LAB MANUAL

Course: Operating System Lab
Department: BCA & MCA
Institution: RVS College of Engineering and Technology, Jamshedpur

Academic Session: 2026-2027Prepared

By: Prof. Deepak Kumar Tiwari

Certificate

This is to certify that this Laboratory Manual has been prepared for the Operating System Lab course to help students understand Linux commands, process management, inter-process communication, threads, synchronization, and operating system concepts through practical experiments.

Acknowledgement

This laboratory manual has been prepared with reference to the prescribed syllabus and standard textbooks. It is intended to provide students with hands-on experience in Linux and Operating System concepts.

Preface

Operating Systems form the backbone of every computing system. This laboratory course complements theory classes by providing practical exposure to Linux commands, process management, memory management, system calls, debugging, threads, pipes, FIFOs, mutexes, and semaphores.

Table of Contents

Course Objectives

- Acquire practical knowledge of Linux operating system commands.
- Understand process creation, execution, and management.
- Learn inter-process communication mechanisms.
- Implement synchronization using mutexes and semaphores.
- Develop basic system programming skills in Linux.

Course Outcomes

- CO1: Implement various Linux commands.
- CO2: Implement process creation and process-related commands.
- CO3: Demonstrate IPC using pipes and FIFOs.
- CO4: Demonstrate thread creation and synchronization using mutexes and semaphores.

Laboratory Rules

- Maintain discipline inside the laboratory.
- Arrive on time and complete attendance.
- Do not modify system configuration without permission.
- Save your work frequently.
- Shutdown/logout properly after the lab.
- Submit observations and programs before leaving.

Hardware & Software Requirements

Hardware:
• Intel/AMD based PC
• Minimum 4 GB RAM (8 GB Recommended)
• 20 GB free disk space

Software:
• Ubuntu Linux 22.04 LTS (or later)
• GCC Compiler
• GNU Make
• GDB Debugger
• POSIX Thread Library (pthread)
• Terminal

Lab Evaluation Scheme

Component	Weightage (%)
Attendance	10
Experiment Performance	40
Observation Record	20
Viva-Voce	20
Lab Test	10

Mapping of Experiments with Course Outcomes

Experiment No.	Title	CO
1-3	Linux Commands & Permissions	CO1
4-7	Process, Memory, System Calls, Debugging	CO2
8-13	Processes, Pipes and FIFO	CO2, CO3
14-17	Synchronization & Threads	CO4

Semester Lab Schedule

Week	Lab Session	Experiment
1	Lab 1	Experiment 1
1	Lab 2	Experiment 2
2	Lab 3	Experiment 3
2	Lab 4	Experiment 4
3	Lab 5	Experiment 5
3	Lab 6	Experiment 6
4	Lab 7	Experiment 7
4	Lab 8	Experiment 8
5	Lab 9	Experiment 9
5	Lab 10	Experiment 10
6	Lab 11	Experiment 11
6	Lab 12	Experiment 12
7	Lab 13	Experiment 13
7	Lab 14	Experiment 14
8	Lab 15	Experiment 15
8	Lab 16	Experiment 16
9	Lab 17	Experiment 17
9	Lab 18	Practice/Viva

List of Experiments

- Experiment 1: File/Directory Commands
- Experiment 2: vi Editor
- Experiment 3: File Permissions

- Experiment 4: Process Commands
- Experiment 5: Memory Commands
- Experiment 6: System Calls
- Experiment 7: Debugging Commands
- Experiment 8: Basic Process Program
- Experiment 9: Zombie & Orphan Process
- Experiment 10: One-way Pipe
- Experiment 11: Two-way Pipe
- Experiment 12: One-way FIFO
- Experiment 13: Two-way FIFO
- Experiment 14: Semaphore
- Experiment 15: Thread Basics
- Experiment 16: Mutex Synchronization
- Experiment 17: Semaphore Synchronization

1. Introduction to Linux

Linux is a free and open-source Unix-like operating system developed around the Linux kernel. It is widely used in servers, cloud computing, embedded systems, supercomputers, and software development. It supports multitasking, multiuser operations, security, portability, and networking.

Features of Linux

- Open-source operating system
- Multiuser and multitasking support
- Portable across hardware platforms
- Strong security and permissions
- Hierarchical file system
- Powerful command-line interface
- Networking and shell scripting support

Linux Directory Structure

/ Root directory
/bin Essential user commands
/boot Boot loader files
/dev Device files
/etc Configuration files
/home User home directories
/lib Shared libraries
/media Mount point for removable devices
/opt Optional software
/proc Process information
/root Root user's home
/tmp Temporary files
/usr User programs
/var Variable data such as logs

Ubuntu Installation (Summary)

- Download Ubuntu LTS ISO.
- Create a bootable USB using Rufus/Balena Etcher.
- Boot from the USB drive.
- Select 'Install Ubuntu'.

- Choose language, keyboard, and time zone.
- Create user account and password.
- Complete installation and reboot.

Useful Linux Commands

Command	Purpose	Example
pwd	Show current directory	pwd
ls	List files	ls -l
cd	Change directory	cd Documents
mkdir	Create directory	mkdir OSLab
rmdir	Remove empty directory	rmdir Test
touch	Create file	touch file.txt
cp	Copy files	cp a.txt b.txt
mv	Move/Rename	mv old.txt new.txt
rm	Delete file	rm file.txt
cat	Display file	cat notes.txt
head	First lines	head file.txt
tail	Last lines	tail file.txt
find	Search files	find . -name '*.c'
grep	Search text	grep main demo.c

Experiment 1

Title: Study of File and Directory Handling Commands

Aim:

To learn and execute basic Linux file and directory handling commands.

Learning Outcomes

- Create, rename and delete files/directories.
- Navigate Linux directory hierarchy.
- Use basic file management commands.

Theory

Linux provides numerous commands to manage files and directories. Understanding these commands is essential before performing advanced operating system tasks.

Commands to Perform

- pwd
- ls
- ls -l
- mkdir OS_Lab
- cd OS_Lab

- touch file1.txt
- cp file1.txt file2.txt
- mv file2.txt sample.txt
- cat sample.txt
- head sample.txt
- tail sample.txt
- find . -name sample.txt
- rm sample.txt
- cd ..
- rmdir OS_Lab

Procedure
- Open Terminal.
- Execute each command one by one.
- Observe the output after each command.
- Record the observations in the lab record.

Expected Output
[Paste screenshots of terminal output for each command here.]

Result
Successfully executed Linux file and directory handling commands.

Precautions
- Be careful while using rm command.
- Verify current directory before deleting files.
- Use correct command syntax.

Viva Questions
1. What is Linux?

2. Difference between pwd and ls?

3. What is the purpose of mkdir?

4. What does rm command do?

5. What is the use of find command?

Assignment
- Create a directory structure for your semester subjects.
- Differentiate cp and mv with examples.
- Write five commonly used Linux commands with syntax.

