

CPU Scheduler

Shahed Ahmed
10-26-2024

Introduction

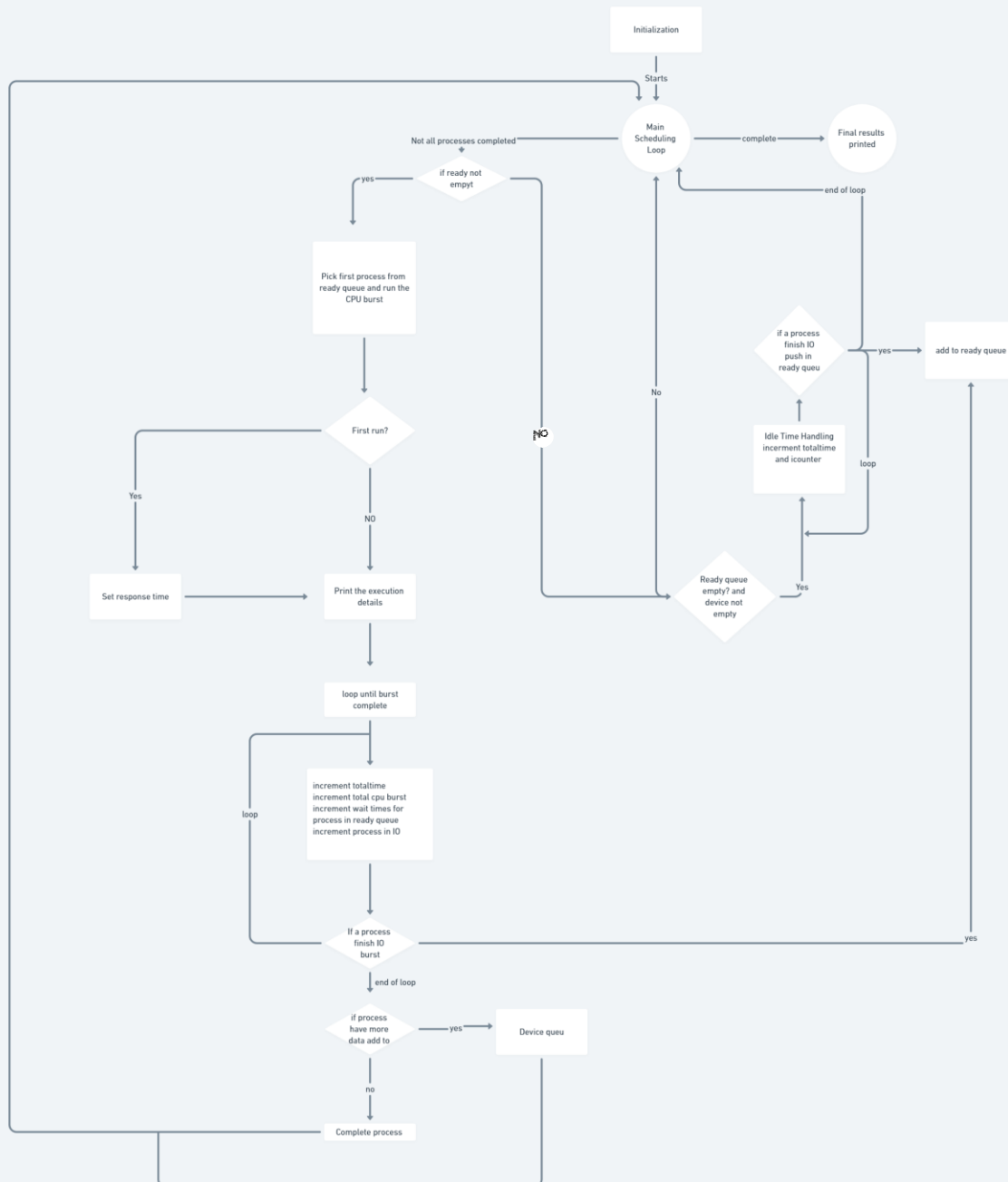
This project focuses on exploring different CPU scheduling algorithms by implementing and simulating three common approaches: First-Come-First-Serve (FCFS), Shortest Job First (SJF), and Multilevel Feedback Queue (MLFQ). CPU scheduling is a key function of operating systems, determining how processes 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

