://www.gutenberg.org/cache/epub/64317/pg64317.txt)

- grep “chapter” GreatGatsby.txt

```
atul@LAPTOP-E9C3P1LP:~/test_grep$ grep "chapter" GreatGasby.txt
chapter of Simon Called Peter—either it was terrible stuff or the
```

- grep “Gatsby” GreatGatsby.txt

```
chapter of Simon Called Peter—either it was terrible stuff or the
atul@LAPTOP-E9C3P1LP:~/test_grep$ grep "Gatsby" GreatGasby.txt
 The Project Gutenberg eBook of The Great Gatsby, by F. Scott Fitzgerald
Title: The Great Gatsby
                                The Great Gatsby
human heart. Only Gatsby, the man who gives his name to this book, was
```

… and many more lines

More Examples

- grep “ate” GreatGatsby.txt
  - Shows all occurrences of the word ate, including text inside other words

```
atul@LAPTOP-E9C3P1LP:~/test_grep$ grep "ate" GreatGasby.txt
This eBook is for the use of anyone anywhere in the United States and
www.gutenberg.org. If you are not located in the United States, you
will have to check the laws of the country where you are located before
Release Date: January 17, 2021 [eBook #64317]
```

- To restrict to whole words only:
- grep -w “ate” GreatGatsby.txt

```
atul@LAPTOP-E9C3P1LP:~/test_grep$ grep -w "ate" GreatGasby.txt
he and the other man ate together. Wilson was quieter now, and
lunch with me,’ I said. He ate more than four dollars’ worth of food
atul@LAPTOP-E9C3P1LP:~/test_grep$
```

More grep Examples

- grep three file1 -> Show all lines having three in file1
- grep t file1
- grep -v t file1 -> Show all lines not having t in file1
- grep -n t file1 -> Line numbers
- grep -c t file1 -> Count of t
- grep -e t -e f file1 -> Multiple patterns
- grep [tf] file1 -> Same as above

Create this file

- unix is great os. unix is opensource. unix is free os.
- learn operating system.
- Unix linux which one you choose.
- uNix is easy to learn.unix is a multiuser os.Learn unix .unix is powerful.

Try these Commands

- grep -i "UNix" geekfile.txt -> Case insensitive search
- grep -c "unix" geekfile.txt -> Count of matching lines
- grep -l "unix" * -> Files containing the given string
- grep -w "unix" geekfile.txt -> Match whole word unix
- grep -o "unix" geekfile.txt -> Display only the matching pattern, not the whole string
- grep -n "unix" geekfile.txt -> Show lines numbers of matching lines
- grep -v "unix" geekfile.txt -> Invert the search pattern
- grep "^unix" geekfile.txt -> Match lines starting with this string
- grep "os$" geekfile.txt -> Match lines ending with this string

sed

- Stream editor, as opposed to a normal interactive editor
- It is a text editor without an interface. You can use it from the command line to manipulate text in files and streams.
- With sed you can do all of the following:
- Select text
- Substitute text
- Add lines to text
- Delete lines from text
- Modify (or preserve) an original file

sed Example

- **echo "This is a test" | sed 's/test/big test/’**
- The echo command sends “This is a test” into sed using pipe symbol
- The simple substitution rule (s = substitute) is applied
- sed searches the input text for an occurrence of the first string (test) and replaces with the second (big test)
- Output will be sent to the terminal window

sed Examples

Note: File contents are not changed, only output is changed.

- Create python.txt
  - Python is a very popular language.
  - Python is easy to use. Python is easy to learn.
  - Python is a cross-platform language
- sed '2 s/Python/perl/g' python.txt → Replacement in the second line, g means all the instances in that line (here, 2)
- sed 's/Python/perl/g2' python.txt → Replace the second occurrence on every line, wherever there is a match
- sed '1 s/Python/perl/' python.txt → Replace the first occurrence on the first line
- sed '$s/Python/Bash/' python.txt → Replace the last line's first occurrence

sed and Delete

- Create os.txt
  - Windows
  - Linux
  - Android
  - OS
- sed '/OS/d' os.txt → Delete lines containing the word OS
- sed '/Linux/,+2d' os.txt → Delete line containing the word Linux and next two lines

sed and Insert

- Create myfile
  - This is a test and I like this test.
  - This is the next line of the test script.
- sed '2i\This is the inserted line.' myfile → Insert a new second line
- sed '2a\This is the appended line.' myfile → Append a new line (Note the position of the new line)
- i is *before* and a is *after*

View a Range of Lines

- Create sedtest.txt
- This is a demo text file.
- It is an amazing file that will help us all.
- The sed command is also great for stream editing.
- Want to learn how to use the command?
- This is another line in the file.
- This is the third general line in the file.
- This file is named as textfile.
- This is a apple.
- This is a orange.

- **sed -n '3,5p' sedtest.txt** => Display lines between 3 and 5
- **sed '3,5p' sedtest.txt** => Display lines between 3 and 5
- **sed ‘3,5d' sedtest.txt** => Display lines other than between 3 and 5

The -n flag prevents sed from displaying the pattern space at the end of each cycle. The p argument stands for print and is used to display the matched lines to the user.
The d flag deletes the matched strings from the output and displays the rest of the content.

sed | More Examples

- sed -n -e '1,2p' -e '5,6p' sedtest.txt => Display non-consecutive lines
- The -e flag helps in executing multiple actions using a single command.

- sed s/originalword/replaceword/g filename => Replace something
- sed s/amazing/super/g sedtest.txt
- The s argument denotes substitution and the g command is used to replace the matched content with the specified replacement content.

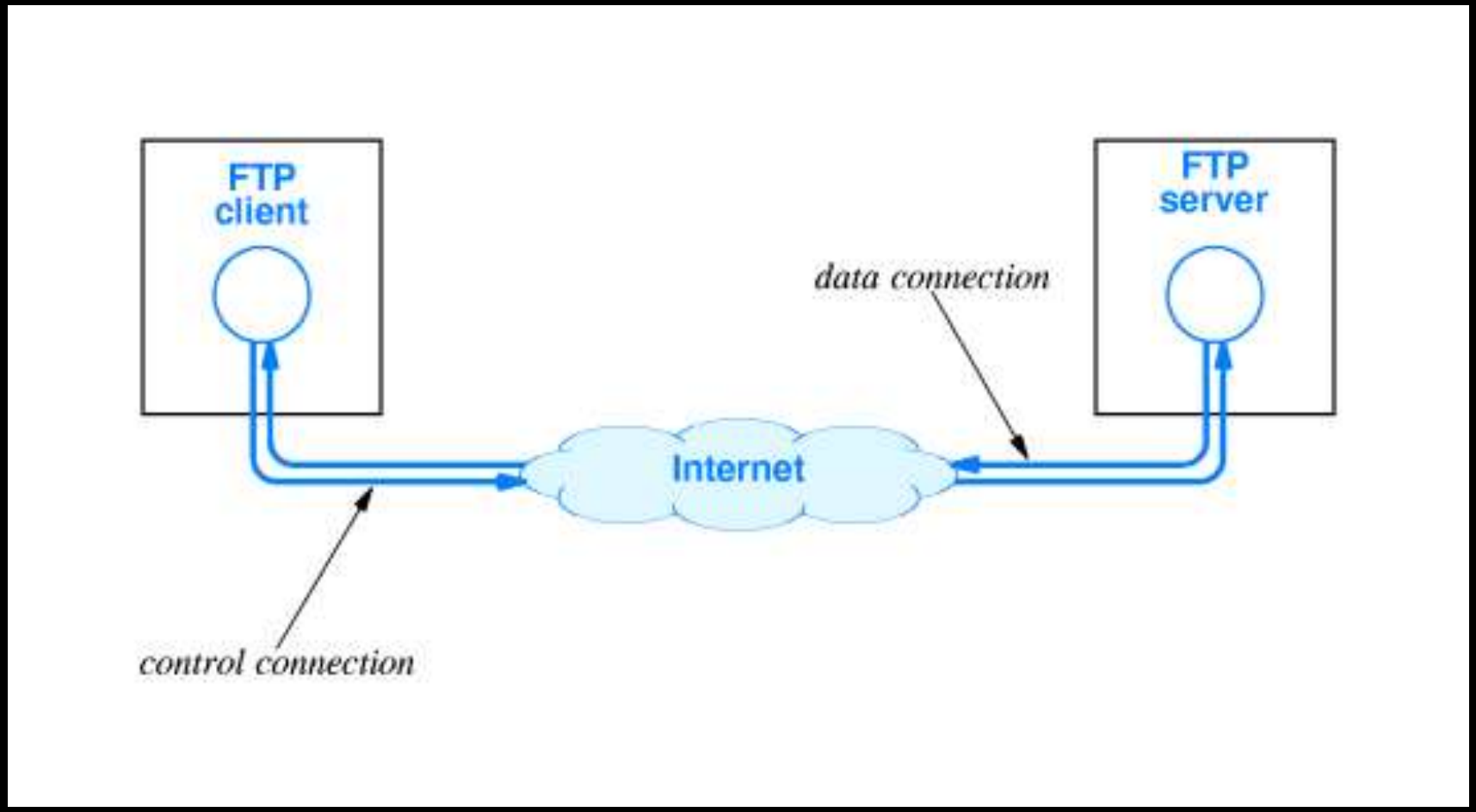
sed | More Examples

- sed '2,5s/This/That/g’ sedtest.txt => Substitute words in a range of lines (2 to 5 in this example)
- sed 's/amazing/super/g;s/command/utility/gi' textfile.txt => Multiple substitutions

- sed -e '/orange/ s/a/an/g’ sedtest.txt
- Replace the word a with an if the word orange is present in the line

FTP, Telnet, etc

FTP Concept


FTP
client
FTP
server
data connection
Internet
control connection

FTP (File Transfer Protocol)

- One of the earliest Internet protocols for file transfers
- Allows downloading (server-to-client) or uploading (client-to-server) of files remotely
- Made up of two parts
  - FTP server: Has files that can be downloaded, or it can also receive files from the users
  - FTP client: Downloads the require files from an FTP server, or it can upload files to the FTP server also

Setting up FTP Server on Ubuntu (on VirtualBox)

- Step 1: A common open-source FTP utility used in Ubuntu is vsftpd.
- sudo apt install vsftpd

- Step 2: Start the FTP server
- sudo systemctl start vsftpd

- Step 3: Backup configuration files for safety
- sudo cp /etc/vsftpd.conf /etc/vsftpd.conf_default

Setting up FTP Server on Ubuntu

- Step 4: Create an FTP user, so that a client can access the FTP server
- sudo useradd -m testuser
- sudo passwd testuser

- Step 5: Allow FTP through the firewall
- sudo ufw allow 20/tcp
- sudo ufw allow 21/tcp

- Step 6: Connect to the FTP server locally
- sudo ftp Ubuntu
- (Here, Ubuntu is considered to be our Linux machine name, so replace it with the actual machine name)

Accessing the FTP Server from our Windows Machine

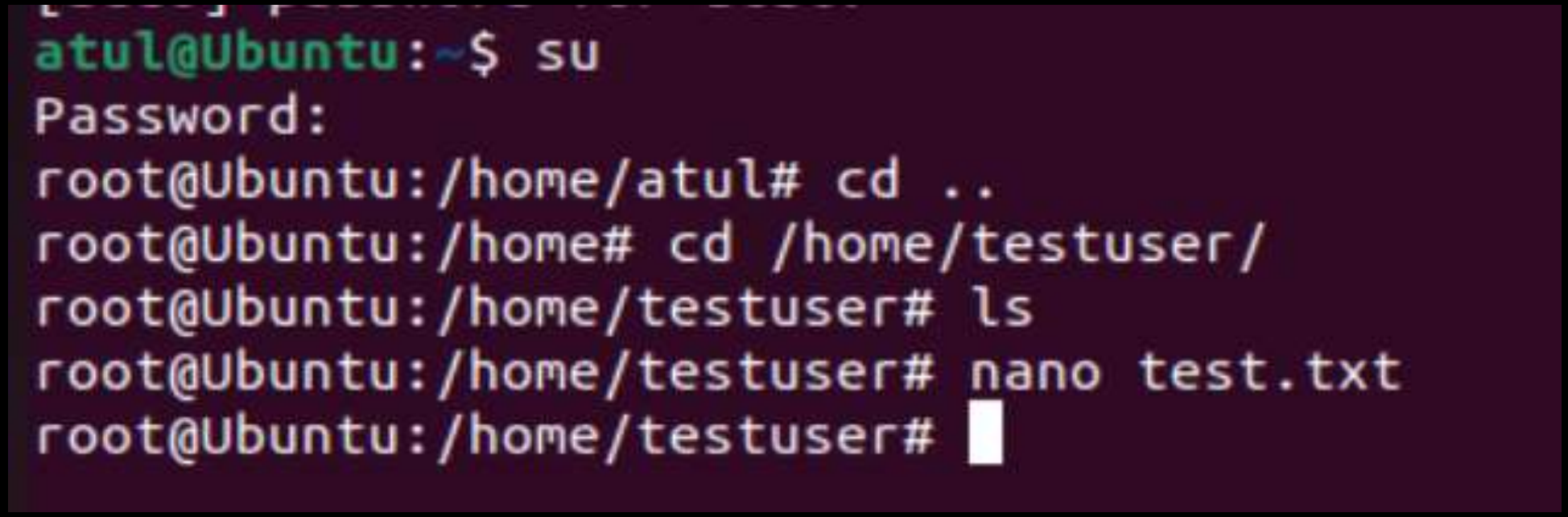
- Open command prompt and type **ftp**
- ftp> **open**
- To 192.168.0.24 … This is the IP address of the Ubuntu machine – to find it, we can type *ip a* or *ifconfig* on the Ubuntu machine's terminal
- Connected to 192.168.0.24.
- 220 (vsFTPd 3.0.5)
- 200 Always in UTF8 mode.
- User (192.168.0.24:(none)): testuser
- 331 Please specify the password.
- Password:
- 230 Login successful.

Accessing the FTP Server from our Windows Machine

- ftp> stat
- Connected to 192.168.0.24.
- Type: ascii; Verbose: On ; Bell: Off ; Prompting: On ; Globbing: On
- Debugging: Off ; Hash mark printing: Off .
- ftp> pwd
- 257 "/home/testuser" is the current directory
- ftp> dir

Checking for Files

- Create a file on the FTP server (i.e. Ubuntu machine)



Now Check in Windows FTP Client

- ftp> pwd
- 257 "/home/testuser" is the current directory
- ftp> dir
- 200 PORT command successful. Consider using PASV.
- 150 Here comes the directory listing.
- -rw-r--r-- 1 0 0 14 Mar 23 19:10 test.txt
- 226 Directory send OK.
- ftp: 69 bytes received in 0.01Seconds 8.63Kbytes/sec.

Download a File using FTP

- ftp> get test.txt
- 200 PORT command successful. Consider using PASV.
- 150 Opening BINARY mode data connection for test.txt (14 bytes).
- 226 Transfer complete.
- ftp: 14 bytes received in 0.00Seconds 14000.00Kbytes/sec.

Telnet and SSH

- **Telnet (Terminal + Network)** is an Internet protocol for allowing a client computer to connect to a remote server computer and work on the server computer
- Somewhat obsolete (outdated)
- Now we generally use **SSH (Secure Shell)** instead of Telnet
- SSH also allows a remote client computer to connect to a server over the Internet
  - Very useful in **cloud computing**

Setting up Telnet Server on Ubuntu

- Step 1: Install Telnet server
- sudo apt install telnetd -y

- Step 2: Check the Telnet status
- sudo systemctl status inetd

- Step 3: Configure firewall for Telnet
- sudo ufw enable
- sudo ufw allow 23

- Step 4: Test
- telnet 192.168.0.24

Accessing Telnet from Windows

- On the command prompt, type telnet
- If you get an error, search for *Turn Windows Features On/Off* and enable *Telnet* there, then retry telnet
- On the telnet prompt, o 192.168.0.24
- User id: testuser, Password: testuser

Setting up SSH on Ubuntu Server

- Step 1: Install SSH server
- sudo apt install openssh-server

- Step 2: Check the SSH status
- sudo systemctl status ssh

- Step 3: Configure firewall for SSH
- sudo ufw allow ssh

- Step 4: Restart if necessary
- sudo service ssh restart

Access SSH using Putty

- Open Putty, enter IP address as 192.168.0.24 and connect
- User id: testuser, Password: testuser

Sessions 4-6: Process Management

Program and Process

Program (Passive, On the Disk)

Set of instructions to solve a problem, written in a programming language

Process (Active, in Memory)

When a program starts running, it becomes a process; which means the operating system has to track it

Process Concept

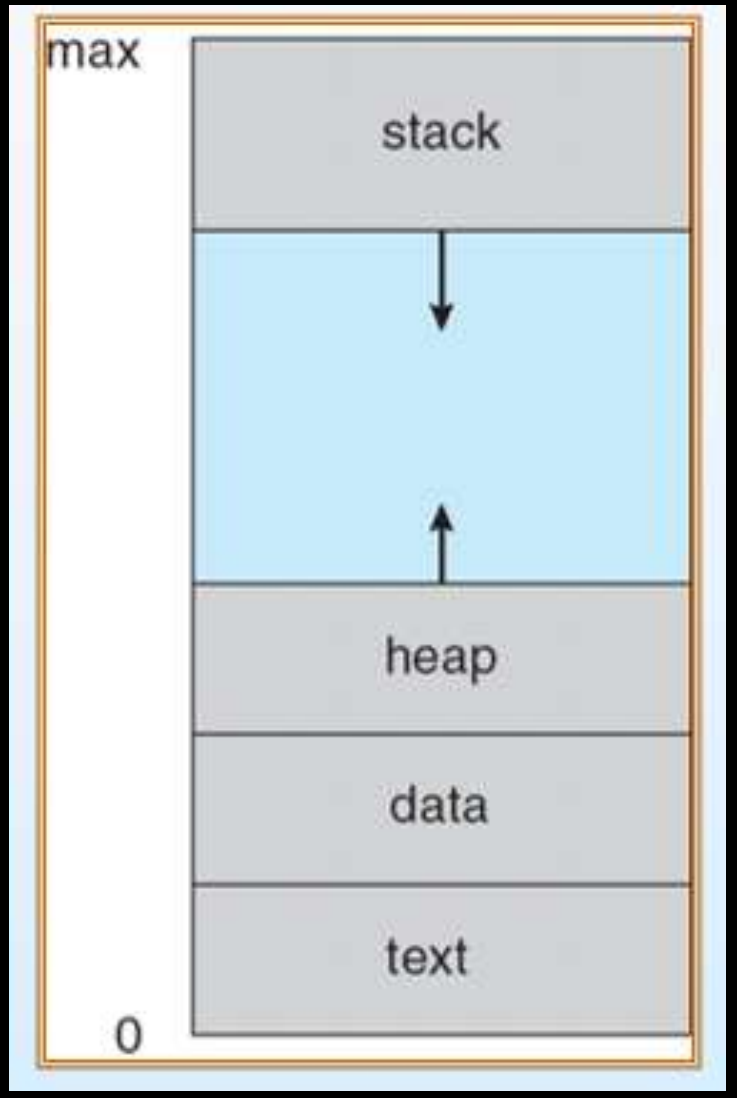
- A **process** is a program in execution. It is a unit of work within the system. Program is a passive entity, process is an active entity.
- Process needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **Program Counter (PC)** specifying location of next instruction to execute
  - Process executes instructions sequentially, one at a time, until completion
- Multi-threaded process has one program counter per thread
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs
  - Concurrency by multiplexing the CPUs among the processes / threads

Process Management

- Creating and deleting both user and system processes
- Suspending and resuming processes
- Providing mechanisms for process synchronization
- Providing mechanisms for process communication
- Providing mechanisms for deadlock handling

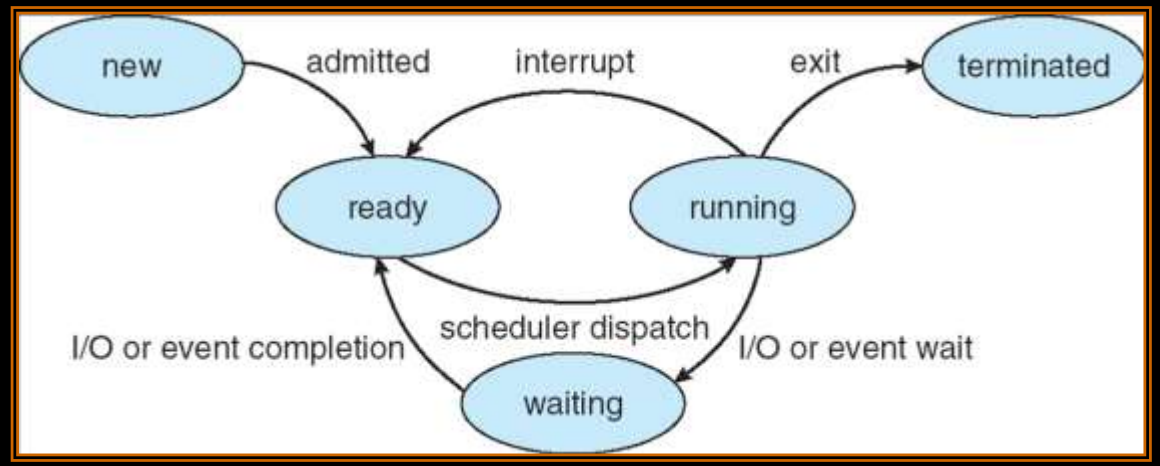
Process in Memory

- A process includes:
  - program counter
  - stack
  - data section



Process States

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



Process Control Block (PCB)

