OS Lab

i) Pwd :- Printing working directory
- it display the current working directory in O.S.
- The full path name of current working directory will be displayed on terminal.

ii) cd :- Your changing directory

iii) cd \ - Change root directory

iv) cd ~ → Shortcut your home directory.

v) cd.. move to one directory back.

vi) cd - move to Home directory but this command not support in windows but is support in linux.

vii) ls - list file in current directory

viii) ls -l list file in long format

ix) ls -a list all files

x) dir /w used in to display the content of directory in wide format, which shown file, folder name in multiple column.

⑪ mkdir make a folder

⑫ (cd.foldername) / cd. directoryname :- open the new folder

⑬ touch file name Create a empty file

⑭ nano file name nano command is text editor that allow you to create or edit files directory in the terminal.

You are inside nano command then

Ctrl + O to save the file.

Ctrl + X to exit the editor

nano +<line_number> file name

This open the file at existing line number.

(15) Cat Command used to display, create or Concatenate file.

Cat file name :- display the content of that file.

Cat > filename_new Create new file.

cat file1 file2 > Combine.text Combine both file content.

tac filename view file in reverse order.

(16) head Command displays the first n line of file.

head filename display 10 first line

head -n 5 filename display specific no. of line

head file1 file2 display 10 first line of both file.

(17) tail Command display the last n line of file.

(18) less Command used to display view the content of a file and output in a paginated manner.

This is ideal for reading large file with the help of Scroll key and press q to quit.

less filename

ls -l | less display one page at time

(19) echo echo command is used to display text or variable in the terminal.

echo "Hello world"

My Name = "Lucky"

echo "My Name is $ My_Name"

echo "Content" > filename Write content in filename

echo "Content" >> filename add content in filename.

(20) cp command is used to copy file to directories

cp source_file destination_file.

cp file1.txt file2.txt = copy file1 to file2

cp filename | path | to destination
copy file to another directory

cp -r folder1 |path| to destination
copy a directory (-r is used for all folder & subdirectory are copy

cp -p file1 file2
preserve file permission | attribute.

(21) mv used to move or rename files and directories.

mv file name move to file.
mv oldfilename newfile name Rename file.
mv folder1 path destination move directory
mv -f filename path
overwrite without Confirmation.

(22) rm rm Command is used to delete file or directories
rm filename delete file
rm filename1 filename2 delete multiple file.
rm -r folder1 delete directory & its Content.
rm -rf folder1 force delete without Confirmation.

(23) file permission in Unix / Linux
i) owner :- person who create file directory
ii) Group :- person who share same permission to another permission.
iii) others :- All other ~~or~~ user who are not the owner or in the group.

(24) Types of permission :-
i) read (r)
ii) write (w)
iii) Execute (x)

(25) By using ls -l are display by using ls -l Command

format -rwx r-x r--
owner Group other

(26) Breaking Down to character

Position	Meaning	Example
1st	filetype (- for file, d directory)	- Regular file
2-4	owner (user permission)	rw- (Read & write)
5-7	Group permission	r-- (Read only)
8-10	other permission	r-- (Read only)

(27) Changing file permission :-

chmod (for change the file permission)

- We can change file permission using chmod command through the use of Symbollic or numeric code.

* Permission category :-

i) owner (u)
- person who create file or directory
- person can have read, write and execute

Ex chmod u+rwx file.txt

ii) Group (G)
- person share permission to another person.

Ex chmod g+rw file.txt

iii) other (o)
- All other users who are neither the owner nor in group.

Ex chmod o-r file.txt

iv) All user (A)
- Affect owner, group and other

Ex chmod a+x script.sh

(28) The two ways to set permission :-

i) Using Numeric mode (octal mode)
Each permission type has a numeric value.

read (r) = 4
write (w) = 2
execute (x) = 1

7 → rwx (4+2+1)
6 → rw- (4+2)
5 → r-x (4+1)
4 → r-- (read only)

Ex chmond 744 file.txt
chmod 644 file.txt

ii) Using Symbolic mode
u — owner
g — ~~user~~ Group
o — other
a — All (owner, group, other)

Ex chmod u+x file.txt
chmod o-w file.txt

(29) Shell Scripting :-
- Shell scripting is a file containging linux and unix commond that automate task
- Shell scripting help us to task by writting shell scripting.

- A Shell script is a series of command stored in file that can be executed in program.

Ex i) nano . Script.sh open terminal & create file
ii) echo "Hello Vpes" → add following lines
iii) Save & exit (in nano, press Ctrl + X, then Enter.

Granting permission :-
chmod + x script.sh :- Before running the script give it Execute permission.

Running the script :-
./script.sh

Q Write a shell script that create a directory moves into it Create a file write text into the file and displays its content.

