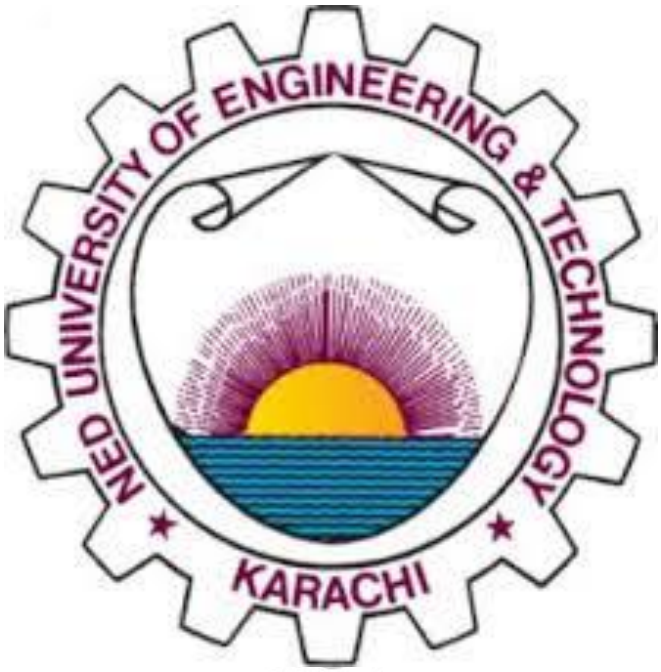
Practical Workbook

SE-303

Operating Systems



Name: _____

Seat #: _____

Batch: _____

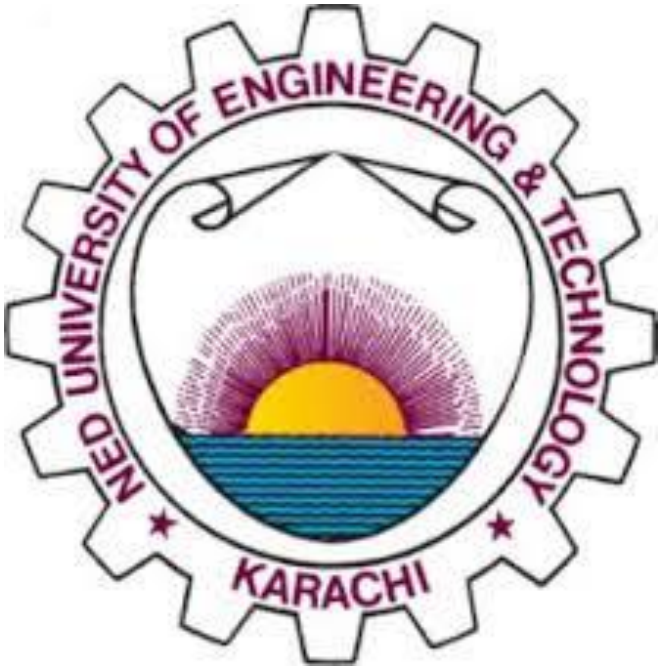
Semester: _____

Year: _____

Practical Workbook

SE-303

Operating Systems



Prepared By

Dr. Mustafa Latif
Assistant Professor

Approved By

Prof. Dr. Shehnila Zardari
Chairperson

DEPARTMENT OF SOFTWARE ENGINEERING, NED UNIVERSITY OF ENGINEERING & TECHNOLOGY

Contents

LAB SESSION 01

Mastering Basic OS Commands

THEORY

Introduction to the Command Line

Welcome to the command-line interface (CLI), or **shell**. It's a powerful text-based program that allows you to interact directly with your operating system. You issue instructions by typing **commands**, and the shell executes them for you.

A command can be one of two things:

1. An **external program**, which is an executable file located on your system (e.g., /bin/ls).
2. A **shell built-in**, which is a command handled directly by the shell itself for efficiency (e.g., cd, pwd, history).

Command Structure

Most commands follow a standard syntax:command [options] [arguments]

- **Command:** The name of the program or built-in you want to run (e.g., ls).
- **Options:** Modify the command's behavior. They are usually preceded by a single dash (-l) for the short form or two dashes (--list) for the long form.
- **Arguments:** The items the command acts upon, typically filenames or directories (e.g., /home/user).

Input/Output Redirection and Piping

The shell provides powerful features for managing a command's input and output, which is normally your keyboard and screen.

- **> (Redirect Output):** Sends the output of a command to a file, **overwriting** the file if it already exists.
    - ls /bin > bin_contents.txt (Saves the list of files in /bin to bin_contents.txt)
- **>> (Append Output):** Sends the output of a command to a file, **appending** to the end of the file without overwriting.
    - ls /etc >> contents.txt (Adds the list of files in /etc to the end of contents.txt)
- **| (Pipe):** Sends the output of one command to be used as the input for a second command. This is one of the most powerful features of the command line.
    - ls /bin | grep "zip" (Lists all files in /bin and then filters that list to show only lines containing "zip")

Essential Linux Commands

Here are some of the most frequently used commands.

1) ls (List)

Lists the contents of a directory.

- ls: Lists contents of the current directory.
- ls /home: Lists contents of the /home directory.
- **Common Options:**
  - -l: Displays a **l**ong format with details (permissions, owner, size, date).
  - -a: Shows **a**ll files, including hidden files that start with a dot (.).
  - -h: Used with -l to show file sizes in **h**uman-readable format (KB, MB, GB).
  - -S: Sorts files by **S**ize, largest first.

2) pwd (Print Working Directory)

Displays the full, absolute path of the directory you are currently in.

3) cd (Change Directory)

Changes your current working directory.

- cd mydir: Changes to the mydir subdirectory.
- cd ..: Moves to the parent directory (one level up).
- cd ~ or just cd: Changes to your user's home directory.

4) less (View File)

Displays the contents of a file one screen at a time, allowing you to scroll up and down. Press q to quit.

- less /etc/passwd: Views the contents of the passwd file.
- **In the less viewer:**
  - Use arrow keys or Page Up/Down to navigate.
  - Type / followed by a word (e.g., /root) and press Enter to search.
  - Press n to find the next occurrence.

5) find (Find Files/Directories)

Searches for files and directories based on criteria like name, size, or modification time.

- find . -name "*.txt": Finds all files ending in .txt in the current directory (.) and all its subdirectories.
- find /home -type d -name "Documents": Searches within /home for a **d**irectory named "Documents".
- **Common Options:**
  - -name: Searches by filename (can use wildcards like *).
  - -type f: Finds only **f**iles.
  - -type d: Finds only **d**irectories.
  - -mtime -2: Finds files **m**odified in the last **2** days.
  - -user ann: Finds files owned by the **user** "ann".

6) grep (Global Regular Expression Print)

Searches for a specific pattern of text inside files.

- grep "root" /etc/passwd: Prints all lines in /etc/passwd that contain the word "root".
- **Common Options:**
  - -i: Performs a case-**i**nsensitive search.
  - -v: In**v**erts the match, showing lines that **do not** contain the pattern.

- -r: Searches **r**ecursively through all files in a directory.
- -l: Prints only the names of fi**l**es that contain the pattern.

7) wc (Word Count)

Counts the lines, words, and characters in a file or from a pipe.

- wc myfile.txt: Shows lines, words, and bytes for myfile.txt.
- **Common Options:**
  - -l: Counts only **l**ines.
  - -w: Counts only **w**ords.

8) su (Substitute User)

Allows you to switch your user identity. **Caution:** Using su to become the root user gives you complete administrative power. Be very careful.

- su: Prompts for the root user's password to become the superuser.
- su jekyll: Prompts for the user jekyll's password to become that user.

9) man (Manual)

Displays the manual page for any command, providing detailed information about its usage and options. Press q to quit.

- man ls: Shows the complete manual for the ls command.

10) history

Displays a list of the commands you have previously executed.

EXERCISE

1. List all files in your home directory, including hidden ones, in the detailed long format with human-readable file sizes.

2. Save the list of all executable programs in the /bin directory into a file in your home folder named bin_list.txt.

3. Find all files in the /etc directory that were modified in the last 5 days.

4. Search for your own username within the /etc/passwd file.

5. List all files in the current directory that end with the .conf extension and sort them by size, with the largest file appearing first.

6. Count the total number of files and directories located in your home directory. (Hint: Pipe the output of one command to another).

7. Recursively search the /home directory for any file containing the text "error" (case-insensitive) and display only the names of the files that contain it.

8. Display the first 10 lines of the history command's output. (Hint: You will need to use a pipe and a command like head).

LAB SESSION 02

Handling Files & Directories

THEORY

The Linux Filesystem

Think of the Linux filesystem as a large digital filing cabinet. The most basic unit is a **file**, which is a distinct chunk of information, like a text document or a program. To keep files organized, we use **directories** (which you can think of as folders).

A key feature of the Linux filesystem is its **tree-like structure**. It all starts from a single, top-level directory called the **root directory**, represented by a slash (/). Every other directory is a **subdirectory** that branches off from the root or another directory.

- A directory that contains another directory is called the **parent directory**.
- A directory inside another is called a **subdirectory**.

Filenames: In Linux, filenames can be up to 256 characters long and are case-sensitive (report.txt is different from Report.txt). You can use letters, numbers, underscores (_), dashes (-), and dots (.). It is best to avoid using spaces or special shell characters like *, ?, & in filenames.

The Home Directory: Every user has a special place to store their personal files called a **home directory**, typically located at /home/username. This is your personal workspace.

The Filesystem Hierarchy Standard (FHS)

Linux organizes system files into standard directories to keep things predictable. Here are some of the most important ones:

Directory	Purpose
/	The **root directory**; the top of the entire filesystem tree.
/home	Contains the personal home directories for all users.
/bin	Holds essential user **bin**aries (executable programs) like ls and cp.
/sbin	Holds essential **s**ystem **bin**aries, mainly for administration.
/etc	Contains system-wide configuration files.
/var	For **var**iable data, such as logs (/var/log).
/tmp	A place for **t**e**mp**orary files.
/dev	Contains special device files that represent hardware (disks, terminals).

Core Commands for File & Directory Management

Creating Files & Directories

- **touch**: The standard way to create a new, empty file.
  - touch new_document.txt
- **mkdir (Make Directory)**: Creates a new, empty directory.

    - mkdir project_files
- **cat (Concatenate)**: A versatile command. You can use it with output redirection (>) to create a text file.
    - cat > story.txt (Lets you type text. Press CTRL+D on a new line to save and exit).

Copying, Moving & Renaming

- **cp (Copy)**: Copies files or directories.
    - cp source.txt destination.txt (Copies a file).
    - cp notes.txt /tmp (Copies notes.txt to the /tmp directory).
    - cp -r project_folder /backup (The -r option recursively copies an entire directory).
- **mv (Move)**: Moves or renames files and directories.
    - mv old_name.txt new_name.txt (Renames a file).
    - mv report.docx /home/user/Documents (Moves a file to a different directory).

Deleting Files & Directories

- **rm (Remove)**: Deletes files. **Use with caution, as this is permanent!**
    - rm old_file.txt
    - -i (interactive) flag will prompt you before every deletion: rm -i *.txt
- **rmdir (Remove Directory)**: Deletes **empty** directories.
    - rmdir old_project
- To delete a directory and all its contents, use rm with the -r (recursive) option.
    - rm -r old_project (**Warning:** This is very powerful and will delete everything inside without confirmation).

File Permissions & Ownership

Every file and directory in Linux has a set of permissions that determines who can read, write, or execute it. When you run ls -l, you see a string like -rwxr-xr--.