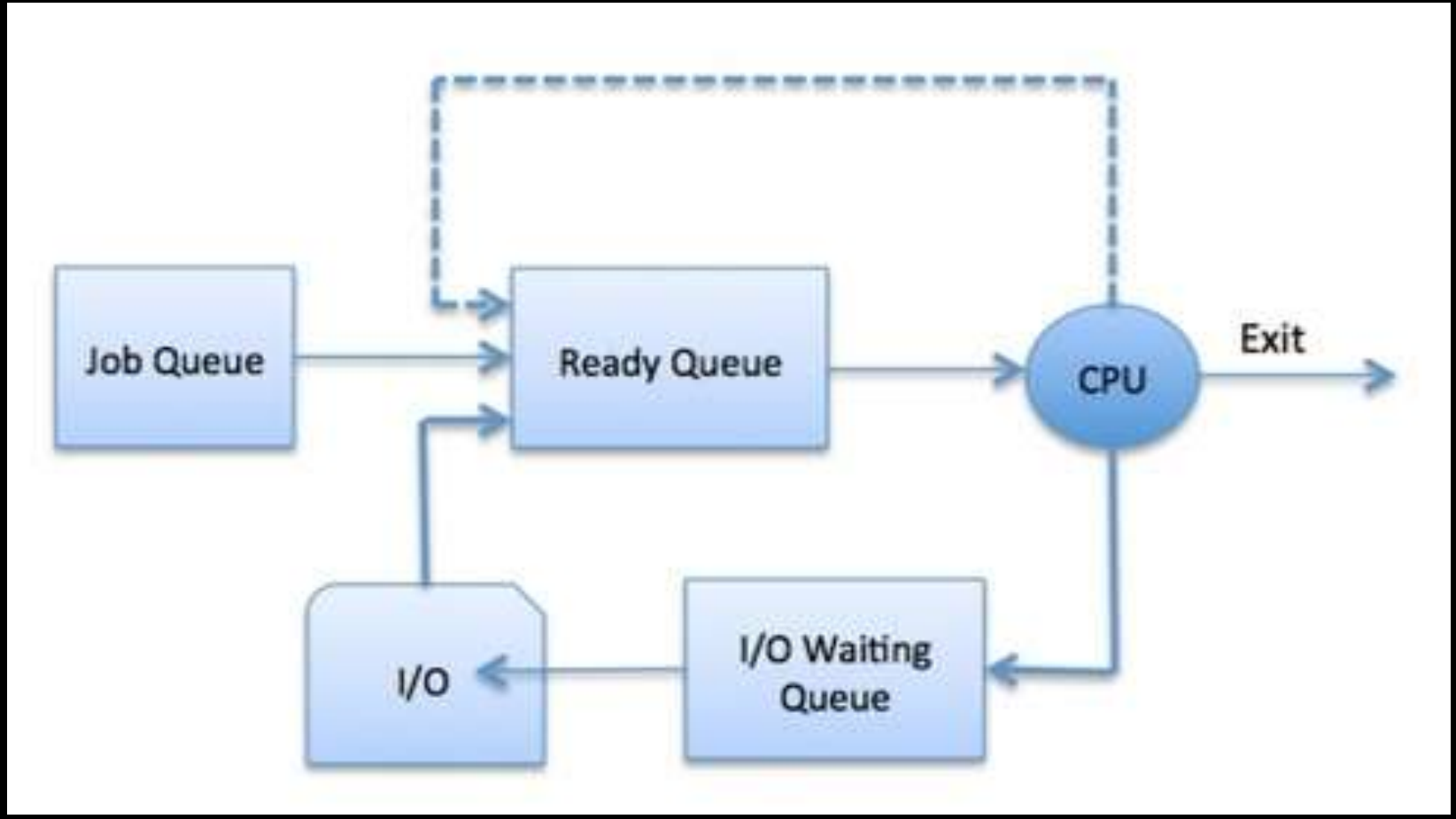
Process Scheduling

- **Process scheduling** Remove the running process from the CPU and select another process on the basis of a particular strategy.
- Process scheduling is an essential part of a **Multiprogramming operating system**.
- Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded processes share the CPU one after the other using **time multiplexing**.

Scheduling Queues

- **Scheduling queues** refers to queues of processes or devices.
- When a process enters into the system, then this process is put into a scheduling queue (also called as **job queue**).
- This queue consists of all processes in the system.
- The operating system also maintains other queues such as device queue. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

Process Scheduling



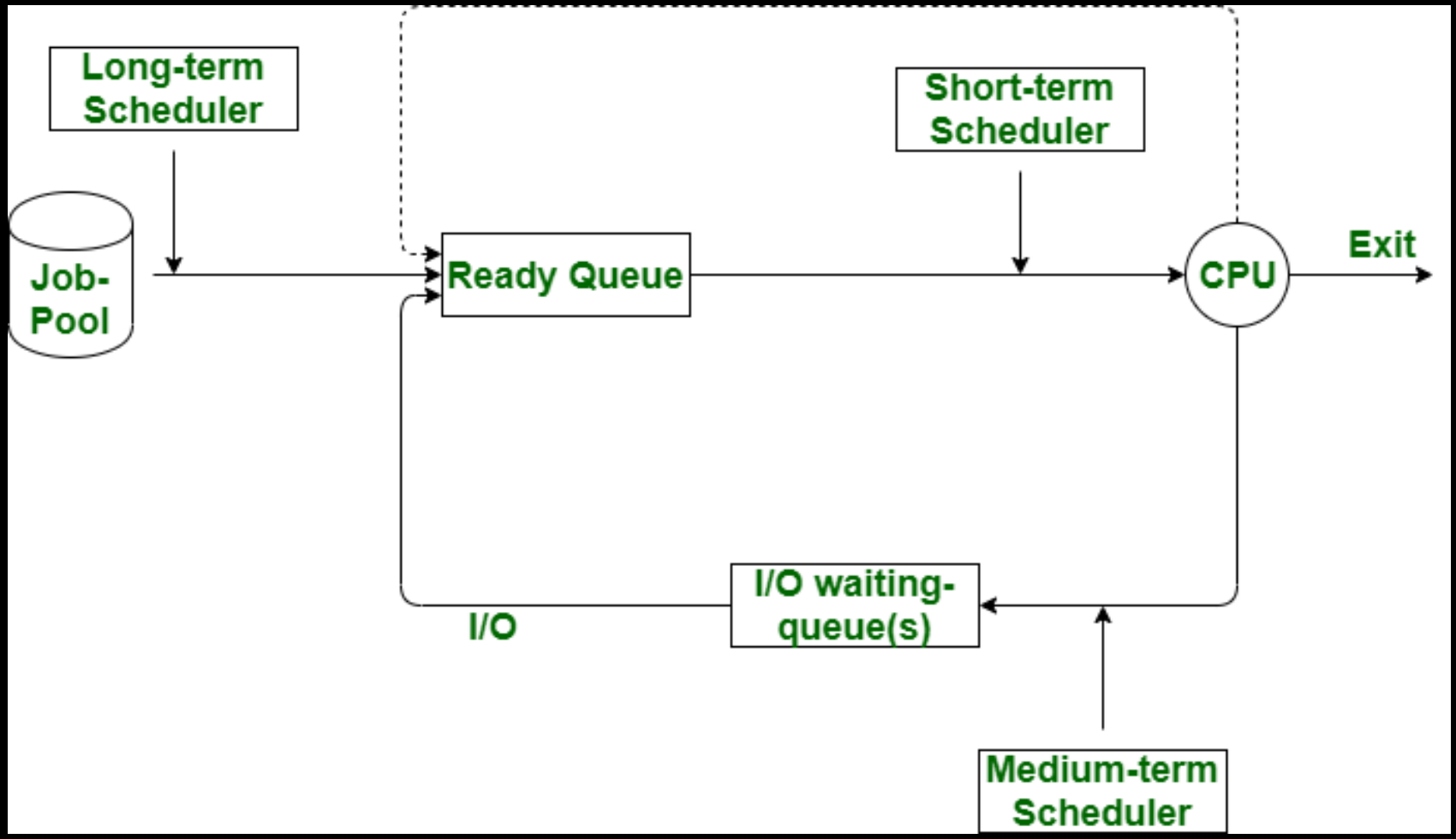
Scheduling Queues

- A newly arrived process is put in the ready queue. Processes wait in ready queue for the CPU. Once the CPU is assigned to a process, then the selected process will execute.
- While executing the process, any one of the following events can occur.
  - The process could issue an I/O request and then it would be placed in an I/O queue.
  - The process could create new sub process and will wait for its termination.
  - The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

Scheduler Types

- A scheduler is a special type of system software which handles process scheduling in various ways. Its main task is to select the jobs to be submitted into the system and to decide which process to run.
- Schedulers are of three types
  - Long Term Scheduler
  - Short Term Scheduler
  - Medium Term Scheduler

Scheduler Types



Long Term Scheduler

- It is also called **job scheduler**.
- **Long term scheduler** determines which programs are admitted to the system for processing.
- Long term/Job scheduler selects processes from the queue and loads them into memory for execution.
- Process loads into the memory for CPU scheduling.
- The primary objective of the job scheduler is to provide a balanced mix of jobs, such as I/O bound and processor bound.
- It also controls the **degree of multiprogramming** (Number of new processes and departing processes).

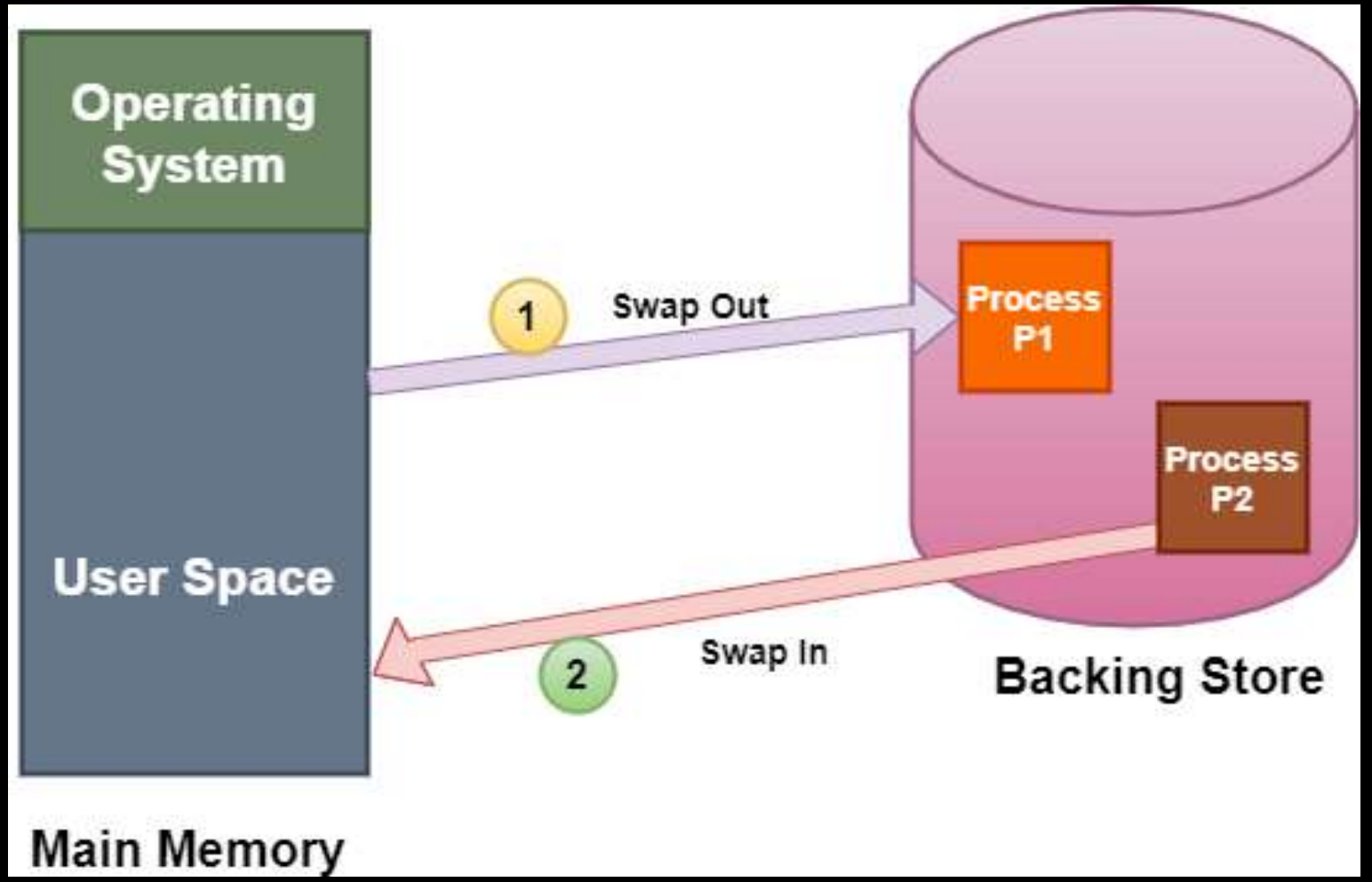
Short Term Scheduler

- It is also called **CPU scheduler**.
- Main objective is increasing system performance in accordance with the chosen set of criteria. It is in charge of moving a ready state process to the running state.
- CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
- **Short term scheduler** also known as **dispatcher**, executes most frequently and makes the fine grained decision of which process to execute next.
- Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

- **Medium term scheduling** is part of the **swapping** (Explained below).
- It removes the processes from the memory.
- It reduces the degree of multiprogramming.
- The medium term scheduler is in-charge of handling the swapped out-processes.
- **Swapping**: Running process may become suspended if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage. This step is called swapping, and the process is said to be swapped out or rolled out. (See next slide)

Swapping


Operating System
User Space
Main Memory
1
Swap Out
2
Swap In
Process P1
Process P2
Backing Store

CPU Scheduling (i.e. Short Term Scheduling)

- **Resources**: the things operated on by processes.
- Resources ranges from CPU time to disk space, to I/O channel time.
- Resources fall into two classes:
  - **Preemptive**: OS can take resource away. Use it for something else, and then give it back later.
    - Examples: processor
  - **Non-preemptive**: Once given, it cannot be reused until the process gives it back.
    - Examples: file, terminal
- Anything is preemptive if it can be saved and restored.

Criteria for a Good CPU Scheduling Algorithm

1. Fairness: Every process gets its fair share of CPU (i.e. **time slice**).
2. Efficiency (utilization): Keep CPU busy.
3. Response time: Minimize response time for interactive users.
4. Throughput: Maximize jobs per hour.
5. Minimize overhead (**Context switch**).

Scheduling Algorithms

- **First-Come First-Served (FCFS)**
- **Round Robin (RR)**
- **Priority**
- **Shortest Job First (SJF) / Earliest Deadline First (EDF)**

Preemptive and Non-preemptive Algorithms

- Preemptive algorithm
  - An algorithm that can be interrupted by an operating system if there is a need to do so
  - This means that if a process is a part of a preemptive algorithm, then while it is running, it can be interrupted
- Non-preemptive algorithm
  - An algorithm that cannot be interrupted by an operating system even if there is a need to do so
  - This means that if a process is a part of a non-preemptive algorithm, then while it is running, it cannot be interrupted

First Come First Serve (FCFS)

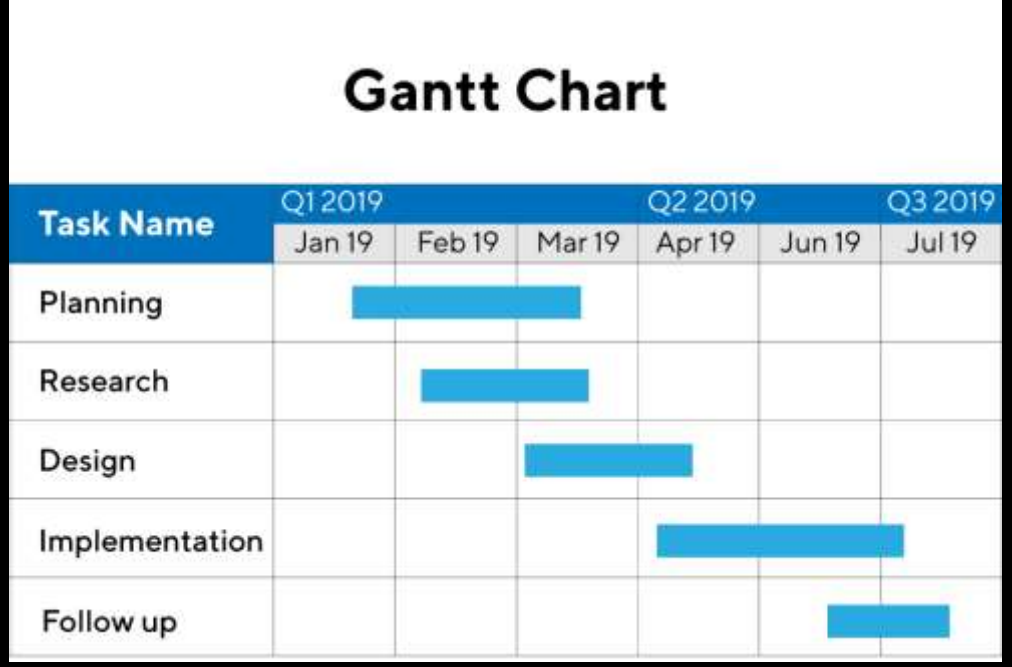
- Also called **First In First Out (FIFO)**
- FCFS is simplest of CPU Scheduling Algorithms, which executes the process that comes first.
- It is non-preemptive algorithm.
- Process that comes in ready queue first gets to be executed by the CPU first, then second one, then third one, and so on.
- The arrival time of processes is deciding factor here.

Measuring Process Timings

- **Arrival time**: Time when the process arrived in the process queue
- **Burst time**: The CPU time used by the process
- **Completion time**: The time at which the process got completed
- **Turn around time**: The time between its turnaround
- **Waiting time**: The time for which the process needed to wait, due to lack of CPU attention

- Turn Around Time = Completion Time - Arrival Time
- Waiting Time = Turnaround time - Burst Time

Gantt Chart Concept



- A **Gantt chart** is commonly used in project management
- One of the most popular and useful ways of showing activities (tasks or events) displayed against time
- Each activity is represented by a bar; the position and length of the bar reflects the start date, duration and end date of the activity
- It is used in showing processes during CPU scheduling activities

Example

- Example: There are 5 processes with process ID P0, P1, P2, P3 and P4. P0 arrives at time 0, P1 at time 1, P2 at time 2, P3 arrives at time 3 and Process P4 arrives at time 6 in the ready queue.

Process ID	Arrival time	Burst time	Completion time	Turnaround time	Waiting time
0	0	2	2	2	0
1	1	6	8	7	1
2	2	4	12	10	6
3	3	9	21	18	9
4	6	12	33	27	15

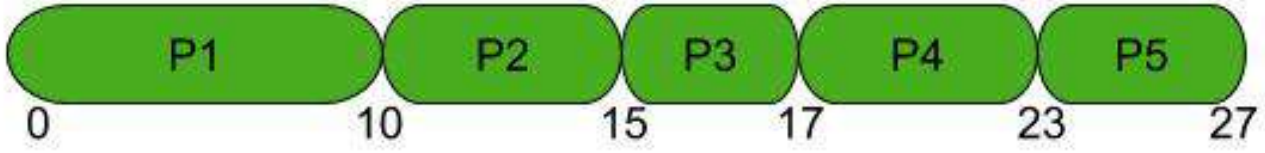
Turn Around Time = Completion Time - Arrival Time

Waiting Time = Turnaround time - Burst Time

FCFS Example

PROCESS	ARRIVAL TIME	BURST TIME
P1	0	10
P2	3	5
P3	5	2
P4	6	6
P5	8	4

GANTT CHART



FCFS Problem

- Suppose there are four processes with process ID's P1, P2, P3, and P4 and they enter into the CPU as follows:

Process ID	Arrival Time (milliseconds)	Burst Time (milliseconds)
P1	0	5
P2	2	3
P3	6	2
P4	7	3

- Calculate total and average turnaround time and waiting time.

Solution

Gant Chart:

P ID	Arrival Time	Burst Time	Completion time (milliseconds)	Turn Around Time (milliseconds)	Waiting Time (milliseconds)
P1	0	5	5	5	0
P2	2	3	8	6	3
P3	6	2	10	4	2
P4	7	3	13	6	3

```
Total Turn around Time = 5 + 6 + 4 + 6
        = 21 milliseconds
Average Turn Around Time= Total Turn Around Time / Total No. of Processes
        = 21 / 4
        = 5.25 milliseconds

Total Waiting Time = 0 + 3 + 2 + 3
        = 8 milliseconds
Average Waiting Time = Total Waiting Time / Total No. of Processes
        = 8 / 4
        = 2 milliseconds
```

Round Robin (RR)

- The **Round Robin (RR)** scheduling algorithm is a preemptive scheduling algorithm.
- It uses concept of **time slice**.
- Process at the beginning of ready queue gets chance to be executed first but only for span of one time slice.
- As new and more processes get added to ready queue, ongoing process gets preempted and gets added to end of ready queue.
- Next process gets the chance, again for span of one time slice.
- This algorithm is designed for time-sharing systems.

Round Robin (RR) – Key Points

- If the process has a CPU burst of less than time slice, then as soon as it is over, the CPU resources are released voluntarily. The next process in the ready queue occupies the CPU.
- If the CPU burst is larger than the slice, the timer will go off and the process is preempted, its state is saved by context switching and process is put at the end of the ready queue. After this, the next process in the ready queue takes over.

