Operating System (OS)

Definition of OS

- **Program** = Set of instructions
- **Software** = Set of programs

Types of Software

1. **Application Software**
    - Uses the OS and performs tasks or solves problems for the user.
    - Examples: Word, PowerPoint, Browser, add.c, DB.java, palindrome.cpp, ex.js
2. **System Software**
    - Interacts with the system.
    - Examples: OS, Compilers, Interpreters, Loaders, Device Drivers, Linkers

OS (Operating System)

- OS is a System Software.
- OS acts as an interface:
    - Between SYSTEM and APPLICATION SOFTWARE
    - Between HUMAN-USER and SYSTEM
- OS manages the system through the following tasks:
    - Process Management
    - Memory Management
    - Device Management
    - Security

What is the System?

1. **Storage -** store data + programs
    - Hard Disk / SSD / Secondary Memory: Non-volatile (data preserved after power off)
    - RAM / Memory / Primary Memory: Volatile (data erased after power off)
    - ROM: Non-volatile
    - Cache Memory: Volatile
    - Registers: Volatile

2. **I/O Devices**
    - **Input Devices:** Keyboard, Mouse, Scanner, Mic, PD, Touch Screen, Webcam, Biometric Scanner (Thumb, Retina), Bar Code Reader, QR Code Scanner, CD
    - **Output Devices:** Printer, Monitor/Console, Speaker, PD, CD, Projector
3. **Processor**
    - **CPU: Executes CPU instructions**
        - Arithmetic: +, -, *, /, %
        - Logical: &&, ||, !
        - Relational: <, >, <=, >=, ==, !=
        - ALU (Arithmetic and Logic Unit): Executes arithmetic and logical operations.
        - Registers: Temporary storage:
            - IR (Instruction Register): Stores the current instruction
            - DR0 (Data Register 0): Stores the first operand
            - DR1 (Data Register 1): Stores the second operand
            - Accumulator: Temporarily stores results
            - PC (Program Counter): Stores the address of the next instruction
    - **I/O Coprocessor (DMA Controller): Executes I/O instructions**
        - DMA (Direct Memory Access): Transfers data
            - From RAM to Output Device
            - From Input Device to RAM

OS as an Interface

1. **GUI (Graphical User Interface)**
2. **CLI (Command Line Interface)**

Types of Operating Systems

1. **Server OS**: Solaris, Unix
2. **GPOS (General Purpose OS):** Windows XP, Win10, Win11, Apple, Linux-based OS
3. **Network OS**: WinNT, Netware
4. **Mobile OS:** iOS, Android, Windows
5. **Real-Time OS:** Deadline-based OS (e.g., RTLinux)
6. **Embedded OS:** Embedded in machines like cars, washing machines
7. **Single-User OS:** Only one user can log in at a time (e.g., Windows)
8. **Multi-User OS:** Many users can stay logged in simultaneously (e.g., Ubuntu)

Linux-Based OS

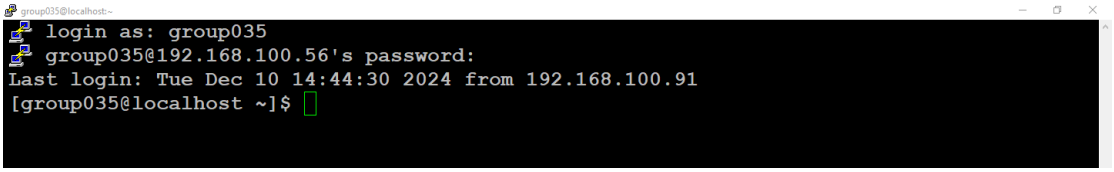
- Examples: Ubuntu, Kali, Red Hat, Mandrake, Arc, Fedora, Mandriva, SUSE, Debian, BOSS, Cent OS
- Features:
    - Open Source: Source code available for reading and modification
    - Maintained by community contributions
    - GPL (GNU Public License): Free to use

LAB – DAY 1

Putty Configuration

- IP Address: 192.168.100.56
- Customization:
    - Change the font size
    - Modify colors

Working with Linux Commands

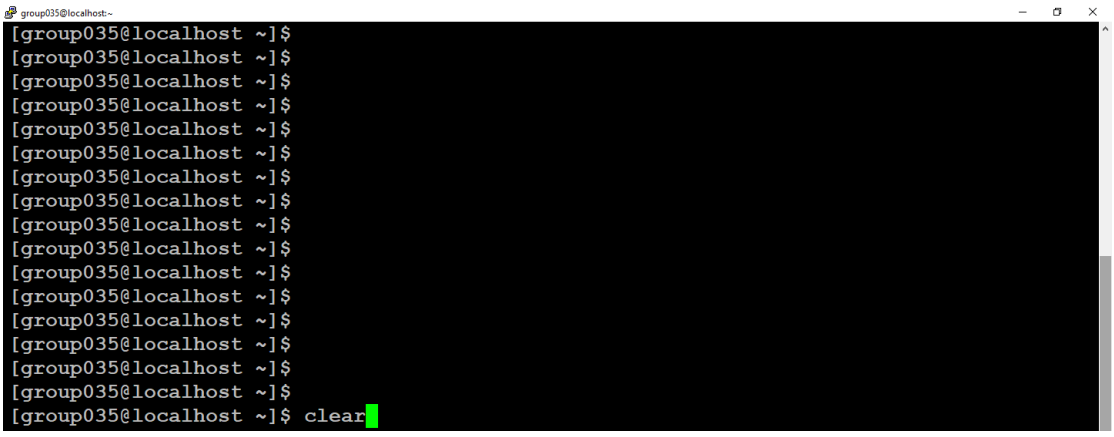


1. Observe the Prompt

- Example: [faculty@localhost ~]$
    - Square Bracket: [ ]
    - Username: faculty
    - Hostname: localhost
    - Tilde: ~
    - Dollar Sign: $

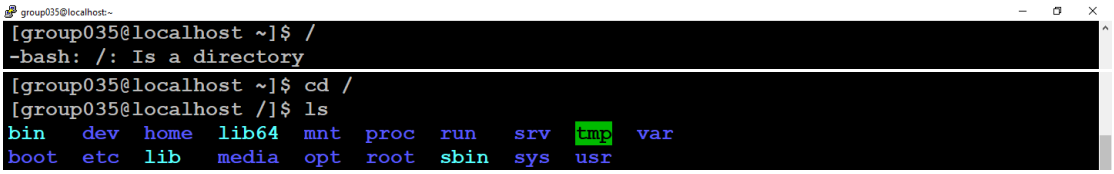
2. Clearing the Terminal

- Press Enter multiple times
- Type **clear**: Clears the screen



3. Directories and Files

- Root Directory: / (Always one root directory)
    - Windows Root Directory: Multiple root directories (C:, D:, F:, etc.)

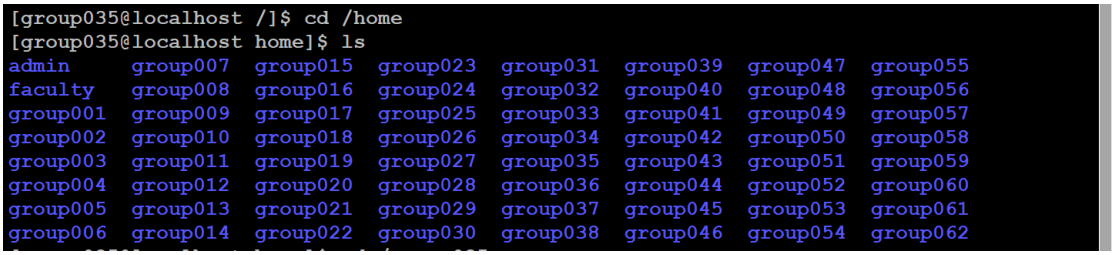


4. Check Current Directory

- Command: **pwd** (Present Working Directory)
    - By default, pwd shows the directory with your username
    - Tilde (~): Symbol for Home Directory

5. Change Directory

- Command: cd
    - To Root Directory: **cd /**
    - Check Directory: **pwd**



6. List Contents

- Command: ls or dir
    - Shows the contents of the present working directory

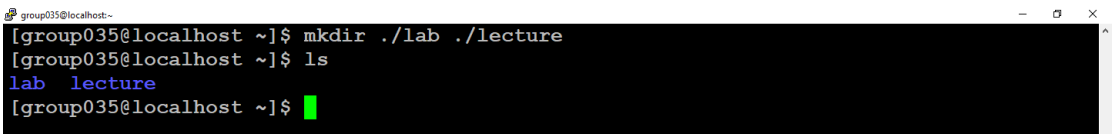
7. Absolute vs. Relative Path

- **Absolute Path:** Path starting from root directory (e.g., **cd /home**)

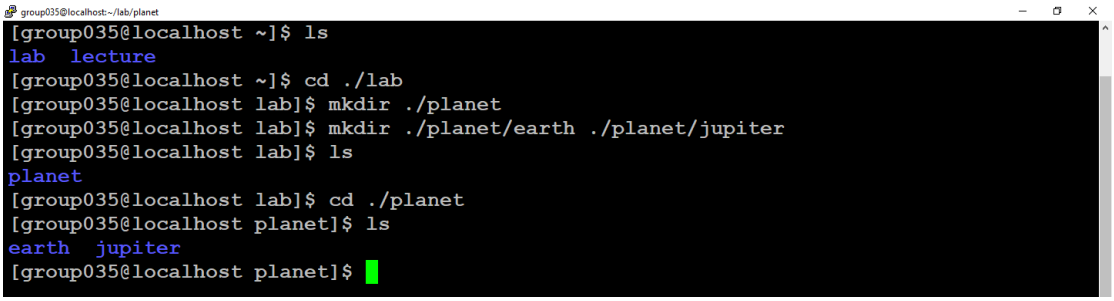
- **Relative Path:** Path from the current folder (e.g., **cd ./home** or **cd home**)

8. Creating Directories

- Ensure your pwd is your home directory.
- Create Folders:
    - Command: **mkdir ./lab ./lecture**
    - Check: **ls**

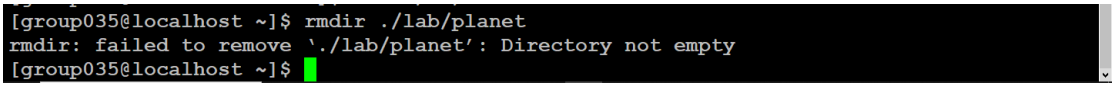


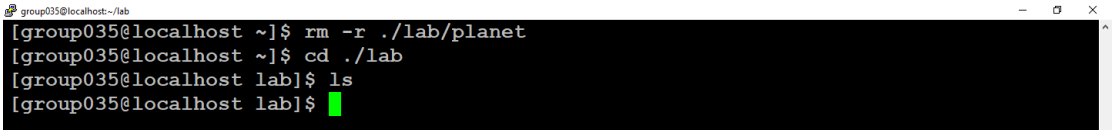
- Nested Directories:
    - Example: mkdir ./lab/planets
- Multiple Folders:
    - Example: mkdir ./lab/planets/earth ./lab/planets/Jupiter



9. Removing Directories

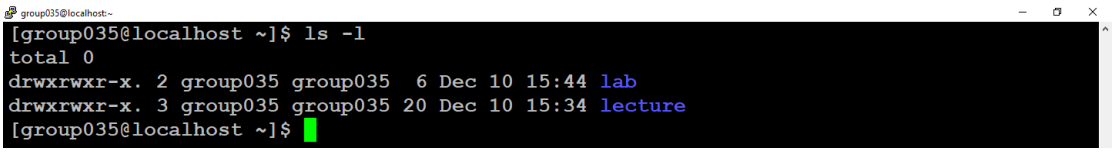
- **Empty Folders:**
  - Command: **rmdir folder_name**
- **Non-Empty Folders:**
  - Command: **rm -r folder_name**





