CPU Scheduler

Shahed Ahmed
10-26-2024

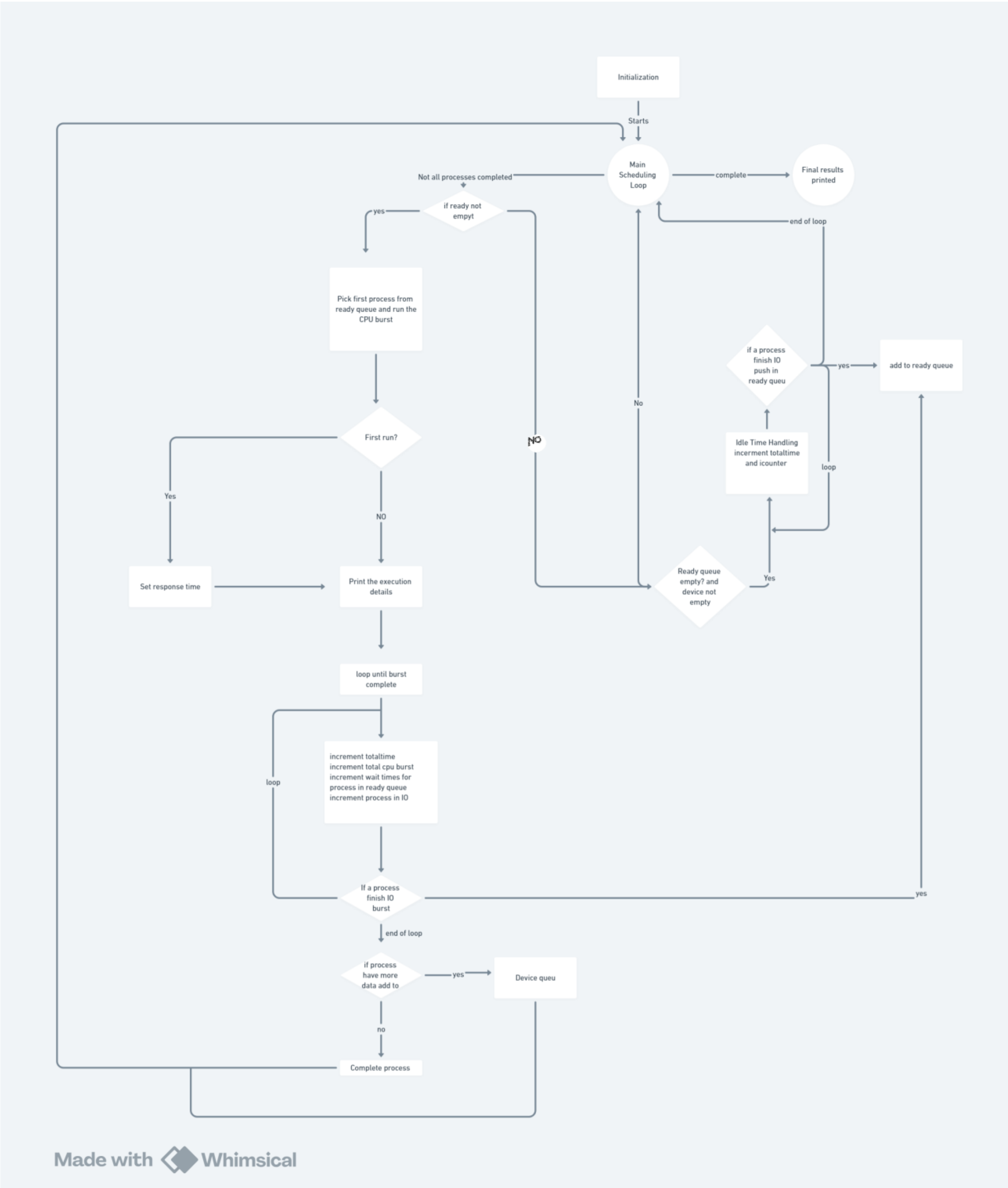
Introduction

This project focuses on exploring different CPU scheduling algorithms by implementing and simulating three common approaches: First-Come-First-Serve (FCFS), Shortest Job First (SJF), and Multilevel Feedback Queue (MLFQ). CPU scheduling is a key function of operating systems, determining how processes share the CPU and are executed over time. By simulating these algorithms, we can better understand how they handle tasks and measure their effectiveness based on performance metrics like CPU utilization, waiting time, turnaround time, and response time. Scheduling is essential to ensure that system resources are used efficiently while maintaining fairness among processes. Each scheduling algorithm operates differently: FCFS is straightforward and processes tasks in the order they arrive, while SJF selects the process with the shortest CPU burst time to minimize the average waiting time. MLFQ is a more complex and flexible approach, using multiple levels of queues to prioritize tasks and preempt lower-priority processes when necessary. In this simulation, we will use a set of predefined processes, each with a series of CPU bursts and I/O operations. These processes will be scheduled by each algorithm, and we'll track their performance based on key metrics. By comparing the results of each approach, we can analyze how each scheduling algorithm performs in different scenarios, giving us valuable insights into which strategies work best for different types of systems.