Experiment 2: Study of vi Editor Commands in Linux

Aim
Acquire hands-on experience with the vi editor for creating and editing text files.

Course Outcome Mapping
CO1

Learning Outcomes
- Understand the concept involved in this experiment.
- Execute Linux commands correctly.
- Interpret the output generated.

Theory
The vi editor is a standard text editor available in Unix/Linux systems. It operates in three modes: Command Mode, Insert Mode, and Last-Line Mode. It is widely used for editing configuration files and source code.

Procedure / Commands
- vi sample.txt
- Press i to enter Insert Mode.
- Type some text and press Esc.
- :w (save the file)
- :q (quit)
- :wq (save and quit)
- :q! (quit without saving)
- yy (copy line)
- dd (delete line)
- p (paste line)
- /word (search text)

Expected Output
[Paste screenshots of terminal output here.]

Result
The experiment was performed successfully.

Precautions
- Execute commands carefully.
- Verify syntax before execution.

- Use the Linux manual pages (man command) whenever required.

Viva Questions

1. What are the three modes of the vi editor?

2. How do you save a file in vi?

3. What is the difference between :q and :q! ?

4. How do you delete a complete line?

5. What command is used to search text?

Assignment

- Create a file using vi and write your resume.
- List ten frequently used vi commands.
- Differentiate vi and nano editors.

Experiment 3: Study of File Access Permissions and Users

Aim
Understand Linux file permissions, ownership, and user management.

Course Outcome Mapping
CO1

Learning Outcomes
- Understand the concept involved in this experiment.
- Execute Linux commands correctly.
- Interpret the output generated.

Theory
Linux provides a permission model based on three types of users: Owner, Group, and Others. Each category can have Read (r), Write (w), and Execute (x) permissions. Permissions are modified using the chmod command, while ownership can be changed using chown and chgrp.

Procedure / Commands
- ls -l
- chmod 755 file.txt
- chmod u+x file.txt
- chmod g-w file.txt
- chmod o-r file.txt
- chown username file.txt
- chgrp groupname file.txt
- id
- whoami
- groups

Expected Output
[Paste screenshots of terminal output here.]

Result
The experiment was performed successfully.

Precautions
- Execute commands carefully.
- Verify syntax before execution.
- Use the Linux manual pages (man command) whenever required.

Viva Questions

1. What do r, w, and x represent?

2. Who are Owner, Group, and Others?

3. What is the purpose of chmod?

4. What does chmod 755 mean?

5. What is the difference between chown and chgrp?

Assignment

- Explain symbolic and numeric permission methods.
- Set permissions for a shell script so only the owner can edit it.
- Compare chmod 644 and chmod 755.

Experiment 4: Study of Process Related Commands

Aim
Learn Linux commands used to view and manage processes.

Course Outcome Mapping
CO2

Learning Outcomes
- Understand the concept involved in this experiment.
- Execute Linux commands correctly.
- Interpret the output generated.

Theory
A process is a program in execution. Linux provides several commands to inspect, monitor, and control running processes. Administrators use these commands for resource management and troubleshooting.

Procedure / Commands
- ps
- ps -ef
- top
- htop (if installed)
- jobs
- bg
- fg
- kill <PID>
- kill -9 <PID>
- nice -n 10 command
- renice 5 -p <PID>

Expected Output
[Paste screenshots of terminal output here.]

Result
The experiment was performed successfully.

Precautions
- Execute commands carefully.
- Verify syntax before execution.
- Use the Linux manual pages (man command) whenever required.

Viva Questions

1. What is a process?

2. What is the difference between ps and top?

3. What is the purpose of kill -9?

4. What are foreground and background processes?

5. What does the nice command do?

Assignment

- Monitor CPU usage using top and note the highest process.
- Terminate a background process using kill.
- Differentiate kill and kill -9.

Experiment 5: Study of Commands Related to Memory Allocation of a Process

Aim
Study Linux commands used to monitor process memory utilization.

Course Outcome Mapping
CO2

Learning Outcomes
- Understand the underlying operating system concept.
- Execute Linux commands/programs correctly.
- Interpret the observed output.

Theory
Linux provides utilities to inspect memory usage of the system and individual processes. These commands help understand virtual memory, physical memory, swap space, and memory maps.

Procedure / Commands
- free -h
- vmstat
- top
- htop (if installed)
- ps -o pid,vsz,rss,comm
- pmap <PID>
- cat /proc/meminfo
- cat /proc/<PID>/status
- cat /proc/<PID>/maps

Expected Output
[Paste screenshots of terminal/program output here.]

Result
The experiment was executed successfully and the objectives were achieved.

Precautions
- Verify command syntax before execution.
- Use 'man' pages whenever required.
- Avoid executing commands as root unless necessary.

Viva Questions

1. What is virtual memory?

2. What is the difference between VSZ and RSS?

3. What information does pmap display?

4. What is swap memory?

5. What is the purpose of /proc/meminfo?

Assignment

- Compare free and vmstat outputs.
- Observe memory usage of three running processes.
- Write a note on the /proc filesystem.

Experiment 6: Study of Commands for Viewing System Calls

Aim
Learn how Linux system calls can be observed and analyzed.

Course Outcome Mapping
CO2

Learning Outcomes
- Understand the underlying operating system concept.
- Execute Linux commands/programs correctly.
- Interpret the observed output.

Theory
System calls provide the interface between user programs and the Linux kernel. Utilities such as strace and ltrace help trace system calls and library calls made by applications.

Procedure / Commands
- strace ls
- strace cat file.txt
- strace -o output.txt ls
- ltrace ls (if installed)
- man 2 open
- man 2 read
- man 2 write
- man 2 fork
- man 2 execve

Expected Output
[Paste screenshots of terminal/program output here.]

Result
The experiment was executed successfully and the objectives were achieved.

Precautions
- Verify command syntax before execution.
- Use 'man' pages whenever required.
- Avoid executing commands as root unless necessary.

Viva Questions
1. What is a system call?

2. What is the difference between strace and ltrace?

3. Which manual section contains system calls?

4. What does execve() do?

5. Why are system calls required?

Assignment

- Trace the execution of the pwd command using strace.
- Study open(), read(), and write() system calls.
- Differentiate library functions and system calls.

Experiment 7: Study of Linux Debugging Commands

Aim
Understand basic debugging tools used for Linux applications.

Course Outcome Mapping
CO2

Learning Outcomes
- Understand the underlying operating system concept.
- Execute Linux commands/programs correctly.
- Interpret the observed output.

Theory
Debugging is the process of identifying and correcting errors in programs. Linux provides tools such as gdb, gcc debug symbols, objdump, nm, addr2line, and dmesg for debugging and analysis.

