T.C.
KONYA FOOD AND AGRICULTURE UNIVERSITY
FACULTY OF ENGINEERING AND ARCHITECTURE
DEPARTMENT OF COMPUTER ENGINEERING

CENG4003 - Operating Systems

GRADUATION PROJECT
TERM REPORT

Prepared by
Nursena Ertuğrul 182010020075
Şeyda Sıla Kara 192010020052
Ayşenur Sucu çap 21201002001

December, 2022

Description of the Project

A CPU Scheduling application was developed in the project. JavaScript is used for the backend and CSS is used for the frontend. FCFS, SJF, SRJF, PP, HRRN algorithms are designed for CPU scheduling. In line with the inputs received from the user, the sequences and durations of the processes are shown on the Gantt Chart, the Completion Time, Turnaround Time, Waiting Time and Response Time values calculated for each algorithm were calculated, displayed on the table and visualized, and comparisons were made between the algorithms.

Algorithms used in the project are divided into categories. Preemptive, Non-preemtive and round robin. Preemptive algorithms are algorithms that can be divided into parts after a part of the process has started and can be scheduled part by part. This is not the case in non-preemptive algorithms. Once a process begins to be scheduled, it must be finished as a whole. Since the round robin algorithm is not included in these two classes, it was found appropriate to create a separate category.

Technologies Used:

While developing the project, HTML, CSS, Vanilla JS, Google Charts and Chart.js technologies were used.

Methodology

In the functions created in the project, time quantum and priority were determined, inputs were taken, gcd and lcm values were calculated to update the colspan value, and process adding and deletion functions were created. The received inputs have processId, priority, arrival time, process time length, , total burst time, parameters. In addition to these, the algorithm, the category of the algorithm and time quantum information are also obtained. Time, remaining time, ready processes, running processes, block and termination parameters are created as outputs. A function is included to generate Average Times. In this function, averages of completion time turnaround time waiting time, respinse time and process values are calculated. For scheduling, the next process was selected among the candidate processes in which Ready Queue was created and updates were made. Then, the gantt chart and table are created with the selected algorithm, the type of this algorithm and the input values.

Background

CPU Scheduling is a process of determining which process will own CPU for execution while another process is on hold. The main task of CPU scheduling is to make sure that whenever the CPU remains idle, the OS at least select one of the processes available in the ready queue for execution. The selection process will be carried out by the CPU scheduler. It selects one of the processes in memory that are ready for execution.

Preemptive Scheduling

In Preemptive Scheduling, the tasks are mostly assigned with their priorities. Sometimes it is important to run a task with a higher priority before another lower priority task, even if the lower priority task is still running. The lower priority task holds for some time and resumes when the higher priority task finishes its execution.

Non-Preemptive Scheduling

In this type of scheduling method, the CPU has been allocated to a specific process. The process that keeps the CPU busy will release the CPU either by switching context or terminating. It is the only method that can be used for various hardware platforms. That's because it doesn't need special hardware (for example, a timer) like preemptive scheduling.

Burst Time/Execution Time: It is a time required by the process to complete execution. It is also called running time.

Arrival Time: when a process enters in a ready state

Finish Time: when process complete and exit from a system

Process: It is the reference that is used for both job and user.

CPU/IO burst cycle: Characterizes process execution, which alternates between CPU and I/O activity. CPU times are usually shorter than the time of I/O.

The main purpose of CPU scheduling algorithms is maximizing CPU utilization and throughput on the other hand minimizing turnaround, waiting and response times.

Maximize:

CPU utilization: CPU utilization is the main task in which the operating system needs to make sure that CPU remains as busy as possible. It can range from 0 to 100 percent. However, for the RTOS, it can be range from 40 percent for low-level and 90 percent for the high-level system.

Throughput: The number of processes that finish their execution per unit time is known Throughput. So, when the CPU is busy executing the process, at that time, work is being done, and the work completed per unit time is called Throughput.

Minimize:

Waiting time: Waiting time is an amount that specific process needs to wait in the ready queue.

Response time: It is an amount to time in which the request was submitted until the first response is produced.