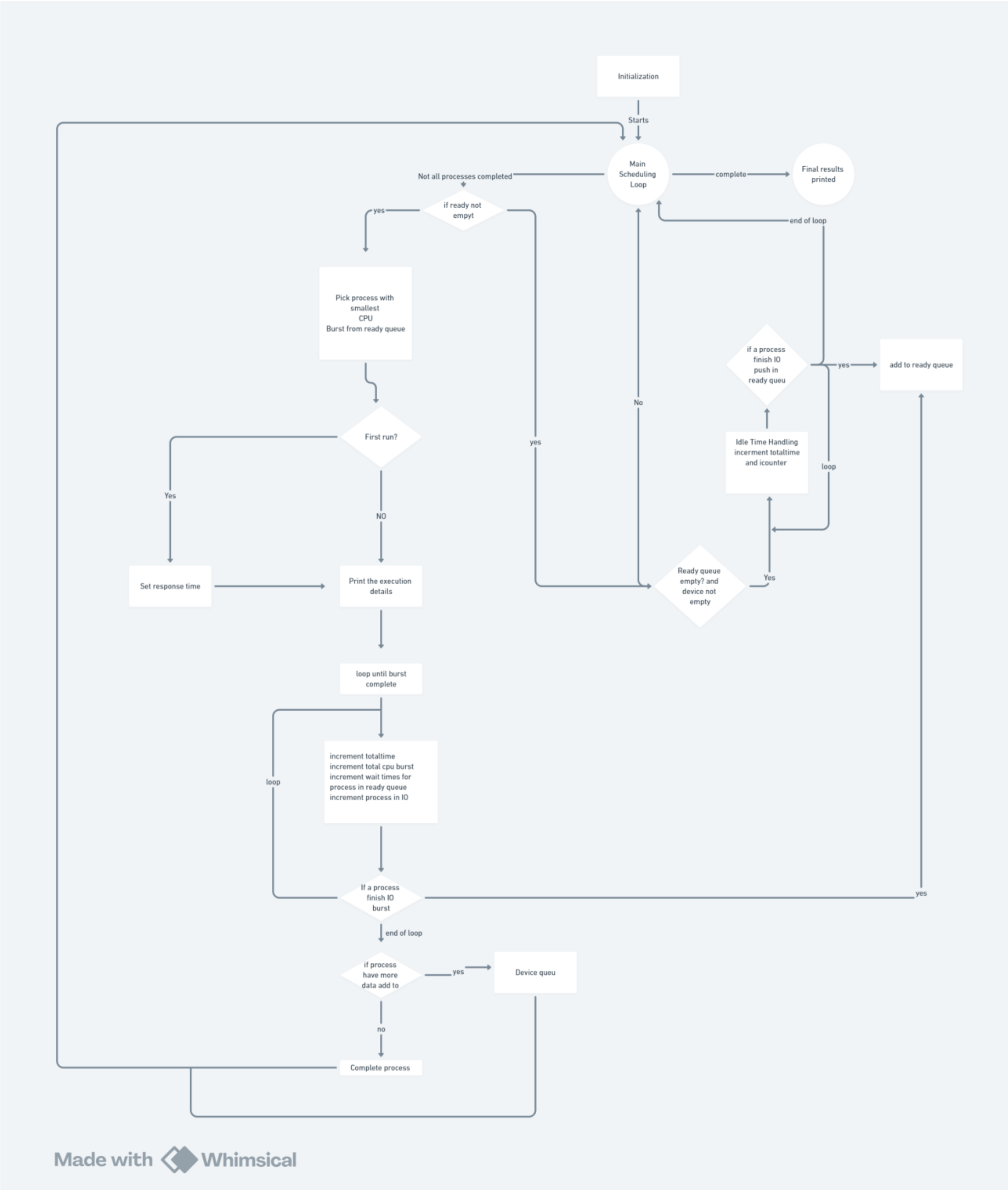
Contents

Flow Chart

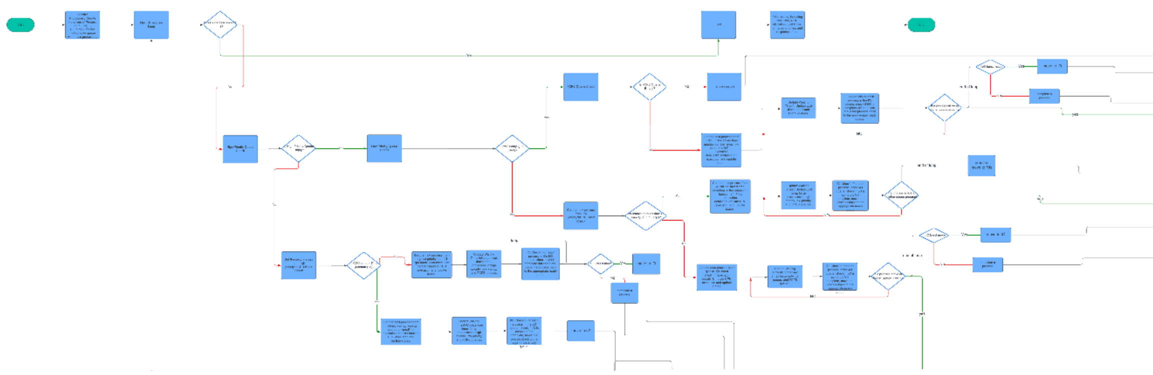
First Come First Serve



Short Job First



Multilevel Feedback Queue



https://lucid.app/lucidchart/2738390e-dd74-4769-9390-a8b825b3aeb5/edit?view_items=I2flJk_xfdNj&invitationId=inv_9cb3b1ae-7b23-4304-a680-06f9f940d394

Discussion and Tables

First Come First Serve

The First-Come, First-Serve (FCFS) scheduling algorithm results in larger turnaround and response times due to its sequential ordering of execution. As processes at the end of the ready queue have to wait longer to finish, this leads to significant inefficiencies in overall system performance. Although CPU utilization is relatively high at 85.0769%, the simplicity of FCFS makes it not very efficient. Consequently, The table shows FCFS can lead to substantial inefficiencies in system performance.

Process	Tw	Ttr	Tr
P1	170	395	0
P2	164	591	5
P3	165	557	9
P4	164	648	17
P5	221	530	20
P6	230	445	36
P7	184	512	47
P8	184	493	61
Avg	185	521.375	24.375
	CPU Utilization = 85.3395%		

Shortest Job First

Although the shortest job shows a small average wait time, small average turnaround time, and a large average response time, it prioritizes processes with low runtimes. This causes larger processes to experience starvation, leading to longer response times, wait times, and turnaround times. The CPU often remains idle, resulting in lower CPU utilization.

Process	Tw	Ttr	Tr
P1	43	268	11
P2	73	500	3
P3	276	668	16
P4	50	534	0
P5	237	546	109
P6	121	336	24
P7	149	477	47
P8	119	428	7
Avg	133.5	469.625	27.125
CPU Utilization = 82.7844%			

Multilevel Feedback Queue

The multilevel feedback queue (MLFQ) demonstrates low average wait time because processes in the higher-priority queues do not have to wait for processes in the lower-priority queues. Lower-priority queues get preempted by higher-priority queues, allowing processes with shorter wait times to execute sooner. Although some processes experience longer wait times, they