- The first character (-) indicates the file type (d for directory, - for a regular file).
- The next nine characters are three sets of permissions for:
    1. **User (Owner):** The person who created the file. (rwx)
    2. **Group:** The user group the file belongs to. (r-x)
    3. **Other:** Everyone else. (r--)

Permission Meanings

Permission	For a File	For a Directory
r (read)	View the contents of the file.	List the contents of the directory (ls).
w (write)	Modify or delete the file.	Create, delete, or rename files inside it.
x (execute)	Run the file (if it's a script/program).	Enter the directory (cd).

Changing Permissions with chmod

The chmod (change mode) command is used to change permissions. The easiest way is with numeric (octal) codes.

Code	Permission
4	Read (r)

Code	Permission
2	Write (w)
1	Execute (x)
0	No Permission (-)

You add the numbers to get the desired permission set. A three-digit code sets permissions for **User**, **Group**, and **Other**.

- chmod 755 script.sh: rwx r-x r-x -> Owner can do everything; others can read and execute. (Common for scripts).
- chmod 644 config.txt: rw- r-- r-- -> Owner can read/write; others can only read. (Common for documents).
- chmod 700 private_folder: rwx --- --- -> Only the owner can access, read, write, and execute. (Common for private directories).

EXERCISE

In this exercise, you will create and manage the file structure for a fictional project.

1. Create the Project Directory

- In your home directory, create a new directory called my_project.

2. Build the Directory Structure

- Navigate into the my_project directory.
- Inside my_project, create three subdirectories: src (for source code), docs (for documentation), and data.

3. Create and Populate Files

- Create an empty file inside the src directory named main.c using the touch command.
- Using the cat command with redirection (>), create a file in the docs directory named readme.txt. Add the following lines of text to it:
- Project Alpha
- This is the main documentation file.

4. Copy and Move Files

- Create a backup of your readme file. Copy docs/readme.txt to docs/readme.bak.
- Create a new directory inside my_project called archive.
- Move the backup file (docs/readme.bak) into the archive directory.

5. Manage Permissions

- Create a simple script: echo "echo Hello World" > src/run.sh
- Use ls -l src/run.sh to view its permissions. Notice it is not executable.
- Use the chmod command to make src/run.sh executable for everyone, but writable only by you (the owner). (Hint: 755).
- Make the readme.txt file read-only for everyone except yourself (the owner), who should have read and write permissions. (Hint: 644).
- Make the data directory completely private so that only you can access it. (Hint: 700).

6. Cleanup

- Remove the entire archive directory and its contents with a single command.

7. Conceptual Questions

- What is the practical difference between the permissions 777 and 775 on a directory?
- What command would you use to find out the file type of an unknown file (e.g., src/main.c)? (Hint: The command is file).
- What is the difference between the cp and mv commands?

NED University of Engineering & Technology
Department of Software Engineering
(SE-303) Operating Systems

Cognitive Domain Assessment Rubric Level C3 (for Laboratory Work)						
Criteria	**Level of Attainment**					
	(5)	**(4)**	**(3)**	**(2)**	**(1)**	**(0)**
Problem Identification	Able to correctly identify the problem(s) and all required data (input and output).	Able to correctly identify the problem(s) but not all the required data (input and output).	Able to partially identify both the problem(s) and the required data (input and output).	Able to partially identify the problem(s) but cannot identify required data (input and output).	Not able to correctly identify the problem(s) and all required data (input and output).	Not tried to identify the problem(s) and all required data (input and output).
Problem Solving Skills	Uses an effective strategy to solve the problem(s) and clearly identifies the steps.	Uses an effective strategy to solve the problem(s) but cannot clearly identify the steps.	Uses an adequate strategy to solve the problem(s) but misses some details.	Uses an inadequate strategy to solve the problem(s) and misses most details.	Uses an ineffective strategy to solve the problem(s) and completely misses the steps.	Didn't use any strategy to solve the problem(s) and completely misses the steps.
Implementation & Debugging	Efficient implementation and excellent debugging skills. Code works without any error.	Efficient implementation and good debugging skills. Minor error in the code.	Adequate implementation and fair debugging skills. Few errors in the code.	Inadequate implementation and debugging skills. Significant errors in the code.	Incomplete implementation and ineffective debugging skills. Major errors in the code.	No implementation and debugging skills.
Deliverable(s)	Generates correct and complete output.	Generates correct but partially complete output.	Generates partially correct and partially complete output.	Generates partially correct and incomplete output.	Cannot generate the desired output.	No output generated.
Depth of Knowledge	Thorough grasp of the concepts covered in the lab and can apply it in real-life situations.	Good grasp of the concepts covered in the lab but cannot effectively apply it in real-life situations.	Fair grasp of the concepts covered in the lab and can vaguely identify a real-life application.	Limited grasp of the concepts covered in the lab and cannot identify any real-life application.	Does not grasp the concepts covered in the lab and cannot identify any real-life application.	Didn't try to grasp the concepts covered in the lab and cannot identify any real-life application.
Weighted CLO Score						
Remarks and Signature						

LAB SESSION 03

Managing System Processes

THEORY

What is a Process?
Everything that runs on a Linux system is a **process**. From the graphical interface you see to the simple commands you type, each is a program in a state of execution. A process is a single running program with its own virtual address space, while a **job** is a task started by the shell, which might consist of one or more processes (e.g., in a command pipeline).

Types of Processes:

- **Interactive Processes:** Started by and interact with a user through a terminal (shell). They can run in the **foreground** (locking your terminal until they finish) or **background** (letting you continue to use the terminal).
- **Daemon Processes:** System-related processes that run in the background without any user interaction. They are typically started at boot and perform essential system tasks (e.g., networking, logging).

1. Viewing and Monitoring Processes
To manage processes, you first need to see what's running.

The ps (Process Status) Command
The ps command provides a snapshot of the currently running processes. It has many options, but two common styles are used most often:

- **BSD Style (ps aux):** Shows all processes for all users. This is a very popular and informative format.
- **System V Style (ps -ef):** Also shows every process on the system, in a different format.

Example ps aux output:

```
 PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
   1  0.0  0.1 169400 13456 ?        Ss   Aug24   0:02 /sbin/init
1234  0.2  0.5 875432 45123 pts/0    Sl+  19:30   0:05 gedit file.txt
1289  0.0  0.0  15640  2456 pts/1    R+   19:55   0:00 ps aux
```

Key Columns Explained:

| Column | Description |

| :--- | :--- |

| USER | The user who owns the process. |

| PID | The Process ID. A unique number for each process. You need this to control a specific process. |

| %CPU | The percentage of CPU time the process is currently using. |

| %MEM | The percentage of system memory (RAM) the process is using. |

| STAT | The current status of the process: R (Running), S (Sleeping), T (Stopped), Z (Zombie - a dead process that hasn't been cleaned up yet). |

| COMMAND | The command that started the process. |

The top Command

While ps is a snapshot, top provides a real-time, interactive view of the system's processes. It continuously updates the display, showing you the most resource-intensive processes at the top.

- To start it, simply type: top
- **Interactive top Commands:**
  - q: **Q**uit top.
  - k: **K**ill a process. It will prompt you for the PID.
  - r: **R**enice a process (change its priority).
  - P: Sort processes by **P**rocessor (CPU) usage (the default).
  - M: Sort processes by **M**emory usage.

2. Controlling Processes

Sending Signals with kill and killall

You don't directly "stop" a process; instead, you send it a **signal**. The kill command is used to send signals to a process using its **PID**.

- **Graceful Termination (SIGTERM - 15):** This is the default. It asks the process to shut down cleanly.
  - kill 1234
- **Forced Termination (SIGKILL - 9):** This is a non-ignorable signal that forces the process to terminate immediately. Use this if the graceful kill doesn't work.
  - kill -9 1234

The **killall** command works similarly but uses the process **name** instead of the PID.

- killall firefox

Managing Process Priorities with nice and renice

You can influence how much CPU time the scheduler gives to a process by adjusting its "niceness" value, which ranges from -20 (highest priority) to +19 (lowest priority).

- **nice**: Starts a new command with a specific priority.
  - nice -n 10 ./my_script.sh (Starts the script with a low priority of 10).
- **renice**: Changes the priority of a running process.
  - renice 15 -p 1234 (Changes the priority of process 1234 to 15).

3. Foreground and Background Job Control

Job control allows you to manage interactive processes within your shell.

- **Run in Background (&):** Add an ampersand (&) to the end of a command to run it in the background immediately, freeing up your terminal.
  - sleep 300 &
- **Stop a Foreground Process (CTRL+Z):** This pauses the currently running process and puts it in the background.
- **jobs**: Lists all jobs (stopped or backgrounded) running from the current shell.
- **fg (Foreground):** Brings a background job to the foreground.
  - fg %1 (Brings job number 1 to the foreground).
- **bg (Background):** Resumes a stopped job and keeps it running in the background.
  - bg %1 (Resumes stopped job number 1 in the background).

EXERCISE

1. Viewing Your Processes
   - Use the ps command to find the PID of your current shell (e.g., bash).
   - Run the top command. While it's running, sort the processes by memory usage. What is the command of the process using the most memory?
   - Exit top.

2. Backgrounding a Process
   - Run the command sleep 500 in the background.
   - Use the jobs command to verify that the process is running in the background.
   - Use ps aux | grep sleep to find the PID of your sleep process.

3. Job Control in Action
   - Run a command that takes a long time, like find / -name "*" > /dev/null 2>&1. (This will search your whole filesystem and may produce permission errors, which we are discarding).
   - After about 5 seconds, stop the process using CTRL+Z.
   - Use the jobs command to see the stopped job.
   - Resume the job in the background using the bg command.
   - Finally, bring the job back to the foreground using the fg command.
   - Terminate it once and for all with CTRL+C.

4. Terminating Processes
   - Using the PID you found in step 2, terminate the sleep 500 process with the standard kill command. Verify with jobs that it is gone.
   - Start another background process: sleep 600 &. This time, use the killall command with the process name to terminate it.

5. Conceptual Questions
   - What is the purpose of the process with PID 1 (often named init or systemd)? Why should you never try to kill it?
   - What is the difference between a normal kill <PID> and kill -9 <PID>? When would you use the second one?
   - What is a "zombie" process? Which command would you use to see if you have any on your system?

LAB SESSION 04

Practice Shell Scripting

THEORY

What is a Shell Script?

Normally, you type commands into the shell one at a time. A **shell script** is simply a text file containing a series of commands. When you run the script, the shell executes these commands in sequence, one after the other. This is incredibly useful for automating repetitive tasks, creating custom commands, and managing complex workflows.

Think of it as writing a recipe for your computer to follow.

Creating and Executing Your First Script

Creating and running a script involves two simple steps:

Step 1: Write the Script Use a text editor (like nano, vim, or gedit) to create a file. The very first line of your script should be the **shebang**, which tells the system which shell to use to interpret the script. For bash, this is #!/bin/bash.

Let's create a file named hello.sh:

```
#!/bin/bash

# This is a comment. The shell ignores it.
echo "Hello, World!"
echo "The current date and time is: $(date)"
```

Step 2: Make the Script Executable By default, new text files do not have permission to be executed. You must grant this permission using the chmod command.

```
chmod +x hello.sh
```

Now, you can run your script by typing its path:

```
./hello.sh
```

Output:

Hello, World!
The current date and time is: Mon Aug 25 20:30:00 PKT 2025

Using Variables

Like any programming language, shell scripting allows you to use variables to store and reuse data. In bash, variables are untyped, meaning they can hold strings, numbers, etc., without special declaration.