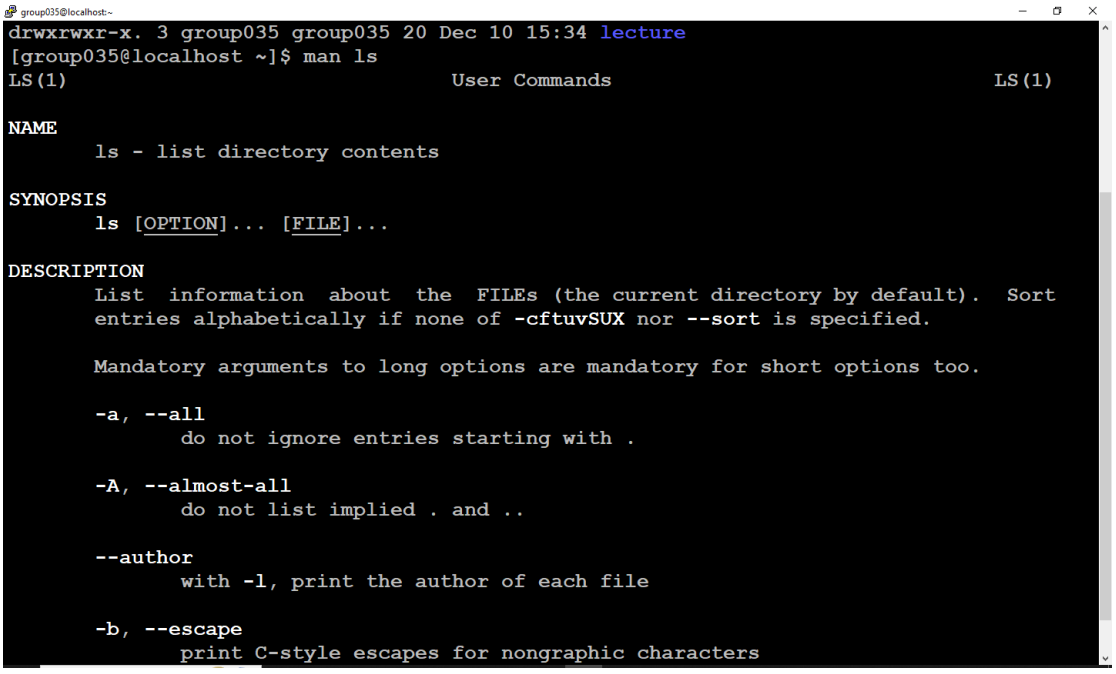
10.Detailed Listing

- Command**: ls -l**: Shows detailed list of contents



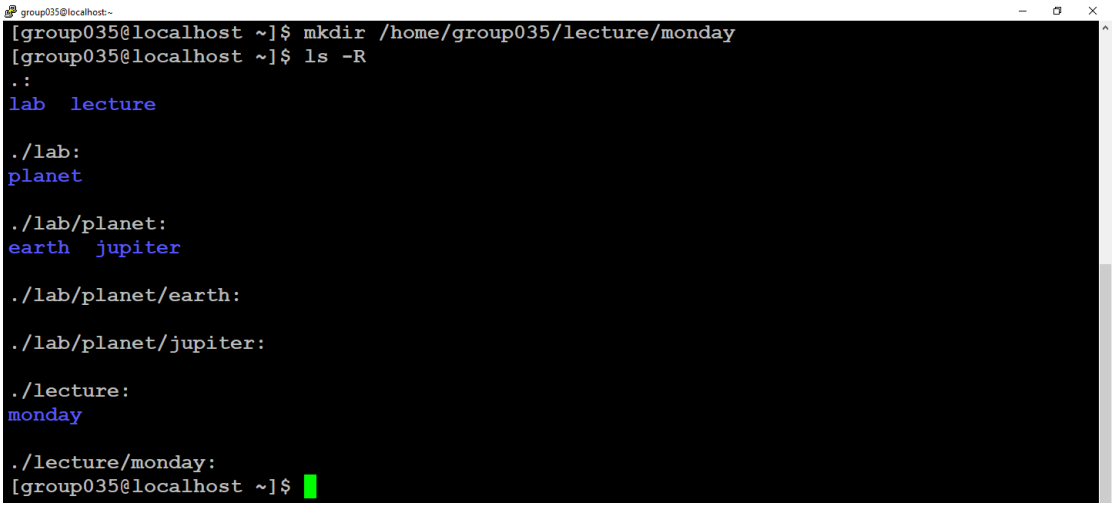
11.Help for Commands

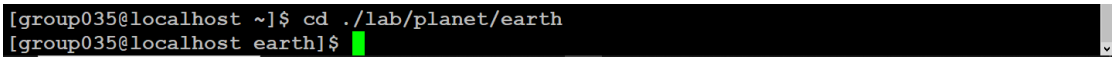
- Command: man command_name
    - Example: **man ls**



12. Move to home folder = cd ../ check pwd

13.Detailed folder/directory structure

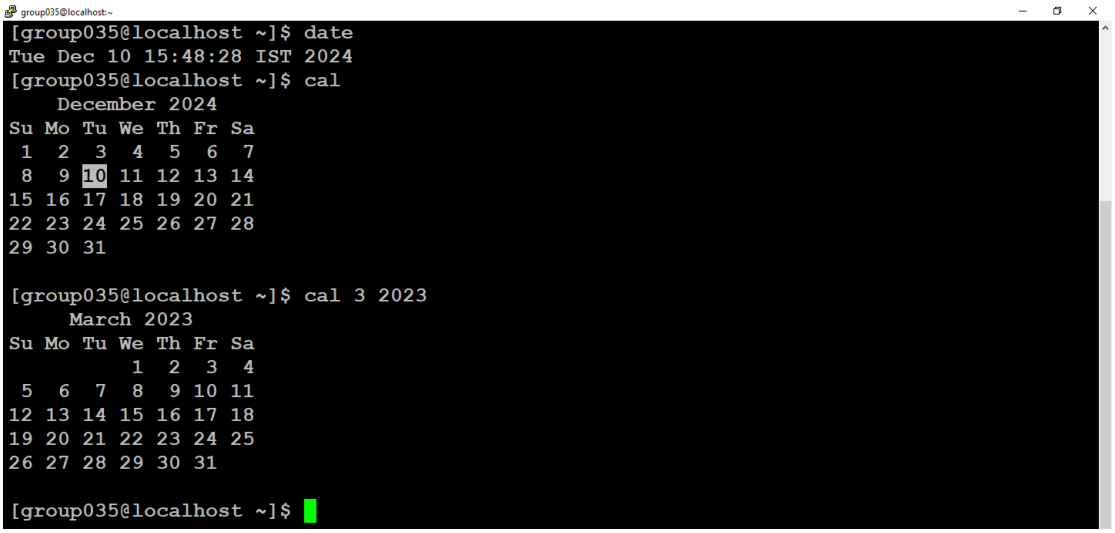




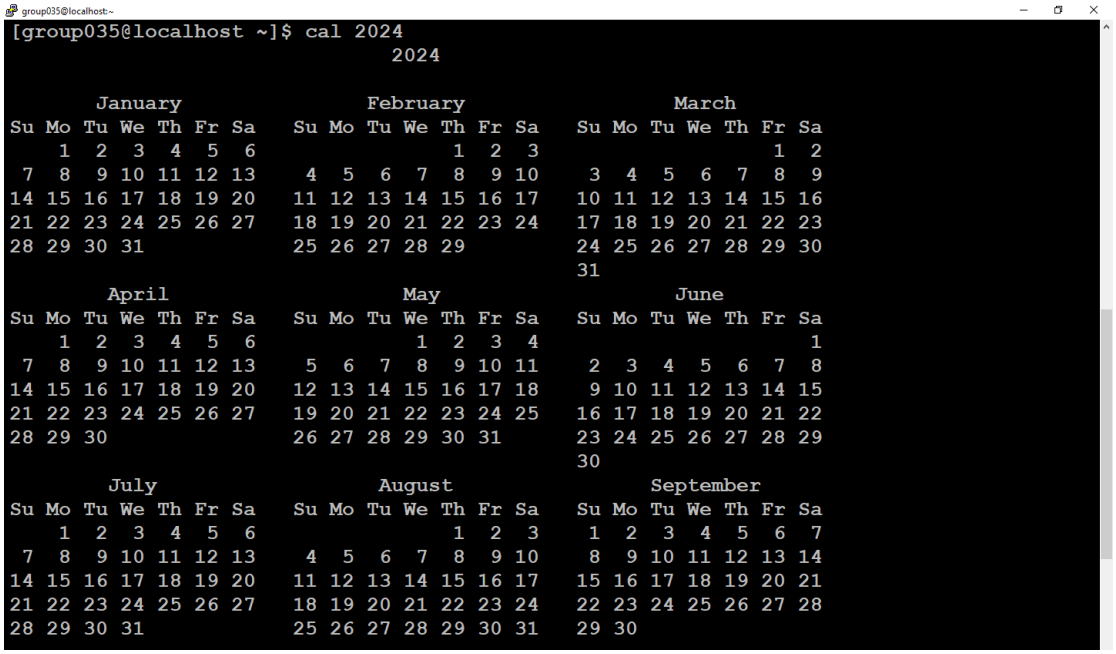
14.Not able to access another person Dir…

15.Miscellaneous Commands

- Date: date
- Calendar: cal
    - Help: man cal (Find the option to see the calendar for the whole year)



- Full Year Cal



Kernel Space and User Space

- **Kernel Space**: Space in RAM where kernel programs are loaded.
- **User Space**: Space in RAM where user programs are loaded.

Kernel Mode and User Mode

- **Kernel Mode**: Privileged mode of execution.
- **User Mode**: Non-privileged mode of execution.

Interrupts

- **Hardware Interrupt (H/W Interrupt)**: Signal passed by an I/O device to the processor.
- **Software Interrupt (S/W Interrupt)**: Signal passed by software to the processor.

Interrupt Handling Process:

1. Interrupt is sent to a CPU pin.
2. When an interrupt occurs, the current job of the CPU is paused.
3. Control goes to the Interrupt Handler.
4. The kernel maintains a table mapping interrupt numbers with handlers. This is called the **Interrupt Table** or **Interrupt Vector Table**:

Number	Pointer to Handler
1	f1
2	f2
...	...

5. The Interrupt Handler executes, and the CPU may resume execution of other programs.

Types of Interrupts:

- **Maskable Interrupts**: Can be ignored.
- **Non-Maskable Interrupts**: Cannot be ignored by the kernel and must be handled.

Process Management

- **Program**: A file stored on the hard disk.
- **Process**: A program that is in execution (loaded in RAM).
- **PID**: Every process gets a unique Process ID.
- **Address Space/Process Space**: Contains the code and data of the program:
    - Code Segment
    - Data Segment
    - Stack Segment
    - Heap Segment

Process Life Cycle:

1. **Creation State**:
    - Allocate Address Space in RAM.
    - Create a Process Control Block (PCB) containing:
        - PID
        - Location of Address Space
        - Current State
        - Context Information
        - Priority
2. **Ready State**:
    - Kernel maintains a **Ready Queue** in Kernel Space.
    - The process is added to the Ready Queue and waits for the CPU.
3. **Running State**:
    - The process uses the CPU to execute instructions.
4. **I/O Wait State**:
    - The process waits for I/O instructions to complete.
5. **Terminate State**:
    - Deallocate Address Space.
    - Release PCB Variables.

Part of the process execution lifecycle

- **CPU Burst Time**: Time required to execute all CPU instructions of a process.
- **I/O Burst Time**: Time required to execute all I/O instructions of a process.

Turnaround Time (Ta):

- **Formula**: Ta = Completion Time – Start Time
- **Alternate Formula**: Ta = CPU Burst Time + Wait Time
- **Lower Ta is good**. To reduce Ta, the kernel can reduce wait time.
- **Average Turnaround Time (Avg Ta)**:

Avg Ta = (Ta1 + Ta2 + ... + Tan) / n

Wait Time (Wt):

- **Definition**: Total time spent in the Ready Queue by a process.
- **Low Wt is good**.
- **Average Wait Time (Avg Wt)**:

Avg Wt = (Wt1 + Wt2 + ... + Wtn) / n

Throughput:

- **Definition**: Number of processes completed in unit time.
- **High throughput is good**. To increase throughput, reduce Wt and Ta.

Response Time:

- **Definition**: Time taken by the process to respond to the request.
- **Low response time is good**.

Process Scheduling

- **CPU Scheduling**: Low-level scheduling handled by the kernel.
- **Objective**: Select a process from the Ready Queue for CPU execution to minimize Avg Wt, maximize throughput, and reduce response time.

Scheduling Algorithms

1. **First In First Out (FIFO)**:
    - **Process Selection Criteria**: Process at the front of the queue is selected for CPU usage.
    - **Execution Duration**: Ideally, the process executes for its CPU Burst Time. However, it may leave the CPU due to I/O or interrupt.
    - **Advantages**:
        - Simple.
        - All processes get a chance to run.
    - **Disadvantages**:
        - Processes waiting behind long CPU Burst Time processes experience high Wt, affecting Avg Wt, Ta, and throughput.