ShellFile.sh

Users > amanchoudhary > Desktop > Operating Systems > Operating-System > Lab1 > Practice > ShellFile.sh

You, last month | 1 author (You)

```
#!/bin/bash
echo "Hello World"
echo "Enter your name:"
read name
echo "Hello , my name is $name"

```

file1.txt

```
Hello Aman Choudhary
MCA
```

Q1: write a script that prints even number from 2 to 20 using a for loop

```
#!/bin/bash
for i in {2..20}
do
    if [ $((i % 2)) -eq 0 ]; then
        echo "Even number: $i"
    fi
done
```

Q2: write a while loop that keep asking for a password until a user enters 12345.

```
#!/bin/bash
while true;
do
    read -p "Enter password: " password
    if [ "$password" == "12345" ]; then
        echo "Password correct!"
        break
    else
        echo "Incorrect password. Try again."
    fi
done
```

Q3.Write an unit loop that counts down from 10 to 1 and then prints go.

```
#!/bin/bash

count=10

until [ $count -lt 1 ]
do
    echo $count
    ((count--))
done
echo "Go"
```

Question :Write a " C/C++ PROGRAM" to exercise the following non-preemptive cpu scheduling to find average turn around time and average waiting time, the program should ask for user input
1.FCFS
2.SJF
3.SRTF
4.PRIORITY BASE
ASK FOR PROCESSES AND
SAMPLE INPUT: process(P0,P1,P2,P3,P4),arrival time(0,1,2,3,4), burst time(4,3,1,2),priority(2,3,4,51)

```c
#include <stdio.h>
#include <stdbool.h>
#include <limits.h>

// Structure to hold process details
struct Process {
    int id, arrival, burst, priority, waiting, turnaround, 
remaining;
};

// Function to calculate First Come First Serve (FCFS) scheduling
void calculateFCFS(struct Process p[], int n) {
    int completion[n];
    p[0].waiting = 0;
    completion[0] = p[0].arrival + p[0].burst;
    p[0].turnaround = p[0].burst;

    for (int i = 1; i < n; i++) {
        completion[i] = (completion[i - 1] > p[i].arrival ? 
completion[i - 1] : p[i].arrival) + p[i].burst;
        p[i].turnaround = completion[i] - p[i].arrival;
        p[i].waiting = p[i].turnaround - p[i].burst;
    }
}

// Function to calculate Shortest Job First (SJF) scheduling (Non-
Preemptive)
void calculateSJF(struct Process p[], int n) {
    bool done[n];
    int completed = 0, time = 0;
    for (int i = 0; i < n; i++) done[i] = false;
```

```
    while (completed < n) {
        int minIndex = -1;
        for (int i = 0; i < n; i++) {
            if (!done[i] && p[i].arrival <= time) {
                if (minIndex == -1 || p[i].burst <
p[minIndex].burst)
                    minIndex = i;
            }
        }
        if (minIndex == -1) {
            time++;
            continue;
        }
        time += p[minIndex].burst;
        p[minIndex].turnaround = time - p[minIndex].arrival;
        p[minIndex].waiting = p[minIndex].turnaround -
p[minIndex].burst;
        done[minIndex] = true;
        completed++;
    }
}

// Function to calculate Shortest Remaining Time First (SRTF)
scheduling (Preemptive SJF)
void calculateSRTF(struct Process p[], int n) {
    int completed = 0, time = 0, minIndex;
    for (int i = 0; i < n; i++) p[i].remaining = p[i].burst;

    while (completed < n) {
        minIndex = -1;
        int minRemaining = INT_MAX;
        for (int i = 0; i < n; i++) {
            if (p[i].arrival <= time && p[i].remaining > 0 &&
p[i].remaining < minRemaining) {
                minRemaining = p[i].remaining;
                minIndex = i;
            }
        }
        if (minIndex == -1) {
            time++;
            continue;
        }
        p[minIndex].remaining--;
        time++;
        if (p[minIndex].remaining == 0) {
            completed++;
            p[minIndex].turnaround = time - p[minIndex].arrival;
            p[minIndex].waiting = p[minIndex].turnaround -
p[minIndex].burst;
        }
    }
```

```
}

// Function to calculate Priority Scheduling (Non-Preemptive)
void calculatePriority(struct Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (p[i].priority > p[j].priority) {
                struct Process temp = p[i];
                p[i] = p[j];
                p[j] = temp;
            }
        }
    }
    calculateFCFS(p, n); // Treat as FCFS after sorting by
priority
}

// Function to print the results (Waiting Time and Turnaround
Time)