Assigning Values: The syntax is VARIABLE_NAME="value". **Important:** There must be **no spaces** around the equals (=) sign. It's a best practice to always enclose the value in double quotes.

```
# Correct syntax
GREETING="Hello there"
USER_COUNT=5

# Incorrect syntax (will cause errors)
# GREETING = "Hello there"
```

Accessing Values: To get the value stored in a variable, precede its name with a dollar sign ($). When using a variable, it's a best practice to enclose it in double quotes to prevent issues with spaces.

```
#!/bin/bash

NAME="Ahmed"
echo "Welcome, $NAME"
```

Command Substitution

Sometimes, you want to store the output of a command in a variable. This is called **command substitution**. The modern, recommended syntax is $(command).

```
#!/bin/bash

# Capture the output of the 'whoami' command into a variable
CURRENT_USER=$(whoami)
echo "The current user is: $CURRENT_USER"

# Capture the list of files in the current directory
FILE_LIST=$(ls)
echo "Files found: $FILE_LIST"
```

Handling Input: Positional Parameters

Your script can accept arguments from the command line, just like regular commands. These are called **positional parameters**.

- $0: The name of the script itself.
- $1: The first argument passed to the script.
- $2: The second argument.
- ... and so on.

Example Script (greet_user.sh):

```
#!/bin/bash

# This script greets a user from a specific city.
# Usage: ./greet_user.sh Ali Karachi

echo "Hello, $1 from $2!"
```

Execution:

```
$ ./greet_user.sh Ali Karachi
Hello, Ali from Karachi!
```

Special Built-in Variables

The shell provides several special variables that are very useful.

Variable	Description
$#	The number of arguments passed to the script.
$?	The **exit status** of the most recently executed command (0 for success, non-zero for failure).
$*	All arguments as a single string.
$@	All arguments as separate, individually quoted strings (generally safer to use).

The Importance of Quotation Marks

Quoting is one of the most important concepts in shell scripting. It controls how the shell interprets special characters.

Type	Example	Behavior
No Quotes	echo $MESSAGE	Risky. Word splitting occurs; special characters are interpreted. Can lead to bugs.
Double Quotes	echo "$MESSAGE"	**Best Practice.** Prevents word splitting but **allows** variable ($VAR) and command ($(cmd)) expansion.
Single Quotes	echo '$MESSAGE'	Takes everything literally. **No** variable or command expansion. The string $MESSAGE is printed.

EXERCISE

1. Basic Greeting Script

- Create a script named welcome.sh that prints a simple welcome message and then on a new line, displays the output of the pwd command.

2. Using Variables

- Create a script named user_info.sh.
- Inside the script, create a variable named USERNAME and set its value to your own name.
- The script should print a message like: "The script owner is: [Your Name]".

3. Using Positional Parameters

- Write a script named file_creator.sh that accepts one argument: a filename.
- The script should create an empty file with the name provided as the argument.
- The script should then print a confirmation message: "File '[filename]' has been created."
- (Hint: Use the touch command and the $1 variable).

4. Using Built-in Variables

- Write a script named arg_check.sh.
- The script should print the following information:
    - "The script's name is: [script name]"
    - "You provided [number] arguments."
- (Hint: Use the $0 and $# variables).

5. Putting It All Together

- Create a script named dir_report.sh that takes a directory path as its only argument.
- The script should perform the following actions:
    1. Print a message: "--- Generating report for directory: [directory path] ---"
    2. Create a variable that stores the number of items in that directory. (Hint: use command substitution with a pipeline like ls [directory path] | wc -l).
    3. Print another message: "The directory contains [number] items."

NED University of Engineering & Technology
Department of Software Engineering
(SE-303) Operating Systems

Cognitive Domain Assessment Rubric Level C3 (for Laboratory Work)						
Criteria	**Level of Attainment**					
	(5)	**(4)**	**(3)**	**(2)**	**(1)**	**(0)**
Problem Identification	Able to correctly identify the problem(s) and all required data (input and output).	Able to correctly identify the problem(s) but not all the required data (input and output).	Able to partially identify both the problem(s) and the required data (input and output).	Able to partially identify the problem(s) but cannot identify required data (input and output).	Not able to correctly identify the problem(s) and all required data (input and output).	Not tried to identify the problem(s) and all required data (input and output).
Problem Solving Skills	Uses an effective strategy to solve the problem(s) and clearly identifies the steps.	Uses an effective strategy to solve the problem(s) but cannot clearly identify the steps.	Uses an adequate strategy to solve the problem(s) but misses some details.	Uses an inadequate strategy to solve the problem(s) and misses most details.	Uses an ineffective strategy to solve the problem(s) and completely misses the steps.	Didn't use any strategy to solve the problem(s) and completely misses the steps.
Implementation & Debugging	Efficient implementation and excellent debugging skills. Code works without any error.	Efficient implementation and good debugging skills. Minor error in the code.	Adequate implementation and fair debugging skills. Few errors in the code.	Inadequate implementation and debugging skills. Significant errors in the code.	Incomplete implementation and ineffective debugging skills. Major errors in the code.	No implementation and debugging skills.
Deliverable(s)	Generates correct and complete output.	Generates correct but partially complete output.	Generates partially correct and partially complete output.	Generates partially correct and incomplete output.	Cannot generate the desired output.	No output generated.
Depth of Knowledge	Thorough grasp of the concepts covered in the lab and can apply it in real-life situations.	Good grasp of the concepts covered in the lab but cannot effectively apply it in real-life situations.	Fair grasp of the concepts covered in the lab and can vaguely identify a real-life application.	Limited grasp of the concepts covered in the lab and cannot identify any real-life application.	Does not grasp the concepts covered in the lab and cannot identify any real-life application.	Didn't try to grasp the concepts covered in the lab and cannot identify any real-life application.
Weighted CLO Score						
Remarks and Signature						

LAB SESSION 05

Logic and Control Flow in Shell Scripting

THEORY

So far, our scripts have been simple sequences of commands. To write truly powerful scripts, we need to add logic: making decisions, repeating actions, and organizing code.

1. Making Decisions (Conditional Statements)

The test Command and [[...]]

To make a decision, we first need to evaluate a condition. In bash, the modern and preferred way to do this is with the double-bracket [[...]] construct. This is a safer and more versatile version of the older single-bracket [...] (or test) command.

[[...]] evaluates the expression inside it and returns an exit status of **0 (success/true)** or **non-zero (failure/false)**.

Common Operators for [[...]] There are three main categories of operators you'll use.

Integer Operators: For comparing numbers.

Operator	Meaning
-eq	is equal to
-ne	is not equal to
-gt	is greater than
-ge	is greater than or equal to
-lt	is less than
-le	is less than or equal to

String Operators: For comparing text.

Operator	Meaning
==	is identical to
!=	is not identical to
-z	string has zero length (is empty)
-n	string is not empty

File Operators: For checking file properties.

Operator	Meaning
-e	file exists
-f	is a regular file
-d	is a directory
-r	is readable
-w	is writable
-x	is executable

The if...elif...else Statement

The if statement runs a block of code only if a condition is true.

Syntax:

```
if [[ condition ]]; then
  # code to run if condition is true
elif [[ another_condition ]]; then
  # code to run if another_condition is true
else
  # code to run if no conditions are true
fi
```

Example:

```
#!/bin/bash
if [[ "$1" -gt 100 ]]; then
  echo "That's a big number!"
elif [[ "$1" -eq 42 ]]; then
  echo "That's the answer to everything."
else
  echo "That's a small number."
fi
```

The case Statement

The case statement is a cleaner way to handle multiple choices when you're comparing a single variable against several possible values.

Syntax:

```
case "$variable" in
  pattern1)
    # code for pattern1
    ;;
  pattern2)
    # code for pattern2
    ;;
  *)
    # default code if no patterns match
```

```
    ;;
esac
```

Example:

```
#!/bin/bash
case "$1" in
  start)
    echo "Starting the service..."
    ;;
  stop)
    echo "Stopping the service..."
    ;;
  *)
    echo "Usage: $0 {start|stop}"
    ;;
esac
```

2. Performing Arithmetic

A critical missing piece for many scripts is the ability to do math. The modern way to perform arithmetic in bash is with the $((...)) construct.

```
x=5
y=10
result=$((x + y))
echo "The sum of $x and $y is $result" # Prints 15

# It can also be used directly in conditions
if [[ $((x * 2)) -eq y ]]; then
  echo "5 times 2 is 10."
fi
```

3. Repeating Actions (Loops)

Loops allow you to run a block of code multiple times.

The for Loop

The for loop is perfect for iterating over a list of items.

```
# Loop over a list of strings
for fruit in apple banana cherry; do
  echo "I like $fruit"
done

# C-style loop for numbers
for (( i=1; i<=5; i++ )); do
  echo "Count: $i"
done
```

The while Loop

The while loop runs as long as its condition is true.

```
#!/bin/bash
counter=1
while [[ $counter -le 5 ]]; do
  echo "Iteration $counter"
  counter=$((counter + 1)) # Increment the counter
done
```

The until Loop

The until loop is the opposite of while: it runs as long as its condition is false.

```
#!/bin/bash
counter=1
until [[ $counter -gt 5 ]]; do
  echo "Iteration $counter"
  counter=$((counter + 1))
done
```

4. Organizing Code with Functions

Functions let you group a block of code under a single name, which you can then call whenever you need it. This makes your scripts cleaner, easier to read, and reusable.

Syntax:

```
function_name() {
  # code to be executed
  # Arguments are passed just like to scripts: $1, $2, etc.
}
```

Example:

```
#!/bin/bash

# Define the function
greet() {
  echo "Hello, $1!"
}

# Call the function
greet "Alice"
greet "Bob"
```

EXERCISE

1. Number Checker
   - Write a script named num_check.sh that takes one number as an argument.
   - The script should use an if...elif...else statement to print whether the number is positive, negative, or zero.
2. File Inspector
   - Write a script named file_info.sh that takes one argument (a name of something in the current directory).
   - The script should use if and file operators (-f, -d) to check and print whether the argument is a regular file, a directory, or does not exist.
3. Simple Animal Facts
   - Write a script named facts.sh that accepts one argument: "cat", "dog", or "bird".
   - Use a case statement to print a simple fact for the given animal. Include a default case for any other input.
4. Countdown Timer
   - Write a script named countdown.sh that uses a C-style for loop to count down from 10 to 1, printing each number. After the loop, it should print "Liftoff!".
5. Summation Calculator
   - Write a script named sum.sh that takes one number as an argument.
   - Using a while loop and the $((...)) syntax for arithmetic, calculate and print the sum of all integers from 1 up to that number. (e.g., if the input is 5, the output should be 15, which is 1+2+3+4+5).
6. Greeting Function
   - Write a script named greeter.sh that defines a function called say_hello.
   - The function should take one argument (a name) and print "Welcome to the team, [name]!".
   - Call the function three times from within the script, with three different names.

LAB SESSION 06

Practice Python Scripting

THEORY

Python is a high-level, interpreted programming language known for its emphasis on code readability and simplicity. Created by Guido van Rossum and first released in 1991, Python's design philosophy allows programmers to express concepts in fewer lines of code.

It's a powerful, general-purpose language used for web development, data science, machine learning, system automation, and more. It features a clean syntax and dynamic typing, which means you don't need to declare a variable's type before using it. This makes Python very approachable for beginners. It also has a massive standard library and an active community that contributes to a rich ecosystem of third-party packages.

