**Day 6**: **28th May 2025**

Task 1:

RegEX Symbols in linux , List them down with description

Here are the key RegEx symbols used in Linux:

**Basic Symbols**

* **.** - Matches any single character except newline
* **\*** - Matches zero or more of the preceding character
* **+** - Matches one or more of the preceding character
* **?** - Matches zero or one of the preceding character
* **^** - Matches beginning of line
* **$** - Matches end of line
* **\** - Escape character for literal interpretation

**Character Classes**

* **[abc]** - Matches any character within brackets
* **[^abc]** - Matches any character NOT within brackets
* **[a-z]** - Matches any character in the range
* **[0-9]** - Matches any digit

**Predefined Classes**

* **\d** - Matches any digit (0-9)
* **\D** - Matches any non-digit
* **\w** - Matches word characters (letters, digits, underscore)
* **\W** - Matches non-word characters
* **\s** - Matches whitespace characters
* **\S** - Matches non-whitespace characters

**Quantifiers**

* **{n}** - Matches exactly n occurrences
* **{n,}** - Matches n or more occurrences
* **{n,m}** - Matches between n and m occurrences

**Grouping & Alternation**

* **()** - Groups expressions together
* **|** - OR operator (alternation)
* **\1, \2** - Backreferences to captured groups

**Anchors**

* **\b** - Word boundary
* **\B** - Non-word boundary

These symbols work across Linux tools like grep, sed, awk, and vim.

Task 2:

What are the imp features of Linux os ?

## Key Features of Linux OS

**Open Source**

* Free to use, modify, and distribute
* Source code is publicly available

**Multi-User & Multi-Tasking**

* Multiple users can work simultaneously
* Runs multiple processes concurrently

**Security**

* Strong permission system and user access controls
* Less vulnerable to malware compared to other OS

**Stability & Reliability**

* Rarely crashes or requires reboots
* Can run for months without restart

**Command Line Interface**

* Powerful shell and terminal access
* Extensive command-line tools and utilities

**File System Support**

* Supports multiple file systems (ext4, NTFS, FAT32, etc.)
* Hierarchical directory structure

**Networking**

* Built-in networking capabilities
* Strong server and network administration tools

**Hardware Compatibility**

* Runs on various hardware architectures
* Efficient resource utilization

**Portability**

* Highly portable across different platforms
* Consistent behavior across systems

**Package Management**

* Easy software installation and updates
* Dependency resolution (apt, yum, pacman)

**Customization**

* Highly configurable and customizable
* Multiple desktop environments available

**Performance**

* Lightweight and fast
* Efficient memory and CPU usage

Task 3:

WHAT IS Kernal and can you explain its functions

## What is a Kernel?

The kernel is the **core component of an operating system** that acts as a bridge between applications and the actual hardware of a computer. It's the first program loaded when the system boots and remains in memory throughout the system's operation, managing all system resources and providing essential services to other programs.

## Key Features of Kernel

**Process Management**

* Creates, schedules, and terminates processes
* Handles process synchronization and inter-process communication
* Manages CPU allocation using scheduling algorithms

**Memory Management**

* Allocates and deallocates memory for processes
* Implements virtual memory and paging mechanisms
* Prevents memory conflicts between processes

**Device Management**

* Controls and coordinates hardware devices through device drivers
* Provides uniform interface for applications to access hardware
* Manages I/O operations and device queues

**File System Management**

* Organizes and manages files on storage devices
* Provides file access methods and maintains file metadata
* Handles file permissions and security

**System Calls Interface**

* Provides controlled access to kernel services for user applications
* Acts as a gateway between user space and kernel space
* Ensures system security and stability

**Interrupt Handling**

* Responds to hardware and software interrupts
* Manages system events and exceptions
* Coordinates asynchronous operations

**Security and Protection**

* Enforces access controls and user privileges
* Isolates processes from each other
* Protects critical system resources

The kernel operates in **privileged mode** (kernel space) with full hardware access, while user applications run in **user mode** (user space) with restricted access, ensuring system stability and security.

The kernel is essentially the heart of any operating system - it's what we call the core program that manages everything happening in our system. When we boot up a machine, the kernel is the first thing that loads into memory and stays there throughout the entire session. What it does is act as that critical bridge between our applications and the actual hardware components.

In our day-to-day operations, we rely on the kernel for several key functions. It handles all our process management - so when we're running multiple applications simultaneously, the kernel is what's scheduling CPU time and managing those processes. We also depend on it for memory management, where it's allocating RAM to different processes and implementing virtual memory to optimize performance.

From a system administration perspective, we see the kernel managing all our hardware devices through drivers - whether we're accessing storage, network interfaces, or any peripheral devices. It also provides us with that secure interface through system calls, which is how our user applications can safely request kernel services without compromising system stability.

What makes the kernel particularly important is that it operates in privileged mode with full hardware access, while keeping our user applications in a protected user space. This separation is what gives us system stability and security - the kernel ensures that processes can't interfere with each other or access critical system resources inappropriately.

Task 4:

What is BASH? Full form with explanation.

## BASH - Bourne Again Shell

**Full Form:** BASH stands for **Bourne Again Shell**

## What is BASH?

BASH is a **command-line interpreter and scripting language** that serves as the default shell for most Linux distributions and macOS systems. It's an enhanced version of the original Bourne Shell (sh) developed by Stephen Bourne.

## Key Features & Explanation

**Command Interpreter**

* Processes and executes commands entered by users
* Interprets both interactive commands and shell scripts
* Provides interface between user and operating system

**Scripting Capabilities**

* Supports variables, loops, conditionals, and functions
* Enables automation through shell scripts
* Handles complex programming constructs

**Job Control**

* Manages foreground and background processes
* Supports process suspension and resumption
* Handles multiple tasks simultaneously

**History & Editing**

* Maintains command history for easy recall
* Provides command-line editing capabilities
* Supports tab completion for files and commands

**I/O Redirection & Piping**

* Redirects input/output to files or other commands
* Connects multiple commands through pipes
* Handles complex data flow operations

**Built-in Commands**

* Includes essential commands like cd, echo, read
* Provides file manipulation and system information commands
* Supports alias creation for command shortcuts

**Environment Management**

* Manages environment variables and system paths
* Handles user sessions and configurations
* Controls system and user-specific settings

BASH is essentially the **gateway to the Linux system**, allowing users to interact with the operating system through text commands and automated scripts.

**Task 5:**

What is the diffrenece between window and linux

Here's how you can explain this in an interview:

In our development environment, we work with both Windows and Linux systems regularly. The main difference we see is that Windows uses a graphical user interface by default and runs on the NT kernel, while Linux is open-source and typically command-line focused with various distributions like Ubuntu or CentOS. We find Linux more stable for server deployments and development work since it's Unix-based, whereas Windows integrates better with Microsoft enterprise tools. From a security perspective, we've noticed Linux requires fewer updates and has better permission controls, but Windows offers broader software compatibility for end-users. The choice usually depends on our project requirements and the existing infrastructure we're working with.

**Task 6: Basic Components of Linux**

In our Linux implementations, we work with four core components daily. The kernel is the heart of the system that manages hardware resources and system calls. We interact with the shell, which is our command-line interface for executing commands and scripts. The file system organizes our data hierarchically, starting from the root directory, and we use various types like ext4 or XFS depending on requirements. Finally, we rely on system utilities and applications that provide the actual functionality users need, from basic commands like ls and grep to complex applications we deploy.

**Task 7: Legal to Edit Kernel**

Yes, it's completely legal to edit the Linux kernel since it's released under the GPL license, which gives us the freedom to modify and redistribute the source code. We've actually customized kernels for specific hardware requirements in our projects, adding device drivers or optimizing performance parameters. The GPL ensures that any modifications we make must also be open-source if we distribute them. This is one of the key advantages we leverage in Linux - we can tailor the operating system to our exact needs rather than being limited by proprietary restrictions.

**Task 8: LILO Explanation**

LILO, or Linux Loader, is a boot loader we used in older Linux systems to start the operating system. We'd configure it in the /etc/lilo.conf file to specify which kernel to load and boot parameters. Unlike GRUB which we use now, LILO required running the lilo command after any configuration changes to write the boot information to the Master Boot Record. We've mostly migrated away from LILO to GRUB2 because it's more flexible and doesn't need reinstallation after kernel updates. However, understanding LILO helps us troubleshoot legacy systems we still maintain.

**Task 9: Shell and Types**

The shell is our command interpreter that processes the commands we type and communicates with the kernel to execute them. We work with several shell types regularly - Bash is our default and most common shell, offering excellent scripting capabilities and command history. We also use Zsh for its advanced features like better auto-completion, and occasionally Fish for its user-friendly syntax highlighting. For lightweight systems or scripts, we sometimes use Dash or the original Bourne shell. Each shell has its strengths, but Bash remains our go-to for most automation and daily tasks because of its widespread compatibility.

**Task 10: Swap Space**

Swap space is our virtual memory extension that we configure when physical RAM gets full. We typically create it as a dedicated partition or swap file, and the system automatically moves inactive memory pages there to free up RAM for active processes. In our server setups, we usually allocate swap space equal to or double the RAM size, depending on the workload. We monitor swap usage closely because excessive swapping indicates memory pressure and can significantly slow down system performance. It's essentially our safety net that prevents out-of-memory crashes when applications demand more RAM than physically available.

**Task 11:**

What is Mount ? how do you mount and unmount file system in Linux?

Mount is the process we use to make a file system accessible by attaching it to a specific directory in the Linux directory tree. We regularly mount external drives, network shares, and partitions using the mount command - for example, 'mount /dev/sdb1 /mnt/backup' to attach a USB drive to our backup directory. To unmount, we use 'umount /mnt/backup' or 'umount /dev/sdb1', and we always ensure no processes are using the mount point before unmounting to avoid data corruption. We configure permanent mounts in /etc/fstab so they automatically mount at boot time, which is essential for our production servers. For temporary mounts, we create mount points as needed and clean them up afterward to maintain a organized file system structure.

**Task 12: chmod Command**

The chmod command is what we use to change file and directory permissions in Linux, controlling who can read, write, or execute files. We use it in two ways - either with numeric notation like 'chmod 755 filename' where 7 gives full permissions to owner, 5 gives read and execute to group and others, or symbolic notation like 'chmod u+x filename' to add execute permission for the user. We commonly use 644 for regular files that need to be readable by everyone but writable only by owner, and 755 for directories and executable scripts. In our daily work, we frequently use it to make scripts executable or secure sensitive configuration files by restricting access. Understanding chmod is essential since proper file permissions are critical for both system security and functionality.

**Task 13:**

Can you add a new user account? Crate a new user in different ways and paste ss

**Task 14:**

Can you change the password of a user?

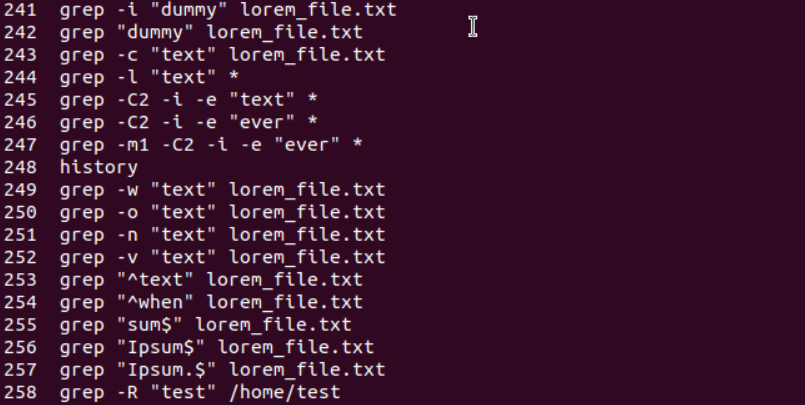
How do you do that? Plz share ss

**Task 15: Difference Between Process and Thread**

The main difference we work with is that a process is a complete independent program running in its own memory space, while threads are smaller tasks running inside a process that share the same memory. When we launch an application, it creates a process with its own resources, but that process can create multiple threads to handle different jobs at the same time. We see processes as completely separate from each other - if one crashes, others keep running normally, but threads within the same process share everything and can affect each other. In our development work, we use processes when we need complete isolation and security, but we use threads when we want fast communication and shared data within the same application. Think of a process as a separate house and threads as different rooms in that house - rooms share utilities but houses are completely independent.

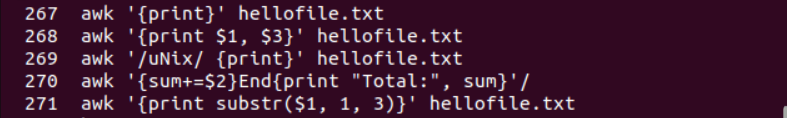
**Task 16:**

**Linux Grep commands**



**Task 17:**

**AWK commands:**



**Task 18: Check File Access Permissions**

ls -l filename

**What it shows:** File permissions in format like -rwxrw-r--

* First character: file type (- for file, d for directory)
* Next 9 characters: permissions for owner/group/others (rwx = read/write/execute)

**Task 19: Default Permissions for New File**

touch newfile.txt

ls -l newfile.txt

**Default permissions:**

* **Owner** → rw- (read, write)
* **Group** → r-- (read only)
* **All others** → r-- (read only)

*Default is typically 644 (rw-r--r--)*

**Task 20: Change to Read-Only for All**

chmod 444 filename

This gives read-only permission to owner, group, and others (r--r--r--)

**Task 21: Owner Read/Write, Group Read, Others None**

chmod 640 filename

* Owner: rw- (6 = 4+2 = read+write)
* Group: r-- (4 = read)
* Others: --- (0 = no permissions)

**Task 22: Command for -rw-r-----**

chmod 640 filename

This creates permissions: -rw-r-----

**Task 23: Change to -rwxr-x--x**

chmod 751 chmod.exercises

* Owner: rwx (7 = 4+2+1 = read+write+execute)
* Group: r-x (5 = 4+1 = read+execute)
* Others: --x (1 = execute only)

**Task 24: Command for -rwxr-x--x**

chmod 751 filename

**Task 25: chown -c master file1.txt**

This command changes the **owner** of file1.txt to "master" user. The -c flag makes it verbose, showing what changes were made.

**Task 26: What is a Process**

A **process** is a running instance of a program in memory. It has:

* Unique Process ID (PID)
* Memory space
* System resources
* Execution state

**Task 27: Check Foreground/Background Processes**

jobs # Shows background jobs in current shell

ps # Shows current processes

ps -ef # Shows all processes with details

**Task 28: List All Running Processes**

ps aux # All processes with detailed info

ps -ef # All processes in full format

htop # Interactive process viewer (if installed)

**Task 29: What ps -f Does**

ps -f

**What it does:** Shows processes in **full format** with additional details:

* UID (User ID)
* PID (Process ID)
* PPID (Parent Process ID)
* Start time
* Command with full path

**Task 30: Create Variables**

Name="prasunamba"

Id=10001

echo $Name

echo $Id

**Output:** Will print "prasunamba" and "10001"

**Task 31: Make Variable Read-Only**

Name="Prasunamba"

readonly Name

Name="Meher" # This will give error

**What happens:** You'll get error: "bash: Name: readonly variable" **Reason:** Once marked readonly, variable cannot be modified.

**Task 32: Unset Variables**

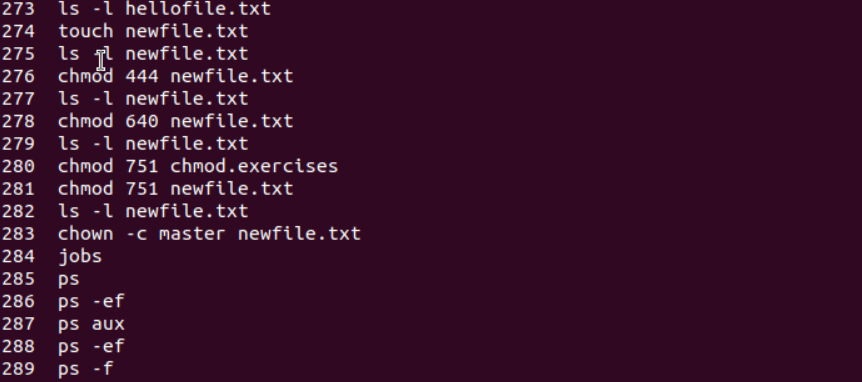
unset Name

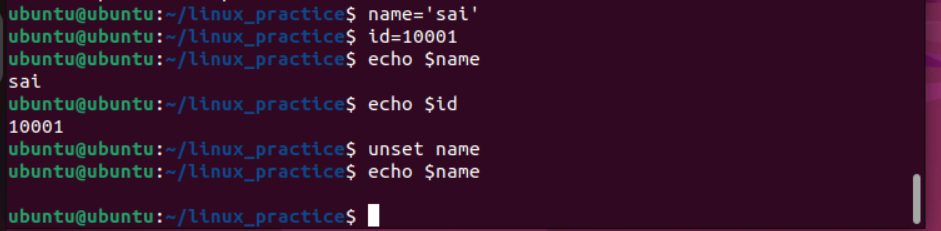
echo $Name # Prints nothing (blank line)

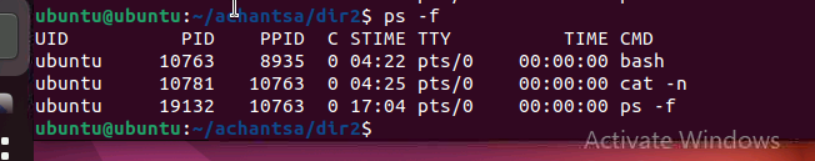
**Why blank:** unset removes the variable from memory, so echo $Name has nothing to display.

**Key Points:**

* Use touch to create test files
* Remember: r=4, w=2, x=1 for chmod calculations
* Variables in bash are case-sensitive
* readonly prevents modification permanently
* unset completely removes variables from memory







**Variable Type:**

When a shell is running, three main types of variables are present −

Local Variables − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

Environment Variables − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

Shell Variables − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

I'll help you complete these tasks step by step:

## ****Task 33: Create Array with Friends' Names****

# Create array

NAME[0]="Ram"

NAME[1]="Sita"

NAME[2]="Tina"

NAME[3]="Veena"

NAME[4]="Tim"

# Print individual elements

echo "First Index: ${NAME[0]}"

echo "Second Index: ${NAME[1]}"

echo "Third Index: ${NAME[2]}"

echo "Fourth Index: ${NAME[3]}"

echo "Fifth Index: ${NAME[4]}"

**Output:**

First Index: Ram

Second Index: Sita

Third Index: Tina

Fourth Index: Veena

Fifth Index: Tim

## ****Task 34: Print All Array Elements at Once****

# Using the same array from above

echo "All elements with [\*]: ${NAME[\*]}"

echo "All elements with [@]: ${NAME[@]}"

**Output:**

All elements with [\*]: Ram Sita Tina Veena Tim

All elements with [@]: Ram Sita Tina Veena Tim

**Difference between [\*] and [@]:**

* ${NAME[\*]} - Treats all elements as single word
* ${NAME[@]} - Treats each element as separate word (better for loops)

## ****Shell Scripting Concepts****

### **Operators:**

1. **Arithmetic:** +, -, \*, /, % (used with expr or $(( )))
2. **Relational:** -eq, -ne, -gt, -lt, -ge, -le (equal, not equal, greater than, etc.)
3. **Boolean:** -a (AND), -o (OR), ! (NOT)
4. **String:** =, !=, -z, -n (equal, not equal, empty, not empty)
5. **File Test:** -f, -d, -r, -w, -x (file exists, directory, readable, writable, executable)

### **Control Structures:**

* **if...fi:** Basic condition
* **if...else...fi:** Two-way condition
* **if...elif...else...fi:** Multiple conditions
* **case...esac:** Multiple choice (like switch)

### **Loops:**

* **while:** Repeat while condition is true
* **for:** Iterate through list/range
* **until:** Repeat until condition becomes true
* **select:** Create menu-driven programs

## ****Task 35: Analyze the Loop Output****

a=0

while [ "$a" -lt 10 ] # this is loop1

do

b="$a"

while [ "$b" -ge 0 ] # this is loop2

do

echo -n "$b "

b=`expr $b - 1`

done

echo

a=`expr $a + 1`

done

**What this code does:**

* Outer loop: a goes from 0 to 9
* Inner loop: For each a, b starts at a and counts down to 0
* echo -n prints without newline, echo adds newline at end of each row

**Expected Output:**

0

1 0

2 1 0

3 2 1 0

4 3 2 1 0

5 4 3 2 1 0

6 5 4 3 2 1 0

7 6 5 4 3 2 1 0

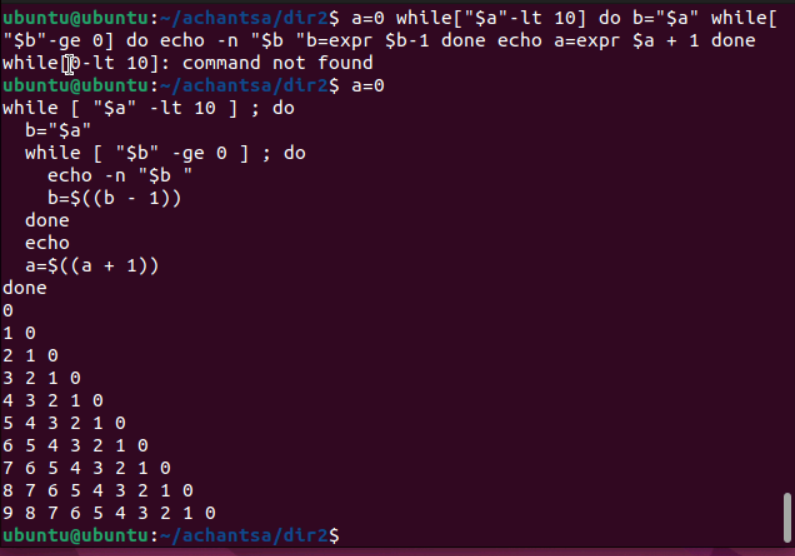
8 7 6 5 4 3 2 1 0

9 8 7 6 5 4 3 2 1 0

**Explanation:**

* When a=0: b starts at 0, prints "0"
* When a=1: b starts at 1, prints "1 0"
* When a=2: b starts at 2, prints "2 1 0"
* And so on...

Each line shows numbers counting down from the current value of a to 0.



**Key Points:**

* -lt means "less than"
* -ge means "greater than or equal to"
* expr is used for arithmetic operations
* Backticks ` ` execute commands and return output