void printResults(struct Process p[], int n) {
    float avgWait = 0, avgTurnaround = 0;
    printf("\nProcess | Arrival | Burst | Priority | Waiting |
Turnaround\n");

printf("-----------------------------------------------------------
\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t  %d\t  %d\t  %d\t   %d\t   %d\n", p[i].id,
p[i].arrival, p[i].burst, p[i].priority, p[i].waiting,
p[i].turnaround);
        avgWait += p[i].waiting;
        avgTurnaround += p[i].turnaround;
    }
    printf("\nAverage Waiting Time: %.2f", avgWait / n);
    printf("\nAverage Turnaround Time: %.2f\n", avgTurnaround /
n);
}

int main() {
    int n, choice;
    printf("Enter number of processes: ");
    scanf("%d", &n);
    struct Process p[n];

    printf("Enter Arrival Time, Burst Time, and Priority for each
process:\n");
    for (int i = 0; i < n; i++) {
        p[i].id = i;
        printf("P%d: ", i);
        scanf("%d %d %d", &p[i].arrival, &p[i].burst,
&p[i].priority);
    }
```

```c
    printf("\nSelect Scheduling Algorithm:\n");
    printf("1. FCFS\n2. SJF\n3. SRTF\n4. Priority\n");
    printf("Enter choice: ");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            calculateFCFS(p, n);
            break;
        case 2:
            calculateSJF(p, n);
            break;
        case 3:
            calculateSRTF(p, n);
            break;
        case 4:
            calculatePriority(p, n);
            break;
        default:
            printf("Invalid choice!\n");
            return 1;
    }

    printResults(p, n);
    return 0;
}
```

```
amanchoudhary@Amans-MacBook-Air CPU Scheduling % gcc -o a a.c
amanchoudhary@Amans-MacBook-Air CPU Scheduling % ./a
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
P0: 0 5 2
P1: 1 3 1
P2: 2 8 3
P3: 3 6 2

Select Scheduling Algorithm:
1. FCFS
2. SJF
3. SRTF
4. Priority
Enter choice: 2

Process | Arrival | Burst | Priority | Waiting | Turnaround
---------------------------------------------------------
P0        0       5       2        0       5
P1        1       3       1        4       7
P2        2       8       3        12      20
P3        3       6       2        5       11

Average Waiting Time: 5.25
Average Turnaround Time: 10.75
```

```
amanchoudhary@Amans-MacBook-Air CPU Scheduling % gcc -o a a.c
amanchoudhary@Amans-MacBook-Air CPU Scheduling % ./a
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
P0: 0 5 2
P1: 1 3 1
P2: 2 8 3
P3: 3 6 2

Select Scheduling Algorithm:
1. FCFS
2. SJF
3. SRTF
4. Priority
Enter choice: 1

Process | Arrival | Burst | Priority | Waiting | Turnaround
---------------------------------------------------------
P0        0       5       2        0       5
P1        1       3       1        4       7
P2        2       8       3        6       14
P3        3       6       2        13      19

Average Waiting Time: 5.75
Average Turnaround Time: 11.25
```

```
amanchoudhary@Amans-MacBook-Air CPU Scheduling % gcc -o a a.c
amanchoudhary@Amans-MacBook-Air CPU Scheduling % ./a
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
P0: 0 5 2
P1: 1 3 1
P2: 2 8 3
P3: 3 6 2

Select Scheduling Algorithm:
1. FCFS
2. SJF
3. SRTF
4. Priority
Enter choice: 3

Process | Arrival | Burst | Priority | Waiting | Turnaround
---------------------------------------------------------
P0        0       5       2        3       8
P1        1       3       1        0       3
P2        2       8       3        12      20
P3        3       6       2        5       11

Average Waiting Time: 5.00
Average Turnaround Time: 10.50
```

```
amanchoudhary@Amans-MacBook-Air CPU Scheduling % gcc -o a a.c
amanchoudhary@Amans-MacBook-Air CPU Scheduling % ./a
Enter number of processes: 4
Enter Arrival Time, Burst Time, and Priority for each process:
P0: 0 5 2
P1: 1 3 1
P2: 2 8 3
P3: 3 6 2

Select Scheduling Algorithm:
1. FCFS
2. SJF
3. SRTF
4. Priority
Enter choice: 4

Process | Arrival | Burst | Priority | Waiting | Turnaround
-----------------------------------------------------------
P1        1       3       1        0       3
P0        0       5       2        4       9
P3        3       6       2        6       12
P2        2       8       3        13      21

Average Waiting Time: 5.75
Average Turnaround Time: 11.25
```

