# Lab 4

Aim: Perform MLQ scheduling with different Algorithms and with different ATs.

Theory: MLQ (Multi-Level Queue) scheduling is a CPU scheduling algorithm used in operating systems. It is a type of scheduling algorithm where processes are divided into different priority levels, and each level is given a different scheduling algorithm.

The primary aim of MLQ scheduling is to provide optimal CPU utilization while minimizing the turnaround time and waiting time of processes. The different scheduling algorithms used in MLQ scheduling are:

1. FCFS (First-Come-First-Serve) Scheduling: This algorithm schedules processes in the order in which they arrive in the queue. The processes with the earliest arrival time get scheduled first.
2. SJF (Shortest-Job-First) Scheduling: This algorithm schedules processes with the shortest burst time first. It is based on the assumption that the process with the shortest burst time will complete first, leading to optimal CPU utilization.
3. Round Robin Scheduling: This algorithm schedules processes in a circular queue. Each process is allocated a fixed time slice, and the CPU switches between processes once the time slice expires.
4. Priority Scheduling: This algorithm schedules processes based on their priority levels. The process with the highest priority level gets scheduled first. If two or more processes have the same priority level, then the FCFS algorithm is used.

MLQ scheduling can be further classified into two types, namely:

1. Fixed Priority MLQ Scheduling: In this type of scheduling, the processes are assigned a priority level, and each priority level is assigned a fixed scheduling algorithm.
2. Dynamic Priority MLQ Scheduling: In this type of scheduling, the priority levels of processes are adjusted dynamically based on the nature of the process.

The Arrival Time (AT) is an important parameter in MLQ scheduling. It represents the time at which the process enters the scheduling queue. The different ATs used in MLQ scheduling are:

1. Zero Arrival Time: In this case, all processes enter the scheduling queue at the same time.
2. Random Arrival Time: In this case, the processes enter the scheduling queue at random times.
3. Predictable Arrival Time: In this case, the arrival time of processes is predictable.

The selection of the scheduling algorithm and the type of MLQ scheduling to be used depends on the nature of the processes being scheduled and the performance goals of the system. The AT also plays a crucial role in the scheduling process, and the selection of the AT depends on the nature of the workload.

In summary, MLQ scheduling is a powerful CPU scheduling algorithm that can be used to optimize CPU utilization and minimize turnaround time and waiting time of processes. The selection of the scheduling algorithm, the type of MLQ scheduling, and the AT to be used depend on the nature of the workload and the performance goals of the system.

**Code for MLQ Scheduling**:

# ScreenShot:-

**INPUT:-**

#include<stdio.h>

// Structure to represent a process

struct process{

    int pid; // Process ID

    int burst\_time; // Burst time of the process

    int priority; // Priority of the process

};

// Function to perform MLQ scheduling with fixed priority scheduling and time slices

void mlq\_scheduling(struct process system\_process[], int n\_system, struct process user\_process[], int n\_user, int time\_slice\_system, int time\_slice\_user){

    // Initialize the time variable to 0

    int time = 0;

    // Initialize the index variable for each queue to 0

    int i\_system = 0, i\_user = 0;

    // Initialize the total waiting time and turnaround time variables to 0

    int total\_waiting\_time = 0, total\_turnaround\_time = 0;

    // Initialize the current process and the time slice for each queue

    struct process \*current\_process = NULL;

    int time\_slice\_current = 0;

    // Loop until all processes have been scheduled

    while(i\_system < n\_system || i\_user < n\_user){

        // Check if there are any system processes waiting in the queue

        if(i\_system < n\_system){

            // If there is no current process or the current process is from the user process queue, select the next system process

            if(current\_process == NULL || current\_process->priority != 0){

                current\_process = &system\_process[i\_system];

                time\_slice\_current = time\_slice\_system;

                i\_system++;

            }

        }

        // Check if there are any user processes waiting in the queue

        if(i\_user < n\_user){

            // If there is no current process or the current process has used up its time slice, select the next user process

            if(current\_process == NULL || time\_slice\_current == 0){

                current\_process = &user\_process[i\_user];

                time\_slice\_current = time\_slice\_user;

                i\_user++;