eventually receive CPU time. With this fair scheduling system, the likelihood of the CPU becoming idle is reduced. As a result, CPU utilization increases, wait times are optimized, and processes do not experience long delays. Turnaround times and response times also improve because the system ensures that every process has a chance to run from the start, preventing starvation.

Process	Tw	Ttr	Tr
P1	40	253	0
P2	141	566	5
P3	220	607	9
P4	17	492	14
P5	282	591	17
P6	177	355	22
P7	229	548	27
P8	140	448	32
Avg	155	482.5	15.75
CPU Utilization = 90.8197%			

Compare results SJF, FCFS, MLFQ

In the table, Shortest Job First (SJF) shows the lowest average waiting time and turnaround time compared to First Come First Serve (FCFS) and Multilevel Feedback Queue (MLFQ). However, the response time for SJF is higher because processes with shorter burst times may cause starvation for longer burst processes. Additionally, CPU utilization in SJF is lower than the other two algorithms because the CPU stays in idle more than First Come First Serve(FCFS) and Multilevel Feedback Queue(MLFQ).FCFS displays middle-range performance. It has a higher turnaround time and average waiting time than SJF and MLFQ because it follows a first-come, first-served order, causing longer processes to block shorter ones. However, FCFS achieves better CPU utilization than SJF but not MLFQ due to the continuous execution of processes in the order they arrive.Finally, MLFQ provides the best overall performance in terms of CPU utilization and balances response time. It prevents starvation and achieves a lower response time than the other two algorithms by ensuring that every process gets an opportunity to run in a fair manner, especially at the beginning. This leads to a better balance of waiting time, turnaround time, and CPU efficiency.

	FCFS	**SJG**	**MLFQ**
CPU utilization	85.3395%	82.7844%	90.8197%
Avg Waiting time (Tw)	185	133.5	155
Avg Turnaround time (Ttr)	521.375	469.625	482.5
Avg Response time (Tr)	24.375	27.125	15.75