Round Robin (RR) Example

- Consider 4 processes where time slice is 3

Process	Burst Time
P1	10
P2	2
P3	5
P4	6

- Calculate average waiting time

Explanation

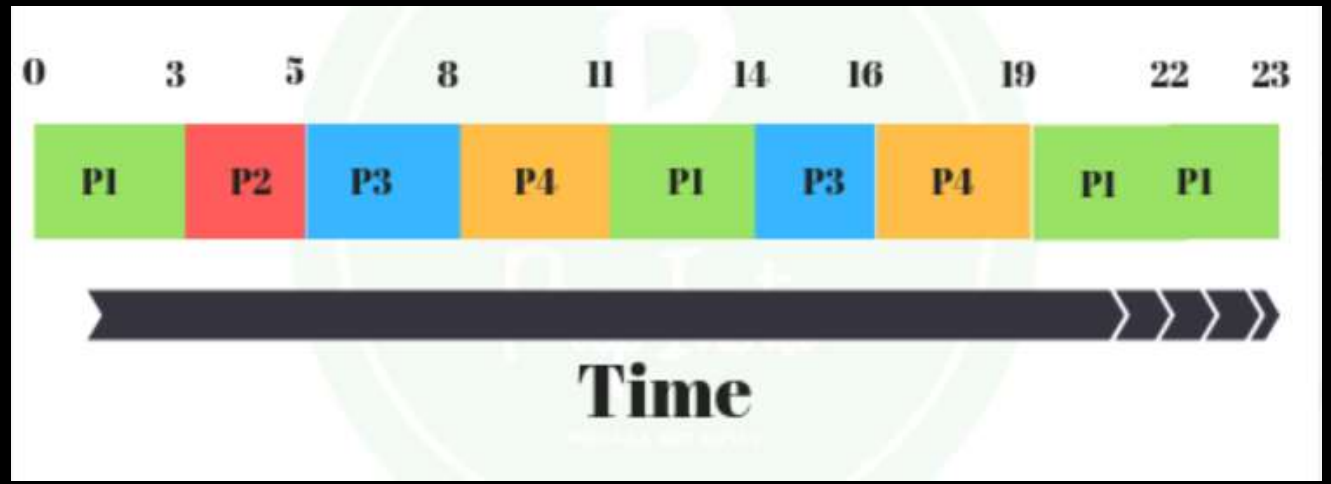
- P1 is executed for 3ms. Then P2 is executed. As, P2 is only 2ms long, So after 2ms, the next process occupies the CPU.
- P3 then is executed for 3ms. So, P3 still needs 2ms. Then, P4 is executed for 3ms. P4 still needs 3ms.
- P1 is again executed for 3ms. P1 still needs 4ms.
- P3 is executed for 2ms. Then, P4 is executed for 3ms.
- Now remaining process is P1 only. It executes for 3ms. After that, Since P2, P3 and P4 are already finished, P1 is executed again for remaining time (1ms).

Gantt Chart

P1 has utilized two 3ms time slices earlier, so we subtract 3 twice from its last start time

- Waiting time for P1 = 19-(3+3) = 13ms
- Waiting time for P2 = 3ms
- Waiting time for P3 = 14-3 = 11ms
- Waiting time for P4 = 16-3 = 13ms
- Total average waiting time = (13+3+11+13)/4 = 10ms

Both P3 and P4 have utilized one time slice each, earlier, so we subtract 3 from them



Round Robin – Summary

- Advantage:
  - It does not cause **starvation**, as all the processes get equal CPU time slices.
- Disadvantages:
  - There is an overhead of context switching.
  - Too small time slices cause overhead and slower execution of processes. So, time slices must be large with respect to the context switch time.

Priority Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- The process priorities are compared (low priority number means higher priority, i.e. priority 0 is the highest priority).
- These steps are repeated.

Priority Scheduling Example

- Consider the following and implement priority scheduling

Process	Arrival Time	Burst Time	Priority
P1	0	11	2
P2	5	28	0
P3	12	2	3
P4	2	10	1
P5	9	16	4

- Gantt chart

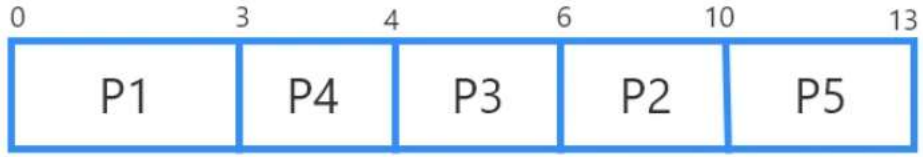
Priority Scheduling – Problem

- Perform priority scheduling for the following processes:

Process	Execution Time	Priority	Arrival Time
P1	3	1 (Highest)	0
P2	4	3	0
P3	2	2	0
P4	1	1 (Highest)	0
P5	3	4 (Lowest)	0

Solution

- We can draw a Gantt chart



Time

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

Explanation

- Both P1 and P4 have the highest Priority(1). Thus we will use FCFS to settle clash and P1 will execute first.
- P4 will execute once P1 is finished executing
- P3 has priority 2 thus it will execute next.
- Then finally P2 and P5

Average Waiting Time for processes are –

Average Waiting Time = Completion Time – Burst Time – Arrival TIme

- Average Waiting Time for P1 = 3-3-0 = 0ms
- Average Waiting Time for P2 = 10-4-0 = 6ms
- Average Waiting Time for P3 = 6-2-0 = 4ms
- Average Waiting Time for P4 = 4-1-0 = 3ms
- Average Waiting Time for P5 = 13-3-0 = 10ms

Average waiting Time for all processes = (0 + 6 + 4 + 3 +10)/5 = 4.6ms

Priority Scheduling – Points

- A process having lower priority can be indefinitely blocked or starved.
- To prevent this, we do **aging**, where the priority of the process is increased as it waits in the queue.
- So, a process which has been in a queue for a long time will reach high priority, and hence it won’t be starved.

Earliest Deadline First (EDF) Scheduling

- **Earliest Deadline First (EDF)** is an optimal dynamic priority scheduling algorithm used in real-time systems.
- EDF uses priorities to the jobs for scheduling.
- It assigns priorities to the task according to the absolute deadline.
- The task whose deadline is closest gets the highest priority.
- The priorities are assigned and changed in a dynamic fashion.

Linux and Process-related Commands

- To find out which processes are running on Linux, we can use the **ps** command

```
atul@LAPTOP-E9C3P1LP:~$ ps
  PID TTY          TIME CMD
   10 pts/0    00:00:00 bash
   48 pts/0    00:00:00 ps
```

- To see a tree of the processes, we can use **pstree**

Process Creation Commands – fork, waitpid, and exec

fork

It is a clone operation.
Here, the current process is taken and cloned.
So, now we have a parent process and a child process with two different process IDs.
Everything (sack, heap, file descriptors etc) are copied from the parent process to the child process. The two are independent processes now.

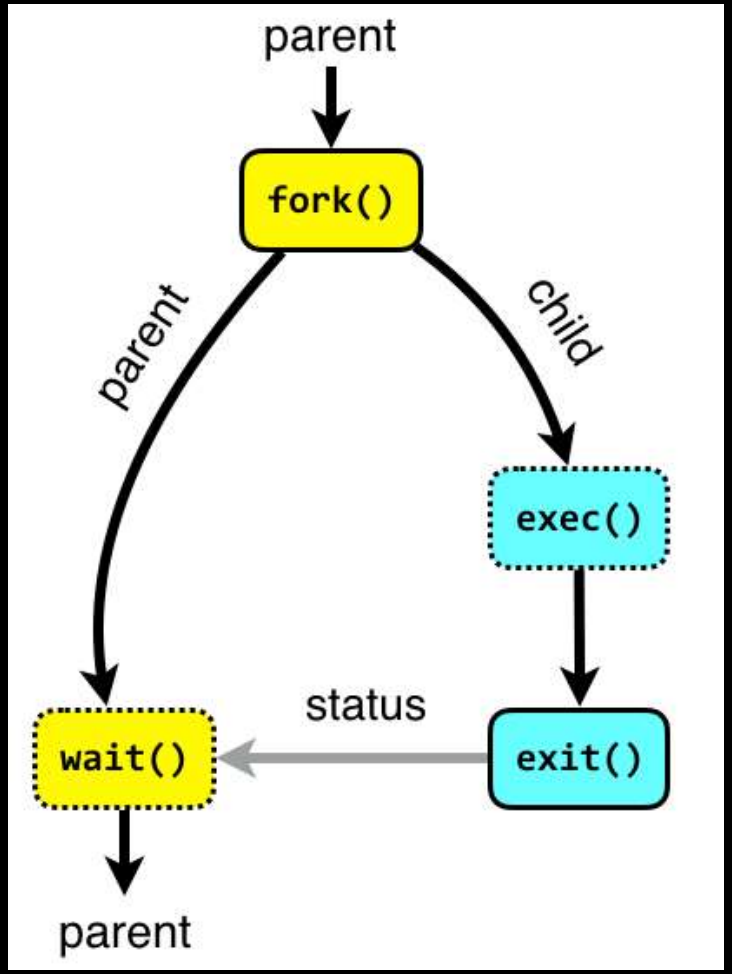
waitpid

Consider the shell. It reads a command from the terminal, forks off a child process, waits for the child process to execute the command, and then reads the next command when the child terminates.
To wait for the child to finish, the parent executes a *waitpid* system call.

exec

The child process must execute the command typed by the user. For this, the *exec* call is used. The entire core image of the process is replaced by the file named in its first parameter (see next slide).

Illustration of fork, waitpid and exec



fork(): Creates a new child process.
fork() returns: 0 in the child process, -1 if an error occurs.
The new child process is identical to the parent but has a unique Process ID (PID).

waitpid(): Makes the parent wait for a specific child process. This is useful for synchronization.

exec(): Replaces the current process with a new program, often used within a child process created by fork().
Commonly used functions in the exec() family include execl, execp, execv, execvp, etc.
If exec() is successful, it doesn't return, because the new program takes over. If it fails, it returns -1.

A Highly Simplified Shell

```
while (TRUE) {                                  /* repeat forever /*/
        type_prompt( );                         /* display prompt on the screen */
        read_command(command, params);          /* read input line from keyboard */

        pid = fork( );                          /* fork off a child process */
        if (pid < 0) {
                printf("Unable to fork0);       /* error condition */
                continue;                       /* repeat the loop */
        }

        if (pid != 0) {
                waitpid (–1, &status, 0);       /* parent waits for child */
        } else {
                execve(command, params, 0);     /* child does the work */
        }
}
```

Testing fork, waitpid and exec

- Install gcc in the Linux machine
- **sudo apt install gcc**

- We will create three files
- fork_test.c
- getpid.c
- exec.c

- To compile and execute (Shown for one case)
- gcc fork_test.c
- ./a.out

fork_test.c

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

/* This program forks and and the prints whether the process is
 *   - the child (the return value of fork() is 0), or
 *   - the parent (the return value of fork() is not zero)
 *
 * When this was run 100 times on the computer the author is
 * on, only twice did the parent process execute before the
 * child process executed.
 *
 * Note, if you juxtapose two strings, the compiler automatically
 * concatenates the two, e.g., "Hello " "world!"
 */

int main( void ) {
                        char *argv[3] = {"Command-line", ".", NULL};

                        int pid = fork();

                        if ( pid == 0 ) {
                                                execvp( "find", argv );
                        }

                        /* Put the parent to sleep for 2 seconds--let the child finished executing */
```

getpid.c

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

/* This program demonstrates that the use of the getpid() function.
 *
 * When this was run 100 times on the computer the author is
 * on, only twice did the parent process execute before the
 * child process executed.
 *
 * Note, if you juxtapose two strings, the compiler automatically
 * concatenates the two, e.g., "Hello " "world!"
 *
 * The return value of fork() is actually pid_t ('pid' 't'ype).
 * When it is assigned to 'int pid', if the type is different, there
 * is an implicit cast; however, when we print the return value
 * of getpid(), it is necessary to explicitly cast it as an
 * integer.
 *
 * The type 'pid_t' is defined in the library header <sys/types.h>
 */

int main( void ) {
                printf( "The process identifier (pid) of the parent process is %d\n", (int)getpid() );

                int pid = fork();
```

exec.c

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

/* This program forks and and the prints whether the process is
 *   - the child (the return value of fork() is 0), or
 *   - the parent (the return value of fork() is not zero)
 *
 * When this was run 100 times on the computer the author is
 * on, only twice did the parent process execute before the
 * child process executed.
 *
 * Note, if you juxtapose two strings, the compiler automatically
 * concatenates the two, e.g., "Hello " "world!"
 */

int main( void ) {
                        char *argv[3] = {"Command-line", ".", NULL};

                        int pid = fork();

                        if ( pid == 0 ) {
                                                execvp( "find", argv );
                        }
```

Sample Execution

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc fork_test.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
This is being printed in the parent process:
 - the process identifier (pid) of the child is 800
This is being printed from the child process
```

Zombie Process

- If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**—the living dead.
- When the parent finally waits for it, the process terminates.
- Let us write a C program for this (zombie.c)
  - The child finishes its execution using exit() system call while the parent sleeps for 50 seconds, hence doesn’t call wait() and the child process’s entry still exists in the process table.

zombie.c

```
// A C program to demonstrate Zombie Process.
// Child becomes Zombie as parent is sleeping
// when child process exits.
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
        // Fork returns process id
        // in parent process
        pid_t child_pid = fork();

        // Parent process
        if (child_pid > 0)
                sleep(50);

        // Child process
        else
                exit(0);

        return 0;
}
```

Orphan Process

- A process whose parent process no more exists i.e. has either finished or terminated without waiting for its child process to terminate is called an **orphan process**.
- Let us write a C program.
  - Here, the parent finishes execution and exits while the child process is still executing and is called an orphan process now.
  - However, the orphan process is soon adopted by init process, once its parent process dies.

Process Type	Definition	Clean-Up Mechanism	Resource Consumption
Zombie Process	A terminated process still in the process table because the parent hasn't read its exit status.	Parent process (using wait())	Minimal (process table entry only)
Orphan Process	A process whose parent has terminated while it's still running.	Adopted and cleaned up by init	Continues running as usual

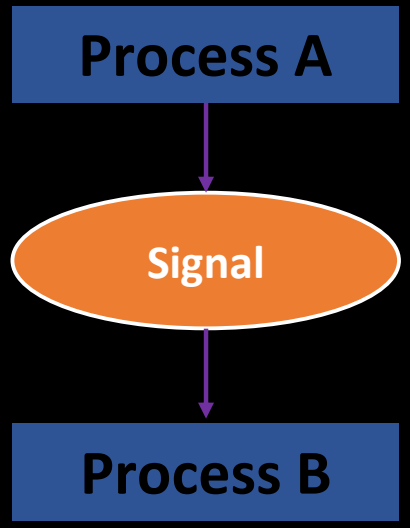
orphan.c

```
// A C program to demonstrate Orphan Process.
// Parent process finishes execution while the
// child process is running. The child process
// becomes orphan.
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main()
{
        // Create a child process
        int pid = fork();

        if (pid > 0) {
                printf("in parent process");
                exit (0);
                }

        // Note that pid is 0 in child process
        // and negative if fork() fails
        else if (pid == 0)
        {
                sleep(30);
                printf("in child process");
        }
```

Session 7: Signals and Threads

Signals

- **Signals** are used by Linux processes to communicate with one another.



Signal	Value	Description
1	SIGHUP	Hang up the process.
2	SIGINT	Interrupt the process.
3	SIGQUIT	Stop the process.
9	SIGKILL	Unconditionally terminate the process.
15	SIGTERM	Terminate the process if possible.
17	SIGSTOP	Unconditionally stop, but don't terminate the process.
18	SIGTSTP	Stop or pause the process, but don't terminate.
19	SIGCONT	Continue a stopped process.

Signal Example

- Simple example of a signal:
- Run the following command on the shell prompt: *sleep 100*
- The terminal will not do anything for 100 seconds now
- But we can press Ctrl+C to break it
- This sends a *SIGINT* signal to the sleeping process and stops it

Signal Handler

- A **signal handler** is special function (defined in the software program code and registered with the kernel) that gets executed when a particular signal arrives.
- This causes the interruption of current executing process and all the current registers are also saved.
- The interrupted process resumes once the signal handler returns.

C Program: signal_example.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

void sig_handler(int sig_num)
{
if(sig_num == SIGINT)
{
printf("\n Caught the SIGINT signal\n");
}
else
{
printf("\n Caught the signal number [%d]\n", sig_num);
}

// Do all the necessary clean-up work here

exit(sig_num);
}

int main(void)
{
//Register signal handler through the signal() function
signal(SIGINT, sig_handler);
```

Running the Program

- You will have to Ctrl+C to end it

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ nano signal_example.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc signal_example.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out

 Caught the SIGINT signal
^Catul@LAPTOP-E9C3P1LP:~/bash_course$
```

Explanation

- We can see that the program was executed and then the signal SIGINT is passed to it by pressing the Ctrl+c key combination. As can be seen from the debug print in the output, the signal handler was executed as soon as the signal SIGINT was delivered to the process.

- Inside the main() function, the signal() function is used to register a handler (sig_handler()) for **SIGINT** signal.
- The while loop simulates an infinite delay. So, the program waits for **SIGNIT** signal infinitely.
- The signal handler 'sig_handler' prints a debug statement by checking if the desired signal is **SIGINT** or not.
- The signal handler is mostly used to do all the clean-up and other related stuff after a signal is delivered to a process.

Absence of Signal Function: signal_example_endless.c

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

int main(void)
{
//Register signal handler through the signal() function
signal(SIGINT, SIG_IGN);

while(1)
{
//simulate a delay -- as if the program is doing some other stuff
sleep(1);
}

return 0;
}
```

There is no signal handler function now as the second argument to the signal function is replaced with **SIG_IGN**

Running the Example

- You will have to press Ctrl+Z now
- It sends SIGSTP signal to the current foreground application. This effectively puts the program in the background. In English, it basically PAUSES the application. So, in the **ps** command, we should see the process still running!

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc signal_example_endless.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
^C^C^C^C^C^C^C^C^C^C^C^C^C^C^Z
[1]+  Stopped                 ./a.out
atul@LAPTOP-E9C3P1LP:~/bash_course$ ps
  PID TTY          TIME CMD
   67 pts/1    00:00:00 bash
  887 pts/1    00:00:00 a.out
  888 pts/1    00:00:00 ps
atul@LAPTOP-E9C3P1LP:~/bash_course$
```

Now Killing the Process by sending a SIGTERM Signal through the kill Command

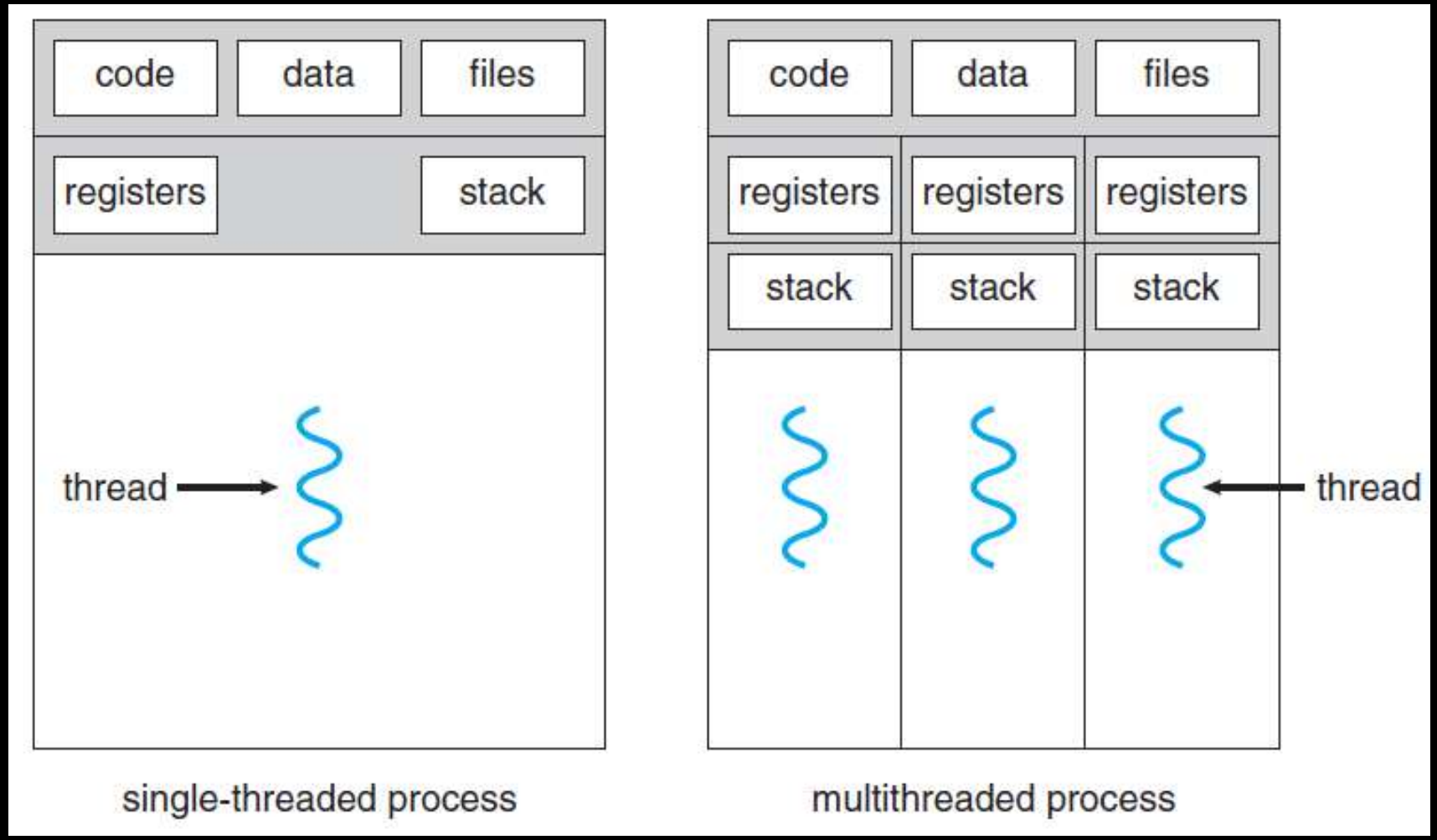
```
atul@LAPTOP-E9C3P1LP:~/bash_course$ kill SIGTERM -887
-bash: kill: SIGTERM: arguments must be process or job IDs

[1]+  Stopped                 ./a.out
atul@LAPTOP-E9C3P1LP:~/bash_course$ ps
  PID TTY          TIME CMD
   67 pts/1    00:00:00 bash
  898 pts/1    00:00:00 ps
[1]+  Terminated              ./a.out
atul@LAPTOP-E9C3P1LP:~/bash_course$
```