Procedure / Commands
- gcc -g program.c -o program
- gdb ./program
- break main
- run
- next
- step
- print variable
- backtrace
- continue
- quit
- dmesg | tail
- objdump -d program

Expected Output
[Paste screenshots of terminal/program output here.]

Result
The experiment was executed successfully and the objectives were achieved.

Precautions
- Verify command syntax before execution.
- Use 'man' pages whenever required.
- Avoid executing commands as root unless necessary.

Viva Questions

1. What is GDB?

2. What is the purpose of the -g option in GCC?

3. What is the difference between step and next?

4. What is a breakpoint?

5. What does backtrace display?

Assignment

- Debug a simple C program using GDB.
- List five commonly used GDB commands.
- Explain the purpose of objdump.

Experiment 8: Basic Operations of a Process using fork()

Aim
Demonstrate creation of a child process using fork().

Course Outcome Mapping
CO2

Theory
fork() creates a new child process. The parent and child execute independently after the fork call.

Algorithm
- Create a C source file.
- Call fork().
- Print parent and child information.
- Compile and execute the program.

Program
```
#include <stdio.h>
#include <unistd.h>
int main(){
 pid_t pid=fork();
 if(pid==0)
   printf("Child Process: PID=%d\n",getpid());
 else
   printf("Parent Process: PID=%d Child=%d\n",getpid(),pid);
 return 0;
}
```

Compilation & Execution
```
gcc process.c -o process
./process
```

Sample Output
Parent Process: PID=1234 Child=1235
Child Process: PID=1235

Result
Program executed successfully.

Viva Questions

1. What is a process?

2. What is fork()?

3. What value does fork() return?

4. Can parent and child execute simultaneously?

5. How are PIDs assigned?

Assignment

- Modify the program to display PPID.
- Create two child processes using fork().

Experiment 9: Creation of Zombie and Orphan Processes

Aim
Study zombie and orphan processes in Linux.

Course Outcome Mapping
CO2

Theory
A zombie process has completed execution but still has an entry in the process table. An orphan process continues execution after its parent terminates and is adopted by init/systemd.

Algorithm
- Write the program.
- Compile the program.
- Run the program.
- Observe the process table using ps -el.

Program
```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(){
 pid_t pid=fork();
 if(pid==0){
   printf("Child exiting...\n");
   exit(0);
 }else{
   sleep(10);
   printf("Parent exiting.\n");
 }
 return 0;
}
```

Compilation & Execution
gcc zombie.c -o zombie
./zombie
ps -el

Sample Output
Observe the process state as Z (Zombie) using ps.

Result
Program executed successfully.

Viva Questions

1. What is a zombie process?

2. What is an orphan process?

3. How can zombies be avoided?

4. Which process adopts orphans?

5. What does wait() do?

Assignment

- Write a separate program to demonstrate an orphan process.
- Study wait() and waitpid().

Experiment 10: One-Way Pipe Between Two Processes

Aim
Demonstrate one-way communication using a pipe.

Course Outcome Mapping
CO2

Theory
A pipe provides unidirectional communication between related processes. One end is used for writing and the other for reading.

Algorithm
- Create a pipe using pipe().
- Create a child using fork().
- Parent writes to the pipe.
- Child reads and displays the message.

Program
```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(){
 int fd[2]; char msg[]="Hello Child"; char buf[50];
 pipe(fd);
 if(fork()==0){
   read(fd[0],buf,sizeof(buf));
   printf("Child received: %s\n",buf);
 }else{
   write(fd[1],msg,strlen(msg)+1);
 }
 return 0;
}
```

Compilation & Execution
```
gcc pipe.c -o pipe
./pipe
```

Sample Output
Child received: Hello Child

Result
Program executed successfully.

Viva Questions

1. What is a pipe?

2. Is a pipe unidirectional?

3. Why is fork() used with pipes?

4. What does pipe() return?

5. Name another IPC mechanism.

Assignment

- Modify the program to send a user-entered string.
- Compare pipe and FIFO.

Experiment 11: Two-Way Pipe Between Two Processes

Aim
Demonstrate bidirectional communication using two unnamed pipes.

Course Outcome Mapping
CO3

Theory
Two unnamed pipes are required for bidirectional communication because a single pipe supports only one-way data transfer.

Algorithm
- Create two pipes.
- Create child process using fork().
- Parent writes to first pipe.
- Child reads and replies using second pipe.
- Parent reads the reply.

Program
```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
int main(){
 int p1[2],p2[2]; char buf[50];
 pipe(p1); pipe(p2);
 if(fork()==0){
   read(p1[0],buf,sizeof(buf));
   printf("Child: %s\n",buf);
   write(p2[1],"Hello Parent",13);
 }else{
   write(p1[1],"Hello Child",12);
   read(p2[0],buf,sizeof(buf));
   printf("Parent: %s\n",buf);
 }
 return 0;
}
```

Compilation & Execution
```
gcc twoway_pipe.c -o twoway_pipe
./twoway_pipe
```

Sample Output

Child: Hello Child
Parent: Hello Parent

Result

The program executed successfully and demonstrated the required IPC mechanism.

Viva Questions

1. Why are two pipes required?

2. Is an unnamed pipe full duplex?

3. What does pipe() return?

4. Can unrelated processes use unnamed pipes?

5. Differentiate pipe and socket.

Assignment

- Modify the program to exchange user-entered messages.
- Compare one-way and two-way pipes.

Experiment 12: One-Way Communication Using FIFO

Aim
Implement one-way inter-process communication using a named pipe (FIFO).

Course Outcome Mapping
CO3

Theory
A FIFO (First In First Out) is a named pipe that allows unrelated processes to communicate through the filesystem.

Algorithm
- Create FIFO using mkfifo().
- Open FIFO in write mode.
- Write a message.
- Read the message from another process.

Program
```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(){
 mkfifo("myfifo",0666);
 int fd=open("myfifo",O_WRONLY);
 write(fd,"Hello FIFO",11);
 close(fd);
 return 0;
}
```

Compilation & Execution
```
gcc fifo_writer.c -o fifo_writer
./fifo_writer
```

Sample Output
Reader Process: Hello FIFO

Result
The program executed successfully and demonstrated the required IPC mechanism.

Viva Questions
1. What is FIFO?

2. How is FIFO different from a pipe?

3. Which function creates a FIFO?

4. Can unrelated processes use FIFO?

5. Where is a FIFO stored?

Assignment

- Write a FIFO reader program.
- Exchange a string entered by the user.

Experiment 13: Two-Way Communication Using FIFO

Aim
Demonstrate full-duplex communication using two FIFOs.

Course Outcome Mapping
CO3

Theory
Two named pipes are created to enable bidirectional communication between independent processes.

