

Operating Systems Laboratory

Course Code: CSE325

Lab Manual

Prepared by

Pushpendra Kumar Pateriya

HoD System Programming

School of Computer Science & Engineering

Lovely Professional University, Punjab, India-144411

Session: 23242

Name of the Student:

Registration Number:

Section and Group:

Roll No:



L OVELY
P ROFESSIONAL
U NIVERSITY

Contents

1	Experiment 1: Introduction to Linux Commands	8
1.1	Objective	8
1.2	Reference Material[1][3][2]	8
1.2.1	Important Linux Commands for File and Directory Management	8
1.2.2	Shell Command	8
1.2.3	Types of Shell Commands	9
1.2.4	Linux File System Hierarchy[2][3]	9
1.2.5	Paths	9
1.2.6	Linux File Permission Stack	10
1.3	Lab Exercises	11
1.4	Sample Viva Questions	11
1.5	Evaluation Parameters and Rating Scale	12
1.5.1	Student's Self Rating	12
1.5.2	Teacher's Rating Student Performance	12
2	Experiment 2: Basics of Shell Scripting	15
2.1	Objective	15
2.2	Reference Material[1][3][2]	15
2.2.1	Shell	15
2.2.2	Shell script	15
2.2.3	Shell Variables	15
2.2.4	Types of Variables	15
2.2.5	Important Environment Variables	16
2.2.6	Shell Redirections	16
2.2.7	Shell Arithmetic using expr and bc Commands	17
2.2.8	Flow Control in Shell Scripting using if Statements	17
2.2.9	Operators in Shell Scripting	18
2.2.10	Case Structure in Shell Scripting	19
2.2.11	Loops in Shell Scripting	20
2.3	Steps to Prepare and Execute a Shell Script	22
2.3.1	Lab Exercises	23
2.4	Sample Viva Questions	24
2.5	Evaluation Parameters and Rating Scale	24
2.5.1	Student's Self Rating	24
2.5.2	Teacher's Rating Student Performance	24
2.6	Instructions for the Teacher to Conduct Lab Assessment 1	25
3	Experiment 3: File Manipulation Using System Calls[1][3][2]	30
3.1	Objective	30
3.2	Reading Material	30
3.2.1	System Calls	30
3.2.2	Sample Programs	31
3.3	Lab Exercises	33
3.4	Sample Viva Questions	34
3.5	Evaluation Parameters and Ratings Scale	34

3.5.1	Student's Self Rating	34
3.5.2	Teacher's Rating Student Performance	34
4	Experiment 4: Directory Manipulation Using System Calls	39
4.1	Objective	39
4.2	Reading Material[1][3][2]	39
4.2.1	Directory Management Related System Calls	39
4.2.2	The dirent structure	39
4.3	Sample Programs	40
4.4	Lab Exercises	41
4.5	Sample Viva Questions	41
4.6	Evaluation Parameters and Rating Scale	42
4.6.1	Student's Self Rating	42
4.6.2	Teacher's Rating Student Performance	42
4.7	Instructions for the Teacher to Conduct Lab Assessment 2	42
5	Experiment 5: Process Management using System Calls	47
5.1	Objective	47
5.2	Reading Material[1][3][2]	47
5.2.1	Process management related system calls	47
5.2.2	Orphan and Zombie Processes	47
5.3	Sample Programs	48
5.4	Lab Exercises	50
5.5	Sample Viva Questions	51
5.6	Evaluation Parameters and Rating Scale	51
5.6.1	Student's Self Rating	51
5.6.2	Teacher's Rating Student Performance	51
6	Experiment 6: Creation of Multithreaded Processes using Pthread Library	56
6.1	Objective	56
6.2	Reading Material[1][3][2]	56
6.2.1	Commonly used library functions related to POSIX threads (pthread)	56
6.3	Sample Programs	56
6.4	Lab Exercises: [Attempt any 3 within the designated lab hours]	58
6.5	Sample Viva Questions	58
6.6	Evaluation Parameters and Rating Scale	59
6.6.1	Student's Self Rating	59
6.6.2	Teacher's Rating Student Performance	59
6.7	Instructions for the Teacher to Conduct Lab Assessment 3	59
7	Experiment 7: Process Synchronization using Semaphore/Mutex	65
7.1	Objective	65
7.2	Reading Material[1][3][2]	65
7.2.1	Synchronization	65
7.2.2	Race Condition	65
7.2.3	Semaphore	65
7.2.4	Mutex	66

7.3	Sample Programs	66
7.4	Lab Exercises	69
7.5	Sample Viva Questions	69
7.6	Evaluation Parameters and Rating Scale	70
7.6.1	Student's Self Rating	70
7.6.2	Teacher's Rating Student Performance	70
8	Experiment 8: Inter Process Communication (IPC)	75
8.1	Objective	75
8.2	Reading Material[1][3][2]	75
8.2.1	Interprocess communication	75
8.2.2	Pipes	75
8.2.3	Unnamed Pipes (Anonymous Pipes)	75
8.2.4	Named Pipes	76
8.2.5	Shared Memory	77
8.2.6	Sample program to demonstrate shared memory segment creation and data addition	78
8.2.7	Message Queues	79
8.2.8	Message Queue Implementation in C	79
8.3	Lab Exercises	81
8.4	Sample Viva Questions	82
8.5	Evaluation Parameters and Rating Scale	82
8.5.1	Student's Self Rating	82
8.5.2	Teacher's Rating Student Performance	83
8.6	Instructions for the Teacher to Conduct Lab Assessment 4	83

Expression of Gratitude

Dear Dr. Rajeev Sobti,
Dean & Head of School,
School of Computer Science,
Lovely Professional University

Dear Dr. Prateek Agrawal,
Deputy Dean & Coordinator of School,
School of Computer Science,
Lovely Professional University

I extend my deepest gratitude to both of you for your invaluable support and guidance throughout the creation of the **CSE325: Operating Systems Laboratory** manual. Your unwavering encouragement, expertise, and mentorship have been pivotal in crafting a comprehensive resource for our students.

Your commitment to academic excellence and dedication to our learning community have been inspirational. Your visionary leadership and continuous support have significantly enriched our educational experiences and have empowered us to produce this manual that will serve as a valuable asset for our students' education.

I am profoundly grateful for the opportunities you have provided us and the trust you have placed in our abilities. Your impactful contributions continue to shape the School of Computer Science at Lovely Professional University and nurture an environment of growth and learning.

Once again, thank you for your profound guidance and belief in our endeavors. Your support has been instrumental, and I am honored to have had the privilege of working under your guidance.

With heartfelt appreciation,
Pushpendra Kumar Pateriya
HoD, System Programming Domain, School of Computer Science & Engineering
34-207, Cabin-1, Lovely Professional University
email: pushpendra.14623@lpu.co.in
You can find me on LinkedIn: <https://www.linkedin.com/in/pushpendra-pateriya-344b5727/>

General Guidelines for Students

- **Completion of Lab Manual Exercises:** The completion of lab manual exercises contributes to the Continuous Assessment (CA) marks.
- **Assigned System Usage:** Sit according to your roll number and use the designated system consistently throughout the semester.
- **Reporting System Issues:** It is the student's responsibility to report any issues related to the designated system to the teacher promptly.
- **Use of Electronic Devices:** Use electronic devices in a professional manner, adhering to academic norms.
- **System Settings:** Avoid changing system settings without prior approval from the teacher.
- **Importance of Self-Practice:** Engage in self-practice as it is essential for understanding the concepts covered in this course.
- If you have any feedback or suggestions for each experiment, you can share them using this link: <https://forms.gle/2PPp3ThtdXY2M1FWA>.

General Guidelines for Teachers

- **Familiarize Students with Lab Equipment:** Introduce students to the lab equipment, including computers, software, and any specific tools they'll use during the experiments.
- **Explain Experiment Objectives:** Clearly explain the objectives and expected outcomes of each experiment to the students before they start working.
- **Provide Detailed Instructions:** Offer step-by-step instructions for conducting the experiments, ensuring clarity and simplicity.
- **Encourage Collaboration:** Foster an environment where students can work together, share knowledge, and assist each other in problem-solving.
- **Address Student Queries:** Be approachable and available to answer students' questions or concerns during the lab sessions.
- **Ensure Safety Measures:** Emphasize safety protocols and guidelines when handling equipment or performing experiments to prevent accidents.
- **Assist with Technical Issues:** Aid students in troubleshooting technical issues they might encounter during the experiments.
- **Evaluate and Provide Feedback:** Evaluate students' work based on predefined parameters and provide constructive feedback to enhance their learning experience.
- **Coordinate with Support Staff:** Collaborate with the IT support team to swiftly resolve technical difficulties beyond your expertise.