Processes and Threads

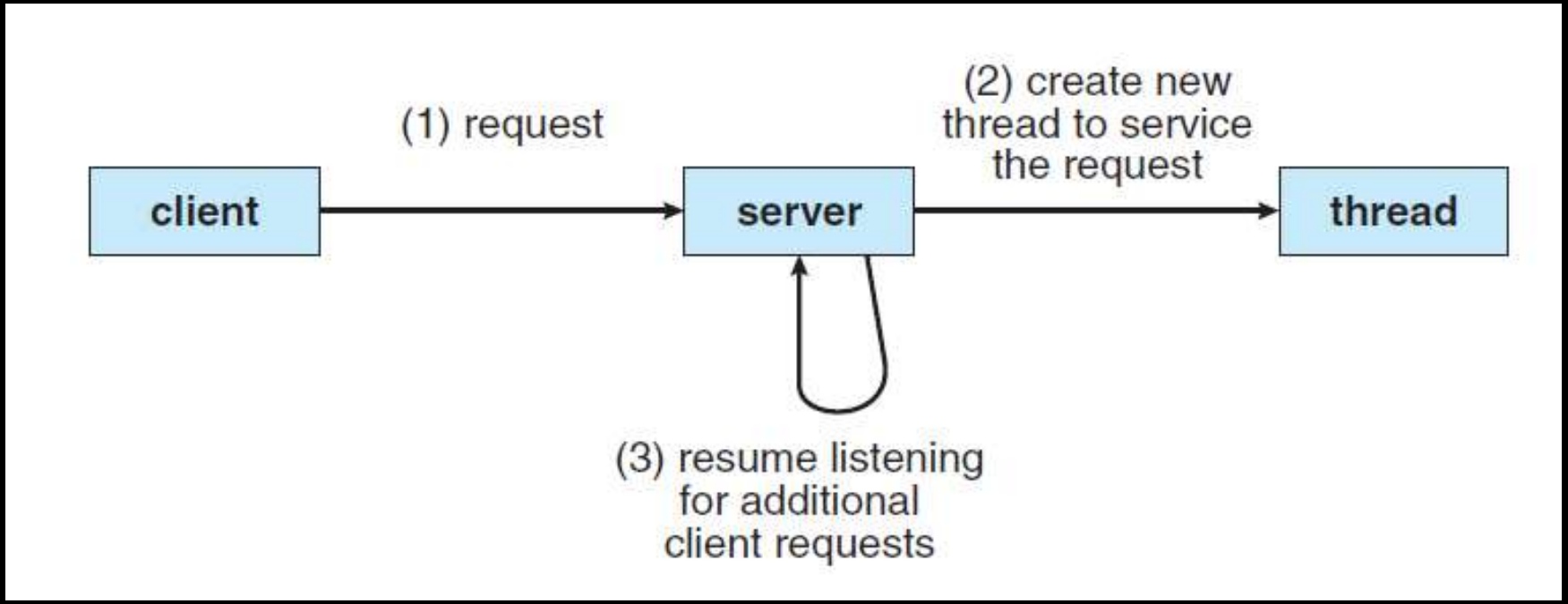
Processes and Threads

- A **thread** is a lightweight process.
- A process can do more than one unit of work concurrently by creating one or more threads. These threads, being lightweight, can be spawned quickly.
- We can identify the process and its thread in Linux using the **ps -eLf** command:
  - PID: Unique process identifier
  - LWP: Unique thread identifier inside a process
  - NLWP: Number of threads for a given process

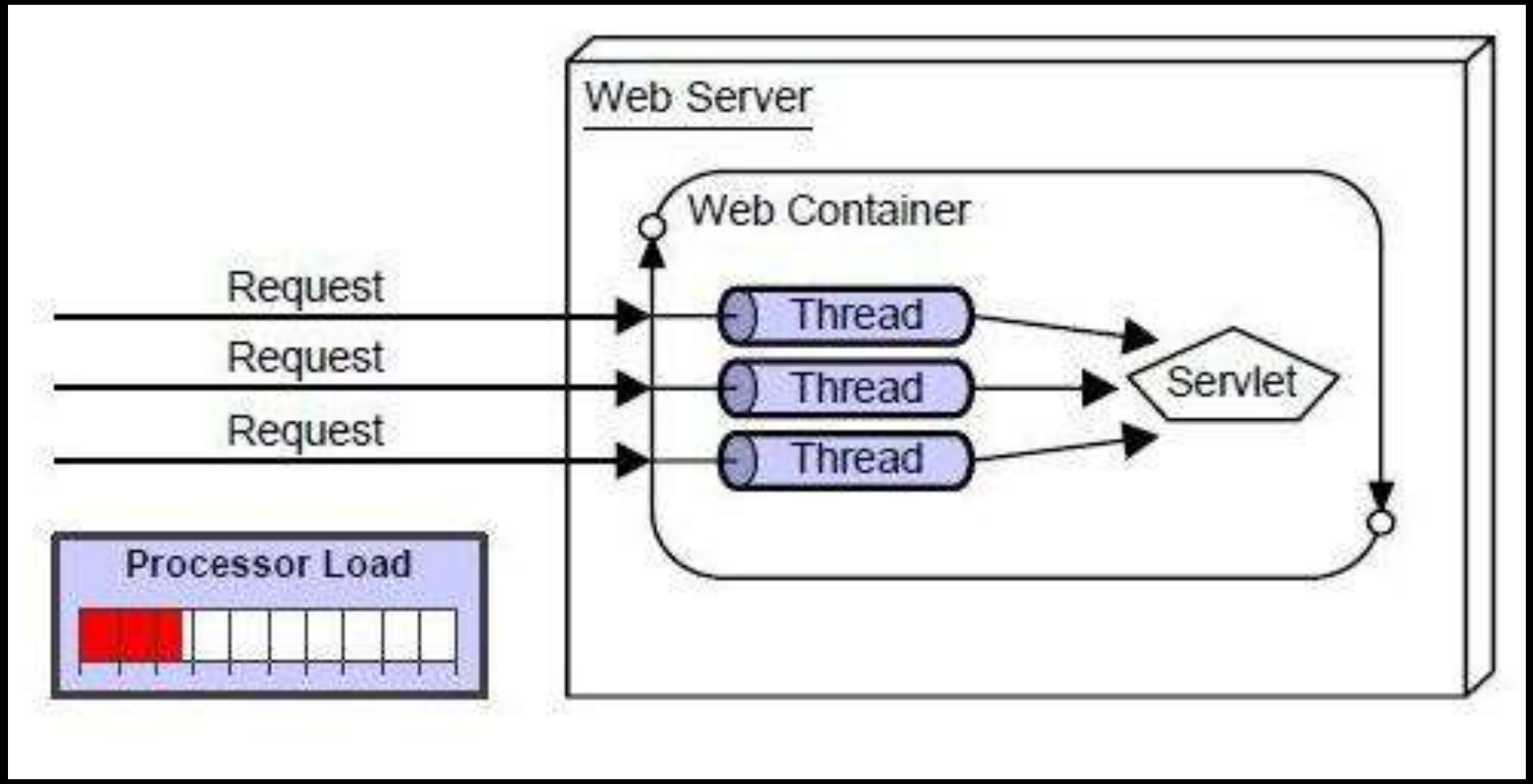
Single-Threaded and Multi-Threaded Processes



Multithreaded Server



Example


Web Server
Web Container
Request
Request
Request
Thread
Thread
Thread
Servlet
Processor Load

The clone Function

- We can use the **clone** function to create a new process as well as a thread
- Of course, we can use the **fork** function to clone a process
- A flag decides what will get created

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Threads in Linux: clone.c

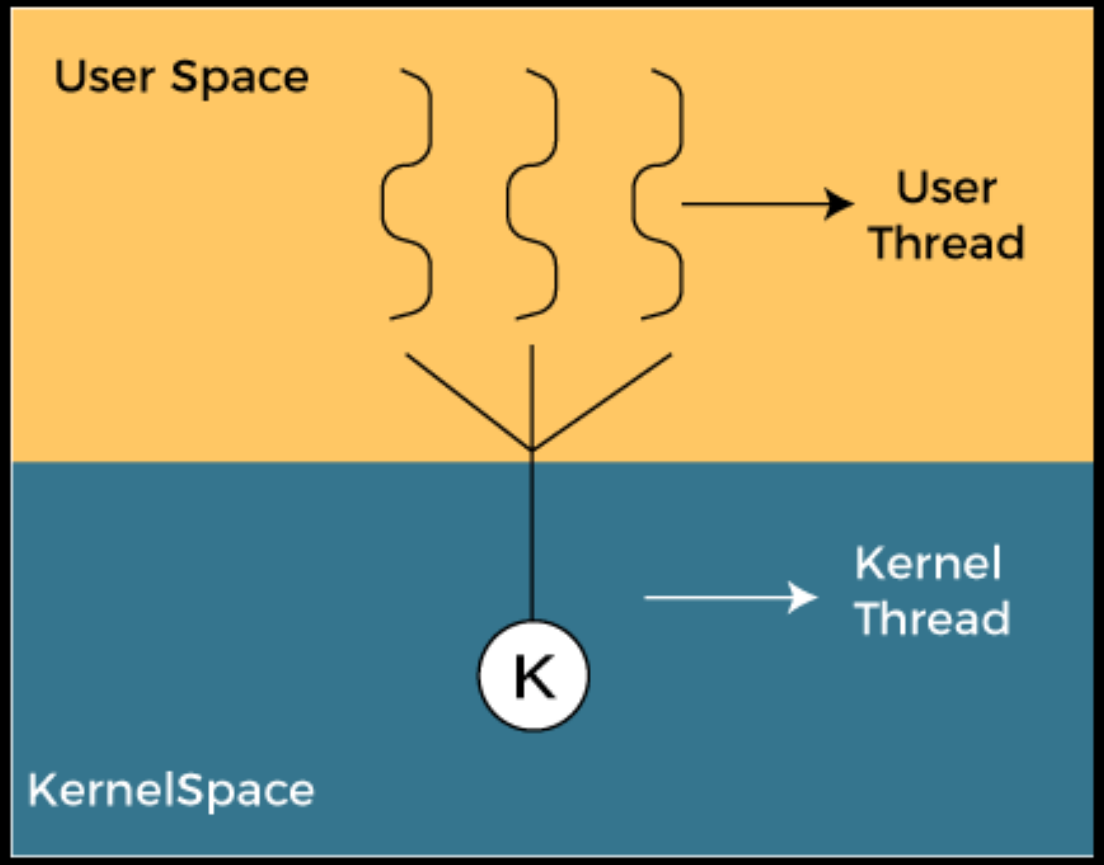
```
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sched.h>
#include <signal.h>

#define STACK 8192

int do_something(){
        printf("Child pid : %d\n", getpid());
        return 0;
}

int main() {
        void *stack = malloc(STACK);    // Stack for new process

        if(!stack) {
                perror("Malloc Failed");
                exit(0);
        }

        if( clone( &do_something, (char *)stack + STACK, CLONE_VM, 0) < 0 ){
                perror("Clone Failed");
```

Running the Example

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
Parent pid : 1003
Child pid : 1004
```

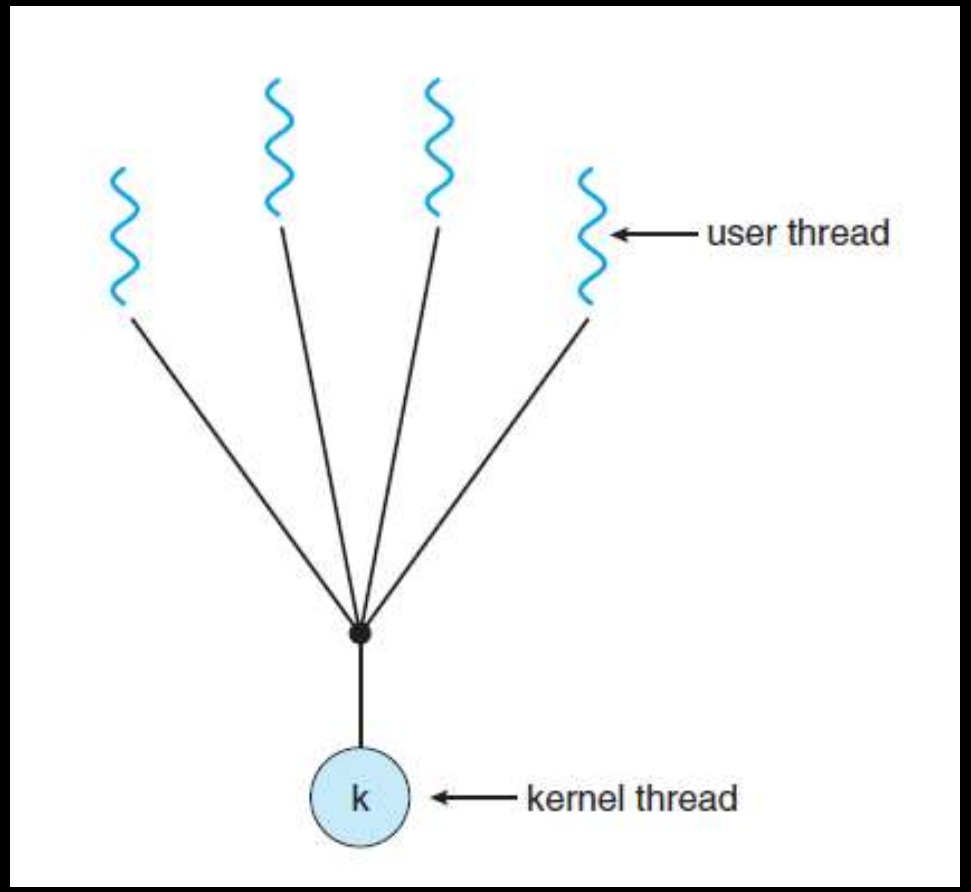
User and Kernel Threads

- **User threads** are supported above the kernel
- **Kernel threads** are supported and managed directly by the operating system
- User threads cannot run on their own, so they need kernel-level threads to execute
- So, for a user-level thread to make progress, the user program has to have its scheduler take a user-level thread and run it on a kernel-level thread
- Mapping user threads to kernel threads:
  - **Many-to-One Model**
  - **One-to-One Model**
  - **Many-to-Many Model**



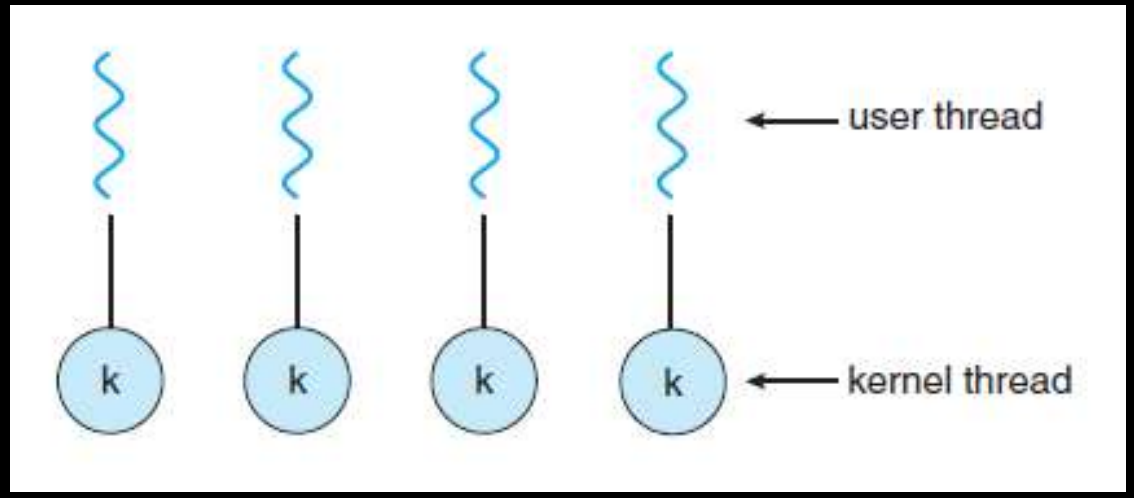
Many-to-One Model

- The **many-to-one model** maps all user threads to only one kernel thread
- The process can run only one user thread at a time, as there is only one kernel thread associated with the process



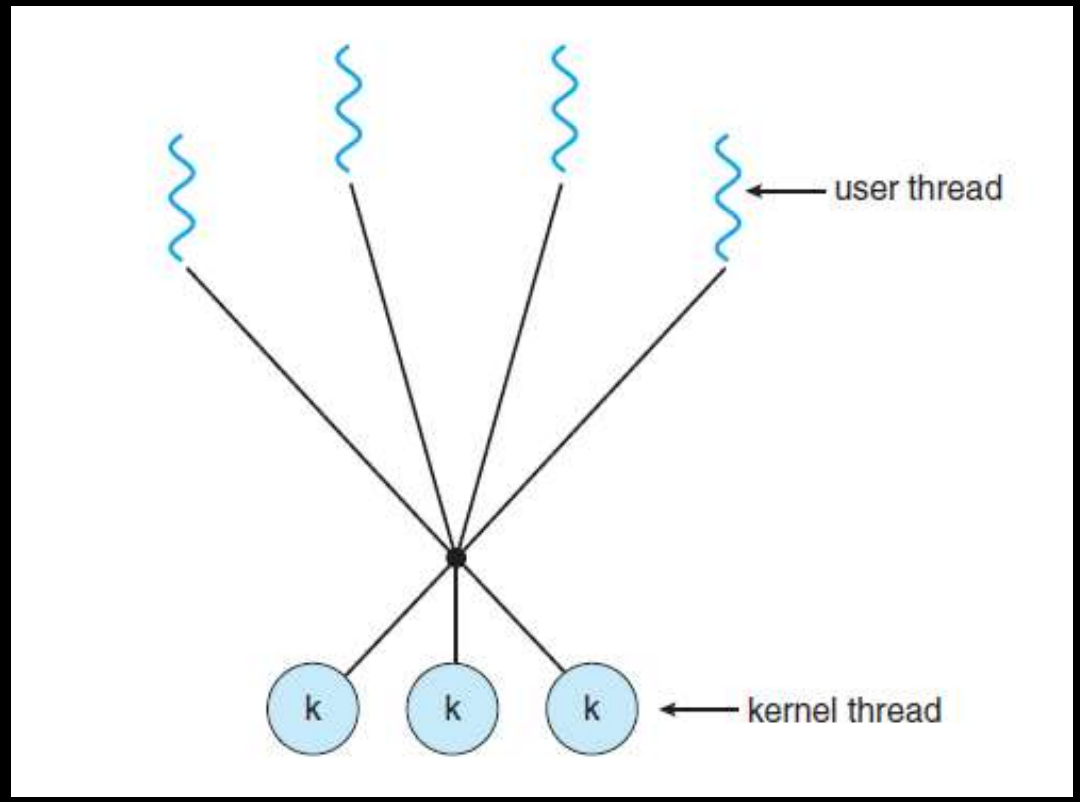
One-to-One Model

- The **one-to-one model** maps each user thread to a kernel thread
- Better concurrency
- Allows multiple threads to run in parallel on multiprocessors
- The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread



Many-to-Many Model

- The **many-to-many model** multiplexes many user-level threads to a smaller or equal number of kernel threads
- A pool of kernel threads is created
- User threads are allocated one or more kerel threads from this pool



Using the Posix Thread (pthread) function – pthread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> //Header file for sleep().
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadCall(void *vargp)
{
        sleep(1);
        printf("Printing CDAC from Thread \n");
        return NULL;
}

int main()
{
        pthread_t thread_id;
        printf("Before Thread\n");
        pthread_create(&thread_id, NULL, myThreadCall, NULL);
        pthread_join(thread_id, NULL);
        printf("After Thread\n");
        exit(0);
}
```