Turnaround Time: Turnaround time is an amount of time to execute a specific process. It is the calculation of the total time spent waiting to get into the memory, waiting in the queue and, executing on the CPU. The period between the time of process submission to the completion time is the turnaround time.

CPU Scheduling Algorithms

First Come First Serve

First Come First Serve is the full form of FCFS. It is the easiest and most simple CPU scheduling algorithm. In this type of algorithm, the process which requests the CPU gets the CPU allocation first. This scheduling method can be managed with a FIFO queue.

As the process enters the ready queue, its PCB (Process Control Block) is linked with the tail of the queue. So, when CPU becomes free, it should be assigned to the process at the beginning of the queue.

Characteristics of FCFS method:

It offers non-preemptive and pre-emptive scheduling algorithm.

Jobs are always executed on a first-come, first-serve basis

It is easy to implement and use.

However, this method is poor in performance, and the general wait time is quite high.

Shortest Job First

SJF is a full form of (Shortest job first) is a scheduling algorithm in which the process with the shortest execution time should be selected for execution next. This scheduling method can be preemptive or non-preemptive. It significantly reduces the average waiting time for other processes awaiting execution.

Characteristics of SJF Scheduling

It is associated with each job as a unit of time to complete.

In this method, when the CPU is available, the next process or job with the shortest completion time will be executed first.

It is Implemented with non-preemptive policy.

This algorithm method is useful for batch-type processing, where waiting for jobs to complete is not critical.

It improves job output by offering shorter jobs, which should be executed first, which mostly have a shorter turnaround time.

Shortest Remaining Job First

The full form of SRJF is Shortest remaining time. It is also known as SJF preemptive scheduling. In this method, the process will be allocated to the task, which is closest to its completion. This method prevents a newer ready state process from holding the completion of an older process.

Characteristics of SRJF scheduling method:

This method is mostly applied in batch environments where short jobs are required to be given preference.This is not an ideal method to implement it in a shared system where the required CPU time is unknown.Associate with each process as the length of its next CPU burst. So that operating system uses these lengths, which helps to schedule the process with the shortest possible time.

Priority Based Scheduling

Priority scheduling is a method of scheduling processes based on priority. In this method, the scheduler selects the tasks to work as per the priority. Priority scheduling also helps OS to involve priority assignments. The processes with higher priority should be carried out first, whereas jobs with equal priorities are carried out on a round-robin or FCFS basis. Priority can be decided based on memory requirements, time requirements, etc.

Round-Robin Scheduling

Round robin is the oldest, simplest scheduling algorithm. The name of this algorithm comes from the round-robin principle, where each person gets an equal share of something in turn. It is mostly used for scheduling algorithms in multitasking. This algorithm method helps for starvation free execution of processes.

Characteristics of Round-Robin Scheduling

Round robin is a hybrid model which is clock-driven

Time slice should be minimum, which is assigned for a specific task to be processed. However, it may vary for different processes.

It is a real time system which responds to the event within a specific time limit.

Summary:

CPU scheduling is a process of determining which process will own CPU for execution while another process is on hold.

In Preemptive Scheduling, the tasks are mostly assigned with their priorities.

In the Non-preemptive scheduling method, the CPU has been allocated to a specific process.

The burst time is the time required for the process to complete execution. It is also called running time.

CPU utilization is the main task in which the operating system needs to ensure that the CPU remains as busy as possible.

The number of processes that finish their execution per unit time is known Throughput.

Waiting time is an amount that a specific process needs to wait in the ready queue.

It is the amount of time in which the request was submitted until the first response is produced.

Turnaround time is the amount of time to execute a specific process.

Timer interruption is a method that is closely related to preemption.

A dispatcher is a module that provides control of the CPU to the process.

Six types of process scheduling algorithms are: First Come First Serve (FCFS), 2) Shortest-Job-First (SJF) Scheduling, 3) Shortest Remaining Time, 4) Priority Scheduling, 5) Round Robin Scheduling, 6) Multilevel Queue Scheduling.

In the First Come First Serve method, the process which requests the CPU gets the CPU allocation first.

In the Shortest Remaining time, the process will be allocated to the task closest to its completion.