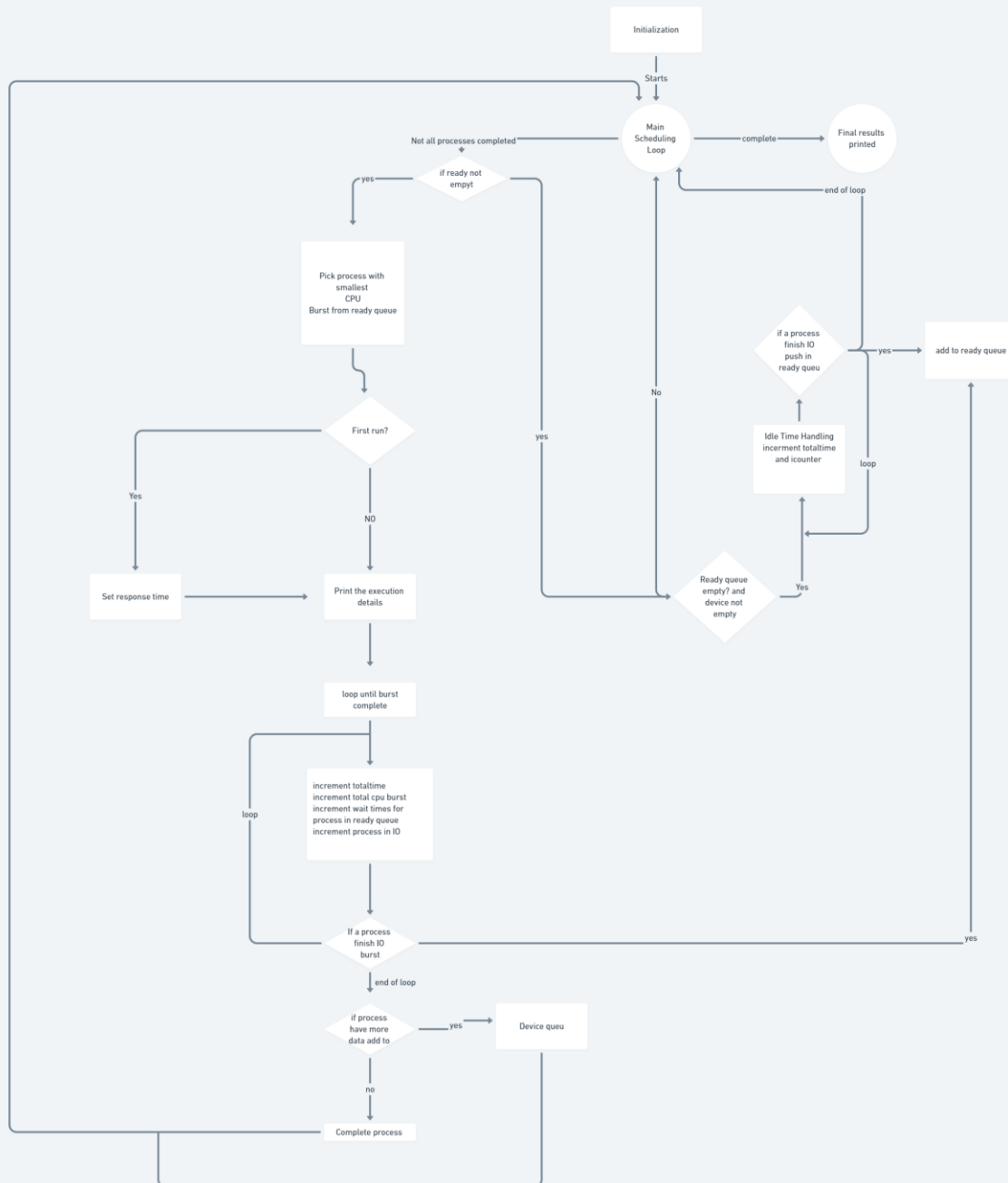
Introduction.....	1
Flow Chart	3
First Come First Serve	3
Short Job First.....	4
Multilevel Feedback Queue	5
Discussion and Tables	6
First Come First Serve	6
Shortest Job First.....	6
Multilevel Feedback Queue	7
Compare results SJF, FCFS, MLFQ	8
Sample of dynamic execution.....	9
FCFS	9
SJF.....	10
MLFQ	11
Results.....	12
FSFC	12
SJF.....	12
MLFQ	13
Partial source code	14
FSCF	14
SJF.....	16
MLFQ	18