            }

        }

        // Decrement the time slice for the current process

        time\_slice\_current--;

        // Decrement the burst time of the current process

        current\_process->burst\_time--;

        // If the current process has completed execution, calculate the waiting time and turnaround time

        if(current\_process->burst\_time == 0){

            int waiting\_time = time - current\_process->burst\_time - current\_process->priority;

            int turnaround\_time = waiting\_time + current\_process->burst\_time;

            total\_waiting\_time += waiting\_time;

            total\_turnaround\_time += turnaround\_time;

            current\_process = NULL;

        }

        // Increment the time variable

        time++;

    }

    // Print the average waiting time and turnaround time

    printf("Average waiting time = %f\n", (float)total\_waiting\_time/(n\_system+n\_user));

    printf("Average turnaround time = %f\n", (float)total\_turnaround\_time/(n\_system+n\_user));

}

int main(){

    // Initialize the system process queue and the user process queue

    struct process system\_process[] = {{1, 5, 0}, {2, 2, 0}, {3, 1, 0}};

    struct process user\_process[] = {{4, 3, 1}, {5, 2, 1}, {6, 4, 1}};

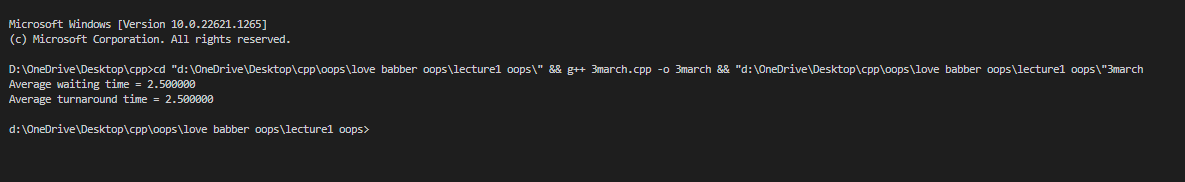
    // Perform MLQ scheduling with fixed priority scheduling and time slices

    mlq\_scheduling(system\_process, 3, user\_process, 3, 2, 3);

    return 0;

}

**OUTPUT:**

****

Aim: Develop MLQ, where a Time slice is given to each queue i.e., each queue gets a

certain amount of CPU time(assume any) that it can schedule amongst its processes.

Theory:

MLQ, or Multi-Level Feedback Queue, is a scheduling algorithm used in operating systems to allocate CPU time to processes. In MLQ, processes are divided into multiple queues based on their priority level. Each queue is given a certain amount of CPU time, known as a time slice or time quantum, that it can schedule amongst its processes.

The queues are organized in a hierarchical fashion, where the highest priority queue is given the shortest time slice and the lowest priority queue is given the longest time slice. Processes in the highest priority queue are scheduled first, and if they do not finish their execution within their time slice, they are moved down to the next lower priority queue.

Processes that wait in a lower priority queue for a long time are eventually promoted to a higher priority queue to prevent starvation. This promotes fairness in resource allocation as processes are given a chance to execute based on their priority level.

The advantage of MLQ is that it provides a balance between responsiveness and throughput. High-priority processes are executed quickly, while low-priority processes are not completely starved of CPU time. However, it can be difficult to determine the appropriate number of queues and time slices for each queue, as this can depend on the workload and system characteristics. Additionally, frequent queue promotions and demotions can add overhead to the scheduling algorithm.

**Code**:

# ScreenShot:-

**INPUT:-**

#include <stdio.h>

#define MAX\_PROCESSES 100

// Process Structure

struct Process {

    int pid;

    int arrival\_time;

    int burst\_time;

    int priority;

    int remaining\_time;

};

// Queue Structure

struct Queue {

    struct Process processes[MAX\_PROCESSES];

    int front;

    int rear;

    int size;

    int time\_slice;

};

// Function to initialize queue

void initialize(struct Queue \*q, int time\_slice) {

    q->front = 0;

    q->rear = -1;

    q->size = 0;

    q->time\_slice = time\_slice;

}

// Function to check if queue is empty

int is\_empty(struct Queue \*q) {

    return (q->size == 0);