In Priority Scheduling, the scheduler selects the tasks to work as per the priority.

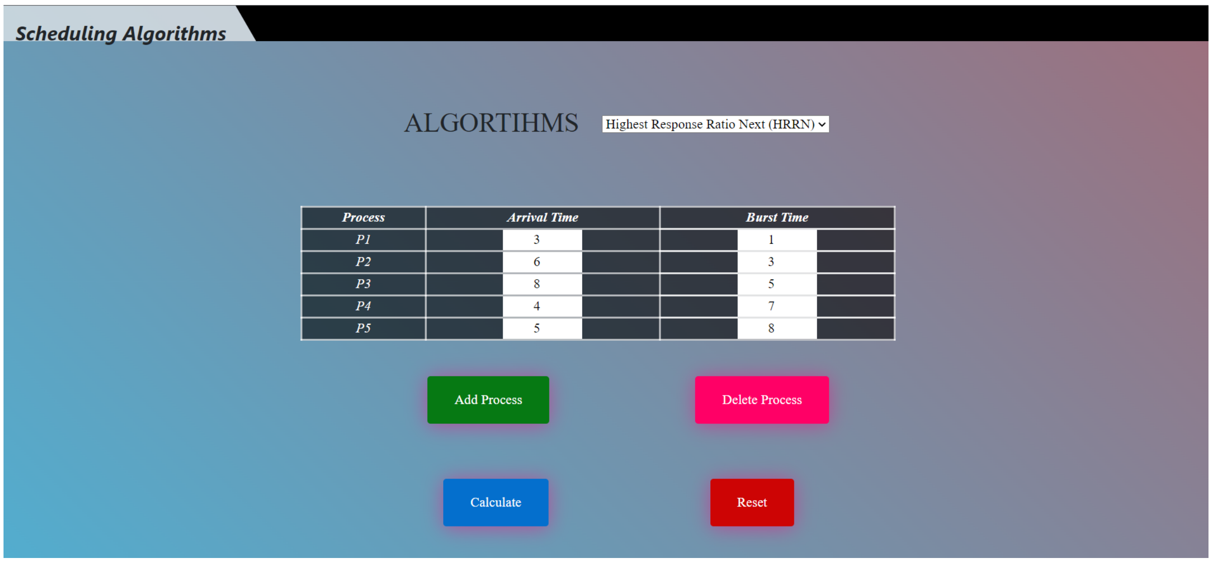
Round robin scheduling works on the principle where each person gets an equal share of something in turn.

In the Shortest job first, the shortest execution time should be selected for execution next.

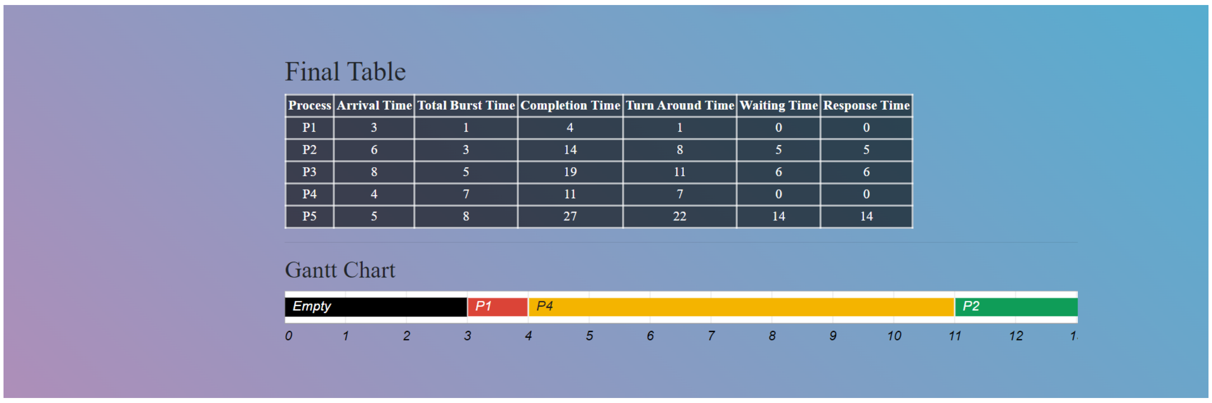
The multilevel scheduling method separates the ready queue into various separate queues. In this method, processes are assigned to a queue based on a specific property.