Example Calculation:

Process	CPU Burst Time	Arrival Time (Ready Queue)
P1	3 ms	0
P2	5 ms	2
P3	4 ms	1

Wait Times:

- Wt1 = 0 - 0 = 0
- Wt2 = 7 - 2 = 5
- Wt3 = 3 - 1 = 2

Average Wait Time (Avg Wt):

Avg Wt = (0 + 5 + 2) / 3 = 2.33 ms

Turnaround Times:

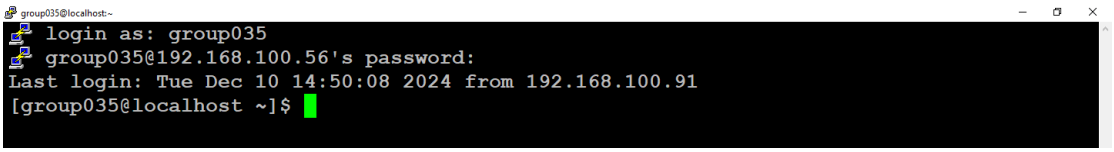
- Ta1 = 3 + 0 = 3 ms
- Ta2 = 5 + 5 = 10 ms

- Ta3 = 4 + 2 = 6 ms

Average Turnaround Time (Avg Ta):

Avg Ta = (3 + 10 + 6) / 3 = 6.33 ms

File and Folder Operations



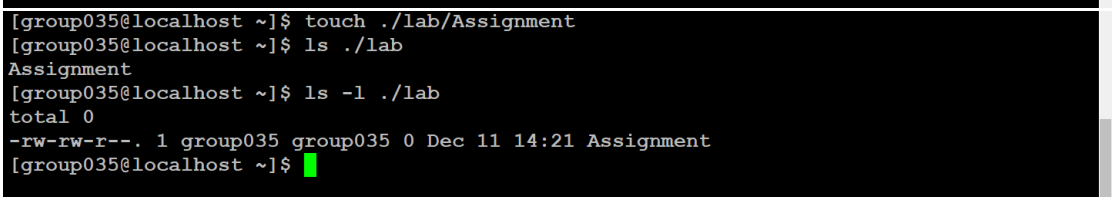
Creating and Managing Files

1. **Create a 0 KB File in lab/assignment**:

```
touch ./lab/assignment
          CHECK in lab folder   ls ./lab
          CHECK the size ls -l ./lab
```

2. **Edit File Using vi Editor**:

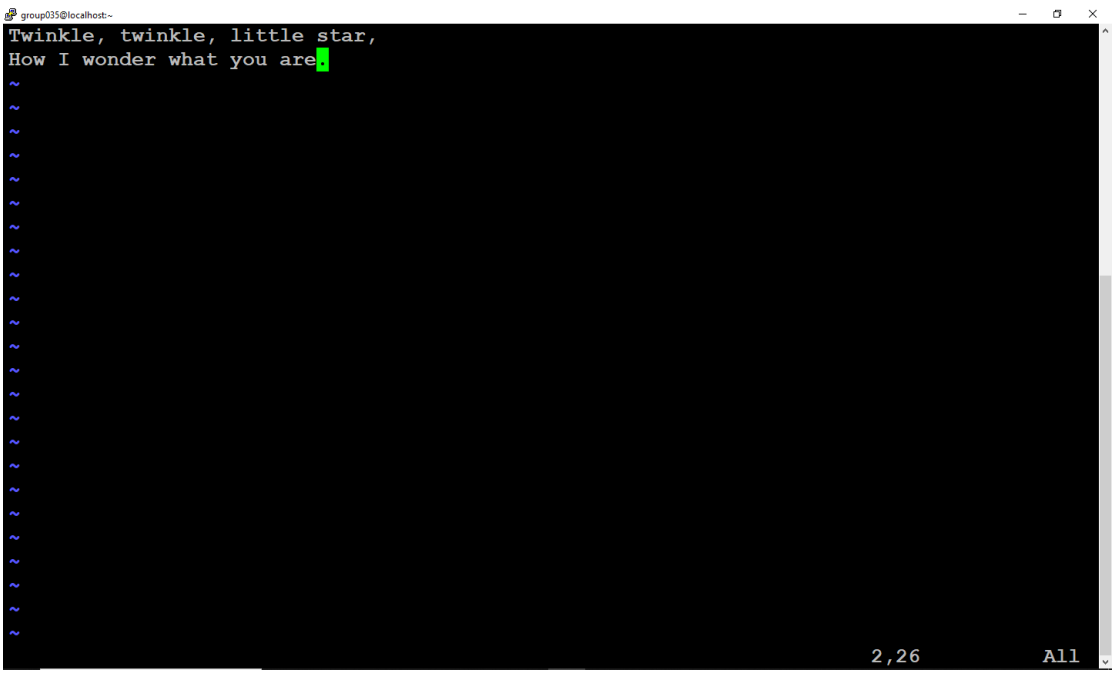
```
vi  ./lab/assignment

            vi works in two MODES  -  edit mode ,command mode
            press i to shift to edit mode
            type your content
            press esc to shift to command mode
            type :w to save the file
            type :wq to save and quit editor
            type :q!  to quit without saving
```

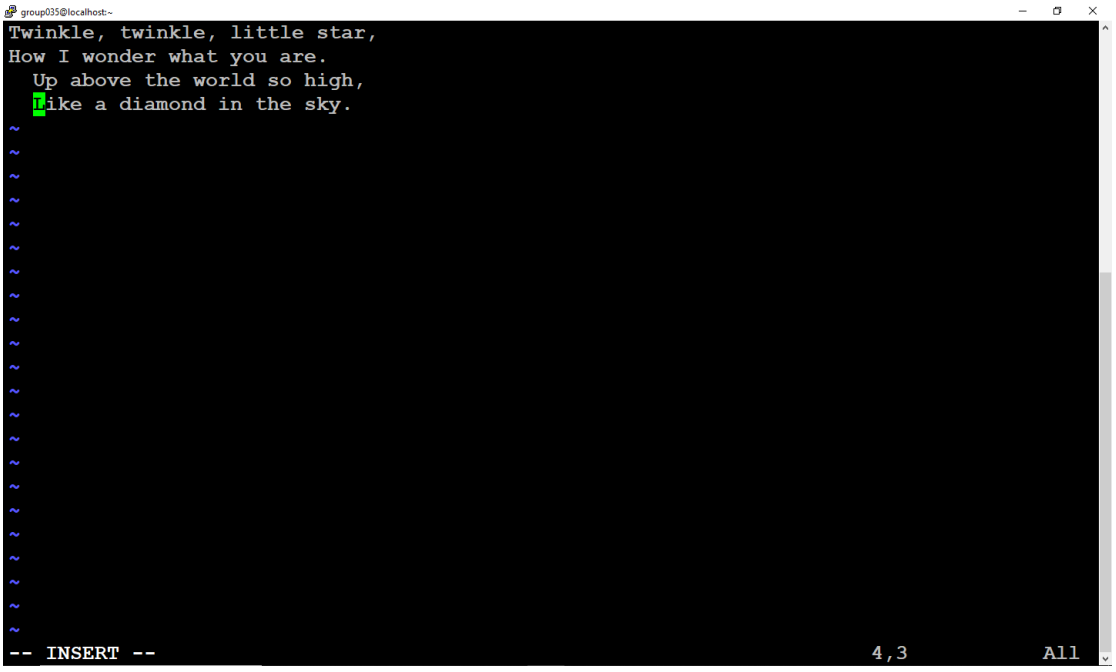


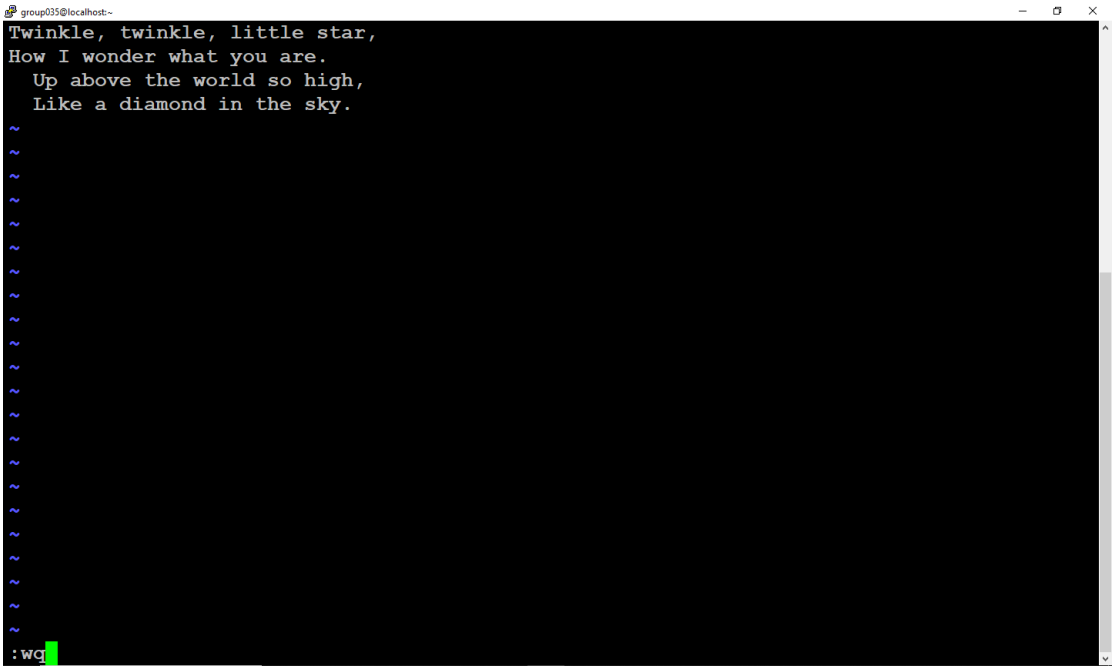
3. **View File Content**:

```
cat ./lab/assignment
tac ./lab/assignment
```

```
[group035@localhost ~]$ vi ./lab/Assignment
[group035@localhost ~]$ cat ./lab/Assignment
Twinkle, twinkle, little star,
How I wonder what you are.
[group035@localhost ~]$
```







4. **Create and Remove Files**:

```
vi test
view content using cat = cat test
remove the file  =    rm ./test
```

```
[group035@localhost ~]$ cat ./lab/Assignment
Twinkle, twinkle, little star,
How I wonder what you are.
  Up above the world so high,
  Like a diamond in the sky.
[group035@localhost ~]$
```

```
[group035@localhost ~]$ touch ./lab/test
[group035@localhost ~]$ ls -R
.:
lab  lecture

./lab:
Assignment  test

./lecture:
[group035@localhost ~]$ vi ./lab/test
[group035@localhost ~]$ cat ./lab/test
Hello
[group035@localhost ~]$ rm ./lab/test
[group035@localhost ~]$ ls -R
.:
lab  lecture

./lab:
Assignment

./lecture:
[group035@localhost ~]$
```

File Copy and Move Operations

1. **Copy File assignment to lecture Folder**:

cp ./lab/assignment ./lecture

```
group035@localhost:~
[group035@localhost ~]$ ls -R
.:
lab  lecture

./lab:
Assignment

./lecture:
[group035@localhost ~]$ cp ./lab/Assignment ./lecture
[group035@localhost ~]$ ls -R
.:
lab  lecture

./lab:
Assignment

./lecture:
Assignment
[group035@localhost ~]$ cat ./lecture/Assignment
Twinkle, twinkle, little star,
How I wonder what you are.
  Up above the world so high,
  Like a diamond in the sky.
[group035@localhost ~]$
```

2. **Copy Entire lecture Folder to backup Folder**:

mkdir ./backup

cp -r ./lecture ./backup

ls ./backup

ls -R ./backup

```
group035@localhost:~
[group035@localhost ~]$ ls -R
.:
lab  lecture

./lab:
Assignment

./lecture:
Assignment
[group035@localhost ~]$ cp -r ./lecture ./backup
[group035@localhost ~]$ ls ./backup
Assignment
[group035@localhost ~]$
```

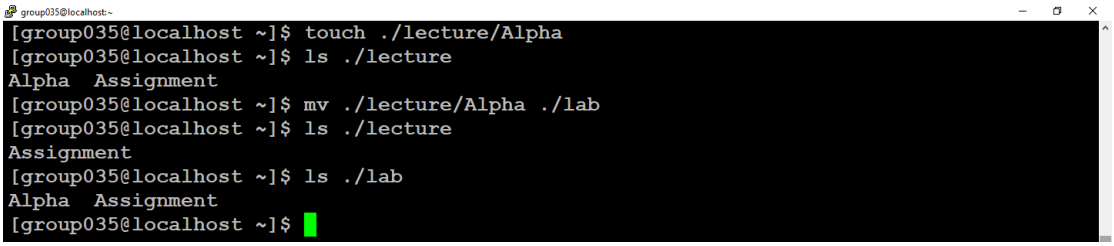
```
group035@localhost:~
[group035@localhost ~]$ ls -R
.:
backup  lab  lecture

./backup:
Assignment

./lab:
Assignment

./lecture:
Assignment
[group035@localhost ~]$
```

3. **Move File alpha from lecture to lab Folder**:

touch ./lecture/alpha

mv ./lecture/alpha ./lab

mv ./lab/alpha ./lab/beta



Environment Variables

1. **View Environment Variables**:

env

echo "hi"

echo $PATH

echo $HOME

echo $LOGNAME

echo $USER

echo $PS1

echo $PS2

2. **Change Prompt String**: **export PS1="\W >>>>"**

   Restore to original: **export PS1="[\u@\h \W]\$"**

3. **Shell Basics**: **echo $SHELL**

- Default shell: bash
- Other shells: Kshell, Cshell, TCShell.

