Programing Assignment CPU Scheduler

Luca Lombardo

3/10/2025

COP 4610- Computer Operating Systems

Spring 2025

Prof. Borko Furht

Florida Atlantic University

Table of Contents

Introduction

In this project, I explore CPU scheduling by implementing and analyzing three scheduling algorithms:

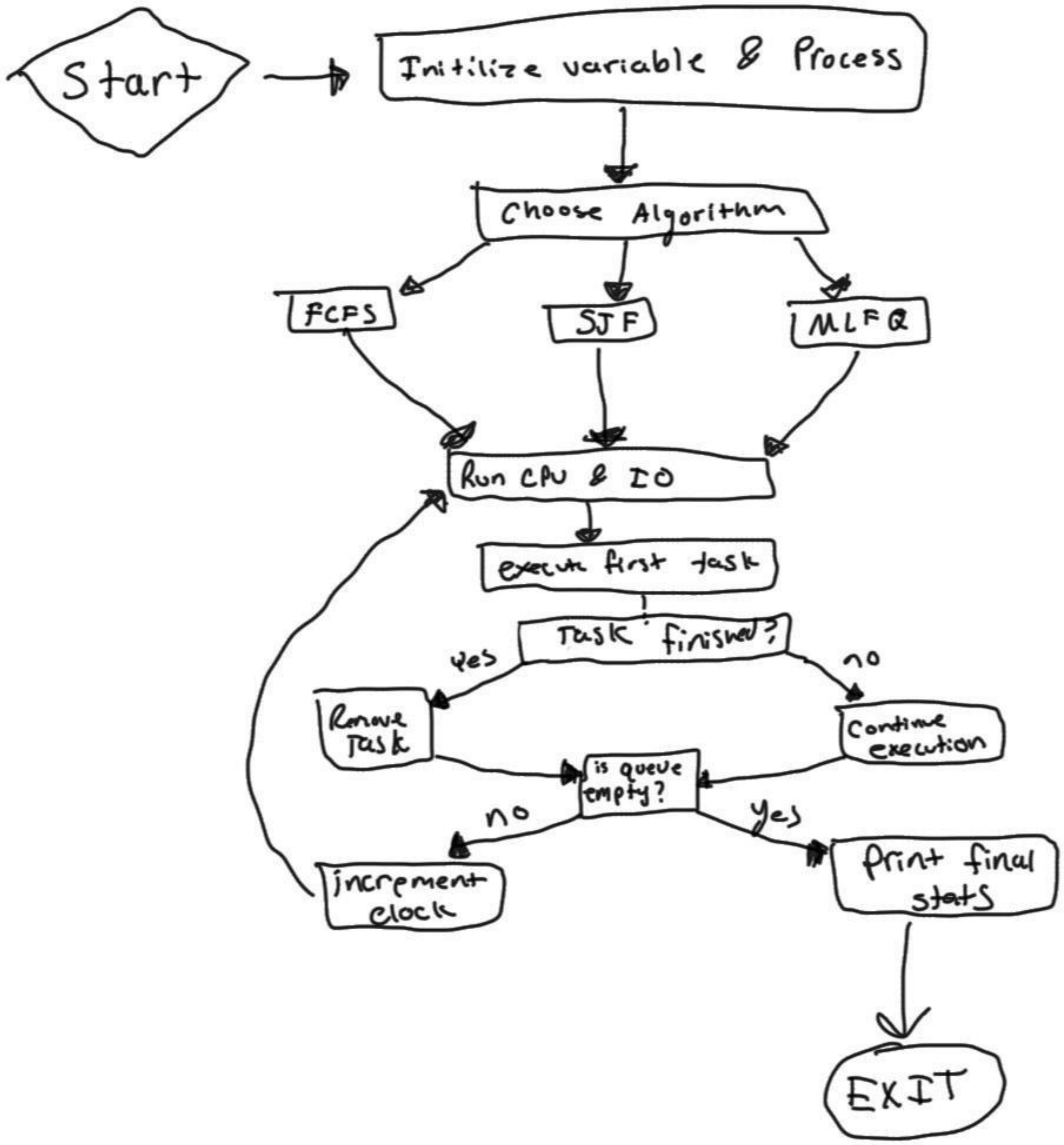
- First-Come, First-Served (FCFS)
- Shortest Job First (SJF)
- Multilevel Feedback Queue (MLFQ).