Sample of dynamic execution

FCFS

```
---------------------------------------------------
Execution Time is 400            P7 is Running
Processes in Ready Queue:
P3 CPU Burst Length 15
P5 CPU Burst Length 11
Processes in I/O:
P2 Time Remaining in I/O: 4
P8 Time Remaining in I/O: 63
P6 Time Remaining in I/O: 21
P4 Time Remaining in I/O: 54

Completed Execution= NO

---------------------------------------------------



---------------------------------------------------
Execution Time is 407            P3 is Running
Processes in Ready Queue:
P5 CPU Burst Length 11
P2 CPU Burst Length 4
Processes in I/O:
P8 Time Remaining in I/O: 56
P6 Time Remaining in I/O: 14
P4 Time Remaining in I/O: 47
P7 Time Remaining in I/O: 19

Completed Execution= NO

---------------------------------------------------



---------------------------------------------------
Execution Time is 422            P5 is Running
Processes in Ready Queue:
P2 CPU Burst Length 4
P6 CPU Burst Length 8
Processes in I/O:
P8 Time Remaining in I/O: 41
P4 Time Remaining in I/O: 32
P7 Time Remaining in I/O: 4
P3 Time Remaining in I/O: 18

Completed Execution= NO

---------------------------------------------------
```

SJF

```
---------------------------------------------------
Execution Time is 406           P7 is Running
Processes in Ready Queue:
Processes in I/O:
P8 Time Remaining in I/O: 5
P4 Time Remaining in I/O: 38
P3 Time Remaining in I/O: 39
P5 Time Remaining in I/O: 19
P2 Time Remaining in I/O: 28

Completed Execution= NO

---------------------------------------------------


---------------------------------------------------
Execution Time is 422           P8 is Running
Processes in Ready Queue:
Processes in I/O:
P4 Time Remaining in I/O: 22
P3 Time Remaining in I/O: 23
P5 Time Remaining in I/O: 3
P2 Time Remaining in I/O: 12
P7 Time Remaining in I/O: 33

Completed Execution= YES

---------------------------------------------------


---------------------------------------------------
Execution Time is 428           P5 is Running
Processes in Ready Queue:
Processes in I/O:
P4 Time Remaining in I/O: 16
P3 Time Remaining in I/O: 17
P2 Time Remaining in I/O: 6
P7 Time Remaining in I/O: 27

Completed Execution= NO

---------------------------------------------------
```

MLFQ

```
----------------------------------------------------
Current Time: 421 | Process P5 is Running

High Priority Queue:
None

Low Priority Queue:
None

FCFS Queue:
None

Processes in I/O:
P8 (Remaining I/O Time: 16)
P2 (Remaining I/O Time: 16)
P3 (Remaining I/O Time: 40)
P7 (Remaining I/O Time: 15)
P4 (Remaining I/O Time: 77)
----------------------------------------------------

Completed Execution= NO

----------------------------------------------------


----------------------------------------------------
Current Time: 436 | Process P7 is Running

High Priority Queue:
None

Low Priority Queue:
None

FCFS Queue:
None

Processes in I/O:
P8 (Remaining I/O Time: 1)
P2 (Remaining I/O Time: 1)
P3 (Remaining I/O Time: 25)
P4 (Remaining I/O Time: 62)
P5 (Remaining I/O Time: 17)
----------------------------------------------------

Completed Execution= NO

----------------------------------------------------


----------------------------------------------------
Current Time: 443 | Process P8 is Running

High Priority Queue:
None

Low Priority Queue:
None

FCFS Queue:
P2 (CPU Burst: 9)

Processes in I/O:
P3 (Remaining I/O Time: 18)
P4 (Remaining I/O Time: 55)
P5 (Remaining I/O Time: 10)
P7 (Remaining I/O Time: 19)
----------------------------------------------------

Completed Execution= YES

----------------------------------------------------
```

Results

FSFC

Total Time = 648
CPU Utilization = 85.3395%

	Wait Times	Turnaround Times	Response Times
p1	170	395	0
p2	164	591	5
p3	165	557	9
p4	164	648	17
p5	221	530	20
p6	230	445	36
p7	184	512	47
p8	184	493	61
Avg	185	521.375	24.375

SJF

Total Time = 668
CPU Utilization = 82.7844%

	Wait Times	Turnaround Times	Response Times
p1	43	268	11
p2	73	500	3
p3	276	668	16
p4	50	534	0
p5	237	546	109
p6	121	336	24
p7	149	477	47
p8	119	428	7
Avg	133.5	469.625	27.125

MLFQ

```
Total Time = 610
CPU Utilization = 90.8197%
Wait Times    Turnaround Times      Response Times
-------------------------------------------------------
 p1|      40              253                    0
 p2|     141              566                    5
 p3|     220              607                    9
 p4|      17              492                   14
 p5|     282              591                   17
 p6|     177              355                   22
 p7|     229              548                   27
 p8|     140              448                   32
-------------------------------------------------------
Avg|     155            482.5                15.75
```