CPU Scheduler

Scheduling Algorithms

1. **FIFO / FCFS**
2. **SJF (Shortest Job First)**

Definition: Process in the ready queue having the lowest CPU burst time is selected for the next CPU usage.

Preemptive

- Forcefully removes a process from the CPU as another process with a higher priority arrives.
- **Preemptive SJF:**
  - Process with the smallest Tcpu in the ready queue gets the CPU.
  - If a process with a lower Tcpu arrives, the current process is **forced to leave** the CPU.

Non-Preemptive

- Process with the smallest Tcpu in the ready queue gets the CPU.
- If a process with a lower Tcpu arrives, the current process **continues** and the new process waits in the ready queue.

How Long Will the Process Get the CPU?

1. **Non-Preemptive:** Ideally for Tcpu time. If an I/O instruction or interrupt occurs, the process leaves the CPU.
2. **Preemptive:** Ideally for Tcpu time. The process leaves the CPU if an I/O instruction or interrupt occurs **or** if a new process with a lower Tcpu arrives.

Advantages of SJF

1. Shorter processes run first, reducing the average wait time.
2. Average turnaround time reduces, and throughput increases.

Disadvantages of SJF

1. **Prediction** of CPU burst time is not possible. This is a **theoretical algorithm**, making it impractical.
2. **Starvation** of larger processes.

Example Calculation 1:

Processes:

Process	Arrival Time	Tcpu
P1	0	5
P2	1	2
P3	4	1

Using Preemptive SJF:

Formula: Wt = (start time - arrival time) + sum of (resume – pre-empt)

- Wt1 = (0-0) + (3-1) + (5-4) = 0 + 2 + 1 = **3**
- Wt2 = (1-1) = **0**
- Wt3 = (4-4) = **0**

Avg Wt: (3 + 0 + 0) / 3 = **1**

Turnaround Time: Formula: Ta = Tcpu + Wt

- Ta1 = 5 + 3 = **8**
- Ta2 = 2 + 0 = **2**
- Ta3 = 1 + 0 = **1**

Avg Ta: (8 + 2 + 1) / 3 = **3.67**

Example Calculation 2:

Process	Arrival Time	Tcpu
P1	0	5
P2	1	2
P3	4	1

Using Non-Preemptive SJF:

Formula: Wt = (start time - arrival time)

- Wt1 = 0 - 0 = **0**
- Wt2 = 6 - 1 = **5**
- Wt3 = 5 - 4 = **1**

Avg Wt: (0 + 5 + 1) / 3 = **2**

Turnaround Time: Formula: Ta = Tcpu + Wt

- Ta1 = 5 + 0 = **5**
- Ta2 = 2 + 5 = **7**
- Ta3 = 1 + 1 = **2**

Avg Ta: (5 + 7 + 2) / 3 = **4.67**

Priority Scheduling

Definition:

Select the process from the ready queue that has the highest priority.

Preemptive Priority

- A process with the highest priority from the ready queue is selected to use the CPU.
- **Higher Priority Arrival:** Current process is moved to the ready queue and the new process runs.

Non-Preemptive Priority

- A process with the highest priority from the ready queue is selected to use the CPU.
- Regardless of priority, the current process **continues** until completion.

Advantages

- Priority is considered, making it practically possible.

Disadvantages

- **Starvation** of lower-priority processes.
- Can be solved by **promoting** starving processes if they wait beyond a threshold time.

Example Calculation:

Process	Arrival Time	Tcpu	Priority
P1	0	5	3
P2	2	6	8
P3	3	3	4

Using Preemptive Priority: Formula: Wt = (start time - arrival time) + sum of (resume - preempt)

- Wt1 = (0 - 0) + (11 - 2) = **9**
- Wt2 = (2 - 2) = **0**
- Wt3 = (8 - 3) = **5**

Avg Wt: (9 + 0 + 5) / 3 = **4.67**

Turnaround Time:

- Ta1 = 5 + 9 = **14**
- Ta2 = 6 + 0 = **6**
- Ta3 = 3 + 5 = **8**

Avg Ta: (14 + 6 + 8) / 3 = **9.33**

Using Non-Preemptive Priority: Formula: Wt = (start time - arrival time)

- Wt1 = 0 - 0 = **0**
- Wt2 = 5 - 2 = **3**
- Wt3 = 11 - 3 = **8**

Avg Wt: (0 + 3 + 8) / 3 = **3.67**

Turnaround Time:

- Ta1 = 5 + 0 = **5**
- Ta2 = 6 + 3 = **9**
- Ta3 = 3 + 8 = **11**

Avg Ta: (5 + 9 + 11) / 3 = **8.33**

Round Robin Scheduling (RR)

Definition: Select the process that is in the front of the ready queue (similar to FIFO).

How Long Does the Process Get the CPU?

- Each process gets the CPU for a specific **time slice**.
- After the time slice, a timer interrupt occurs, and the process returns to the ready queue.

Advantages

1. No starvation.
2. Multitasking effect.
3. Improved response time.

Disadvantages

1. Increased wait time for each process.
2. Increased turnaround time.
3. Poor throughput.
4. Context switching overhead on the kernel.

Example Calculation:

Processes:

Process	Tcpu (ms)	Arrival Time
P1	5	0
P2	6	1
P3	8	2

Using RR (Time Slice = 2ms):

Formula: Wt = (start - arrival) + sum of (resume - interrupted)

- Wt1 = (0 - 0) + (6 - 2) + (12 - 8) = 0 + 4 + 4 = **8**
- Wt2 = (2 - 1) + (8 - 4) + (13 - 10) = 1 + 4 + 3 = **8**
- Wt3 = (4 - 2) + (10 - 6) + (15 - 12) = 2 + 4 + 3 = **9**

Avg Wt: (8 + 8 + 9) / 3 = **8.33**

Turnaround Time:

- Ta1 = 5 + 8 = **13**
- Ta2 = 6 + 8 = **14**
- Ta3 = 8 + 9 = **17**

Avg Ta: (13 + 14 + 17) / 3 = **14.67**

HW - Scheduling Algorithms

Process Details:

Process	Arrival Time	Tcpu	Priority
P1	0	5	3
P2	1	2	5
P3	3	3	7

Tasks:

1. **Calculate Average Waiting Time (WT) and Average Turnaround Time (TAT)** using:
    - FCFS
    - SJF Non-Preemptive
    - SJF Preemptive
    - Priority Non-Preemptive (1 is lowest, 10 is highest)
    - Priority Preemptive
    - RR (time slice = 2ms)

System Calls

Definition:

Kernel uses system calls to interface between Application Programs and the System.

Application Program

|

|

System Call (call to the kernel function)

|

|

Perform the Task and return the result to the Application process

---

My program has printf("hello")

|

|

Converted to write() system call

|

|

Kernal gets the call

|

|

Use the Device Drivers and send the string to **monitor**

Example:

Program:

```
printf("hello");
```

Converted to:

```
write(); // System Call
```

Process:

1. Kernel gets the call.
2. Uses the Device Drivers.
3. Sends the string to the monitor.

Different Operating Systems, Different System Calls.

Linux Commands:

Process Management:

- **ps command:** Shows the processes for the current bash (CLI), terminal.
- **ps -e:** Shows the process status for the entire system.

Example:

1. Write a C program that runs an infinite loop.

```
#include<stdio.h>
int main() {
   while(1) {
      // Infinite Loop
   }
   return 0;
}
```

Compile and run the program:

gcc pidex.c
./a.out

```
[group035@localhost Day_1]$ vi forkex.c
[group035@localhost Day_1]$ gcc -o forkex forkex.c
[group035@localhost Day_1]$ ./forkex
```

2. **From another terminal, find the PID of the process using ps -e.**

```
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
```

```
 1785 ?        00:00:00 sshd
 1829 ?        00:00:00 sshd
 1832 pts/9    00:00:00 bash
 1859 ?        00:00:00 abrt-dbus
 1882 pts/12   00:00:00 vim
 1915 ?        00:00:00 boltd
 1918 ?        00:00:01 wpa_supplicant
 1920 ?        00:00:06 packagekitd
 1959 pts/5    00:00:00 vim
 1968 pts/6    00:00:00 nano
 1976 ?        00:00:00 sleep
 1990 pts/17   00:00:02 forkex
 1992 pts/35   00:00:00 ps
 2010 ?        00:00:00 colord
 2018 ?        00:01:15 pcscd
 2448 ?        00:00:01 sshd
 2558 ?        00:00:01 sshd
 2560 pts/6    00:00:00 bash
 3522 ?        00:00:01 gdm-session-wor
 3586 ?        00:00:00 gnome-keyring-d
 3592 ?        00:00:01 gnome-session-b
 3615 ?        00:00:00 dbus-launch
 3621 ?        00:00:01 dbus-daemon
 3648 ?        00:00:00 imsettings-daem
 3652 ?        00:00:00 gvfsd
 3658 ?        00:00:00 gvfsd-fuse
```

3. **Kill the process using:**

kill pid_of_the_process

```
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720
pid=1990, ppid=61720Terminated
[group035@localhost Day_1]$
[group035@localhost Day_1]$
```

```
62784 pts/27   00:00:00 bash
62940 ?        00:00:00 sshd
62962 ?        00:00:00 sshd
62966 pts/37   00:00:00 bash
63135 ?        00:00:00 sshd
63156 ?        00:00:00 sshd
63161 pts/38   00:00:00 bash
63282 ?        00:00:00 kworker/u128:1
63454 ?        00:00:00 sshd
63502 ?        00:00:00 sshd
63505 pts/14   00:00:00 bash
63635 ?        00:00:00 sshd
63650 ?        00:00:00 sshd
63652 pts/16   00:00:00 bash
63866 ?        00:00:00 sshd
63884 pts/25   00:00:00 cat
63888 ?        00:00:00 sshd
63890 pts/31   00:00:00 bash
64640 ?        00:00:00 kworker/u128:0
65256 pts/17   00:00:47 forkex
65395 ?        00:00:02 kworker/0:1
65428 ?        00:00:00 sshd
65457 ?        00:00:00 sshd
65461 pts/7    00:00:00 bash
[group035@localhost ~]$ kill 1990
[group035@localhost ~]$
```

System Calls:

- **getpid:** A Linux system call to get the PID of the current process.

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

int main() {
   printf("PID: %d\n", getpid());
   return 0;
}
```

- **getppid**: A Linux system call to get the parent process ID of the current process.

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

int main() {
   printf("Parent PID: %d\n", getppid());
   return 0;
}
```

ps -ef: Shows full details of processes for the entire system.

Note:

Every Linux system has a **ppid** for each process, except for process **0**.

Fork System Call:

Definition: The fork system call is used by the parent process to create a child process.