The goal is to understand how different scheduling methods affect process execution time, waiting time, and CPU utilization.

The simulation will run on a set of eight processes, each with predefined CPU bursts and I/O times. We will compare the performance of each algorithm based on key metrics, such as turnaround time, waiting time, response time, and CPU utilization.

By completing this project, I gained direct experience with operating system scheduling and also learned about how different strategies impact overall system efficiency.

Flow Chart


Start
Initilize variable & Process
Choose Algorithm
FCFS
SJF
MLFQ
Run CPU & IO
execute first task
Task finished?
yes
no
Remove Task
Continue execution
is queue empty?
no
yes
increment clock
print final stats
EXIT

Simulation Results

	SJF	FCFS	MLFQ
CPU UTILIZATION	**84 %**	**85 %**	**83%**
Avg Waiting time (Tw)	**120 %**	**185 %**	**191 %**
Avg Turnaround time (Ttr)	**457 %**	**521 %**	**527 %**
Avg Response time (Tr)	**27 %**	**24%**	**21 %**

SJF	**CPU U = 84%**		
	TW	**TTR**	**TR**
P1	21	246	11
P2	16	443	3
P3	267	659	16
P4	15	499	0
P5	259	568	109
P6	81	296	24
P7	203	531	47
P8	105	414	7
AVG	**120**	**457**	**27**

FCFS	**CPU U = 85%**		
	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**	**24**

MLFQ	**CPU U = 83%**		
	TW	**TTR**	**TR**
P1	185	410	0
P2	155	582	5
P3	165	557	9
P4	183	667	17
P5	229	538	20
P6	249	463	30
P7	187	515	40
P8	175	484	50

AVG	**191**	**527**	**21**

Discussion

Using Object-Oriented Programming (OOP), creating classes to represent processes, tasks, and scheduling data. Using getters and setters to manage data access smoothly. By applying encapsulation, I kept process details securely within each class, and abstraction helped hide the complex scheduling algorithms. This made it easier for me to modify or expand the program without breaking other parts. Relying on vector functions like push_back, erase, and other functions to manage memory efficiently and navigate through data. These functions made it easier for me to visualize the process flow and develop a better solution.

The code simulates how a computer manages processes using three scheduling methods:

FCFS, SJF, and MLFQ.

Each process has tasks with separate times that the CPU processes before sending them to the I/O queue when needed.

First, the program sets up a list of processes and their tasks. Each process has data, such as burst time and waiting time which are then stored in a class ProcessData. These processes are then put through functions that calculate the results.

The three scheduling methods work like this:

- FCFS (First Come First Serve): Processes are managed in the order they arrive. The first process runs first, and each processes wait for the previous one to finish.