The CPU uses scheduling to improve its efficiency.

ScreenShots of the Project



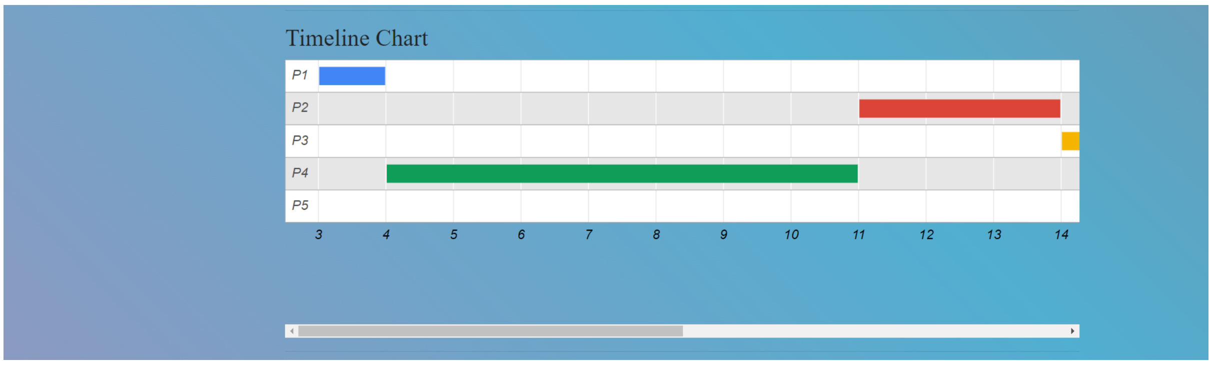
Process	Arrival Time	Burst Time
P1	3	1
P2	6	3
P3	8	5
P4	4	7
P5	5	8