Problem Statement 1:

HRRN (Highest Response Ratio Next) Scheduling

- HRRN is a non-preemptive CPU scheduling algorithm.
- It selects the process with the highest response ratio (HRR).
- Response Ratio Formula:

$$\text{Response Ratio} = \text{Waiting Time} + \text{Burst Time} \,/\, \text{Burst Time}$$

Task to be Done:

- Write a C/C++ program to simulate HRRN and compute AWT & ATAT.
- Input: Number of processes, arrival time and burst time.
- Output: Execution order, AWT, ATAT.

Code:

```c
#include <stdio.h>

typedef struct {
    int id, arrival_time, burst_time, waiting_time,
turnaround_time, completed;
} Process;

void sort_by_arrival(Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].arrival_time > p[j + 1].arrival_time) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

void hrrn_scheduling(Process p[], int n) {
    int completed = 0, current_time = 0;
    float total_wt = 0, total_tat = 0;

    // Sort by arrival time
```

```
    sort_by_arrival(p, n);

    printf("\nExecution Order:\n");

    while (completed < n) {
        int selected = -1;
        float max_response_ratio = -1.0;

        for (int i = 0; i < n; i++) {
            if (p[i].completed == 0 && p[i].arrival_time <=
current_time) {
                int waiting_time = current_time -
p[i].arrival_time;
                float response_ratio = (waiting_time +
p[i].burst_time) / (float)p[i].burst_time;

                if (response_ratio > max_response_ratio) {
                    max_response_ratio = response_ratio;
                    selected = i;
                }
            }
        }

        if (selected == -1) {
            current_time++;
        } else {
            p[selected].waiting_time = current_time -
p[selected].arrival_time;
            p[selected].turnaround_time = p[selected].waiting_time
+ p[selected].burst_time;
            total_wt += p[selected].waiting_time;
            total_tat += p[selected].turnaround_time;

            current_time += p[selected].burst_time;
            p[selected].completed = 1;
            completed++;

            printf("P%d -> ", p[selected].id - 1);
        }
    }

    printf("END\n");

    // Display process details
    printf("\nProcess\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        printf("P%d\t%d\t%d\t%d\t%d\n", p[i].id - 1,
p[i].arrival_time, p[i].burst_time, p[i].waiting_time,
p[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
```

```
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
}

int main() {
    // Sample Data
    Process sample_data[] = {
        {1, 1, 3, 0, 0, 0},
        {2, 3, 6, 0, 0, 0},
        {3, 5, 8, 0, 0, 0},
        {4, 7, 4, 0, 0, 0},
        {5, 8, 5, 0, 0, 0}
    };
    int sample_n = 5;

    printf("=== Sample Data Execution ===\n");
    hrrn_scheduling(sample_data, sample_n);

    // User Input
    int n;
    printf("\n\nEnter the number of processes: ");
    scanf("%d", &n);

    Process user_p[n];

    printf("Enter Arrival Time and Burst Time for each process:
\n");
    for (int i = 0; i < n; i++) {
        user_p[i].id = i + 1;
        printf("Process %d: ", i);
        scanf("%d %d", &user_p[i].arrival_time,
&user_p[i].burst_time);
        user_p[i].completed = 0;
    }

    printf("\n=== User Input Execution ===\n");
    hrrn_scheduling(user_p, n);

    return 0;
}
```

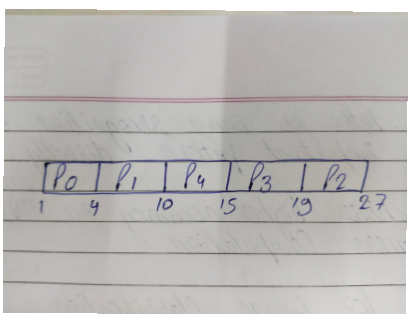
Sample Data Output:

```
amanchoudhary@Amans-MacBook-Air 21 March Lab % gcc hrrn.c -o hrrn
amanchoudhary@Amans-MacBook-Air 21 March Lab % ./hrrn
=== Sample Data Execution ===

Execution Order:
P0 -> P1 -> P3 -> P4 -> P2 -> END

Process AT      BT      WT      TAT
P0      1       3       0       3
P1      3       6       1       7
P2      5       8       14      22
P3      7       4       3       7
P4      8       5       6       11

Average Waiting Time: 4.80
Average Turnaround Time: 10.00
```

User Data Output:

```
Enter the number of processes: 4
Enter Arrival Time and Burst Time for each process:
Process 0: 2 5
Process 1: 4 3
Process 2: 6 9
Process 3: 7 2

=== User Input Execution ===

Execution Order:
P0 -> P1 -> P3 -> P2 -> END

Process AT  BT  WT  TAT
P0  2   5   0   5
P1  4   3   1   4
P2  6   9   7   16
P3  7   2   2   4

Average Waiting Time: 2.50
Average Turnaround Time: 7.25
```

Gantt chart:



Problem Statement 2 :

Multilevel Feedback Queue (MLFQ) Scheduling

- 3 Queues with Different Scheduling Algorithms:
  Queue 1 → Round Robin (Time Quantum = 4)
  Queue 2 → Round Robin (Time Quantum = 8)
  Queue 3 → First-Come, First-Serve (FCFS)

Task to be Done:

- Write a C/C++ program to simulate this MLFQ setup.
- Process moves down queues if not completed within a quantum.
- Input: Number of processes, arrival time, burst time.
- Output: Scheduling order, AWT, ATAT.