Algorithm
- Create fifo1 and fifo2.
- Run Process 1.
- Run Process 2.
- Exchange messages through both FIFOs.
- Close and remove FIFOs.

Program
```
/* Process 1 */
mkfifo("fifo1",0666);
mkfifo("fifo2",0666);
/* One process writes to fifo1 and reads from fifo2.
   The second process reads fifo1 and writes fifo2,
   enabling two-way communication. */
```

Compilation & Execution
```
gcc process1.c -o p1
gcc process2.c -o p2
./p1
./p2
```

Sample Output
Process 1: Hello
Process 2: Welcome

Result
The program executed successfully and demonstrated the required IPC mechanism.

Viva Questions
1. Why are two FIFOs needed?

2. What is full-duplex communication?

3. How do you remove a FIFO?

4. Name another IPC mechanism.

5. What happens if no reader is available?

Assignment

- Implement a simple chat application using FIFOs.
- Compare FIFO and message queues.

Experiment 14: Process Synchronization using Semaphore

Aim
Demonstrate synchronization using a binary semaphore.

Course Outcome Mapping
CO4

Theory
Semaphores are synchronization primitives used to protect critical sections and coordinate concurrent execution.

Algorithm
- Initialize semaphore.
- Create concurrent threads/processes.
- Enter critical section using sem_wait().
- Leave using sem_post().
- Destroy semaphore.

Program
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem;
void* task(void *arg){
   sem_wait(&sem);
   printf("Critical Section\n");
   sem_post(&sem);
   return NULL;
}
int main(){
   pthread_t t1,t2;
   sem_init(&sem,0,1);
   pthread_create(&t1,NULL,task,NULL);
   pthread_create(&t2,NULL,task,NULL);
   pthread_join(t1,NULL);
   pthread_join(t2,NULL);
   sem_destroy(&sem);
   return 0;
}
```

Compilation & Execution

```
gcc semaphore.c -o semaphore -pthread
./semaphore
```

Sample Output

```
Critical Section
Critical Section
```

Result

The program executed successfully and demonstrated the required synchronization concept.

Precautions

- Compile with the pthread library when required.
- Initialize synchronization primitives correctly.
- Destroy mutexes and semaphores after use.

Viva Questions

1. What is a semaphore?

2. Difference between binary and counting semaphore?

3. What is a critical section?

4. Why use sem_wait()?

5. What is mutual exclusion?

Assignment

- Implement a counting semaphore.
- Modify the program for producer-consumer synchronization.

Experiment 15: Basic Operations of Thread

Aim
Create and execute a POSIX thread.

Course Outcome Mapping
CO4

Theory
Threads are lightweight units of execution within the same process. They share memory while executing independently.

Algorithm
- Define thread function.
- Create thread using pthread_create().
- Wait using pthread_join().

Program
```
#include <stdio.h>
#include <pthread.h>
void* worker(void *arg){
   printf("Thread Executing\n");
   return NULL;
}
int main(){
   pthread_t t;
   pthread_create(&t,NULL,worker,NULL);
   pthread_join(t,NULL);
   return 0;
}
```

Compilation & Execution
```
gcc thread.c -o thread -pthread
./thread
```

Sample Output
Thread Executing

Result
The program executed successfully and demonstrated the required synchronization concept.

Precautions
- Compile with the pthread library when required.
- Initialize synchronization primitives correctly.

- Destroy mutexes and semaphores after use.

Viva Questions

1. What is a thread?

2. Difference between process and thread?

3. Why use pthread_join()?

4. What does pthread_create() return?

5. What resources are shared among threads?

Assignment

- Create three threads.
- Display thread IDs using pthread_self().

Experiment 16: Thread Synchronization using Mutex

Aim

Synchronize multiple threads using a mutex lock.

Course Outcome Mapping

CO4

Theory

A mutex ensures that only one thread can enter the critical section at a time, preventing race conditions.

Algorithm

- Initialize mutex.
- Lock before critical section.
- Unlock after execution.
- Destroy mutex.

Program

```
#include <stdio.h>
#include <pthread.h>
pthread_mutex_t lock;
int count=0;
void* worker(void *arg){
  pthread_mutex_lock(&lock);
  count++;
  printf("Count=%d\n",count);
  pthread_mutex_unlock(&lock);
  return NULL;
}
int main(){
 pthread_t t1,t2;
 pthread_mutex_init(&lock,NULL);
 pthread_create(&t1,NULL,worker,NULL);
 pthread_create(&t2,NULL,worker,NULL);
 pthread_join(t1,NULL); pthread_join(t2,NULL);
 pthread_mutex_destroy(&lock);
}
```

Compilation & Execution