Partial source code

FSCF

```
int main() {
    Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
    Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
    Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
    Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
    Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
    Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
    Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
    Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});

    readyqueue.push(&p1);
    readyqueue.push(&p2);
    readyqueue.push(&p3);
    readyqueue.push(&p4);
    readyqueue.push(&p5);
    readyqueue.push(&p6);
    readyqueue.push(&p7);
    readyqueue.push(&p8);

    int cpuburst = 0;
    int numprocesscomplete = 0;

    // Main scheduling loop
    while (numprocesscomplete != 8) {
        if (!readyqueue.empty()) {
            Process* p = readyqueue.front(); // Get the next process
            readyqueue.pop();

            // First run logic
            if (p->firstrun) {
                p->responsetime = p->wait;  // Capture the response time when first run
                p->firstrun = false;
            }

            cpuburst = p->data[0]; // Get the CPU burst time
            p->totalcpu += cpuburst; // Accumulate total CPU time
            p->data.erase(p->data.begin()); // Remove the used burst time

            printexecution(p); // Print the execution details

            // Execute CPU burst
            for (int i = 0; i < cpuburst; i++) {
                totalallcpu++; // Increment total CPU time
                totaltime++; // Increment total time

                for (int j = 0; j < readyqueue.size(); j++) {
                    Process* temp = readyqueue.front();
                    readyqueue.pop();
                    temp->wait++; // Increment wait time
                    readyqueue.push(temp); // Push back
                }

                // Check for processes in I/O
                for (int k = 0; k < deviceQueue.size(); k++) {
                    deviceQueue[k]->iocounter++; // Increment I/O counter

                    // If I/O is complete, move back to the ready queue
                    if (deviceQueue[k]->iocounter == deviceQueue[k]->ioburst) {
                        deviceQueue[k]->ioburst = 0;
                        deviceQueue[k]->iocounter = 0; // Reset I/O counter
                        readyqueue.push(deviceQueue[k]); // Push to ready queue
                        deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
                        k--; // Adjust index due to removal
                    }
                }

            }
```

```
            // Handle I/O burst
            if (!p->data.empty()) {
                p->ioburst = p->data[0]; // Set the I/O burst
                p->totalio += p->ioburst; // Accumulate I/O time
                p->data.erase(p->data.begin()); // Remove the used burst time
                deviceQueue.push_back(p); // Move to I/O
                cout << "\nCompleted Execution= NO" << endl;
            } else {
                p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
                numprocesscomplete++; // Increment completed process count
                cout << "\nCompleted Execution= YES" << endl;
            }
            cout << "\n---------------------------------------------------\n\n\n";
        }

    else if (readyqueue.empty() && !deviceQueue.empty()) { // Handles CPU Idle incrementing Processes in I/O Burst counters
            totaltime++; // Increment total time once per iteration
            // Iterate through the device queue and update I/O counters
            for (int n = 0; n < deviceQueue.size(); n++) {
                deviceQueue[n]->iocounter++; // Increment I/O counter for each process

                // Check if the I/O is complete
                if (deviceQueue[n]->iocounter >= deviceQueue[n]->ioburst) {
                    // Move the completed process back to the ready queue
                    deviceQueue[n]->ioburst = 0;
                    deviceQueue[n]->iocounter = 0;
                    readyqueue.push(deviceQueue[n]);
                    deviceQueue.erase(deviceQueue.begin() + n); // Remove from device queue
                    n--; // Adjust index because we removed the current element
                }
            }
        }

    }
    printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results
    return 0;
}
```

SJF

```
int main() {
    Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
    Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
    Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
    Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
    Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
    Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
    Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
    Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});

    readyqueue.push_back(&p1);
    readyqueue.push_back(&p2);
    readyqueue.push_back(&p3);
    readyqueue.push_back(&p4);
    readyqueue.push_back(&p5);
    readyqueue.push_back(&p6);
    readyqueue.push_back(&p7);
    readyqueue.push_back(&p8);

    int cpuburst = 0;
    int numprocesscomplete = 0;

    // Main scheduling loop
    while (numprocesscomplete != 8) {
        if (!readyqueue.empty()) {

            // get the process with the smallest cpu burst
             auto min_it = min_element(readyqueue.begin(), readyqueue.end(), [](Process* a,
Process* b) {
                return a->data.front() < b->data.front();
            });

            Process* p = *min_it; // Get the process with the smallest CPU burst
            readyqueue.erase(min_it); // Remove it from the ready queue

            // First run logic
            if (p->firstrun) {
                p->responsetime = p->wait;  // Capture the response time when first run
                p->firstrun = false;
            }

            cpuburst = p->data[0]; // Get the CPU burst time
            p->totalcpu += cpuburst; // Accumulate total CPU time
            p->data.erase(p->data.begin()); // Remove the used burst time

            printexecution(p); // Print the execution details