Code:

```
#include <stdio.h>

#define MAX 100

typedef struct {
    int id, arrival_time, burst_time, remaining_time,
completion_time;
    int waiting_time, turnaround_time;
} Process;

void round_robin(Process processes[], int n, int tq, int *time,
int queue_level) {
    printf("\nExecuting Queue %d (Time Quantum = %d):\n",
queue_level, tq);

    int all_completed;

    do {
        all_completed = 1;
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0 &&
processes[i].arrival_time <= *time) {
                all_completed = 0;
                if (processes[i].remaining_time > tq) {
                    printf("P%d -> (Remaining Time: %d)\n",
processes[i].id, processes[i].remaining_time - tq);
```

```
                    processes[i].remaining_time -= tq;
                    *time += tq;
                } else {
                    printf("P%d -> (Completed)\n",
processes[i].id);
                    *time += processes[i].remaining_time;
                    processes[i].completion_time = *time;
                    processes[i].turnaround_time =
processes[i].completion_time - processes[i].arrival_time;
                    processes[i].waiting_time =
processes[i].turnaround_time - processes[i].burst_time;
                    processes[i].remaining_time = 0;
                }
            }
        }
        if (!all_completed) (*time)++;
    } while (!all_completed);

    printf("END\n");
}

void fcfs(Process processes[], int n, int *time) {
    printf("\nExecuting Queue 3 (FCFS):\n");
    for (int i = 0; i < n; i++) {
        if (processes[i].remaining_time > 0) {
            printf("P%d -> (Completed)\n", processes[i].id);
            *time += processes[i].remaining_time;
            processes[i].completion_time = *time;
            processes[i].turnaround_time =
processes[i].completion_time - processes[i].arrival_time;
            processes[i].waiting_time =
processes[i].turnaround_time - processes[i].burst_time;
            processes[i].remaining_time = 0;
        }
    }
    printf("END\n");
}

void execute_mlfq(Process processes[], int n) {
    int time = 0;

    // Execute Round Robin Queue 1
    round_robin(processes, n, 4, &time, 1);

    // Move unfinished processes to Queue 2 (Round Robin with TQ =
8)
    round_robin(processes, n, 8, &time, 2);

    // Move remaining unfinished processes to Queue 3 (FCFS)
    fcfs(processes, n, &time);

    // Calculate and display results
```

```
    float total_wt = 0, total_tat = 0;
    printf("\nProcess\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        total_wt += processes[i].waiting_time;
        total_tat += processes[i].turnaround_time;
        printf("P%d\t%d\t%d\t%d\t%d\n", processes[i].id,
processes[i].arrival_time, processes[i].burst_time,
processes[i].waiting_time, processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
}

int main() {
    // Sample Data
    Process sample_data[] = {
        {0, 1, 3, 3, 0, 0, 0},
        {1, 3, 6, 6, 0, 0, 0},
        {2, 5, 8, 8, 0, 0, 0},
        {3, 7, 4, 4, 0, 0, 0},
        {4, 8, 5, 5, 0, 0, 0}
    };
    int sample_n = 5;

    printf("=== Sample Data Execution ===\n");
    execute_mlfq(sample_data, sample_n);

    // User Input
    int n;
    printf("\n\nEnter the number of processes: ");
    scanf("%d", &n);

    Process user_p[n];

    printf("Enter Arrival Time and Burst Time for each process:
\n");
    for (int i = 0; i < n; i++) {
        user_p[i].id = i;
        printf("Process %d: ", i);
        scanf("%d %d", &user_p[i].arrival_time,
&user_p[i].burst_time);
        user_p[i].remaining_time = user_p[i].burst_time;
    }

    printf("\n=== User Input Execution ===\n");
    execute_mlfq(user_p, n);

    return 0;
}
```

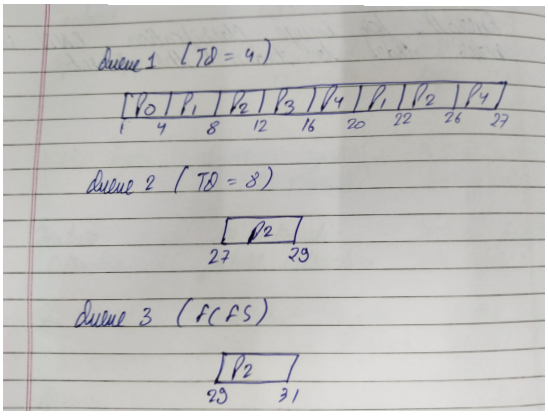
Sample Data Output:

```
amanchoudhary@Amans-MacBook-Air 21 March Lab % gcc mlfq.c -o mlfq
amanchoudhary@Amans-MacBook-Air 21 March Lab % ./mlfq
=== Sample Data Execution ===

Executing Queue 1 (Time Quantum = 4):
END

Executing Queue 2 (Time Quantum = 8):
END

Executing Queue 3 (FCFS):
P0 -> (Completed)
P1 -> (Completed)
P2 -> (Completed)
P3 -> (Completed)
P4 -> (Completed)
END

Process AT      BT      WT      TAT
P0      1       3       -1      2
P1      3       6       0       6
P2      5       8       4       12
P3      7       4       10      14
P4      8       5       13      18

Average Waiting Time: 5.20
Average Turnaround Time: 10.40
```

User Data Output:

```
Enter the number of processes: 3
Enter Arrival Time and Burst Time for each process:
Process 0: 2 5
Process 1: 4 3
Process 2: 6 9

=== User Input Execution ===

Executing Queue 1 (Time Quantum = 4):
END

Executing Queue 2 (Time Quantum = 8):
END

Executing Queue 3 (FCFS):
P0 -> (Completed)
P1 -> (Completed)
P2 -> (Completed)
END

Process AT      BT      WT      TAT
P0      2       5       -2      3
P1      4       3       1       4
P2      6       9       2       11

Average Waiting Time: 0.33
Average Turnaround Time: 6.00
```

Gantt chart:



Problem Statement 3 :

Multilevel Queue (MLQ) Scheduling

- Predefined Queue Allocation
  Queue 1 → P0, P1, P4
  Queue 2 → P2, P3
- Each queue follows different scheduling:
  Queue 1 → Higher Priority (Executes First)
  Queue 2 → Lower Priority (Executes After Queue 1 is Empty)

Task to be Done:

- Write a C/C++ program to simulate MLQ scheduling.
- Input: Number of processes, arrival time, burst time.
- Output: Scheduling order, AWT, ATAT.