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc pthread.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
Before Thread
Printing CDAC from Thread
After Thread
```

Note: We might have to do **gcc pthread.c –lpthread** if we get linking errors.

Code Explanation

- In main(), we declare a variable called thread_id, which is of type pthread_t, which is an integer used to identify the thread in the system. After declaring thread_id, we call pthread_create() function to create a thread.
- pthread_create() takes 4 arguments.
  - The first argument is a pointer to thread_id which is set by this function.
  - The second argument specifies attributes. If the value is NULL, then default attributes shall be used.
  - The third argument is name of function to be executed for the thread to be created.
  - The fourth argument is used to pass arguments to the function, myThreadFun.
- The pthread_join() function for threads is the equivalent of wait() for processes. A call to pthread_join blocks the calling thread until the thread with identifier equal to the first argument terminates.

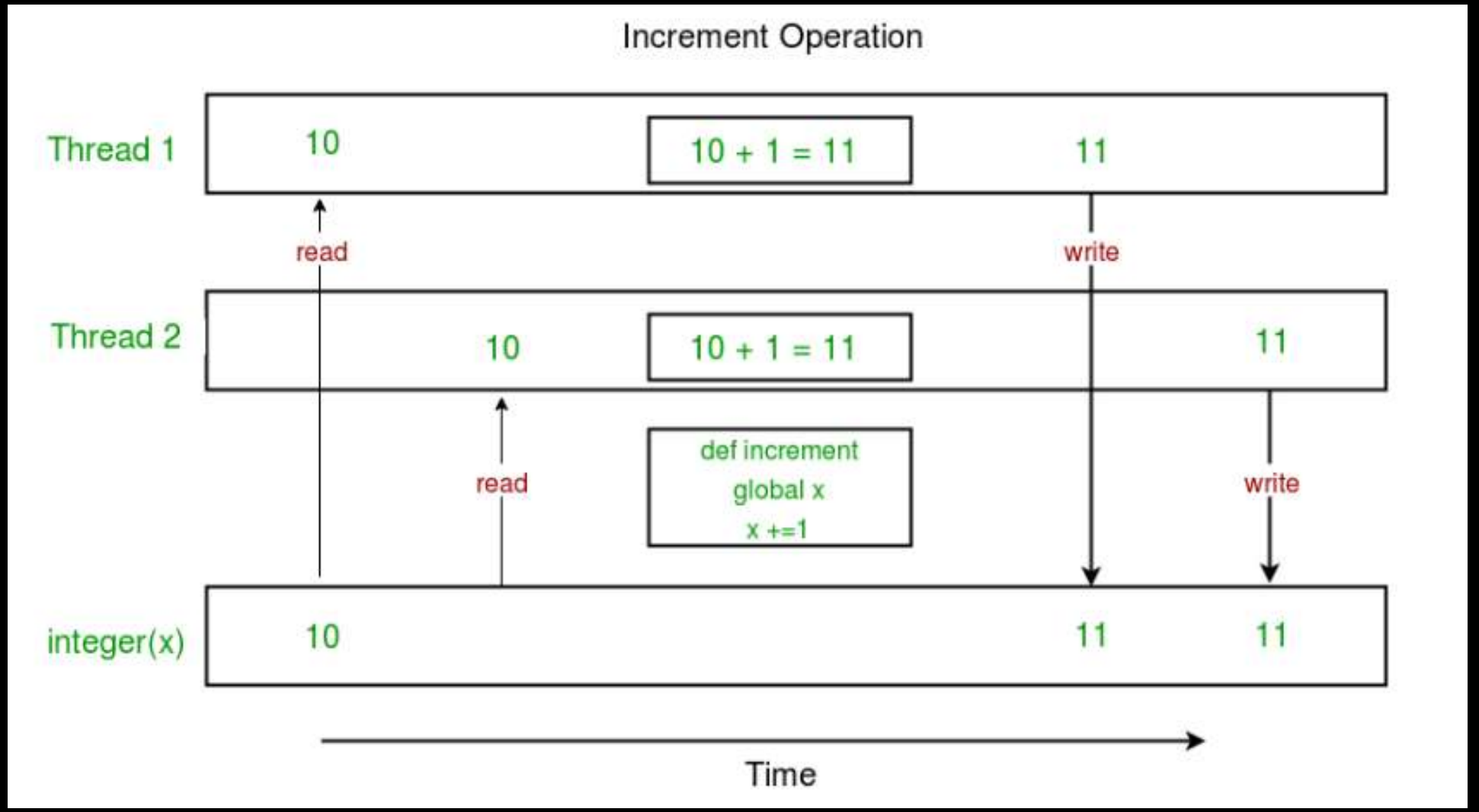
Multithreading Example – multithread.c

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>

// Let us create a global variable to change it in threads
int g = 0;

// The function to be executed by all threads
void *myThreadFCall(void *vargp)
{
        // Store the value argument passed to this thread
        int *myid = (int *)vargp;

        // Let us create a static variable to observe its changes
        static int s = 0;

        // Change static and global variables
        ++s; ++g;

        // Print the argument, static and global variables
        printf("Thread ID: %d, Static: %d, Global: %d\n", *myid, ++s, ++g);
}

int main()
```

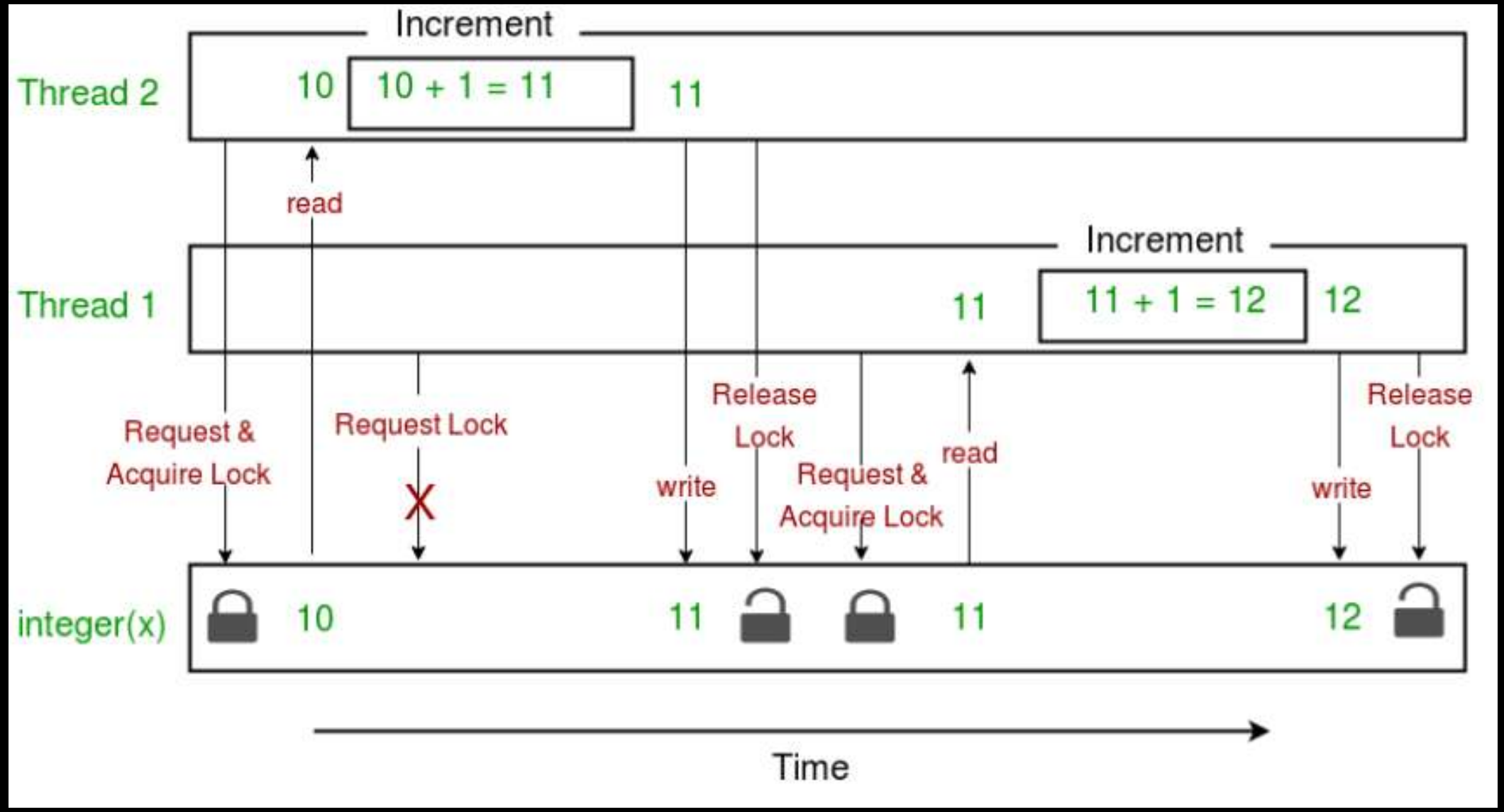
Thread Synchronization

- When we have two or more threads sharing resources, updates to these shared resources by one thread can cause problems for the other thread(s) – See next slide
- The overall processing/output may not remain stable
- Let us understand this with a series of programs
  - thread_1.c: Simple program using a single thread (i.e. no multithreading)
  - thread_2.c: Multi-threading is used, No thread synchronization is done, Output is inconsistent
  - thread_3.c: Multi-threading is used, Thread synchronization is done using mutual exclusion (mutex), Output is consistent

Thread Synchronization Problem – Example



Thread Synchronization Problem – Solution



thread_1.c (Single-threaded)

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

volatile long int a = 0;

int main()
{
   int i;
   a = 0;


   for(i = 0; i < 100000; i++)
   {
      a = a + i;
   }


   printf("%ld\n",a);
   return 0;
}
```

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc thread_1.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
4999950000
```

thread_2.c (Multithreading used, no Synchronization)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>



volatile long int a = 0;
pthread_mutex_t aLock;



void threadFunc()
{
   int i;



   for (i = 1; i < 200000 ; i++)
   {
      a = a + 1;
   }
}
```

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc thread_2.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
1049540723 81
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
1049743555 44
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
8979843688 9
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
6082214478 8
atul@LAPTOP-E9C3P1LP:~/bash_course$
```

thread_3.c (Multithreading used with Synchronization)

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>



volatile long int a = 0;
pthread_mutex_t aLock;



void threadFunc(void* arg)
{
   int i;



   for (i = 1; i < 200000 ; i++)
   {
      pthread_mutex_lock(&aLock);  // Lock a mutex for a
      a = a + 1;
      pthread_mutex_unlock(&aLock); // Unlock a mutex for a
   }
}