Flow Chart

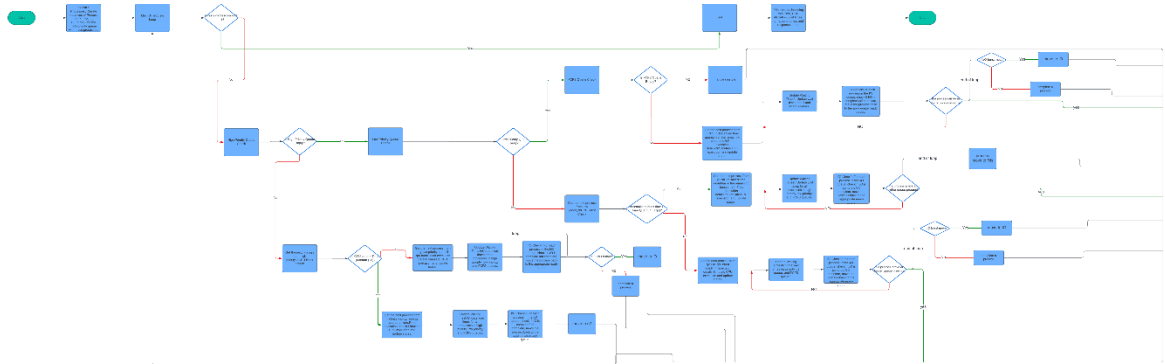
First Come First Serve



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	SJF	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

```

81  int main() {
82      Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
83      Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
84      Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
85      Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
86      Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
87      Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
88      Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
89      Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});
90
91      readyqueue.push(&p1);
92      readyqueue.push(&p2);
93      readyqueue.push(&p3);
94      readyqueue.push(&p4);
95      readyqueue.push(&p5);
96      readyqueue.push(&p6);
97      readyqueue.push(&p7);
98      readyqueue.push(&p8);
99
100     int cpuburst = 0;
101     int numprocesscomplete = 0;
102
103     // Main scheduling loop
104     while (numprocesscomplete != 8) {
105         if (!readyqueue.empty()) {
106             Process* p = readyqueue.front(); // Get the next process
107             readyqueue.pop();
108
109             // First run logic
110             if (p->firstrun) {
111                 p->responsetime = p->wait; // Capture the response time when first run
112                 p->firstrun = false;
113             }
114
115             cpuburst = p->data[0]; // Get the CPU burst time
116             p->totalcpu += cpuburst; // Accumulate total CPU time
117             p->data.erase(p->data.begin()); // Remove the used burst time
118
119             printexecution(p); // Print the execution details
120
121             // Execute CPU burst
122             for (int i = 0; i < cpuburst; i++) {
123                 totalallcpu++; // Increment total CPU time
124                 totaltime++; // Increment total time
125
126                 for (int j = 0; j < readyqueue.size(); j++) {
127                     Process* temp = readyqueue.front();
128                     readyqueue.pop();
129                     temp->wait++; // Increment wait time
130                     readyqueue.push(temp); // Push back
131                 }
132
133                 // Check for processes in I/O
134                 for (int k = 0; k < deviceQueue.size(); k++) {
135                     deviceQueue[k]->iocounter++; // Increment I/O counter
136
137                     // If I/O is complete, move back to the ready queue
138                     if (deviceQueue[k]->iocounter == deviceQueue[k]->ioburst) {
139                         deviceQueue[k]->ioburst = 0;
140                         deviceQueue[k]->iocounter = 0; // Reset I/O counter
141                         readyqueue.push(deviceQueue[k]); // Push to ready queue
142                         deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
143                         k--; // Adjust index due to removal
144                     }
145                 }
146             }
147             numprocesscomplete++;
148         }
149     }
150
151     return 0;
152 }
```

```

148
149 // Handle I/O burst
150 if (!p->data.empty()) {

151     p->ioburst = p->data[0]; // Set the I/O burst
152     p->totalio += p->ioburst; // Accumulate I/O time
153     p->data.erase(p->data.begin()); // Remove the used burst time
154     deviceQueue.push_back(p); // Move to I/O
155     cout << "\nCompleted Execution= NO" << endl;
156 } else {
157     p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
158     numprocesscomplete++; // Increment completed process count
159     cout << "\nCompleted Execution= YES" << endl;
160 }
161 cout << "\n-----\n\n";
162 }
163
164 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
        }
    }
}

181 }
182
183 printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results
184 return 0;
185 }
186

```


SJF

```

80 int main() {
81     Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
82     Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
83     Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
84     Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
85     Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
86     Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 8, 18, 12, 24, 15, 30, 8});
87     Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
88     Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});
89
90     readyqueue.push_back(&p1);
91     readyqueue.push_back(&p2);
92     readyqueue.push_back(&p3);
93     readyqueue.push_back(&p4);
94     readyqueue.push_back(&p5);
95     readyqueue.push_back(&p6);
96     readyqueue.push_back(&p7);
97     readyqueue.push_back(&p8);
98
99     int cpuburst = 0;
100    int numprocesscomplete = 0;
101
102    // Main scheduling loop
103    while (numprocesscomplete != 8) {
104        if (!readyqueue.empty()) {
105
106            // get the process with the smallest cpu burst
107            auto min it = min_element(readyqueue.begin(), readyqueue.end(), [](Process* a,
Process* b) {
108                return a->data.front() < b->data.front();
109            });
110
111            Process* p = *min it; // Get the process with the smallest CPU burst
112            readyqueue.erase(min it); // Remove it from the ready queue
113
114            // First run logic
115            if (p->firstrun) {
116                p->responsetime = p->wait; // Capture the response time when first run
117                p->firstrun = false;
118            }
119
120            cpuburst = p->data[0]; // Get the CPU burst time
121            p->totalcpu += cpuburst; // Accumulate total CPU time
122            p->data.erase(p->data.begin()); // Remove the used burst time
123
124            printexecution(p); // Print the execution details
125

```

```

126         // Execute CPU burst
127         for (int i = 0; i < cpuburst; i++) {
128             totalallcpu++; // Increment total CPU time
129             totaltime++; // Increment total time
130
131             for (int j = 0; j < readyqueue.size(); j++) {
132                 Process* temp = readyqueue[j];
133                 temp->wait++; // Increment wait time
134             }
135
136             // Check for processes in I/O
137             for (int k = 0; k < deviceQueue.size(); k++) {
138                 deviceQueue[k]->iocounter++; // Increment I/O counter
139             }
140
141             // If I/O is complete, move back to the ready queue
142             if (deviceQueue[k]->iocounter == deviceQueue[k]->ioburst) {
143                 deviceQueue[k]->ioburst = 0;
144                 deviceQueue[k]->iocounter = 0; // Reset I/O counter
145                 readyqueue.push_back(deviceQueue[k]); // Push to ready queue
146                 deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
147                 k--; // Adjust index due to removal
148             }
149         }
150     }
151
152     // Handle I/O burst
153     if (!p->data.empty()) {
154         p->ioburst = p->data[0]; // Set the I/O burst
155         p->totalio += p->ioburst; // Accumulate I/O time
156         p->data.erase(p->data.begin()); // Remove the used burst time
157         deviceQueue.push_back(p); // Move to I/O
158         cout << "\nCompleted Execution= NO" << endl;
159     } else {
160         p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
161         numprocesscomplete++; // Increment completed process count
162         cout << "\nCompleted Execution= YES" << endl;
163     }
164     cout << "\n-----\n\n";
165 }
166
167 else if (readyqueue.empty() && !deviceQueue.empty()) { // Handles CPU Idle incrementing
    // Processes in I/O Burst counters
168     totaltime++; // Increment total time once per iteration
169     // Iterate through the device queue and update I/O counters
170     for (int n = 0; n < deviceQueue.size(); n++) {
171         deviceQueue[n]->iocounter++; // Increment I/O counter for each process
172
173         // Check if the I/O is complete
174         if (deviceQueue[n]->iocounter >= deviceQueue[n]->ioburst) {
175             // Move the completed process back to the ready queue
176             deviceQueue[n]->ioburst = 0;
177             deviceQueue[n]->iocounter = 0;
178             readyqueue.push_back(deviceQueue[n]);
179             deviceQueue.erase(deviceQueue.begin() + n); // Remove from device queue
180             n--; // Adjust index because we removed the current element
181         }
182     }
183 }
184
185 printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results
186 return 0;
187
188 }
189

```

MLFQ

```

116 int main() {
117     int Tq = 5; // Time quantum for high-priority processes
118     int Tq2 = 10; // Time quantum for low-priority processes
119
120     // Queue to transfer processes from one queue to another
121     queue<Process*> collect;
122
123     // Initialize processes with their respective burst times
124     Process p1("P1", {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
125     Process p2("P2", {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
126     Process p3("P3", {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
127     Process p4("P4", {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
128     Process p5("P5", {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
129     Process p6("P6", {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
130     Process p7("P7", {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
131     Process p8("P8", {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});
132
133     // Populate the high priority queue with the processes
134     highPriority.push_back(&p1);
135     highPriority.push_back(&p2);
136     highPriority.push_back(&p3);
137     highPriority.push_back(&p4);
138     highPriority.push_back(&p5);
139     highPriority.push_back(&p6);
140     highPriority.push_back(&p7);
141     highPriority.push_back(&p8);
142
143     int cpuburst = 0; // Variable to track the current CPU burst time

```

```

144     int numprocesscomplete = 0; // Counter for completed processes
145
146     // Main scheduling loop
147     while (numprocesscomplete != 8) { // Continue until all processes are complete
148
149         if (!highPriority.empty()) {
150             Process* p = highPriority.front(); // Get the next process from high priority
151
152             // Execute CPU burst
153             if (p->data[0] > Tq) { // If the burst time is greater than the time quantum
154                 p->data[0] -= Tq; // Reduce the burst time
155                 cpuburst = Tq; // Set the CPU burst time
156                 p->Queue = 2;
157                 collect.push(highPriority.front()); // Move process to collect queue
158                 highPriority.pop_front(); // Remove process from high priority
159             }
160
161             else { // If the burst time is less than or equal to the time quantum
162                 cpuburst = p->data[0]; // Set CPU burst time to remaining burst
163                 p->data.erase(p->data.begin()); // Remove the burst time
164                 highPriority.pop_front(); // Remove process from high priority
165             }
166
167             // First run logic to capture response time
168             if (p->firstrun) {
169                 p->responsetime = p->wait; // Capture the response time when first run
170                 p->firstrun = false; // Mark as first run completed
171             }
172
173             p->totalcpu += cpuburst; // Accumulate total CPU time
174
175             // Print execution details for the current process
176             printexecution(p);
177
178             // Simulate CPU execution for the duration of the burst
179             for (int i = 0; i < cpuburst; i++) {
180                 totalallcpu++; // Increment total CPU time
181                 totaltime++; // Increment total time
182
183                 // Update waiting times for processes in highPriority
184                 for (int j = 0; j < highPriority.size(); j++) {
185                     highPriority[j]->wait++; // Increment wait time for each process
186                 }
187
188                 // Update waiting times for processes in q2
189                 for (int j = 0; j < q2.size(); j++) {
190                     q2[j]->wait++; // Increment wait time for each process
191                 }
192
193                 // Update waiting times for processes in FCFS queue
194                 for (int j = 0; j < fcfs.size(); j++) {
195                     fcfs[j]->wait++; // Increment wait time for each process
196                 }
197
198                 // Check for processes in I/O and increment their counters
199                 for (int k = 0; k < deviceQueue.size(); k++) {
200                     deviceQueue[k]->iocounter++; // Increment I/O counter
201
202                     // If I/O is complete, move back to the ready queue
203                     if (deviceQueue[k]->iocounter >= deviceQueue[k]->ioburst) {
204                         deviceQueue[k]->ioburst = 0;
205                         deviceQueue[k]->iocounter = 0; // Reset I/O counter
206
207                         if (deviceQueue[k]->Queue == 1)
208                             highPriority.push_back(deviceQueue[k]); // Move to ready queue
209                         if (deviceQueue[k]->Queue == 2)
210                             q2.push_back(deviceQueue[k]); // Move to ready queue
211                         if (deviceQueue[k]->Queue == 3)
212                             fcfs.push_back(deviceQueue[k]); // Move to ready queue
213                         deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
214                         k--; // Adjust index due to removal
215                     }
216                 }
217             }
218         }

```

```

219
220 // If there are processes to move from collect to q2
221 if (!collect.empty()) {

222     q2.push_back(collect.front()); // Move from collect to q2
223     collect.pop(); // Pop the front of collect
224 } else {
225     // Handle I/O burst for current process
226     if (!p->data.empty()) {
227         p->ioburst = p->data[0]; // Set the I/O burst
228         p->totalio += p->ioburst; // Accumulate I/O time
229         deviceQueue.push_back(p); // Move to I/O
230         p->data.erase(p->data.begin()); // Remove the used burst time
231     }
232     cout << "\nCompleted Execution= NO" << endl;
233 } else {
234     p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround
235     numprocesscomplete++; // Increment completed process count
236     cout << "\nCompleted Execution= YES" << endl;
237 }
238 cout << "\n-----\n\n";
239 }
240
241 //////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
242 } else if (highPriority.empty() && !q2.empty()) {
243     Process* p = q2.front(); // Get the next process from low priority
244     q2.pop_front(); // Remove from low priority
245     // Execute CPU burst
246     if (p->data[0] + p->counter > Tq2) { // Check if the remaining burst time exceeds time
247         quantum
248         p->Queue = 3;
249         if (p->tqsave != 0) {
250             p->data[0] -= p->tqsave; // Use the saved time quantum
251             cpuburst = p->tqsave; // Set the CPU burst time
252             p->tqsave = 0; // Reset saved time quantum
253             p->counter = 0; // Reset counter
254             collect.push(p); // Move process to collect queue
255         }
256         else {
257             p->data[0] -= Tq2; // Use the full time quantum
258             cpuburst = Tq2; // Set the CPU burst time
259             p->totalcpu += cpuburst; // Accumulate total CPU time
260             collect.push(p); // Move process to collect queue
261         }
262     }
263 }
264 }
265 else { // If the remaining burst time is less than or equal to the time quantum
266     p->Queue = 2;
267     if (p->tqsave != 0) {
268         cpuburst = p->tqsave; // Set CPU burst time to saved time quantum
269         p->data.erase(p->data.begin()); // Remove the burst time
270         p->tqsave = 0; // Reset saved time quantum
271     }
272     else {
273         cpuburst = p->data[0]; // Set CPU burst time to remaining burst time
274         p->data.erase(p->data.begin()); // Remove the burst time
275     }
276 }
277 }
278 }
279 }

```

```

280
281 // Print execution details for the current low-priority process
282 printExecution(p);
283 // Execute CPU burst
284 for (int i = 0; i < cpuburst; i++) {
285     p->counter++; // Increment counter for CPU burst
286     totalCpu++; // Increment total CPU time
287     totalTime++; // Increment total time
288
289 // Update waiting times for processes in q1
290 for (int j = 0; j < q2.size(); j++) {
291     q2[j]->wait++; // Increment wait time for each process
292 }
293
294 // Update waiting times for processes in FCFS queue
295 for (int j = 0; j < fcfs.size(); j++) {
296
297     fcfs[j]->wait+=1; // Increment wait time for each process
298 }
299
300 // Check for processes in I/O and increment their counters
301 for (int k = 0; k < deviceQueue.size(); k++) {
302     deviceQueue[k]->iocounter++; // Increment I/O counter
303
304 // If I/O is complete, move back to the ready queue
305 if (deviceQueue[k]->iocounter == deviceQueue[k]->ioburst) {
306     deviceQueue[k]->ioburst = 0;
307     deviceQueue[k]->iocounter = 0; // Reset I/O counter
308     if (deviceQueue[k]->Queue == 1)
309         highPriority.push_back(deviceQueue[k]); // Move to ready queue
310     if (deviceQueue[k]->Queue == 2)
311         q2.push_back(deviceQueue[k]); // Move to ready queue
312     if (deviceQueue[k]->Queue == 3)
313         fcfs.push_back(deviceQueue[k]); // Move to ready queue
314     deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
315     k--; // Adjust index due to removal
316 }
317
318 // Break if there are processes in high priority queue
319 if (!highPriority.empty())
320     break;
321 }
322
323 // If highPriority is not empty, move process from collect to low priority
324 if (!highPriority.empty()) {
325     if (!collect.empty()) {
326         // Increment the burst time of the front process by 'count' before pushing
327         collect.front()->data[0] += cpuburst - p->counter; // Adjust the burst time
328         collect.front()->tqsave = cpuburst - p->counter; // Save the remaining burst time
329         q2.push_back(collect.front()); // Push the process to low priority queue
330         collect.pop(); // Remove it from the collect queue
331         continue; // Continue to the next iteration of the loop
332     } else {
333         p->data.insert(p->data.begin(), cpuburst - p->counter); // Insert remaining
334         p->tqsave = cpuburst - p->counter; // Save the remaining burst time
335         p->counter = 0; // Reset counter
336         q2.push_back(p); // Push to low priority queue
337         continue; // Continue to the next iteration of the loop
338     }
339 } else {
340     p->counter = 0; // Reset counter if no high priority processes
341 }
342
343 // If there are processes to move from collect to FCFS
344 if (!collect.empty()) {
345     fcfs.push_back(collect.front()); // Move from collect to q1
346     collect.pop(); // Pop the front of collect
347 }
348 } else {
349     // Handle I/O burst for the current process
350     if (!p->data.empty()) {
351         p->ioburst = p->data[0]; // Set the I/O burst
352         p->totalio += p->ioburst; // Accumulate I/O time
353         deviceQueue.push_back(p); // Move to I/O
354         p->data.erase(p->data.begin()); // Remove the used burst time
355
356         cout << "\nCompleted Execution= NO" << endl;
357     } else {
358         p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround
359         numprocesscomplete++; // Increment completed process count
360         cout << "\nCompleted Execution= YES" << endl;
361     }
362     cout << "\n===== \n\n";
363 }
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565

```

```

366         } else if (highPriority.empty() && q2.empty() && !fcfs.empty()) {
367             Process* p = fcfs.front(); // Get the next process from FCFS
368             fcfs.pop_front();
369
370
371             if (p->tqsave != 0) { // If there's a saved time quantum
372                 cpuburst = p->tqsave; // Set CPU burst time to saved time quantum
373                 p->data.erase(p->data.begin()); // Remove the burst time
374                 p->tqsave = 0; // Reset saved time quantum
375
376             } else {
377                 cpuburst = p->data[0]; // Get the CPU burst time
378                 p->totalcpu += cpuburst; // Accumulate total CPU time
379                 p->data.erase(p->data.begin()); // Remove the used burst time
380
381             }
382
383             // Print execution details for the current process
384             printexecution(p);
385
386             for (int i = 0; i < cpuburst; i++) {
387                 totalallcpu++; // Increment total CPU time
388                 totaltime++; // Increment total time
389                 p->counter++; // Increment counter
390
391                 // Update waiting times for processes in FCFS queue
392                 for (int j = 0; j < fcfs.size(); j++) {
393                     fcfs[j]->wait += 1;
394                 }
395
396                 // Check for processes in I/O and increment their counters
397                 for (int k = 0; k < deviceQueue.size(); k++) {
398                     deviceQueue[k]->iocounter++; // Increment I/O counter
399
400                     // If I/O is complete, move back to the ready queue
401                     if (deviceQueue[k]->iocounter >= deviceQueue[k]->ioburst) {
402                         deviceQueue[k]->ioburst = 0; // Reset I/O counter
403                         if (deviceQueue[k]->Queue == 1)
404                             highPriority.push_back(deviceQueue[k]); // Move to ready queue
405                         if (deviceQueue[k]->Queue == 2)
406                             q2.push_back(deviceQueue[k]); // Move to ready queue
407                         if (deviceQueue[k]->Queue == 3)
408                             fcfs.push_back(deviceQueue[k]); // Move to ready queue
409                         deviceQueue.erase(deviceQueue.begin() + k); // Remove from I/O
410                         k--; // Adjust index due to removal
411                     }
412                 }
413                 // Break if there are processes in high priority queue
414                 if (!highPriority.empty() || !q2.empty())
415                     break;
416             }
417
418             if (!highPriority.empty() || !q2.empty()) {
419                 p->data.insert(p->data.begin(), cpuburst - p->counter); // Insert remaining burst
420
421                 p->tqsave = cpuburst - p->counter; // Save remaining burst time
422                 p->counter = 0; // Reset counter
423                 fcfs.push_back(p); // Push to fcfs queue
424                 continue; // Continue to next iteration
425             }
426             else {
427                 p->counter = 0; // Reset counter if no high priority processes
428             }
429
430             // Handle I/O burst for the current process
431             if (!p->data.empty()) {
432                 p->ioburst = p->data[0]; // Set the I/O burst
433                 p->totalio += p->ioburst; // Accumulate I/O time
434                 deviceQueue.push_back(p); // Move to I/O
435                 p->data.erase(p->data.begin()); // Remove the used burst time
436
437                 cout << "\nCompleted Execution= NO" << endl;
438             }
439             else {
440                 p->turnaroundtime = p->totalcpu + p->totalio + p->wait; // Calculate turnaround time
441                 numprocesscomplete++; // Increment completed process count
442                 cout << "\nCompleted Execution= YES" << endl;
443             }
444             cout << "\n-----\n\n";

```

```

445
446     else if (q2.empty() && fcfs.empty() && highPriority.empty() && !deviceQueue.empty()) {

447         // Handles CPU idle incrementing Processes in I/O burst counters
448         totalTime++; // Increment total time once per iteration
449         // Iterate through the device queue and update I/O counters
450         for (int n = 0; n < deviceQueue.size(); n++) {
451             deviceQueue[n]->iocounter++; // Increment I/O counter for each process
452
453             // Check if the I/O is complete
454             if (deviceQueue[n]->iocounter >= deviceQueue[n]->ioburst) {
455                 // Move the completed process back to the ready queue
456                 deviceQueue[n]->ioburst = 0;
457                 deviceQueue[n]->iocounter = 0;
458                 if (deviceQueue[n]->Queue == 1)
459                     highPriority.push_back(deviceQueue[n]); // Move to ready queue
460                 if (deviceQueue[n]->Queue == 2)
461                     q2.push_back(deviceQueue[n]); // Move to ready queue
462                 if (deviceQueue[n]->Queue == 3)
463                     fcfs.push_back(deviceQueue[n]); // Move to ready queue
464                 deviceQueue.erase(deviceQueue.begin() + n); // Remove from device queue
465                 n--; // Adjust index because we removed the current element
466             }
467         }
468     }
469 }
470
471 printresults(p1, p2, p3, p4, p5, p6, p7, p8); // Print final results for all processes
472 return 0; // Return success status
473 }
474

```