```

```
            // Execute CPU burst
            for (int i = 0; i < cpuburst; i++) {
                totalallcpu++; // Increment total CPU time
                totaltime++; // Increment total time

                for (int j = 0; j < readyqueue.size(); j++) {
                    Process* temp = readyqueue[j];
                    temp->wait++; // Increment wait time

                }

                // Check for processes in I/O
                for (int k = 0; k < deviceQueue.size(); k++) {
                    deviceQueue[k]->iocounter++; // Increment I/O counter

                    // If I/O is complete, move back to the ready queue
                    if (deviceQueue[k]->iocounter == deviceQueue[k]->ioburst) {
                        deviceQueue[k]->ioburst = 0;
                        deviceQueue[k]->iocounter = 0; // Reset I/O counter
                        readyqueue.push_back(deviceQueue[k]); // Push to ready queue
                        deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
                        k--; // Adjust index due to removal
                    }
                }
            }

            // Handle I/O burst
            if (!p->data.empty()) {
                p->ioburst = p->data[0]; // Set the I/O burst
                p->totalio += p->ioburst; // Accumulate I/O time
                p->data.erase(p->data.begin()); // Remove the used burst time
                deviceQueue.push_back(p); // Move to I/O
                cout << "\nCompleted Execution= NO" << endl;
            } else {
                p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
                numprocesscomplete++; // Increment completed process count
                cout << "\nCompleted Execution= YES" << endl;
            }
            cout << "\n---------------------------------------------------\n\n\n";
        }

        else if (readyqueue.empty() && !deviceQueue.empty()) { // Handles CPU Idle incrementing Processes in I/O Burst counters
            totaltime++; // Increment total time once per iteration
            // Iterate through the device queue and update I/O counters
            for (int n = 0; n < deviceQueue.size(); n++) {
                deviceQueue[n]->iocounter++; // Increment I/O counter for each process

                // Check if the I/O is complete
                if (deviceQueue[n]->iocounter >= deviceQueue[n]->ioburst) {
                    // Move the completed process back to the ready queue
                    deviceQueue[n]->ioburst = 0;
                    deviceQueue[n]->iocounter = 0;
                    readyqueue.push_back(deviceQueue[n]);
                    deviceQueue.erase(deviceQueue.begin() + n); // Remove from device queue
                    n--; // Adjust index because we removed the current element
                }
            }
        }
    }

    printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results
    return 0;
}
```