void threadFunc2(void *arg)
```

```
atul@LAPTOP-E9C3P1LP:~/bash_course$ gcc thread_3.c
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
105000549999
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
105000549999
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
105000549999
atul@LAPTOP-E9C3P1LP:~/bash_course$ ./a.out
105000549999
atul@LAPTOP-E9C3P1LP:~/bash_course$
```

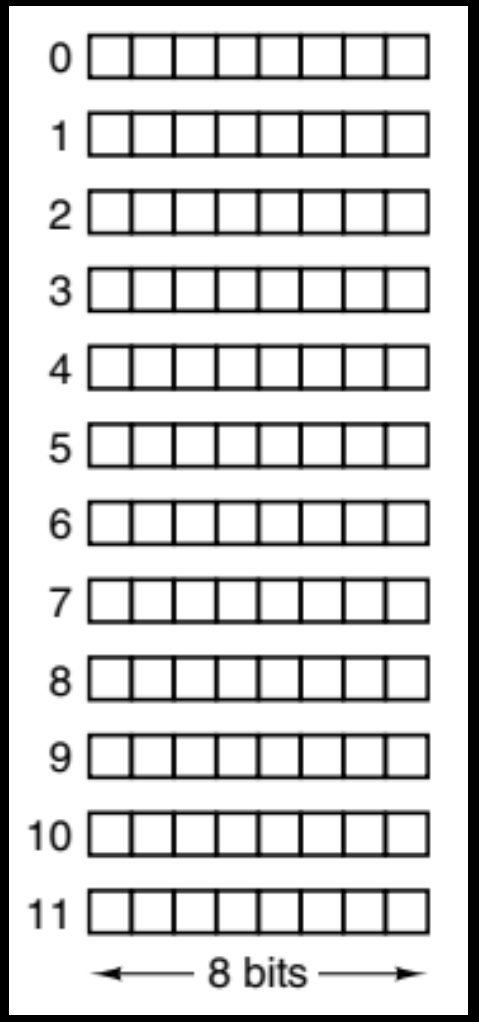
Sessions 8, 9 and 10: Memory Management and Virtual Memory

Main Memory: Basic Concepts

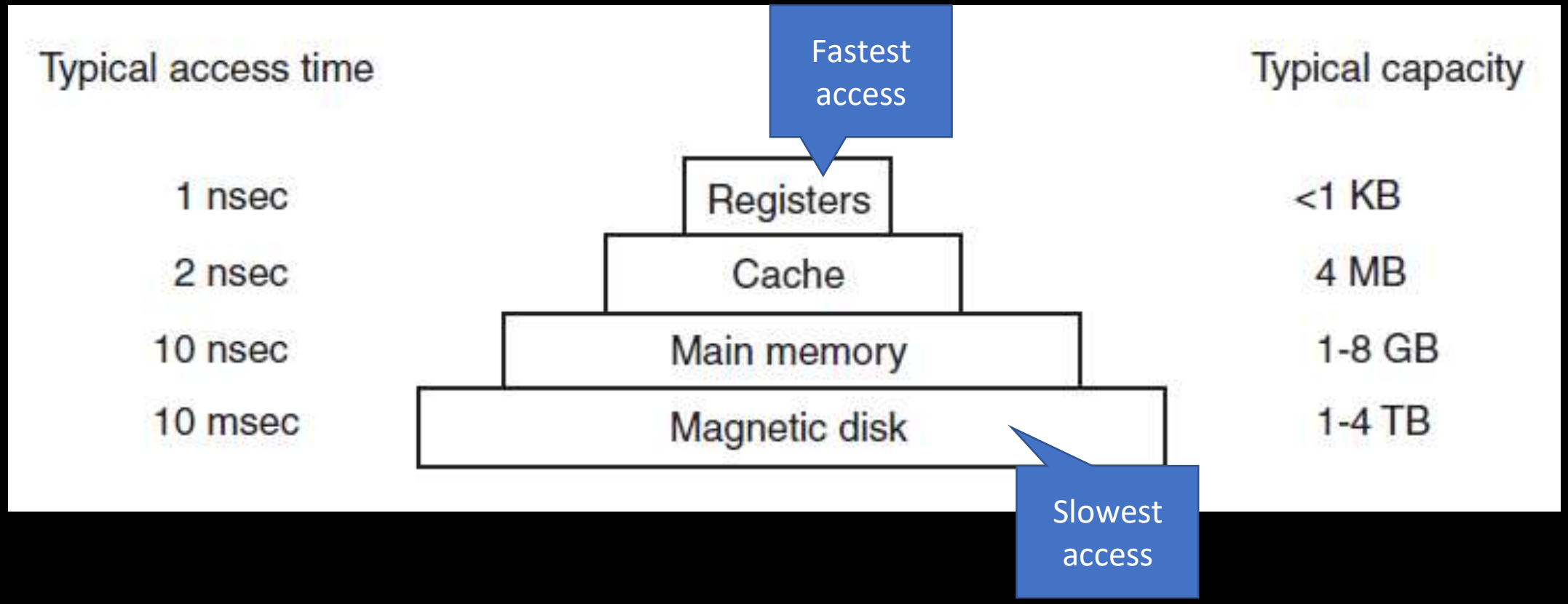
Memory

- The memory is that part of the computer where programs and data are stored.
- Memories consist of a number of **cells** (or locations), each of which can store a piece of information.
- Each cell has a number, called its **address**, by which programs can refer to it.
- If a memory has n cells, they will have addresses 0 to n − 1.
- All cells in a memory contain the same number of bits.

Memory Example



Memory Hierarchy



Technologies used in Memory Design

- Two types of memories
  - **Random Access Memory (RAM)**: Memory from where data can be read and where data can be written at any random location
    - **Static RAM (SRAM)**
    - **Dynamic RAM (DRAM)**
  - **Read Only Memory (ROM)**: Memory from where data can be read, but not where data can be written
    - Useful for booting up, fixed data

Static RAM (SRAM)

- **Static RAMs (SRAMs)** are constructed internally using circuits similar to a D flip-flop
- These memories retain their contents as long as the power is kept on: seconds, minutes, hours, even days
- They are very fast
- A typical access time is a nanosecond or less
- For this reason, static RAMS are popular as **cache memory**

Dynamic RAM (DRAM)

- Dynamic RAMs (DRAMs) do not use flip-flops
- Instead, a dynamic RAM is an array of cells, each cell containing one transistor and a tiny capacitor
- The capacitors can be charged or discharged, allowing 0s and 1s to be stored
- Because the electric charge tends to leak out, each bit in a dynamic RAM must be refreshed (reloaded) every few milliseconds to prevent the data from leaking away
- Since dynamic RAMs need only one transistor and one capacitor per bit (vs six transistors per bit for the best static RAM), dynamic RAMs have a very high density (many bits per chip)
- For this reason, main memories are nearly always built out of dynamic RAMs
- However, this large capacity has a price: dynamic RAMs are slow (tens of nanoseconds)

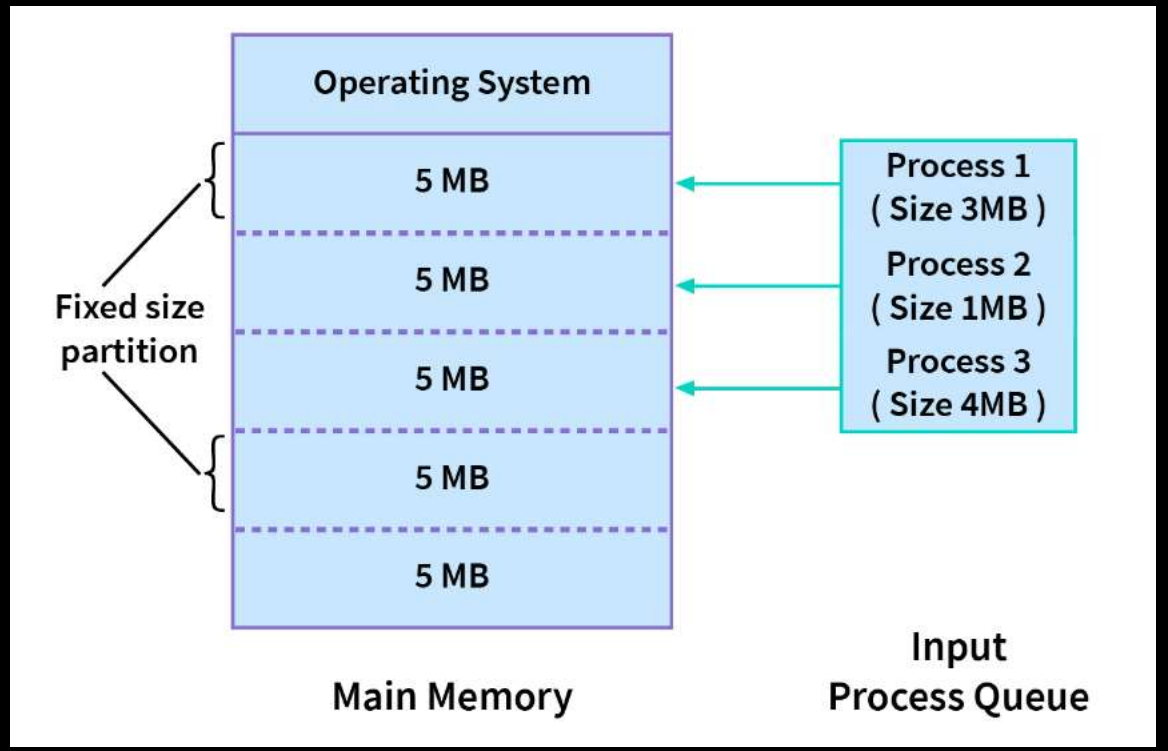
Read Only Memory (ROM)

- The data in a ROM are inserted during its manufacture, essentially by exposing a photosensitive material through a mask containing the desired bit pattern and then etching away the exposed (or unexposed) surface
- The only way to change the program in a ROM is to replace the entire chip
- ROMs are much cheaper than RAMs when ordered in large enough volumes to defray the cost of making the mask
- However, they are inflexible, because they cannot be changed after manufacture

Contiguous and Dynamic Memory Allocation

Contiguous Memory Allocation

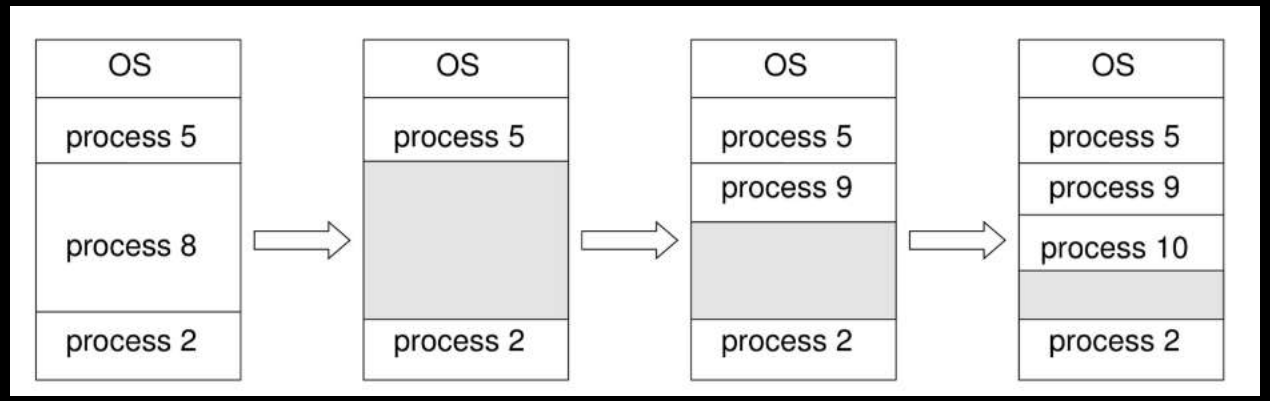
- The memory is usually divided into two partitions: one for the operating system and one for the user processes.
- In **contiguous memory allocation**, each process is allocated a single section of memory that is contiguous to the section containing the next process.
- Contiguous memory allocation can be of **fixed-size partitions** (as shown here) or **variable-size partitions** (different sized partitions)



Dynamic Memory Allocation

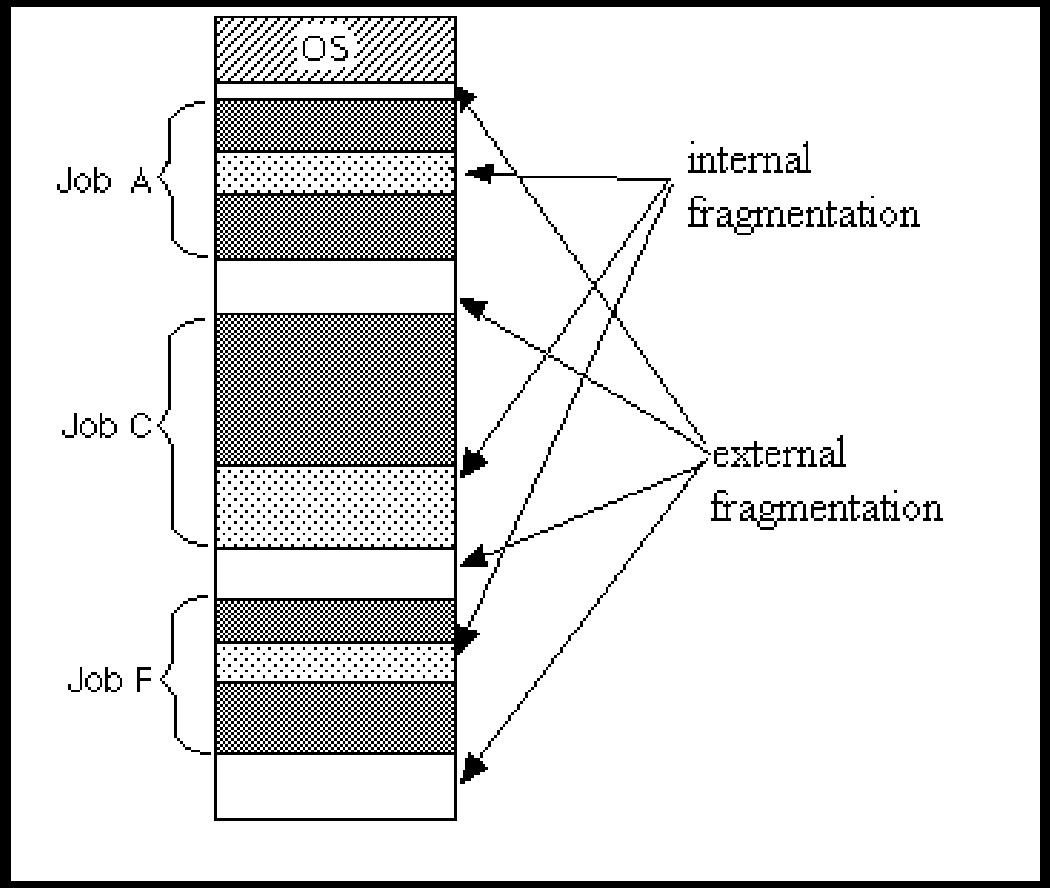
- In **dynamic memory allocation**, memory is not allocated contiguously to the incoming processes.
- Here, one of the three strategies is used:
  - **First fit**. Allocate the first free block that is big enough.
  - **Best fit**. Allocate the smallest free block that is large enough.
  - **Worst fit**. Allocate the largest free block.

- Process 8 goes out, 9 and 10 come in



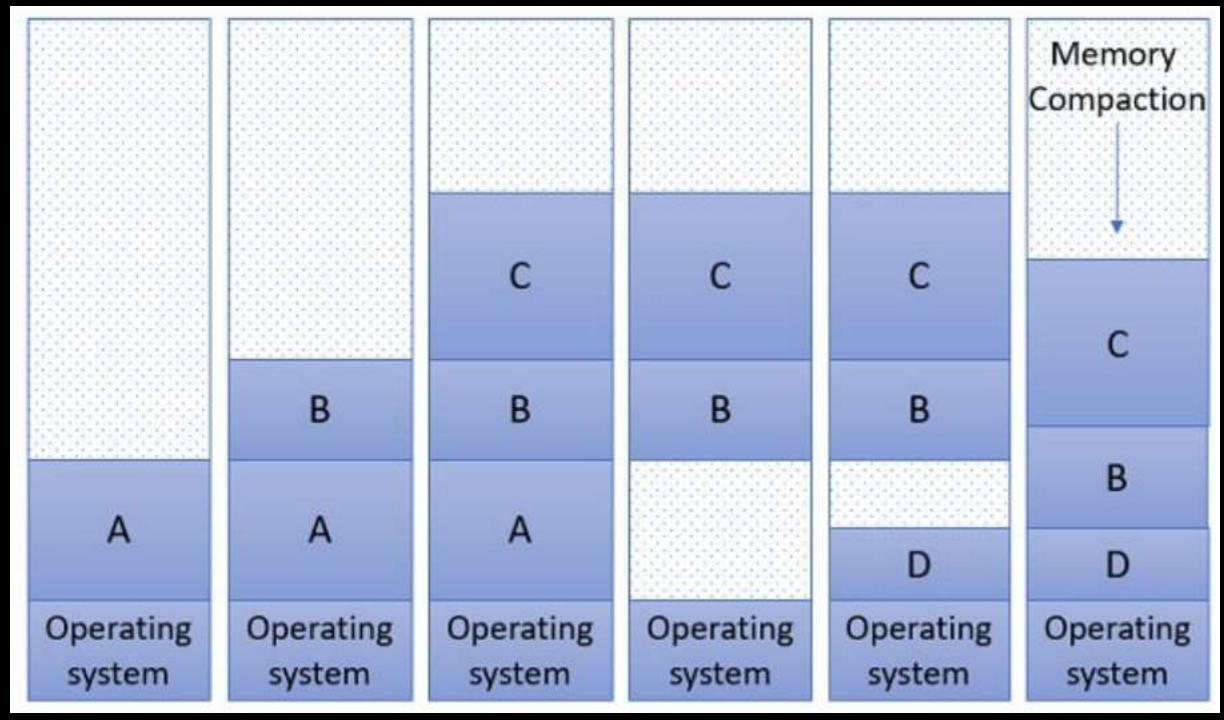
Fragmentation

- Both the first-fit and best-fit strategies for memory allocation suffer from **external fragmentation** – memory blocks available in the memory are not too small and non-contiguous
- **Internal fragmentation—** Unused memory that is internal to a partition, because a process does not fully utilize the allocated memory.



Compaction

- One solution to the problem of external fragmentation is **compaction**.
- The goal is to shuffle the memory contents so as to place all free memory together in one large block.
- Compaction is not always possible, however. If relocation is static and is done at assembly or load time.



Segmentation

Segmentation

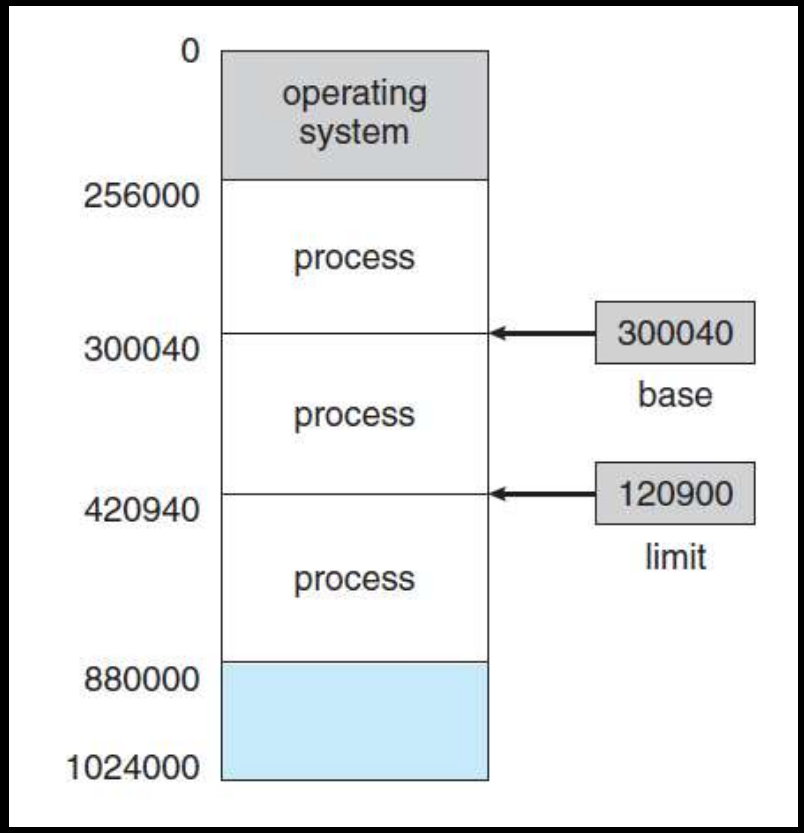
- In **segmentation**, we consider the main memory to be made up of a collection of **segments**
- Each segment can be of a different size
- Logical address of a segment has two parts: segment number and offset

Memory and Protection

- A separate block of memory is allocated to each process.
- This protects the processes from each other for concurrent execution.
- To separate memory spaces, we need the ability to determine the range of legal addresses that the process may access and to ensure that the process can access only these legal addresses.
- For this, two registers are used:
  - **Base register**: Holds the smallest legal physical address.
  - **Limit register**: Specifies the range.
  - Example: See next slide.

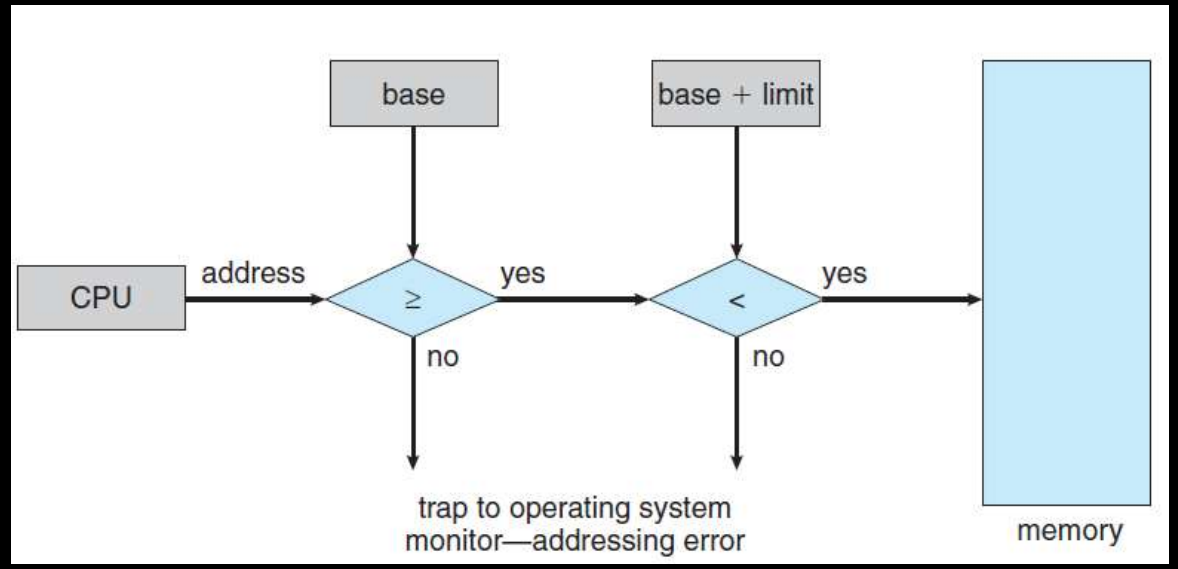
Memory and Protection: Example

- If the base register contains 300040 and limit register contains 120900, then the program can legally access all addresses from 300040 to 420939.



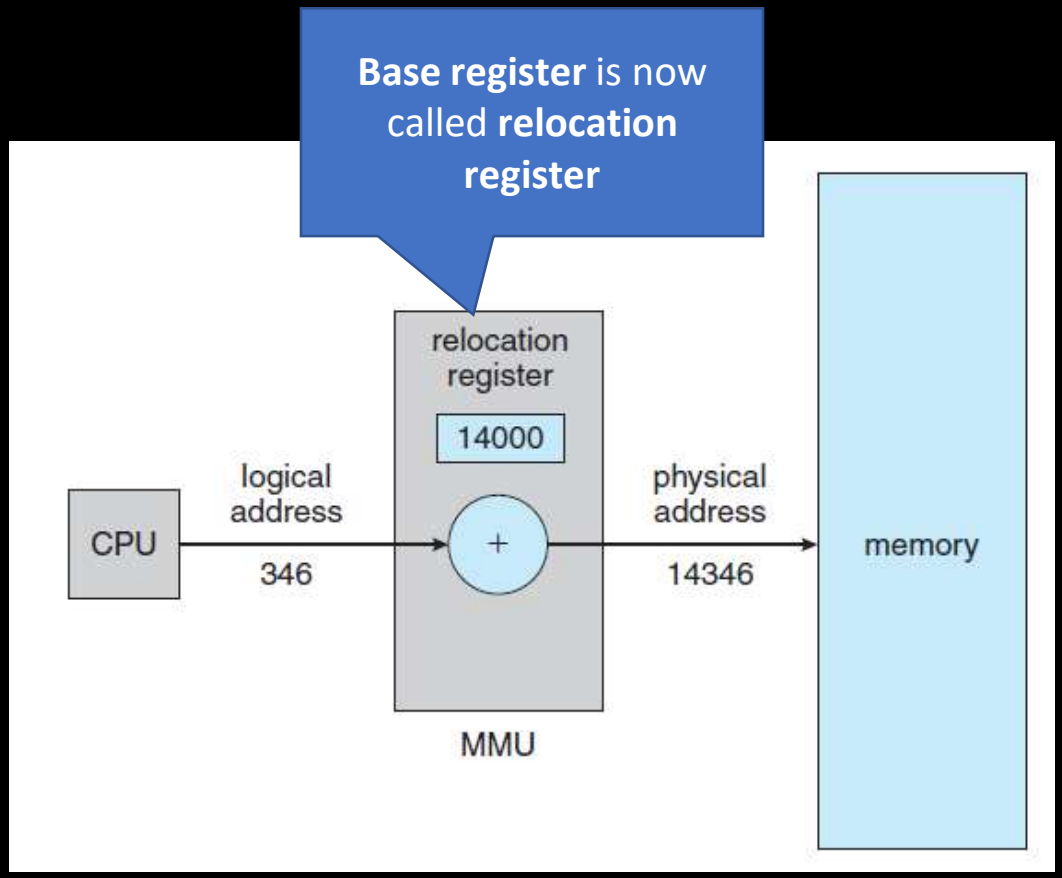
Trap Concept

- Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a **trap** to the operating system, which treats the attempt as a fatal error.
- This scheme prevents a user program from (accidentally or deliberately) modifying the code or data structures of either the operating system or other users.



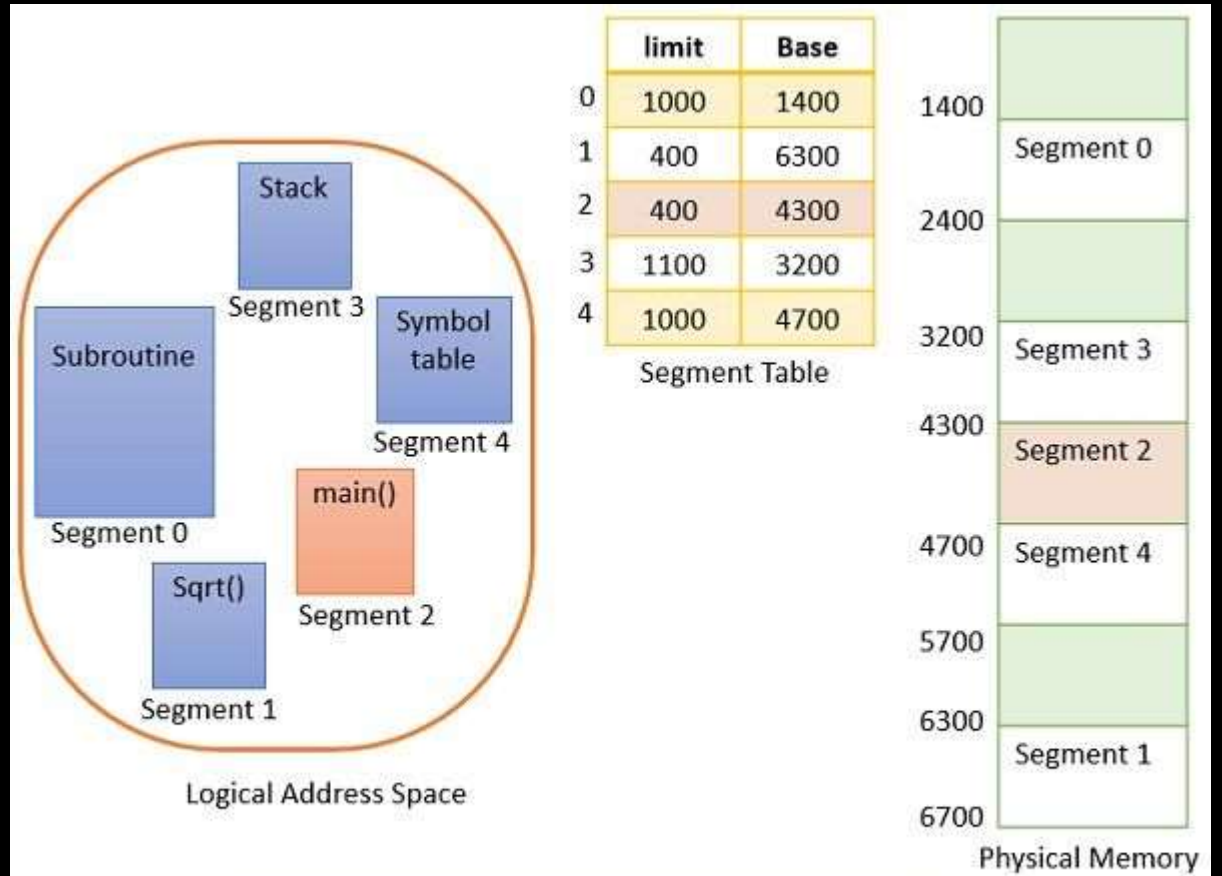
Logical and Physical Address Space

- An address generated by the CPU is commonly referred to as a **logical address** (or **virtual address**), whereas an address seen by the memory unit is the **physical address**.
- Virtual-to-Physical address mapping is done.
- Actual Physical memory address = Relocation register value + Logical address generated by the user process



Segmentation Example

- Consider a user program divided into five segments, numbered 0 to 4



Segmentation Example

- Suppose the CPU calls for segment number 2, whose length is 400 bytes, starting at physical memory address 4300
- The CPU wants to read the 53rd byte in this segment
- So, the CPU informs the operating system that I want data at segment number 2, offset 53
- As we can see, the offset 53 is between 0 and the limit of the segment, which is 400
- So, the operating system will read this byte as: Base register + Offset for the given segment, which is: 4300 + 53 = Byte at physical memory location 4353

Virtual Memory and Paging

Virtual memory – The need

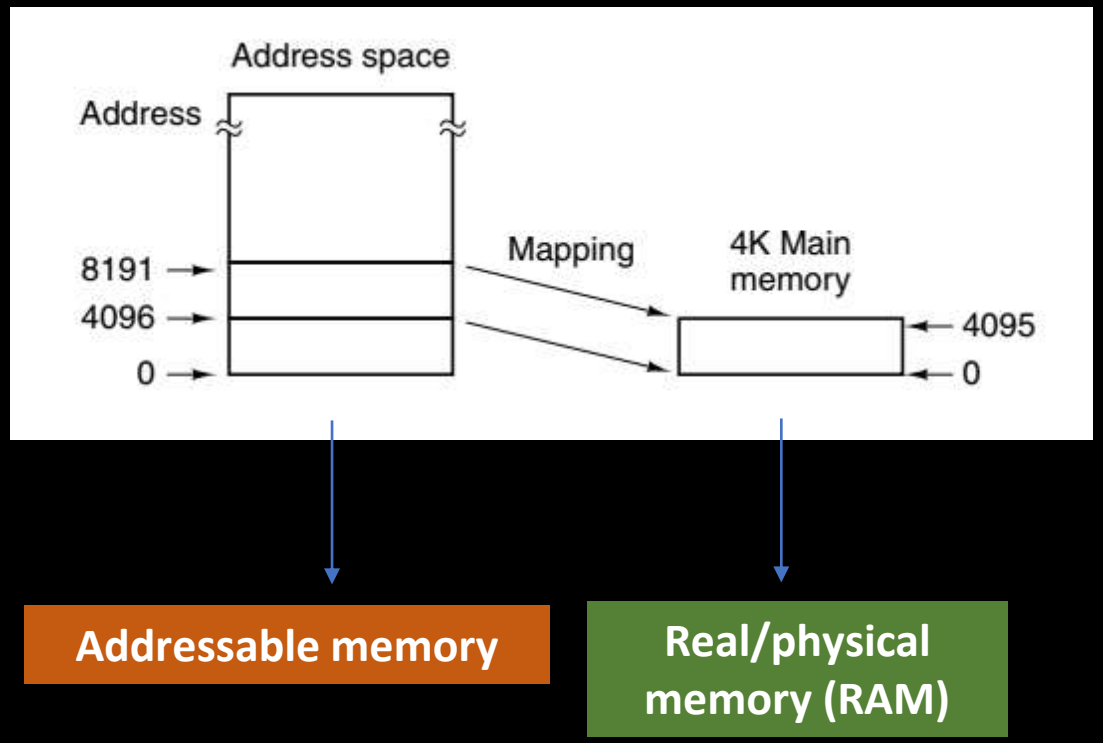
- Consider a computer that has 16-bit memory addresses, but actual physical memory of 4096 words
- Real-life example: Suppose we have kept a capacity of 20 digits in the phone number, but actually phone numbers are only of 10 digits
- Let us take a similar example for computer memory

Virtual memory – The need

- Suppose we have a 16-bit computer
- This means that the computer has 16 wires between the CPU and the main memory of the computer
- So, the actual *addressable* addresses (2^{16}) are 65,536
- But suppose the main memory only has 4096 memory locations
- Result: Memory addresses from 4096 to 65,535 are useless for us
- See next slide

Main Memory and Addressable Memory

Separating address space and memory locations – Virtual Memory!



- At any instant of time, 4096 words of memory can be directly accessed, but they need not correspond to memory locations 0 to 4095
- We could, *tell* the computer that
  - Henceforth whenever address 4096 is referenced, the memory word at address 0 is to be used
  - Whenever address 4097 is referenced, the memory word at address 1 is to be used
  - Whenever address 8191 is referenced, the memory word at address 4095 is to be used, and so forth
- In other words, we have defined a mapping from the address space onto the actual memory locations

Virtual Memory – Benefits

- Now a programmer can write a program, which assumes that there are 65,536 locations in the main memory, and not just 4096
- In other words, the programmer can utilize the full 16-bit address space, when in reality there are only 4096 real memory locations
- Obvious question
  - What happens if a program branches to an address between 8192 and 12287?

Understanding virtual memory

1. The contents of main memory would be saved on disk
2. Words 8192 to 12287 would be found on the disk
3. Words 8192 to 12287 would be loaded into main memory.
4. The address map would be changed to map addresses 8192 to 12287 onto memory locations 0 to 4095
5. Execution would continue as though nothing unusual had happened

- This technique for automatic overlaying is called **paging** and the chunks of program read in from disk are called **pages**.

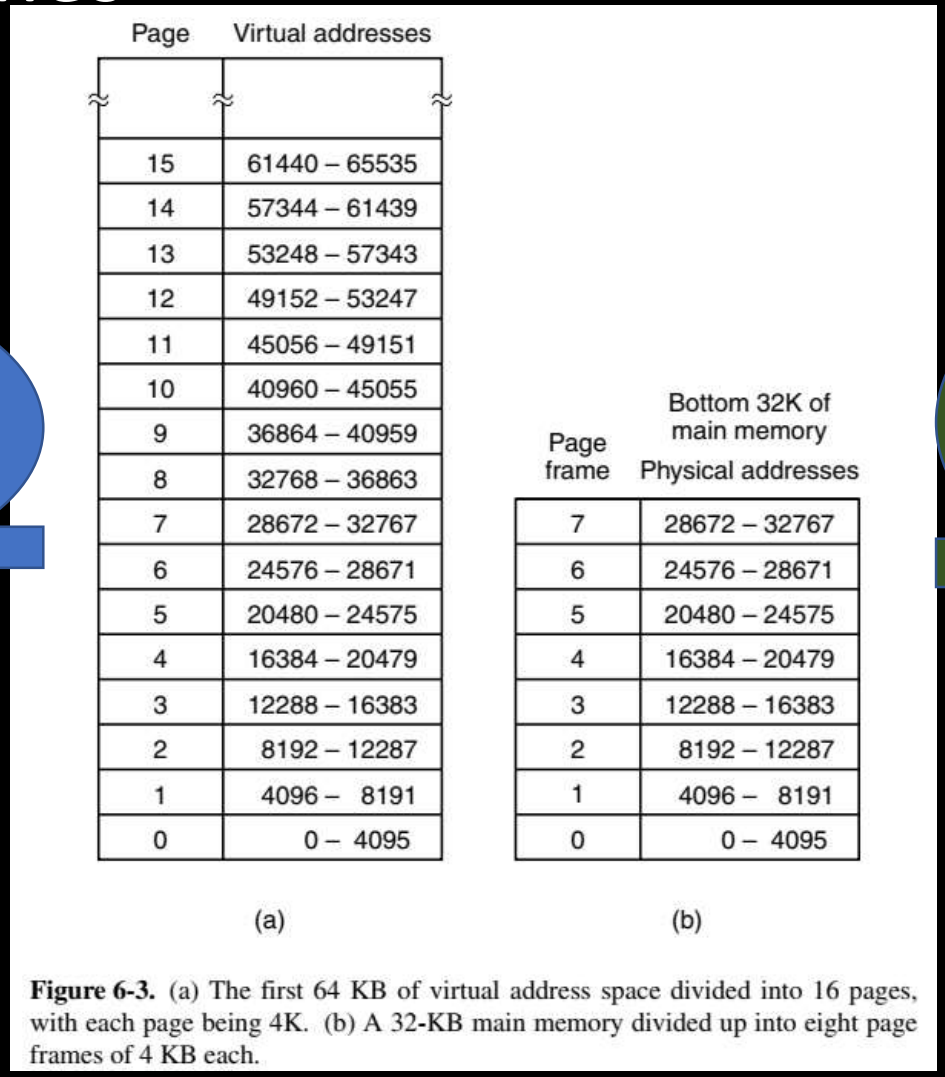
More terminology

- The addresses that the program can refer to are in the **virtual address space**, and the actual, hardwired (physical) memory locations are in the **physical address space**
- A **memory map** or **page table** specifies for each virtual address what the corresponding physical address is
- For this, both the virtual memory and physical memory is divided into equal-sized pages

Divide Virtual Memory into Pages and Physical Memory into Page Frames

Virtual memory

Physical memory



Page	Virtual addresses
15	61440 – 65535
14	57344 – 61439
13	53248 – 57343
12	49152 – 53247
11	45056 – 49151
10	40960 – 45055
9	36864 – 40959
8	32768 – 36863
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(a)

Page frame	Bottom 32K of main memory Physical addresses
7	28672 – 32767
6	24576 – 28671
5	20480 – 24575
4	16384 – 20479
3	12288 – 16383
2	8192 – 12287
1	4096 – 8191
0	0 – 4095

(b)

Figure 6-3. (a) The first 64 KB of virtual address space divided into 16 pages, with each page being 4K. (b) A 32-KB main memory divided up into eight page frames of 4 KB each.

Page Replacement – Basic idea