A Simple Python Program

The following is a simple Python program that prints a short message.

```
#!/usr/bin/env python3
print("My first Python program")
```

That's the entire program. Save it to a file called my_program.py. You can run it in two common ways on a UNIX or Linux system.

1. Make the file executable using the chmod command and run it directly:

   ```
   $ chmod +x my_program.py
   $ ./my_program.py
   ```

2. Use the Python interpreter to run the script:

   ```
   $ python3 my_program.py
   ```

Comments

The # character indicates that the rest of the line is a comment. The interpreter simply ignores it.

```
# This is a comment. Python will ignore this line.
print("Hello, World!") # This is an inline comment.
```

Python Data Types

Python is case-sensitive, so world and World are two different variables. A variable name must start with a letter or an underscore, followed by letters, numbers, or underscores.

The type of a variable is determined dynamically by the value assigned to it. The primary built-in data types you'll use are:

- **Numeric Types:** Integers (int), floating-point numbers (float).

- **Strings (str):** A sequence of characters.
- **Lists (list):** An ordered, mutable (changeable) collection of items.
- **Dictionaries (dict):** An unordered collection of key-value pairs.

Assigning values is done with the equals sign (=).

Examples:

```
# 1. Numeric and String assignments
item_price = 5        # An integer assignment
item_name = "Apple"    # A string assignment

# Using an f-string for easy printing
print(f"The price of one {item_name} is {item_price} gold coins.")

# 2. List assignments
# The index of the first element is 0.
item_price_list = [5, 8, 24]
item_name_list = ["Apple", "Banana", "Mushroom"]

print(f"The price of one {item_name_list[0]} is {item_price_list[0]} gold coins.")
print(f"The price of one {item_name_list[1]} is {item_price_list[1]} gold coins.")

# 3. Dictionary assignments
# Access elements using their key inside square brackets [].
item_catalog = {"Apple": 5, "Banana": 8, "Mushroom": 24}

print(f"The price of one Apple is {item_catalog['Apple']} gold coins.")

item_name = "Banana"
print(f"The price of one {item_name} is {item_catalog[item_name]} gold coins.")
```

Operators and Conditionals

Comparison Operators

Python uses an intuitive set of operators for comparing values within conditional statements.

Operator	Description	Operator	Description
==	Equal to	and	Logical AND
!=	Not equal to	or	Logical OR
>	Greater than	not	Logical NOT
<	Less than		
>=	Greater than or equal to		
<=	Less than or equal to		

Arithmetic Operators

Operator	Name	Description
+	Addition	Adds two operands
-	Subtraction	Subtracts the right operand from the left
*	Multiplication	Multiplies two operands
/	Division	Divides and returns a floating-point result
//	Floor Division	Divides and returns the largest whole number result
%	Modulus	Returns the remainder of a division
**	Exponentiation	Raises the left operand to the power of the right

Important Note: Python does **not** have the auto-increment (++) or auto-decrement (--) operators. Instead, you use augmented assignment operators like x += 1 and x -= 1.

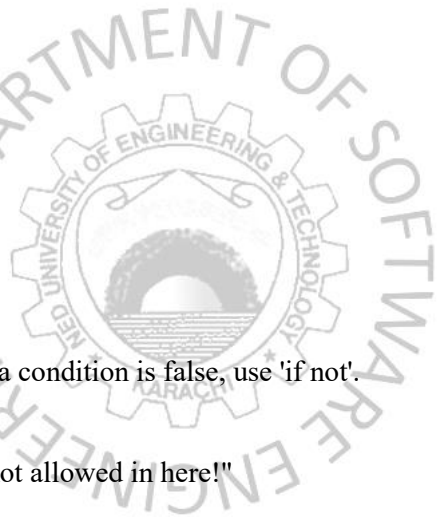
Conditional Statements

Python uses **indentation** (spaces or tabs) to define code blocks. A colon (:) is used to start a block.

```
#!/usr/bin/env python3
name = "master"

if name == "master":
   print("Hello master")
elif name == "teacher":
   print("Hello teacher")
else:
   print("Who are you?")

# To execute a block when a condition is false, use 'if not'.
name = "Bob"
if not name == "Rich":
   print("Go away, you're not allowed in here!"
```



Looping

Python has two main looping constructs: for and while.

for loop

The for loop iterates over a sequence (like a list or range).

```
# Prints numbers from 1 to 10
for i in range(1, 11):
   print(i)

# Loop over each item in a list
import os
file_list = os.listdir('.')
for name in file_list:
   print(name)
```

while loop

The while loop executes as long as a condition is true.

```
x = 0
while x < 10:
   print(x)
   x += 1 # Remember to increment the counter!
```

Loop Control Statements

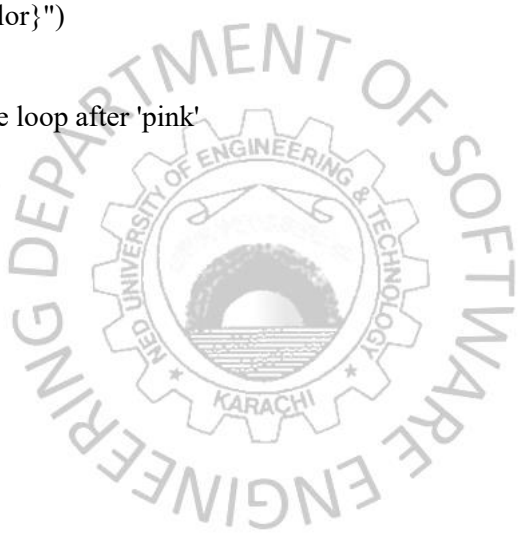
You can alter the flow of a loop using break (to exit the loop entirely) and continue (to skip to the next iteration).

```
colors = ['red', 'blue', 'yellow', 'pink', 'black']
for color in colors:
   if color == 'blue':
      continue  # Skip 'blue' and go to the next color

   print(f"Color: {color}")

   if color == 'pink':
      break  # Exit the loop after 'pink'

print("Exited loop!")
```



EXERCISE

1. Family Shoe Data: A list family holds family member names. A dictionary shoe_color contains the favorite shoe color per person, and a dictionary shoe_size contains the shoe size. Print the favorite shoe color and shoe size for each family member. For shoe sizes 10 and above, add the word 'large' to the output line. *(Expected Output Format: "Homer wears large brown shoes size 12")*

2. Filter Even Numbers: Loop through and print out all even numbers from a given list of numbers. Do not print any numbers that come after 237 in the list.

3. Sequential Arithmetic: Follow these instructions and print the result after each step:
   1. Assign a variable a to a starting value of 5.
   2. Add 6 to the result.
   3. Multiply the result by 2.
   4. Increment the result by 1.
   5. Subtract 9 from the result.
   6. Divide the final result by 7.

4. Disk Space Monitor: Write a Python script that monitors disk space on your system and prints a warning if the usage is above 90%.
Hints:
- You'll need to install a third-party library called psutil. Open your terminal and run: pip install psutil
- Use the following code snippet as a starting point:

```
import psutil

# Get usage statistics for the root directory '/'
usage = psutil.disk_usage('/')

# Get the usage percentage
percent_used = usage.percent

print(f"Disk usage is at {percent_used}%")

# Now, add an if statement to check if it's over the threshold
```

NED University of Engineering & Technology
Department of Software Engineering
(SE-303) Operating Systems

Cognitive Domain Assessment Rubric Level C3 (for Laboratory Work)						
Criteria	**Level of Attainment**					
	(5)	**(4)**	**(3)**	**(2)**	**(1)**	**(0)**
Problem Identification	Able to correctly identify the problem(s) and all required data (input and output).	Able to correctly identify the problem(s) but not all the required data (input and output).	Able to partially identify both the problem(s) and the required data (input and output).	Able to partially identify the problem(s) but cannot identify required data (input and output).	Not able to correctly identify the problem(s) and all required data (input and output).	Not tried to identify the problem(s) and all required data (input and output).
Problem Solving Skills	Uses an effective strategy to solve the problem(s) and clearly identifies the steps.	Uses an effective strategy to solve the problem(s) but cannot clearly identify the steps.	Uses an adequate strategy to solve the problem(s) but misses some details.	Uses an inadequate strategy to solve the problem(s) and misses most details.	Uses an ineffective strategy to solve the problem(s) and completely misses the steps.	Didn't use any strategy to solve the problem(s) and completely misses the steps.
Implementation & Debugging	Efficient implementation and excellent debugging skills. Code works without any error.	Efficient implementation and good debugging skills. Minor error in the code.	Adequate implementation and fair debugging skills. Few errors in the code.	Inadequate implementation and debugging skills. Significant errors in the code.	Incomplete implementation and ineffective debugging skills. Major errors in the code.	No implementation and debugging skills.
Deliverable(s)	Generates correct and complete output.	Generates correct but partially complete output.	Generates partially correct and partially complete output.	Generates partially correct and incomplete output.	Cannot generate the desired output.	No output generated.
Depth of Knowledge	Thorough grasp of the concepts covered in the lab and can apply it in real-life situations.	Good grasp of the concepts covered in the lab but cannot effectively apply it in real-life situations.	Fair grasp of the concepts covered in the lab and can vaguely identify a real-life application.	Limited grasp of the concepts covered in the lab and cannot identify any real-life application.	Does not grasp the concepts covered in the lab and cannot identify any real-life application.	Didn't try to grasp the concepts covered in the lab and cannot identify any real-life application.
Weighted CLO Score						
Remarks and Signature						

LAB SESSION 07

Demonstrate multi-threading using POSIX Thread Library (pthread)

THEORY

The POSIX thread (Pthread) library is a standardized C/C++ API for creating and managing threads. A **thread** is a concurrent execution flow *within the same process*. All threads created within a single process share the same memory space, which makes them lightweight and efficient.
The primary purpose of using Pthreads is to improve application performance. This is achieved in two main ways:

- **Parallelism on Multi-Core Systems:** On machines with multiple CPUs or cores, different threads can be scheduled to run on different cores simultaneously, leading to significant speed gains.
- **Concurrency on Single-Core Systems:** Even on a single core, threads can improve performance. If one thread is blocked (e.g., waiting for a file to be read from a slow disk), another thread can continue executing. This makes the program more responsive and efficient by overlapping computation with I/O latency.

Core Concepts of Threads

Shared vs. Private Resources

Because all threads exist within the same process, they share most resources, but they also have a few private ones.

- **Shared Resources:**
  - Address Space (memory, including global variables)
  - Process Instructions (the code)
  - Open file descriptors
  - Signal handlers
  - Current working directory
  - User and Group ID
- **Private (Per-Thread) Resources:**
  - **Thread ID:** A unique identifier for each thread.
  - **Stack:** For local variables and function call management.
  - **Registers:** Including the program counter and stack pointer.
  - Scheduling priority and policy.
  - Return value (errno).

The Pthreads API

The Pthreads API is a rich library of functions, typically grouped into four categories:

1. **Thread Management:** Functions for creating, joining, and terminating threads (e.g., pthread_create(), pthread_join()).
2. **Mutexes:** Functions for synchronization to prevent data corruption. A **mutex** (short for **mutual exclusion**) acts as a lock to protect shared data.
3. **Condition Variables:** Allow threads to wait for a specific condition to become true before proceeding.
4. **Synchronization:** Advanced routines for managing read/write locks and barriers.