Orphan Process:

- A child process whose parent terminated before the child.
- It is adopted by process 1.

```
[group035@localhost Day_1]$ vi forkex.c
[group035@localhost Day_1]$ gcc -o forkex forkex.c
[group035@localhost Day_1]$ ./forkex
pid=11437, ppid=61720
[group035@localhost Day_1]$ pid=11438, ppid=1
```

Solving Orphan Problem

```
[group035@localhost Day_1]$ vi forkex.c
[group035@localhost Day_1]$ gcc -o forkex forkex.c
[group035@localhost Day_1]$ ./forkex
pid=11397, ppid=61720
pid=11398, ppid=11397
^C
[group035@localhost Day_1]$
```

Shell Scripts:

Definition: Programs written using bash shell commands. These are interpreted programs and typically have a **.sh** extension (not mandatory).

LAB

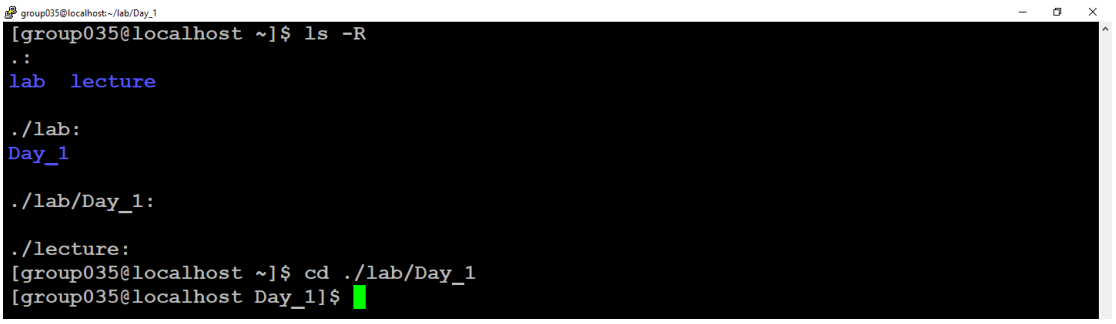
Examples:

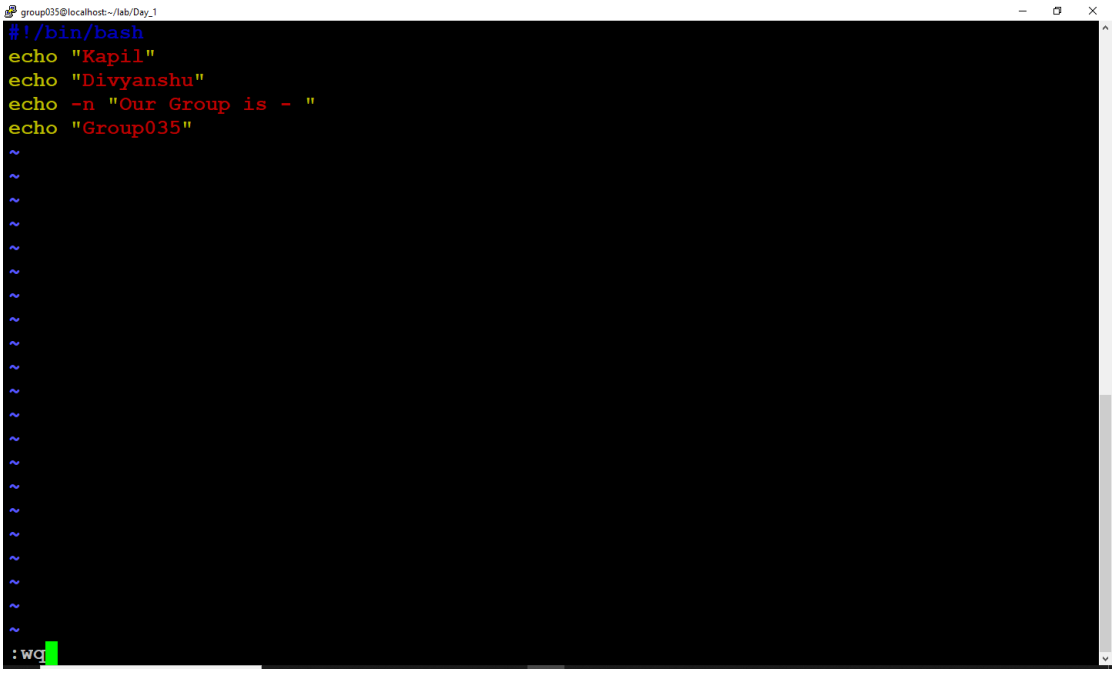
1. Write a script to print "hello":

```
#!/bin/bash
echo "hello"
```

Run the script:

```
bash first.sh
```

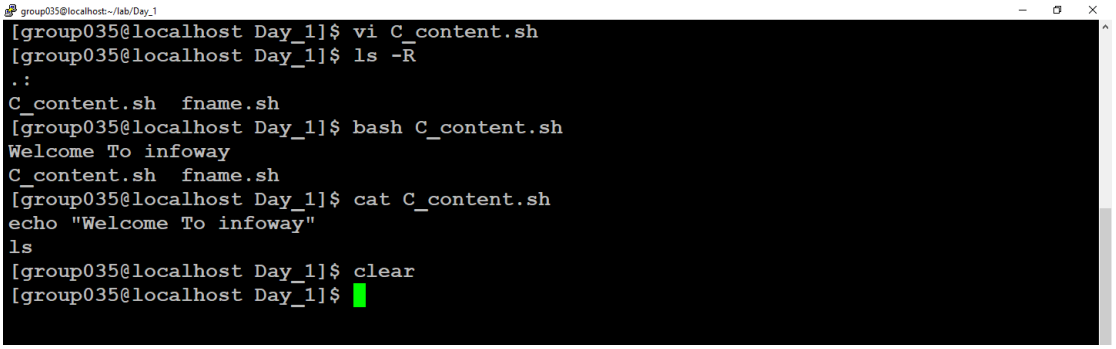


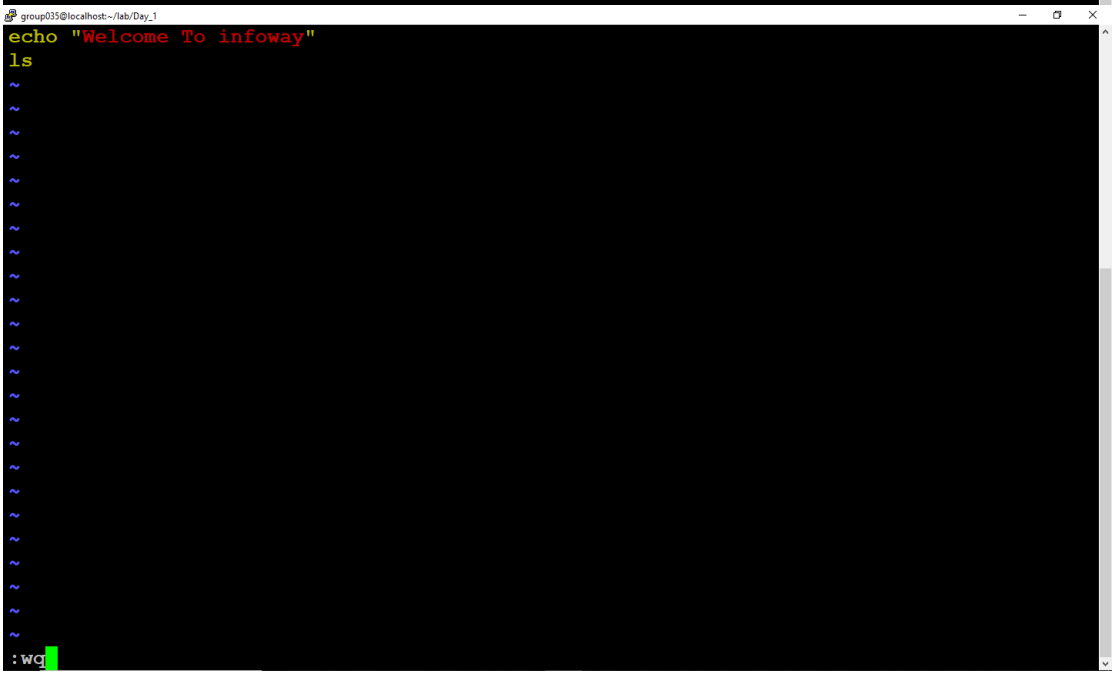


---

2. Show contents of the current folder:

```
#!/bin/bash
echo "The contents are:"
ls
```

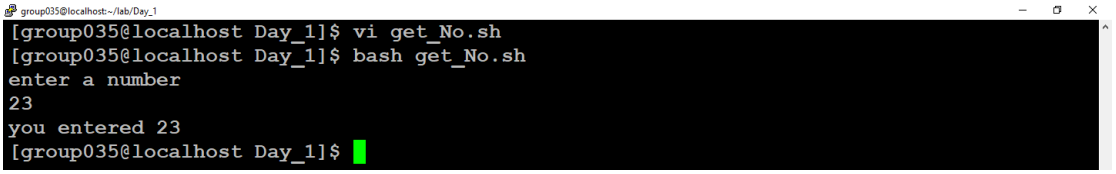




---

3. Get a number from the user and print it:

```
#!/bin/bash
echo "Enter a number:"
read num
echo "You entered $num"
```



```
group035@localhost:~/lab/Day_1
echo "enter a number"
read num

echo "you entered $num "
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

---

4. Create a 0kb file in the current folder:

```
#!/bin/bash
echo "Enter a file name:"
read fname
touch ./$fname
echo "Check if file is created:"
ls -l
```

```
[group035@localhost Day_1]$ vi get_file_name.sh
[group035@localhost Day_1]$ bash get_file_name.sh
enter a file name
OS
CHECK if file is created :-
total 16
-rw-rw-r--. 1 group035 group035 29 Dec 11 15:22 C_content.sh
-rw-rw-r--. 1 group035 group035 84 Dec 11 15:14 fname.sh
-rw-rw-r--. 1 group035 group035 94 Dec 11 15:29 get_file_name.sh
-rw-rw-r--. 1 group035 group035 57 Dec 11 15:27 get_No.sh
-rw-rw-r--. 1 group035 group035  0 Dec 11 15:30 OS
[group035@localhost Day_1]$
```

```
group035@localhost:~/lab/Day_1
echo "enter a file name"
read fname

touch ./$fname

echo "CHECK if file is created :-"
ls -l
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

---

5. Create nested folders:

```
#!/bin/bash
echo "Enter 2 names:"
read first second
mkdir ./$first
mkdir ./$first/$second
ls -R
```

```
group035@localhost:~/lab/Day_1
[group035@localhost ~]$ cd ./lab/Day_1
[group035@localhost Day_1]$ vi C_folder.sh
[group035@localhost Day_1]$ bash C_folder.sh
enter 2 names
Parent Child
.:
alpha  C_content.sh  C_folder.sh  fname.sh  get_file_name.sh  get_No.sh  OS  Parent

./alpha:
beta

./alpha/beta:

./Parent:
Child

./Parent/Child:
[group035@localhost Day_1]$
```

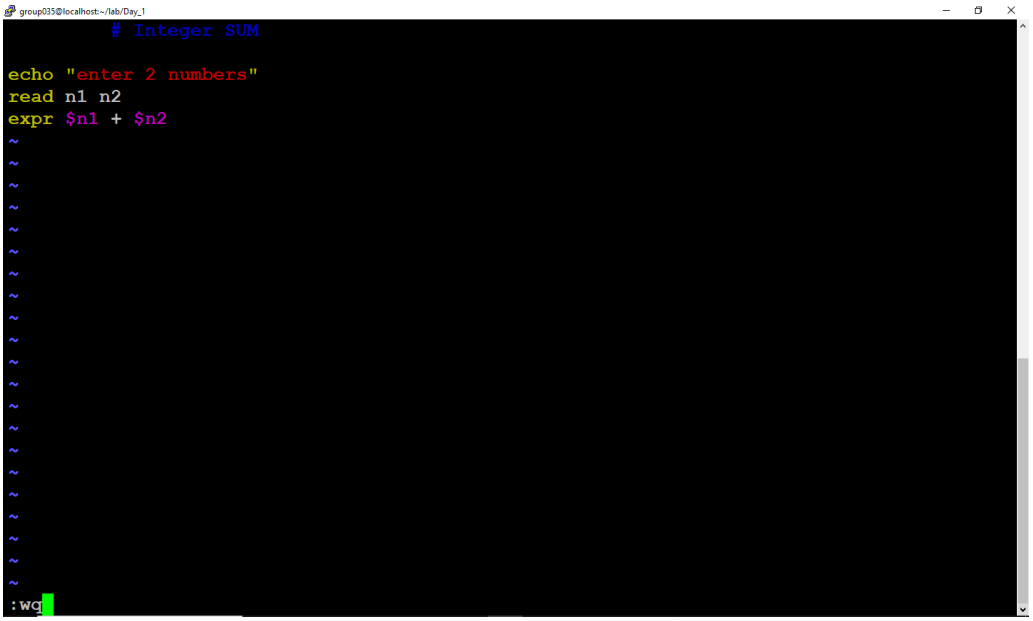
```
group035@localhost:~/lab/Day_1
echo "enter 2 names"
 read first second

mkdir ./$first

mkdir ./$first/"$second"

ls -R
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

---

6. Accept two numbers and show their sum:

```
#!/bin/bash
echo "Enter 2 numbers:"
read n1 n2
sum=$((n1 + n2))
echo "Sum = $sum"
```

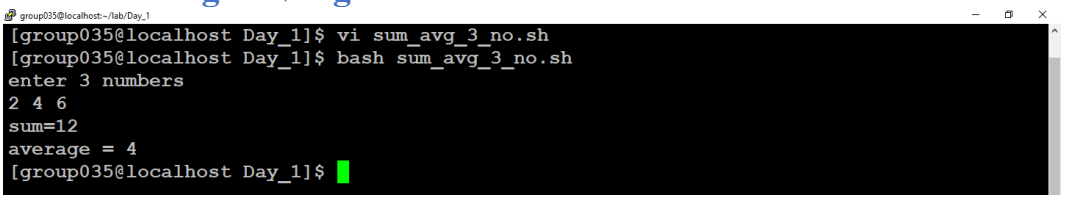
```
group035@localhost:~/lab/Day_1
[group035@localhost Day_1]$ vi sum_2_no.sh
[group035@localhost Day_1]$ bash sum_2_no.sh
enter 2 numbers
2 4
6
```

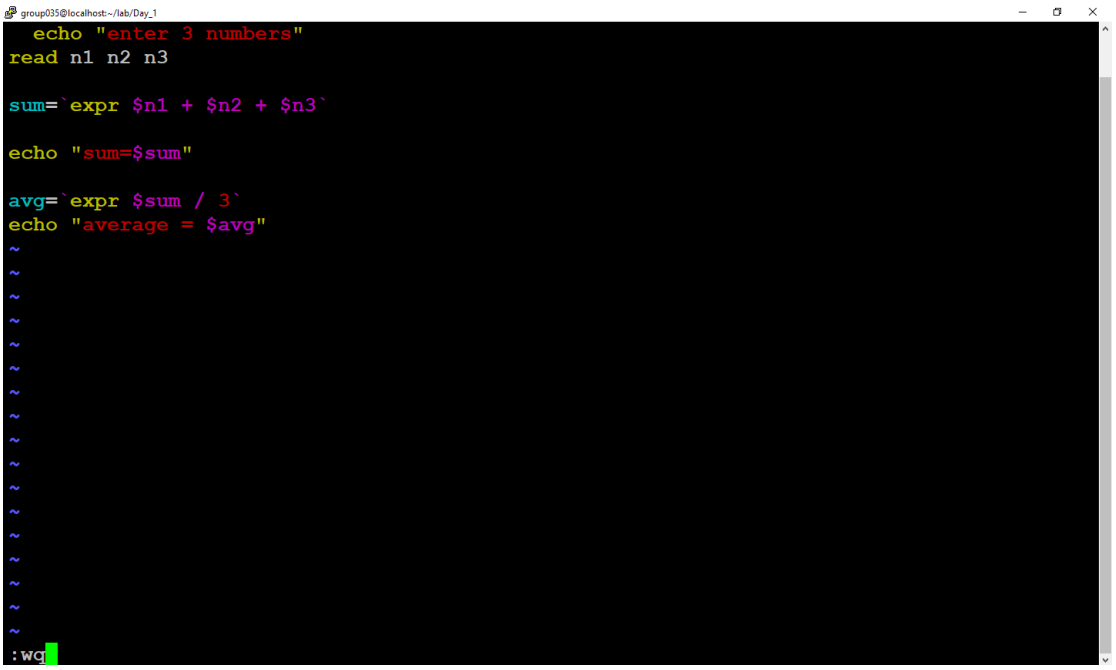


---

7. Accept three numbers and show their sum and average:

```
#!/bin/bash
echo "Enter 3 numbers:"
read n1 n2 n3
sum=$((n1 + n2 + n3))
avg=$((sum / 3))
echo "Sum = $sum"
echo "Average = $avg"
```



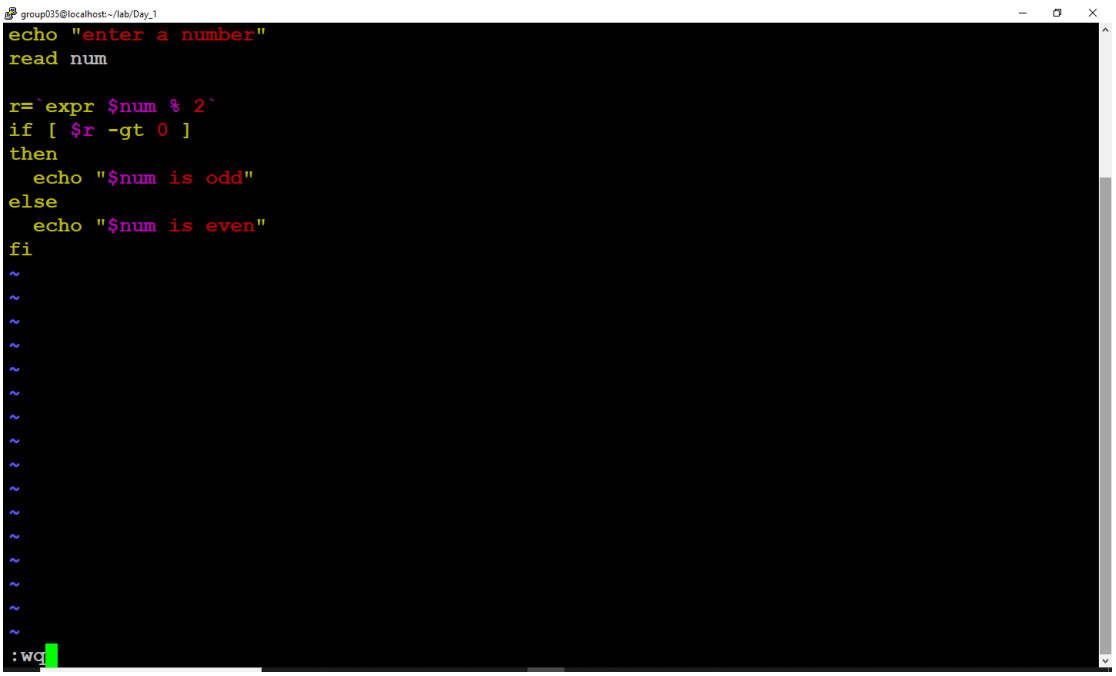


---

8. Check if a number is odd or even:

```
#!/bin/bash
echo "Enter a number:"
read num
if ((num % 2 == 0)); then
   echo "$num is even"
else
   echo "$num is odd"
fi
```

```
[group035@localhost Day_1]$ vi even_odd.sh
[group035@localhost Day_1]$ bash even_odd.sh
enter a number
3
3 is odd
[group035@localhost Day_1]$ bash even_odd.sh
enter a number
4
4 is even
[group035@localhost Day_1]$
```



---

9. Compare a number with 100:

```
#!/bin/bash
echo "Enter a number:"
read num
if [ $num -gt 100 ]; then
   echo "$num is greater than 100"
elif [ $num -lt 100 ]; then
   echo "$num is less than 100"
else
   echo "$num is 100"
fi
```

```
[group035@localhost Day_1]$ vi gt_lt_eq.sh
[group035@localhost Day_1]$ bash gt_lt_eq.sh
enter a number
45
45 is less than 100
[group035@localhost Day_1]$ bash gt_lt_eq.sh
enter a number
101
101 is greater than 100
[group035@localhost Day_1]$
```

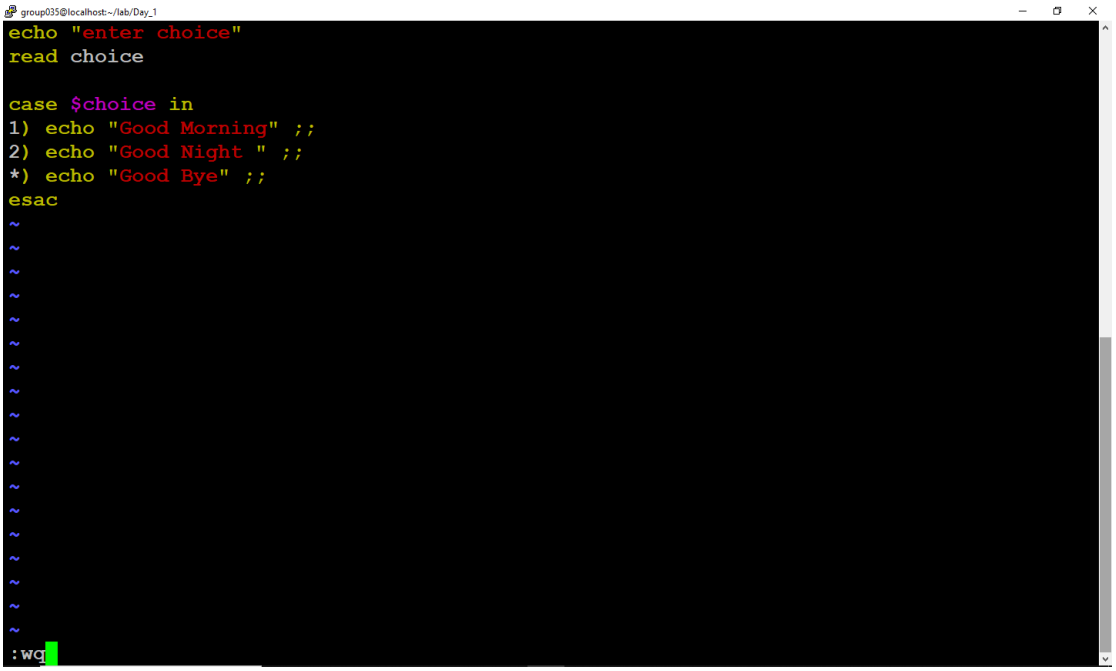
```
group035@localhost:~/lab/Day_1
echo "enter a number"
read num1

if [ $num1 -gt 100 ];then
   echo "$num1 is greater than 100"
elif [ $num1 -lt 100 ];then
   echo "$num1 is less than 100"
else
  echo "num1 is 100"
fi
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

---

10. Use switch-case to print messages based on user input:

```
#!/bin/bash
echo "Enter choice:"
read choice
case $choice in
   1) echo "Good Morning" ;;
   2) echo "Good Night" ;;
   *) echo "Good Bye" ;;
Esac
```

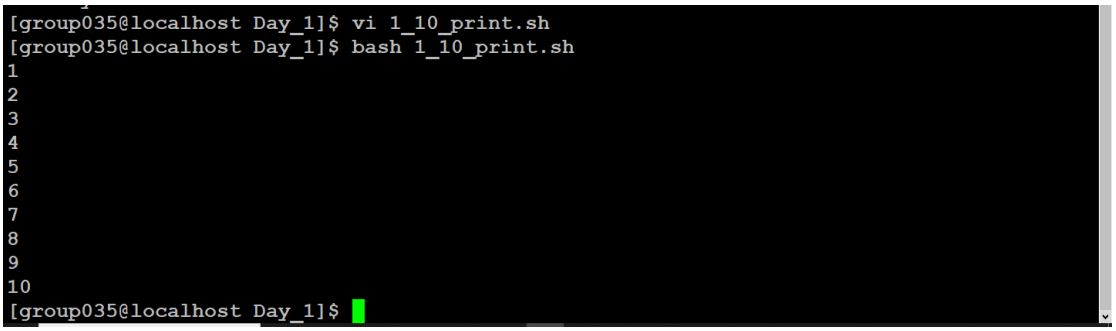
```
[group035@localhost Day_1]$ vi switch_case.sh
[group035@localhost Day_1]$ bash switch_case.sh
enter choice
1
Good Morning
[group035@localhost Day_1]$ bash switch_case.sh
enter choice
2
Good Night
[group035@localhost Day_1]$ bash switch_case.sh
enter choice
3
Good Bye
[group035@localhost Day_1]$ bash switch_case.sh
enter choice
4
Good Bye
[group035@localhost Day_1]$
```