- When a program refers to a virtual memory address that is not present in the physical/real main memory, a **page fault** occurs
- Ideally, the set of pages that a program is actively and heavily using, called the **working set**, can be kept in memory to reduce page faults
- When a page fault happens, the operating system must read in the required page from the disk into the main memory
- To accommodate it, the operating system must also remove one of the pages from the main memory
- Thus an algorithm that decides which page to remove is needed – this is called as **page replacement policy**

Page Replacement algorithms

- **Least Recently Used (LRU)** algorithm: Evicts the page least recently used because of the probability of its not being in the current working set is high
- **First In First Out (FIFO)**: Removes the least recently loaded page, independent of when this page was last referenced. Associated with each page frame is a counter. Initially, all the counters are set to 0. After each page fault has been handled, the counter for each page presently in memory is increased by one, and the counter for the page just brought in is set to 0. When it becomes necessary to choose a page to remove, the page whose counter is highest is chosen.

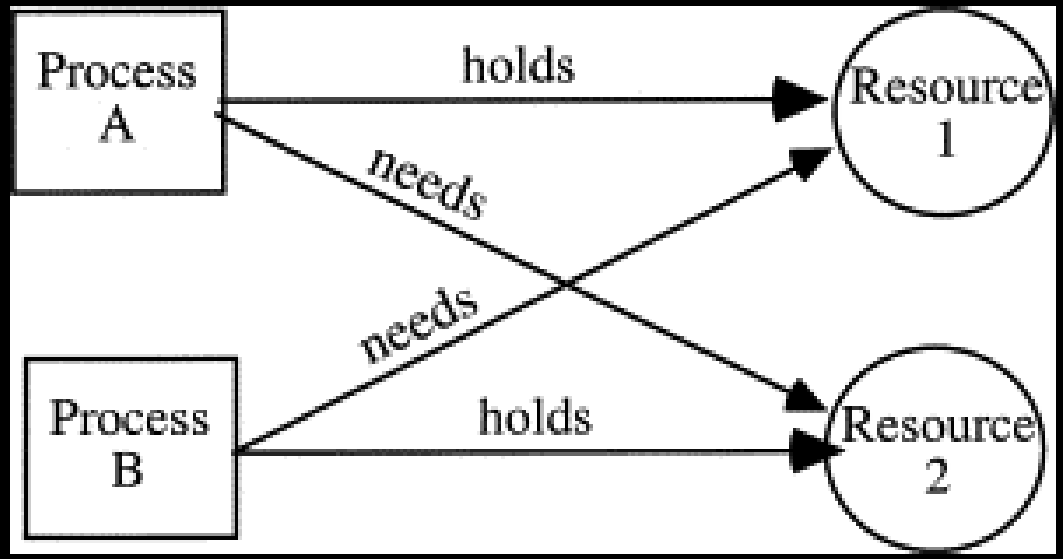
Thrashing concept

- A program that generates page faults frequently and continuously is said to be **thrashing**
- Needless to say, thrashing is an undesirable characteristic to have in your system

Session 11: Deadlocks

Deadlock

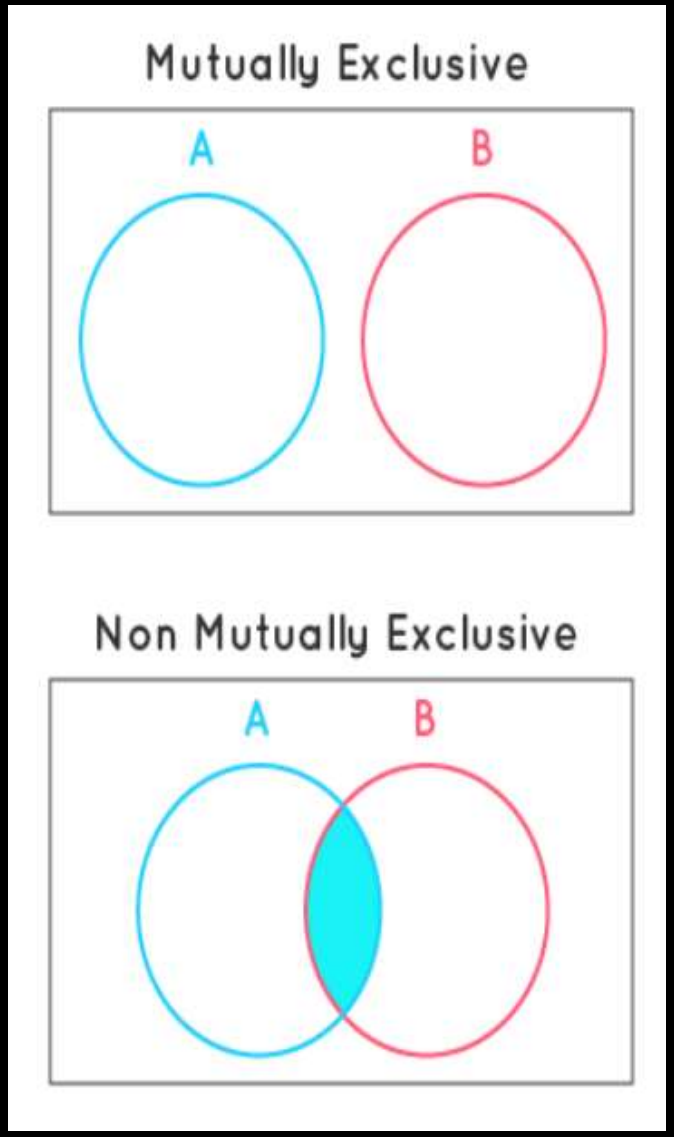
- In a multiprogramming environment, several processes may compete for a finite number of resources.
- A process requests resources; if the resources are not available at that time, the process enters a waiting state.
- Sometimes, a waiting process is never again able to change state, because the resources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



Necessary Conditions for a Deadlock

- The following four conditions must be true at the same time:
  1. **Mutual exclusion**. Only one process can use a resource.
  2. **Hold and wait**. A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  3. **No preemption**. Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it.
  4. **Circular wait**. A set {P0, P1, ..., Pn} of waiting processes must exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn−1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

“Mutually Exclusive”

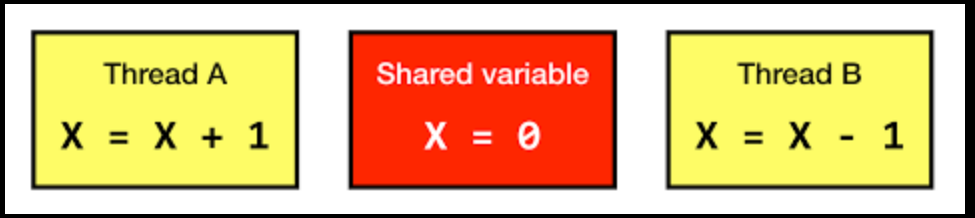


Deadlock Prevention and Avoidance

- To ensure that deadlocks never occur, the system can use either a **deadlock prevention** or a **deadlock avoidance** scheme.
- **Deadlock prevention:** Provides a set of methods to ensure that at least one of the necessary conditions listed earlier is false. These methods prevent deadlocks by constraining how requests for resources can be made.
- **Deadlock avoidance:** Requires that the operating system be given additional information in advance concerning which resources a process will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Deadlock Prevention

- To prevent a deadlock, we need to examine the four necessary conditions for it and see what can be done



- **Condition 1: Mutual exclusion**
  - For a deadlock to occur, at least one resource must be non-sharable (i.e. mutual exclusion condition must become true).
  - Consider a read-only file – it is sharable – processes do not require an exclusive lock on it.
  - However, how can we practically mandate that all the resources be sharable?
  - Hence, avoiding mutual exclusion is very difficult.

Deadlock Prevention

- **Condition 2: Hold and wait**
  - To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
  - Two approaches can be used to handle this:
    - Each process must request and be allocated all its resources before it begins execution.
    - Allow a process to request resources only when it has none.
  - Both these protocols have two main disadvantages.
    - Resource utilization may be low, since resources may be allocated but unused for a long period.
    - **Starvation** is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

Deadlock Prevention

- **Condition 3: No pre-emption**
  - If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources the process is currently holding are preempted.
  - In other words, these resources are implicitly released.
  - The preempted resources are added to the list of resources for which the process is waiting.
  - The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
  - This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space.

Deadlock Prevention

- **Condition 4: Circular wait**
  - Impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
  - Suppose our numbering for resources is *Tape drive = 1, Disk drive = 5, Printer = 12.*
  - Now a process that needs the tape drive and the printer at the same time must first request the tape drive and then the printer.

Deadlock Avoidance

- Here, we require additional information about how resources are to be requested.
- For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process Q will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.

Banker's Algorithm for Deadlock Avoidance

- Suppose we have $24 with us and three friends A, B, and C desperately need money
  - A needs $8, B needs $13, C needs $10
  - We lend $6 to A, $8 to B, and $7 to C
  - So, now we have 24 – (6 + 8 + 7) = $3 left with us
  - Now A needs $2, B needs $5, C needs $3
  - Till they get this money they cannot complete their tasks and also will not return the money already taken from us
  - We have a choice to pay $2 to A or $3 to C so that A or C can complete their task and return the whole loan – we cannot fulfils B's needs at the moment ($5) – we can pay B only after either A or C pays the whole loan amount back
- This will be called as *safe state* – Everybody's tasks will be completed and we will get our entire money back

Second Scenario

- Suppose instead of giving $8 to B, we had given $10
  - So, now we are left with only $1
  - As we know, A needs $2, B needs $3, C needs $3
- This is the *unsafe state*, which causes a **deadlock**
- To avoid this, we discuss our previous safe state
  - We give $2 to A and let her complete his work.
  - A returns our $8, which leaves us with $9. Out of this $9, we can give $5 to B and let her finish his task with total $13 and then return the amount to us, which can be forwarded to C to eventually let her complete her task.
- It's called the **Banker's algorithm** because it could be used in the banking system so that banks never run out of resources and always stay in a safe state.

Banker's Algorithm in Operating Systems

- We have *processes* and *resources*; and we must know three things about them:
  - How much maximum of each resource type can each process request? [Max]
  - How much of each resource type has already been allocated to each process [Allocated]
  - How much of each resource type is available in the system? [Available]
- Let us consider an example where we have four processes (P1-P4) and three resources (A-C)

Banker's Algorithm – Step 1

- Since P2's needs < Available, first allocation will be made to P2

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2	7	4	3
P2	2	0	0	3	2	2				1	2	2
P3	4	0	1	9	0	4				5	0	3
P4	2	1	1	2	2	2				0	1	1

Banker's Algorithm – Step 2

- The state after P2's completion will be as follows

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	5	3	2	7	4	3
P3	4	0	1	9	0	4				5	0	3
P4	2	1	1	2	2	2				0	1	1

Banker's Algorithm – Step 3

- Since P4's needs < Available, the next allocation will be made to P4. After P4, completes:

Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	7	4	3	7	4	3
P3	4	0	1	9	0	4				5	0	3

Banker's Algorithm – Step 3

- Now P3 will get the allocation, followed by P1. So the sequence of P2-P4-P3-P1 will be a *safe state* without a deadlock

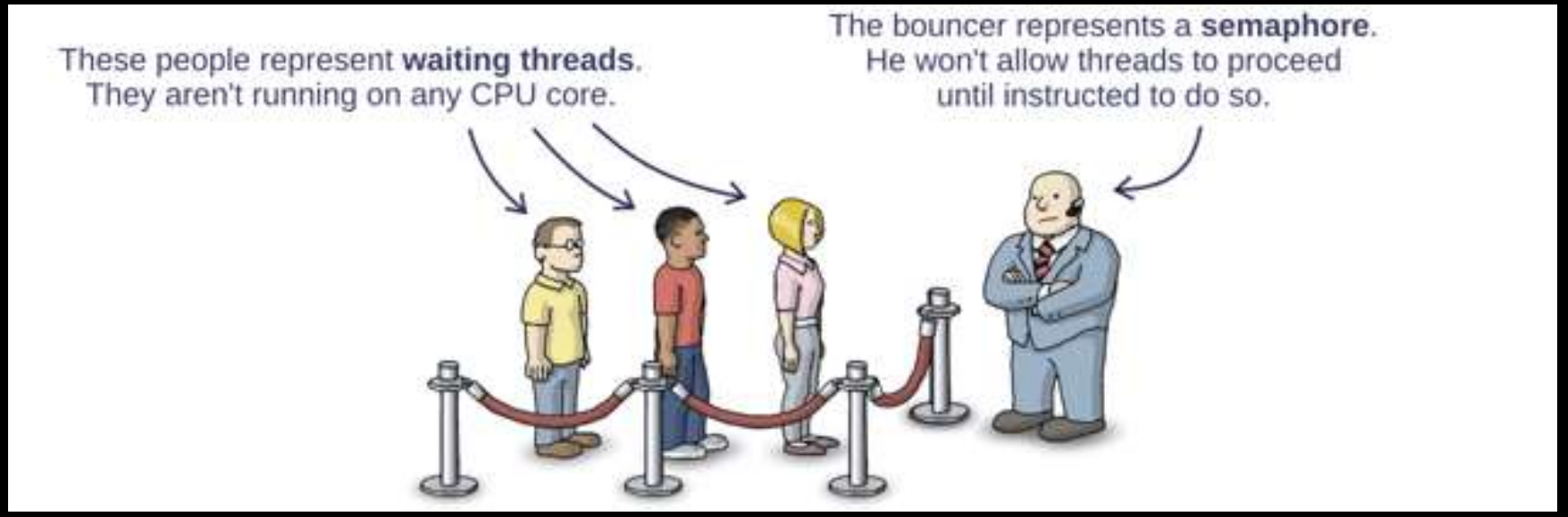
Process	Allocated			Maximum			Available			Need (Maximum Allocated)		
	A	B	C	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	7	4	3	7	4	3
P3	4	0	1	9	0	4				5	0	3

Semaphore

- When a process needs an exclusive access to a resource, it must wait until the resource is free
- To make processes wait till the needed resource can be allocated exclusively, we can make them *sleep*, so that they do not waste the CPU time
- We can use a **semaphore** to achieve this, which is an integer variable
- Helps in process synchronization to prevent race conditions

Understanding Semaphore

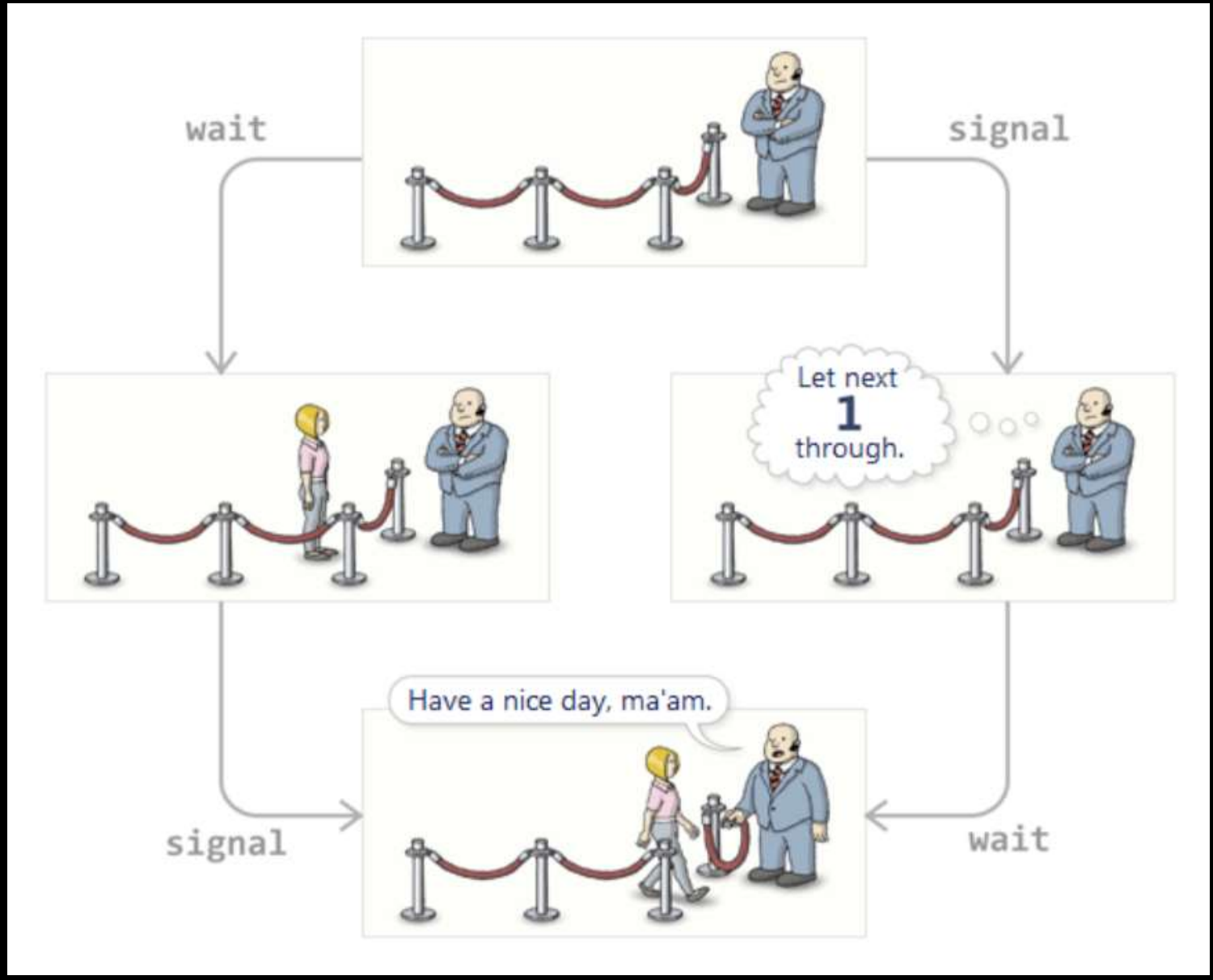
- Imagine a set of waiting threads, lined up in a queue – much like a lineup in front of a busy nightclub or theatre.
- A semaphore is like a bouncer at the front of the lineup. He only allows threads to proceed when instructed to do so.



Semaphore Operations

- A waiting process (needing the resource) makes a call to a *wait ()* function of the semaphore. In other words, it stands in the waiting queue of the semaphore
- When a running process (who has utilized the resource as per its needs) frees the resource, it calls a *signal ()* function on the semaphore. In other words, one of the waiting processes can now get an access to the resource
- To keep a track of how many processes are waiting, the semaphore maintains an integer count
  - A *wait ()* call increments the counter
  - A *signal ()* call decrements the counter
- See diagram on the next page

Wait and Signal Operations



Semaphore in OS

- **Semaphore** in OS is an integer value that indicates whether the resource required by the process is available or not
- The value of a semaphore is modified by wait() or signal() operation
- The wait() operation decrements the value of semaphore and the signal() operation increments the value of the semaphore
- The wait() operation is performed when the process wants to access the resources and the signal() operation is performed when the process want to release the resources
- The semaphore can be binary semaphore or the counting semaphore

Mutual Exclusion (Mutex)

- **Mutual Exclusion (Mutex)** is an object that allows multiple program threads to share the same resource, such as a file, but not simultaneously.
- When a program is started, a Mutex is created with a unique name.
- After this, any thread that needs the resource must lock the Mutex from other threads while it is using the resource. The Mutex is set to unlock when data is no longer needed.
- Thus, Mutex is a binary variable whose purpose is to provide locking mechanism.

General Mutex Operation