Naming and Compiling:

- All Pthreads functions begin with the prefix pthread_.
- You must include the header file: #include <pthread.h>.
- When compiling with GCC on Linux, you must link the Pthreads library using the -pthread flag:

```
gcc -pthread my_program.c -o my_program
```

Thread Lifecycle Management

a) Creating Threads

Your main() function starts as a single, default thread. You create additional threads using pthread_create().

- int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine)(void *), void *arg);
    - thread: A pointer to a pthread_t variable, which will hold the ID of the newly created thread.
    - attr: Attributes for the thread. Passing NULL selects the default attributes.
    - start_routine: The name of the function the thread will execute.
    - arg: A single void* argument to be passed to the start_routine function. If no argument is needed, pass NULL.

b) Terminating Threads

A thread can terminate in several ways:

1. It returns normally from its start_routine function.
2. It explicitly calls pthread_exit().
3. It is canceled by another thread using pthread_cancel().
4. The entire process terminates (e.g., if main() calls exit() or reaches its end).

CRITICAL POINT: Preventing Premature Process Termination If your main() function finishes and exits before the threads it created are done, the entire process will terminate, killing all active threads.

To prevent this, the main thread must wait for the other threads to complete. This is done using pthread_join() or by having main() call pthread_exit() as its final action, which keeps the process alive until all threads have finished.

c) Joining Threads

Joining is the standard way for a thread to wait for another thread to finish.

- int pthread_join(pthread_t thread, void **retval);

When you call pthread_join(), the calling thread (e.g., main()) will block and wait until the specified thread terminates. This is essential for two reasons:

1. It ensures the main thread doesn't exit prematurely.
2. It allows the main thread to safely collect results from the completed thread.

Synchronization with Mutexes

When multiple threads access and modify the same shared data, you can get a **race condition**, where the final outcome depends on the unpredictable order of execution. This leads to data corruption.

A **mutex** is a lock that ensures only one thread can access a piece of shared data at a time.

How to Use a Mutex

The standard procedure for protecting shared data with a mutex is:

1. **Initialize a Mutex:** Create a pthread_mutex_t variable.
2. **Lock the Mutex:** Before a thread accesses the shared data, it calls pthread_mutex_lock(). If another thread already has the mutex locked, this thread will block and wait until it's available.
3. **Access Shared Data:** The thread performs its operations on the data (this is the "critical section").
4. **Unlock the Mutex:** After it's done, the thread calls pthread_mutex_unlock(), allowing another waiting thread to acquire the lock.
5. **Destroy the Mutex:** When the mutex is no longer needed, free its resources.

Mutex Functions

- **Initialization:**
  - Static: pthread_mutex_t my_mutex = PTHREAD_MUTEX_INITIALIZER;
  - Dynamic: pthread_mutex_init(&my_mutex, NULL);
- **Locking / Unlocking:**
  - pthread_mutex_lock(&my_mutex); (Blocks if locked)
  - pthread_mutex_trylock(&my_mutex); (Returns immediately if locked)
  - pthread_mutex_unlock(&my_mutex);
- **Destruction:**
  - pthread_mutex_destroy(&my_mutex);

Example: Vector Dot Product

Here is a simple serial program that computes the dot product of two vectors. The multi-threaded version can be `dotprod_mutex.c`

```
/******************************************************************
***************
* FILE: dotprod_serial.c
* DESCRIPTION: A simple serial program to compute a vector dot
product.
******************************************************************
**************/
#include <stdio.h>
#include <stdlib.h>

// This structure holds all data needed for the calculation.
typedef struct {
    double *a;
    double *b;
    double sum;
    int    veclen;
} DOTDATA;

#define VECLEN 100000
DOTDATA dotstr;

// The function that performs the dot product calculation.
void dotprod() {
    int start, end, i;
    double mysum, *x, *y;

    start = 0;
    end = dotstr.veclen;
    x = dotstr.a;
    y = dotstr.b;

    // Perform the dot product
    mysum = 0;
    for (i = start; i < end; i++) {
        mysum += (x[i] * y[i]);
    }
    dotstr.sum = mysum;
}
```

```
   // The main program initializes data, calls the function, and
   prints the result.
   int main (int argc, char *argv[]) {
       int i, len;
       double *a, *b;

       // Assign storage and initialize values
       len = VECLEN;
       a = (double*) malloc(len * sizeof(double));
       b = (double*) malloc(len * sizeof(double));

       for (i = 0; i < len; i++) {
           a[i] = 1.0;
           b[i] = a[i];
       }

       dotstr.veclen = len;
       dotstr.a = a;
       dotstr.b = b;
       dotstr.sum = 0;

       // Perform the dot product
       dotprod();

       // Print result and release storage
       printf("Sum =  %f \n", dotstr.sum);
       free(a);
       free(b);

       return 0;
}
```

EXERCISE

1) Write a C program that creates five threads using the pthread_create() routine. Each thread should perform the following simple tasks:

- Print a message, such as "Hello from Thread #X!", where X is its identifier.
- Terminate cleanly using pthread_exit().

The main() function should wait for all five threads to complete their execution before it prints a final "All threads are done." message and exits.

2) Write a C program that creates multiple threads and passes a unique integer ID to each one.

Requirements:

- The main() function should create four threads.
- For each thread, you must pass it a unique integer (e.g., Thread 0 gets 0, Thread 1 gets 1, etc.).
- Important: You must ensure that each thread receives the correct, unique integer. To do this safely, create a separate, persistent data structure (like a struct) for each thread's arguments. Simply passing the address of a single loop variable will cause a race condition and is incorrect.
- Each thread will receive the pointer to its data structure, extract the integer, print a message like "Thread received ID: X", and then exit.
- The main() function must wait for all threads to complete using pthread_join().

3) Write a C program that demonstrates how to parallelize the summation of an array using four threads.

Requirements:

1. In main(), create a large integer array (e.g., 1,000,000 elements) and initialize all its elements to 1.
2. Create a single global variable, long long global_sum = 0;, which will hold the final sum.
3. Initialize a global mutex variable (pthread_mutex_t) to protect access to global_sum.
4. Create four worker threads. Each thread will be responsible for summing a specific quarter of the array. For example:
    - Thread 0 sums indices 0 to 249,999.
    - Thread 1 sums indices 250,000 to 499,999.
    - ...and so on.
5. Each thread should calculate its partial sum in a local variable. Once its calculation is complete, it must lock the mutex, add its partial sum to the global_sum, and then immediately unlock the mutex.
6. The main() function must wait for all four threads to complete using pthread_join().
7. After all threads are joined, main() should print the final global_sum. The correct result should be the total number of elements in the array.

4) Compile and execute the provided dotprod_serial.c program. Observe its behavior and confirm that it produces the correct output (Sum = 100000.000000). Attach a screenshot of this output.

LAB SESSION 08

Open Ended Lab

OBJECTIVE

The goal of this lab is to apply your knowledge of POSIX threads to a practical problem. You will convert a sequential program that computes the dot product of two vectors into a multi-threaded version. This will require you to manage thread creation, partition work, and use a **mutex** to protect a shared global variable from **race conditions**, ensuring a correct final result.

EVALUATION METHOD

Students will be evaluated through open-ended problem-solving tasks using the OEL rubrics (5 marks) with criteria levels 0 and 1.

PROVIDED CODE: dotprod_serial.c

This is the sequential program you will start with.

```
/******************************************************************************
* FILE: dotprod_serial.c
* DESCRIPTION: A simple serial program to compute a vector dot product.
******************************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <time.h> // For measuring time

// This structure holds all data needed for the calculation.
typedef struct {
   double *a;
   double *b;
   double sum;
   int    veclen;
} DOTDATA;

#define VECLEN 1000000 // Using a larger vector for more meaningful timing
DOTDATA dotstr;

// The function that performs the dot product calculation.
void dotprod() {
   int i;
   double mysum, *x, *y;

   x = dotstr.a;
   y = dotstr.b;

   // Perform the dot product
```

```
    mysum = 0;
    for (i = 0; i < dotstr.veclen; i++) {
        mysum += (x[i] * y[i]);
    }
    dotstr.sum = mysum;
}

// The main program initializes data, calls the function, and prints the result.
int main (int argc, char *argv[]) {
    int i, len;
    double *a, *b;

    // Assign storage and initialize values
    len = VECLEN;
    a = (double*) malloc(len * sizeof(double));
    b = (double*) malloc(len * sizeof(double));

    for (i = 0; i < len; i++) {
        a[i] = 1.0;
        b[i] = a[i];
    }

    dotstr.veclen = len;
    dotstr.a = a;
    dotstr.b = b;
    dotstr.sum = 0;

    // --- Time Measurement Start ---
    clock_t start = clock();

    // Perform the dot product
    dotprod();

    // --- Time Measurement End ---
    clock_t end = clock();
    double time_spent = (double)(end - start) / CLOCKS_PER_SEC;

    // Print result and release storage
    printf("Serial Execution Finished.\n");
    printf("Sum =  %f \n", dotstr.sum);
    printf("Execution Time: %f seconds\n", time_spent);

    free(a);
    free(b);

    return 0;
}
```

TASKS

Task 1 – Analyze the Serial Version

1. **Compile and Execute:** Compile the provided dotprod_serial.c program.
   gcc dotprod_serial.c -o dotprod_serial
2. **Run and Record:** Execute the program and record its output, paying close attention to the final sum and the execution time.
   ./dotprod_serial
3. **Discussion Question:** In your report, explain why this serial program is inherently free of race conditions and does not require any synchronization mechanisms.

Task 2 – Implement the Multi-Threaded Version

Create a new program named dotprod_mutex.c. This program must parallelize the dot product calculation.

Requirements:

1. **Use 4 Threads:** The program should be hardcoded to use four worker threads.
2. **Divide the Work:** The vector of VECLEN elements must be divided equally among the four threads. Each thread will be responsible for computing the dot product for its assigned chunk.
3. **Thread Function:** The thread's start routine will calculate a partial sum for its chunk.
4. **Synchronization:** Use a single global pthread_mutex_t variable. Each thread must **lock** this mutex before adding its partial sum to the global dotstr.sum and **unlock** it immediately afterward.
5. **Wait for Completion:** The main thread must use pthread_join() to wait for all four worker threads to finish before it prints the final result and execution time.

Hints:

- You will need to define a struct to pass multiple pieces of information to each thread (e.g., its unique ID, and perhaps the start and end index of its chunk).
- Initialize your mutex in main() before creating the threads.

Task 3 – Compare and Analyze

Once your dotprod_mutex.c program is working correctly, perform a comparative analysis.

1. **Verify Correctness:** Run both the serial and the multi-threaded versions. Is the final Sum identical in both programs? It should be.
2. **Demonstrate the Race Condition:** Temporarily comment out the pthread_mutex_lock() and pthread_mutex_unlock() calls in your multi-threaded program. Recompile and run it several times. What happens to the final Sum? Is it correct? Is it consistent? Explain this behavior in your report.
3. **Performance Analysis:** Compare the execution time of dotprod_serial.c with dotprod_mutex.c. Did you observe a speedup or a slowdown? Discuss why this might be the case, considering factors like thread creation overhead and mutex contention.

DELIVERABLES

Submit the following for evaluation:

1. The complete source code for your dotprod_mutex.c program.
2. Screenshots showing the output of:
   - The original dotprod_serial.c.
   - Your correct dotprod_mutex.c.
   - The output of your multi-threaded version with the mutex disabled, showing an incorrect result.
3. A brief written report answering the discussion questions from Task 1 and Task 3.

NED University of Engineering & Technology
Department of Software Engineering
OEL RUBRICS
(SE-303) Operating Systems

Criteria	Achieved (1)	Not Achieved (0)	Marks
Problem Identification	Able to correctly identify the problem(s) and all required data (input and output)	Not able to correctly identify the problem(s) and all required data (input and output).	
Problem Solving Skills	Uses an effective strategy to solve the problem(s) and clearly identifies the steps	Uses an ineffective strategy to solve the problem(s) and completely misses the steps	
Implementation & Debugging	Efficient implementation and excellent debugging skills. Solution works without any error	Incomplete implementation and ineffective debugging skills. Major errors in the implementation.	
Deliverable(s)	Generates correct and complete output.	Cannot generate the desired output.	
Depth of Knowledge	Thorough grasp of the concepts covered in the lab and can apply it in real-life situations	Does not grasp the concepts covered in the lab and cannot identify any real-life application	
Weighted CLO Score			
Remarks and Signature			

LAB SESSION 09

Design and Simulation of CPU Scheduling Algorithms

THEORY

In a multi-programming operating system, the CPU must be scheduled to switch between multiple ready-to-run processes to maximize utilization and provide fair service. A **CPU scheduling algorithm** determines which process in the ready queue is allocated to the CPU.

Key Scheduling Algorithms

- **First-Come, First-Served (FCFS):** The simplest algorithm where processes are executed in the order they arrive. It's a non-preemptive algorithm. While fair, it can lead to long average waiting times if a short process gets stuck behind a long one.
- **Priority Scheduling:** Each process is assigned a priority, and the CPU is allocated to the process with the highest priority. A major challenge is **starvation**, where low-priority processes may never execute.
- **Round Robin (RR):** A preemptive algorithm where each process is given a small unit of CPU time called a **time quantum**. When the quantum expires, the process is moved to the end of the ready queue. This ensures that all processes get a fair share of the CPU.
- **Shortest-Job-First (SJF):** This algorithm selects the process with the smallest burst time to run next. It is provably optimal for minimizing the average waiting time but requires knowing the burst time in advance.

Designing a Simulator in Python

Python is an excellent language for simulating OS concepts due to its readability and powerful data structures. To properly model a process, we will use a Python **class**. This is an effective design choice as it allows us to encapsulate all the data related to a process in one neat package.

The Process Component

A well-designed Process class will be the core component of our simulator.

```
class Process:
    def __init__(self, pid, arrival_time, burst_time, priority=0):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.priority = priority
        # These will be calculated by the simulator
        self.completion_time = 0
        self.turnaround_time = 0
        self.waiting_time = 0
        self.start_time = 0
```

Example Implementation: FCFS Scheduler

This runnable script implements the FCFS algorithm, including the Process class, calculation/display functions, the scheduler, and a main function to run the simulation.

```
import copy
class Process:
    """A class to represent a process with its scheduling attributes."""
    def __init__(self, pid, arrival_time, burst_time, priority=0):
        self.pid = pid
        self.arrival_time = arrival_time
        self.burst_time = burst_time
        self.priority = priority
        # These will be calculated by the simulator
        self.completion_time = 0
        self.turnaround_time = 0
        self.waiting_time = 0
        self.start_time = 0

# --- Utility Functions for Calculation and Display ---

def calculate_metrics(processes):
    """Calculates turnaround and waiting time for a list of completed processes."""
    total_turnaround_time = 0
    total_waiting_time = 0
    for p in processes:
        p.turnaround_time = p.completion_time - p.arrival_time
        p.waiting_time = p.turnaround_time - p.burst_time
        total_turnaround_time += p.turnaround_time
        total_waiting_time += p.waiting_time

    avg_turnaround_time = total_turnaround_time / len(processes)
    avg_waiting_time = total_waiting_time / len(processes)

    return avg_turnaround_time, avg_waiting_time

def print_results(title, processes, avg_turnaround, avg_waiting):
    """Prints the final results in a formatted table."""
    processes.sort(key=lambda x: x.pid)
    print(f"\n--- {title} Scheduling Results ---")
    print("=" * 70)
    print("PID | Arrival | Burst | Priority | Completion | Turnaround | Waiting")
    print("-" * 70)
    for p in processes:
        print(f"{p.pid:^3} | {p.arrival_time:^7} | {p.burst_time:^5} | {p.priority:^8} | "
              f"{p.completion_time:^10} | {p.turnaround_time:^10} | {p.waiting_time:^7}")
    print("-" * 70)
    print(f"Average Turnaround Time: {avg_turnaround:.2f}")
    print(f"Average Waiting Time: {avg_waiting:.2f}")
    print("=" * 70)

# --- FCFS Scheduling Algorithm Implementation ---

def fcfs_schedule(processes):
```

```
    """
    First-Come, First-Served scheduling algorithm.
    Sorts by arrival time and executes in that order.
    """
    # Sort processes based on arrival time
    processes_sorted = sorted(processes, key=lambda p: p.arrival_time)
    current_time = 0

    for p in processes_sorted:
        if current_time < p.arrival_time:
            current_time = p.arrival_time

        p.start_time = current_time
        p.completion_time = current_time + p.burst_time
        current_time = p.completion_time

    avg_tt, avg_wt = calculate_metrics(processes_sorted)
    print_results("First-Come, First-Served (FCFS)", processes_sorted, avg_tt, avg_wt)
    return avg_tt, avg_wt

# --- Main Function to Run the Simulator ---

if __name__ == "__main__":
    # Define a sample workload: list of Process objects
    # (PID, Arrival Time, Burst Time, Priority)
    workload = [
        Process(1, 0, 8, 3),
        Process(2, 1, 4, 1),
        Process(3, 2, 9, 4),
        Process(4, 3, 5, 2),
    ]

    print("--- Starting FCFS Simulation ---")
    # Run the FCFS scheduler with a deep copy of the workload
    fcfs_schedule(copy.deepcopy(workload))
```

EXERCISE

1. Implement the Priority Scheduling Component

Design and implement a non-preemptive priority scheduling algorithm.

- Logic:
    1. The algorithm must select the process with the highest priority from the ready queue. Assume a lower number indicates a higher priority.
    2. You must handle processes that arrive while another is running. The active ready queue should always be sorted by priority.
    3. If two processes have the same priority, the one that arrived earlier (FCFS) should be chosen as the tie-breaker.
- Deliverable:
    - A Python function priority_schedule(processes) that implements this logic and uses the provided print_results function to display the outcome.

2. Implement the Round Robin (RR) Component

Design and implement a preemptive Round Robin scheduler. This is a more complex design challenge.

- Logic:
    1. You will need a ready queue to manage processes. (A deque from Python's collections module is ideal for this).
    2. A process runs for one time quantum.
    3. If a process finishes before its quantum expires, it exits the system.
    4. If it does not finish, it is preempted and moved to the end of the ready queue.
    5. New processes are added to the end of the ready queue as they arrive.
- Deliverable:
    - A Python function rr_schedule(processes, time_quantum) that implements the RR logic and uses the provided print_results function.

3. Analysis and Comparison

The final step is to use your simulator to analyze the performance of the components you designed and compare them against the provided FCFS implementation.

1. Create a Test Workload: Use the sample workload from the main function or create your own list of at least 5 Process objects with varying arrival times, burst times, and priorities.
2. Run Simulations: Modify the main driver block to run this workload through the provided fcfs_schedule function as well as your two new functions, priority_schedule and rr_schedule.
3. Analyze Results: Create a summary table comparing the Average Waiting Time and Average Turnaround Time for all three algorithms.
4. Conclusion: Write a short conclusion answering the following:
    - Based on your results, which of the three algorithms performed best for your specific test workload and why?
    - Under what circumstances would you choose RR over FCFS, even if its average waiting time is higher?
    - What is the main drawback of the Priority scheduling algorithm you designed?

LAB SESSION 10

Design and Simulation of Page Replacement Algorithms

THEORY

In an operating system that uses paging for virtual memory, the physical memory is divided into fixed-size blocks called **frames**. When the system needs to bring a new page into memory and all frames are already occupied, a **page replacement algorithm** must decide which existing page to evict (swap out) to make room.

The efficiency of a page replacement algorithm is measured by its ability to minimize the number of **page faults**. A page fault occurs when the CPU references a page that is not currently in physical memory, forcing the OS to load it from the disk, which is a very slow operation.

Key Page Replacement Algorithms

- **First-In, First-Out (FIFO):** This is the simplest algorithm. The OS maintains a queue of all pages currently in memory. When a replacement is needed, the page at the front of the queue (the one that has been in memory the longest) is evicted. While simple to implement, FIFO can perform poorly as it may evict a frequently used page.
- **Least Recently Used (LRU):** This algorithm is based on the principle of locality, which suggests that pages used recently are likely to be used again soon. When a replacement is needed, LRU evicts the page that has not been accessed for the longest period. It generally performs much better than FIFO but is more complex to implement as the OS must track when each page was last used.

Designing a Simulator in Python

Python's clear syntax and data structures make it ideal for simulating page replacement algorithms. Our simulator will process a **page reference string** (a sequence of page numbers being requested) and track the state of memory frames over time.

Example Implementation: FIFO Page Replacement

Here is a complete, runnable script that simulates the FIFO algorithm. It includes a main function to run the simulation with a sample workload. Use this as a foundation for your own work.

```
from collections import deque

def fifo_page_replacement(page_reference_string, num_frames):
    """
    Simulates the First-In, First-Out (FIFO) page replacement algorithm.

    Args:
        page_reference_string (list): The sequence of page numbers
requested.
        num_frames (int): The number of available frames in memory.

    Returns:
        int: The total number of page faults.
    """
    print("--- Starting FIFO Simulation ---")
    print(f"Page Reference String: {page_reference_string}")
```

```
    print(f"Number of Frames: {num_frames}\n")

    frames = []  # Represents the physical memory frames
    page_faults = 0

    # Use a deque as a queue to easily track the first-in page
    page_queue = deque()

    for page in page_reference_string:
        # Check if the page is already in a frame (a "hit")
        if page in frames:
            status = "Hit"
        # If not in a frame, a "fault" occurs
        else:
            page_faults += 1
            status = "Fault"
            # If there is space in memory, simply add the page
            if len(frames) < num_frames:
                frames.append(page)
                page_queue.append(page)
            # If memory is full, replace the oldest page
            else:
                page_to_replace = page_queue.popleft() # Get the oldest page
                frame_index = frames.index(page_to_replace)
                frames[frame_index] = page # Replace it with the new page
                page_queue.append(page) # Add new page to the queue

        # Print the state at each step for clarity
        print(f"Request: {page}, Frames: {list(frames)}, Status: {status}")

    print("\n--- FIFO Simulation Complete ---")
    print(f"Total Page Faults: {page_faults}")
    return page_faults

# --- Main Function to Run the Simulator ---

if __name__ == "__main__":
    # Define a sample workload
    reference_string = [7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1,
7, 0, 1]
    frame_count = 3

    # Run the FIFO simulation
    fifo_page_replacement(reference_string, frame_count)
```

EXERCISE

1. Implement the LRU Scheduling Component

Design and implement the Least Recently Used (LRU) page replacement algorithm.

- Logic:
    1. When a page fault occurs and memory is full, you must identify and evict the page that has been unused for the longest time.
    2. To achieve this, you need a mechanism to track the "recency" of each page's use. Every time a page is referenced (either on a hit or a fault), it becomes the *most* recently used.
- Hint: A simple way to track recency is to maintain a list or queue. When a page is accessed, you can move it to the "most recent" end of the list. The page at the "least recent" end is the one to evict.
- Deliverable:
    - A Python function `lru_page_replacement(page_reference_string, num_frames)` that implements this logic and prints the step-by-step state of memory and the final page fault count, just like the FIFO example.

2. Analysis and Comparison

The final step is to analyze the performance of the component you designed.

1. Run Simulations: In your main script block, run both the `fifo_page_replacement` and your new `lru_page_replacement` functions using the same `reference_string` and `frame_count`.
2. Analyze Results: Create a simple summary table in your output that compares the Total Page Faults for both algorithms.
3. Answering the following:
    - Based on your simulation results, which algorithm performed better for this specific workload and why?
    - Explain the fundamental difference in the replacement strategy between FIFO and LRU that leads to this performance difference.

LAB SESSION 11

Design and Simulation of Memory Allocation Algorithms

THEORY

Memory management is a core function of an operating system. The memory manager must dynamically allocate blocks of memory to processes when they are requested and free those blocks when the processes terminate. For contiguous allocation, the manager maintains a list of free memory blocks (holes) and searches this list to fulfill a process's request.

Key Allocation Strategies

- **First Fit:** The memory manager scans the list of holes from the beginning and allocates the **first** hole that is large enough to accommodate the process. It's fast but can leave behind unusable small fragments.
- **Best Fit:** The manager searches the **entire** list of holes to find the **smallest** hole that is large enough for the process. This strategy aims to minimize the size of the leftover hole (internal fragmentation).
- **Worst Fit:** The manager searches the entire list and allocates the **largest** available hole. The idea is that the leftover fragment will be large enough to be useful for future processes.

Designing a Simulator in Python

To model this system, we can use a class-based approach in Python, which is a clean and effective design. We'll represent each memory block and process as an object.

- **MemoryBlock:** A class to represent a block of memory. It will have attributes for its size, whether it's free, and which process (if any) is currently using it.
- **Process:** A simple class to represent a process with its required memory size.

Example Implementation: First Fit Memory Allocation

Here is a complete, runnable Python script that simulates the First Fit algorithm. Use this code as the foundation for your own implementations.

```
import copy
# --- Component Design: Classes for Memory and Processes ---
class MemoryBlock:
    def __init__(self, size, is_free=True, process_id=None):
        self.size = size
        self.is_free = is_free
        self.process_id = process_id

class Process:
    def __init__(self, pid, size):
        self.pid = pid
        self.size = size

# --- Utility Function to Display Memory State ---
def print_memory_layout(memory):
    print("\n--- Memory Layout ---")
    layout = "|"
```

```
    for block in memory:
        if block.is_free:
            layout += f" Free({block.size}K) |"
        else:
            layout += f" P{block.process_id}({block.size}K) |"
    print(layout)
    print("-" * len(layout))

# --- First Fit Algorithm Implementation ---
def first_fit(initial_holes, processes):
    """Simulates the First Fit memory allocation algorithm."""
    print("\n\n>>>--- Starting First Fit Simulation ---<<<")
    memory = [MemoryBlock(size=h) for h in initial_holes]

    for process in processes:
        allocated = False
        for i, block in enumerate(memory):
            if block.is_free and block.size >= process.size:
                # This is the first block that fits. Allocate here.
                print(f"Allocating Process P{process.pid} (size {process.size}K) to first available hole of size
{block.size}K.")

                # If the block is larger than the process, split it
                if block.size > process.size:
                    remaining_size = block.size - process.size
                    new_block = MemoryBlock(size=remaining_size)
                    memory.insert(i + 1, new_block)

                # Update the current block to be allocated
                block.size = process.size
                block.is_free = False
                block.process_id = process.pid

                allocated = True
                break # Move to the next process

        if not allocated:
            print(f"Process P{process.pid} (size {process.size}K) could not be allocated.")

    print_memory_layout(memory)
    return memory


# --- Main Function to Run the Simulator ---
if __name__ == "__main__":
    # Define initial memory state and processes
    initial_memory_holes = [100, 500, 200, 300, 600] # Sizes in KB
    processes_to_allocate = [
        Process(1, 212), Process(2, 417), Process(3, 112), Process(4, 426)
    ]

    # Run the First Fit simulation
    first_fit_memory = first_fit(copy.deepcopy(initial_memory_holes), processes_to_allocate)
```

EXERCISE

1. Implement the Best Fit Component

   Design and implement the Best Fit memory allocation algorithm.

   - Logic: For each process, you must search the entire list of free memory blocks. You need to find the block that is large enough for the process but also results in the smallest possible leftover fragment (i.e., block.size - process.size should be minimized).
   - Deliverable: A Python function best_fit(initial_holes, processes) that implements this logic and prints the final memory layout.

2. Implement the Worst Fit Component

   Design and implement the Worst Fit memory allocation algorithm.

   - Logic: For each process, you must search the entire list of free memory blocks. You need to find the block that is large enough for the process and results in the largest possible leftover fragment (i.e., block.size - process.size should be maximized).
   - Deliverable: A Python function worst_fit(initial_holes, processes) that implements this logic and prints the final memory layout.

3. Analysis and Comparison

   The final step is to use your simulator to analyze the fragmentation caused by each algorithm.

   1. Run Simulations: Modify your main block to run all three algorithms (the provided first_fit, your best_fit, and your worst_fit) using the same initial memory holes and process list.
   2. Analyze Results: After each simulation, calculate and print the total amount of free memory and the size of the largest contiguous free block.
   3. Conclusion: Write a short conclusion answering the following:
      - Based on your results, which algorithm was most successful at preserving large, contiguous free blocks of memory for this specific workload?
      - Which algorithm resulted in the most external fragmentation (i.e., many small, useless holes)?
      - What is the primary trade-off between Best Fit and Worst Fit in terms of memory utilization?

LAB SESSION 12

Solving the Producer-Consumer Problem with Semaphores

THEORY

The Producer-Consumer Problem

This is a classic synchronization problem where two types of processes (or threads), **Producers** and **Consumers**, share a common, fixed-size buffer.

- The **Producer's** job is to generate data (items) and put them into the buffer.
- The **Consumer's** job is to remove items from the buffer and consume them.

A correct solution must satisfy three critical conditions:

1. Producers must not add items to the buffer if it is full.
2. Consumers must not remove items from the buffer if it is empty.
3. Producers and Consumers must not access the buffer at the same time (mutual exclusion).

Designing the Solution with Semaphores

To solve this, we use three synchronization primitives. In Python's threading module, these are Semaphore objects and a Lock (which acts as a mutex).

1. **empty (A Counting Semaphore):**
    - **Purpose:** Tracks the number of empty slots in the buffer.
    - **Initial Value:** The size of the buffer (e.g., N).
    - **Used by:** The Producer. Before adding an item, a producer must acquire() this semaphore (decrementing the count of empty slots). If empty is zero, the producer will block until a consumer frees up a slot.
2. **full (A Counting Semaphore):**
    - **Purpose:** Tracks the number of filled slots in the buffer.
    - **Initial Value:** 0.
    - **Used by:** The Consumer. Before removing an item, a consumer must acquire() this semaphore (decrementing the count of filled slots). If full is zero, the consumer will block until a producer adds an item.
3. **mutex (A Lock / Binary Semaphore):**
    - **Purpose:** Ensures mutual exclusion. Only one thread (producer or consumer) can be physically manipulating the buffer's data structure at any given moment.
    - **Initial Value:** 1 (unlocked).
    - **Used by:** Both Producers and Consumers. A thread must acquire() the lock before accessing the buffer and release() it immediately after.

Python Implementation Framework

You are provided with the following Python script skeleton. Your task is to complete the producer and consumer functions by adding the correct semaphore logic.

```
# producer_consumer.py
import threading
import time
import random
from collections import deque

# --- Configuration ---
BUFFER_SIZE = 5
PRODUCER_COUNT = 2
CONSUMER_COUNT = 3

# --- Shared Resources ---
buffer = deque(maxlen=BUFFER_SIZE)

# --- Synchronization Primitives ---
# empty: Tracks empty slots. Producers wait on this.
empty = threading.Semaphore(BUFFER_SIZE)

# full: Tracks filled slots. Consumers wait on this.
full = threading.Semaphore(0)

# mutex: Ensures mutual exclusion for buffer access.
mutex = threading.Lock()


# --- Producer Thread ---
def producer(producer_id):
    global buffer
    item_counter = 0
    while True:
        item = f"Item-{producer_id}-{item_counter}"

        # --- EXERCISE: Add producer synchronization logic here ---

        # Simulate producing an item
        time.sleep(random.uniform(0.5, 2))

        # Add to buffer
        buffer.append(item)
        print(f"Producer {producer_id} produced {item}. Buffer: {list(buffer)}")

        # --- EXERCISE: Add producer synchronization logic here ---

        item_counter += 1

# --- Consumer Thread ---
def consumer(consumer_id):
```

```
    global buffer
    while True:
        # --- EXERCISE: Add consumer synchronization logic here ---

        # Remove from buffer
        item = buffer.popleft()
        print(f"Consumer {consumer_id} consumed {item}. Buffer: {list(buffer)}")

        # --- EXERCISE: Add consumer synchronization logic here ---

        # Simulate consuming an item
        time.sleep(random.uniform(1, 3))

# --- Main Application ---
if __name__ == "__main__":
    threads = []

    # Create and start producer threads
    for i in range(PRODUCER_COUNT):
        thread = threading.Thread(target=producer, args=(i,))
        threads.append(thread)
        thread.start()

    # Create and start consumer threads
    for i in range(CONSUMER_COUNT):
        thread = threading.Thread(target=consumer, args=(i,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete (they won't in this infinite loop example)
    for thread in threads:
        thread.join()
```

EXERCISE

1. Implement the Synchronized Logic

   Complete the producer and consumer functions in the producer_consumer.py script by adding the correct calls to acquire() and release() on the empty, full, and mutex primitives.

   - Producer Logic:
     1. A producer must first wait for an empty slot to be available.
     2. Then, it must acquire the lock to ensure exclusive access to the buffer.
     3. After adding the item, it must release the lock.
     4. Finally, it must signal that a new item is available for consumers.
   - Consumer Logic:
     1. A consumer must first wait for a filled slot to be available.
     2. Then, it must acquire the lock for exclusive access.
     3. After removing the item, it must release the lock.
     4. Finally, it must signal that a new empty slot is available for producers.

   Deliverable:

   - The complete, working producer_consumer.py script. Attach a printout of your code and a screenshot of the output showing producers and consumers running concurrently without error.

2. Analysis and Verification

   After completing the implementation, answer the following conceptual questions.

   1. Role of mutex: What would happen if you removed the mutex.acquire() and mutex.release() calls but kept the empty and full semaphores? Describe a specific race condition that could occur.
   2. Role of empty: What problem does the empty semaphore solve? What would happen if it were removed?
   3. Role of full: What problem does the full semaphore solve? What would happen if it were removed?

LAB SESSION 13

Complex Engineering Activity

CLO Covered

CLO 3: Apply understanding of design and development principles in the construction of Operating Systems Components. **(C3-PLO3)**

CPA1: Depth of analysis required: Have no obvious solution and require abstract thinking, originality in analysis to formulate suitable models.
CPA2: Level of interaction: Require resolution of significant problems arising from interactions between wide-ranging or conflicting technical, engineering or other issues
CPA3: Familiarity: Can extend beyond previous experiences by applying principles-based approaches

Problem Statement

The goal of this activity is to analyze and modify the xv6 operating system's process scheduler to understand its direct impact on the performance of I/O-bound and CPU-bound applications. Students will act as OS performance engineers. They are tasked with implementing a new scheduling policy in xv6 and then developing custom user-level applications to benchmark and analyze how this change in OS design affects application performance. This requires a deep understanding of the relationship between kernel-level scheduling decisions and user-level application behavior.

Tasks:

1. OS Initialization and Setup

- **Environment Setup:** Configure the xv6 development environment using the QEMU emulator and the RISC-V toolchain.
- **Code Familiarization:** Study the existing xv6 source code for the scheduler (kernel/proc.c), system call interface, and process management.
- **Baseline Compilation:** Compile and run the unmodified xv6 to ensure the environment is working correctly and establish a performance baseline.

2. Kernel Modification: Scheduler Implementation

- **Implement a Multi-Level Feedback Queue (MLFQ) Scheduler:** Replace xv6's default round-robin scheduler with an MLFQ scheduler. This scheduler should have at least three priority levels.
    - Processes start at the highest priority.
    - If a process uses its entire time slice, it is demoted to a lower priority.
    - If a process yields the CPU (e.g., for I/O), it remains at its current priority level.

- **System Call for Performance Analysis:** Add a new system call, getprocinfo(int pid), that returns performance metrics for a specific process, such as total CPU ticks consumed and the number of times it has been scheduled.

3. Application Development for Performance Analysis

- **Develop a CPU-Bound Application:** Write a user-level program that performs an intensive computational task (e.g., calculating a large number of prime numbers) to simulate a CPU-bound workload.
- **Develop an I/O-Bound Application:** Write a user-level program that frequently performs I/O operations (e.g., repeatedly creating and writing small amounts of data to a file) to simulate an I/O-bound workload.

4. Testing and Performance Analysis

- **Functional Testing:** Verify that the new MLFQ scheduler correctly manages processes and that the getprocinfo system call returns accurate data.
- **Benchmark Execution:** Run both the CPU-bound and I/O-bound applications simultaneously on both the original round-robin scheduler and your new MLFQ scheduler.
- **Performance Report:** Write a technical report analyzing your findings. The report must:
    - Compare the performance (e.g., completion time, CPU ticks) of both applications under each scheduler.
    - Explain *why* the MLFQ scheduler impacts the performance of CPU-bound and I/O-bound tasks differently.
    - Conclude with a clear analysis of how this specific OS design choice (the scheduler) directly influences application design and performance.

Guidelines and Evaluation Criteria Overview

1. All kernel modifications must be performed within the xv6 RISC-V source code.
2. Students must use a controlled QEMU virtual environment for all testing to ensure consistent results.
3. The analysis must be supported by quantitative data (benchmarks) collected from your test applications.

Submission Details

1. Submit complete working project as zip file containing all the files, code, data etc.

NED University of Engineering & Technology
Department of Software Engineering
CEA RUBRICS
(SE-303) Operating Systems

Criteria	Good (2)	Satisfactory (1)	Not Achieved (0)	CPA	Marks
1. Scheduler Logic & Correctness	The MLFQ scheduler is implemented correctly and efficiently, adhering to all specified logic (priority levels, promotion/demotion rules).	The scheduler is functional but contains minor logical flaws or inefficiencies. It may not fully handle all edge cases correctly.	The scheduler fails to compile, is non-functional, or does not implement the core requirements of an MLFQ.	CPA1 CPA2 CPA3	
2. Kernel Integration & Stability	The new scheduler and system call are cleanly integrated into the kernel without causing instability, memory leaks, or race conditions.	The new code is functional but may cause minor, intermittent issues or does not follow xv6's existing design patterns.	The implementation causes frequent kernel panics, data corruption, or fails to integrate with the existing kernel code.	CPA1 CPA2 CPA3	
3. Test Application Design	Both the CPU-bound and I/O-bound applications are well-designed, effectively simulating their respective workloads to clearly test the scheduler.	The applications are functional but are not distinct enough in their behavior, making the performance difference less clear.	The applications fail to run, do not correctly simulate the intended workloads, or were not developed.	CPA1 CPA2 CPA3	
4. Functional Demonstration	The functionality of the new scheduler is clearly and correctly demonstrated, showing how it prioritizes processes differently than the original scheduler.	A demonstration is provided, but it is unclear or fails to convincingly show the impact of the new scheduling algorithm.	No functional demonstration is provided, or the demonstration shows the implementation does not work.	CPA1 CPA3	
5. Code Quality & Readability	The code written (both kernel and user-level) is clean, well-commented, easy to understand, and follows standard C programming practices.	The code is functional but is poorly formatted, lacks sufficient comments, or is difficult to follow.	The code is unreadable, uncommented, and does not follow basic programming standards.	CPA3	
Weighted CLO Score					
Remarks and Signature					

LAB SESSION 14

Solving the Dining Philosophers Problem

THEORY

The Dining Philosophers Problem

The problem describes five philosophers sitting around a circular table. Between each pair of philosophers is a single fork. A philosopher's life consists of alternating between two states: **thinking** and **eating**.

To eat, a philosopher needs to pick up **both** the fork to their left and the fork to their right. The challenge is to design a protocol that allows them to do this without creating a **deadlock** (a situation where everyone is stuck waiting for a resource held by someone else) or **starvation** (where a philosopher is perpetually denied a chance to eat).

A naive solution, where each philosopher picks up their left fork and then waits for their right fork, leads to a classic deadlock scenario: if all five pick up their left fork simultaneously, no one can pick up their right fork, and they all wait forever.

Designing a Deadlock-Free Solution

The solution we will implement is a monitor-like approach that ensures a philosopher can only pick up their forks if **neither of their neighbors is currently eating**.

We will manage this using three key components:

1. **A states array:** This shared array tracks the current state of each philosopher (THINKING, HUNGRY, or EATING).
2. **A Lock:** A single mutex lock is used to ensure that only one philosopher can change the states array or their own state at any given moment. This prevents race conditions.
3. **A Condition variable:** This powerful primitive is tied to the lock. It allows a philosopher who cannot eat (because a neighbor is eating) to wait() patiently. When a neighbor finishes eating, they can notify() the waiting philosopher that it's now possible to check for forks again.

Python Implementation Framework

You are provided with the following Python script skeleton. Your task is to complete the take_forks and put_forks methods by implementing the described logic.

```
# dining_philosophers.py
import threading
import time
import random

# --- Configuration ---
PHILOSOPHER_COUNT = 5

# --- Shared State and Synchronization ---
# Possible states for a philosopher
```

```
THINKING = 'THINKING'
HUNGRY = 'HUNGRY'
EATING = 'EATING'

# Shared array to track the state of each philosopher
states = [THINKING] * PHILOSOPHER_COUNT

# A condition variable combines a lock with wait/notify capabilities.
# This single object will manage all synchronization.
condition = threading.Condition()

# --- Philosopher Thread Class ---
class Philosopher(threading.Thread):
    def __init__(self, index):
        super().__init__()
        self.index = index

    def run(self):
        while True:
            self.think()
            self.take_forks()
            self.eat()
            self.put_forks()

    def think(self):
        print(f"Philosopher {self.index} is thinking.")
        time.sleep(random.uniform(1, 3))

    def eat(self):
        print(f"Philosopher {self.index} is EATING.")
        time.sleep(random.uniform(1, 4))

    def _test(self, i):
        # A philosopher can only eat if they are hungry and their neighbors are not eating.
        left_neighbor = (i - 1) % PHILOSOPHER_COUNT
        right_neighbor = (i + 1) % PHILOSOPHER_COUNT

        if (states[i] == HUNGRY and
            states[left_neighbor] != EATING and
            states[right_neighbor] != EATING):

            states[i] = EATING
            # Notify the waiting philosopher that they can now proceed.
            condition.notify_all()
            return True
        return False

    def take_forks(self):
        print(f"Philosopher {self.index} is hungry.")
        # --- EXERCISE: Implement the logic for taking forks ---
        # 1. Acquire the condition lock.
        # 2. Set your state to HUNGRY.
        # 3. Try to acquire forks by calling _test().
        # 4. If you can't eat, wait() on the condition variable.
        # 5. Release the lock.
```

```
        pass # Remove this line after implementing

    def put_forks(self):
        print(f"Philosopher {self.index} is putting forks down.")
        # --- EXERCISE: Implement the logic for putting forks down ---
        # 1. Acquire the condition lock.
        # 2. Set your state back to THINKING.
        # 3. Notify your neighbors that you are done, so they can check if they can eat now.
        #    (Hint: call _test() for both the left and right neighbor).
        # 4. Release the lock.
        pass # Remove this line after implementing

# --- Main Application ---
if __name__ == "__main__":
    philosophers = [Philosopher(i) for i in range(PHILOSOPHER_COUNT)]

    for p in philosophers:
        p.start()

    for p in philosophers:
        p.join()
```

EXERCISE

1. Implement the Deadlock-Free Solution

   Complete the take_forks and put_forks methods in the dining_philosophers.py script. You must use the shared condition variable to acquire the lock and to wait() and notify() other threads.

   - take_forks Logic:
     1. Acquire the lock using with condition:.
     2. Set your own state to HUNGRY.
     3. Attempt to start eating by calling self._test(self.index).
     4. If _test returns False (meaning you couldn't start eating), you must wait() on the condition variable until another philosopher notifies you.
   - put_forks Logic:
     1. Acquire the lock using with condition:.
     2. Set your own state back to THINKING.
     3. Check if your neighbors can now eat. Call self._test(...) for your left neighbor and your right neighbor. This "wakes up" a neighbor if they were waiting.

   Deliverable:

   - The complete, working dining_philosophers.py script. Attach a printout of your code and a screenshot of the output showing philosophers successfully alternating between thinking and eating without deadlocking.

2. Analysis and Verification

   After completing the implementation, answer the following conceptual questions.

   1. Deadlock Prevention: How does this specific solution prevent the deadlock scenario where every philosopher holds one fork and waits for another?
   2. Starvation Prevention: How does the solution ensure that a philosopher will eventually get to eat and not be "starved" by their neighbors constantly eating?
   3. Role of the Condition Variable: What are the two distinct roles that the condition variable plays in this solution? (Hint: Think about locks and notifications).

LAB SESSION 15

Final Exam

OBJECTIVE

The objective of the Final Lab is to evaluate students on the overall knowledge and skills gained throughout all lab sessions of the course. This session consolidates learning by requiring students to demonstrate mastery of Operating Systems through integrated tasks that reflect real-world problem-solving.

EVALUATION METHOD

Students will be assessed using the Final Lab Rubrics (5 marks), consisting of five criteria, each marked on the basis of 0 or 1.

NED University of Engineering & Technology
Department of Software Engineering
FINAL EXAMS RUBRICS
(SE-303) Operating Systems

Criteria	Achieved (1)	Not Achieved (0)	Marks
Problem Identification	Able to correctly identify the problem(s) and all required data (input and output)	Not able to correctly identify the problem(s) and all required data (input and output).	
Problem Solving Skills	Uses an effective strategy to solve the problem(s) and clearly identifies the steps	Uses an ineffective strategy to solve the problem(s) and completely misses the steps	
Implementation & Debugging	Efficient implementation and excellent debugging skills. Solution works without any error	Incomplete implementation and ineffective debugging skills. Major errors in the implementation.	
Deliverable(s)	Generates correct and complete output.	Cannot generate the desired output.	
Depth of Knowledge	Thorough grasp of the concepts covered in the lab and can apply it in real-life situations	Does not grasp the concepts covered in the lab and cannot identify any real-life application	
Weighted CLO Score			
Remarks and Signature			