- **Encourage Exploration and Understanding:** Promote a deeper understanding of operating systems by encouraging students to explore beyond the prescribed experiments.
- **Maintain a Positive Learning Atmosphere:** Create a positive and supportive learning environment that encourages students to engage actively in their learning process.

Guidelines related to Continuous Practical Assessment

- The teacher will check students' lab manual regularly, sign the manual and ensure the progressive learning of the students.
- The teacher should create practical components by following the path on UMS: UMS Navigation >> Learning Management System (LMS) >> Practical Components.
- Create two components (1. J/E: Job Execution, 2. LM: Lab Manual Completion) of 50-50 marks each.

Comp. 1	Comp. 2
<input type="checkbox"/>	<input checked="" type="checkbox"/>
J/E	LM
50	50

- "J/E grading will be based on the student's performance during the CAP day, marked out of 50."
- "LM assessment will be out of 50, determined by the teacher's average overall rating from the evaluation of the previous two experiments in the lab manual. The average overall rating will be multiplied by 5."
- There will be total 4 continuous assessment practical (CAP) conducted during the semester (100 marks each).
- Best 3 out of 4 will be considered in grade calculation by the end of the semester.

Lab Experiments

1 Experiment 1: Introduction to Linux Commands

1.1 Objective

The objective of this lab experiment is to introduce students to fundamental Linux commands used for navigating the file system, managing files and directories, and performing basic system operations. By the end of this experiment, students should be familiar with commonly used commands such as `ls`, `cd`, `mkdir`, `rm`, `cp`, and `mv`, gaining a foundational understanding of the Linux command-line interface.

1.2 Reference Material[1][3][2]

1.2.1 Important Linux Commands for File and Directory Management

Command	Description	Example
<code>ls</code>	List directory contents	<code>ls -l</code>
<code>cd</code>	Change the current directory	<code>cd Documents</code>
<code>pwd</code>	Print the current working directory	<code>pwd</code>
<code>mkdir</code>	Make directories	<code>mkdir new_directory</code>
<code>rmdir</code>	Remove directories	<code>rmdir directory_to_remove</code>
<code>cp</code>	Copy files and directories	<code>cp file1.txt file2.txt</code>
<code>mv</code>	Move or rename files and directories	<code>mv file1.txt new_location</code>
<code>rm</code>	Remove files or directories	<code>rm file_to_delete.txt</code>
<code>touch</code>	Create an empty file	<code>touch new_file.txt</code>
<code>cat</code>	Display or concatenate files	<code>cat file.txt</code>
<code>head</code>	Display the beginning of a file	<code>head -n 5 file.txt</code>
<code>tail</code>	Display the end of a file	<code>tail -n 10 file.txt</code>
<code>grep</code>	Search text in files	<code>grep "pattern" file.txt</code>
<code>chmod</code>	Change file permissions	<code>chmod 644 file.txt</code>
<code>chown</code>	Change file ownership	<code>chown user:group file.txt</code>
<code>ln</code>	Create links between files	<code>ln -s /path/to/file linkname</code>

Table 1: Important Linux Commands for File and Directory Management[2]

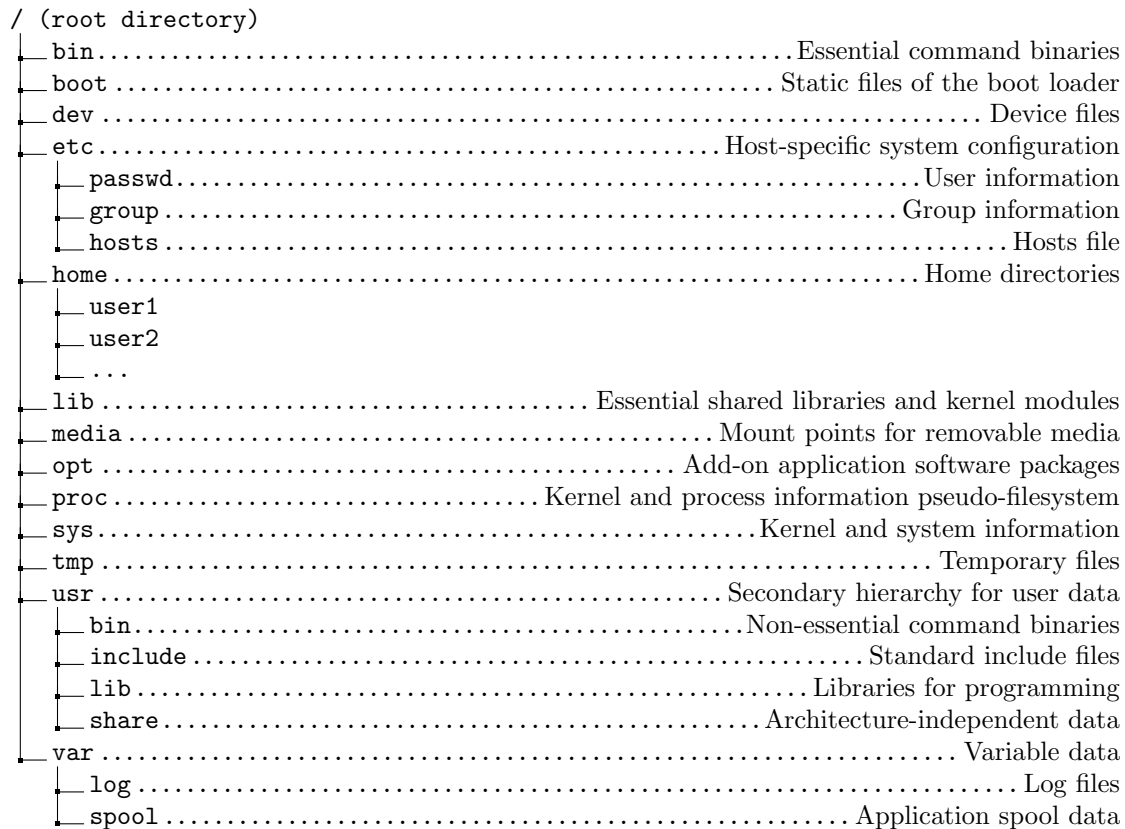
1.2.2 Shell Command

A shell command is a directive or instruction provided by a user to a shell (a command-line interpreter) in an operating system.

1.2.3 Types of Shell Commands

- Internal Commands: These commands are built into the shell itself. They are part of the shell's functionalities and do not exist as separate executable files. Examples include `cd`, `echo`, `exit`, `alias`, `export`, etc.
- External Commands: These commands are separate executable files located in directories listed in the system's `PATH` variable. When a user inputs an external command, the shell searches for the command's executable file in these directories and executes it if found.

1.2.4 Linux File System Hierarchy[2][3]



1.2.5 Paths

Paths refer to the location or address of a file or directory in the file system.

Types of paths

- (i) Absolute Path: An absolute path defines the complete location of a file or directory starting from the root directory (`/`). It includes the entire directory hierarchy from the root directory to the specific file or directory. For instance, `/home/user/documents/file.txt` is an absolute

path where the file.txt is located in the 'documents' directory inside the 'user' directory within the 'home' directory, starting from the root (/) directory.

- (ii) Relative Path: A relative path defines the location of a file or directory with respect to the current working directory. It doesn't start from the root directory but refers to a location relative to the current directory. For example, if the current directory is /home/user/, a file located in the 'documents' directory can be referenced using a relative path like documents/file.txt.

1.2.6 Linux File Permission Stack

In Linux, file and directory permissions are represented using a permission stack. The format of the permission stack is as follows:

```

- r w x   r - x   r - x
|   | |   | |   | |
|   | |   | |   | +---- Others (permissions for users not covered by owner or group)
|   | |   | +----- Group (permissions for users in the file's group)
|   | +----- Owner (permissions for the file or directory owner)
|   +----- Type of the file (e.g., - for a regular file, d for a directory)
+----- Special permission bits (e.g., s, t, etc.)