```
Lock (Mutex)

Do Work

Unlock (Mutex)
```

Semaphore versus Mutex

- Semaphore is a signaling mechanism.
- Semaphore restricts the number of simultaneous users of a shared resource up to a maximum number.
- Threads can request access to the resource (decrementing the semaphore) and can signal that they have finished using the resource (incrementing the semaphore).
- It allows number of thread to access shared resources.

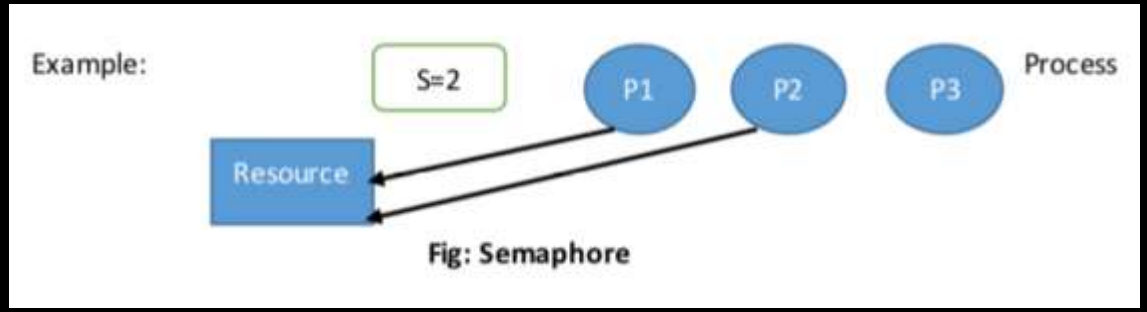


Fig: Semaphore

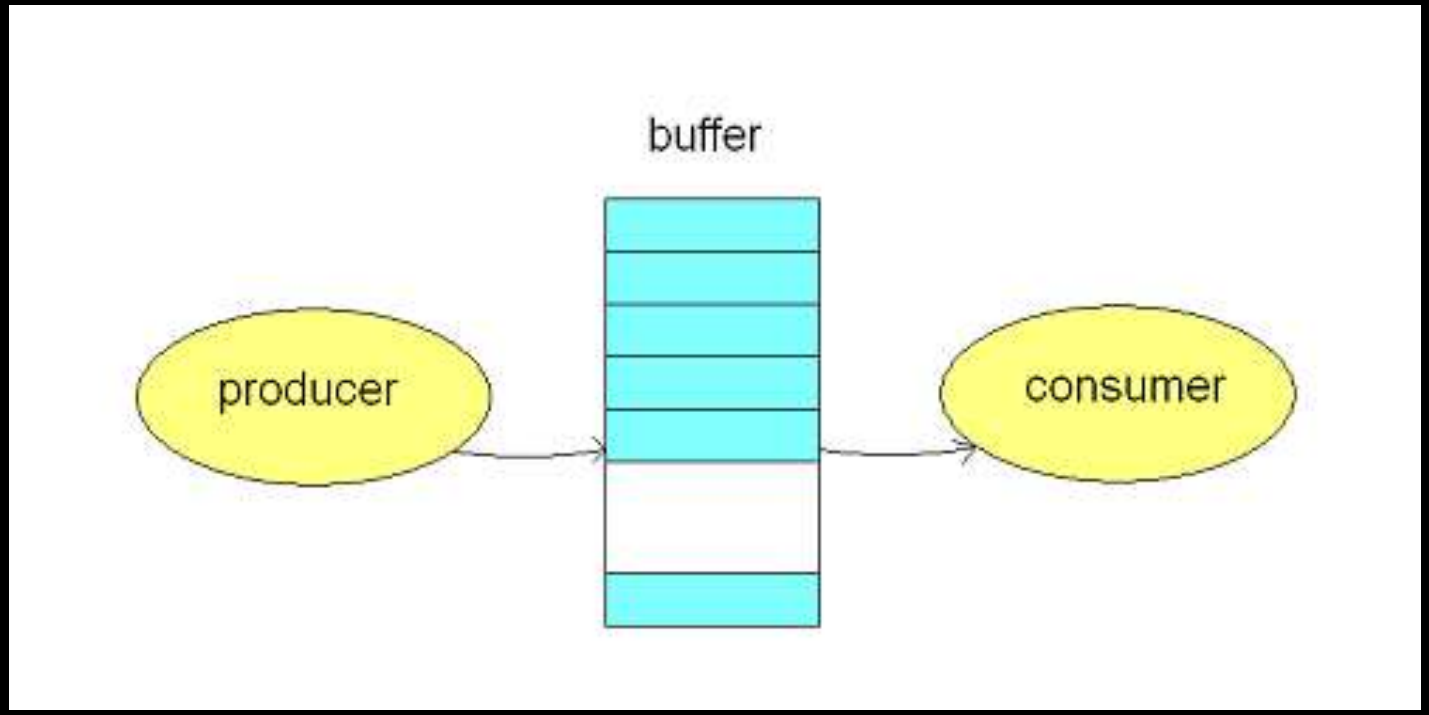
Main Differences

- Mutex is locking mechanism ownership method. Semaphore is a signaling mechanism.
- Mutex works in user space but semaphore works in kernel space.
- Mutex is generally binary, semaphore is an integer.

Producer-Consumer Problem

- **Producer-consumer problem** is also known as the **bounded-buffer problem**
- Two processes share a common fixed-size buffer
  - The producer puts information in
  - The consumer takes it out
- If the producer wants to write more information but the buffer is full, the producer goes to sleep and is woken up when the consumer removes one or more items from the buffer
- If the consumer wants to read more information but the buffer is empty, the consumer goes to sleep and is woken up when the producer adds one or more items to the buffer

Producer-Consumer Problem



The Variables

- We will have a *count* variable that indicates the current number of items in the buffer
- Another variable *n* will indicate the maximum capacity of the buffer
- The producer checks if count equals n
  - If yes, it goes to sleep
  - If not, it adds an item and increments count
- The consumer checks if n equals 0
  - If yes, it goes to sleep
  - If not, it removes an item and decrements count
- Both producer and consumer check if the other needs to be awakened, and if yes, do so

Race Condition

- A problem called as **race condition** can occur because there are no restrictions on accessing the *count* variable
- Suppose the buffer is empty and the consumer has just read the count variable to find its value as 0
- Naturally, the scheduler will stop the consumer and decide to start the producer
- The producer inserts an item in the buffer and increments the count and notices that it is now 1
- Since count was 0 and thus the consumer must be sleeping, the producer tries to wake up the consumer
- However, the consumer was not sleeping, and hence this wake up call is lost
- The consumer still thinks that count = 0, so it will go to sleep
- Sooner or later, the producer will fill the buffer and also go to sleep
- Both will sleep forever
- Solution: **Semaphore**, discussed earlier

semaphore.py

```
# importing the modules
from threading import *
import time

# creating thread instance where count = 3
obj = Semaphore(3)

# creating instance
def display(name):

                # calling acquire method
                obj.acquire()
                for i in range(5):
                                print('Hello, ', end = '')
                                time.sleep(1)
                                print(name)

                                # calling release method
                                obj.release()

# creating multiple thread
t1 = Thread(target = display , args = ('Thread-1',))
t2 = Thread(target = display , args = ('Thread-2',))
t3 = Thread(target = display , args = ('Thread-3',))
t4 = Thread(target = display , args = ('Thread-4',))
```

mutex.py

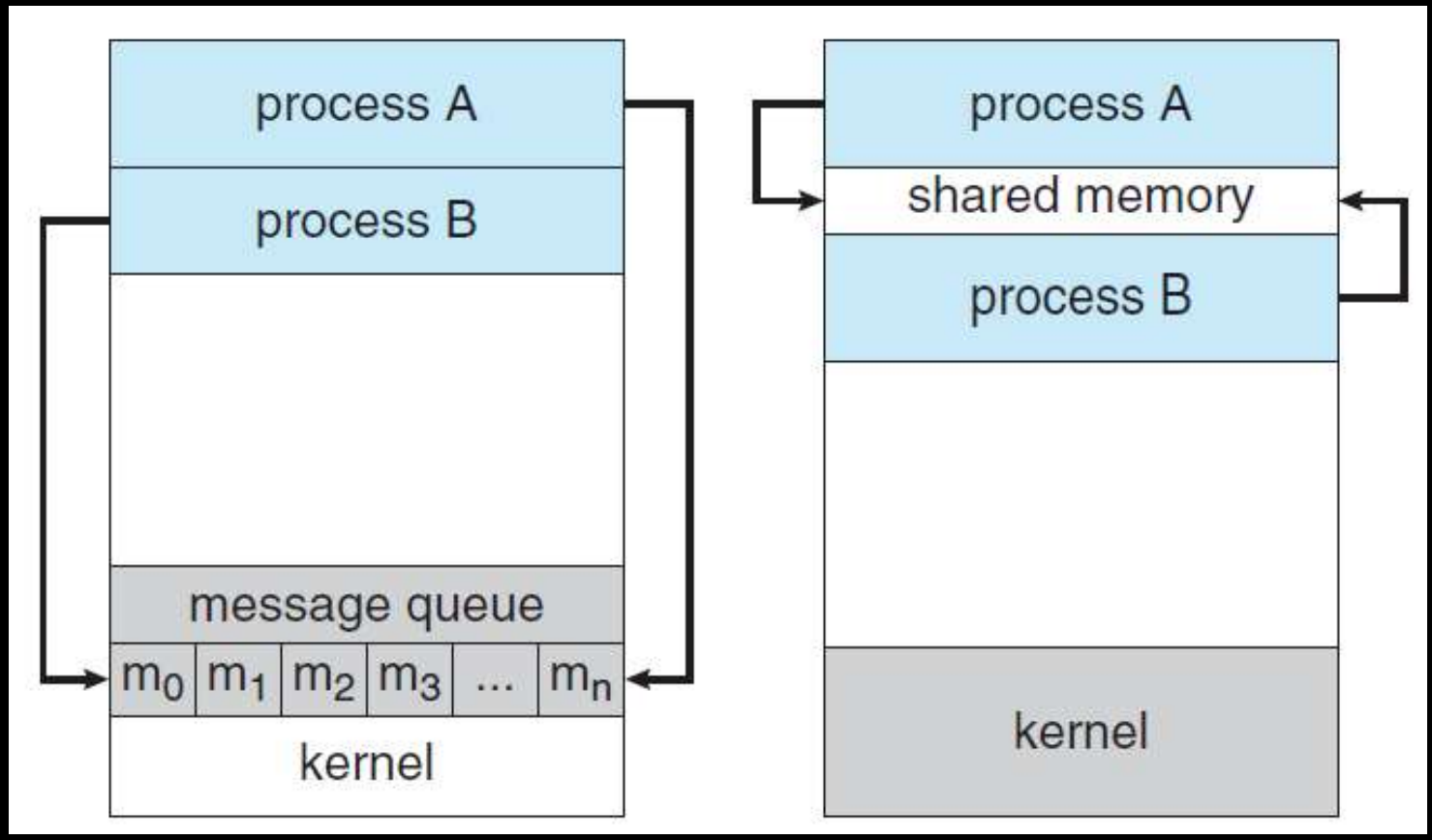
```
# example of a mutual exclusion (mutex) lock
from time import sleep
from random import random
from threading import Thread
from threading import Lock

# work function
def task(lock, identifier, value):
    # acquire the lock
    with lock:
        print(f'>thread {identifier} got the lock, sleeping for {value}')
        sleep(value)

# create a shared lock
lock = Lock()
# start a few threads that attempt to execute the same critical section
for i in range(10):
    # start a thread
    Thread(target=task, args=(lock, i, random())).start()
# wait for all threads to finish...
```

Inter Process Communication (IPC)

Message Queues

- Cooperating processes require an **Inter Process Communication (IPC)** mechanism that will allow them to exchange data and information.
- There are two fundamental models of inter process communication: **shared memory** and **message passing**.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message Passing versus Shared Memory



Shared Memory Systems

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

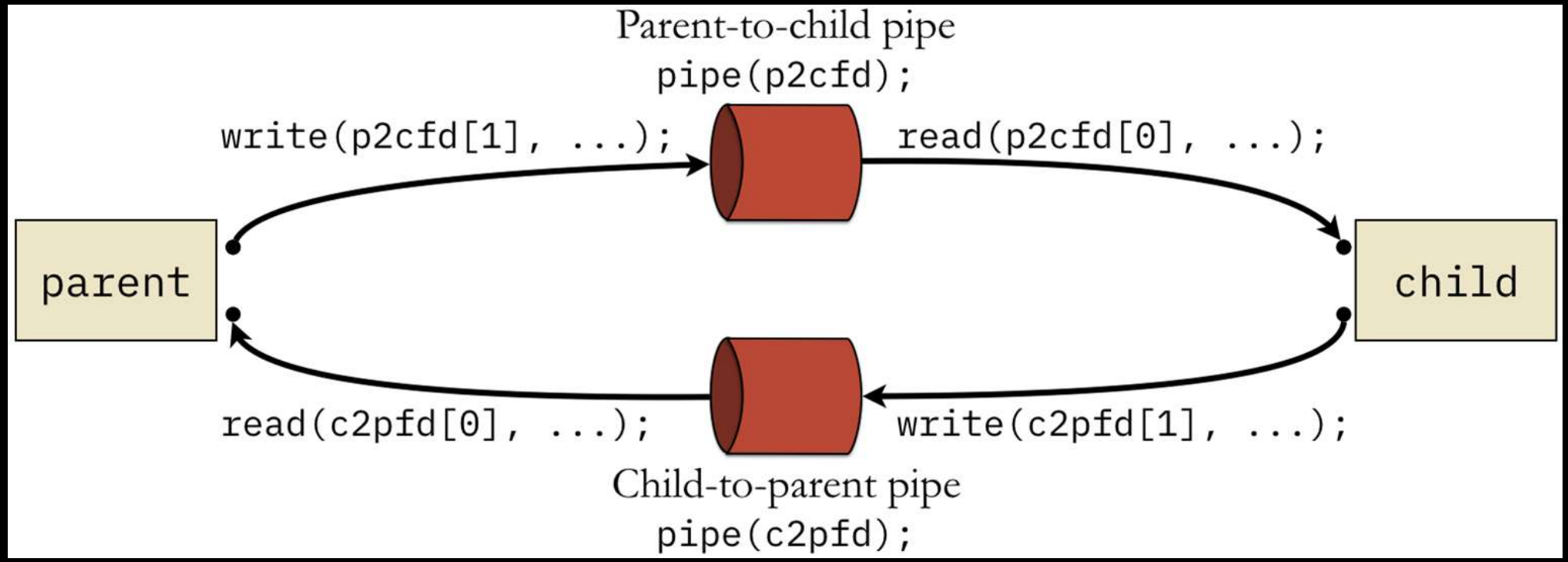
Message Passing Systems

- **Message passing** provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
- It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- Has to make several choices
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering

Pipes

- **Pipes** allow processes to communicate using a unidirectional byte stream, with two ends designated by distinct **file descriptors**.
- A common visual analogy for a pipe is a real-world water pipe; water that is poured into one end of the pipe comes out the other end.
- In the case of IPC, the “water” is the sequence of bytes being sent between the processes; the bytes are written into one end of the pipe and read from the other end.

Pipe Concept



Pipe Characteristics

- Unidirectional: One end must be designated as the reading end and the other as the writing end. Note that there is no restriction that different processes must read from and write to the pipe; rather, if one process writes to a pipe then immediately reads from it, the process will receive its own message. If two processes need to exchange messages back and forth, they should use two pipes.
- Order preserving: All data read from the receiving end of the pipe will match the order in which it was written into the pipe. There is no way to designate some data as higher priority to ensure it is read first.
- Unstructured byte streams: There are no pre-defined characteristics to the data exchanged, such as a predictable message length. The processes using the pipe must agree on a communication protocol and handle errors appropriately.

Pipe Example – Linux Shell

- Create a file called as file1.txt:
- Zaheer
- Aman
- Trisha
- Raman
- Lokesh
- Bharat
- Pallavi
- Susan
- Dilip

- cat file1.txt | sort

Pipe Example – Linux Shell

- Create a file called as file2.txt:
- OS
- CPP
- Java
- C
- Python
- DS
- DBMS
- OS
- Java
- CPP

- cat file2.txt | sort | uniq

Pipe Example – Linux Shell

- cat file1.txt | grep h

Pipe Example - Linux Shell

- ls -l | sort -n -k 5 | tail -n 1 | awk '{print $NF}’

- This command line creates four processes that are linked together. First, the ls command prints out the list of files along with their details.
- This list is sent as input to sort, which sorts numerically based on the 5th field (the file size).
- The tail process then grabs the last line, which is the line for the largest file.
- Finally, awk will print the last field of that line, which is the file name of whatever file is the largest.

FIFO

- Pipes can be used for related processes only (i.e. parent and child)
- When random processes need to communicate with each other, we cannot use pipes
- Solution: **FIFO (First In First Out)**
- FIFO attaches a file name to a pipe
- Hence FIFO is also called as **named pipe**

FIFO Example

- A common use for FIFOs is to create client/server applications on the same machine.
- For example, consider an anti-virus server that runs in the background, scanning for corrupted files.
- When the system administrator wants to get a report on potentially bad files, they run a client application that uses a FIFO to initiate contact with the server.
- Both the server and the client application are distinct processes that are running separate programs. That is, neither process was created by either of them calling fork().
- As such, an anonymous pipe() call would not work. Instead, both processes use the name attached to the FIFO to set up the communication.

FIFO Example

- On the VirtualBox Ubuntu (not WSL)
- Open two terminal windows
  - First terminal: fifowrite.c (See next slides)
    - Compile and run (gcc fifowrite.c then ./a.out)
  - Second terminal: fiforead.c (See next slides)
    - Compile and run (gcc fiforead.c then ./a.out)

fifowrite.c

```
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)
    {
        // Open FIFO for write only
        fd = open(myfifo, O_WRONLY);
```

fiforead.c

```
// C program to implement one side of FIFO
// This side reads first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    int fd1;

    // FIFO file path
    char * myfifo = "/tmp/myfifo";

    // Creating the named file(FIFO)
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);

    char str1[80], str2[80];
    while (1)
    {
        // First open in read only and read
        fd1 = open(myfifo,O_RDONLY);
```

Running

- Type something in the first terminal and see what happens to the second terminal

```
atul@Ubuntu:~$ mkdir bash_course
atul@Ubuntu:~$ cd bash_course/
atul@Ubuntu:~/bash_course$ nano fifo_write.c
atul@Ubuntu:~/bash_course$ gcc fifo_write.c
atul@Ubuntu:~/bash_course$ ./a.out
test
```

```
atul@Ubuntu:~/bash_course$ nano fifo_read.c
atul@Ubuntu:~/bash_course$ gcc fifo_read.c
atul@Ubuntu:~/bash_course$ ./a.out
User1: test
```

- Now the opposite

```
atul@Ubuntu:~/bash_course$ nano fifo_read.c
atul@Ubuntu:~/bash_course$ gcc fifo_read.c
atul@Ubuntu:~/bash_course$ ./a.out
User1: test

test
```

```
atul@Ubuntu:~$ mkdir bash_course
atul@Ubuntu:~$ cd bash_course/
atul@Ubuntu:~/bash_course$ nano fifo_write.c
atul@Ubuntu:~/bash_course$ gcc fifo_write.c
atul@Ubuntu:~/bash_course$ ./a.out
test
User2: test
```