MLFQ

```cpp
int main() {
    int Tq = 5; // Time quantum for high-priority processes
    int Tq2 = 10; // Time quantum for low-priority processes

    // Queue to transfer processes from one queue to another
    queue<Process*> collect;

    // Initialize processes with their respective burst times
    Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
    Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
    Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
    Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
    Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
    Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
    Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
    Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});

    // Populate the high priority queue with the processes
    highPriority.push_back(&p1);
    highPriority.push_back(&p2);
    highPriority.push_back(&p3);
    highPriority.push_back(&p4);
    highPriority.push_back(&p5);
    highPriority.push_back(&p6);
    highPriority.push_back(&p7);
    highPriority.push_back(&p8);

    int cpuburst = 0; // Variable to track the current CPU burst time
```

```
    int numprocesscomplete = 0; // Counter for completed processes

    // Main scheduling loop
    while (numprocesscomplete != 8) { // Continue until all processes are complete

        if (!highPriority.empty()) {
            Process* p = highPriority.front(); // Get the next process from high priority

            // Execute CPU burst
            if (p->data[0] > Tq) { // If the burst time is greater than the time quantum
                p->data[0] -= Tq; // Reduce the burst time
                cpuburst = Tq; // Set the CPU burst time
                p->Queue = 2;
                collect.push(highPriority.front()); // Move process to collect queue
                highPriority.pop_front(); // Remove process from high priority

            }
            else { // If the burst time is less than or equal to the time quantum
                cpuburst = p->data[0]; // Set CPU burst time to remaining burst
                p->data.erase(p->data.begin()); // Remove the burst time
                highPriority.pop_front(); // Remove process from high priority

            }

            // First run logic to capture response time
            if (p->firstrun) {
                p->responsetime = p->wait;  // Capture the response time when first run
                p->firstrun = false; // Mark as first run completed
            }

            p->totalcpu += cpuburst; // Accumulate total CPU time

            // Print execution details for the current process
            printexecution(p);

            // Simulate CPU execution for the duration of the burst
            for (int i = 0; i < cpuburst; i++) {
                totalallcpu++; // Increment total CPU time
                totaltime++; // Increment total time

                // Update waiting times for processes in highPriority
                for (int j = 0; j < highPriority.size(); j++) {
                   highPriority[j]->wait++; // Increment wait time for each process
                }

                // Update waiting times for processes in q2
                for (int j = 0; j < q2.size(); j++) {
                  q2[j]->wait++; // Increment wait time for each process
                }

                // Update waiting times for processes in FCFS queue
                for (int j = 0; j < fcfs.size(); j++) {
                    fcfs[j]->wait++; // Increment wait time for each process
                }

                // Check for processes in I/O and increment their counters
                for (int k = 0; k < deviceQueue.size(); k++) {
                    deviceQueue[k]->iocounter++; // Increment I/O counter

                    // If I/O is complete, move back to the ready queue
                    if (deviceQueue[k]->iocounter >= deviceQueue[k]->ioburst) {
                        deviceQueue[k]->ioburst = 0;
                        deviceQueue[k]->iocounter = 0; // Reset I/O counter

                        if (deviceQueue[k]->Queue == 1)
                            highPriority.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 2)
                            q2.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 3)
                            fcfs.push_back(deviceQueue[k]); // Move to ready queue
                        deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
                        k--; // Adjust index due to removal
                    }
                }
            }
```

```
            // If there are processes to move from collect to q2
            if (!collect.empty()) {
                q2.push_back(collect.front()); // Move from collect to q2
                collect.pop(); // Pop the front of collect
            } else {
                // Handle I/O burst for current process
                if (!p->data.empty()) {
                    p->ioburst = p->data[0]; // Set the I/O burst
                    p->totalio += p->ioburst; // Accumulate I/O time
                    deviceQueue.push_back(p); // Move to I/O
                    p->data.erase(p->data.begin()); // Remove the used burst time

                    cout << "\nCompleted Execution= NO" << endl;
                } else {
                    p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
                    numprocesscomplete++; // Increment completed process count
                    cout << "\nCompleted Execution= YES" << endl;
                }
                cout << "\n---------------------------------------------------\n\n\n";
            }

//////////////////////////////////////////////////////////////////////////////////////////////////////
/////////////////////////////////////////////////////////

        } else if (highPriority.empty() && !q2.empty()) {
            Process* p = q2.front(); // Get the next process from low priority
            q2.pop_front(); // Remove from low priority
            // Execute CPU burst
            if (p->data[0] + p->counter > Tq2) { // Check if the remaining burst time exceeds time quantum
                p->Queue = 3;

                if (p->tqsave != 0) {

                    p->data[0] -= p->tqsave; // Use the saved time quantum
                    cpuburst = p->tqsave; // Set the CPU burst time
                    p->tqsave = 0; // Reset saved time quantum
                    p->counter = 0; // Reset counter
                    collect.push(p); // Move process to collect queue
                }
                else {
                    p->data[0] -= Tq2; // Use the full time quantum
                    cpuburst = Tq2; // Set the CPU burst time
                    p->totalcpu += cpuburst; // Accumulate total CPU time
                    collect.push(p); // Move process to collect queue

                }
            }
            else { // If the remaining burst time is less than or equal to the time quantum
                p->Queue = 2;

                if (p->tqsave != 0) {
                    cpuburst = p->tqsave; // Set CPU burst time to saved time quantum
                    p->data.erase(p->data.begin()); // Remove the burst time
                    p->tqsave = 0; // Reset saved time quantum
                }
                else {

                    cpuburst = p->data[0]; // Set CPU burst time to remaining burst time
                    p->data.erase(p->data.begin()); // Remove the burst time

                }
            }
```

```
            // Print execution details for the current low-priority process
            printexecution(p);
            // Execute CPU burst
            for (int i = 0; i < cpuburst; i++) {
                p->counter++; // Increment counter for CPU burst
                totalallcpu++; // Increment total CPU time
                totaltime++; // Increment total time

                // Update waiting times for processes in q2
                for (int j = 0; j < q2.size(); j++) {
                    q2[j]->wait++; // Increment wait time for each process
                }

                // Update waiting times for processes in FCFS queue
                for (int j = 0; j < fcfs.size(); j++) {
                    fcfs[j]->wait+=1; // Increment wait time for each process
                }

                // Check for processes in I/O and increment their counters
                for (int k = 0; k < deviceQueue.size(); k++) {
                    deviceQueue[k]->iocounter++; // Increment I/O counter

                    // If I/O is complete, move back to the ready queue
                    if (deviceQueue[k]->iocounter >= deviceQueue[k]->ioburst) {
                        deviceQueue[k]->ioburst = 0;
                        deviceQueue[k]->iocounter = 0; // Reset I/O counter
                        if (deviceQueue[k]->Queue == 1)
                            highPriority.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 2)
                            q2.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 3)
                            fcfs.push_back(deviceQueue[k]); // Move to ready queue
                        deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
                        k--; // Adjust index due to removal
                    }
                }

                // Break if there are processes in high priority queue
                if (!highPriority.empty())
                    break;
            }

            // If highPriority is not empty, move process from collect to low priority
            if (!highPriority.empty()) {
                if (!collect.empty()) {
                    // Increment the burst time of the front process by 'count' before pushing
                    collect.front()->data[0] += cpuburst - p->counter; // Adjust the burst time
                    collect.front()->tqsave = cpuburst - p->counter; // Save the remaining burst time
                    q2.push_back(collect.front()); // Push the process to low priority queue
                    collect.pop(); // Remove it from the collect queue
                    continue; // Continue to the next iteration of the loop
                } else {
                    p->data.insert(p->data.begin(), cpuburst - p->counter); // Insert remaining burst time
                    p->tqsave = cpuburst - p->counter; // Save the remaining burst time
                    p->counter = 0; // Reset counter
                    q2.push_back(p); // Push to low priority queue
                    continue; // Continue to the next iteration of the loop
                }
            } else {
                p->counter = 0; // Reset counter if no high priority processes
            }

            // If there are processes to move from collect to FCFS
            if (!collect.empty()) {
                fcfs.push_back(collect.front()); // Move from collect to fcfs
                collect.pop(); // Pop the front of collect
            }
            else {
                // Handle I/O burst for the current process
                if (!p->data.empty()) {
                    p->ioburst = p->data[0]; // Set the I/O burst
                    p->totalio += p->ioburst; // Accumulate I/O time
                    deviceQueue.push_back(p); // Move to I/O
                    p->data.erase(p->data.begin()); // Remove the used burst time

                    cout << "\nCompleted Execution= NO" << endl;
                } else {
                    p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
                    numprocesscomplete++; // Increment completed process count
                    cout << "\nCompleted Execution= YES" << endl;
                }
                cout << "\n=====================================================\n\n\n";
            }

////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
```

```
        } else if (highPriority.empty() && q2.empty() && !fcfs.empty()) {
            Process* p = fcfs.front(); // Get the next process from FCFS
            fcfs.pop_front();


            if (p->tqsave != 0) { // If there's a saved time quantum
                cpuburst = p->tqsave; // Set CPU burst time to saved time quantum
                p->data.erase(p->data.begin()); // Remove the burst time
                p->tqsave = 0; // Reset saved time quantum


            } else {
                cpuburst = p->data[0]; // Get the CPU burst time
                p->totalcpu += cpuburst; // Accumulate total CPU time
                p->data.erase(p->data.begin()); // Remove the used burst time

            }

            // Print execution details for the current process
            printexecution(p);
            for (int i = 0; i < cpuburst; i++) {
                totalallcpu++; // Increment total CPU time
                totaltime++; // Increment total time
                p->counter++; // Increment counter

                // Update waiting times for processes in FCFS queue
                for (int j = 0; j < fcfs.size(); j++) {
                    fcfs[j]->wait +=1 ;
                }

                // Check for processes in I/O and increment their counters
                for (int k = 0; k < deviceQueue.size(); k++) {
                    deviceQueue[k]->iocounter++; // Increment I/O counter

                    // If I/O is complete, move back to the ready queue
                    if (deviceQueue[k]->iocounter >= deviceQueue[k]->ioburst) {
                        deviceQueue[k]->ioburst = 0;
                        deviceQueue[k]->iocounter = 0; // Reset I/O counter
                       if (deviceQueue[k]->Queue == 1)
                           highPriority.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 2)
                           q2.push_back(deviceQueue[k]); // Move to ready queue
                        if (deviceQueue[k]->Queue == 3)
                           fcfs.push_back(deviceQueue[k]); // Move to ready queue
                        deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
                        k--; // Adjust index due to removal
                    }
                }
                // Break if there are processes in high priority queue
                if (!highPriority.empty() || !q2.empty())
                    break;
            }

            if (!highPriority.empty()|| !q2.empty()) {
                p->data.insert(p->data.begin(), cpuburst - p->counter); // Insert remaining burst time
                p->tqsave = cpuburst - p->counter; // Save remaining burst time
                p->counter = 0; // Reset counter
                fcfs.push_back(p); // Push to fcfs queue
                continue; // Continue to next iteration
            }
            else {
                p->counter = 0; // Reset counter if no high priority processes
            }

            // Handle I/O burst for the current process

            if (!p->data.empty()) {
                p->ioburst = p->data[0]; // Set the I/O burst
                p->totalio += p->ioburst; // Accumulate I/O time
                deviceQueue.push_back(p); // Move to I/O
                p->data.erase(p->data.begin()); // Remove the used burst time

                cout << "\nCompleted Execution= NO" << endl;
            } else {
                p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
                numprocesscomplete++; // Increment completed process count
                cout << "\nCompleted Execution= YES" << endl;
            }
            cout << "\n-----------------------------------------------------\n\n\n";
        }
```

```

        else if (q2.empty() && fcfs.empty() && highPriority.empty() && !deviceQueue.empty()) {
            // Handles CPU idle incrementing Processes in I/O burst counters
            totaltime++; // Increment total time once per iteration
            // Iterate through the device queue and update I/O counters
            for (int n = 0; n < deviceQueue.size(); n++) {
                deviceQueue[n]->iocounter++; // Increment I/O counter for each process

                // Check if the I/O is complete
                if (deviceQueue[n]->iocounter >= deviceQueue[n]->ioburst) {
                    // Move the completed process back to the ready queue
                    deviceQueue[n]->ioburst = 0;
                    deviceQueue[n]->iocounter = 0;
                    if (deviceQueue[n]->Queue == 1)
                            highPriority.push_back(deviceQueue[n]); // Move to ready queue
                    if (deviceQueue[n]->Queue == 2)
                            q2.push_back(deviceQueue[n]); // Move to ready queue
                    if (deviceQueue[n]->Queue == 3)
                            fcfs.push_back(deviceQueue[n]); // Move to ready queue
                    deviceQueue.erase(deviceQueue.begin() + n); // Remove from device queue
                    n--; // Adjust index because we removed the current element
                }
            }
        }
    }

    printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results for all processes
    return 0; // Return success status
}

```