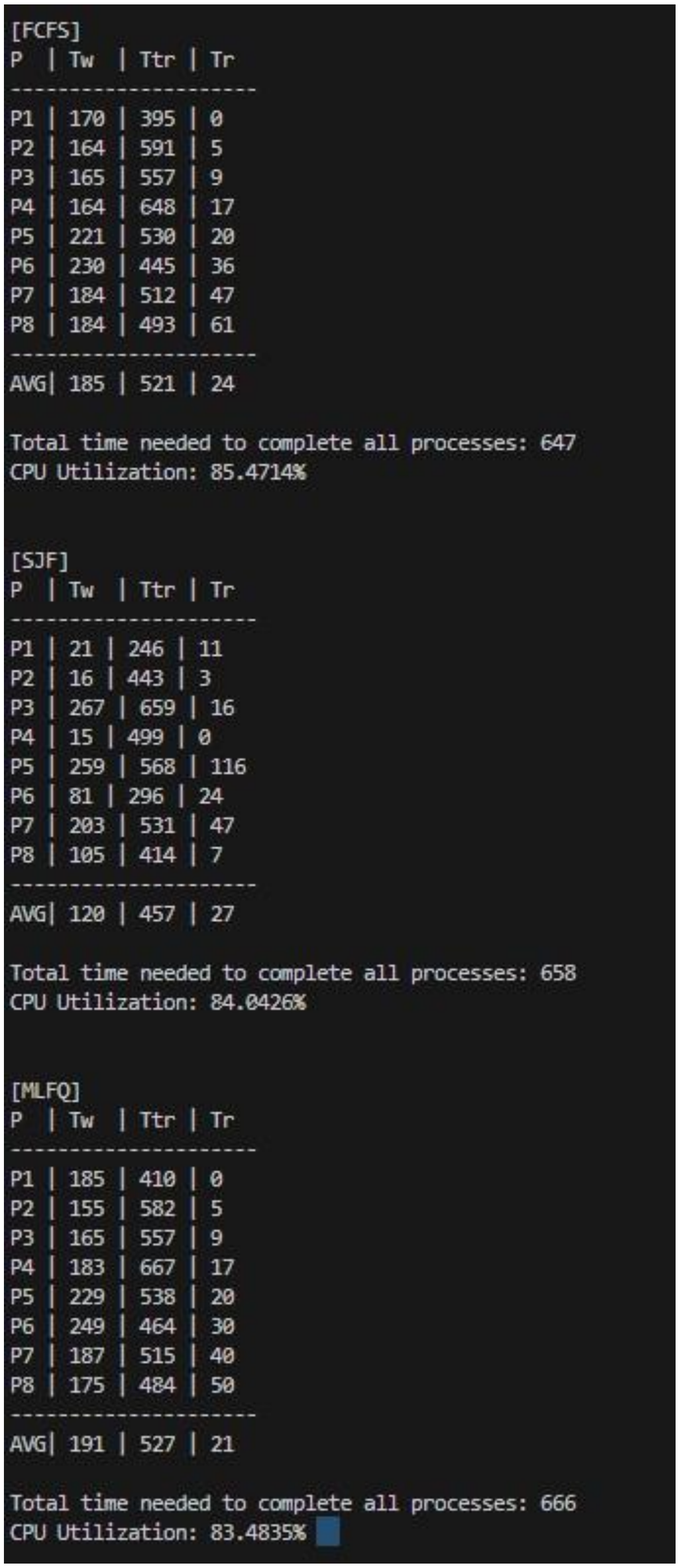
- SJF (Shortest Job First): The processes are sorted by shortest task next so The process with the shortest task runs next.

- MLFQ (Multi-Level Feedback Queue): This method uses multiple queues with different priorities. Processes start in a low-priority queue and move to a higher-priority queue if they don’t finish in their time slot.

During execution each task in a process runs for one unit of time. Once a task is done, the process either goes to the I/O queue or is marked as finished, and its stats are updated.

After the algorithm finishes, the program calculates and shows performance stats like waiting time, turnaround time, and CPU usage.

Dynamic Execution



```
[FCFS]
P   | 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 | 24

Total time needed to complete all processes: 647
CPU Utilization: 85.4714%


[SJF]
P   | Tw   | Ttr | Tr
---------------------
P1 | 21 | 246 | 11
P2 | 16 | 443 | 3
P3 | 267 | 659 | 16
P4 | 15 | 499 | 0
P5 | 259 | 568 | 116
P6 | 81 | 296 | 24
P7 | 203 | 531 | 47
P8 | 105 | 414 | 7
---------------------
AVG| 120 | 457 | 27

Total time needed to complete all processes: 658
CPU Utilization: 84.0426%


[MLFQ]
P   | Tw   | Ttr | Tr
---------------------
P1 | 185 | 410 | 0
P2 | 155 | 582 | 5
P3 | 165 | 557 | 9
P4 | 183 | 667 | 17
P5 | 229 | 538 | 20
P6 | 249 | 464 | 30
P7 | 187 | 515 | 40
P8 | 175 | 484 | 50
---------------------
AVG| 191 | 527 | 21

Total time needed to complete all processes: 666
CPU Utilization: 83.4835%
```

Source Code

```
// COP 4600 - Operating Systems
// Simulation Code

#include <iostream>
#include <vector>
#include <functional>
#include <string>

#define P 8  // Number of tasks in data set
#define TASKS 50 // Number of max possible tasks in each process
 using namespace
std;

// Enum for the algorithm types enum
AlgorithmType {
    FCFS,  // First Come First Serve
    SJF,   // Shortest Job First
    MLFQ   // Multi-Level Feedback Queue
};

// Class for process
statistics class ProcessData {
public:
    ProcessData() : AT(0), WT(0), BT(0), IOT(0), CT(0), TTR(0), RT(-1) {}
     int getWT() const { return WT; }
int getBT() const { return BT; }
int getIoT() const { return IOT; }
int getCT() const { return CT; }
int getTTR() const { return TTR; }
int getRT() const { return RT; }

    void setWT(int value) { WT = value; }
void setBT(int value) { BT = value; }
void setIoT(int value) { IOT = value; }
void setCT(int value) { CT = value; }
void setTTR(int value) { TTR = value; }
void setRT(int value) { RT = value; }

private:
    int AT;  // Arrival time
```

```
   int WT;  // Waiting time
int BT;  // Burst time     int
IOT; // I/O time     int CT;
// Completion time
```

```
    int TTR; // Turn Around Time
int RT;  // Response Time
};

// Class for task
structure class Task {
public:
    Task(int t = 0) : time(t) {}     int
getTime() const { return time; }     void
setTime(int value) { time = value; }

private:
    int time;
};

// Class for process
structure class Process {
public:
    Process(int idx = 0) : index(idx), extraData(nullptr) {}
     int getIndex() const { return index; }
vector<Task*>& getTasks() { return tasks; }
vector<Task*>& getDone() { return done; }
void* getExData() const { return extraData; }
    ProcessData& getStats() { return stats; }
     void setExData(void* value) { extraData = value;
}     void setIndex(int value) { index = value; }

private:
    int index;                    // Process index
vector<Task*> tasks;        // Tasks for the process
vector<Task*> done;         // Completed tasks
void* extraData;            // Extra data for MLFQ
    ProcessData stats;          // Process statistics
};

// Class for MLFQ-specific data
class MLFQDefinition { public:
    MLFQDefinition(int tq = 0, int maxPriority = 0, int priority = 0)
```

```
        : tq(tq), maxPriority(maxPriority), priority(priority) {}
     int getMaxPriority() const { return maxPriority;
}     int getPriority() const { return priority; }
int getTq() const { return tq; }
     void setPriority(int value) { priority = value;
}

private:
    int tq;          // Time quantum
int maxPriority; // Maximum priority
int priority;    // Current priority
};

// Function prototypes void initilizeP(vector<Process*>& cpu, vector<Process*>&
io, Process processes[]); void initilizeMLFQ(vector<Process*>& cpu); void
runOperations(vector<Process*>& cpu, vector<Process*>& io); void
runFCFS(vector<Process*>& cpu, vector<Process*>& io); void
runSJF(vector<Process*>& cpu, vector<Process*>& io); void
runMLFQ(vector<Process*>& cpu, vector<Process*>& io); void completeTask(Process*
cur_P); void printProcess(const char* P_NAME, Process processes[]); bool
compPT(Process* first, Process* second); void sort(vector<Process*>& cpu, bool
(*compare)(Process*, Process*)); void combinedAlgorithm(Process* curProcess,
Task* curTask, vector<Process*>* cpu, vector<Process*>* io, AlgorithmType type);



// Universal variables int pClock = 0; //
Global clock for simulation int cpuBusyTime =
0; // Tracks CPU busy time

// Data set for processes and tasks int
pDataTable[P][TASKS] = {
    { 5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4 },
    { 4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8 },
    { 8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6 },
    { 3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3 },
    { 16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4 },
    { 11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8 },
    { 14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10 },
    { 4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6 }
```

```
};  int main() {
vector<Process*> cpu;
    vector<Process*> io;
    Process processes[P];

    // FCFS     initilizeP(cpu, io, processes); // Initialize
the processes     runFCFS(cpu, io); // Run the FCFS algorithm
printProcess("[FCFS]", processes);  //Print the process stats
    // SJF     initilizeP(cpu, io, processes);     sort(cpu, compPT); // Sort
the processes based on the time of the first task for SJF     runSJF(cpu, io);
printProcess("[SJF]", processes);
    // MLFQ     initilizeP(cpu, io, processes);     initilizeMLFQ(cpu); //
Initialize the MLFQ data stored in the processes class     runMLFQ(cpu, io);
printProcess("[MLFQ]", processes);
     return
0;
}

// Function to initialize processes so each process can be run to completion
void initilizeP(vector<Process*>& cpu, vector<Process*>& io, Process
processes[])
{
    cpu.clear();  // making sure cpu and io vectors are empty before
initilization
    io.clear();   // ------------------------------------------------------------
        pClock = 0;
cpuBusyTime = 0;

    for (int i = 0; i < P; i++) {
        Process p;
        p.getStats() = ProcessData();
        for (int j = 0; j < TASKS && pDataTable[i][j]; j++) {
Task* t = new Task(pDataTable[i][j]);
            p.getTasks().push_back(t);
        }
        p.setIndex(i);
        processes[i] = p;
cpu.push_back(&processes[i]);
    }
}

// Function to initialize MLFQ data so that the processes can be run void
initilizeMLFQ(vector<Process*>& cpu) {     for (auto& p : cpu)          p-
>setExData(new MLFQDefinition(5, 3, 1)); // Time quantum, maximum
priority, priority
}

// Unified function to handle both CPU and I/O operations
void runOperations(vector<Process*>& cpu, vector<Process*>& io) {

// CPU
    if (!cpu.empty()) {
        Process* curProcess = cpu.front();
        if (!curProcess->getTasks().empty()) { // If the process has tasks
Task* curTask = curProcess->getTasks().front(); // Get the first task`
curTask->setTime(curTask->getTime() - 1); // Decrease the time of the task
            curProcess->getStats().setBT(curProcess->getStats().getBT() + 1);
// Increase the burst time of the process             cpuBusyTime++;
             if (curProcess->getStats().getRT() < 0) { // If the response time
is not set                 curProcess->getStats().setRT(pClock); // Set the
response time to the current clock
            }

            for (size_t i = 1; i < cpu.size(); ++i) {                 cpu[i]-
>getStats().setWT(cpu[i]->getStats().getWT() + 1);
            }
        }
    }

    // I/O
    for (size_t i = 0; i < io.size();) {         Process* curProcess = io[i];
if (curProcess->getTasks().empty()) { // If the process has no tasks
io.erase(io.begin() + i); // Remove the process from the I/O queue
```

```
        continue;
    }

    Task* curTask = curProcess->getTasks().front(); // Get the first task
```

```
        curTask->setTime(curTask->getTime() - 1); // Decrease the time of the
task
        curProcess->getStats().setIoT(curProcess->getStats().getIoT() + 1); //
Increase the I/O time of the process
         if (curTask->getTime() == 0) { // If the task is completed
            curProcess->getTasks().erase(curProcess->getTasks().begin()); //
Remove the task from the process                  cpu.push_back(curProcess); //
Move the process to the CPU queue                  io.erase(io.begin() + i); //
Remove the process from the I/O queue
        } else {
            ++i;
        }
    }
}  void runFCFS(vector<Process*>& cpu, vector<Process*>&
io) {     while (!cpu.empty() || !io.empty()) {
runOperations(cpu, io);
         if (!cpu.empty() && !cpu.front()->getTasks().empty()) {// If the CPU is
not empty and the process [HAS] tasks              combinedAlgorithm(cpu.front(),
cpu.front()->getTasks().front(), &cpu,
&io, FCFS); //call function to run the algorithm
        }
         if (!cpu.empty() && cpu.front()->getTasks().empty()) { // If the CPU is
not empty and the process has [NO] tasks
            cpu.erase(cpu.begin()); // delete the process from the CPU queue
}

        ++pClock;
    }
}  void runSJF(vector<Process*>& cpu, vector<Process*>&
io) {     while (!cpu.empty() || !io.empty()) {
runOperations(cpu, io);
         if (!cpu.empty() && !cpu.front()->getTasks().empty())
{
            combinedAlgorithm(cpu.front(), cpu.front()->getTasks().front(), &cpu,
&io, SJF);            sort(cpu,
compPT);
        }
         if (!cpu.empty() && cpu.front()->getTasks().empty())
{              cpu.erase(cpu.begin());
        }

        ++pClock;
    }
}  void runMLFQ(vector<Process*>& cpu, vector<Process*>&
io) {     while (!cpu.empty() || !io.empty()) {
runOperations(cpu, io);
         if (!cpu.empty() && !cpu.front()->getTasks().empty()) {
combinedAlgorithm(cpu.front(), cpu.front()->getTasks().front(), &cpu,
&io, MLFQ);
        }
         if (!cpu.empty() && cpu.front()->getTasks().empty())
{               cpu.erase(cpu.begin());
        }

        ++pClock;
    }
}

// Function to complete the task and move it to the done list so that the next
task can be run
void completeTask(Process* cur_P) {
    if (!cur_P->getTasks().empty() && cur_P->getTasks().front()->getTime() == 0)
{         cur_P->getDone().push_back(cur_P-
>getTasks().front());         cur_P->getTasks().erase(cur_P-
>getTasks().begin());         cur_P-
>getStats().setCT(pClock);
    }
}

// Function to print process stats
void printProcess(const char* P_NAME, Process processes[])
{     ProcessData avg;     int totalTime = 0;
```

```
    cout << endl << P_NAME << endl;     cout
<< "P  | Tw  | Ttr | Tr" << endl;     cout
<< "---------------------" << endl;

    for (int i = 0; i < P; i++) {
        Process& p = processes[i];
        // Calculate Turnaround Time TTR for each process
        p.getStats().setTTR(p.getStats().getWT() + p.getStats().getBT() +
p.getStats().getIoT());

        // Update totalTime to the maximum completion time (CT) of all
processes         totalTime = max(totalTime, p.getStats().getCT());
        // Print process stats         cout <<
"P" << p.getIndex() + 1 << " | ";          cout
<< p.getStats().getWT() << " | ";          cout
<< p.getStats().getTTR() << " | ";          cout
<< p.getStats().getRT() << endl;

        // formulas for averages
avg.setWT(avg.getWT() + p.getStats().getWT());
avg.setTTR(avg.getTTR() + p.getStats().getTTR());
avg.setRT(avg.getRT() + p.getStats().getRT());     }

    // Calculate averages
avg.setWT(avg.getWT() / P);
avg.setTTR(avg.getTTR() / P);
avg.setRT(avg.getRT() / P);

    // Print averages
    cout << "---------------------" << endl;
    cout << "AVG| " << avg.getWT() << " | " << avg.getTTR() << " | " <<
avg.getRT() << endl;

    // Print total time and CPU utilization
    cout << endl << "Total time needed to complete all processes: " << totalTime
<< endl;

    // CPU Utilization     if (totalTime > 0) {         float cpuUtilization
= (static_cast<float>(cpuBusyTime) / totalTime) *
100;         cout << "CPU Utilization: " << cpuUtilization << "%" <<
endl;
```

[redacted]

```
    }      cout
<< endl;
}

// Comparison of processes based on the time of the first
task bool compPT(Process* first, Process* second) {     int
time1 = 0;     if (!first->getTasks().empty()) {
time1 = first->getTasks().front()->getTime();
    }     int time2 = 0;     if (!second-
>getTasks().empty()) {         time2 = second-
>getTasks().front()->getTime();
    }     return time1 <
time2;
}

// Function to sort the processes based on the comparison function
void sort(vector<Process*>& cpu, bool (*compare)(Process*, Process*))
{     for (size_t i = 0; i < cpu.size(); ++i) {         for (size_t j
= 0; j < cpu.size() - i - 1; ++j) {             if (compare(cpu[j +
1], cpu[j])) {                 Process* temp = cpu[j];
cpu[j] = cpu[j + 1];                 cpu[j + 1] = temp;
            }
        }
    }
}


// Function to implement the algorithms void combinedAlgorithm(Process*
curProcess, Task* curTask, vector<Process*>* cpu, vector<Process*>* io,
AlgorithmType type) {     if (curTask->getTime() == 0) {
completeTask(curProcess);
        io->push_back(curProcess); // Move the process to the I/O queue
cpu->erase(cpu->begin()); // Remove the process from the CPU queue
         if (type == SJF && !cpu->empty())
{
            sort(*cpu, compPT);  // calls a function that sorts the processes
based on the time of the first task
        }
return;
```

```
    }
  // MLFQ algorithm
if (type == MLFQ) {
        MLFQDefinition* exData = (MLFQDefinition*)(curProcess->getExData());
        // If the process has not reached the maximum priority and the burst
time is greater than the time quantum         if (exData->getPriority() <
exData->getMaxPriority() && curProcess-
>getStats().getBT() >= exData->getTq() * (exData->getPriority() + 1))
         {             exData->setPriority(exData->getPriority() + 1); //
Increase the priority              io->push_back(curProcess); // Move the
process to the I/O queue              cpu->erase(cpu->begin()); // Remove the
process from the CPU queue
        }
    }
}
```