CHUA, Mary Elizabeth E.
NISAY, Deiondre Judd V.

CMSC 125 Machine Problem
CPU Scheduling
Documentation

I. Introduction

As with most operating systems concepts, each contains different ways in order to solve a problem. One of which is the problem of scheduling processes in a computer system. This program aims to simulate three of the different methods of CPU scheduling.

1. Shortest-Job-First (SJF)
2. Priority Scheduling (PS)
3. Round-Robin (RR)

Furthermore, the program allows the user to enter a file in which the process information is indicated.

Instructions in using the program is as follows:

1. Double click the launcher.bat file. This will open the console window.
2. The program will ask you to choose which algorithm to use. The key to be entered must only be “A”, “B”, or “C”. Any other character will not be allowed by the program.
3. Afterwards, the program will ask for the filename including the extension (e.g. Test.txt). The capitalization will matter. The following is a guide to making the text file for the process information.

Process	Burst Time	Arrival Time	Priority
P1	10	0	1
P2	4	0	1
P3	6	0	1

- ★ There must be **at least two lines**. One header, one process.
- ★ There must not be an empty row in the input text file.
- ★ The **header must be properly formatted.** Capitalization does not matter. The order matters (process, burst time, arrival time, priority.). Each title can be separated by space or tabs. If the titles are only “burst” and “arrival”, therefore missing the word “time” in it, there will be an error.
- ★ The **process names can be anything**. The **burst time, arrival time, and priority must be a nonnegative integer. The burst time must not be 0.**
- ★ **Missing values will result in an error.** However, if improperly formatted with extra values, the program will take in the values by its order and not its indent. For example,

Process	Burst Time	Arrival Time	Priority		
	10		1	2	
P2	4		1		
P3		0	1	3	7

This is an acceptable file, and will take in the values according to the order of each value in each line. This must be noted. The first process will be named "10" and will have a burst time of 1 and arrival time of 2.

If the file doesn't exist, the program will ask you if you'd want to enter another filename.

The text file **must be in the same folder as the MP folder.**

4. Once the program gets a valid file, it will check its contents and see if it is okay to run.
    a. If the user chooses **round-robin** the program will ask for the time quantum and must enter a **nonnegative integer greater than 0.**
    b. If the user chooses **SJF or PS,** the program will continue.
5. Afterwards, you will see the running processes and the program will provide the summary of the waiting and turnaround times. The program will then ask the user if they want to restart the program or terminate the process. The choices must be "A" or "B" only.

II. Data Structures (Primitive and Non-primitive)

A. Shortest-Job-First (SJF)

Int:

- int burst - takes the burst time of the current process (seconds) for nextProcessForSJF()
- int burstTime - takes the burst time of the current process (seconds) for SJF()
- int current - takes the index of the current column of the 2D array data[][]
- int isFirstIter - indicates if the current iteration is the first one
- int lowest - takes the arrival time of the current process that is the lowest among all
- int lowestBurst - takes the burst time of the current process that is the lowest among all
- int next - takes the index of the next process to queue
- int number - takes the arrival time of the current process
- int sec - counter of the whole process in seconds
- int waitFirst - takes the arrival time of the current process

Double:

- double avgWait - average waiting time of the current process
- double avgTurn - average turnaround time of the current process
- double totalWait - total waiting time of the current process
- double totalTurn - total turnaround time of the current process

String:

- String[] endtime - takes the timestamp of when the process has ended
- final String[][] origData - takes the untouched copy of the data
- String[] startTime - takes the timestamp of when the process has started
- String timeStamp - takes the current timestamp

Misc:

- Duration duration - takes the duration between the time when the process arrived and when it started processing
- LocalDateTime arrived - time when the process arrived
- LocalDateTime startEverything - time when the whole CPU scheduling process started
- LocalDateTime[] startTimes - takes the start times of every process into an array
- long TAT - takes the turnaround time of the current process

- Timer timer - initialization of the timer

B. Priority Scheduling (PS)
   - int burstTime - takes the burst time of the current process (seconds) for PS()
   - int current - takes the index of the current column of the 2D array data[][]
   - int highestpriority - takes the highest priority of all processes at that instant
   - int isFirstIter - indicates if the current iteration is the first one
   - int lowest - takes the arrival time of the current process that is the lowest among all
   - int next - takes the index of the next process to queue
   - int number - takes the arrival time of the current process
   - int priority - takes the priority of the current process
   - int sec - counter of the whole process in seconds
   - int waitFirst - takes the arrival time of the current process

   Double:
   - double avgWait - average waiting time of the current process
   - double avgTurn - average turnaround time of the current process
   - double totalWait - total waiting time of the current process
   - double totalTurn - total turnaround time of the current process

   String:
   - String[] endtime - takes the timestamp of when the process has ended
   - final String[][] origData - takes the untouched copy of the data
   - String[] startTime - takes the timestamp of when the process has started
   - String timeStamp - takes the current timestamp

   Misc:
   - Duration duration - takes the duration between the time when the process arrived and when it started processing
   - LocalDateTime arrived - time when the process arrived
   - LocalDateTime startEverything - time when the whole CPU scheduling process started
   - LocalDateTime[] startTimes - takes the start times of every process into an array
   - long TAT - takes the turnaround time of the current process
   - Timer timer - initialization of the timer

C. Round Robin (RR)

   Int:
   - int ctHolder - position value for completionTime array
   - int lesserDur - placeholder for printing time markers
   - int marker - position value for the timeMarkers array
   - int mostDur - placeholder for printing time markers
   - int myAnti - used to determine if the queue is empty, filled, or idle
   - int myCheck - used to determine if a spot in the queue is taken
   - int myFit - determines if the queue is ready to be processed
   - Int hasRan - determines if the program has started executing
   - int myNewSpace - to give a size for processesToRun array
   - int myNonOut - checks if process has been placed in a queue position
   - int myRepos - a counter which determines if the program will execute a process or be idle
   - int myTurn - used to determine if the process is already in the queue array

- int num - holds the parsed integer value of the string/character
- int numberOfProcesses - number of processes to execute
- int pcounter - counter to determine if all processes have already been executed
- int qcounter - counter leading up to the time quantum
- int temp - placeholder for processes to print
- int tempCounter - determines the queue position for the processes
- int temper - placeholder for tempCounter
- int timeQuantum - time quantum
- int timerCounter - counter for every iteration
- int validLoop - used to determine if the inputted time quantum is valid
- int[ ] burstTime - array of burst times of each process
- int[ ] completionTime - array of completion times of each process
- int[ ] processesToRun - queue array for the processes to run, also lists down if idle

Double:

- int avgTaT - average turnaround time
- int avgWT - average waiting time

String:

- String timeStamp - placeholder for the time the process ran
- String[ ] data - array for user's input for burst time, arrival time, priority, and number of processes
- String[ ] timeMarkers - array for all processes' time duration of each time quantum

Misc:

- Scanner tq - placeholder for inputted time quantum
- Timer timer - variable to implement time constraints

D. InputData() and check()

Int:

- int columns - columns counter for the processes
- int headerTitles - counts the appropriate header titles
- int intValue - takes in the value of the cells/data and turns it into an integer from string
- int lines - counts the number of lines there is in the given file
- int numberOfProcesses - counts the number of processes
- int required - the number of required headerTitles for the given algorithm
- int rows - rows counter for the processes
- int skip - tells when to skip the loop if an error is already found
- int words - the number of words in the given file
- int wpl - words per line

String:

- String[][] data - the 2D array containing all the information about the processes
- String decision - decision of the user if they want to enter a valid file
- String error - stores the type of error
- String fileName - takes the name of the file
- String[][] ha - placeholder for data[][]
- String s3 - takes the next word while reading the file

Boolean:

- boolean answer - checks if the words are an integer

- boolean exists - checks if the file exists
- boolean okayFile - checks if the file have valid data

Misc:

- File f - stores the filename
- Scanner s - scans the file
- Scanner s2 - scans the lines of the file
- Scanner s3 - scans the file again
- Scanner s4 - scans the lines of the file again

III. Algorithms

At the start of the program, the user will be prompted to enter their desired algorithm. Once they have chosen a method, they will then be prompted to enter the filename that contains the information about the processes that the program will simulate. In this section, the algorithms for the three different scheduling algorithms, Shortest-Job-First, Priority Scheduling, and Round Robin will be discussed. Other than this, some error handling and technicalities when processing the file information will also be discussed.

A. Shortest-Job First (SJF)

In order to simulate how the CPU Scheduler uses the non-preemptive Shortest-Job-First algorithm or the shortest next CPU burst algorithm, the program must also simulate how each processes arrive according to their stated arrival times. Since the algorithm is non-preemptive, once a process starts, it cannot be interrupted and must finish until completion before another process can start. Therefore, the algorithm is split into two parts when it decides which process it will start. The first part of the algorithm will determine at the start of the whole processes, which process will go first according to the current timestamp 0:00:00, since it is the start of the whole thing, most often the program would choose the process with the lowest arrival time if it is exclusively the most early process to arrive. If there are others with the same arrival time, it will then compare their burst times and choose the one with the least burst time. If no process has arrived yet, the CPU will be idle.

The second part of the algorithm comes in when the first process is completed. This algorithm will then run until the last process is done. The second part is different from the first part only because it now takes in the current timestamp only **after** 0:00:00, in this way, since the current running time is non-zero, we can consider the processes that have arrived and are waiting. Those processes that arrived already will only be compared by their burst times first, if the burst times are the same, we then choose the process that came first. If no process is ready yet, the program will simulate that the CPU is idle.

For the technicalities, the program simulates the CPU scheduling by incrementing the timer by 1 second every time something is happening, such as the CPU being idle or when a process is in progress, which decrements the burst time of the process until it is zero. Once the burst time of a process is zero, the program makes the arrival time -1 to indicate that it is a finished process and it does not need to be considered.

The overall process can be simplified looking at this pseudocode:

```
start of SJF:
get data 2d array
save a copy of the 2d array

take the timestamp indicating the start of the cpu scheduler
call another function to determine the next process
...
next process:
if at time 0:00:00
    for every process
        if the process has not finished and there is no lowest arrival time yet
            then this process has the lowest arrival time so far
            this process also has the lowest burst time so far
        else if process has not finished and lower than the current lowest arrivaltime
            then this process has the lowest arrival time so far
            this process also has the lowest burst time so far
        else if process has not finished and equal to the lowest arrivaltime
            if the process has a lower burst time than the lowest burst time
                then this process also has the lowest burst time so far
else the time is not at 0:00:00
    for every process
        if the process has not finished and there is no lowest burst time yet
            then this process has the lowest arrival time so far
            this process also has the lowest burst time so far
        else if process has not finished and lower than the current lowest bursttime
            then this process has the lowest arrival time so far
            this process also has the lowest burst time so far
        else if process has not finished and it is equal to the lowest burst time
            if the process has a lower arrival time than the lowest arrival time
                then this process has the lowest burst time so far
                then this process also has the lowest arrival time so far
return the process with the lowest burst time that came first
```

```
//actual cpu processing

for every process
    get current timestamp
    get arrival time
    get burst time

    while the arrival time is less than current time
        get time stamp
        print "idle"
        time increment 1 second

    while burst time is greater than 0
        get time stamp
        print "running process"
```

```
        time increment 1 seconds
        burst time decrement

    tag the process as finished
    get time stamp
    call again the function that determines the next process

print summary
```

Furthermore, we can discuss snippets of the code to see the innerworkings of the actual program.

When the user chooses the SJF algorithm, the method SJF(inputData(algo)); will be called.

```
SJF(inputData(algo));
```

This calls the inputData() first and passes the value of the algo that will signal the method that the SJF is chosen, and will check if the file is valid for an SJF algorithm. Then, when it successfully retrieves data from the user's file, it will then pass the **2D array** of the information about the processes to the SJF method.

```
for(int i=0; i<data.length; i++){
            for(int j=0; j<data[0].length; j++){
                origData[i][j] = data[i][j];
```

This nested for loop merely copies the original data to another 2D array so that we can retain the information to be printed in the summary later.

```
LocalDateTime startEverything = LocalDateTime.now();
        LocalDateTime arrived;
        LocalDateTime[] startTimes = new LocalDateTime[data.length];
```

Furthermore, we then save the timestamp that signifies the start of the CPU scheduling, we also initialize the array of start times to indicate the processes' start times.

```
current = nextProcessForSJF(data, isFirstIter, sec);
```

Here we call on the function nextProcessForSJF() and pass the parameters. This part of the code will check each of the processes and determine which one will go first.

```
 // FOR THE FIRST PROCESS
        if(sec==0){
        for(int x=0; x<data.length; x++){
            …
            if((lowest==-1)&&(number!=-1)){
```

```
                lowest = number;lowestBurst = burst;next = x;}
            else if((number<lowest)&&(number!=-1)){
                lowest = number;lowestBurst = burst;next = x;}
            else if((number==lowest)&&(number!=-1)){
                if(burst<lowestBurst){
                    lowestBurst = burst;next = x;
                }}}}
```

We then check if the CPU had just started, meaning, the method in which we choose the first ever process to schedule would be different than when the CPU had already finished processing the first one. So in this case, if the seconds is at 0 (timer 0:00:00) then we go into a for loop looking at every process to check the lowest arrival time and lowest burst. We then store the index of the process that we chose to the variable next

```
      else{
        for(int x=0; x<data.length; x++){
            …
            if(number>sec){continue;}
            if((lowestBurst==-1)&&(number!=-1)){
                lowest = number;lowestBurst = burst;next = x;}
            else if((burst<lowestBurst)&&(number!=-1)){
                lowest = number;lowestBurst = burst;next = x;}
            else if((burst==lowestBurst)&&(number!=-1)){
                if(number<lowest){
                    lowest = number;lowestBurst = burst;next = x;
                }}}}
return next;
```

On the other hand, if the seconds is not at 0 (or timer has started and exceeds 0:00:00) then we go into a for loop that is the same as the previous one, but instead of checking the arrival time first, we check the lowest burst time first and then consequently checking the least arrival time after. Furthermore, `if(number>sec){continue;}` checks if the process has arrived yet, if not, it skips its iteration and checks other processes. Afterwards, the function will then return the index of the chosen process.

```
for(int y=0; y<data.length; y++){
…
      while(waitFirst>sec){
      …
      System.out.println(timeStamp + "\tIdle... ");
      sec++;Thread.sleep(1000);}
      …
      while(burstTime>0){
      …
```

```
      System.out.println(timeStamp + "\tRunning Process "…
      burstTime--;sec++;Thread.sleep(1000);}

      isFirstIter = 0;
      data[current][2] = "-1";
      endTime[current] = timeStamp;
      current = nextProcessForSJF(data, isFirstIter, sec);
      }
```

Now that we know which process to run, let's actually run it. If the process that we chose hasn't arrived yet `while(waitFirst>sec)` we print out that the CPU scheduling is idle and then increment seconds until the process arrives. Afterwards, we print that the CPU scheduler is running the processes and then decrement its burst time while incrementing the seconds still. After the burst turns 0, we then set the indicator that this is no longer the first iteration, then change the arrival time of the process to "-1" to show that it's finished. We also save the timestamps of when it starts and ends so that we can print it later. We then start queueing up the next process to be run.

```
//PRINTING OF SUMMARY
      System.out.println("\nSummary");
      …
      System.out.println("Average Turnaround Time: " + avgTurn +
"seconds\n");
}
```

The next part of the code is just computing and printing the summary of the CPU scheduling process.

```
Duration duration = Duration.between(arrived, startTimes[j]);
long TAT = duration.getSeconds() + Long.parseLong(origData[j][1]);

totalWait = totalWait + duration.getSeconds();
totalTurn = totalTurn + TAT;
```

The variable duration gets the time between when the process arrived and when it started. The variable TAT then gets that duration and adds the burst time. This then shows us the turnaround time for that process. Furthermore, we get the total waiting time by adding the waiting time of every process to totalWait and the turnaround time of every process to totalTurn.

```
avgWait = totalWait/data.length;
avgTurn = totalTurn/data.length;
```

We then get the average of each by dividing it by the total number of processes.

B. Priority Scheduling (PS)

In order to simulate the CPU scheduling using Priority Scheduling method, we have to queue the processes by their priority. As discussed earlier, SJF queues the processes by their burst times. SJF may resemble the algorithm for PS especially if both are non preemptive. So the implementation of the PS algorithm for this program basically is the copy-paste of the SJF algorithm except we have changed a few parts of the code.

```
if(sec==0){
for(int x=0; x<data.length; x++){
…
    if((lowest==-1)&&(number!=-1)){
        lowest = number;highestpriority = priority;next = x;}
    else if((number<lowest)&&(number!=-1)){
        lowest = number;highestpriority = priority;next = x;\}
    else if((number==lowest)&&(number!=-1)){
        if(priority<highestpriority){
            highestpriority = priority;next = x;}}}}
```

Recall that the method called for SJF to determine the next process to handle is called `nextProcessForSJF();`, for PS it is called `nextProcessForPS();` and only 2 parts of the code is changed. Instead of comparing which process has the least burst time, we compare which process has the highestpriority. As seen in the code snippet above, when the whole CPU scheduler has just started, we first check which process has first arrived, and then compare which has the highest priority. That process will then have its index saved and returned to the main algorithm for the program to process.

```
else{
for(int x=0; x<data.length; x++){
    …
    if(number>sec){continue;}
    if((highestpriority==-1)&&(number!=-1)){
        lowest = number;highestpriority = priority;next = x;}
    else if((priority<highestpriority)&&(number!=-1)){
        lowest = number;highestpriority = priority;next = x;}
    else if((priority==highestpriority)&&(number!=-1)){
        if(number<lowest){
            lowest = number;highestpriority = priority;next = x;}}}}
```

On the other hand, when the seconds is not equal to 0, we first check if the process has already arrived, if not, we skip it. Then for those processes that already arrived, we check which of these have the highest priority and if there are tied processes in the top priority, then we check if one arrived earlier than the other.

Other than this, the PS code is basically the same as the SJF. The printing of the summary is basically the same as well.

C. Round Robin Algorithm (RR)

All arrays will be initialized with values in each array position

```
String[] timeMarkers = new String[100];
        for(int j = 0; j<100; j++){
            timeMarkers[j]=String.valueOf(-1);
        }
int[] burstTime = new int[numberOfProcesses];
        for(int x = 0; x<numberOfProcesses ; x++){
            burstTime[x] = Integer.parseInt(data[x][1]);
        }
int myNewSpace = numberOfProcesses + numberOfProcesses;
int[] processesToRun = new int[myNewSpace];
for (int i = 0; i <= numberOfProcesses; i++) {
            processesToRun[i]=-2;
        }
```

This is so that when the Round Robin algorithm runs, the way it'll check if the array is empty is by checking if the initialized value exists in a certain array position, wherein it'll add values to it if it is indeed empty. As for the integer array burstTime, it simply records all the burst times of all processes since the math will affect the values of the data array.

After declaring and initializing the variables needed for the Round Robin algorithm, the program will ask for the time quantum wherein only integers greater than 0 will be accepted. The algorithm for scanning the user's time quantum input is as follows:

```
Scanner tq = new Scanner(System.in);
while (validLoop==0) {
            if (tq.hasNextInt()) {
                timeQuantum = tq.nextInt();
                validLoop=1;
            }
            else{
                System.out.println("Invalid input, try again");
                System.out.println();
                tq = new Scanner(System.in);
                System.out.print("Provide time quantum in seconds: ");
                timeQuantum=1;
            }
```

```
            if (timeQuantum <= 0) {
                System.out.println("Time quantum must be greater than 0");
                System.out.println();
                tq = new Scanner(System.in);
                System.out.print("Provide time quantum in seconds: ");
                validLoop=0;
            }
        }
```

After the scanner has been declared, it will go through an if-else statement to see if the input is an integer or not. If it's an integer, the timeQuantum variable will inherit the input. Otherwise, it will alert the user that the input is invalid and they have to try again. To add to that, if the time quantum is an integer but is equal or less than 0, even after it accepts the input, this algorithm will clear the scan buffer and alert the user to input once again.

The round robin algorithm will happen within this while loop. Every after a process has finished running completely, pcounter will increment by 1 until it reaches the total number of processes in which it signifies the completion of running all processes.

```
while(pcounter < numberOfProcesses){
...
}
```

Inside the while loop, there is an if statement where it checks if the qcounter is equal to the time quantum value.

```
if (qcounter==timeQuantum) {
                qcounter = 0;
                processesToRun[0]=-2;
            }
while (qcounter < timeQuantum) {
...
}
```

If qcounter is equal, the algorithm will reset qcounter. Afterwards, it will assign -2 to the first position in the queue array, processesToRun[0]. The qcounter is a counter incrementing by 1 after every while loop displayed below until it's equal to the time quantum, as displayed in the while loop below.

Within the second while loop, while (qcounter < timeQuantum), this for-loop will check if the queue array is empty

```
for (int i = 0; i <= numberOfProcesses; i++) {       // to see if the
queue is empty (no wait yet, not even idle)
```

```
                    if (processesToRun[i]!= -2) {
                        myAnti = 1;
                    }
                }
```

If it's empty, it will assign a value of 1 to the variable myAnti, which will be used in the code snippet below.

```
if (myAnti == 1) {
                    for (int k = 0; k <= numberOfProcesses; k++) {
                        if (processesToRun[k] == -2) {
                            for(int m = k; m < numberOfProcesses; m++){
                                if(m < numberOfProcesses){
                                    processesToRun[m]=processesToRun[m+1];
                                    processesToRun[m+1]=-2;
                                }
                                else{
                                    processesToRun[m]=-2;
                                }
                            }

                        }
                    }
                }
myAnti = 0;
```

If myAnti is 1, it will run a for loop where in what this does is check if the queue array position before them is empty, which would have a value of -2. If it does, it will move all the values forward, and the array position they were at before will be assigned a value of -2, which would indicate an empty position. Afterwards, myAnti will be assigned a value of 0 to restart the process after it loops back.

After the queue has been sorted, it will go through this while-loop which lists down processes to be added in a queue by going through a for-loop to see if the process is ready to be added or not by placing the arrival time of the process in the integer num. The integer temper will be assigned tempCounter's value since there will be a case later where we will need to increase the queue position to insert the process into the queue.

```
while (myFit == 0) {

                    for (int i = 0; i < numberOfProcesses; i++) {
                        num = Integer.parseInt(data[i][2]);
                        temper = tempCounter;
```

Once num has been assigned a value, it will then check if num is less than or equal to timerCounter, which means it'll check if the timer has gone on or past the arrival time of a certain process. If it meets the condition, it will then check if the position in the queue array is taken. If it's empty, it will undergo another if-else statement to see if the burst time of the process is 0 or not. If the burst time of the process is 0, it does nothing. Otherwise, it will enter a for-loop to check the entire queue array to see if the process in question is already in the queue array. As it goes through every position in the queue array, it checks if the process in question matches the process in a queue position. If it matches, the value 1 will be assigned to myTurn. Otherwise, myTurn will stay 0.

```
                    if (num <= timerCounter){
                        if (processesToRun[tempCounter]==-2) {
                            if (Integer.parseInt(data[i][1])!=0) {
                                for(int g=0;g<=numberOfProcesses;g++){
                                    if (processesToRun[g] == i) {
                                        myTurn = 1;
                                    }
                                }
```

After the for-loop ends, it will check if the value of myTurn remains 0 or not. If it remains 0, meaning the process was not in the queue, it will assign the process into that queue position which was empty in the first place. However, if myTurn is 1, it lets the queue position remain empty for the next process. Afterwhich, it gives the myTurn a value of 0 again so its process can be repeated

```
                                if(myTurn == 0){
                                    processesToRun[tempCounter]=i;
                                    tempCounter = tempCounter + 1;
                                }
                                else{
                                    processesToRun[tempCounter]=-2;
                                }
                                myTurn = 0;
                            }
                        }
```

Since the sort algorithm before this while-loop, pushed all the processes in the queue position to the front, meaning the only empty positions will be the values behind the queue positions in which are occupied by processes. So this algorithm will only consider the queue positions at the back of the queue array while checking the processes in the front of the queue array if the process in question is already in the queue. This will avoid any overtaking in positions for the process to execute.

Now, if the queue position is not empty and it's taken by another process in the queue array, it will enter a while-loop wherein, same as before, it'll check if the process' burst time is 0 or not. If the burst time is zero, it simply leaves the while-loop. Otherwise, it will enter the for-loop which checks if the process in question is already queued in the queue array. If the process in question matches with a process in the queue array, myTurn will be assigned a value of 1. On the other hand, if the process in question doesn't match with any processes in the queue array, myTurn will remain 0.

```
                    else{
                        while(myCheck == 0){
                            if (Integer.parseInt(data[i][1])!=0) {
                               for(int g=0;g<=numberOfProcesses;g++){
                                        if (processesToRun[g] == i) {
                                            myTurn = 1;
                                    }
                                }
```

If myTurn is 1, it simply leaves the while-loop since the process is already in the queue. However, if myTurn is 0, it enters another while-loop wherein its purpose is to check for the next available queue position to insert the process in question.

```
                                if(myTurn == 0){
                                  while(myNonOut==0){
                                    if (temper<=numberOfProcesses){
                                      temper = temper + 1;
                                      if (processesToRun[temper]==-2){
                                        processesToRun[temper]=i;
                                        tempCounter=tempCounter+1;
                                        myNonOut = 1;
                                                }
                                            }
                                        }
                                        myNonOut=0;
                                        temper = 0;
                                        myCheck = 1;
                                    }
                                 else{
                                   myCheck = 1;
                                 }
                             }
                             else{
                               myCheck = 1;
                             }
```

```
                    }
                    myCheck = 0;
                }
```

The integer temper will increase every loop until it finds a queue position that is empty, which would have a value of -2. If it finds an empty queue position, it will assign the process in question to that queue position, increase the tempCounter by 1, and leave the while-loop. It then resets all the values variable placeholders used so the process can repeat every iteration, which then will leave the other while loop and reset its placeholder variable.

That was all under the premise that the timer has gone on or past the arrival times of the processes. Now, if the outermost for-loop cycles again and the process' arrival time is greater than the current timer, meaning it hasn't arrived it, it will go through this else statement wherein it increments myRepos by 1 in which afterwards it checks if myRepos is equal to the number of processes.

```
                else{
                    // System.out.println("my empty???");
                    myRepos = myRepos + 1;
                    if (myRepos == numberOfProcesses) {
                        processesToRun[tempCounter]=-1;
                        tempCounter = numberOfProcesses+1;
                    }

                }

                myTurn=0;

            }
            myCheck = 0;
            myFit = 1;
            myRepos = 0;

        }

        myFit = 0;
        tempCounter = 0;
```

The reason myRepos checks if it's equal to the number of processes is that if the for-loop is notified that the processes haven't arrived yet, the processes' burst times are 0, or both, that means it will keep going through this else statement, therefore increasing myRepos. If myRepos is equal to the number of processes, this signifies that there are no processes to run, therefore making the program idle for a second. To make that happen, the first position in the queue array

will be assigned a value of -1, which signifies that the value in the queue position is not a process and it's not empty, but it's running and it has no process to run. Afterwards, it resets all placeholder variables leaving the for-loop, then leaves the while-loop as it also resets more placeholder variables.

After all the listing, sorting, and queuing, it's time for the printing process. It first gives the timeStamp variable the current time down to seconds. Then it checks which of the if-else statements it will go into. If the first position in the queue array is -1, which means there's no processes to run yet, it will display that the program is idle, which then the variable hasRan will be assigned a value of 1 which would indicate that the program has already started.

```
                timeStamp =
DateTimeFormatter.ofPattern("HH:mm:ss").format(LocalDateTime.now());
                if (processesToRun[0]==-1) {
                    System.out.println(timeStamp + "\tIdle... ");
                    hasRan=1;
                    qcounter = timeQuantum;
                    timerCounter++;
                    Thread.sleep(1000);
                }
```

Afterwards, qcounter will be given the value of timeQuantum which will help get out of the while-loop with the time quantum constraint. Then it'll increase the timerCounter by 1, which is essentially the counter for the iterations of the entire round robin, and lastly Thread.sleep(1000) will let the program pause for 1 second, then continue.

On the other hand, if the first position in the queue array is -2, which indicates an empty queue, it will first run a for-loop to see all processes with burst times of 0. This comes in hand with the error-handling if the user input 0 as a value for burst time for a process. If there is a process with a burst time of 0, it will assign 3 positions in the timeMarkers array with the process in the first position of the set, second position will have "No Burst Time", and the third position in the 3-position assignment will have 'No Duration." Afterwards, pcounter will increase by 1, which will indicate that a process has finished running. After the process is assigned in the timeMarkers array, it will also then get assigned to the completionTime array which will hold both the process and its time of completion, which in this case would be 0. The if-statement will then finish off by assigning the value of timeQuantum to qcounter so that the program will go out of the while-;pp[with the time quantum constraint.

```
                else if (processesToRun[0]==-2) {
                    for(int x = 0; x < numberOfProcesses; x++){
                        if(burstTime[x]==0){
                            timeMarkers[marker]=String.valueOf(x);
                            marker=marker+1;
                            timeMarkers[marker]="No Burst Time";
```

```
                        marker=marker+1;
                        timeMarkers[marker]="No Duration";
                        marker=marker+1;
                        pcounter++;
                        completionTime[ctHolder]= x;
                        ctHolder=ctHolder + 1;
                        completionTime[ctHolder]=0;
                        ctHolder=ctHolder + 1;
                        qcounter = timeQuantum;
                    }
                }
                if(hasRan==1){
                    System.out.println(timeStamp + "\tIdle... ");
                    qcounter = timeQuantum;
                    timerCounter++;
                    Thread.sleep(1000);
                }
            }
```

After the for-loop, it will check if the value of hasRan is equal to 1, which essentially means it checks if the program has officially started whether by running a process or being idle. If hasRan is 0, this means that the program hasn't started yet which is connected to an error handling when the first processes to run haven't been queued yet. If hasRan is 1, this means the program has started, which would then let the program enter this if statement wherein it'll print out "Idle..." and set the following variables accordingly which was how it was set up in the previous if statement when the queue position was equal to -1. The reason why there's a second idle condition is that in the middle of the program, when there are no processes to run, the queue array will be empty, this will then enter the else if statement where the first position in the queue array is -2, which would supposedly connect to the error-handling when the first processes to run haven't been queued yet. However, since it already ran and the program will go through this else if statement, the program will enter a secondary if statement which can then explicitly alert the user that it's in an idle state and will pause for 1 second.

Lastly, if the first queue position isn't empty or idle, this would mean that there is a process first in line ready to run. This would result in the program going through the else statement, where it sets the value of hasRan to 1, again to signify that the program has officially started, then it'll first check if the process' burst time has been fully used up. If the process has used up all its burst time, it will simply assign the timeQuantum value to qcounter and leave the else statement and the while-loop with the time quantum constraint. However, if the process' burst time isn't fully used up, it will enter the else statement wherein it'll go through a series of if statements.

```
            else{
                hasRan=1;
                if (Integer.parseInt(data[processesToRun[0]][1])==0){
```

```
                        qcounter = timeQuantum;
                    }
                    else{
```

The first if statement, being if(qcounter==0), is to check if the process being executed right now is being executed at time quantum 0, meaning it's the start time of the process within the time quantum. If it is, it will record the process in one position in the timeMarkers array, followed by the time stamp on the next position.

```
                        if (qcounter==0) {

timeMarkers[marker]=String.valueOf(processesToRun[0]);
                            marker=marker+1;
                            timeMarkers[marker]=timeStamp;
                            marker=marker+1;
                        }
```

The next if-else statement will check if the process being executed right now is being executed at the time quantum limitm meaning it's the end time of the process within the time quantum. If it is, it will record the time stamp on the following position, after the position in which the start time is in, in the timeMarkers array. This will complete a subset of the timeMarkers array wherein it would have the process, its start time, and its end time.

```
                        if ((qcounter+1)==timeQuantum) {
                            timeMarkers[marker]=timeStamp;
                            marker=marker+1;
                        }
                        else if
(Integer.parseInt(data[processesToRun[0]][1])-1 == 0){
                            timeMarkers[marker]=timeStamp;
                            marker=marker+1;
                        }
```

However, if the process is not running at the time quantum limit, and the process that's running has just used up its burst time, regardless of reaching the time quantum, it will record the time stamp of the process as its end time in the timeMarkers array, which then completes the subset also as mentioned earlier.

After the series of if statements, it will print out the process that's running, then assign the burst time of the process in question to the variable num, which will then assign that value minus 1 to the variable temp. The program will then get the string value of temp and assign it to the data 2d array in the column of burst times. What all of this means is that it subtracted 1 to the burst time since it ran for 1 second.

```
                        System.out.println(timeStamp + "\tRunning Process
" + data[processesToRun[0]][0] + "...");
                        num =
Integer.parseInt(data[processesToRun[0]][1]);
                        temp = num - 1;
                        data[processesToRun[0]][1] = String.valueOf(temp);
```

The program will once again encounter an if statement that checks if the burst time of that process is zero after the decrement before. If it is, it will increase the pcounter by 1, then assign one position in the completionTime array to the process that finished, and the next position in the array to the time it finished running totally.

```
                    if(Integer.parseInt(data[processesToRun[0]][1])==0){
                        pcounter++;
                        completionTime[ctHolder]= processesToRun[0];
                        ctHolder=ctHolder + 1;
                        completionTime[ctHolder]=timerCounter;
                        ctHolder=ctHolder + 1;
                    }
                    qcounter++;
                    timerCounter++;
                    Thread.sleep(1000);
                }
            }
        }
        timer.cancel();
```

Afterwards, it will increase the qcounter and timercounter, then pause the process for 1 second. If the entire round robin is done, it will exit the outermost while-loop and stop the timer.

We have this for loop that basically checks which has the most duration which will help in the printing process by finding out which has the most iterations of durations.

```
        for(int x = 0; x<numberOfProcesses; x++){
            for(int y = 0; y<100 ; y++){
                if(timeMarkers[y].equals(String.valueOf(x))){
                    lesserDur++;
                }
            }
```

```
            if (mostDur<lesserDur) {
                mostDur=lesserDur;
                lesserDur=0;
            }
        }
```

Next, it will go through a for-loop which is used to display the process together with its respective durations. It enters another for loop which will cycle through all the timeMarkers' positions. If the timeMarkers' position is equal to the process number in question, which comes from the outer for-loop, it will then print the next position in the timeMarker array for the start time, and the position after that for the end time, separated by a dash.

```
            for(int i=0; i<numberOfProcesses; i++){
                lesserDur=mostDur;
                System.out.print(data[i][0] + "\t");
                for(int j = 0; j<100; j++){
                    if(timeMarkers[j].equals(String.valueOf(i))){
                        if (lesserDur==mostDur) {
                            System.out.print(timeMarkers[j+1] + "-" +
timeMarkers[j+2]);
                            lesserDur = lesserDur - 1;
                        }
                        else{
                            System.out.print(",\t");
                            System.out.print(timeMarkers[j+1] + "-" +
timeMarkers[j+2]);
                            lesserDur = lesserDur - 1;
                        }

                    }
                }

                System.out.println();

            }
```

The lesserDur and mostDur variables are used just to print the durations with or without the comma.

The next printing process will look like this. It enters the first for-loop which cycles through all the processes again, which you can observe after the for statement, and it prints out the process number. Afterwards, it enters another for statement which will be used in cycling

through the completionTime array in increments of 2. This for-loop will determine the waiting time.

```
            for(int i=0; i<numberOfProcesses; i++){
                System.out.print(data[i][0] + "\t");
                for(int l = 0; l < (numberOfProcesses*2); l=l+2){
                    if(completionTime[l]==i){
                        if(burstTime[completionTime[l]]==0){
                            System.out.print("\t\t"+ 0);
                        }
                        else{
                            System.out.print("\t\t"+
(((completionTime[l+1])-(Integer.parseInt(data[completionTime[l]][2]))-bur
stTime[completionTime[l]])+1));
                            avgWT = avgWT +
(((completionTime[l+1])-(Integer.parseInt(data[completionTime[l]][2]))-bur
stTime[completionTime[l]])+1);
                        }
                    }
                }
```

The reason why it goes by increments of 2 is because every even numbered positions in the completionTime array contains the process numbers, which can be used in the next if statement wherein it checks if the process number in a certain position in the completionTime array matches the process in question, which is the number that's in the outer for-loop. If it matches, it will undergo another if-else statement wherein if the burst time of the process in question is equal to 0 in the first place, then it will simply print out 0. Otherwise, it will print out the difference of the process' completion time, arrival time, and burst time, then added by 1. Afterwards, it'll add that value to avgWT which will then loop after every process to get the total waiting time which will be used later for the average waiting time. Then the for-loop ends there.

Inside the same outer for-loop, there will be another for-loop which will determine the turnaround time. The for-loop also increments by 2 since the even numbered position in the burstTime array are process numbers while the odd numbered positions are the respective burst times of the process numbers prior to their positions in the burstTime array.

```
                for(int k = 0; k < (numberOfProcesses*2); k=k+2){
                    if(completionTime[k]==i){
                        if (burstTime[completionTime[k]]==0) {
                            System.out.print("\t\t\t\t"+ 0);
                        }
                        else{
                            System.out.print("\t\t\t\t"+
((completionTime[k+1])-(Integer.parseInt(data[completionTime[k]][2]))+1));
```

```
                                avgTaT = avgTaT +
((completionTime[k+1])-(Integer.parseInt(data[completionTime[k]][2]))+1);
                            }
                        }
                    }
                    System.out.println();
                }
```

The for-loop checks if a position in the completionTime is equal to the process in question. If it's equal, then it goes through another if-else statement. Same as before, if the burst time of the process in the first place is 0, then it will print out 0. Otherwise, it will print out the difference of the process' completion time and arrival time, then added by 1. Afterwards, it will add that value to the integer avgTaT which will be used initially as the total turnaround time, then used later to calculate the average turnaround time.

Then we have a simple equation here where the total turnaround time divided by the number of processes will be the average turnaround while the total average waiting time divided by the number of processes will be the average waiting time.

```
            System.out.println();
            avgTaT = avgTaT/numberOfProcesses;
            avgWT = avgWT/numberOfProcesses;
```

After printing the average turnaround time and average waiting time, the round robin function/algorithm officially ends.

IV. Error Handling

The error handling for this program includes if the user enters an invalid input, and will then allow the user to re-enter their desired values. The following are error handling features implemented in the code:

A. Menu
   1. Invalid Input - when the user enters something not within the choices, the program will ask the user to enter a valid input.
   2. Invalid file - when the user enters a file that doesn't exist, the program will ask the user to enter an existing file.
   3. Invalid file contents - the program checks if the text file can be processed.
      a) Invalid Input (shown in item B)
      b) Invalid header
      c) Invalid number of columns (for SJF and RR, must be at least 3, for PS must be 4)
      d) Invalid number of rows (must be at least 2)

B. InputData
   1. Invalid Input - the program checks if the text file is valid; must have the correct values.
      a) Character input
      b) Negative value input
      c) Decimal value input

C. Shortest-Job-First
    1. Invalid number of columns - the process information must include the process name, arrival time, and burst time.
D. Priority Scheduling
E. Round Robin
    1. Invalid input - the program checks if the user inputs a valid time quantum value.
        a) Character input
        b) Negative value input
        c) Decimal value input
        d) Integer 0 as input

V. Difficulties

As with any coding project, one would encounter many problems when faced with a programming challenge. Some of the difficulties faced is as follows:

1. Deciding which methods to use to implement queuing (arrays or linked lists)
2. Thinking of error handling possibilities and implementing solutions
3. Dealing with different arrival times
4. Taking the timestamps
5. Calculating the waiting time and arrival times
6. Validating values from the file
7. (And many more)

One of the hardest tasks in creating the algorithms for CPU scheduling is simulating how multiple processes will run in real time. There would be an added aspect in programming, the programmers had to take into account how time passes. In short, it's like coding in 2 dimensions, not just in 1. Adding to this, we had to take in account that **multiple** processes will do something at the same time in that dimension, either they are starting to arrive, or they are being run, or that they are finished, etc.

When it comes to implementing the round robin algorithm, the most challenging part was implementing the waiting queue since not only did we have to list down the processes needed to run, or the idle states, but we also had to consider the order since it requires First-In-First-Out algorithm. To add to that, when a process finishes running, or the program stops being in an idle state, the processes in the queue all need to be moved forward, therefore requiring a lot of conditional statements and loops.

Other small problems that were faced were just some technicality issues during coding. Such as how to save 2D array values to another 2D array without merely creating a pointer address, operations on variables, for and while loops conditions, passing variables, packages, dealing with localdateandtime variables, and so on and so forth.

VI. Conclusion

The programmers took a few days into planning and implementing the algorithms. The algorithms in and of itself was relatively difficult, and the hardest algorithm to implement was the Round-Robin method, however, is considered one of the most efficient algorithms for CPU scheduling. Adding to the fact that implementing these algorithms were already difficult, having to make the program user-friendly was already hard. The programmers had to rack their brain into making the algorithm accurate and also

making sure that the error handling covers most of the possible situations. Integrating everything together was difficult as well.

This machine project then effectively illustrates how operating systems concepts are significantly hard to implement in code. A relatively whole and complete operating system is exceedingly broad and includes an immense amount of features and methods (a book that tries to explain its concepts as briefly as it can and with minimal coding contains 1000+ pages). In comparison, these CPU scheduling algorithms could be a tiny speck in the galaxy of operating systems concepts algorithms. Consequently, it's humbling to know that the operating systems we use in everyday life (PCs, tablets, phones, appliances), is the result of months (probably years) of programming work and testing by tens (probably hundreds) of programmers. A regular consumer wouldn't have thought of how much work goes into making their device since the system in and of itself is already cohesive and intuitively easy to use.