}

// Function to check if queue is full

int is\_full(struct Queue \*q) {

    return (q->size == MAX\_PROCESSES);

}

// Function to add process to queue

void enqueue(struct Queue \*q, struct Process process) {

    if (!is\_full(q)) {

        q->rear = (q->rear + 1) % MAX\_PROCESSES;

        q->processes[q->rear] = process;

        q->size++;

    }

}

// Function to remove process from queue

struct Process dequeue(struct Queue \*q) {

    struct Process process;

    if (!is\_empty(q)) {

        process = q->processes[q->front];

        q->front = (q->front + 1) % MAX\_PROCESSES;

        q->size--;

        return process;

    }

}

// Function to get highest priority process from queue

int get\_highest\_priority(struct Queue \*q) {

    int highest\_priority = q->front;

    for (int i = q->front; i <= q->rear; i++) {

        if (q->processes[i].priority < q->processes[highest\_priority].priority) {

            highest\_priority = i;

        }

    }

    return highest\_priority;

}

// Function to perform SJF scheduling on System Process Ready Queue

void sjf(struct Queue \*system\_process\_queue) {

    // Sort queue based on burst time

    for (int i = system\_process\_queue->front; i <= system\_process\_queue->rear; i++) {

        for (int j = i + 1; j <= system\_process\_queue->rear; j++) {

            if (system\_process\_queue->processes[j].burst\_time < system\_process\_queue->processes[i].burst\_time) {

                struct Process temp = system\_process\_queue->processes[i];

                system\_process\_queue->processes[i] = system\_process\_queue->processes[j];

                system\_process\_queue->processes[j] = temp;

            }

        }

    }

}

// Function to perform FCFS scheduling on User Process Ready Queue

void fcfs(struct Queue \*user\_process\_queue) {

    // No sorting required as FCFS is a non-preemptive algorithm

}

// Function to perform Fixed Priority scheduling on Multi-Level Queue

void fixed\_priority(struct Queue \*system\_process\_queue, struct Queue \*user\_process\_queue) {

    // Serve all processes from higher priority queue first

    if (!is\_empty(system\_process\_queue)) {

        sjf(system\_process\_queue);

        struct Process system\_process = dequeue(system\_process\_queue);

        printf("System Process %d with Burst Time %d and Priority %d is running\n", system\_process.pid, system\_process.burst\_time, system\_process.priority);

        system\_process.remaining\_time -= system\_process\_queue->time\_slice;

      if(  system\_process.remaining\_time > 0) {

enqueue(system\_process\_queue, system\_process);

}

} else if (!is\_empty(user\_process\_queue)) {

fcfs(user\_process\_queue);

struct Process user\_process = dequeue(user\_process\_queue);

printf("User Process %d with Burst Time %d and Priority %d is running\n", user\_process.pid, user\_process.burst\_time, user\_process.priority);

user\_process.remaining\_time -= user\_process\_queue->time\_slice;

if (user\_process.remaining\_time > 0) {

enqueue(user\_process\_queue, user\_process);

}

}

}

int main() {

// Initialize Queues

struct Queue system\_process\_queue, user\_process\_queue;

initialize(&system\_process\_queue, 4); // time slice = 4 for System Process Queue

initialize(&user\_process\_queue, 8); // time slice = 8 for User Process Queue

// Add processes to queues

struct Process processes[] = {

    {1, 0, 10, 1, 10},

    {2, 0, 5, 2, 5},

    {3, 0, 3, 1, 3},

    {4, 0, 2, 2, 2},

    {5, 0, 4, 2, 4},

    {6, 0, 8, 1, 8},

    {7, 0, 6, 2, 6},

    {8, 0, 1, 1, 1}

};

for (int i = 0; i < 8; i++) {

    if (processes[i].priority == 1) {

        enqueue(&system\_process\_queue, processes[i]);

    } else {

        enqueue(&user\_process\_queue, processes[i]);

    }

}

// Perform scheduling

while (!is\_empty(&system\_process\_queue) || !is\_empty(&user\_process\_queue)) {

    fixed\_priority(&system\_process\_queue, &user\_process\_queue);

}

return 0;

}

**OUTPUT:**

