Practical Number: 1

CO/PO: CO1, PO1, PO2

Problem Definition

As a Linux user, perform essential file system navigation and command-line operations using the Linux terminal. This includes accessing and understanding the shell environment, creating and managing files and directories using commands like mkdir, rmdir, cd, ls, touch, cp, mv, and rm. Students are also expected to handle text viewing and editing using tools such as cat, more, less, head, and vi/gedit. Additionally, commands like whoami, hostname, date, and cal are used to retrieve system and user-related information. The goal is to build familiarity with navigating the Linux shell and performing common file operations efficiently.

Key Questions / Analysis / Interpretation to be evaluated during/after Implementation

Que / Key Point
How to access the terminal and navigate directories using cd, pwd, and ls?
How to create and remove directories and files using mkdir, rmdir, rm, and touch?
How to view and manipulate the content of files using cat, more, less, head, tail, cut, grep, and sort?
What are the differences between vi and gedit? When and how are they used?
How do commands like who, whoami, hostname, and history help understand the system and user environment?
How to find built-in help or manual pages using man and help?

Supplementary Problems

- Create a simple shell script that displays system information using date, who, cal, hostname, and uname
- List all files containing a specific word using grep
- Combine and sort the contents of multiple text files
- Use uniq, tr, comm, and cmp to compare and manipulate text data
- Clear the screen and echo formatted messages with clear and echo

Key Skills to be addressed

- Terminal navigation and command usage
- Directory and file management
- Text file viewing and editing
- Basic shell utilities and redirection
- Using help systems effectively (man, help)
- Understanding system/user information retrieval commands

Applications

- Crucial for all Linux-based environments
- Required in shell scripting and automation tasks
- Foundational for DevOps, administration, cybersecurity, and programming roles
- Supports faster debugging, data processing, and server-side development

Learning Outcome

Students will be able to:

- Navigate the Linux file system using basic terminal commands.
- Create, delete, and manage files and directories through the command line.
- View, edit, and process text files using standard Linux utilities.
- Retrieve system and user information using built-in commands.
- Utilize man and help to independently explore and understand command usage.
- Apply basic screen and output control commands for better terminal usage.
- Build confidence in using the Linux shell for routine operations in an OS environment.

Tools/Technology To Be Used

- Ubuntu/Linux Terminal
- Bash Shell
- Text Editors (vi, gedit, nano/vim)

- **Total Hours of Problem Definition Implementation**
  3 Hours
- **Total Hours of Engagement**
  4 Hours
  (*Includes implementation + modification + faculty testing*)

Post Laboratory Work Description

- Submit screenshots or logs of all commands executed
- Prepare a write-up highlighting:
    - New commands learned
    - Errors encountered and how they were resolved
    - Key observations about command behavior and output
    - Short summary of practical significance

Evaluation Strategy Including Viva

- Live demonstration of command execution and output
- Viva covering command purpose, syntax, options, and scenarios
- Evaluation of submitted logs, screenshots, and reflective write-up

Feedback on Problem Definition Implementation
(Satisfaction Level 0 to 4):

Level	**Criteria**
0 - Not Attempted	No practical demonstration; unable to answer viva; no logs or write-up submitted
1 - Poor	Partial or incorrect execution; weak command understanding; incomplete or irrelevant logs
2 - Fair	Basic functionality shown with some issues; limited conceptual clarity; logs show effort but lack depth
3 - Good	Most commands executed correctly; clear understanding of utilities and options; logs/write-up mostly complete
4 - Excellent	All tasks done accurately; confident in command usage; insightful, organized logs and reflections submitted

Practical Number: 2

CO/PO: CO1, PO1, PO2

Problem Definition

As a user administration tasks in a Linux-based operating system, including the creation, modification, and deletion of user accounts using tools like useradd, usermod, and userdel. This involves managing user and group IDs (UIDs/GIDs), configuring home directories, setting shell environments, and enforcing password policies. Additionally, it includes comprehensive group management through commands like groupadd, gpasswd, and groupdel to organize users based on roles and access levels. The task also covers the assignment and management of file and directory permissions using chmod, chown, and chgrp to enforce access control through permission bits (read, write, execute) and advanced permission schemes like SUID, SGID, and sticky bits. Regular monitoring of user activities using tools such as who, w, last, and log files (/var/log/auth.log, /var/log/secure) is also emphasized to ensure system security, policy compliance, and efficient resource utilization.

Key Questions to be evaluated during Implementation

Question(s)
How to delete a user with the home directory?
How to change a user's shell or home directory?
How to account expiry dates for users, and why is this important in enterprise environments?
Describe the impact of SGID on directories. How does it affect file creation within a group-shared directory?
How to enforce two-factor authentication (2FA) in a Linux system using tools like Google Authenticator or PAM modules?
How to configure for monitoring specific file access or privilege escalation attempts?