```
gcc mutex.c -o mutex -pthread
./mutex
```

Sample Output

Count=1
Count=2

Result

The program executed successfully and demonstrated the required synchronization concept.

Precautions

- Compile with the pthread library when required.
- Initialize synchronization primitives correctly.
- Destroy mutexes and semaphores after use.

Viva Questions

1. What is a mutex?

2. What is a race condition?

3. Difference between mutex and semaphore?

4. Why initialize a mutex?

5. When is pthread_mutex_unlock() called?

Assignment

- Modify the program using five threads.
- Observe behavior after removing the mutex.

Experiment 17: Thread Synchronization using Semaphore

Aim
Implement thread synchronization using POSIX semaphores.

Course Outcome Mapping
CO4

Theory
Semaphores can also synchronize threads by controlling access to shared resources.

Algorithm
- Initialize semaphore.
- Create threads.
- Protect critical section with sem_wait()/sem_post().
- Destroy semaphore.

Program
```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
sem_t sem;
void* task(void *arg){
 sem_wait(&sem);
 printf("Thread in Critical Section\n");
 sem_post(&sem);
 return NULL;
}
int main(){
 pthread_t t1,t2;
 sem_init(&sem,0,1);
 pthread_create(&t1,NULL,task,NULL);
 pthread_create(&t2,NULL,task,NULL);
 pthread_join(t1,NULL); pthread_join(t2,NULL);
 sem_destroy(&sem);
}
```

Compilation & Execution
```
gcc thread_sem.c -o thread_sem -pthread
./thread_sem
```

Sample Output
```
Thread in Critical Section
Thread in Critical Section
```

Result
The program executed successfully and demonstrated the required synchronization concept.

Precautions
- Compile with the pthread library when required.
- Initialize synchronization primitives correctly.
- Destroy mutexes and semaphores after use.

Viva Questions
1. Can semaphores synchronize threads?

2. What is the purpose of sem_init()?

3. Difference between mutex and binary semaphore?

4. What is starvation?

5. What is deadlock?

Assignment
- Implement producer-consumer using semaphores.
- Compare mutex and semaphore with examples.

1. Frequently Asked Viva Questions

Linux Basics

1. What is an operating system?

2. What is Linux?

3. Differentiate Linux and Unix.

4. What is the Linux kernel?

5. What is a shell?

6. Difference between CLI and GUI?

7. What is the root directory?

8. What is the purpose of /home?

9. What is the use of man command?

10. What is a terminal?

Processes

1. What is a process?

2. What is a process state?

3. Difference between process and program?

4. What is PID?

5. What is fork()?

6. What is exec()?

7. What is wait()?

8. What is a zombie process?

9. What is an orphan process?

10. What is context switching?

IPC

1. What is IPC?

2. Name IPC mechanisms.

3. What is a pipe?

4. Difference between pipe and FIFO?

5. What is shared memory?

6. What is a message queue?

7. Why are two pipes required for two-way communication?

8. What is a socket?

9. What is synchronization?

10. What is a critical section?

Threads & Synchronization

1. What is a thread?

2. Difference between thread and process?

3. What is pthread?

4. What is a mutex?

5. What is a semaphore?

6. Difference between mutex and semaphore?

7. What is deadlock?

8. What is starvation?

9. What is race condition?

10. What is mutual exclusion?

2. Model Practical Examination Questions

- Execute file and directory handling commands.
- Demonstrate file permission commands.
- Create a child process using fork().
- Demonstrate a zombie process.
- Implement one-way pipe communication.
- Implement two-way FIFO communication.

- Demonstrate synchronization using mutex.
- Create two POSIX threads.
- Implement semaphore synchronization.
- Trace system calls using strace.

3. Linux Command Cheat Sheet

Command	Description	Example
pwd	Present working directory	pwd
ls -l	Long listing	ls -l
cd	Change directory	cd /home
mkdir	Create directory	mkdir test
rm	Remove file	rm file.txt
cp	Copy file	cp a b
mv	Move/Rename	mv a b
chmod	Change permissions	chmod 755 file
ps -ef	List processes	ps -ef
top	Monitor processes	top
kill	Terminate process	kill PID
free -h	Memory usage	free -h
gcc	Compile C program	gcc demo.c -o demo
gdb	Debugger	gdb demo
man	Manual page	man ls

4. Frequently Used System Calls

- fork()
- exec()
- wait()
- exit()
- open()
- read()
- write()
- close()
- pipe()
- mkfifo()
- pthread_create()
- pthread_join()

5. Compilation Commands Reference

- gcc program.c -o program
- gcc -g program.c -o program

- gcc program.c -o program -pthread
- ./program

6. Common Errors and Troubleshooting

Error	Possible Solution
Permission denied	Use chmod or execute with proper permissions.
command not found	Check PATH or install the package.
Segmentation fault	Check pointer initialization and array bounds.
Undefined reference to pthread	Compile using -pthread.
Broken pipe	Ensure the reader process is active.

7. Practical Record Format

Experiment No.: _____

Date: _____

Aim:

Commands / Program:

Output:

Result:

Faculty Signature: _____

8. References

- A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts.
- Andrew S. Tanenbaum, Modern Operating Systems.
- William Stallings, Operating Systems: Internals and Design Principles.
- Linux Manual Pages (man).
- GNU GCC and GDB Documentation.

9. Glossary

- Kernel: Core component of an operating system.
- Process: Program in execution.
- Thread: Lightweight execution unit within a process.
- Semaphore: Synchronization primitive.
- Mutex: Mutual exclusion lock.

- FIFO: Named pipe used for IPC.
- IPC: Inter-Process Communication.
- Context Switch: Switching CPU from one process/thread to another.

Appendix A: Linux Keyboard Shortcuts

Shortcut	Purpose
Ctrl + C	Terminate the running process