Code:

```c
#include <stdio.h>

#define MAX 100

typedef struct {
    int id, arrival_time, burst_time, completion_time;
    int waiting_time, turnaround_time;
} Process;

// Function to sort processes by arrival time (FCFS)
void sort_by_arrival(Process p[], int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (p[j].arrival_time > p[j + 1].arrival_time) {
                Process temp = p[j];
                p[j] = p[j + 1];
                p[j + 1] = temp;
            }
        }
    }
}

// Function to execute processes in FCFS order
void execute_fcfs(Process queue[], int n, int *current_time) {
    for (int i = 0; i < n; i++) {
        if (*current_time < queue[i].arrival_time) {
            *current_time = queue[i].arrival_time;
        }
        queue[i].waiting_time = *current_time -
queue[i].arrival_time;
        queue[i].completion_time = *current_time +
queue[i].burst_time;
        queue[i].turnaround_time = queue[i].completion_time -
queue[i].arrival_time;
        *current_time += queue[i].burst_time;

        printf("P%d -> ", queue[i].id);
    }
}

// Function to update the main process array after execution
void update_processes(Process processes[], int n, Process queue[],
int queue_size) {
    for (int i = 0; i < queue_size; i++) {
        for (int j = 0; j < n; j++) {
            if (processes[j].id == queue[i].id) {
                processes[j] = queue[i]; // Update main process
array
                break;
```

```
            }
        }
    }
}

void execute_mlq(Process processes[], int n) {
    Process queue1[MAX], queue2[MAX];
    int q1_size = 0, q2_size = 0;
    int current_time = 0;

    // Allocate processes to predefined queues
    for (int i = 0; i < n; i++) {
        if (processes[i].id == 0 || processes[i].id == 1 ||
processes[i].id == 4) {
            queue1[q1_size++] = processes[i];
        } else {
            queue2[q2_size++] = processes[i];
        }
    }

    // Sort both queues by arrival time
    sort_by_arrival(queue1, q1_size);
    sort_by_arrival(queue2, q2_size);

    // Execute Queue 1 (Higher Priority)
    printf("\nExecuting Queue 1 (Higher Priority, FCFS):\n");
    execute_fcfs(queue1, q1_size, &current_time);
    printf("END\n");

    // Update main process array after Queue 1 execution
    update_processes(processes, n, queue1, q1_size);

    // Execute Queue 2 (Lower Priority)
    printf("\nExecuting Queue 2 (Lower Priority, FCFS):\n");
    execute_fcfs(queue2, q2_size, &current_time);
    printf("END\n");

    // Update main process array after Queue 2 execution
    update_processes(processes, n, queue2, q2_size);

    // Calculate AWT and ATAT
    float total_wt = 0, total_tat = 0;
    printf("\nProcess\tAT\tBT\tWT\tTAT\n");
    for (int i = 0; i < n; i++) {
        total_wt += processes[i].waiting_time;
        total_tat += processes[i].turnaround_time;
        printf("P%d\t%d\t%d\t%d\t%d\n", processes[i].id,
processes[i].arrival_time, processes[i].burst_time,
processes[i].waiting_time, processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f", total_wt / n);
```

```
    printf("\nAverage Turnaround Time: %.2f\n", total_tat / n);
}

int main() {
    // Sample Data
    Process sample_data[] = {
        {0, 1, 3, 0, 0, 0},
        {1, 3, 6, 0, 0, 0},
        {2, 5, 8, 0, 0, 0},
        {3, 7, 4, 0, 0, 0},
        {4, 8, 5, 0, 0, 0}
    };
    int sample_n = 5;

    printf("=== Sample Data Execution ===\n");
    execute_mlq(sample_data, sample_n);

    // User Input
    int n;
    printf("\n\nEnter the number of processes: ");
    scanf("%d", &n);

    Process user_p[n];

    printf("Enter Arrival Time and Burst Time for each process:
\n");
    for (int i = 0; i < n; i++) {
        user_p[i].id = i;
        printf("Process %d: ", i);
        scanf("%d %d", &user_p[i].arrival_time,
&user_p[i].burst_time);
        user_p[i].completion_time = 0;
        user_p[i].waiting_time = 0;
        user_p[i].turnaround_time = 0;
    }

    printf("\n=== User Input Execution ===\n");
    execute_mlq(user_p, n);

    return 0;
}
```

