Florida Atlantic University
Department of Computer & Electrical Engineering & Computer Science
Computer Operating Systems – COP4610

CPU Scheduling

Submitted by:
Joshua Cidoni-Walker
jcidoniwalke2017@fau.edu

Sunday, March 15, 2020

Table of Contents

Introduction

A computer system is only truly capable of concurrently running as many processes as there are processors (or cores). To achieve the illusion of process concurrency, computer operating systems utilize process scheduling algorithms to order process execution. The more statistically *efficient* such an algorithm is, the more optimal it becomes.

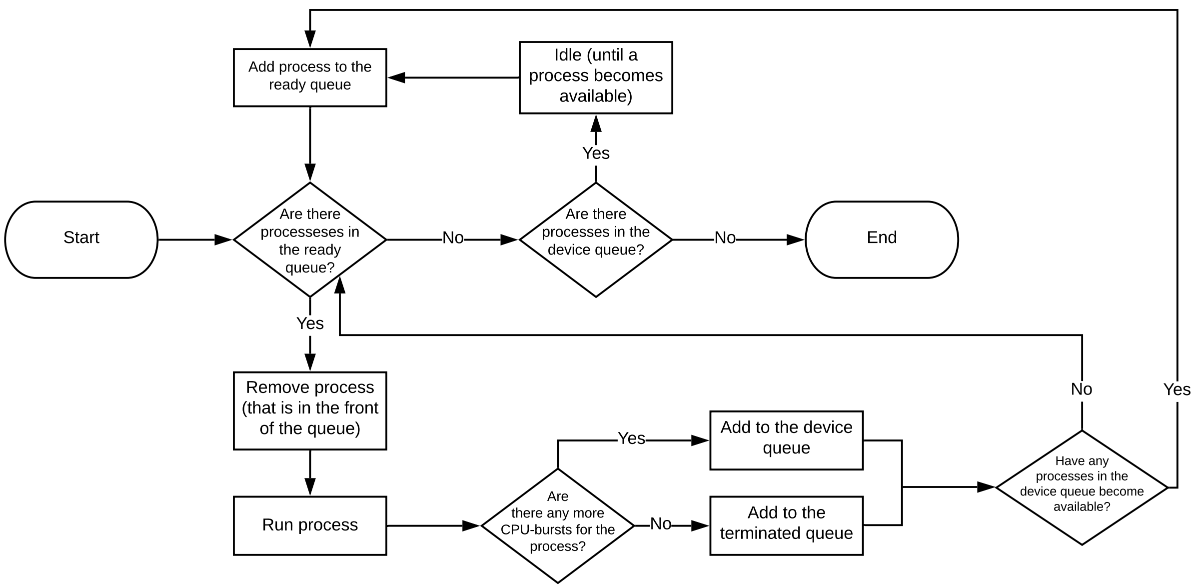
The illusion of process concurrency is achieved by running processes in fragments. These fragments are ran in an order such that multiple processes execute non-sequentially, rapidly, and frequently. By running processes in such a fashion, processes are not blocked by large, or hanging, processes which aids in the illusion of process concurrency.

The efficiency of a process scheduling algorithm is determined by its ability to maximize processor utilization and process throughout, and minimize average process wait, and response, time. That is, an algorithm should strive to keep the processor busy and maximize the amount of processes finished within a given time interval. The algorithm should also lessen the amount of time processes spend waiting to execute. The more it is able to achieve both principles, the more efficient, and thus optimal the algorithm becomes.

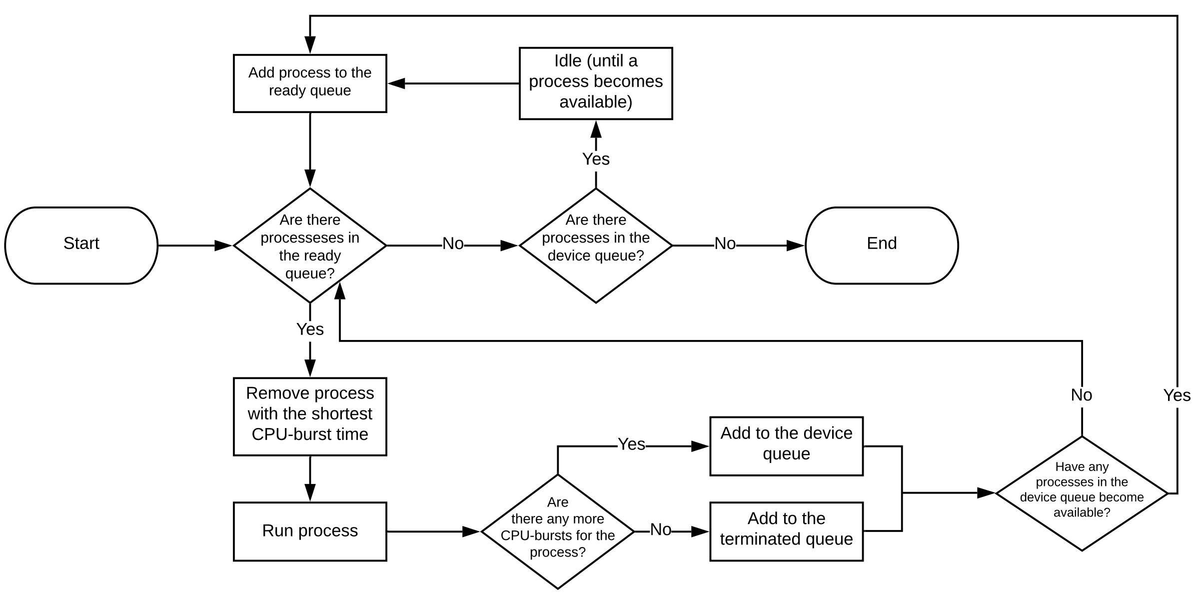
This paper statistically presents and analyzes the results of several process scheduling algorithms through the use of a simulation program.

Conceptual Diagrams

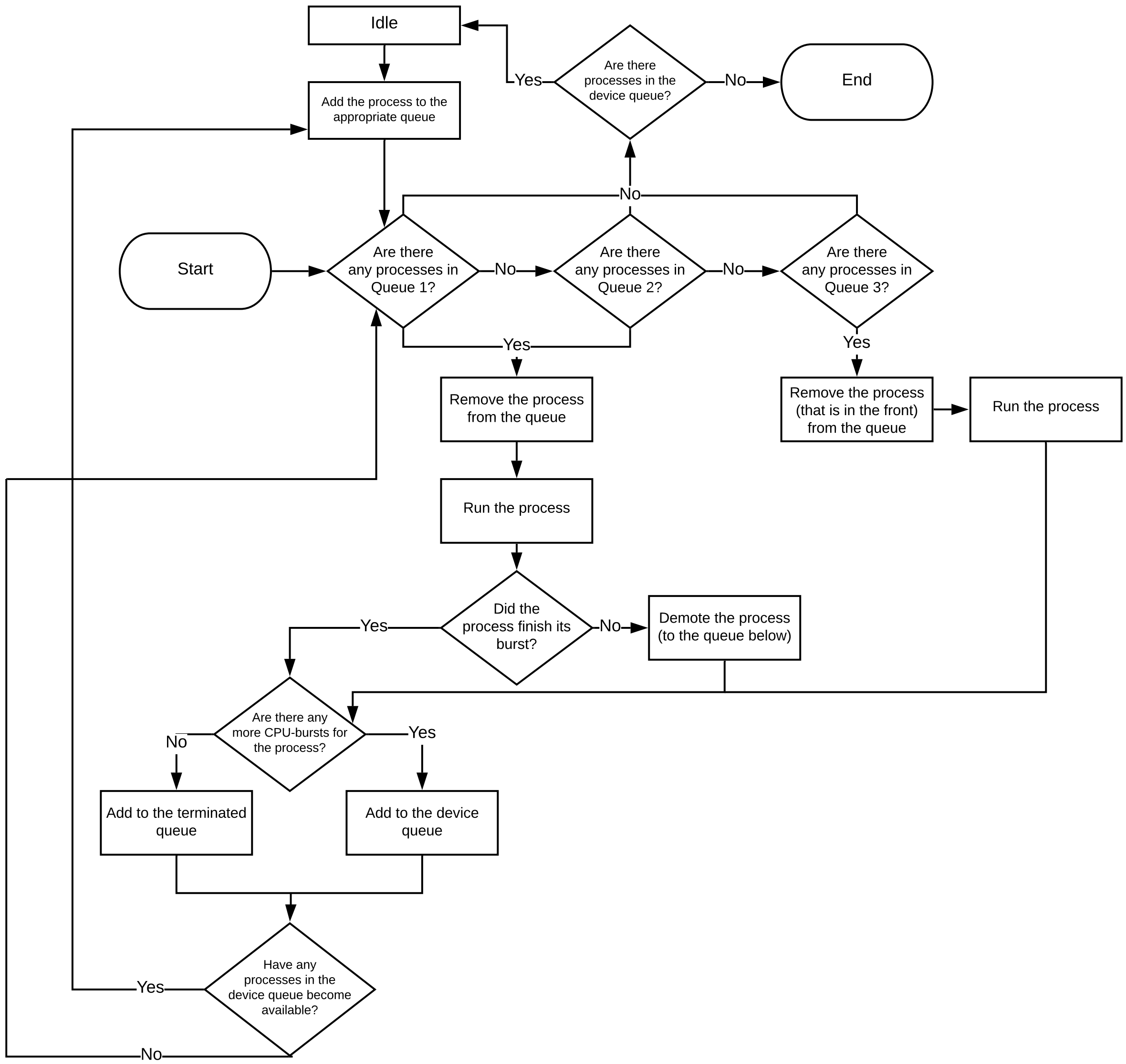
First Come, First Served



Shortest Job First



Multilevel Feedback Queue



Final Results and Discussion

First Come, First Served

This algorithm presents with a relatively large turn waiting time, large turnaround time, and a relatively above-average response time. The response time average is fully dependent on the lengths of the first processor burst.

Process	Tw	Ttr	Tr
1	170	395	0
2	164	591	5
3	165	557	9
4	164	648	17
5	221	530	20
6	230	445	36
7	184	512	47
8	184	493	61
	185.25	**521.38**	**24.38**
	CPU Utilization: 85.34%		

Shortest Job First

This algorithm presents with a relatively low waiting time, an extremely low turnaround time; yet, a rather large response time. This response time average is not only dependent on the lengths of the first processor burst (of each process); it is also dependent on the amount of processes (with relatively short processor bursts) that prevent larger processes from promptly running.

Process	Tw	Ttr	Tr
1	43	268	11
2	73	500	3
3	276	668	16
4	50	534	0
5	237	546	109

Process	Tw	Ttr	Tr
6	121	336	24
7	149	477	47
8	119	428	7
	133.50	**469.63**	**27.13**
	CPU Utilization: 82.78%		

Multilevel Feedback Queue

This algorithm presents with a relatively average waiting time and turnaround time; however, it also presents with an extremely low (relative) response time. The response time average remains fairly consistent; due to the nature of MLFQ (assuming there is a fixed time quantum for Queue 1).

Process	Tw	Ttr	Tr
1	46	266	0
2	161	573	5
3	222	599	9
4	86	565	14
5	283	577	17
6	182	382	22
7	240	553	27
8	119	413	32
	167.38	**491**	**15.75**
	CPU Utilization: 92.32%		

Comparison

- First Come, First Served presents with the largest *wait time,* largest *turnaround time*, and the second-largest (or second-least) *response time* averages. It also presents the second-largest (or second-least) processor *utilization*.

- Shortest Job First presents with the lowest *wait time*, the lowest *turnaround time*, and the largest *response time* averages. It also presents with the lowest processor *utilization*.

- Multilevel Feedback Queue (MLFQ) presents with the second-largest *wait time*, the second-largest *turnaround time*, and the lowest *response time* averages. It also presents with the largest processor *utilization*.

Average	First Come, First Served	Shortest Job First	Multilevel Feedback Queue
Wait Time (Tw)	185.25	133.5	167.38
Turnaround Time (Ttr)	521.37	469.63	491
Response Time (Tr)	24.37	27.13	15.75
CPU Utilization	85.34	82.78	92.32

Conclusion

Shortest Job First (SJF) yields the shortest wait time and turnaround time averages. Therefore, it is deemed the most optimal process scheduling algorithm*. Multilevel Feedback Queue (MLFQ) yielded the shortest response time, and the second shortest wait time and turnaround time averages. First Come, First Served (FCFS) provided the least optimal results, yielding the highest wait time and turnaround time averages.

* It is important to note that Shortest Job First (SJF) in itself is an idealistic algorithm. To implement SJF practically, prediction algorithms must be used to obtain future process execution time.

Appendix

Sample Simulation Output

Listed below is a **sample** of dynamic execution. Each process scheduling algorithm simulator created for this paper produces an output satisfying the criteria below (in a similar format).

- The ready queue *Information* tag displays the process identification numbers and burst length for the processes in it.
-
- The device queue *Information* tag displays the process identification numbers and remaining amount of time that each process has before it becomes available again.

Sample output for a Context-Switch

[Information] RQ: P1(5) P2(4) P3(8) P4(3) P5(16) P6(11) P7(14) P8(4)
[Information] DQ: Empty
[Te: 5] Process 1 has ran
[Te: 5] Total execution finished: No

Sample output for Results

Turnaround time for P1: 395	Response time for P1: 0	Wait time for P1: 121
Turnaround time for P2: 591	Response time for P2: 5	Wait time for P2: 149
Turnaround time for P3: 557	Response time for P3: 9	Wait time for P3: 195
Turnaround time for P4: 648	Response time for P4: 17	Wait time for P4: 184
Turnaround time for P5: 530	Response time for P5: 20	Wait time for P5: 229
Turnaround time for P6: 445	Response time for P6: 36	Wait time for P6: 214
Turnaround time for P7: 512	Response time for P7: 47	Wait time for P7: 183
Turnaround time for P8: 493	Response time for P8: 61	Wait time for P8: 207
Average turnaround time is: 521.375	Average response time is: 24.375	Average wait time is: 185.25

Time needed to complete all processes: 648

CPU Utilization: 85.34%

Simulation Source Code

General

The source code that is unique to each scheduling algorithm references several *super* classes. These classes (files) are shown below.

Main.java

```
package com.fau;

import com.fau.process.Process;

import com.fau.schedulers.FCFS;
import com.fau.schedulers.MLFQ;
import com.fau.schedulers.SJF;

public class Main {

    public static void main(String[] args) {
        Process p1 = new Process(1, new int[] {5, 27, 3, 31, 5, 43, 4, 18, 6, 22, 4, 26, 3, 24, 4});
        Process p2 = new Process(2, new int[] {4, 48, 5, 44, 7, 42, 12, 37, 9, 76, 4, 41, 9, 31, 7, 43, 8});
        Process p3 = new Process(3, new int[] {8, 33, 12, 41, 18, 65, 14, 21, 4, 61, 15, 18, 14, 26, 5, 31, 6});
        Process p4 = new Process(4, new int[] {3, 35, 4, 41, 5, 45, 3, 51, 4, 61, 5, 54, 6, 82, 5, 77, 3});
        Process p5 = new Process(5, new int[] {16, 24, 17, 21, 5, 36, 16, 26, 7, 31, 13, 28, 11, 21, 6, 13, 3, 11, 4});
        Process p6 = new Process(6, new int[] {11, 22, 4, 8, 5, 10, 6, 12, 7, 14, 9, 18, 12, 24, 15, 30, 8});
        Process p7 = new Process(7, new int[] {14, 46, 17, 41, 11, 42, 15, 21, 4, 32, 7, 19, 16, 33, 10});
        Process p8 = new Process(8, new int[] {4, 14, 5, 33, 6, 51, 14, 73, 16, 87, 6});

        p1.parse();
        p2.parse();
        p3.parse();
        p4.parse();
        p5.parse();
        p6.parse();
        p7.parse();
        p8.parse();

        FCFS fcfs = new FCFS();
        fcfs.add(p1);
        fcfs.add(p2);
        fcfs.add(p3);
        fcfs.add(p4);
        fcfs.add(p5);
        fcfs.add(p6);
        fcfs.add(p7);
        fcfs.add(p8);
        fcfs.start();

        SJF sjf = new SJF();
        sjf.add(p1);
        sjf.add(p2);
        sjf.add(p3);
        sjf.add(p4);
        sjf.add(p5);
        sjf.add(p6);
        sjf.add(p7);
        sjf.add(p8);
        //sjf.start();

        MLFQ mlfq = new MLFQ();
        mlfq.add(p1);
        mlfq.add(p2);
        mlfq.add(p3);
        mlfq.add(p4);
        mlfq.add(p5);
        mlfq.add(p6);
        mlfq.add(p7);
        mlfq.add(p8);
        //mlfq.start();
    }
}
```

Process.java

```java
package com.fau.process;

import java.util.LinkedList;
import java.util.List;

public class Process {

    public int pId;

    int[] burst;

    public List<Integer> cpu_burst;
    public List<Integer> io_burst;

    public int io_wait = 0;

    public int queue = 1;

    public Process(int pId, int[] burst) {
        this.pId = pId;
        this.burst = burst;

        cpu_burst = new LinkedList<Integer>();
        io_burst = new LinkedList<Integer>();
    }

    public void parse() {
        for(int i = 0; i < burst.length; i++) {
            if(i % 2 == 0) {
                cpu_burst.add(burst[i]);
            } else {
                io_burst.add(burst[i]);
            }
        }
    }
}
```

Scheduler.java

```
package com.fau.process;

import java.util.LinkedList;
import java.util.List;

public class Process {

    public int pId;

    int[] burst;

    public List<Integer> cpu_burst;
    public List<Integer> io_burst;

    public int io_wait = 0;
    public int tmp_i = 0;
    public int turn_around_time = 0;
    public boolean first = false;
    public int time_response = 0;
    public int wait_time = 0;

    public int queue = 1;

    public Process(int pId, int[] burst) {
        this.pId = pId;
        this.burst = burst;

        cpu_burst = new LinkedList<Integer>();
        io_burst = new LinkedList<Integer>();
    }

    public void parse() {
        for(int i = 0; i < burst.length; i++) {
            if(i % 2 == 0) {
                cpu_burst.add(burst[i]);
            } else {
                io_burst.add(burst[i]);
            }
        }
    }
}
```

First Come, First Served

```java
package com.fau.schedulers;

import com.fau.process.Process;
import java.util.Collections;
import java.util.Comparator;

public class FCFS extends Scheduler {

    public void start() {
        while(!super.ready.isEmpty()) {
            System.out.println("[Information] RQ: " + rq_toString());
            System.out.println("[Information] DQ: " + devicequeue_toString());
            Process process = super.ready.remove(); // remove process from the queue

            if(super.pcb[process.pId - 1][0] == 0) { // if the process is running for the first time
                super.pcb[process.pId - 1][2] = super.time_execution; // record response time
                super.pcb[process.pId - 1][3] += super.time_execution; // save time waiting in ready queue
                super.pcb[process.pId - 1][0] = 1; // save that the process has ran
            }

            super.time_execution += process.cpu_burst.get(super.pcb[process.pId - 1][1]); // run a cpu burst
            System.out.println("[Te: " + super.time_execution + "] Process " + process.pId + " has ran");

            /* This should be modified in the future to not depend on an exception */
            System.out.print("[Te: " + super.time_execution + "] Total execution finished: ");
            try {
                process.io_wait = super.time_execution + process.io_burst.get(super.pcb[process.pId - 1][1]); // time the process will become available
                System.out.println("No");
                super.device.add(process); // Add to the device/io queue
            } catch (IndexOutOfBoundsException e) { // If this exception occurs, the process is finished (There are no more IO-bursts)
                super.terminated.add(process); // Add to the terminated queue
                super.pcb[process.pId - 1][4] = super.time_execution; // Record turn around time
                System.out.println("Yes");
            } finally { System.out.println(); }

            if(super.ready.isEmpty() && !super.device.isEmpty()) {
                Process p = super.device.peek();
                for(Process temp_p : super.device) { // This loop finds the process in the device queue that is soonest to become eligible to run
                    if(temp_p.io_wait < p.io_wait) {
                        p = temp_p;
                    }
                }

                if(super.time_execution < p.io_wait) { // If the soonest available process is greater than the current time execution, idle
                    System.out.println("[CPU] - Idling");
                    cpu_util += (p.io_wait - super.time_execution); // add idle time for analytics
                    super.time_execution += p.io_wait - super.time_execution;
                }

                super.pcb[p.pId - 1][1] +=1 ;
                super.ready.add(p);
                super.device.remove(p);
                super.pcb[process.pId - 1][3] += super.time_execution - p.io_wait; // add to wait time, for analytics
            }
        }

        super.print_analytics();
    }

}
```

Shortest Job First

```java
package com.fau.schedulers;

import com.fau.process.Process;

import java.util.Collections;
import java.util.Comparator;
import java.util.LinkedList;

public class SJF extends Scheduler {

    public void start() {

        while (!ready.isEmpty()) {

            Collections.sort(ready, new Comparator<Process>() { // Sort the ready queue, so that the shortest job "pops"
                @Override
                public int compare(Process o1, Process o2) {
                    return o1.cpu_burst.get(pcb[o1.pId - 1][1]) - o2.cpu_burst.get(pcb[o2.pId - 1][1]);
                }
            });


            System.out.println("[Information] RQ: " + rq_toString());
            System.out.println("[Information] DQ: " + devicequeue_toString());
            Process process = ready.peek(); // get process with shortest cpu burst
            for(Process p : ready) {
                if(p.cpu_burst.get(pcb[p.pId - 1][1]) == process.cpu_burst.get(pcb[process.pId - 1][1]) && p.pId != process.pId) { // if burst == another burst, fcfs (pick one with lesser
arrival time)
                    System.out.println("Process " + p.pId + " has the same CPU burst (" + p.cpu_burst.get(pcb[p.pId - 1][1]) + ") as Process " + process.pId);
                    if(p.io_wait < process.io_wait) {
                        int arrival_time = process.io_wait;
                        process = p;
                        System.out.println("Running Process " + process.pId + " because " + process.io_wait + " < " + arrival_time);
                    }
                }
            }

            if(pcb[process.pId - 1][0] == 0) { // for recording response time
                pcb[process.pId - 1][2] = super.time_execution;
                pcb[process.pId - 1][0] = 1;
            }

            // for recording wait time
            int wait_time = super.time_execution - process.io_wait;
            pcb[process.pId - 1][3] += wait_time;

            super.time_execution += process.cpu_burst.get(pcb[process.pId - 1][1]); // "run" the process
            System.out.println("[Te: " + super.time_execution + "] - " + "Process " + process.pId + " has ran");

            System.out.print("Total execution has finished: ");
            /* in the future, try and catch should not be used for flow control; only simluation project! */
            try { // if this block does not enter the exception, there are more io bursts
                process.io_wait = super.time_execution + process.io_burst.get(pcb[process.pId - 1][1]);
                device.add(process);
                System.out.println("No");

            } catch(IndexOutOfBoundsException e) { // if exception occurs, there are no more io bursts
                pcb[process.pId - 1][4] = super.time_execution;
                terminated.add(process);
                System.out.println("Yes");
            }

            ready.remove(process); // process ran, remove from ready queue

            if(!device.isEmpty()) { // check if any processes have become available from device queue

                LinkedList<Process> tmp = new LinkedList<>(); // temp store all processes that have become available
                for(Process p : device) {
                    if(p.io_wait <= super.time_execution) {
                        tmp.add(p);
                        ready.add(p); // add processes that have become available to ready queue
                    }
                }

                for(Process p : tmp) { // remove all processes that have become available from device queue
                    pcb[p.pId - 1][1] += 1;
                    device.remove(p);
                }
            }

            if(ready.isEmpty() && !device.isEmpty()) { // if this is true, cpu must idle
                Process temp_p = device.peek();
                for(Process p : device) { // find the process that is soonest to become available
                    if(p.io_wait < temp_p.io_wait) {
                        temp_p = p;
                    }
                }

                cpu_util += (temp_p.io_wait - super.time_execution);

                super.time_execution += (temp_p.io_wait - super.time_execution);

                ready.add(temp_p);
                pcb[temp_p.pId - 1][1] += 1;
                device.remove(temp_p);
            }

            System.out.println();

        }

        super.print_analytics();
    }
}
```

Multilevel Feedback Queue

```java
package com.fau.schedulers;

import java.util.*;

import com.fau.process.Process;

public class MLFQ extends Scheduler {

    private final int RR_TQ_QUEUE_1 = 5;
    private final int RR_TQ_QUEUE_2 = 10;

    private Queue<Process> queue_1;
    private Queue<Process> queue_2;
    private Queue<Process> queue_3;

    public MLFQ() {
        queue_1  = new LinkedList();
        queue_2  = new LinkedList();
        queue_3  = new LinkedList();
    }

    public void start() {
        for(Process p : ready) {
            queue_1.add(p);
        }

        while(!queue_1.isEmpty() || !queue_2.isEmpty() || !queue_3.isEmpty() || !device.isEmpty()) {

            while(!queue_1.isEmpty()) {
                System.out.println();
                System.out.println();
                System.out.println("Q1: " + this.q1_toString() );
                System.out.println("Q2: " + this.q2_toString() );
                System.out.println("Q3: " + this.q3_toString() );
                System.out.println("DQ: " + this.devicequeue_toString() );

                Process p = queue_1.remove();
                if(pcb[p.pId - 1][0] == 0) { pcb[p.pId - 1][0] = 1; pcb[p.pId - 1][2] = time_execution; } // time response

                if(p.cpu_burst.get(pcb[p.pId - 1][1]) <= RR_TQ_QUEUE_1) { // run in full
                    time_execution += p.cpu_burst.get(pcb[p.pId - 1][1]);
                    System.out.println("[Te: " + time_execution + "] - Process " + p.pId + " ran");
                    System.out.print("[Te: " +  time_execution + "] Total execution has finished: ");
                    try {
                        pcb[p.pId - 1][3] += (time_execution - p.cpu_burst.get(pcb[p.pId - 1][1])) - p.io_wait;
                        p.io_wait = time_execution + p.io_burst.get(pcb[p.pId - 1][1]);
                        device.add(p);
                        System.out.print("No");
                    } catch(IndexOutOfBoundsException e) {
                        terminated.add(p);
                        System.out.println("Yes");
                        pcb[p.pId - 1][4] = time_execution;
                    }

                } else { // dont run in full, downgrade process
                    time_execution += RR_TQ_QUEUE_1;
                    System.out.println("[Te: " + time_execution + "] - Process " + p.pId + " ran");
                    p.cpu_burst.set(pcb[p.pId - 1][1], p.cpu_burst.get(pcb[p.pId - 1][1]) - RR_TQ_QUEUE_1);
                    queue_2.add(p);
                    p.queue = 2; // track which queue process belongs in (processes can never be upgraded after being downgraded)
                    System.out.println("[Te: " + time_execution + "] Total execution has finished: No");
                }

                check_io();
            }

            while(!queue_2.isEmpty() && queue_1.isEmpty()) {
                System.out.println();
                System.out.println();
                System.out.println("Q1: " + this.q1_toString() );
                System.out.println("Q2: " + this.q2_toString() );
                System.out.println("Q3: " + this.q3_toString() );
                System.out.println("DQ: " + this.devicequeue_toString() );


                Process p = queue_2.remove();
                if(p.cpu_burst.get(pcb[p.pId - 1][1]) <= RR_TQ_QUEUE_2) { // run in full
                    time_execution += p.cpu_burst.get(pcb[p.pId - 1][1]);
                    System.out.println("[Te: " + time_execution + "] - Process " + p.pId + " ran");
                    System.out.print("[Te: " +  time_execution + "] Total execution has finished: ");

                    try {
                        pcb[p.pId - 1][3] += (time_execution - p.cpu_burst.get(pcb[p.pId - 1][1])) - p.io_wait;
                        p.io_wait = time_execution + p.io_burst.get(pcb[p.pId - 1][1]);
                        device.add(p);
                        System.out.print("No");
                    } catch(IndexOutOfBoundsException e) {
                        terminated.add(p);
                        pcb[p.pId - 1][4] = time_execution;
                        System.out.print("Yes");
                    }
                } else { // dont run in full, downgrade process
                    time_execution += RR_TQ_QUEUE_2;
                    System.out.println("[Te: " + time_execution + "] - Process " + p.pId + " ran");
                    System.out.print("[Te: " +  time_execution + "] Total execution has finished: No");

                    p.cpu_burst.set(pcb[p.pId - 1][1], p.cpu_burst.get(pcb[p.pId - 1][1]) - RR_TQ_QUEUE_2);
                    queue_3.add(p);
                    p.queue = 3; // track which queue process belongs in (processes can never be upgraded after being downgraded)
                }
```

```java
                check_io();
            }

            while(!queue_3.isEmpty() && queue_1.isEmpty() && queue_2.isEmpty()) { // fcfs ** possibly where things are going wrong
                System.out.println();
                System.out.println();

                System.out.println("Q1: " + this.q1_toString() );
                System.out.println("Q2: " + this.q2_toString() );
                System.out.println("Q3: " + this.q3_toString() );
                System.out.println("DQ: " + this.devicequeue_toString() );

                Process p = queue_3.remove();
                time_execution += p.cpu_burst.get(pcb[p.pId - 1][1]);
                System.out.println("[Te: " + time_execution + "] - Process " + p.pId + " ran");
                System.out.print("[Te: " +  time_execution + "] Total execution has finished: ");

                try {
                    pcb[p.pId - 1][3] += (time_execution - p.cpu_burst.get(pcb[p.pId - 1][1])) - p.io_wait;
                    p.io_wait = time_execution + p.io_burst.get(pcb[p.pId - 1][1]);
                    device.add(p);
                    System.out.print("No");
                } catch(IndexOutOfBoundsException e) {
                    terminated.add(p);
                    pcb[p.pId - 1][4] = time_execution;
                    System.out.print("Yes");
                }

                check_io();
            }
        }

        super.print_analytics();
    }

    public void check_io() {
         if(!device.isEmpty()) { // might want to add multiple processes
            Process p = device.peek();
            for(Process process : device) {
                if(process.io_wait < p.io_wait) {
                    p = process;
                }
            }

            if(p.io_wait <= time_execution) {
                device.remove(p);

                pcb[p.pId - 1][1] += 1;
                switch(p.queue) {
                    case 1: queue_1.add(p); break;
                    case 2: queue_2.add(p); break;
                    case 3: queue_3.add(p); break;
                }
            }
         }

         if(!device.isEmpty() && queue_1.isEmpty() && queue_2.isEmpty() && queue_3.isEmpty()) { // cpu is going to be idle
             Process p = device.peek();
             for(Process process : device) {
                 if(process.io_wait < p.io_wait) {
                     p = process;
                 }
             }

             super.cpu_util += (p.io_wait - time_execution);
             time_execution += p.io_wait - time_execution;
             device.remove(p);

             pcb[p.pId - 1][1] += 1;
             switch(p.queue) {
                 case 1: queue_1.add(p); break;
                 case 2: queue_2.add(p); break;
                 case 3: queue_3.add(p); break;
             }
         }
    }

    private String q1_toString() {
        if (this.queue_1.isEmpty()) return "Empty";

        StringBuilder sb = new StringBuilder();
        for(Process p : this.queue_1) {
            sb.append("P" + p.pId + "(" + p.cpu_burst.get(this.pcb[p.pId - 1][1]) + ") ");
        }
        return sb.toString();
    }

    private String q2_toString() {
        if (this.queue_2.isEmpty()) return "Empty";

        StringBuilder sb = new StringBuilder();
        for(Process p : this.queue_2) {
            sb.append("P" + p.pId + "(" + p.cpu_burst.get(this.pcb[p.pId - 1][1]) + ") ");
        }
        return sb.toString();
    }

    private String q3_toString() {
        if (this.queue_3.isEmpty()) return "Empty";

        StringBuilder sb = new StringBuilder();
        for(Process p : this.queue_3) {
            sb.append("P" + p.pId + "(" + p.cpu_burst.get(this.pcb[p.pId - 1][1]) + ") ");
        }
        return sb.toString();
    }
}
```