Ctrl + Z	Suspend the running process
Ctrl + D	Logout / End of input
Ctrl + L	Clear the terminal screen
Ctrl + R	Search command history
Tab	Auto-complete commands and filenames
Up/Down Arrow	Browse command history
Ctrl + A	Move cursor to beginning of line
Ctrl + E	Move cursor to end of line
Ctrl + U	Delete from cursor to beginning of line

Appendix B: Additional Useful Linux Commands

- history
- clear
- who
- whoami
- uname -a
- hostname
- date
- cal
- df -h
- du -sh
- tar
- zip
- unzip
- ping
- ifconfig / ip addr
- netstat / ss
- passwd
- echo
- wc
- sort
- uniq

Appendix C: Mini Practice Exercises

- Create a directory named OS_Lab and five subdirectories.
- Create three text files and copy them into another directory.
- Assign 755 permission to a shell script and execute it.
- Write a program to create two child processes.
- Display all running processes sorted by memory usage.
- Trace the system calls of the ls command using strace.
- Implement producer-consumer using semaphores.
- Write a multithreaded program to calculate the sum of an array.
- Compare the output of top and ps -ef.
- Demonstrate page information using the /proc filesystem.

Appendix D: Mini Project Ideas

- Linux System Monitoring Dashboard
- CPU Scheduling Algorithm Simulator
- Memory Management Simulator
- Deadlock Detection Simulator
- Producer–Consumer Simulation
- Dining Philosophers Problem
- Banker's Algorithm Simulator
- File Permission Management Tool
- Mini Linux Shell in C
- Thread-Based Matrix Multiplication

Appendix E: Practical Assessment Rubric

Assessment Parameter	Marks
Attendance & Preparation	5
Program Logic / Commands	10
Execution & Output	10
Record Book	5
Viva-Voce	10
Total	40

Appendix F: Learning Resources

- Official Linux Documentation
- GNU GCC Documentation
- GDB User Manual
- POSIX Threads Documentation

- Operating System Concepts by Silberschatz
- Modern Operating Systems by Andrew S. Tanenbaum
- Operating Systems: Internals and Design Principles by William Stallings

Faculty Remarks

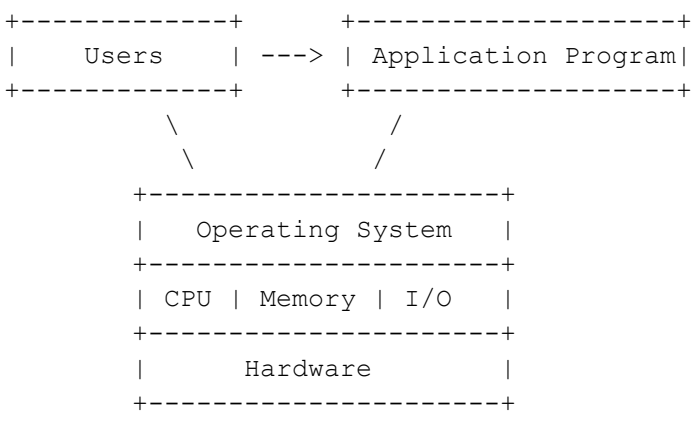
Student Name: _____
Roll No.: _____
Batch: _____

Faculty Remarks:

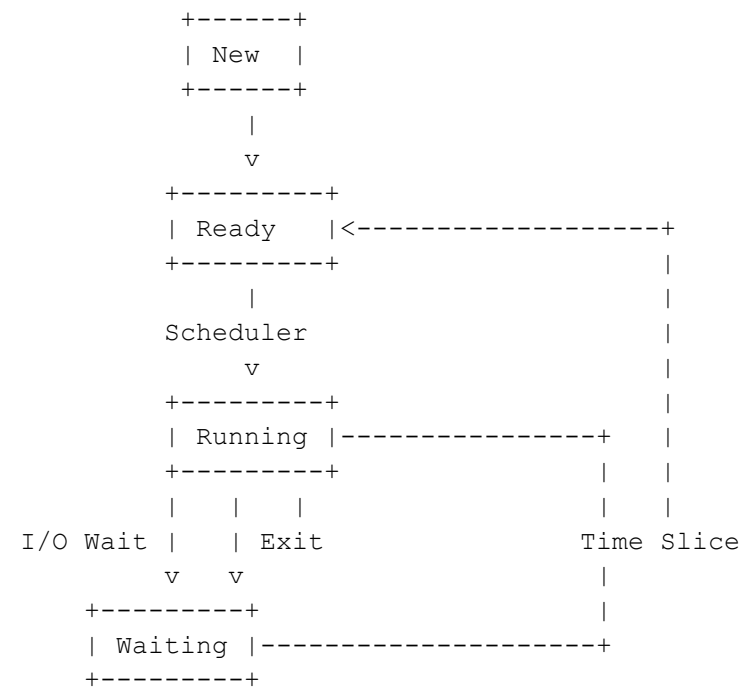
_____

Faculty Signature: _____

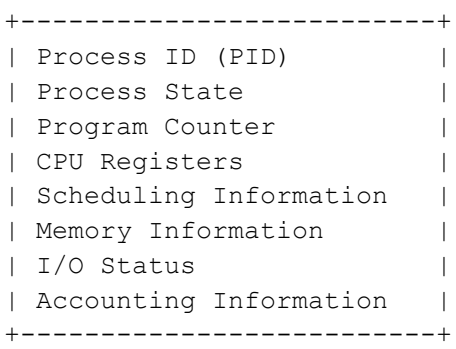
1. Computer System Architecture



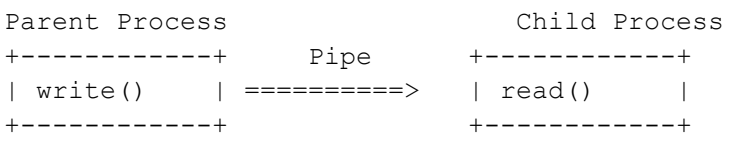
2. Process State Diagram



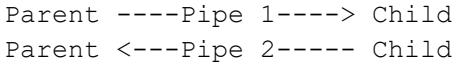
3. Process Control Block (PCB)



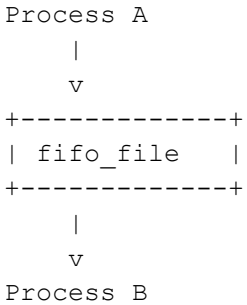
4. One-Way Pipe Communication



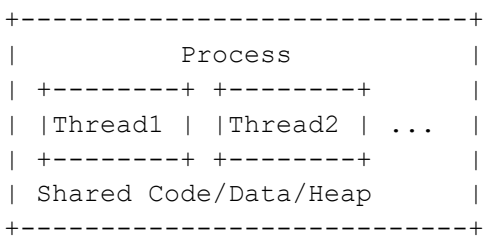
5. Two-Way Pipe Communication



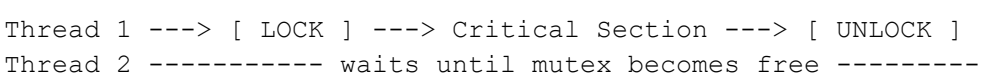
6. FIFO Communication



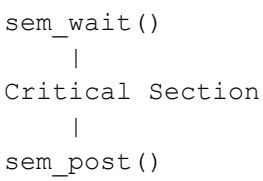
7. Thread Model



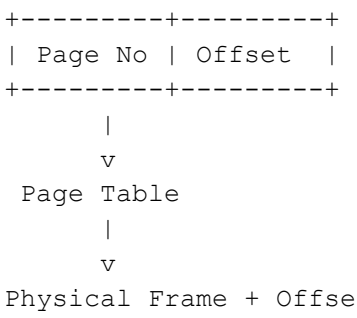
8. Mutex Synchronization



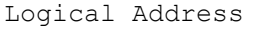
9. Semaphore Synchronization

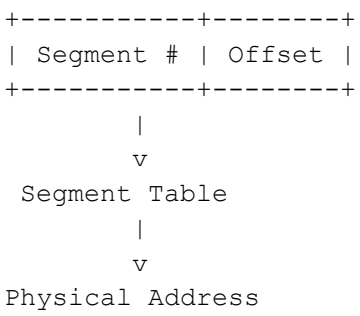


10. Paging

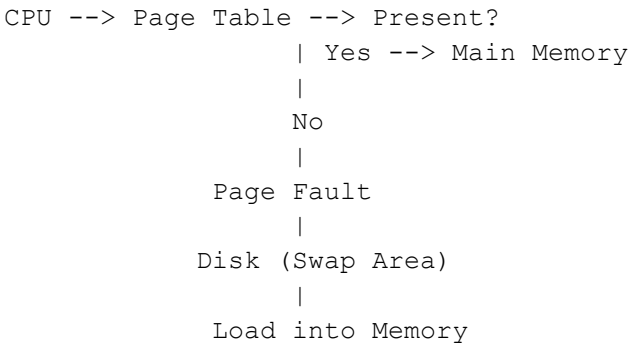


11. Segmentation

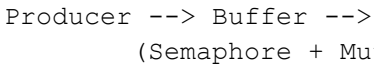




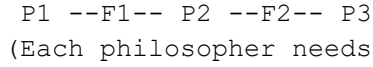
12. Demand Paging



13. Producer-Consumer



14. Dining Philosophers



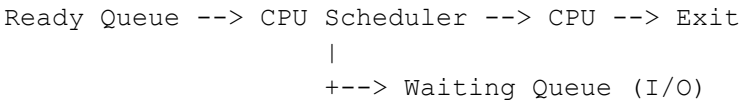
(Each philosopher needs left and right fork.)

15. Deadlock Conditions

Necessary conditions:

- Mutual Exclusion
- Hold and Wait
- No Preemption
- Circular Wait

16. CPU Scheduling Overview



Purpose
This section provides representative terminal outputs and placeholders where students can paste screenshots captured during laboratory sessions.

pwd
Command:
pwd

Sample Output:
$ pwd
/home/student

[Paste your terminal screenshot here after executing the command.]

ls -l
Command:
ls -l

Sample Output:
$ ls -l
-rw-r--r-- file1.txt
drwxr-xr-x Documents

[Paste your terminal screenshot here after executing the command.]

mkdir OS_Lab
Command:
mkdir OS_Lab

Sample Output:
$ mkdir OS_Lab

[Paste your terminal screenshot here after executing the command.]

chmod 755 script.sh
Command:
chmod 755 script.sh

Sample Output:
$ chmod 755 script.sh

[Paste your terminal screenshot here after executing the command.]

ps -ef
Command:
ps -ef

Sample Output:
UID PID PPID CMD
root 1 0 systemd

[Paste your terminal screenshot here after executing the command.]

top
Command:
top

Sample Output:
top - 10:20:15 ...
PID USER %CPU %MEM COMMAND

[Paste your terminal screenshot here after executing the command.]

free -h
Command:
free -h

Sample Output:
Mem: 7.6Gi Used 2.1Gi Free 4.8Gi

[Paste your terminal screenshot here after executing the command.]

vmstat
Command:
vmstat

Sample Output:
procs --------memory-------- ...

[Paste your terminal screenshot here after executing the command.]

strace ls
Command:
strace ls

Sample Output:
execve("/usr/bin/ls", ...) = 0

[Paste your terminal screenshot here after executing the command.]

gdb ./program
Command:
gdb ./program

Sample Output:
(gdb) break main
(gdb) run

[Paste your terminal screenshot here after executing the command.]

fork() Program
Command:
fork() Program

Sample Output:
Parent Process: PID=1200
Child Process: PID=1201

[Paste your terminal screenshot here after executing the command.]

Zombie Process
Command:
Zombie Process

Sample Output:
$ ps -el
... Z ...

[Paste your terminal screenshot here after executing the command.]

Pipe Program
Command:
Pipe Program

Sample Output:
Child received: Hello Child

[Paste your terminal screenshot here after executing the command.]

FIFO Program
Command:
FIFO Program

Sample Output:
Reader: Hello FIFO

[Paste your terminal screenshot here after executing the command.]

Thread Program
Command:
Thread Program

Sample Output:
Thread Executing

[Paste your terminal screenshot here after executing the command.]

Mutex Program
Command:
Mutex Program

Sample Output:
Count=1
Count=2

[Paste your terminal screenshot here after executing the command.]

Semaphore Program
Command:
Semaphore Program

Sample Output:
Critical Section
Critical Section

[Paste your terminal screenshot here after executing the command.]

Screenshot Submission Checklist

1. Screenshot clearly shows the executed command.
2. Prompt and output are visible.
3. Student name/roll number written in the record.
4. Screenshot corresponds to the experiment.
5. Output is readable and not cropped.

Faculty Verification

Experiment No.: _____
Date: _____
Verified By: _____
Faculty Signature: _____

Linux Fundamentals

Q. What is Linux?

Ans. Linux is an open-source, Unix-like operating system based on the Linux kernel.

Q. What is the kernel?

Ans. The kernel is the core component that manages hardware, memory, processes and devices.

Q. What is a shell?

Ans. A shell is a command-line interpreter that accepts user commands.

Q. Difference between CLI and GUI?

Ans. CLI uses text commands, while GUI provides graphical interaction.

Q. What is the root directory?

Ans. The root directory (/) is the top-level directory of the Linux file system.

Processes

Q. What is a process?

Ans. A process is a program in execution.

Q. What is PID?

Ans. PID is the unique Process Identifier assigned to each process.

Q. What does fork() do?

Ans. fork() creates a new child process.

Q. What is exec()?

Ans. exec() replaces the current process image with a new program.

Q. What is context switching?

Ans. It is the CPU switching from one process or thread to another.

IPC

Q. What is IPC?

Ans. Inter-Process Communication enables processes to exchange data.

Q. What is a pipe?

Ans. A pipe is a unidirectional IPC mechanism between related processes.

Q. What is FIFO?

Ans. FIFO is a named pipe that allows unrelated processes to communicate.

Q. Why are two pipes needed for two-way communication?

Ans. Because one pipe supports only one direction of communication.

Q. Name IPC mechanisms.

Ans. Pipes, FIFOs, shared memory, message queues, sockets and semaphores.

Threads & Synchronization

Q. What is a thread?

Ans. A thread is the smallest unit of CPU execution within a process.

Q. What is a mutex?

Ans. A mutex is a mutual exclusion lock used to protect critical sections.

Q. What is a semaphore?

Ans. A semaphore is a synchronization primitive used to coordinate concurrent execution.

Q. What is a race condition?

Ans. It occurs when multiple threads access shared data without proper synchronization.

Q. What is deadlock?

Ans. A deadlock occurs when processes wait indefinitely for resources held by one another.

Memory Management

Q. What is virtual memory?

Ans. Virtual memory extends main memory using secondary storage.

Q. What is paging?

Ans. Paging divides memory into fixed-size pages and frames.

Q. What is segmentation?

Ans. Segmentation divides memory into logical segments of variable size.

Q. What is thrashing?

Ans. Thrashing occurs when excessive page faults reduce system performance.

Q. What is demand paging?

Ans. Pages are loaded into memory only when required.

Rapid Revision Points

- Compile thread programs using: gcc program.c -o program -pthread
- Use man <command> to read Linux manual pages.
- Use ps, top, and htop for process monitoring.
- Use chmod to change permissions and chown to change ownership.
- Use gdb for debugging and strace for tracing system calls.
- Always release mutexes and destroy semaphores after use.

Common Practical Mistakes

- Forgetting to compile with -pthread.
- Using incorrect file permissions.
- Not checking fork() return value.
- Leaving FIFOs after program execution.
- Forgetting to close file descriptors.
- Not destroying mutexes or semaphores.

End of Lab Manual

Instructions

This section contains model practical questions commonly asked in university examinations. Students should understand the concepts and practice the commands/programs independently.

1. Create a directory structure and demonstrate file operations.

Model Approach:

- Use mkdir to create directories.
- Create files with touch.
- Copy using cp.
- Rename using mv.
- Delete using rm/rmdir.

2. Demonstrate Linux file permissions.

Model Approach:

- Display permissions using ls -l.
- Modify permissions with chmod symbolic/numeric.
- Verify the changes.

3. Create a child process using fork().

Model Approach:

- Write a C program using fork().
- Display parent and child PIDs.
- Compile using gcc.
- Execute and verify output.

4. Demonstrate a Zombie process.

Model Approach:

- Create a child process.
- Terminate child before parent.

- Observe state using ps -el.

5. Demonstrate an Orphan process.

Model Approach:

- Terminate parent before child.
- Observe adoption by init/systemd using getppid().

6. Implement one-way pipe communication.

Model Approach:

- Create pipe().
- Use fork().
- Parent writes.
- Child reads.

7. Implement two-way communication using FIFO.

Model Approach:

- Create two FIFOs.
- Open in read/write modes.
- Exchange messages.
- Remove FIFOs after execution.

8. Synchronize threads using mutex.

Model Approach:

- Initialize mutex.
- Lock critical section.
- Unlock after execution.
- Destroy mutex.

9. Synchronize threads using semaphore.

Model Approach:

- Initialize semaphore.
- Use sem_wait()/sem_post().

- Join threads.
- Destroy semaphore.

10. Trace system calls using strace.

Model Approach:

- Run strace on a command.
- Observe open/read/write calls.
- Explain observations.

Frequently Asked External Practical Viva Questions

1. Explain the difference between process and thread.

2. Why is fork() preferred for process creation?

3. Why are two pipes required for full-duplex communication?

4. Differentiate unnamed pipe and FIFO.

5. Explain the purpose of mutex and semaphore.

6. What is the role of the scheduler?

7. What is a page fault?

8. Explain demand paging.

9. What is context switching?

10. Why is synchronization necessary?

Practical Examination Tips

- Read the question carefully before coding.
- Write the algorithm before the program.
- Compile with warning flags when possible.
- Test the program with multiple inputs.
- Keep terminal output neat for screenshots.
- Be prepared to explain every system call and function used.
- Revise Linux commands and pthread APIs before the exam.

Self-Assessment Checklist

- ✓ I can use Linux file and directory commands.
- ✓ I can edit files using vi.
- ✓ I understand Linux permissions.
- ✓ I can create and manage processes.
- ✓ I can implement IPC using pipes and FIFOs.
- ✓ I can create POSIX threads.
- ✓ I can synchronize threads using mutexes and semaphores.
- ✓ I can compile, execute, and debug C programs in Linux.

1. Student Observation Record Sheet

Student Name: _____

Roll No.: _____

Semester: _____ Section: _____

Experiment No.: _____

Date: _____

Aim:

_____

Commands / Program:

_____

Observations:

_____

Output:

_____

Result:

_____

Faculty Signature: _____

2. Lab Attendance Register Format

S.No.	Roll No.	Student Name	Exp-1 to 6	Exp-7 to 12	Exp-13 to 17	Remarks
1						

2						
3						
4						
5						
6						
7						
8						
9						
10						

3. Internal Practical Assessment Rubric

Assessment Criterion	Marks
Attendance & Punctuality	5
Preparation & Observation Book	5
Program Logic / Linux Commands	10
Execution & Output	10
Debugging Ability	5
Viva-Voce	5
Total	40

4. Experiment Evaluation Sheet

Experiment	Completed (Y/N)	Faculty Initial	Remarks
Experiment 1			
Experiment 2			
Experiment 3			
Experiment 4			
Experiment 5			
Experiment 6			
Experiment 7			
Experiment 8			
Experiment 9			
Experiment 10			
Experiment 11			
Experiment 12			
Experiment 13			
Experiment 14			
Experiment 15			
Experiment 16			
Experiment 17			

5. Student Feedback Form

1. The objectives of the lab were clearly explained. Rating: 1 2 3 4 5

2. The laboratory equipment/software was adequate. Rating: 1 2 3 4 5

3. The experiments helped me understand Operating System concepts. Rating: 1 2 3 4 5

4. The manual was easy to follow. Rating: 1 2 3 4 5

5. The instructor explained concepts effectively. Rating: 1 2 3 4 5

6. I am confident in writing Linux system programs. Rating: 1 2 3 4 5

6. Course Exit Survey

What concepts did you learn best?

Answer: _____

Which experiment was most challenging?

Answer: _____

What improvements do you suggest for the laboratory?

Answer: _____

Would you recommend additional Linux practice sessions?

Answer: _____

7. Department Verification

Course Coordinator: _____

Lab In-Charge: _____

Head of Department: _____

Academic Session: _____

Remarks:

_____

Department Seal

Signature: _____

8. Final Student Completion Certificate

This is to certify that Mr./Ms. _____
Roll No. _____ has successfully completed all
Operating System Laboratory experiments prescribed in the syllabus.

Faculty Signature: _____

Lab In-Charge: _____

Head of Department: _____

Date: _____

9. Document Checklist

☐ Observation record completed
☐ All 17 experiments performed
☐ Faculty signatures obtained
☐ Internal assessment completed
☐ Viva completed
☐ Attendance requirements fulfilled
☐ Feedback form submitted

Introduction

This section provides mini projects and advanced practice activities to help students apply Operating System concepts beyond the prescribed experiments.

Mini Project 1: Linux File Manager

- Objective: Build a menu-driven file management utility.
- Features: Create, rename, copy, move and delete files/directories.
- Suggested APIs: mkdir(), rename(), remove(), system().
- Learning Outcome: File system operations.

Mini Project 2: Process Manager

- Display running processes.
- Create child processes using fork().
- Terminate processes using kill().
- Learning Outcome: Process management.

Mini Project 3: CPU Scheduling Simulator

- Implement FCFS scheduling.
- Extend to SJF and Round Robin.
- Calculate waiting and turnaround times.
- Compare algorithm performance.

Mini Project 4: Producer–Consumer Simulation

- Create producer and consumer threads.
- Use mutexes and semaphores.
- Display buffer status graphically or in text.
- Learning Outcome: Synchronization.

Mini Project 5: Memory Allocation Simulator

- Implement First Fit, Best Fit and Worst Fit.
- Compare fragmentation.
- Display allocation table.

Case Studies

1. Analyze why deadlocks occur in database servers.
2. Study process scheduling in modern Linux systems.
3. Compare Windows and Linux process management.

4. Investigate memory usage using free, vmstat and top.
5. Study thread synchronization in web servers.

Advanced Practice Programs

- Implement FCFS CPU scheduling in C.
- Implement Round Robin scheduling.
- Write a multithreaded matrix multiplication program.
- Implement Reader–Writer synchronization.
- Implement Dining Philosophers using semaphores.
- Implement Banker's Algorithm.
- Simulate Page Replacement (FIFO/LRU).

Outcome-Based Assessment

Activity	Course Outcome	Bloom Level
Mini Project	CO3, CO4	Apply/Create
Case Study	CO2	Analyze
Advanced Programming	CO4	Create
Presentation	CO5	Evaluate

Recommended Learning Resources

- The Linux Programming Interface by Michael Kerrisk
- Operating System Concepts by Silberschatz, Galvin & Gagne
- Advanced Programming in the UNIX Environment by W. Richard Stevens
- Linux man pages (man command)
- GCC, GDB and POSIX Threads documentation

Faculty Remarks

Strengths:

_____

Areas for Improvement:

_____

Faculty Signature: _____