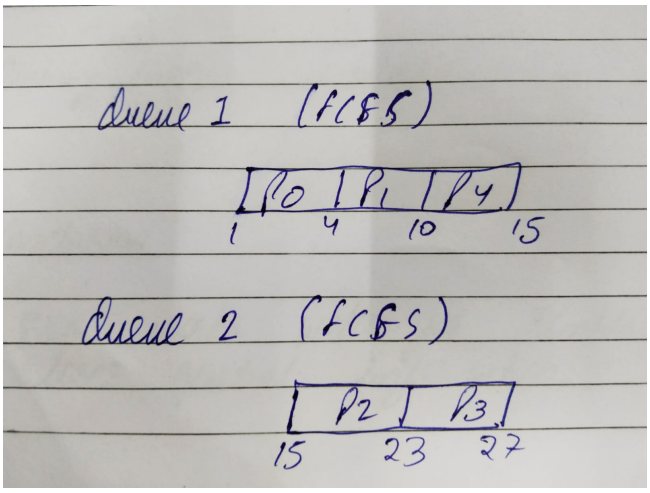
Sample Data Output:

```
amanchoudhary@Amans-MacBook-Air 21 March Lab % gcc mlq.c -o mlq
amanchoudhary@Amans-MacBook-Air 21 March Lab % ./mlq
=== Sample Data Execution ===

Executing Queue 1 (Higher Priority, FCFS):
P0 -> P1 -> P4 -> END

Executing Queue 2 (Lower Priority, FCFS):
P2 -> P3 -> END

Process AT      BT      WT      TAT
P0      1       3       0       3
P1      3       6       1       7
P2      5       8       10      18
P3      7       4       16      20
P4      8       5       2       7

Average Waiting Time: 5.80
Average Turnaround Time: 11.00
```

User Data Output:

```
Enter the number of processes: 3
Enter Arrival Time and Burst Time for each process:
Process 0: 2 5
Process 1: 4 3
Process 2: 6 9

=== User Input Execution ===

Executing Queue 1 (Higher Priority, FCFS):
P0 -> P1 -> END

Executing Queue 2 (Lower Priority, FCFS):
P2 -> END

Process AT      BT      WT      TAT
P0      2       5       0       5
P1      4       3       3       6
P2      6       9       4       13

Average Waiting Time: 2.33
Average Turnaround Time: 8.00
```

Gantt chart:



Problem Statement 1 :

Producer-Consumer Simulation (Bounded Buffer)

- Simulate a producer that creates items and a consumer that consumes them.
- Use three semaphores: `mutex`, `empty`, and `full`.
- Assume buffer size = 5.
- Show how mutual exclusion and buffer limits are handled.

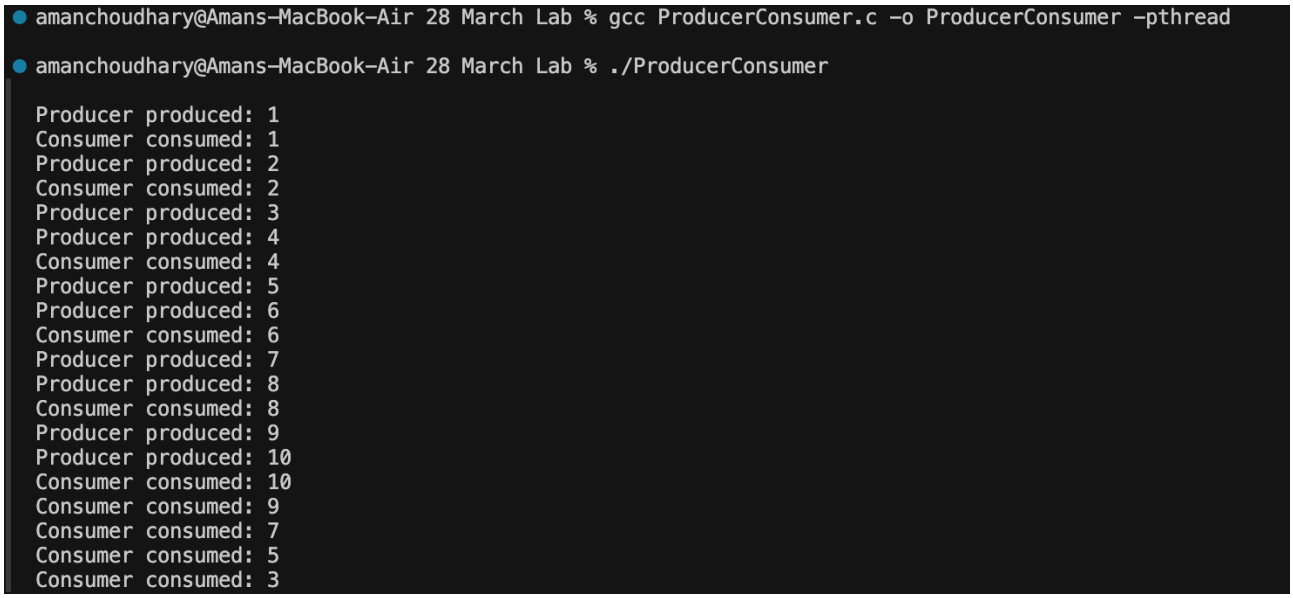
Hint: Use `wait(&empty)`, `wait(&mutex)`, `signal(&mutex)`, `signal(&full)` in proper sequence.