---

11. Print numbers from 1 to 10 using a while loop:

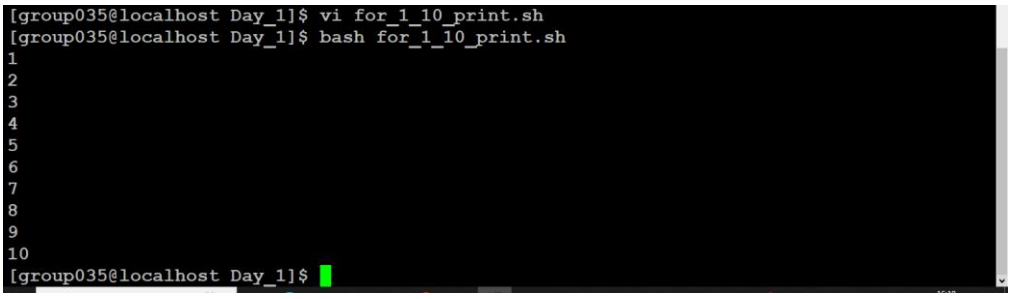
```
#!/bin/bash
num=1
while [ $num -le 10 ]
do
   echo "$num"
   num=$((num + 1))
done
```



---

12. Print numbers from 1 to 10 using a for loop:

```
#!/bin/bash
for ((num=1; num<=10; num++))
do
   echo "$num"
done
```



13. Write a shell script to accept a number and show whether it is prime

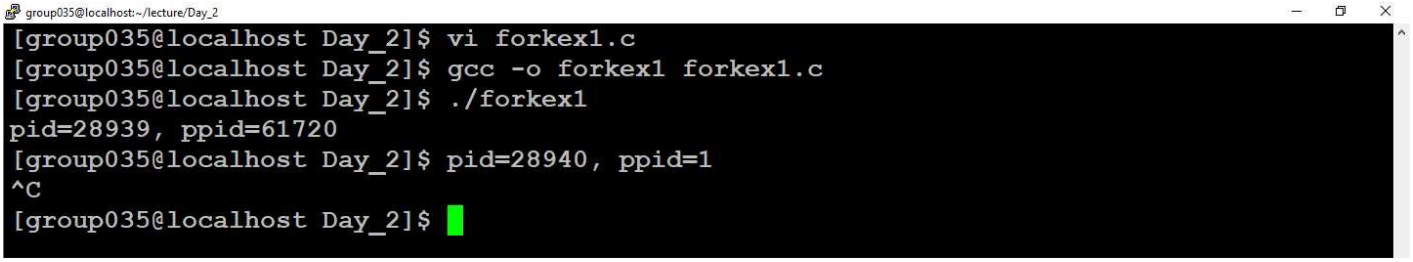
```
[group035@localhost Day_1]$ vi prime_no.sh
[group035@localhost Day_1]$ bash prime_no.sh
Enter a number
7
7 is a prime number
[group035@localhost Day_1]$ bash prime_no.sh
Enter a number
4
4 is not a prime number
[group035@localhost Day_1]$
```

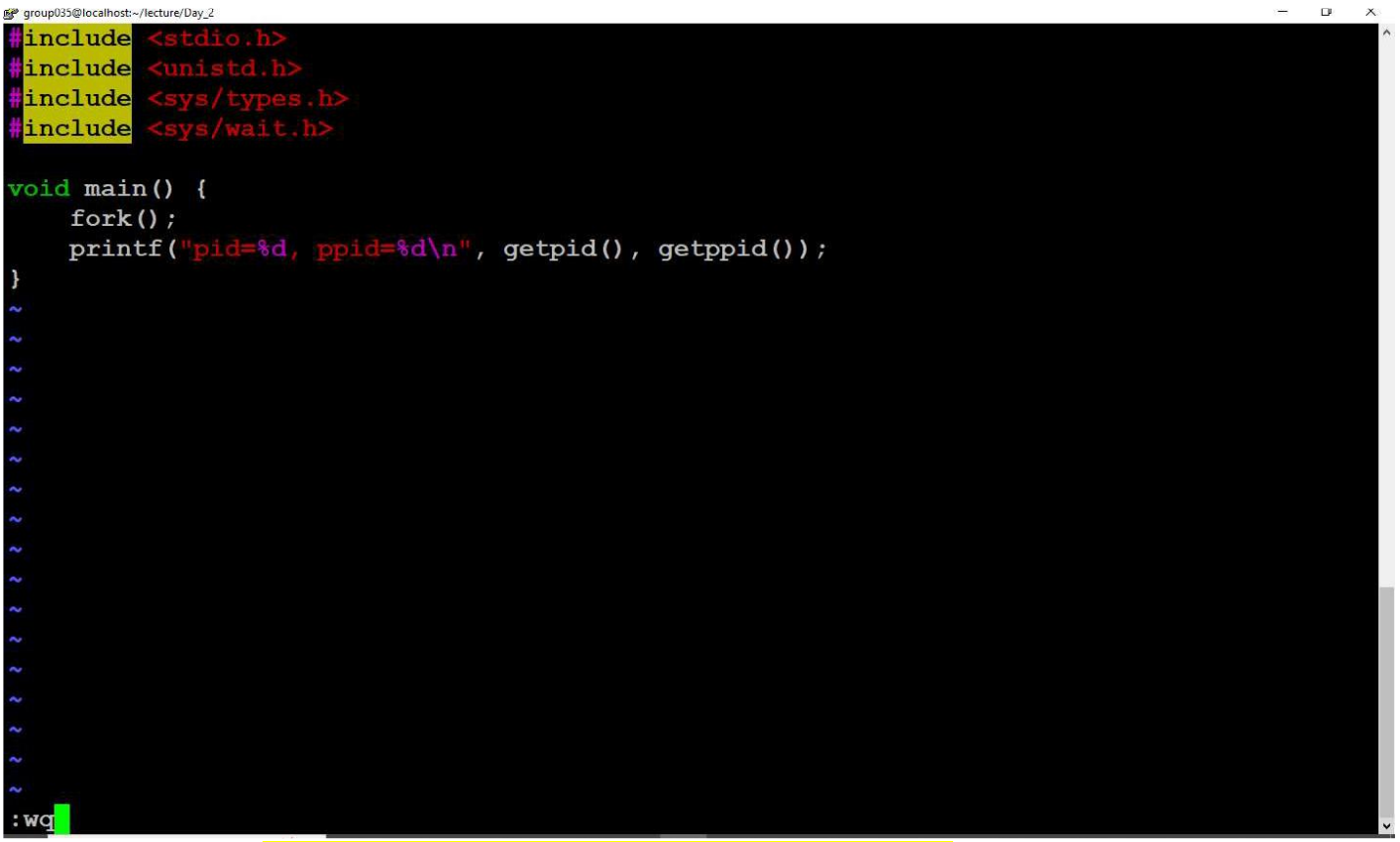
group035@localhost:~/lab/Day_1

```
#!/bin/bash

echo "Enter a number"
read num

is_prime=1

if [ $num -le 1 ]; then
    is_prime=0
else
    for ((i=2; i*i<=num; i++)); do
        if [ $((num % i)) -eq 0 ]; then
            is_prime=0
            break
        fi
    done
fi

if [ $is_prime -eq 1 ]; then
    echo "$num is a prime number"
else
    echo "$num is not a prime number"
fi

~
~
:wq
```

Fork System Call

- **Number of Processes Created**:
    - If consecutive fork calls are present, the number of processes created is **2 raised to the number of consecutive fork calls**.
- **Wait System Call**:
    - **Purpose**: Blocks the parent process, ensuring it does not proceed before the child process terminates.
    - **Advantage**: Ensures the parent process clears up the resources of the terminated child.
    - **Zombie Process**: If the parent terminates before the child, the child becomes a **Zombie Process**. Solution: Use the **wait** system call in the parent process.





Zombie State

```
[group035@localhost Day_2]$ cp forkex1.c forkex2.c
[group035@localhost Day_2]$ vi forkex2.c
[group035@localhost Day_2]$ gcc -o forkex2 forkex2.c
[group035@localhost Day_2]$ ./forkex2
pid=37912, ppid=61720
[group035@localhost Day_2]$ pid=37916, ppid=1
pid=37914, ppid=1
pid=37915, ppid=1
pid=37918, ppid=1
pid=37919, ppid=1
pid=37920, ppid=1
pid=37922, ppid=1

[group035@localhost Day_2]$
```

group035@localhost:~/lecture/Day_2

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

void main() {
    fork();
    fork();
    fork();
    printf("pid=%d, ppid=%d\n", getpid(), getppid());
}
~
~
~
~
~
~
~
~
~
~
~
~
~
~
:wq
```

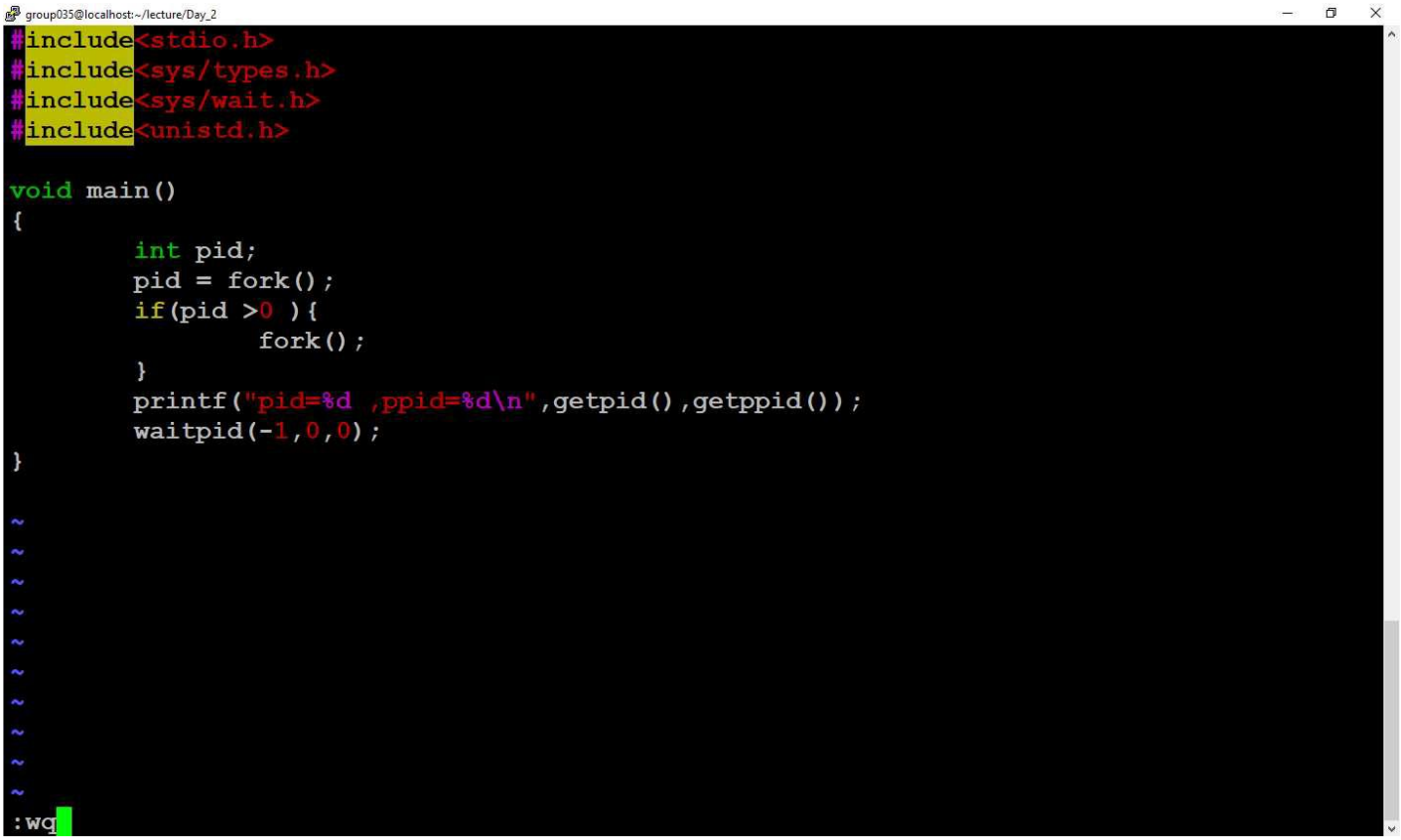
To Overcome Zombie State---

```
[group035@localhost Day_2]$ cp forkex2.c forkex3.c
[group035@localhost Day_2]$ vi forkex3.c
[group035@localhost Day_2]$ gcc -o forkex3 forkex3.c
[group035@localhost Day_2]$ ./forkex3
pid=38496, ppid=61720
pid=38498, ppid=38496
[group035@localhost Day_2]$ pid=38497, ppid=1
pid=38499, ppid=38497
```

Programs to Understand Process Creation

Program 1:

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

void main() {
   int pid;
   pid = fork();
   if(pid > 0) {
      fork();
   }
   printf("pid=%d ,ppid=%d\n", getpid(), getppid());
   waitpid(-1, 0, 0);
}
```



Program 2:

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

void main() {
   int pid;
   pid = fork();
   fork();
   printf("pid=%d ,ppid=%d\n", getpid(), getppid());
   waitpid(-1, 0, 0);
}
```

Program 3:

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

void main() {
   int pid1, pid2;
   pid1 = fork();
   pid2 = fork();
   if (pid1 > 0 || pid2 > 0) {
      fork();
   }
   printf("pid=%d ,ppid=%d\n", getpid(), getppid());
   waitpid(-1, 0, 0);
}
```

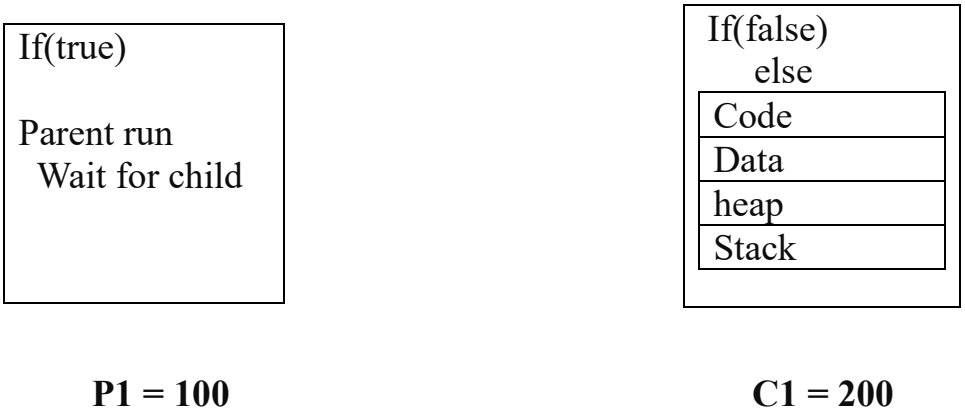
- For each program, calculate the number of processes created.

System Calls

1. **fork**: Creates a child process.
2. **getpid**: Gets the process ID of the current process.
3. **getppid**: Gets the parent process ID.
4. **waitpid**: Waits for a specific child process to terminate.
5. **exec**: Replaces the current process image with a new process image.

Example Program Using exec():

```
#include <unistd.h>

int main(void) {
   char *programName = "./a.out";
   int returnval;

   returnval = fork();
   if (returnval > 0) { // Parent process
      printf("Parent runs\n");
      waitpid(returnval, 0, 0);
   } else { // Child process
      execlp(programName, programName, NULL);
   }
   return 0;
}
```



Memory Management

Basic Concepts:

- **Memory = RAM = Primary Memory = Main Memory (Volatile)**
- **Actual Address of Instruction or Data in RAM**:
  - Actual Address = Base Address + Offset Address
  - Example: If Base Address = 3 and Offset = 1, Actual Address = 4

Schedulers:

1. **Short-Term Scheduler**:
   - **Resource**: CPU
   - **Selection**: Process in the Ready Queue
2. **High-Level Scheduler (Long-Term Scheduler)**:
   - **Resource**: Memory Space
   - **Selection**: Process that the user wants to execute before the process life cycle starts.

Partitioning Schemes:

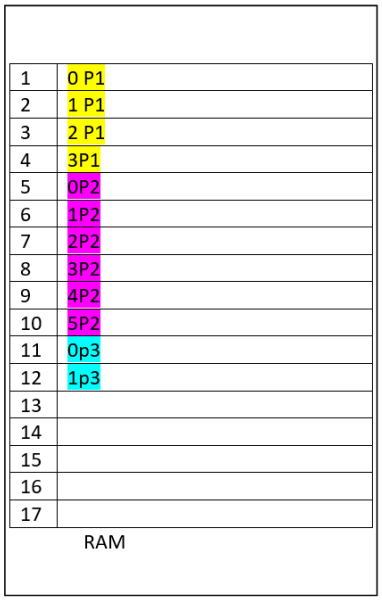
1. Variable Partition Scheme:

- **Characteristics**:
  - Sizes and number of partitions are not fixed.
  - Partitions are created or destroyed as processes are loaded into RAM.
- **Allocation Techniques**:
  1. **Best Fit**: Allocate the free hole closest to the process size.

2. **First Fit**: Allocate the first free hole that is ≥ process size.
3. **Worst Fit**: Allocate the free hole where the difference between process size and free hole size is maximum.

- **Problem**: External Fragmentation (Space is free, but not consecutive).
  - **Solution**:
    - **Theoretical**: Compaction (Shift processes to one end of RAM).
    - **Practical**: Paging.

1. **LTS – yes—P1(4), P2(6), P3(2)**



1	0 P1
2	1 P1
3	2 P1
4	3P1
5	0P2
6	1P2
7	2P2
8	3P2
9	4P2
10	5P2
11	0p3
12	1p3
13	
14	
15	
16	
17	

RAM

P1

0	I1
1	I2
2	I3
3	I4

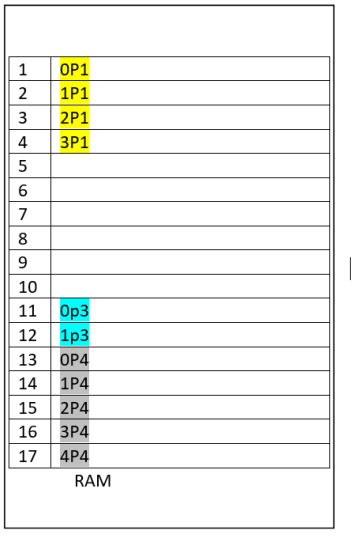
P2

0	I1
1	I2
2	I3
3	I4
4	I5
5	I6

P3

0	I1
1	I2

2. **LTS – yes—P1(4), P2(6) FREE, P3(2)**
   LTS – yes – P4(5)



1	0P1
2	1P1
3	2P1
4	3P1
5	
6	
7	
8	
9	
10	
11	0p3
12	1p3
13	0P4
14	1P4
15	2P4
16	3P4
17	4P4

RAM

P1

0	I1
1	I2
2	I3
3	I4

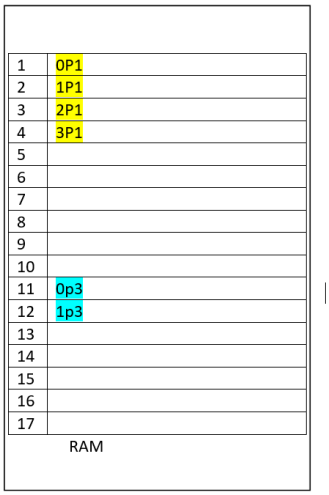
P3

0	I1
1	I2

P4

0	I1
1	I2
2	I3
3	I4
4	I5

3. **LTS – yes—P1(4), P2(6) FREE, P3(2), P4(5) FREE**



1	OP1
2	1P1
3	2P1
4	3P1
5	
6	
7	
8	
9	
10	
11	Op3
12	1p3
13	
14	
15	
16	
17	

RAM

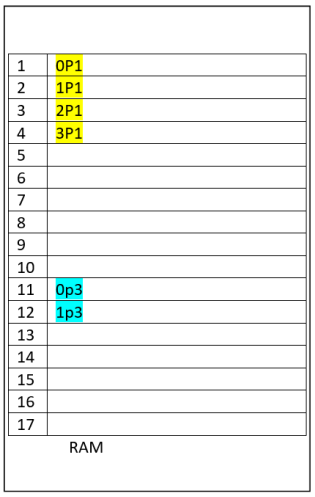
P1

0	I1
1	I2
2	I3
3	I4

P3

0	I1
1	I2

4. **LTS – yes—P1(4), P2(6) FREE, P3(2), P4(5) FREE**
   LTS – NO– P4(5)P5(8) [Don't have continuous Space]



1	OP1
2	1P1
3	2P1
4	3P1
5	
6	
7	
8	
9	
10	
11	Op3
12	1p3
13	
14	
15	
16	
17	

RAM

P1

0	I1
1	I2
2	I3
3	I4

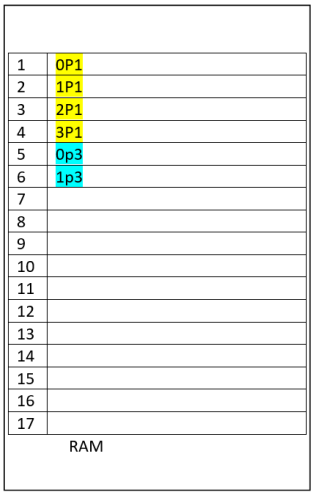
P3

0	I1
1	I2

P5

0	I1
1	I2
2	I3
3	I4
4	I5
5	I6
6	I7
7	I8

Solution – (Theoretical - Compaction (Shift processes to one end of RAM))



1	OP1
2	1P1
3	2P1
4	3P1
5	Op3
6	1p3
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	

RAM

P1

0	I1
1	I2
2	I3
3	I4

P3

0	I1
1	I2

P5

0	I1
1	I2
2	I3
3	I4
4	I5
5	I6
6	I7
7	I8

2. Fixed Partition Scheme:

- **Characteristics**:
  - Sizes and number of partitions (frames) are fixed.
  - Example: Frame size = 4 KB.

Frame	4kb Size

F0	P1
F1	p1
F2	P2
F3	P2
F4	P2 EMPTY-UNUSED---- WASTE (3kb) = INTERNAL FRAGMENTATION PROBLEM
F5	
F6	

P1	8kb

P2	9kb

- **Problems**:
  1. External Fragmentation: Free frames not consecutive.
  2. Internal Fragmentation: Process size not an exact multiple of frame size.

Segmentation:

- **Process Division**: Based on content (Code Segment, Data Segment, Stack Segment, Heap Segment).
- **Data Structure**: Segment Table.

Segment Number	Segment Base Address	Segment Size

100	Seg 1
210	Seg 2
250	Seg 3
440	Seg 4

RAM - segs stored in non-consecutive locations

Seg 0	0 I1 1i2 2i3 4i4
Seg1	0 d1
Seg2	0 l1 1l2 2l3
Seg3	0a1 1a2

Process internally divided into segment. every seg has a seg offset

Segment Number	Segment Base Address	Segment Size
0	100	5
1	210	1
2	250	3
3	440	2

- **Advantages**:
  - Reduces external fragmentation.
- **Disadvantages**:
  - Does not completely eliminate external fragmentation.
  - Overhead of maintaining the segment table.
- **Actual Address** = Segment Base Address + Segment Offset

File and Folder Permissions

- **Command**: chmod

- **Permissions**:
  - **r**: Read
  - **w**: Write
  - **x**: Execute

Symbolic Representation	r w x	Octal Number
---	0 0 0	0
--x	0 0 1	1
-w-	0 1 0	2
-wx	0 1 1	3
r--	1 0 0	4
r-x	1 0 1	5
rw-	1 1 0	6
rwx	1 1 1	7

Loops

1. Until Loop:

```
num=1
until [ $num -gt 10 ]
do
  echo $num
  ((num=num+1))
done
```

2. Command-Line Arguments:

```
while [ $# -gt 0 ]
do
  echo $1 >> file1
  shift
done
```

Redirection:

Move output to file

- >: Overwrite file.
- >>: Append to file.

```
echo "good night" >> file1
```

Pipes:

PIPES

cmd1 | cmd2 | cmd3

```
echo "enter two nums"
read n1 n2

sum=`echo "$n1 + $n2" | bc`

echo "$sum"
```

bc = basic calculator