```

Each group of permissions (Owner, Group, Others) consists of three characters representing read (r), write (w), and execute (x) permissions. If a permission is allowed, the respective character is displayed, and if it's denied, a hyphen (-) is shown.

For example:

```
-rw-r--r-- 1 user group 24 Jan 7 13:20 myfile.txt
```

In this example: - The first character (-) indicates that it's a regular file. - The next three characters (rw-) represent the owner's permissions (read and write, but not execute). - The following three characters (r-) represent the group's permissions (read-only). - The last three characters (r-) represent permissions for others (read-only).

These permissions can be changed using commands like `chmod` in Linux to alter the read, write, and execute permissions for the owner, group, and others.

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

1.3 Lab Exercises

- Exercise 1:** Using the `ls` command, list all the files and directories in the current directory.
- Exercise 2:** Create a new directory named `lab_files`. Navigate into this directory and create a new file named `example.txt`.
- Exercise 3:** Use the `cp` command to make a copy of `example.txt` and name it `example.backup.txt`.
- Exercise 4:** Remove the file `example.txt` from the `lab_files` directory.
- Exercise 5:** Use the `mkdir` command to create three new directories named `folder1`, `folder2`, and `folder3` within `lab_files`.
- Exercise 6:** Explain the purpose of the `/etc` directory in the Linux file system.
- Exercise 7:** Identify and list the contents of the `/usr` directory.
- Exercise 8:** Investigate the contents of the `/bin` and `/sbin` directories and explain their significance.

1.4 Sample Viva Questions

- Question 1:** Can you illustrate the process of changing file permissions using `chmod` with symbolic and octal notation?
- Question 2:** How do you switch directories using relative and absolute paths in the `cd` command?
- Question 3:** Can you provide an example of using `grep` to search for a specific pattern within a file or multiple files?
- Question 4:** Distinguish between internal and external commands.
- Question 5:** Explain file system hierarchy of Linux.
- Question 6:** List different file systems used in Windows and Linux.

1.5 Evaluation Parameters and Rating Scale

1.5.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

1.5.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

2 Experiment 2: Basics of Shell Scripting

2.1 Objective

The primary objectives include creating simple yet functional scripts, grasping scripting syntax elements such as variables, loops, and control structures, and executing these scripts effectively. Moreover, this experiment will emphasize the real-world utility of shell scripting in automating routine tasks, performing file manipulations, and simplifying system administration processes.

2.2 Reference Material[1][3][2]

2.2.1 Shell

In computing, a shell refers to a user interface that allows users to interact with an operating system (such as Linux, Unix, or Windows) to execute commands, run programs, and manage files and directories. It acts as an intermediary between the user and the core functionalities of the operating system.

2.2.2 Shell script

A shell program, also known as a shell script, is like a recipe made up of step-by-step instructions written in a special language that the computer understands. Each instruction in this recipe tells the computer what to do. These scripts are saved as files with a `.sh` ending and are created using programs like `vi`. To use them, we have to give permission to the computer to run these script files. We do this by using a command called `chmod`. Then, in a terminal, we use commands like `sh` or `bash` to tell the computer to follow the instructions written in the script file. It's like giving the computer a set of tasks to perform, and it does them one after another.

2.2.3 Shell Variables

Shell variables are placeholders used by a shell (like Bash or PowerShell) to store information or data values. These variables act as containers to hold data temporarily, such as strings of text, numbers, file paths, or configuration settings, for example: `VAR_NAME = value`.

2.2.4 Types of Variables

Some commonly used shell variables include:

- (i) Environment Variables: These are variables that contain information about the environment in which the shell operates, such as user settings, system paths, or configuration preferences.
- (ii) User-Defined Variables: These are variables created by users to store custom data or information required for specific tasks or scripts.

2.2.5 Important Environment Variables

- **PATH:** Specifies directories where executable programs are located.
- **HOME:** Represents the current user's home directory.
- **USER:** Displays the username of the current user.
- **SHELL:** Specifies the default shell for the user.
- **PWD:** Indicates the present working directory.
- **LANG:** Determines the language and localization settings.
- **TERM:** Defines the terminal type or emulator being used.
- **EDITOR:** Specifies the default text editor.

2.2.6 Shell Redirections

Shell redirections in Linux/Unix allow users to control input and output streams of commands. Here are common redirection symbols:

- **Standard Input Redirection (<):** Changes the command's input source to a file.

```
command < input_file.txt
```

- **Standard Output Redirection (>):** Redirects command output to a file (overwrites existing content).

```
command > output_file.txt
```

- **Appending Output (>>):** Appends command output to a file.

```
command >> output_file.txt
```

- **Piping Output (|):** Redirects output of one command as input to another.

```
command1 | command2
```

- **Standard Error Redirection (2> or 2>>):** Redirects error messages to a file.

```
command 2> error_file.txt
```


2.2.7 Shell Arithmetic using `expr` and `bc` Commands

In shell scripting, arithmetic operations can be performed using different commands:

- **`expr` Command:** Used for integer arithmetic operations within shell scripts. It evaluates and prints the result of expressions.

Example usage:

```
result=$(expr 5 + 3)    # Adds 5 and 3
echo "Result: $result"  # Output: Result: 8
```

Supported operators in `expr` include addition (+), subtraction (-), multiplication (*), division (/), modulus (%)

- **`bc` Command:** Stands for 'Basic Calculator' and supports floating-point arithmetic and advanced mathematical functions.

Example usage:

```
result=$(echo "5.5 + 3.2" | bc) # Adds 5.5 and 3.2
echo "Result: $result"          # Output: Result: 8.7
```

`bc` handles floating-point arithmetic and provides functions like sine, cosine, square root, etc., for more complex calculations.

These commands offer basic arithmetic functionalities within shell scripts. `expr` is suitable for simple integer arithmetic, while `bc` provides a broader range of mathematical operations and supports floating-point numbers.

2.2.8 Flow Control in Shell Scripting using `if` Statements

Shell scripting offers various forms of `if` statements for conditional flow control:

1. Basic `if` Statement:

```
if [ condition ]; then
    # Commands to execute if the condition is true
fi
```

2. `if-else` Statement:

```
if [ condition ]; then
    # Commands to execute if the condition is true
else
    # Commands to execute if the condition is false
fi
```

3. if-elif-else Statement:

```
if [ condition1 ]; then
    # Commands to execute if condition1 is true
elif [ condition2 ]; then
    # Commands to execute if condition2 is true
else
    # Commands to execute if both condition1 and condition2 are false
fi
```

4. Nested if Statements:

```
if [ condition1 ]; then
    if [ condition2 ]; then
        # Commands to execute if both condition1 and condition2 are true
    fi
fi
```

5. if Statement with Logical Operators:

```
if [ condition1 -a condition2 ]; then
    # Commands to execute if condition1 AND condition2 are true
fi

if [ condition1 -o condition2 ]; then
    # Commands to execute if condition1 OR condition2 is true
fi
```

These variations enable conditional execution of commands based on different conditions within shell scripts, offering flexibility in controlling the flow of the script.

2.2.9 Operators in Shell Scripting

In shell scripting, operators are used to perform various operations on variables, constants, and expressions. Here are the common types of operators:

1. Arithmetic Operators

- `+`, `-`, `*`, `/`, `%`: Perform addition, subtraction, multiplication, division, and modulus respectively.

2. Relational Operators

- `-eq`, `-ne`, `-gt`, `-lt`, `-ge`, `-le`: Compare numbers (equal, not equal, greater than, less than, greater than or equal to, less than or equal to) within `test` or `[]` brackets.

3. String Operators

- `=`, `!=`, `<`, `>`: Compare strings (equal, not equal, less than, greater than) within `test` or `[]` brackets.

4. Logical Operators

- `&&`, `||`, `!`: Perform logical AND, logical OR, and logical NOT operations respectively.

5. Assignment Operators

- `=`, `+=`: Assign values to variables or concatenate strings.

6. Bitwise Operators

- `&`, `|`, `^`, `<<`, `>>`, `~`: Perform bitwise AND, OR, XOR, left shift, right shift, and bitwise NOT operations respectively.

7. File Test Operators

- `-f`, `-d`, `-r`, `-w`, `-x`: Check file properties (existence, directory, readability, writability, executability) within `test` or `[]` brackets.

Understanding and utilizing these operators enable the creation of conditions, calculations, string manipulations, and file property checks within shell scripts, enhancing their functionality and flexibility.

