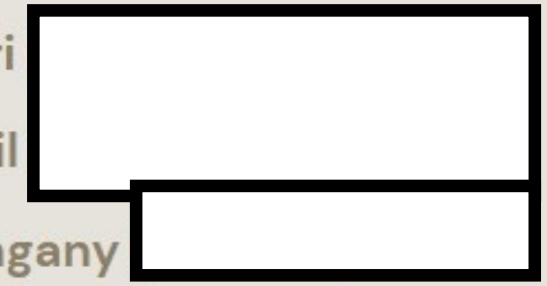
Multilevel-Feedback-Queue Scheduling

Khalid Nimri

Aseel Suhail

Mohammed allagany



Breakup of tasks

Student name	Student ID	Tasks completed
Khalid Nimri		Code implementation
Aseel Suhail		Introduction, Code implementation
Mohammed allagany		User manual, featres and capabilities, Conclusion

Breakup of tasks

Introduction to Multilevel-feedback-queue scheduler

Benefits of Multilevel-feedback-queue scheduler

Features and Capabilities of Multilevel Feedback Queue Scheduler

Code Implementation

User manual

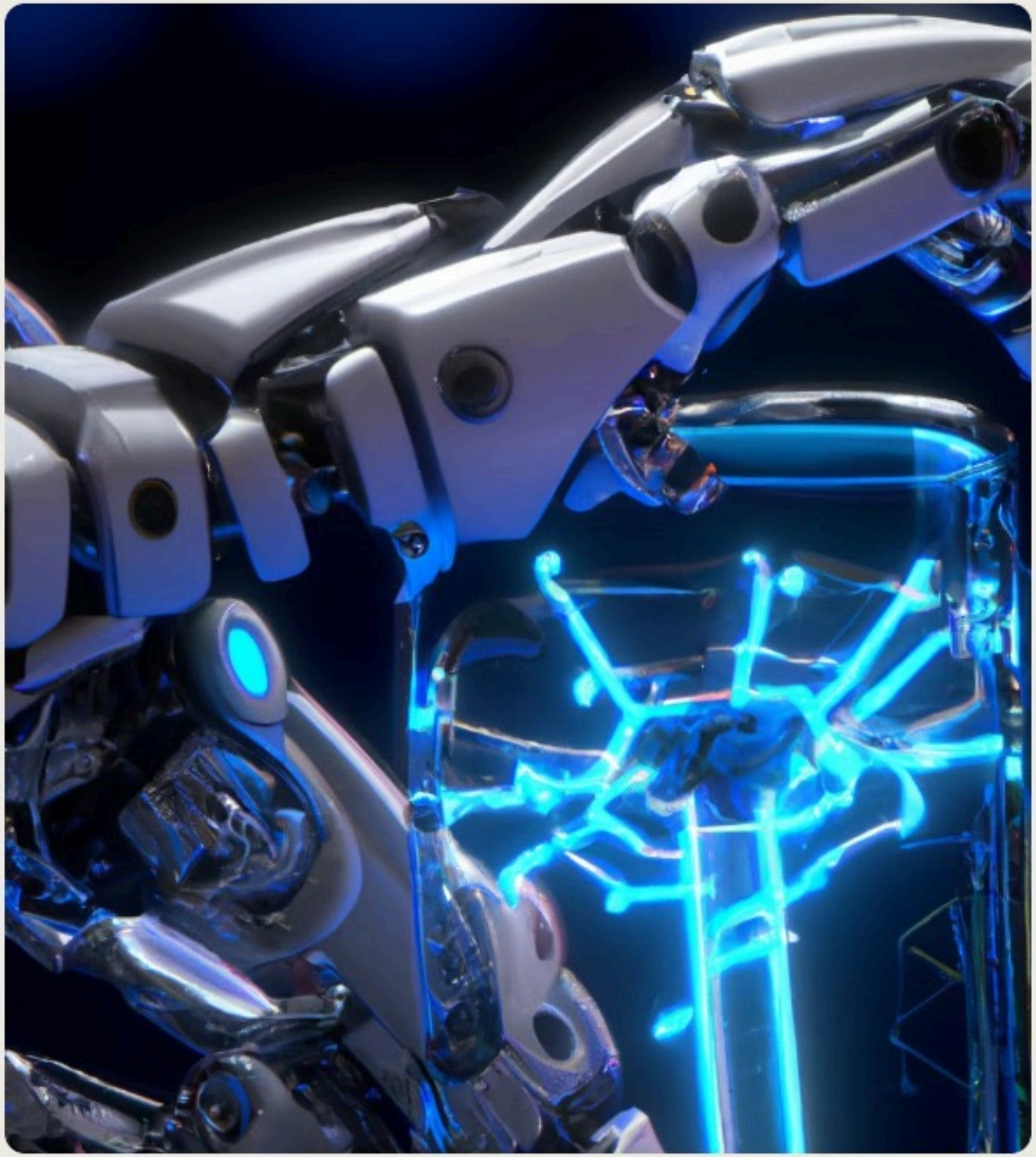
Conclusion

References

Introduction to MFQS

Multilevel-feedback-queue scheduler is a scheduling algorithm that divides processes into different priority levels. It allows processes to move between different queues and provides different scheduling algorithms for each queue. The scheduler dynamically adjusts the priority of processes based on their recent CPU usage and other factors.

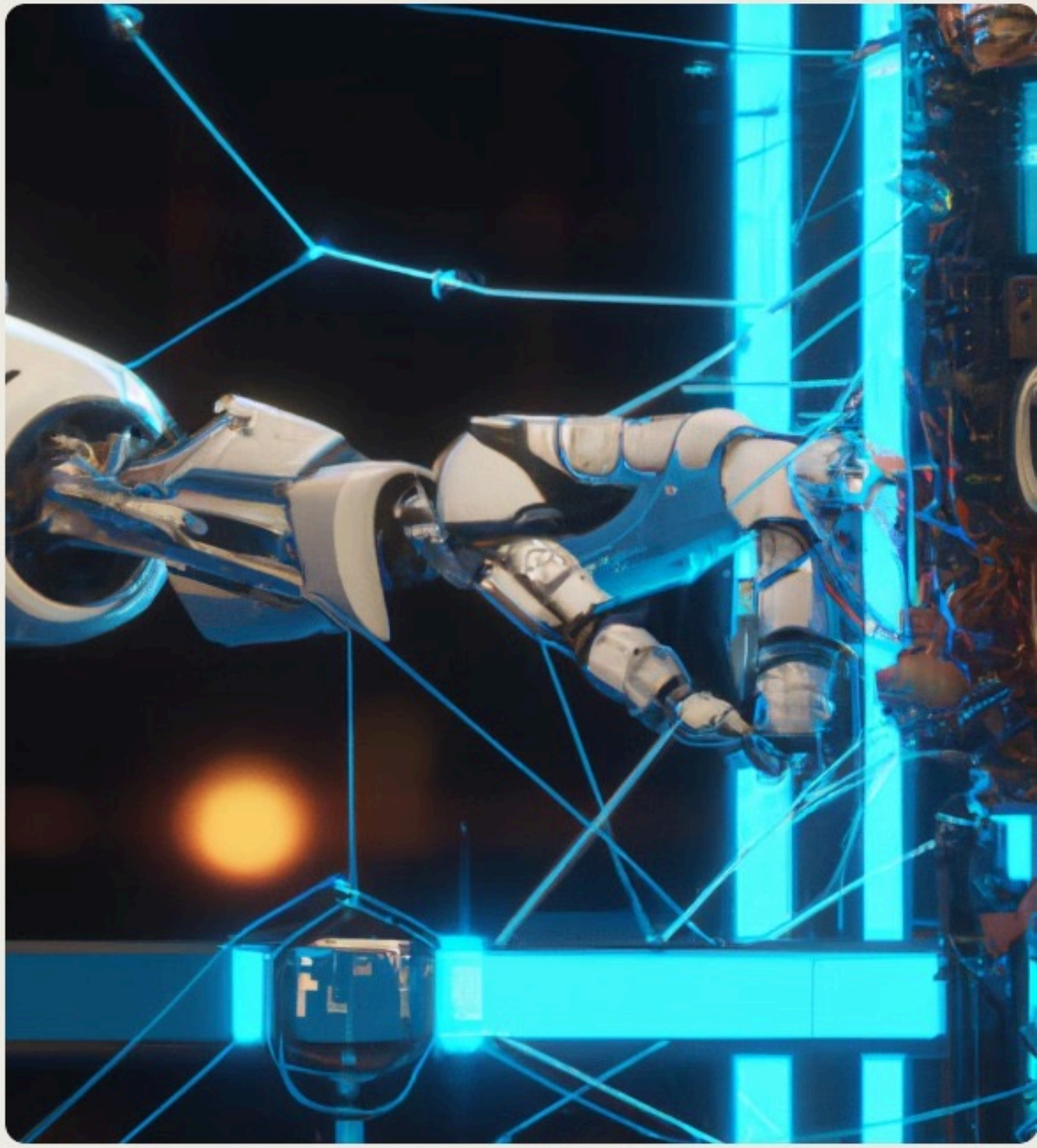
The scheduler also allows processes to move between queues, depending on their recent CPU usage. This allows processes to be executed in an efficient manner, as processes with higher priority are given more CPU time.



Benefits of Multilevel-feedback-queue scheduler

Multilevel-feedback-queue scheduler provides more efficient CPU utilization as processes are given more CPU time based on their priority. It also allows processes to move between queues, depending on their recent CPU usage, which helps to ensure that processes are executed in an efficient manner.

The scheduler also allows for dynamic adjustment of the priority of processes, ensuring that processes with higher priority are given more CPU time. This helps to ensure that processes with higher priority are executed in a timely manner.



Features and Capabilities of Multilevel Feedback Queue Scheduler

A Multilevel Feedback Queue Scheduler offers the ability to assign processes different priority levels, allowing for more efficient allocation of resources.

It also provides more precise control of the scheduling process, allowing for processes to be moved between queues quickly and easily.

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

#define Q0_TIME_QUANTUM 8
#define Q1_TIME_QUANTUM 16

struct PCB {
    int process_id;
    int arrival_time;
    int burst_time;
    int remaining_time;
    int waiting_time;
    int response_time;
    int completion_time;
};

// Function to compare two processes based on arriva
int compare_arrival_time(const void* a, const void* 
    struct PCB *p1 = (struct PCB*)a;
    struct PCB *p2 = (struct PCB*)b;
    return (p1->arrival_time - p2->arrival_time);
}

int main() {
    int num_processes, i;
    printf( format: "Enter the number of processes: ");
    scanf( format: "%d", &num_processes);

    struct PCB processes[num_processes];

    printf( format: "Enter the arrival time and burst t
    for (i = 0; i < num_processes; i++) {
        printf( format: "Process %d: ", i+1);
        scanf( format: "%d %d", &processes[i].arrival_t
        processes[i].process_id = i+1;
        processes[i].remaining_time = processes[i].bu
    }
```

```c
    // Sort the processes based on arrival time
    qsort( Base: processes, NumOfElements: num_processes, SizeOfElements: sizeof(struct PCB), PtFuncCompare: compare_arriva

    int current_time = 0, completed_processes = 0;
    int q0_processes = 0, q1_processes = 0, q2_processes = 0;

    // Initialize the queues
    struct PCB q0[num_processes], q1[num_processes], q2[num_processes];

    while (completed_processes < num_processes) {
        // Move new processes to Q0
        while (processes[q0_processes].arrival_time <= current_time && q0_processes < num_processes) {
            q0[q0_processes] = processes[q0_processes];
            q0_processes++;
        }

        // Check if Q0 is empty
        if (q0_processes == 0) {
            current_time = processes[q0_processes].arrival_time;
            continue;
        }

        // Execute process in Q0
        struct PCB current_process = q0[0];
        int time_quantum;
        if (current_process.remaining_time > Q0_TIME_QUANTUM) {
            time_quantum = Q0_TIME_QUANTUM;
        } else {
            time_quantum = current_process.remaining_time;
        }
        current_time += time_quantum;
        current_process.remaining_time -= time_quantum;

        // Check if process is completed
        if (current_process.remaining_time == 0) {
            current_process.completion_time = current_time;
            current_process.waiting_time = current_process.completion_time - current_process.burst_time - current_
```

```
            current_process.waiting_time = current_process.completion_time - current_process.bu
            current_process.response_time = current_process.waiting_time;
            completed_processes++;
        } else {
// Move process to next queue
            if (current_process.remaining_time > Q1_TIME_QUANTUM) {
                q1[q1_processes++] = current_process;
            } else {
                q2[q2_processes++] = current_process;
            }
        }
        // Shift the remaining processes in Q0
        for (i = 1; i < q0_processes; i++) {
            q0[i-1] = q0[i];
        }
        q0_processes--;

        // Move completed process from Q1 to Q2
        for (i = 0; i < q1_processes; i++) {
            if (q1[i].remaining_time == 0) {
                q2[q2_processes++] = q1[i];
                for (int j = i+1; j < q1_processes; j++) {
                    q1[j-1] = q1[j];
                }
                q1_processes--;
                i--;
            }
        }

        // Execute process in Q1
        if (q1_processes > 0) {
            current_process = q1[0];
            time_quantum;
            if (current_process.remaining_time > Q1_TIME_QUANTUM) {
                time_quantum = Q1_TIME_QUANTUM;
            } else {
                time_quantum = current_process.remaining_time;
            }
            current_time += time_quantum;
            current_process.remaining_time -= time_quantum;

            // Check if process is completed
            if (current_process.remaining_time == 0) {
                current_process.completion_time = current_time;
                current_process.waiting_time = current_process.completion_time - current_proces
                completed_processes++;
            } else {
                // Move process to next queue
                q2[q2_processes++] = current_process;
            }

            // Shift the remaining processes in Q1
            for (i = 1; i < q1_processes; i++) {
                q1[i-1] = q1[i];
            }
            q1_processes--;
        }

        // Execute process in Q2
        if (q2_processes > 0) {
            current_process = q2[0];
            current_time += current_process.remaining_time;
            current_process.remaining_time = 0;
            current_process.completion_time = current_time;
            current_process.waiting_time = current_process.completion_time - current_process.bu
            current_process.arrival_time;
            completed_processes++;
            // Shift the remaining processes in Q2
            for (i = 1; i < q2_processes; i++) {
                q2[i-1] = q2[i];
            }
            q2_processes--;
        }
    }

// Calculate average waiting time and throughput
```

```c
    int total_waiting_time = 0;
    for (i = 0; i < num_processes; i++) {
        total_waiting_time += processes[i].waiting_time;
    }
    float average_waiting_time = (float)total_waiting_time / num_processes;
    float throughput = (float)num_processes / current_time;

// Display results
    printf( format: "Response Time: %d\n", processes[0].response_time);
    printf( format: "Throughput: %f\n", throughput);
    printf( format: "Average Waiting Time: %f\n", average_waiting_time);

    return 0;
}
```

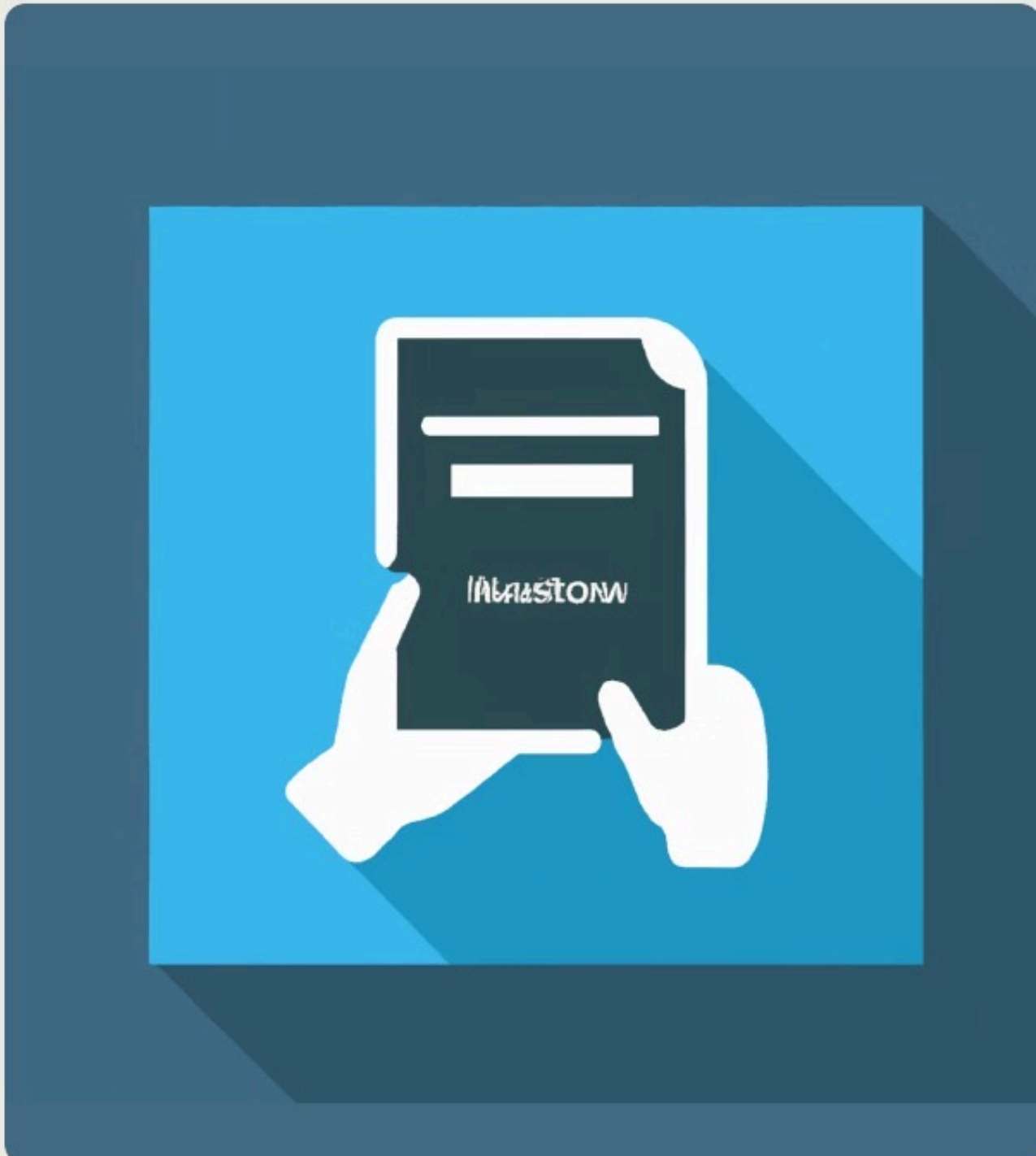
```
Enter the number of processes:3
 Enter the arrival time and burst time for each process:
Process 1:0 50
 Process 2:16 30
 Process 3:20 20
 Response Time: 124
Throughput: 0.023077
Average Waiting Time: 689132352.000000

Process finished with exit code 0
```

A Simple User Manual for Multilevel Feedback Queue Scheduler

This user manual provides a step-by-step guide on how to use the multilevel feedback queue scheduler, allowing the user to create efficient resource utilization and prioritize processes for optimal performance.

This guide explains the features of the scheduler, how to set up and configure it, and how to use it to manage processes and resources.



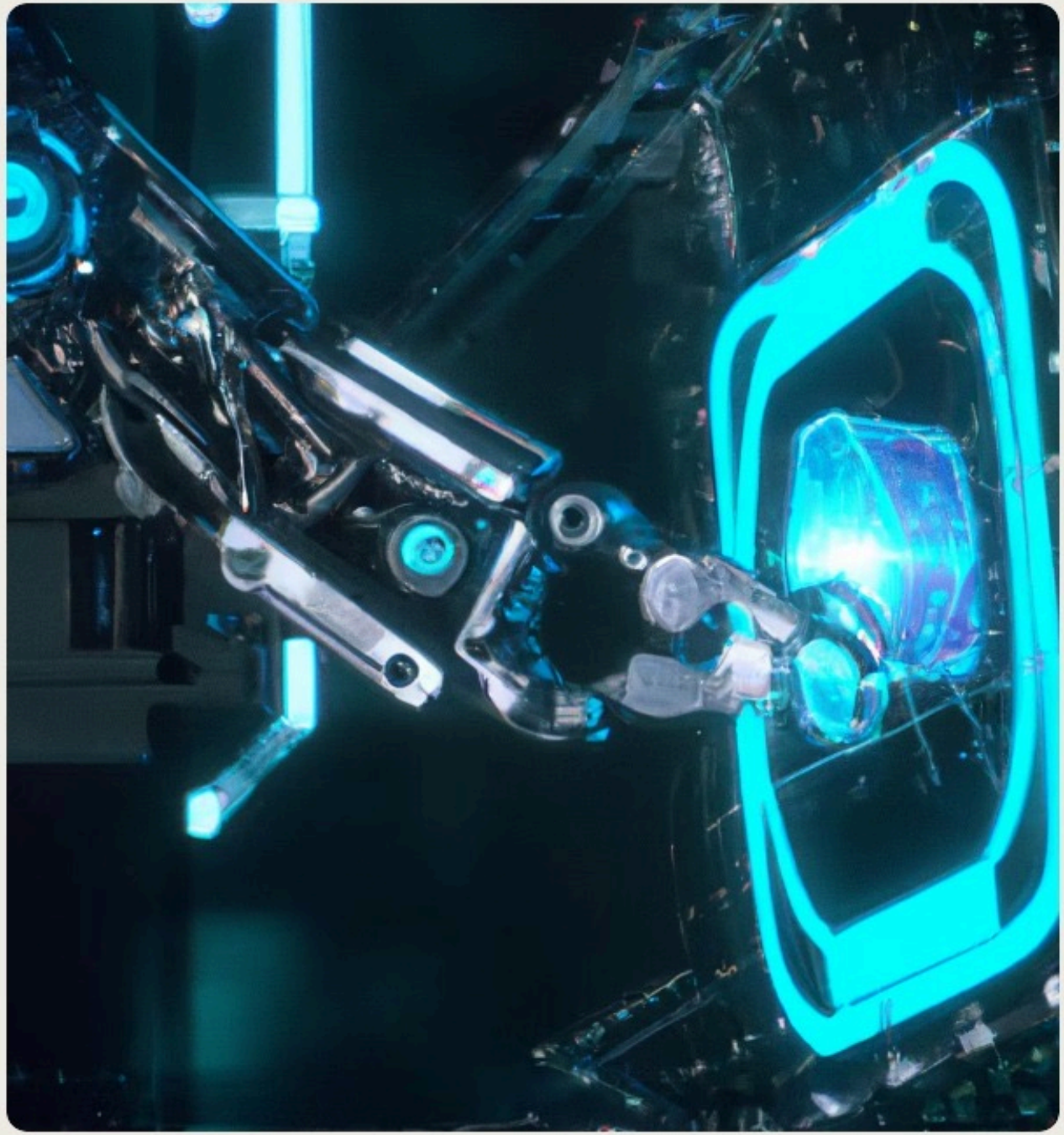
User manual

- First things first, the compiler asks the user to enter how many procceses are there.
- The compiler asks the user to enter the arrival time and the burst time for each procces
- The compiler repeats the same operation until you finally enter all the needed values.
- Finally, the complier displays the output of the response time, the throughput, and the average waiting time

Conclusion

Multilevel-feedback-queue scheduler is a powerful scheduling algorithm that allows for efficient utilization of resources and ensures that processes with higher priority are executed in a timely manner. It is used in many operating systems, real-time systems, cloud computing systems, and high-performance computing systems.

The scheduler also allows for dynamic adjustment of the priority of processes, ensuring that processes with higher priority are given more CPU time. However, the scheduler can be difficult to implement and maintain, and it requires a significant amount of resources.



References

1. Multilevel-Feedback-Queue Scheduling. (n.d.). Retrieved from https://www.geeksforgeeks.org/multilevel-feedback-queue-scheduling/

2. Multilevel Feedback Queue Scheduling. (n.d.). Retrieved from https://www.tutorialspoint.com/operating_system/os_multilevel_feedback_queue_scheduling.htm