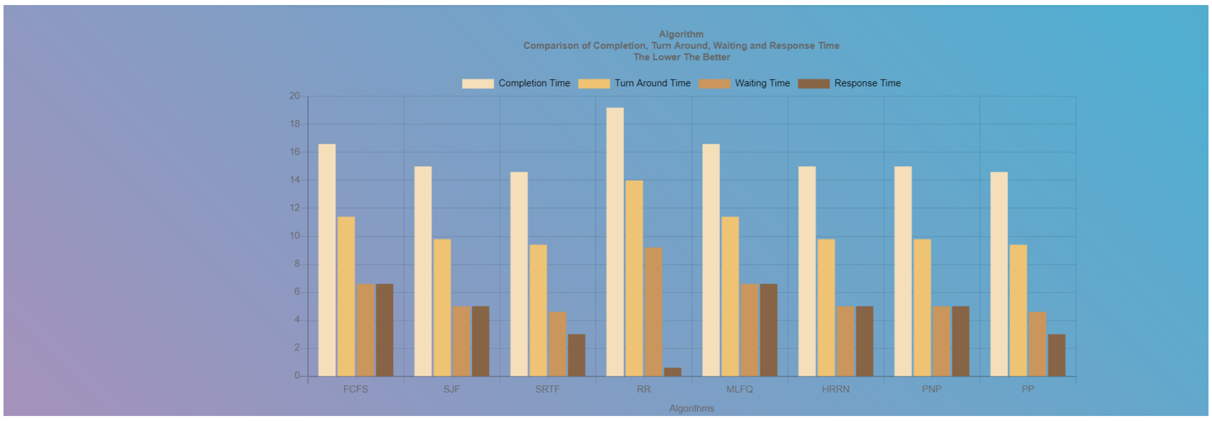


Final Table

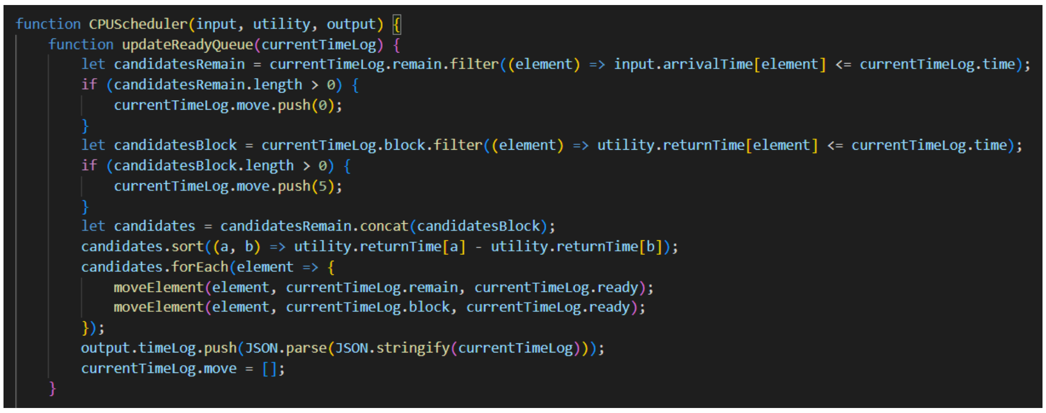
Process	Arrival Time	Total Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
P1	3	1	4	1	0	0
P2	6	3	14	8	5	5
P3	8	5	19	11	6	6
P4	4	7	11	7	0	0
P5	5	8	27	22	14	14

Gantt Chart

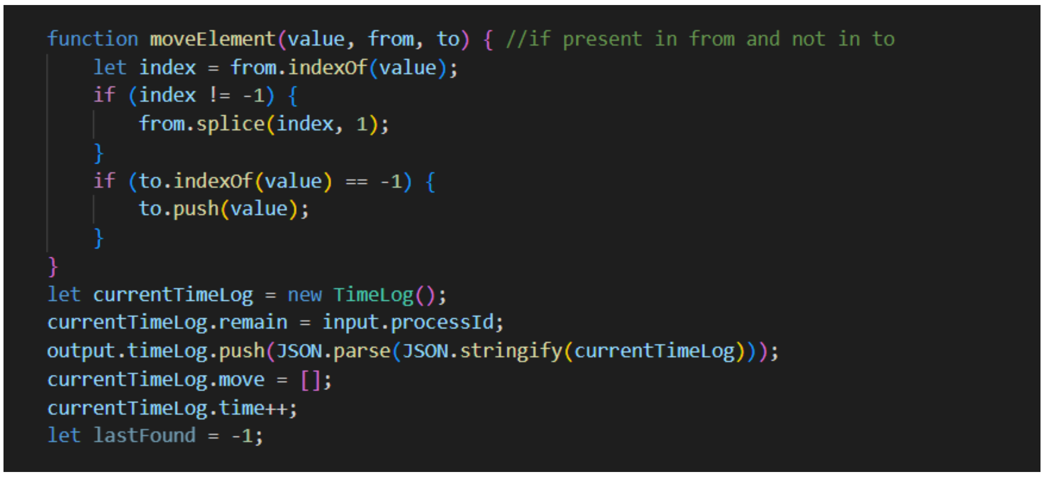




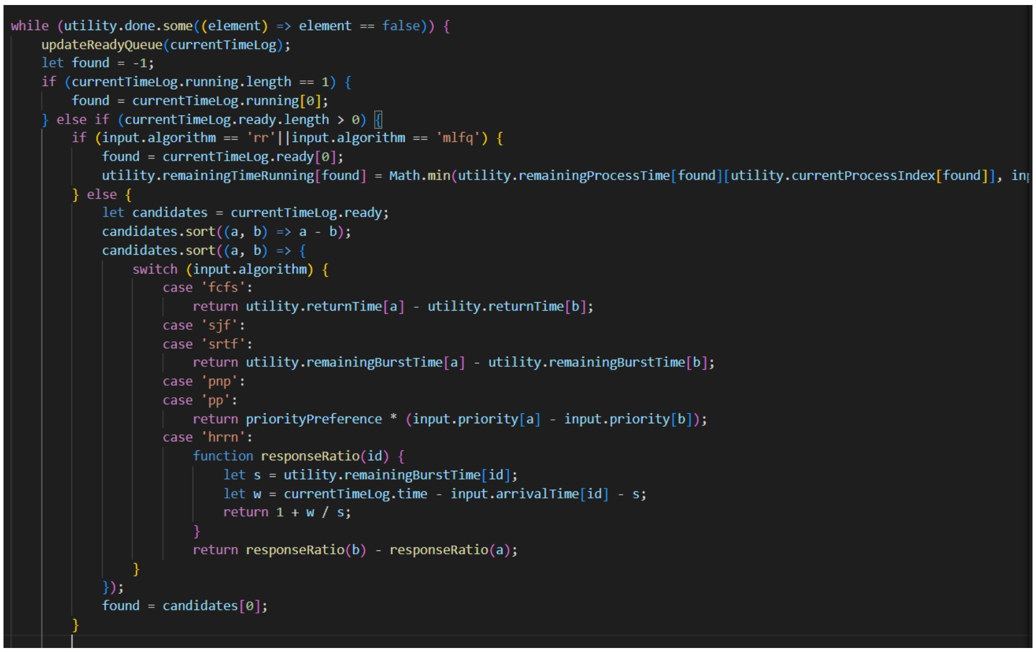
Code of CPUSchedular Function



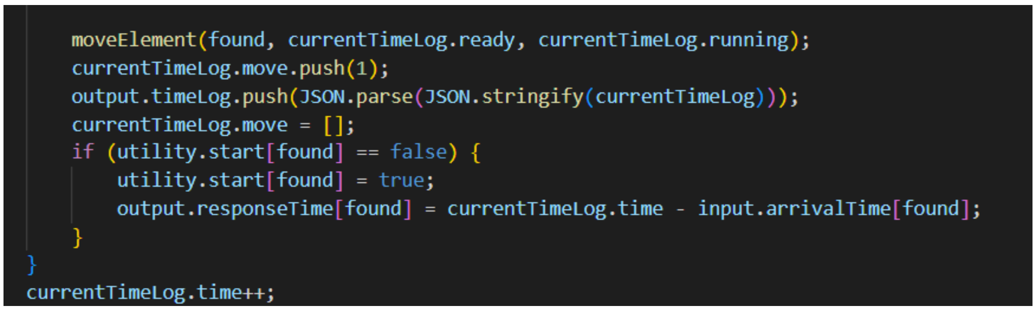
```
function CPUScheduler(input, utility, output) {
    function updateReadyQueue(currentTimeLog) {
        let candidatesRemain = currentTimeLog.remain.filter((element) => input.arrivalTime[element] <= currentTimeLog.time);
        if (candidatesRemain.length > 0) {
            currentTimeLog.move.push(0);
        }
        let candidatesBlock = currentTimeLog.block.filter((element) => utility.returnTime[element] <= currentTimeLog.time);
        if (candidatesBlock.length > 0) {
            currentTimeLog.move.push(5);
        }
        let candidates = candidatesRemain.concat(candidatesBlock);
        candidates.sort((a, b) => utility.returnTime[a] - utility.returnTime[b]);
        candidates.forEach(element => {
            moveElement(element, currentTimeLog.remain, currentTimeLog.ready);
            moveElement(element, currentTimeLog.block, currentTimeLog.ready);
        });
        output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
        currentTimeLog.move = [];
    }
```



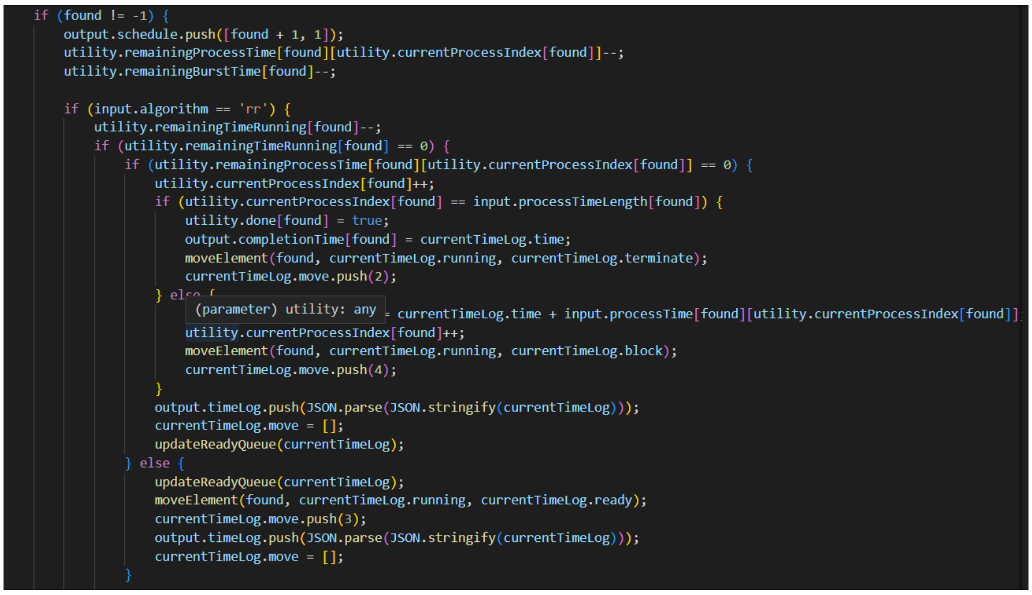
```
function moveElement(value, from, to) { //if present in from and not in to
    let index = from.indexOf(value);
    if (index != -1) {
        from.splice(index, 1);
    }
    if (to.indexOf(value) == -1) {
        to.push(value);
    }
}
let currentTimeLog = new TimeLog();
currentTimeLog.remain = input.processId;
output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
currentTimeLog.move = [];
currentTimeLog.time++;
let lastFound = -1;
```



```
while (utility.done.some((element) => element == false)) {
    updateReadyQueue(currentTimeLog);
    let found = -1;
    if (currentTimeLog.running.length == 1) {
        found = currentTimeLog.running[0];
    } else if (currentTimeLog.ready.length > 0) {
        if (input.algorithm == 'rr'||input.algorithm == 'mlfq') {
            found = currentTimeLog.ready[0];
            utility.remainingTimeRunning[found] = Math.min(utility.remainingProcessTime[found][utility.currentProcessIndex[found]], in
        } else {
            let candidates = currentTimeLog.ready;
            candidates.sort((a, b) => a - b);
            candidates.sort((a, b) => {
                switch (input.algorithm) {
                    case 'fcfs':
                        return utility.returnTime[a] - utility.returnTime[b];
                    case 'sjf':
                    case 'srtf':
                        return utility.remainingBurstTime[a] - utility.remainingBurstTime[b];
                    case 'pnp':
                    case 'pp':
                        return priorityPreference * (input.priority[a] - input.priority[b]);
                    case 'hrrn':
                        function responseRatio(id) {
                            let s = utility.remainingBurstTime[id];
                            let w = currentTimeLog.time - input.arrivalTime[id] - s;
                            return 1 + w / s;
                        }
                        return responseRatio(b) - responseRatio(a);
                }
            });
            found = candidates[0];
        }
```

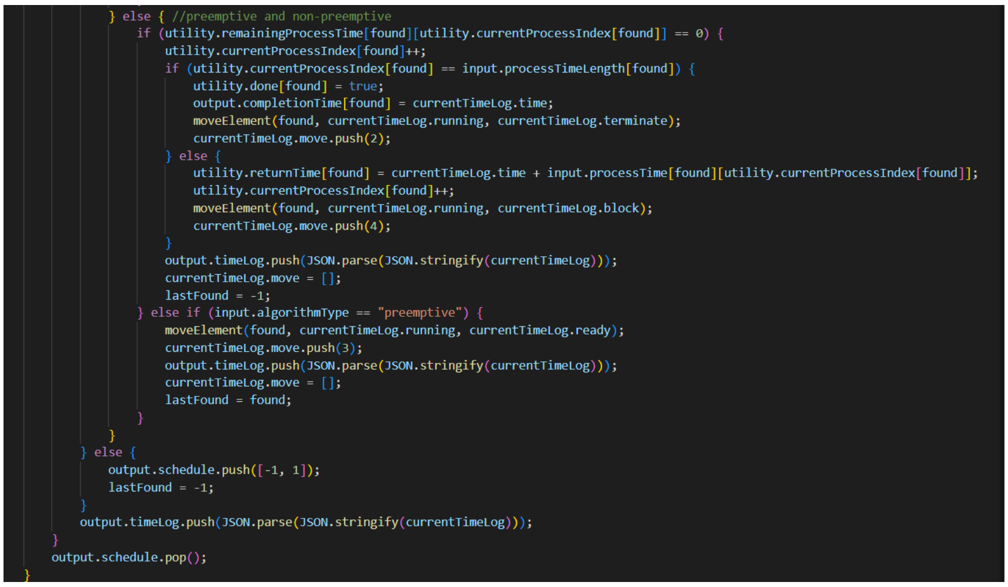


```
        moveElement(found, currentTimeLog.ready, currentTimeLog.running);
        currentTimeLog.move.push(1);
        output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
        currentTimeLog.move = [];
        if (utility.start[found] == false) {
            utility.start[found] = true;
            output.responseTime[found] = currentTimeLog.time - input.arrivalTime[found];
        }
    }
    currentTimeLog.time++;
```



```
    if (found != -1) {
        output.schedule.push([found + 1, 1]);
        utility.remainingProcessTime[found][utility.currentProcessIndex[found]]--;
        utility.remainingBurstTime[found]--;

        if (input.algorithm == 'rr') {
            utility.remainingTimeRunning[found]--;
            if (utility.remainingTimeRunning[found] == 0) {
                if (utility.remainingProcessTime[found][utility.currentProcessIndex[found]] == 0) {
                    utility.currentProcessIndex[found]++;
                    if (utility.currentProcessIndex[found] == input.processTimeLength[found]) {
                        utility.done[found] = true;
                        output.completionTime[found] = currentTimeLog.time;
                        moveElement(found, currentTimeLog.running, currentTimeLog.terminate);
                        currentTimeLog.move.push(2);
                    } el
                        (parameter) utility: any = currentTimeLog.time + input.processTime[found][utility.currentProcessIndex[found]]
                        utility.currentProcessIndex[found]++;
                        moveElement(found, currentTimeLog.running, currentTimeLog.block);
                        currentTimeLog.move.push(4);
                    }
                    output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
                    currentTimeLog.move = [];
                    updateReadyQueue(currentTimeLog);
                } else {
                    updateReadyQueue(currentTimeLog);
                    moveElement(found, currentTimeLog.running, currentTimeLog.ready);
                    currentTimeLog.move.push(3);
                    output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
                    currentTimeLog.move = [];
                }
```



```
            } else { //preemptive and non-preemptive
                if (utility.remainingProcessTime[found][utility.currentProcessIndex[found]] == 0) {
                    utility.currentProcessIndex[found]++;
                    if (utility.currentProcessIndex[found] == input.processTimeLength[found]) {
                        utility.done[found] = true;
                        output.completionTime[found] = currentTimeLog.time;
                        moveElement(found, currentTimeLog.running, currentTimeLog.terminate);
                        currentTimeLog.move.push(2);
                    } else {
                        utility.returnTime[found] = currentTimeLog.time + input.processTime[found][utility.currentProcessIndex[found]];
                        utility.currentProcessIndex[found]++;
                        moveElement(found, currentTimeLog.running, currentTimeLog.block);
                        currentTimeLog.move.push(4);
                    }
                    output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
                    currentTimeLog.move = [];
                    lastFound = -1;
                } else if (input.algorithmType == "preemptive") {
                    moveElement(found, currentTimeLog.running, currentTimeLog.ready);
                    currentTimeLog.move.push(3);
                    output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
                    currentTimeLog.move = [];
                    lastFound = found;
                }
            }
        } else {
            output.schedule.push([-1, 1]);
            lastFound = -1;
        }
        output.timeLog.push(JSON.parse(JSON.stringify(currentTimeLog)));
    }
    output.schedule.pop();
}
```