2.2.10 Case Structure in Shell Scripting

In shell scripting, the `case` structure provides a way to perform conditional branching based on the value of a variable or expression. It allows evaluation against multiple patterns and executes commands based on the matching pattern.

The basic syntax of the `case` structure is as follows:

```
case variable in
    pattern1)
        # Commands to execute if variable matches pattern1
        ;;
    pattern2)
        # Commands to execute if variable matches pattern2
        ;;
    pattern3|pattern4)
        # Commands to execute if variable matches pattern3 or pattern4
        ;;
    *)
        # Default commands to execute if no pattern matches
        ;;
esac
```

Example:

```
fruit="apple"

case $fruit in
    apple)
        echo "It's an apple."
        ;;
    banana|orange)
        echo "It's a banana or an orange."
        ;;
    *)
        echo "It's another fruit."
        ;;
esac
```

This example evaluates the variable `$fruit` against different patterns and executes respective blocks based on the matching pattern.

The `case` structure simplifies code readability when multiple conditions need evaluation against a single variable or expression.

2.2.11 Loops in Shell Scripting

In shell scripting, loops are used to execute a block of code repeatedly based on certain conditions. There are various types of loops available:

1. for Loop

The `for` loop iterates through a list of items or values. It is suitable when you have a known set of elements to loop through.

Syntax:

```
for variable in list
do
    # Commands to execute for each iteration
done
```

Example:

```
for i in 1 2 3 4 5
do
    echo "Iteration: $i"
done
```

2. while Loop

The `while` loop executes a block of code as long as a specified condition remains true.

Syntax:

```
while [ condition ]
do
    # Commands to execute as long as the condition is true
done
```

Example:

```
count=1
while [ $count -le 5 ]
do
    echo "Count: $count"
    ((count++))
done
```

3. until Loop

The until loop executes a block of code until a specified condition becomes true.

Syntax:

```
until [ condition ]
do
    # Commands to execute until the condition becomes true
done
```

Example:

```
count=1
until [ $count -gt 5 ]
do
    echo "Count: $count"
    ((count++))
done
```

4. Nested Loops

You can nest loops within each other to create more complex control structures.

Example:

```
for i in {1..3}
do
    echo "Outer Loop Iteration: $i"
    for j in A B C
    do
        echo "    Inner Loop Iteration: $j"
    done
done
```

These loops in shell scripting provide various ways to iterate through data, execute code repeatedly based on conditions, and control the flow of a script.

2.3 Steps to Prepare and Execute a Shell Script

1. Create the Shell Script:

- Open a text editor or an Integrated Development Environment (IDE) to write the shell script.
- Write the desired commands and save the file with a `.sh` extension (e.g., `script.sh`).

2. Set Execution Permissions (if needed):

- If the script doesn't have execution permissions, use the `chmod` command to set the execution permission:

```
chmod +x script.sh
```

3. Run the Shell Script:

- Open the terminal.
- Navigate to the directory where the script is saved using the `cd` command.
- Execute the script using one of the following methods:

- Using Bash:

```
bash script.sh
```

- Using Shorthand (if the script has execution permissions):

```
./script.sh
```

- Using Absolute Path:

```
/path/to/your/script.sh
```

4. Verify Output:

- After executing the script, verify the output or actions performed by the script in the terminal or through any generated files or changes made by the script.

5. Debug and Modify (if needed):

- If there are errors or the script doesn't behave as expected, edit the script in the text editor, save the changes, and rerun the script.

6. Exit Code Analysis (Optional):

- After script execution, check the exit code by typing:

```
echo $?
```

An exit code of 0 generally indicates success, while other codes signify different types of errors or warnings.

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVVHh8jIve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

2.3.1 Lab Exercises

- Exercise 1:** Write a shell script that calculates the sum of numbers from 1 to a user-provided value using a **for** loop.
- Exercise 2:** Write a script to generate the Fibonacci series up to a specified number of terms using a **for** loop.
- Exercise 3:** Develop a script that checks whether a user-input number is prime or not using a **for** loop.
- Exercise 4:** Create a script that reverses a given number using a **while** loop.
- Exercise 5:** Write a script to display all files in a directory using a **for** loop.
- Exercise 6:** Develop a script that asks the user for a username and password, allowing access only if both are correct using control structures.
- Exercise 7:** Create a menu-driven calculator script that performs basic arithmetic operations based on user selection using a **case** structure.

2.4 Sample Viva Questions

Question 1: Describe how to redirect input and output in a shell script.

Question 2: How do you read user input in a shell script? Explain with an example.

Question 3: Discuss different types of operators used in shell scripting (arithmetic, comparison, logical, etc.).

Question 4: What are variables in shell scripting? How do you declare and use them?

Question 5: What is the significance of the shebang (!) line in a shell script?

Question 6: What are the different types of shells commonly used in Unix/Linux?

2.5 Evaluation Parameters and Rating Scale

2.5.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

2.5.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

2.6 Instructions for the Teacher to Conduct Lab Assessment 1

- Upon finishing the initial two experiments, you are required to carry out CAP1, which is worth 100 marks [comprising two parts: J/E (50) and LM (50)].
- The calculation for LM marks is determined by: (Average of the teacher's overall ratings in Experiment 1 and Experiment 2) multiplied by 5.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

3 Experiment 3: File Manipulation Using System Calls[1][3][2]

3.1 Objective

The goal of this lab is to introduce and apply fundamental system calls like open, read, write, lseek, and close through programming exercises, addressing practical challenges encountered in real-world scenarios.

3.2 Reading Material

3.2.1 System Calls

System calls are functions provided by the operating system kernel that enable user-level processes to request services from the operating system. Examples: Common system calls include open, read, write (for file operations), fork, exec (for process control), malloc, free (for memory management), socket, bind, connect (for networking), and many others.

System Call	Description	Programming Syntax
open	Used to open a file and obtain a file descriptor. Allows specifying flags for read, write, create, and permissions.	<code>int fd = open("file.txt", O_RDONLY);</code>
close	Closes a file descriptor, releasing resources associated with the file.	<code>close(fd);</code>
read	Reads data from an open file into a buffer in memory.	<code>read(fd, buffer, nbytes);</code>
write	Writes data from a buffer in memory to an open file.	<code>write(fd, buffer, nbytes);</code>
lseek	Moves the file pointer to a specified position in the file.	<code>lseek(fd, offset, SEEK_SET);</code>
unlink	Deletes a file by name.	<code>unlink("file.txt");</code>
mkdir	Creates a new directory.	<code>mkdir("new_dir", 0755);</code>
rmdir	Removes a directory if it's empty.	<code>rmdir("directory");</code>
rename	Renames a file or directory.	<code>rename("old_name", "new_name");</code>
stat	Retrieves file status information like permissions, size, and timestamps.	<code>struct stat fileStat; stat("file.txt", &fileStat);</code>
chmod	Changes file permissions.	<code>chmod("file.txt", 0644);</code>
chown	Changes file ownership.	<code>chown("file.txt", uid, gid);</code>
link/symlink	Creates hard or symbolic links to files.	<code>link("source", "target");</code>

Table 2: File management related system calls

3.2.2 Sample Programs

1. Program to Create and Open a File for Reading

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

int main() {
    int fileDescriptor;

    // Create a new file named "file.txt" and open it for reading
    fileDescriptor = open("file.txt", O_CREAT | O_RDONLY, 0644);

    close(fileDescriptor); // Close the file

    return 0;
}
```

2. Program to Read from Console and Write to Console

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

#define BUFFER_SIZE 1024

int main() {
    char buffer[BUFFER_SIZE];
    //0 is the file descriptor of standard input.
    ssize_t bytesRead = read(0, buffer, BUFFER_SIZE);
    write(1, buffer, bytesRead); //1 is the file descriptor of standard output.

    return 0;
}
```

3. Program to Append Data into a File

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

int main() {
    char data[] = "This data will be appended to the file.\n";
```

```

    int fileDescriptor;

    fileDescriptor = open("file.txt", O_WRONLY | O_CREAT | O_APPEND, 0644);
    write(fileDescriptor, data, strlen(data));
    close(fileDescriptor);

    return 0;
}

```

4. Program to Read from and Write to Files

```

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

#define BUFFER_SIZE 1024

int main() {
    char buffer[BUFFER_SIZE];
    int readFd, writeFd;

    readFd = open("source.txt", ORDONLY);
    writeFd = open("destination.txt", O_WRONLY | O_CREAT | O_TRUNC, 0644);

    ssize_t bytesRead;
    while ((bytesRead = read(readFd, buffer, BUFFER_SIZE)) > 0) {
        write(writeFd, buffer, bytesRead);
    }

    close(readFd);
    close(writeFd);

    return 0;
}

```

5. a C program that reads characters from the 11th to the 20th position from a file named "input.txt" using the lseek system call.

```

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

#define BUFFER_SIZE 11

int main() {

```



```

    int fileDescriptor;
    char buffer[BUFFER_SIZE];

    fileDescriptor = open("input.txt", O_RDONLY);
    lseek(fileDescriptor, 10, SEEK_SET);
    read(fileDescriptor, buffer, BUFFER_SIZE - 1);
    buffer[BUFFER_SIZE - 1] = '\0';
    printf(" Characters from 11th to 20th position: %s\n", buffer);
    close(fileDescriptor);

    return 0;
}

```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

3.3 Lab Exercises

Exercise 1: Write a program in C using system calls that lets users choose to copy either the first half or the second half of a file by entering 1 or 2.

Exercise 2: Create a C program using system calls that keeps reading from the console until the user types '\$'. Save the input data to a file called 'input.txt'.

Exercise 3: Write a C program that encrypts a text file using a simple encryption technique and saves the encrypted content to a new file.

Requirements:

Input: Provide a text file named "input.txt" with plain text content.

Encryption Technique: Shift each character in the file content by a fixed number

of positions (e.g., shifting each character by 3 positions in the ASCII table).

Output: Save the encrypted content to a new file named "encrypted.txt".

3.4 Sample Viva Questions

Question 1: What is the purpose of the open system call?

Question 2: Discuss the parameters of the read system call.

Question 3: What is the significance of the close system call?

Question 4: What parameters does the lseek system call take?

Question 5: How is the lseek system call used to move the file offset?

3.5 Evaluation Parameters and Ratings Scale

3.5.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
How Confident you are in Practical Implementation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Timely Completion of Lab Exercises	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
How confident you are in viva questions	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Lab-manual Readiness on the date of evaluation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Overall Rating	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Student's Signature: _____

Date: _____

3.5.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Practical Implementation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Completion of Lab Exercises	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Student adhering to lab conduct standards	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Lab-manual Readiness	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Overall Rating	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Teacher's Signature: _____

Date: _____

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

4 Experiment 4: Directory Manipulation Using System Calls

4.1 Objective

In this lab, we'll explore how shell commands like `mkdir` (used to create directories), `rmdir` (used to remove directories) and `ls` (used to list directory contents) actually work behind the scenes. These commands rely on special functions called system calls such as `mkdir`, `opendir`, and `readdir`. By learning about these functions, you'll be able to understand how these commands function and even write your own code using them.

4.2 Reading Material[1][3][2]

4.2.1 Directory Management Related System Calls

System Call	Description	Programming Syntax
<code>opendir</code>	Opens a directory stream corresponding to the given directory name.	<code>DIR *opendir(const char *dirname);</code>
<code>readdir</code>	Reads the next directory entry from the directory stream.	<code>struct dirent *readdir(DIR *dir_stream);</code>
<code>closedir</code>	Closes the directory stream.	<code>int closedir(DIR *dir_stream);</code>
<code>chdir</code>	Changes the current working directory.	<code>int chdir(const char *path);</code>
<code>mkdir</code>	Creates a new directory with the specified name and permission mode.	<code>int mkdir(const char *pathname, mode_t mode);</code>
<code>rmdir</code>	Removes a directory.	<code>int rmdir(const char *pathname);</code>
<code>getcwd</code>	Gets the pathname of the current working directory.	<code>char *getcwd(char *buf, size_t size);</code>

Table 3: System calls for directory management

4.2.2 The dirent structure

The `dirent` structure in C is used to store information about directory entries when working with directory-related system calls. It's commonly associated with functions like `readdir` and is defined in the `<dirent.h>` header file in Linux systems.

The structure `dirent` typically contains the following members:

- `ino_t d_ino`: This member represents the inode number of the directory entry.
- `off_t d_off`: It stores the offset of the next `readdir` call within the directory stream.

- `unsigned short int d_reclen`: It denotes the length of this record.
- `unsigned char d_type`: This member identifies the type of the file. For example, `DT_DIR` for directories, `DT_REG` for regular files, and others based on the file type.
- `char d_name[]`: This member holds the name of the directory entry. It is a character array representing the name of the file or directory.

4.3 Sample Programs

1. A C program that prints the contents of a directory using system calls like `opendir`, `readdir`, and `closedir`

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

int main() {
    DIR *dir;
    struct dirent *entry;

    dir = opendir(".");

    if (dir) {
        printf("Contents of the directory:\n");
        while ((entry = readdir(dir)) != NULL) {
            printf("%s\n", entry->d_name);
        }
        closedir(dir);
    }

    return 0;
}
```

2. Write a C program to create a new directory named "NewDirectory" within the file system

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main() {
    const char *dirname = "NewDirectory";

    // Creating a new directory named "NewDirectory"
    mkdir(dirname, 0777);
}
```



```
    return 0;
}
```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

4.4 Lab Exercises

- Exercise 1:** Create a C program that prompts the user to enter a directory name and uses the `mkdir` system call to create the directory.
- Exercise 2:** Write a program that opens the current directory using `opendir` and reads its contents using `readdir`, then displays the list of directory entries.
- Exercise 3:** Create a C program to delete a directory specified by the user using the `rmdir` system call.
- Exercise 4:** Write a program that uses the `getcwd` system call to retrieve the current working directory and displays it to the user.

4.5 Sample Viva Questions

- Question 1:** How is the `mkdir` system call used in C programming?
- Question 2:** Discuss the significance of the `rmdir` system call.
- Question 3:** How is the `opendir` system call used in C programming?
- Question 4:** Explain the `chdir` system call and its significance.
- Question 5:** What are the main fields or members of the `dirent` structure?

4.6 Evaluation Parameters and Rating Scale

4.6.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

4.6.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

4.7 Instructions for the Teacher to Conduct Lab Assessment 2

- Upon finishing the experiment 3 and experiment 4, you are required to carry out CAP2, which is worth 100 marks [comprising two parts: J/E (50) and LM (50)].
- The calculation for LM marks is determined by: (Average of the teacher's overall ratings in Experiment 3 and Experiment 4) multiplied by 5.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

5 Experiment 5: Process Management using System Calls

5.1 Objective

By the end of this experiment, students will learn about different system commands used to manage processes. They will also gain an understanding of orphan and zombie processes.

5.2 Reading Material[1][3][2]

5.2.1 Process management related system calls

System Call	Description	Programming Syntax
fork()	Creates a new process by duplicating the calling process.	<code>pid_t fork(void);</code>
exec()	Replaces the current process with a new program.	<code>int exec(const char *pathname, char *const argv[]);</code>
wait()	Causes the parent process to wait until one of its child processes terminates.	<code>pid_t wait(int *status);</code>
exit()	Terminates the calling process and returns an exit status to the operating system.	<code>void exit(int status);</code>
getpid()	Retrieves the process ID (PID) of the calling process.	<code>pid_t getpid(void);</code>
getppid()	Retrieves the parent process ID (PPID) of the calling process.	<code>pid_t getppid(void);</code>
kill()	Sends a signal to a specified process or group of processes.	<code>int kill(pid_t pid, int sig);</code>
nice()	Modifies the priority of a process.	<code>int nice(int incr);</code>
sleep()	Causes the calling process to sleep for a specified number of seconds.	<code>unsigned int sleep(unsigned int seconds);</code>

Table 4: Process management related system calls

5.2.2 Orphan and Zombie Processes

Orphan Process: An orphan process is a child process whose parent process has terminated or finished before the child process completes. When the parent process exits or is terminated unexpectedly without properly waiting for the child to finish, the operating system reassigns the orphaned child process to the init process (PID 1 in Linux). The init process adopts and manages orphan processes until they complete execution.

Zombie Process: A zombie process is a terminated process that has completed its execution but still has an entry in the process table. After a process completes, it sends an exit status to its parent process and becomes a zombie waiting for the parent to retrieve the exit status using the `wait()`

system call. If the parent fails to fetch the exit status of the terminated child (due to neglect or termination), the zombie process remains in the process table as an inactive process entry.

5.3 Sample Programs

1. A program to create a child process using fork system call.

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

int main() {
    pid_t child_pid;

    // Create a child process
    child_pid = fork();

    if (child_pid == 0) {
        // The child process code section
        printf("Child-process:-PID=-%d\n", getpid());
    } else if (child_pid > 0) {
        // The parent process code section
        printf("Parent-process:-Child-PID=-%d\n", child_pid);
    } else {
        // Fork failed
        printf("Fork-failed\n");
        return 1;
    }

    return 0;
}
```

2. C program to demonstrates the creation of an orphan process.

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

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        // Child process
        printf("Child-process:-PID=-%d\n", getpid());
        sleep(2); // Sleep to ensure the parent process terminates first
        printf("Child-process:-My-parent's-PID=-%d\n", getppid());
    } else if (child_pid > 0) {
```



```

        // Parent process
        printf("Parent process: PID=%d\n", getpid());
        printf("Parent process: Terminating...\n");
    } else {
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}

```

3. C program to demonstrate the creation of a Zombie process.

```

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>

int main() {
    pid_t child_pid = fork();

    if (child_pid == 0) {
        // Child process
        printf("Child process: PID=%d\n", getpid());
        exit(0); // Child process exits immediately
    } else if (child_pid > 0) {
        // Parent process
        printf("Parent process: PID=%d\n", getpid());
        printf("Parent process: Child PID=%d\n", child_pid);
        sleep(10); // Sleep to allow time for the child to become a zombie
        printf("Parent process: Terminating...\n");
    } else {
        printf("Fork failed\n");
        return 1;
    }

    return 0;
}

```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

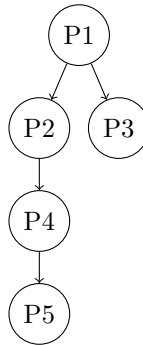
- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

5.4 Lab Exercises

Exercise 1: Write a C program to illustrate that performing 'n' consecutive `fork()` system calls generates a total of $2^n - 1$ child processes. The program should prompt the user to input the value of 'n'."

Exercise 2: Write a C program utilizing the `fork()` system call to generate the following process hierarchy: $P1 \rightarrow P2 \rightarrow P3$. The program should display the Process ID (PID) and Parent Process IDs (PPID) for each process created.

Exercise 3: Write a C program to generate a process hierarchy as follows:



The program should create the specified process structure using the appropriate sequence of '`fork()`' system calls. Print PID and PPID of each process.

5.5 Sample Viva Questions

Question 1: Explain the fork system call in process management.

Question 2: Discuss the significance of the return values of the fork system call.

Question 3: What is the purpose of the wait system call?

Question 4: Discuss the significance of the getpid and getppid system calls in obtaining process IDs.

Question 5: How many child processes will be created if three consecutive fork statements are used in a main function?

5.6 Evaluation Parameters and Rating Scale

5.6.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

5.6.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

6 Experiment 6: Creation of Multithreaded Processes using Pthread Library

6.1 Objective

Introduce the operations on threads, which include initialization, creation, join and exit functions of thread using pthread library.

6.2 Reading Material[1][3][2]

6.2.1 Commonly used library functions related to POSIX threads (pthread)

Function	Description	Programming Syntax
pthread_create	Create a new thread	int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
pthread_join	Wait for termination of a specific thread	int pthread_join(pthread_t thread, void **retval);
pthread_exit	Terminate calling thread	void pthread_exit(void *retval);
pthread_cancel	Request cancellation of a thread	int pthread_cancel(pthread_t thread);

Table 5: Commonly used library functions related to POSIX threads (pthread)

6.3 Sample Programs

1. A C program using the pthread library to create a thread with NULL attributes.

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

void *thread_function(void *arg) {
    printf("Inside the new thread!\n");
    return NULL;
}

int main() {
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thread_function, NULL);
    pthread_join(thread_id, NULL);
    return 0;
}
```


2. A C program that creates a thread and passes a message from the main function to the thread.

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

void *thread_function(void *message) {
    printf("Message received in thread: %s\n", (char *)message);
    return NULL;
}

int main() {
    pthread_t thread_id;
    pthread_create(&thread_id, NULL, thread_function, "Hello from the main thread!");
    pthread_join(thread_id, NULL);
    return 0;
}
```

3. A C program where a thread returns a value to the main function using pointers.

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

#define NUM_THREADS 1

void *thread_function(void *arg) {
    int *returnValue = malloc(sizeof(int));
    *returnValue = 143; // Set the return value
    pthread_exit(returnValue);
}

int main() {
    pthread_t threads[NUM_THREADS];
    int *thread_return;

    pthread_create(&threads[0], NULL, thread_function, NULL);
    pthread_join(threads[0], (void **)&thread_return);

    printf("Value returned from thread: %d\n", *thread_return);

    free(thread_return); // Free allocated memory for return value
    return 0;
}
```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

6.4 Lab Exercises: [Attempt any 3 within the designated lab hours]

Exercise 1: Develop a program using `pthread` to concatenate multiple strings passed to the thread function.

Exercise 2: Create a `pthread` program to find the length of strings passed to the thread function.

Exercise 3: Implement a program that performs statistical operations (calculating average, maximum, and minimum) for a set of numbers. Utilize three threads, where each thread performs its respective operation.

Exercise 4: Write a multithreaded program where a globally passed array of integers is divided into two smaller lists and given as input to two threads. Each thread sorts their half of the list and then passes the sorted lists to a third thread, which merges and sorts them. The final sorted list is printed by the parent thread.

Exercise 5: Create a program using `pthread_create` to generate multiple threads. Each thread should display its unique ID and execution sequence.

Exercise 6: Create a threaded application that demonstrates graceful thread termination using `pthread_exit` for resource cleanup compared to abrupt termination via `pthread_cancel`.

6.5 Sample Viva Questions

Question 1: Discuss the steps involved in creating a thread using the `pthread_create` function.

Question 2: What parameters does the `pthread_create` function take, and what are their purposes?

Question 3: Explain the role and usage of the `pthread_join` function in managing threads.

Question 4: Explain the differences between a thread and a process in terms of memory sharing and execution context.

Question 5: Give two instances where a multi-threaded process offers benefits compared to a single-threaded solution.

6.6 Evaluation Parameters and Rating Scale

6.6.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

6.6.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

6.7 Instructions for the Teacher to Conduct Lab Assessment 3

- Upon finishing the experiment 5 and experiment 6, you are required to carry out CAP3, which is worth 100 marks [comprising two parts: J/E (50) and LM (50)].
- The calculation for LM marks is determined by: (Average of the teacher's overall ratings in Experiment 5 and Experiment 6) multiplied by 5.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

7 Experiment 7: Process Synchronization using Semaphore/-Mutex

7.1 Objective

The objective of this lab experiment is to acquaint students with the concept of process synchronization in concurrent programming using semaphore or mutex mechanisms.

7.2 Reading Material[1][3][2]

7.2.1 Synchronization

Process or thread synchronization refers to the coordination and orderly execution of concurrent processes or threads in a multi-threaded or multi-process system.

7.2.2 Race Condition

A race condition is a situation in concurrent programming where the outcome of the program depends on the order of execution of threads or processes. It arises when multiple threads or processes access shared resources or critical sections without proper synchronization or coordination, leading to unpredictable or incorrect behavior.

7.2.3 Semaphore

Definition	A semaphore is an abstract data type used for process synchronization in concurrent programming. It controls access to shared resources among multiple processes or threads by maintaining a counter that can be incremented or decremented.
Functionality	Semaphores manage access to shared resources, prevent race conditions, and ensure synchronization. They offer operations like initialization (<code>sem_init</code>), waiting (<code>sem_wait</code>), signaling (<code>sem_post</code>), and destruction (<code>sem_destroy</code>).
Types	Common types include Binary Semaphores (with values 0 and 1) and Counting Semaphores (with values greater than 1).

Function	Description	Programming Syntax
<code>sem_init</code>	Initialize a semaphore	<code>int sem_init(sem_t *sem, int pshared, unsigned int value);</code>
<code>sem_destroy</code>	Destroy a semaphore	<code>int sem_destroy(sem_t *sem);</code>
<code>sem_post</code>	Increment (signal) a semaphore	<code>int sem_post(sem_t *sem);</code>
<code>sem_wait</code>	Decrement (wait/block) a semaphore	<code>int sem_wait(sem_t *sem);</code>

Table 6: Library Functions Related to Semaphore

7.2.4 Mutex

Definition	A mutex (Mutual Exclusion) is a synchronization primitive used in multi-threaded programming to control access to shared resources. It allows only one thread at a time to access the resource, preventing concurrent access.
Functionality	Mutexes ensure mutual exclusion, preventing race conditions and maintaining data integrity. Operations include initialization (<code>pthread_mutex_init</code>), locking (<code>pthread_mutex_lock</code>), unlocking (<code>pthread_mutex_unlock</code>), and destruction (<code>pthread_mutex_destroy</code>).
Types	Mutexes can be recursive (allows the same thread to lock it multiple times) or non-recursive (deadlocks if the same thread tries to lock it multiple times).

Function	Description	Programming Syntax
<code>pthread_mutex_init</code>	Initialize a mutex	<code>int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr);</code>
<code>pthread_mutex_destroy</code>	Destroy a mutex	<code>int pthread_mutex_destroy(pthread_mutex_t *mutex);</code>
<code>pthread_mutex_lock</code>	Lock a mutex	<code>int pthread_mutex_lock(pthread_mutex_t *mutex);</code>
<code>pthread_mutex_unlock</code>	Unlock a mutex	<code>int pthread_mutex_unlock(pthread_mutex_t *mutex);</code>

Table 7: Library Functions Related to Mutex

7.3 Sample Programs

1. C Program Simulating Race Condition.

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

#define NUMTHREADS 5
#define MAXCOUNT 1000000

int shared_variable = 0;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAXCOUNT; i++) {
        shared_variable++;
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUMTHREADS];
```

```

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    printf("Value of shared variable after race condition: %d\n", shared_variable);

    return 0;
}

```

2. C Program with Semaphore to Avoid Race Condition.

```

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

#define NUMTHREADS 5
#define MAXCOUNT 1000000

int shared_variable = 0;
sem_t semaphore;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAXCOUNT; i++) {
        sem_wait(&semaphore);
        shared_variable++;
        sem_post(&semaphore);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUMTHREADS];
    sem_init(&semaphore, 0, 1); // Initializing semaphore with value 1

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }
}

```

```

    sem_destroy(&semaphore); // Destroying semaphore

    printf("Value of shared variable after synchronization: %d\n", shared_variable);

    return 0;
}

```

3. C Program with Mutex to Prevent Race Condition

```

#include <stdio.h>
#include <pthread.h>

#define NUMTHREADS 5
#define MAXCOUNT 1000000

int shared_variable = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *increment_variable(void *thread_id) {
    for (int i = 0; i < MAXCOUNT; i++) {
        pthread_mutex_lock(&mutex);
        shared_variable++;
        pthread_mutex_unlock(&mutex);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t threads[NUMTHREADS];
    pthread_mutex_init(&mutex, NULL); // Initializing mutex

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_create(&threads[i], NULL, increment_variable, (void *)i);
    }

    for (int i = 0; i < NUMTHREADS; i++) {
        pthread_join(threads[i], NULL);
    }

    pthread_mutex_destroy(&mutex); // Destroying mutex

    printf("Value of shared variable after synchronization: %d\n", shared_variable);

    return 0;
}

```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

7.4 Lab Exercises

Exercise 1: Implement the producer-consumer problem using pthreads and mutex operations.

Constraints:

- (a) A producer only produces if the buffer is empty, and the consumer only consumes if some content is in the buffer.
- (b) A producer writes an item into the buffer, and the consumer deletes the last produced item in the buffer.
- (c) A producer writes on the last consumed index of the buffer.

Exercise 2: Implement the reader-writer problem using semaphore and mutex operations to synchronize n readers active in the reader section at the same time and one writer active at a time.

Constraints:

- (a) If n readers are active, no writer is allowed to write.
- (b) If one writer is writing, no other writer should be allowed to read or write on the shared variable.

7.5 Sample Viva Questions

Question 1: What is a semaphore, and how does it facilitate synchronization among processes or threads?

Question 2: Discuss the functions `sem_init`, `sem_wait`, and `sem_post` in semaphore usage.

Question 3: Compare and contrast mutexes with semaphores in terms of functionality and usage.

Question 4: Explain the functions `pthread_mutex_init`, `pthread_mutex_lock`, and `pthread_mutex_unlock` in mutex usage.

Question 5: Discuss scenarios where mutexes are preferred over other synchronization mechanisms.

7.6 Evaluation Parameters and Rating Scale

7.6.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How Confident you are in Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Timely Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
How confident you are in viva questions	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness on the date of evaluation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Student's Signature: _____

Date: _____

7.6.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Practical Implementation	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Completion of Lab Exercises	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Student adhering to lab conduct standards	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Lab-manual Readiness	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩
Overall Rating	① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩

Teacher's Signature: _____

Date: _____

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

8 Experiment 8: Inter Process Communication (IPC)

8.1 Objective

The aim of this laboratory is to introduce the Interprocess communication (IPC) mechanism of operating system to allow the processes to communicate with each other.

8.2 Reading Material[1][3][2]

8.2.1 Interprocess communication

Interprocess communication (IPC) involves methods and tools that enable different processes on a computer system to exchange data, coordinate activities, and synchronize operations. It allows programs to communicate with each other, facilitating collaboration and data sharing between processes running concurrently. Examples of IPC methods include pipes, sockets, shared memory, and message queues.

8.2.2 Pipes

In C programming, pipes are a form of interprocess communication (IPC) that allows communication between two processes, with one process writing data into the pipe and the other process reading from it. Pipes are one-way communication channels that can be either anonymous or named.

Function	Description	Programming Syntax
pipe()	Creates an anonymous pipe, returning two file descriptors - one for reading and one for writing.	<code>int pipe(int filedes[2]);</code>
mkfifo()	Creates a named pipe (FIFO) in the file system.	<code>int mkfifo(const char *pathname, mode_t mode);</code>
dup()	Duplicates a file descriptor, creating a copy of the specified descriptor.	<code>int dup(int oldfd);</code>
close()	Closes a file descriptor.	<code>int close(int fd);</code>

Table 8: Functions related to pipes

8.2.3 Unnamed Pipes (Anonymous Pipes)

Functioning: Unnamed pipes are created using the pipe() system call. They provide a one-way communication channel between two related processes, typically a parent and its child process. They allow communication by connecting the standard output (stdout) of one process to the standard input (stdin) of another.

- **Syntax:** `int pipe(int filedes[2]);`

- `filedes[0]` refers to the read end of the pipe.
- `filedes[1]` refers to the write end of the pipe.

Sample Program

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

int main() {
    int pipe_fd[2];
    char data[100];

    if (pipe(pipe_fd) < 0) {
        printf("pipe-creation-failed");
        return 1;
    }

    pid_t pid = fork();

    if (pid < 0) {
        printf("fork-failed\n");
        return 1;
    }

    if (pid > 0) { // Parent process
        // Write to pipe
        close(pipe_fd[0]); // Close the reading end
        char message[] = "Hello, -Child-Process!";
        write(pipe_fd[1], message, sizeof(message));
        close(pipe_fd[1]); // Close the writing end
    } else { // Child process
        // Read from pipe
        close(pipe_fd[1]); // Close the writing end
        read(pipe_fd[0], data, sizeof(data));
        printf("Received-message-in-child: %s\n", data);
        close(pipe_fd[0]); // Close the reading end
    }

    return 0;
}
```

8.2.4 Named Pipes

Named pipes, also known as FIFOs (First In, First Out), are created using the `mkfifo()` system call. They are files residing in the file system and allow communication between unrelated processes. Named pipes provide bi-directional communication.

- **Syntax:** `int mkfifo(const char *pathname, mode_t mode);`
- `pathname` is the path and name of the named pipe.
- `mode` specifies the permissions for the named pipe.

Sample Code[3][1]

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>

int main() {
    char *fifo = "/tmp/myfifo"; // Path to the named pipe

    mkfifo(fifo , 0666); // Creating a named pipe

    int fd;
    char data[100];

    fd = open(fifo , O_WRONLY); // Open the pipe for writing
    write(fd , "Hello -from- writer!" , sizeof("Hello -from- writer!"));
    close(fd);

    fd = open(fifo , O_RDONLY); // Open the pipe for reading
    read(fd , data , sizeof(data));
    printf("Received -message: -%s\n" , data);
    close(fd);

    return 0;
}
```

8.2.5 Shared Memory

Shared memory in C programming is a mechanism that allows multiple processes to share a region of memory. This shared memory segment is created by one process and can be accessed by multiple processes, enabling efficient inter-process communication (IPC).

The primary steps involved in using shared memory are:

1. **Allocation:** A process allocates a shared memory segment using the `shmget()` system call. This call either creates a new shared memory segment or accesses an existing one based on a provided key and size.
2. **Attachment:** After allocation, the process attaches the shared memory segment to its address space using `shmat()`. This attaches the segment to a virtual address in the process's memory, allowing it to read from and write to the shared memory.

3. **Usage:** Processes that share this segment can read from and write to it, treating it like any other memory region. Synchronization mechanisms such as semaphores or mutexes are typically used to control access and prevent race conditions.
4. **Detachment:** When the process finishes using the shared memory, it detaches the segment using `shmdt()`. This detaches the shared memory segment from the process's address space.
5. **Control Operations:** The `shmctl()` function allows for control operations on the shared memory segment, such as removing or modifying it.

Function	Description	Programming Syntax
<code>shmget()</code>	Allocates a new shared memory segment or accesses an existing one.	<code>int shmget(key_t key, size_t size, int shmflg);</code>
<code>shmat()</code>	Attaches the shared memory segment to the address space of the calling process.	<code>void *shmat(int shmid, const void *shmaddr, int shmflg);</code>
<code>shmdt()</code>	Detaches the shared memory segment from the calling process.	<code>int shmdt(const void *shmaddr);</code>
<code>shmctl()</code>	Performs control operations on the shared memory segment, such as removing or modifying it.	<code>int shmctl(int shmid, int cmd, struct shmid_ds *buf);</code>

Table 9: Functions related to shared memory

8.2.6 Sample program to demonstrate shared memory segment creation and data addition

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/shm.h>
#include <string.h>
#include <errno.h>

#define SHM_SIZE 1024

int main() {
    void *shm;
    char buf[100];
    int shmid;

    // Creating a shared memory segment
    shmid = shmget((key_t)123, SHM_SIZE, 0666 | IPC_CREAT);
    if (shmid == -1) {
        printf("shmget - Error\n");
        exit(EXIT_FAILURE);
    }
}
```

```

printf("The Key-value of shared-memory is -%d\n", shmid);

// Attaching the process to the shared memory segment
shm = shmat(shmid, NULL, 0);
if (shm == (void *)-1) {
    printf("shmat- Error\n");
    exit(EXIT_FAILURE);
}
printf("Process-attached-to-the-address-of-%p\n", shm);

printf("Write-the-data-to-shared-memory-(max-99-characters):-");
fgets(buf, sizeof(buf), stdin);
buf[strcspn(buf, "\n")] = '\0'; // Removing newline character if present

strncpy((char *)shm, buf, SHM.SIZE); // Ensuring data doesn't exceed SHM.SIZE
printf("The-stored-data-in-shared-memory-is:-%s\n", (char *)shm);

// Detaching shared memory
if (shmdt(shm) == -1) {
    printf("shmdt- Error\n");
    exit(EXIT_FAILURE);
}

return 0;
}

```

8.2.7 Message Queues

Function	Description	Programming Syntax
msgget()	Creates a new message queue or gets the identifier of an existing queue.	int msgget(key_t key, int msgflg);
msgsnd()	Sends a message to a message queue.	int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
msgrcv()	Receives a message from a message queue.	ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
msgctl()	Performs control operations on a message queue, such as deleting or modifying it.	int msgctl(int msqid, int cmd, struct msqid_ds *buf);

Table 10: Funtions related to message queues

8.2.8 Message Queue Implementation in C

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <errno.h>

#define MAX_MSG_SIZE 100

struct msg_buffer {
    long msg_type;
    char msg_text[MAX_MSG_SIZE];
};

int main() {
    key_t key;
    int msg_id;
    struct msg_buffer message;

    // Manually generate a key without using ftok
    if ((key = 0x12345678) == -1) {
        printf("key-Error\n");
        exit(EXIT_FAILURE);
    }

    if ((msg_id = msgget(key, 0666 | IPC_CREAT)) == -1) {
        printf("msgget-Error\n");
        exit(EXIT_FAILURE);
    }

    printf("Message-Queue-Created-with-ID: %d\n", msg_id);

    printf("Enter-a-message-to-send-to-the-queue:-");
    fgets(message.msg_text, MAX_MSG_SIZE, stdin);
    message.msg_type = 1;

    if (msgsnd(msg_id, &message, sizeof(message), 0) == -1) {
        printf("msgsnd-Error\n");
        exit(EXIT_FAILURE);
    }

    printf("Message-Sent-to-the-Queue\n");

    if (msgrcv(msg_id, &message, sizeof(message), 1, 0) == -1) {
        printf("msgrcv-Error\n");
    }
}

```



```

        exit(EXIT_FAILURE);
    }

    printf("Message Received from the Queue: %s\n", message.msg_text);

    if (msgctl(msg_id, IPC_RMID, NULL) == -1) {
        printf("msgctl Error\n");
        exit(EXIT_FAILURE);
    }

    printf("Message Queue Removed\n");

    return 0;
}

```

Video Reference:



<https://youtube.com/playlist?list=PLWjmN065fOfGdAZrIP6316HVHh8jlve>

References

- [1] Greg Gagne Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. Wiley, 10 edition, 2018.
- [2] Sumitabha Das. *Unix Concepts And Applications*. Wiley, 4 edition, 2006.
- [3] Richard Stones Neil Matthew. *Beginning Linux Programming*. Wiley, 4 edition, 2007.

8.3 Lab Exercises

Exercise 1: Develop a program that demonstrates Inter-Process Communication (IPC) using named pipes.

Tasks:

- Create a pair of named pipes: one for sending data and another for receiving data.
- Develop a sender program that writes a message to the sending pipe.
- Create a receiver program that reads from the receiving pipe and displays the received message.

Exercise 2: Demonstrate the usage of Shared Memory for IPC.

Tasks:

- Create a shared memory segment and attach it to multiple processes.
- Develop a producer-consumer model, where one process writes data into the shared memory, and another process reads from it.

Exercise 3: Explore IPC using Message Passing techniques.

Tasks:

- Design two processes where one process sends a signal to another process.
- Develop signal handlers in both processes to manage incoming signals and perform specific actions based on the received signal.

8.4 Sample Viva Questions

Question 1: Discuss the characteristics and limitations of unnamed pipes in inter-process communication.

Question 2: How are unnamed pipes created and used in C programming for IPC between parent and child processes?

Question 3: Can unnamed pipes be used for communication between unrelated processes? Explain.

Question 4: What distinguishes named pipes (FIFOs) from unnamed pipes in inter-process communication?

Question 5: What are the advantages of using shared memory for IPC compared to other methods like pipes or message queues?

Question 6: How is shared memory accessed and utilized by multiple processes concurrently?

Question 7: Explain the process of creating and using message queues for communication between processes.

8.5 Evaluation Parameters and Rating Scale

8.5.1 Student's Self Rating

Evaluation Parameters	Rating (Out of 10)
Understanding of Required Concepts	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
How Confident you are in Practical Implementation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Timely Completion of Lab Exercises	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
How confident you are in viva questions	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Lab-manual Readiness on the date of evaluation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Overall Rating	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Student's Signature: _____

Date: _____

8.5.2 Teacher's Rating Student Performance

Evaluation Parameters	Rating (Out of 10)
Understanding of Concepts	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Practical Implementation	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Completion of Lab Exercises	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Student adhering to lab conduct standards	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Lab-manual Readiness	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)
Overall Rating	(1) (2) (3) (4) (5) (6) (7) (8) (9) (10)

Teacher's Signature: _____

Date: _____

8.6 Instructions for the Teacher to Conduct Lab Assessment 4

- Upon finishing the Experiment 7 and Experiment 8, you are required to carry out CAP4, which is worth 100 marks [comprising two parts: J/E (50) and LM (50)].
- The calculation for LM marks is determined by: (Average of the teacher's overall ratings in Experiment 7 and Experiment 8) multiplied by 5.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.

Intentionally left blank for writing solutions to the provided lab exercises.