Supplementary Problems

- Create batch of users using shell script
- Assigning users to multiple groups.
- Restricting user access to certain directories using permissions.

Key Skills to be addressed

- Linux terminal usage
- Command-line user and group administration

- File system permission management
- Troubleshooting access issues

Applications

- Used in system administration
- Crucial for managing access in multi-user environments
- Foundational skill in DevOps and Cybersecurity

Learning Outcome

Student will be able to

- Understand the user management lifecycle in Linux
- Demonstrate the working of Linux administrative commands
- Learn how to enforce access control using permissions

Tools/Technology to Be Used

- Ubuntu/Linux Terminal
- Bash Shell
- Text Editors (nano/vim)

- **Total Hours of Problem Definition Implementation**

3 Hours

- **Total Hours of Engagement**

4 Hours

(*Includes implementation + modification + faculty testing*)

Post Laboratory Work Description

- Submit screenshots/logs of each command used.
- Prepare a reflection on what commands were new and what errors were encountered.
- Short write-up on practical significance.

Evaluation Strategy Including Viva

- Demonstration of all user admin tasks
- Viva covering command syntax, configuration files, and error handling
- Evaluation of submitted log and write-up

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4)

Level	Criteria
0 - Not Attempted	No practical demonstration; unable to answer viva questions; no log or write-up submitted.
1 - Poor	Partial or incorrect demonstration of tasks; minimal understanding of commands and config files in viva; log/write-up incomplete or lacks relevance.
2 - Fair	Basic user admin tasks shown with errors; limited knowledge of command syntax and error handling in viva; log and write-up show effort but lack depth.
3 - Good	Most user admin tasks are correctly demonstrated; good understanding of commands, key configuration files (/etc/passwd, /etc/group, etc.); log/write-up mostly complete and accurate.
4 - Excellent	All user admin tasks demonstrated correctly and efficiently; excellent command over syntax, config files, and error handling during viva; comprehensive and well-documented log and write-up with insights or best practices.

Practical Number: 3

CO/PO: CO2

Problem Definition

This practical focuses on essential aspects of Linux system administration, including managing and monitoring processes, controlling services and daemons, and enhancing command-line productivity. Students will learn how to observe and manage active system processes, terminate them when necessary, and monitor system performance using commands like top, ps, kill, pkill, w, and lscpu. It also includes working with system services using the systemctl command with options such as start, stop, restart, enable, disable, is-active, is-enabled, and is-failed. Additionally, the practical emphasizes improving command-line efficiency through the use of input/output redirection operators (<, >, >>) and pipes (|) to streamline data processing between commands.

Key Questions to be evaluated during Implementation

Question(s)
What commands are used to list, monitor, or kill processes in Linux?
How can system services be managed using the systemctl command?
How do piping and redirection contribute to efficient command-line usage?
What is the practical impact of enabling or disabling a service permanently?

Supplementary Problems

- Write a shell script that logs real-time CPU and memory usage.
- Create a utility script that checks the status of critical system services,
- Automate the restart of failed services and log the actions taken.

Key Skills to be addressed

- Effective process monitoring and control using terminal commands.
- Practical management of Linux services and daemons.
- Productivity enhancement through piping and redirection.
- Development of small automation scripts using system utilities.

Applications

- Server and system monitoring for system administrators.
- Automating repetitive administrative tasks.
- Ensuring high system uptime by managing critical services.
- Improving workflow efficiency with command-line utilities.

Learning Outcome

Students will be able to

- Gain proficiency in observing and managing system processes.
- Develop familiarity with the systemctl utility and its parameters.
- Understand and apply redirection and piping for data flow between commands.
- Build foundational knowledge in Linux system administration through scripting and terminal operations.

- **Tools/Technology to Be Used**

- Ubuntu/Linux Terminal
- Bash Shell
- Text Editors (nano/vim)

- **Total Hours of Problem Definition Implementation**

3 Hours

- **Total Hours of Engagement**

4 Hours

(Includes implementation + modification + faculty testing)

Post Laboratory Work Description

- Explore additional process monitoring tools such as htop, atop, and glances.
- Refine scripts to include log file generation and service health monitoring.
- Practice automating daily monitoring tasks using cron jobs.

Evaluation Strategy Including Viva

- Demonstration of process control and service management commands.
- Explanation of command usage and their practical scenarios.
- Viva covering syntax, command options, and their outcomes.
- Evaluation based on script clarity, logic, and correctness.

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4)

Level	Criteria
0 – Not Attempted	No practical demonstration; unable to explain process-related commands or service management; no log files or write-up submitted.
1 – Poor	Partial or incorrect execution of commands like `ps`, `kill`, or `systemctl`; minimal understanding of process states, service control, or command-line redirection during viva; log/write-up is incomplete or irrelevant.

2 – Fair	Basic tasks like viewing processes or controlling a service are demonstrated with minor errors; limited grasp of redirection, piping, and command syntax; log/write-up shows effort but lacks completeness or accuracy.
3 – Good	Most process and service management tasks are demonstrated correctly; good understanding of commands (`top`, `systemctl`, etc.), I/O redirection, and pipes; log/write-up is mostly complete and correctly documents outcomes.
4 – Excellent	All tasks are accurately and efficiently demonstrated including process monitoring, service control, and productivity features; strong command of syntax, flags, and service states; log and write-up are thorough, well-structured, and reflect best practices or insights.

Practical Number: 4

CO/PO: CO5,CO6, PO1, PO2

Problem Definition

As a Linux system user or administrator, understand and explore the structure and behavior of the Linux File System. Perform tasks to analyze disk usage, mount and unmount file systems, and locate files using system utilities. This includes the use of commands such as df and du for disk space monitoring, mount and umount for managing external or virtual storage, and locate and find for efficiently identifying files and directories across the file hierarchy. Through these activities, students will gain familiarity with how Linux organizes data, manages storage, and tracks file system usage. This foundational knowledge is critical for advanced system management, performance monitoring, and troubleshooting.

Key Questions to be Evaluated During Implementation

- What is the structure of the Linux file system and how is it organized?
- How can you check the available and used disk space on mounted file systems?
- How do you calculate the size of specific directories or files?
- What is the difference between mount and umount, and when are they used?
- How do find and locate differ in terms of file searching efficiency?

- What precautions must be taken when mounting/unmounting devices?
- How do these tools help with troubleshooting or space management in real environments?

Supplementary Problems

- Mount a USB drive or ISO file and verify with mount and df
- Use find to locate all .log files modified in the last 7 days
- Use du to check the size of the /home directory and sort output
- Create a script that reports disk usage every 10 minutes
- Compare outputs of locate vs find for a given filename

Key Skills to be Addressed

- Understanding Linux directory structure
- Disk usage monitoring and analysis
- File system mounting/unmounting
- File search techniques using indexing and real-time scanning
- Basic storage troubleshooting

Applications

- Crucial for system administration and disk space management
- Essential in setting up new storage or removable devices
- Important for log monitoring and cleaning up large files
- Foundational for performance tuning and automation scripts

Learning Outcome

Students will be able to:

- Describe the hierarchy and layout of the Linux file system
- Monitor disk and directory usage using commands like df and du
- Mount and unmount external or virtual file systems safely
- Search and locate files using both indexed and recursive methods
- Interpret the output of file system tools to assess storage health
- Apply file system-related commands to solve storage and organization issues in real scenarios

Tools/Technology to Be Used

- Ubuntu/Linux Terminal
- Bash Shell
- System Utilities (df, du, mount, umount, locate, find)

Total Hours of Problem Definition Implementation

3 Hours

Total Hours of Engagement

4 Hours (*Includes implementation + modification + faculty testing*)

Post Laboratory Work Description

- Submit screenshots/logs of commands executed
- Provide a write-up discussing observed outputs, problems faced, and key learnings
- Mention any unexpected findings or performance insights related to disk usage

Evaluation Strategy Including Viva

- Hands-on demonstration of all listed file system commands
- Viva covering command flags, output interpretation, and safety concerns
- Evaluation of logs and reflective write-up quality

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4)

Level	Criteria
0 - Not Attempted	**No attempt made; unable to answer viva; no logs submitted**
1 - Poor	**Attempted partially; weak understanding of file system tools**
2 - Fair	**Demonstrated basics; some errors; incomplete understanding/logs**
3 - Good	**Good usage of commands; shows system understanding; clean documentation**
4 - Excellent	**Complete and accurate; clear output interpretation; insightful reflection**

Practical Number: 5

CO/PO: CO4, CO5, CO6, PO1, PO2, PO5

Problem Definition

As a system user or administrator, it is often necessary to automate routine tasks, validate inputs, manage files, and interact with users using shell scripts. In this practical, students will write a series of shell scripts to perform common automation tasks that are frequently required in system administration and DevOps environments.

Exercise:

1. Check whether the given file exists or not.
2. Check whether the argument passed from the command line is a file or directory.
3. List out all empty files in the current working directory and its subdirectories.
4. Compare two files passed as command-line arguments. If they are the same, delete the second one. Otherwise, suggest changes.
5. Print the multiplication table of a given number.
6. Check executable rights for files in the current directory. If missing, make them executable.
7. Arithmetic calculator using command-line arguments (addition, subtraction, etc.).
8. Reverse a given number using shell script.
9. Convert a string from lowercase to uppercase and vice versa.
10. Create a menu using **case** to:
    - List files
    - Show today's date
    - Show system users
    - Show user's processes

- Exit to prompt

Key Questions / Analysis / Interpretation to be evaluated during/after Implementation

- How do you validate input parameters and file types in a shell script?
- What is the importance of automating permission checking for executable files?
- How do conditional (`if`, `case`) and loop constructs (`for`, `while`) work in shell scripting?
- How is file comparison handled in bash and how can differences be displayed?
- How does a menu-driven shell script enhance usability and interactivity?

Supplementary Problems

- Write a script that counts the number of users currently logged in.
- Create a backup script that compresses files modified in the last 24 hours.
- Write a script to check disk usage and alert if space usage exceeds 90%.

Key Skills to be addressed

- Command-line automation and scripting fundamentals
- File handling and permission management
- Input validation and logic building
- Process control and user interaction in scripts

Applications

- Automation of repetitive administrative and data-processing tasks
- Foundation for DevOps pipelines and CI/CD scripts
- Basic file management and reporting for system health
- Useful for data cleaning and custom tooling

Learning Outcome

Student will be able to

- Write efficient shell scripts to automate routine tasks
- Validate and manage file types, permissions, and command-line arguments
- Implement control structures like `if`, `case`, `for`, and `while`
- Create interactive scripts using menu-driven options
- Understand practical applications of scripting in real system environments

Tools/Technology to Be Used

- Ubuntu/Linux Terminal
- Bash Shell
- Text Editors (nano/vim)

- **Total Hours of Problem Definition Implementation**
  4 Hours
- **Total Hours of Engagement**
  6 Hours
  (*Includes implementation + modification + faculty testing*)

Post Laboratory Work Description

- Submit script files (.sh) and command-line outputs (screenshots or log files).
- Write-up should include:
  - Description of each script and its use case
  - Challenges or errors faced and how they were resolved
  - New bash commands or concepts learned
  - Summary of real-world relevance

Evaluation Strategy Including Viva

- Live demonstration of each script and its execution
- Viva questions based on logic, syntax, and command usage
- Review of submitted scripts and documentation
- Bonus for creativity in logic or use of advanced bash features

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4)

Level	Criteria
0 - Not Attempted	No practical demonstration; unable to answer viva questions; no log or write-up submitted.
1 - Poor	Partial or incorrect demonstration of tasks; minimal understanding of commands and config files in viva; log/write-up incomplete or lacks relevance.
2 - Fair	Basic user admin tasks shown with errors; limited knowledge of command syntax and error handling in viva; log and write-up show effort but lack depth.
3 - Good	Most user admin tasks are correctly demonstrated; good understanding of commands, key configuration files (/etc/passwd, /etc/group, etc.); log/write-up mostly complete and accurate.
4 - Excellent	All user admin tasks demonstrated correctly and efficiently; excellent command over syntax, config files, and error handling during viva; comprehensive and well-documented log and write-up with insights or best practices.

Practical Number: 6

CO/PO: CO2, CO3 PO1, PO2

Problem Definition

In a multi-user web server environment, the server must handle multiple incoming client requests simultaneously. This is achieved by creating new processes to handle each request, allowing the server to respond to multiple users concurrently. A similar scenario can be simulated using the `fork()` system call in Linux.

In this practical, students will write a C program that mimics how a basic server creates child processes to handle multiple client requests. The number of concurrent "clients" (or tasks) will be provided by the user, and the program will create a corresponding number of processes using `fork()`. Each child process will simulate processing time using `sleep()` and print its process ID and parent process ID. This exercise demonstrates how operating

systems manage multitasking, resource sharing, and process creation in real-world systems.

Key Questions / Analysis / Interpretation to be evaluated during/after Implementation

- What is a fork() system call? What does it return in parent and child?
- How does process creation using fork() lead to exponential growth?
- How can you prevent creation of zombie processes?
- How does each process get its unique process ID and parent process ID?
- What are best practices to avoid unnecessary resource usage when using multiple fork() calls?
- How does the use of fork() simulate the behavior of a web server handling multiple client connections?

Sample Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main()
{
   int n, i;
   printf("Enter number of fork levels: ");
   scanf("%d", &n);
   int total = 1 << n; // Total expected processes = 2^n
   for (i = 0; i < n; i++) {
         if (fork() == 0) {
         printf("Child process: PID = %d, PPID = %d\n", getpid(), getppid());
         } else {
         break;
         }
   }
   sleep(1); // Let child complete execution
   return 0;
}
```

Supplementary Problems

- Modify the above program to print the depth level of each process.

- Track the full process tree using the pstree or ps -f command.
- Create a shell script to kill all child processes after a delay.

Key Skills to be addressed

- System-level process creation using fork()
- Process control using PID and PPID
- Understanding of concurrent execution and hierarchy
- Avoiding zombie/orphan processes

Applications

- Core to OS-level programming and process scheduling
- Foundation for implementing multitasking applications
- Useful in server-client modelling and multiprocessing systems

Learning Outcome

Student will be able to

- Demonstrate and explain the working of fork() system call.
- Understand how process hierarchy is created and maintained.
- Trace the execution flow and PID/PPID relationships.
- Write and test C programs that use fork() for multiple processes.

Tools/Technology to Be Used

- Ubuntu/Linux Terminal
- GCC Compiler (for compiling C programs)
- Text Editors (nano/vim)

- **Total Hours of Problem Definition Implementation**
  1.5 Hours
- **Total Hours of Engagement**

2 Hours

(*Includes implementation + modification + faculty testing*)

Post Laboratory Work Description

- Submit screenshot of output showing process IDs and fork structure.
- Include C code, compilation steps, and `ps/pstree` results.
- Short write-up must include:
    - Behavior of fork at each level
    - Common errors and resolution (e.g., infinite forks)
    - Understanding of PID/PPID relationship

Evaluation Strategy Including Viva

- Viva on return values of `fork()`, process hierarchy
- Observation of process tree behavior
- Output screenshots + code submission
- Discussion on CPU/memory usage for large number of processes

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4)

Level	Criteria
0 - Not Attempted	No practical demonstration; unable to answer viva questions; no log or write-up submitted.
1 - Poor	Partial or incorrect demonstration of tasks; minimal understanding of commands and config files in viva; log/write-up incomplete or lacks relevance.
2 - Fair	Basic user admin tasks shown with errors; limited knowledge of command syntax and error handling in viva; log and write-up show effort but lack depth.

3 - Good	Most user admin tasks are correctly demonstrated; good understanding of commands, key configuration files (/etc/passwd, /etc/group, etc.); log/write-up mostly complete and accurate.
4 - Excellent	All user admin tasks demonstrated correctly and efficiently; excellent command over syntax, config files, and error handling during viva; comprehensive and well-documented log and write-up with insights or best practices.

Practical 7 - File System Operations

CO/PO: CO2, CO6 | PO1, PO2

Problem Definition

Implement and analyze key process-related system calls in Linux using the C programming language. This includes creating child processes using fork(), synchronizing processes with wait() and waitpid(), executing new programs using the exec family of calls, and manipulating file descriptors with open(), close(), read(), and write(). Additionally, concepts of orphans, zombies, and multiple child process creation (both recursive and sequential) should be demonstrated. The goal is to build a strong understanding of process life cycles and inter-process relationships within the Linux operating system.

Key Questions / Analysis / Interpretation to be Evaluated During/After Implementation

· How does fork() work and what is returned in the child vs. parent process?

· What is the purpose of wait() and waitpid()?

· How does exec() replace the current process image, and what variants are available?

· How are file descriptors inherited during fork() and how are they manipulated?

· What is the difference between orphan and zombie processes?

· How to observe process states using tools like ps and how to interpret them?

· How does recursive vs. sequential forking differ in terms of execution flow?

Supplementary Problems

- Create a program that redirects input and output using file descriptors.
- Demonstrate the use of dup/dup2 for file descriptor duplication.
- Use ps and top to monitor process behavior in real-time.
- Extend the fork-exec model to launch different programs conditionally.
- Implement child process logging with timestamps to a file.

Key Skills to be Addressed

- Process creation and synchronization
- Understanding and handling file descriptors
- Redirection and execution of external programs
- Observation and analysis of process states
- Shell-level interaction with process management utilities
- Building confidence in system-level programming

Applications

- Essential for OS-level software development
- Foundation for writing daemons, servers, and concurrent applications
- Required in cybersecurity (process monitoring, privilege management)
- Used in containerization, automation scripts, and system diagnostics

Learning Outcome

Students will be able to:

- Use fork(), exec(), and wait() calls to manage processes in C
- Understand the relationship between parent and child processes

- Perform file operations using system-level calls
- Monitor and interpret process states in Linux
- Handle process synchronization and termination
- Design multi-process applications in a Unix/Linux environment

Tools/Technology To Be Used

- Ubuntu/Linux Terminal
- GCC Compiler
- System Utilities: ps, top, kill
- Bash Shell for testing redirection
- Text Editor (vi, gedit, nano)

Total Hours

- Total Hours of Problem Definition Implementation: 3 Hours
- Total Hours of Engagement: 4 Hours (Includes implementation + modification + faculty testing)

Post Laboratory Work Description

- Submit .c source files and executable binaries
- Provide execution screenshots/logs for each task
- Write-up must include:
    - New concepts/commands learned
    - Debugging challenges and resolutions
    - Observation on process states and behavior
    - Summary of practical relevance

Evaluation Strategy Including Viva

· Demonstration of each program and its output

· Viva covering process states, system call syntax, and practical usage

· Review of submitted logs, source codes, and reflective write-up

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4 — Measurable Criteria)

Level	Criteria
0 – Not Attempted	· No code submitted or executed. · Unable to explain or demonstrate any concept during viva. · No screenshots, logs, or write-up provided.
1 – Poor	· Submitted code has major compilation/runtime errors. · Demonstration fails for more than 50% of tasks. · Viva responses show lack of basic understanding. · Logs or write-up missing or irrelevant.

2 – Fair	· Code runs partially or with limited functionality (≤60% tasks implemented) · Demonstration covers basic process creation but lacks advanced tasks (e.g., exec, zombie/orphan). · Viva answers are basic; understanding is shallow. · Logs and write-up show effort but lack depth or clarity.
3 – Good	· Correct implementation of ≥75% of tasks including fork, wait, and exec. · Able to explain process relationships (parent-child, zombie, orphan) in viva. · Logs/screenshots are mostly complete and organized. · Write-up includes observations and brief error handling.
4 – Excellent	· All core and supplementary tasks implemented correctly without errors. · Viva responses are confident and cover edge cases and system behavior. · Submission includes detailed logs, screenshots, and clearly written reflections. · Write-up highlights learned concepts, challenges faced, and practical significance with clarity.

Practical Number: 8

CO/PO: CO3, CO6 | PO1, PO2, PO4

Problem Definition

Design and implement synchronization mechanisms for concurrent processes and threads using the pthreads library in Linux. Demonstrate race conditions using shared variables and resolve them using synchronization primitives such as mutex locks, semaphores, and condition variables. Implement thread-safe counter programs, parallel updates using multiple threads, and a deterministic toggle sequence between parent and child threads.

Key Questions / Analysis / Interpretation to be Evaluated During/After Implementation

· What happens when multiple threads access shared data without synchronization?

· How does pthread_mutex prevent race conditions?

· When to use condition variables instead of simple locks?

· How can thread execution be synchronized to follow a pattern (e.g., toggle)?

· What are the differences between race conditions and deadlocks?

· How to structure thread-safe C programs using pthreads?

Supplementary Problems

· Use semaphores (`sem_init`, `sem_wait`, `sem_post`) to replace mutex in Task 2.

· Modify the multi-threaded counter to use a single mutex for all indexes and observe performance.

· Log the thread ID and timestamp for each access to shared variable.

· Add random delays using `usleep()` in threads to simulate timing variations.

· Create a deadlock scenario using two mutexes and resolve it using proper lock ordering.

Key Skills to be Addressed

- Understanding and applying synchronization primitives
- Thread creation and management in Linux
- Mutex and condition variable usage in pthreads
- Implementing deterministic thread behavior
- Debugging multithreaded programs
- Avoiding race conditions and deadlocks

Applications

- Essential in multi-threaded software and system programming
- Critical for performance and reliability in concurrent systems
- Used in real-time systems and operating system components
- Applied in cloud computing, parallel processing, and networking
- Fundamental for understanding process/thread behavior in high-performance systems

Learning Outcome

Students will be able to:

- Identify and analyze race conditions in threaded programs.
- Apply pthread mutex and condition variables to control thread execution
- Implement safe data access in multi-threaded applications
- Build deterministic thread interaction patterns
- Develop confidence in writing and testing concurrent code in Linux environment

Tools/Technology To Be Used

- Ubuntu/Linux Terminal

- GCC Compiler
- pthreads library
- Text Editor (gedit, nano, or vi)
- Debugger (gdb), Valgrind (optional)

Total Hours

- Total Hours of Problem Definition Implementation: 4 Hours
- Total Hours of Engagement: 4 Hours (Includes implementation + modification + faculty testing)

Post Laboratory Work Description

- Submit source code for all programs (counter, indexed lock, toggle)
- Provide execution screenshots/logs for each test case
- Write-up must include:
    - Description of synchronization strategy used
    - Errors encountered and solutions applied
    - Comparative results with/without locks
    - Reflection on thread synchronization impact

Evaluation Strategy Including Viva

- Demonstration of each program and its output
- Viva covering pthread concepts, synchronization constructs, and logic
- Evaluation of code structure, correctness, and documentation quality

Feedback on Problem Definition Implementation (Satisfaction Level 0 to 4 — Measurable Criteria)

Level	Criteria
0 – Not Attempted	No submission or execution attempt Unable to explain thread or lock concepts. No logs, code, or write-up submitted.
1 – Poor	Code has major errors or does not compile. Thread output is inconsistent or incorrect. Weak or no understanding in viva. Logs or write-up are missing or off-topic.
2 – Fair	Partial implementation (≤60% programs working). Some threads are synchronized, some are not. Basic but unclear understanding in Write-up lacks clarity or completeness.
3 – Good	Most programs (≥75%) work with correct synchronization. Able to explain core pthread concepts in viva. Logs and output mostly consistent. Write-up addresses key issues and fixes.
4 – Excellent	All programs run correctly with clean synchronization. Output matches expected deterministic behavior. Viva responses are confident and precise. Logs and write-up are insightful and well-structured.

Practical Number 9

CO/PO: CO1, CO2, CO3(As per IT Booklet)

Problem Definition:

To implement multi-threaded programming using POSIX threads (pthreads) and solve classic synchronization problems like mutual exclusion, reader-writer locks, and producer-consumer using mutexes, semaphores, and condition variables in C language.

1 hello pthreads!

In this section, we will use the pthread (POSIX threads) library to understand the working of threads, share data across threads and synchronization.

(a) Read the sample program threads.c provided. In this program, 100 threads are created which execute a function that updates a shared counter. pthreads are declared using pthread_t. To start a thread, use pthread_create(). This function takes four arguments: address of the pthread variable representing this thread (pthread ID), attributes of the new thread, start routine (function) and parameters for the start routine. On pthread_create() a new thread is created with the above arguments and commences execution at the start routine. The main thread waits for the thread to exit and then reaps the thread using pthread_join(). Compile the program using:

gcc threads.c -lpthread

Read the following man pages to understand pthreads in more detail: pthreads, pthread_create, pthread_join, pthread_detach, pthread_exit, pthread_kill, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_unlock, pthread_mutex_trylock, pthread_spin_lock, pthread_spin_unlock, pthread_cond_signal, pthread_cond_wait, pthread_cond_broadcast, pthread_cond_init, ...

(b) Next, use pthread locks to synchronize access to shared data across threads. Write a program threads-with-mutex.c which synchronizes access to the shared counter in threads.c using a mutex lock. You can declare the mutex as a global variable.

The functions of interest are: pthread_mutex_init(), pthread_mutex_lock(), pthread_mutex_unlock(), and pthread_mutex_destroy().

---

2 argumentative pthreads

The pthread_create() function allows a single argument to the start function. The argument has to be passed by reference as a void pointer. Think about how to pass multiple arguments to the start function.

Write a program nlocks.c that does the following:

- Creates and initializes 10 shared counters and 10 locks.
- The parent process creates 10 threads.
- Each thread adds 1 to its corresponding data value 1000 times.
- The parent also updates the data items — adds 1 to data[0], then to data[1], etc. for all 10 data items and then repeats for a total of 1000 iterations.
- With correct synchronization, each of the data values should be 2000.
- Note: The index to process data items by each thread must be carefully passed to the start function.

3 To read or to write, is that the question?
Assume a situation where several threads read and write to a shared data item. Reads do not necessarily need locked access, but with parallel reads and writes, locks are required.

With a reader-writer lock, multiple readers can concurrently access the data. However, a writer must not access the data concurrently when either readers or writers access the data. A writer has to wait until all the readers or writers complete access to the shared data, while a reader does not need to wait if readers are currently active.

It would be unfair for a reader to jump in immediately ahead of a waiting writer. If that happened often enough, writers would starve.

Implement a reader-writer lock with writer preference, where new readers do not access the shared data in case of writers waiting to write, even if readers are active.

Semantics:

- With readers already present, a new reader can read only if no writer is already waiting to write.
- If a writer is waiting to write, writers get preference; readers have to wait until all writers clear out.
- A writer cannot write if readers are active or another writer is active.

Use the sample program reader-writer.c as skeleton code to get started.
Note: Implementation should use pthread conditional variables and mutex.

---

4 extra sync (For Fast Learner)

- Implement the producer-consumer using pthread condition variables and mutex locks. Assume a bounded buffer for production.

- Assume two threads, with functions hydrogen and oxygen. Each thread periodically creates a hydrogen or oxygen atom and when two hydrogen atoms and one oxygen atom is available, water is formed. Incomplete formations block further partial molecule completions. Implement this using semaphores.

Key Questions / Analysis / Interpretation [1] to be evaluated during/after [2] Implementation

1. How are pthreads created and synchronized?
2. Why is mutual exclusion required when multiple threads access shared data?
3. How does a reader-writer lock work with writer preference? How do you pass multiple arguments to a thread's start routine?
4. How are classic problems like producer-consumer modeled using condition variables?

Supplementary Problems

- Implement hydrogen-oxygen molecule formation using semaphores
- Add real-time thread scheduling priorities in pthread creation

Key Skills to be Addressed

- Multi-threaded Programming
- Synchronization using Mutex, Semaphore, Condition Variables
- Deadlock Avoidance and Process Coordination
- Classic OS concurrency problem solutions

Applications

- Server-client concurrent systems
- Real-time embedded control systems
- Operating System kernel process management

Learning Outcome

- Understand thread creation and management using pthreads
- Implement synchronization techniques for concurrent threads
- Demonstrate concurrency control through classical problems
- Apply synchronization mechanisms to real-world application systems

Tools / Technology to Be Used

- C Language
- POSIX Thread Library (pthreads)
- Linux OS / Ubuntu Terminal
- GCC Compiler

Total Hours

Problem Definition Implementation: 3 hours

Engagement: 4 hours (Implementation + Modification + Testing)

Post Laboratory Work Description

- Compile and test programs over multiple test cases
- Document observations when synchronization primitives are removed
- Demonstrate concurrent behavior and outputs
- Submit code files and execution outputs

Evaluation Strategy Including Viva

- Correct implementation and synchronization technique usage
- Code correctness, readability, and output validation
- Viva on pthreads, mutex, semaphores, reader-writer locks
- Satisfaction Level: 0 to 4

Practical Number 10:

CO/PO: CO1, CO2, CO3

Case Problem Definition:

A software firm is developing a small real-time OS for a cloud server cluster where different types of user and system processes arrive for execution.

Four types of processes exist:

- High-priority urgent system tasks (e.g. security updates)
- Short interactive tasks (e.g. command executions)
- Bulk jobs (e.g. report generation)
- Background maintenance jobs (e.g. data backups)

The OS development team must implement and test multiple CPU scheduling algorithms to analyze how each performs on a sample workload and recommend the most efficient scheduling policy.

Objective:

Implement and compare:

- First-Come-First-Serve (FCFS)
- Shortest Job First (SJF)
- Round Robin (with time quantum)
- Priority Scheduling

Process Data (Test Data)

Process	Arrival Time (ms)	Burst Time (ms)	Priority (Lower = Higher)	Category
P1	0	4	1	High-priority system task
P2	2	6	3	Background maintenance

P3	4	3	2	Interactive command
P4	6	5	4	Bulk job

Key Questions / Interpretation

- Which algorithm minimizes average waiting time and turnaround time?
- How does preemption in RR impact response time for interactive processes?
- Under what conditions does Priority Scheduling cause starvation?
- What policy would you recommend for this OS case and why?

Key Skills Addressed

- Process Scheduling Techniques
- Preemptive and Non-Preemptive Scheduling
- Turnaround, Waiting, and Response Time Calculation
- Starvation Handling and Priority Inversion Mitigation

Applications

- Cloud-based OS scheduling
- Embedded real-time systems
- Process handling in multi-user operating systems
- Database transaction scheduling

Learning Outcomes

- Apply different CPU scheduling algorithms on a simulated OS environment
- Compare their efficiency quantitatively
- Identify practical issues like starvation and response delays
- Recommend scheduling policy improvements based on data

Total Hours

- Problem Understanding: 1 hr

- Implementation: 2 hrs
- Testing & Analysis: 1 hr

Post Laboratory Work

- Execute all 4 algorithms on the provided process data
- Record individual and average Waiting Time, Turnaround Time, Response Time
- Draw Gantt Charts for each
- Compare outcomes and analyze
- Justify the best-fit scheduling policy for this case

Evaluation Rubric

Criteria	Marks
Correct Implementation of 4 Algorithms	8
Accurate Gantt Charts & Time Calculations	6
Analysis & Recommendation with Justification	4
Viva on Scheduling Concepts	2
Total	20

Report Deliverables

- Code files
- Gantt charts (drawn or plotted)
- Time calculation tables
- Recommendation section