Code:

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

#define BUFFER_SIZE 5

int buffer[BUFFER_SIZE];
int count = 0;

sem_t *mutex, *empty, *full;

void *producer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        item = i + 1;

        sem_wait(empty);
        sem_wait(mutex);

        buffer[count++] = item;
        printf("🛠️ Producer ne item produce kiya: %d\n", item);

        sem_post(mutex);
        sem_post(full);

        sleep(1);
    }
    return NULL;
}
```

```
void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; i++) {
        sem_wait(full);
        sem_wait(mutex);

        item = buffer[--count];
        printf(" Consumer ne item consume kiya: %d\n", item);

        sem_post(mutex);
        sem_post(empty);

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t prod, cons;

    mutex = sem_open("/mutex", O_CREAT, 0644, 1);
    empty = sem_open("/empty", O_CREAT, 0644, BUFFER_SIZE);
    full = sem_open("/full", O_CREAT, 0644, 0);

    pthread_create(&prod, NULL, producer, NULL);
    pthread_create(&cons, NULL, consumer, NULL);

    pthread_join(prod, NULL);
    pthread_join(cons, NULL);

    sem_close(mutex);
    sem_unlink("/mutex");

    sem_close(empty);
    sem_unlink("/empty");

    sem_close(full);
    sem_unlink("/full");

    return 0;
}
```

Output:



Problem Statement 2 :

Reader-Writer Problem

- Implement a simulation where multiple readers can read simultaneously, but only one writer can write at a time.
- Show use of two semaphores: `mutex` and `wrt`, and a shared counter `readcount`.
- Simulate at least one reader and one writer.

Hint: Readers increment/decrement `readcount`, writer waits on `wrt`.

Code:

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

sem_t *mutex, *wrt;
int readcount = 0;

void *reader(void *arg) {
    int id = *(int *)arg;
    for (int i = 0; i < 3; i++) {
        sem_wait(mutex);
        readcount++;
        if (readcount == 1)
            sem_wait(wrt);
        sem_post(mutex);

        printf("Reader %d is reading\n", id);
        sleep(1);

        sem_wait(mutex);
        readcount--;
        if (readcount == 0)
            sem_post(wrt);
        sem_post(mutex);

        sleep(2);
    }
    return NULL;
}

void *writer(void *arg) {
    int id = *(int *)arg;
    for (int i = 0; i < 2; i++) {
        sem_wait(wrt);

        printf("Writer %d is writing\n", id);
        sleep(2);

        sem_post(wrt);

        sleep(2);
    }
    return NULL;
}

int main() {
    pthread_t r[2], w[1];
```

```
    int id[2] = {1, 2};
    int wid[1] = {1};

    mutex = sem_open("/mutex", O_CREAT, 0644, 1);

    for (int i = 0; i < 2; i++)
        pthread_create(&r[i], NULL, reader, &id[i]);

    for (int i = 0; i < 1; i++)
        pthread_create(&w[i], NULL, writer, &wid[i]);

    for (int i = 0; i < 2; i++)
        pthread_join(r[i], NULL);

    for (int i = 0; i < 1; i++)
        pthread_join(w[i], NULL);

    sem_close(wrt);
    sem_unlink("/wrt");

    sem_close(mutex);
    sem_unlink("/mutex");

    return 0;
}
```

Output:

```
amanchoudhary@Amans-MacBook-Air 28 March Lab % gcc ReaderWriter.c -o ReaderWriter -pthread

amanchoudhary@Amans-MacBook-Air 28 March Lab % ./ReaderWriter

Reader 1 is reading
Reader 2 is reading
Writer 1 is writing
Reader 1 is reading
Reader 2 is reading
Writer 1 is writing
Reader 2 is reading
Reader 1 is reading
```

Problem Statement 3 :

Dining Philosophers Simulation

- Five philosophers sit at a table and need two forks to eat.
- Simulate one round of each philosopher attempting to eat.
- Use array of forks to simulate resource allocation.
- Avoid deadlock using simple logic.

Hint: Each philosopher needs fork[i] and fork[(I+1)%5].

Code:

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

#define NUM_PHILOSOPHERS 5

sem_t *forks[NUM_PHILOSOPHERS];

void *philosopher(void *arg) {
    int id = *(int *)arg;
    int left_fork = id;
    int right_fork = (id + 1) % NUM_PHILOSOPHERS;

    if (id % 2 == 0) {
        sem_wait(forks[left_fork]);
        sem_wait(forks[right_fork]);
    } else {
        sem_wait(forks[right_fork]);
        sem_wait(forks[left_fork]);
    }

    printf("Philosopher %d is eating 🍽\n", id);
    sleep(1);

    sem_post(forks[left_fork]);
    sem_post(forks[right_fork]);

    printf("Philosopher %d has finished eating 🍴\n", id);

    return NULL;
}
```

```
int main() {
    pthread_t philosophers[NUM_PHILOSOPHERS];
    int id[NUM_PHILOSOPHERS];

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        char sem_name[10];
        sprintf(sem_name, "/fork%d", i);
        forks[i] = sem_open(sem_name, O_CREAT, 0644, 1);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        id[i] = i;
        pthread_create(&philosophers[i], NULL, philosopher,
&id[i]);
    }

    for (int i = 0; i < NUM_PHILOSOPHERS; i++)
        pthread_join(philosophers[i], NULL);

    for (int i = 0; i < NUM_PHILOSOPHERS; i++) {
        sem_close(forks[i]);
        char sem_name[10];
        sprintf(sem_name, "/fork%d", i);
        sem_unlink(sem_name);
    }

    return 0;
}
```

Output: