FUNDAMENTALS OF OPERATING SYSTEMS

First Come First Served Algorithm Report

In my algorithm, I used Hash Map in an ArrayList data structure to put a key-value relationship between process id (PID), and their burst times (CPU, I/O times).

I use this file as an example to test my program:

P1:(2,5);(3,1);(4,6);(2,-1)

P2:(4,2);(6,10);(1,-1)

P3:(3,4);(1,2);(8,3);(4,-1)

I separated each line as a string, and each string has a key-value relationship in the hash map. For the first line, “P1” is a key and the rest (after colon) is a value which is in an ArrayList data structure. To split the times as CPU time and I/O time, I used two different arrays. One of them is for burst times (e.g., (2,5)), the other is for CPU and I/O times (e.g., 2 5).

```
List<HashMap<String, ArrayList<int[]>>> processArray = new ArrayList<>();

List<String> text = processes.lines().collect(Collectors.toList());

for(int i=0;i<text.size();i++){

    String[] parts = text.get(i).split(":");
    String PID = parts[0];
    String burstTimes = parts[1];

    HashMap<String, ArrayList<int[]>> processList = new HashMap<String, ArrayList<int[]> >();
    ArrayList<int[]> arr = new ArrayList<int[]>();
    String[] b = burstTimes.split(";");
    for (int j=0;j<b.length;j++){
        String[] aa = (b[j].split("[\\(\\)]")[1]).split(",");
        int[] bb = new int[2];
        for(int k=0;k<2;k++){
            bb[k] = Integer.parseInt(aa[k]);
            if(bb[k]<0) bb[k]=0;
        }
        arr.add(bb);
    }

    processList.put(PID, arr);
    processArray.add(processList);
```

Each process has different return time value. So, I created an array for different return times as large as processArray.size(). In the beginning, time and return time is zero.

```
int[] returnTimes = new int[processArray.size()];
Arrays.fill(returnTimes,0);

int time = 0;
int index = 0;
int[] bb = null;
int[] cpuTimes = new int[processArray.size()];
```

I used two main methods for my program. One of them is “firstNotNull” to find a burst time which is not null yet. (Because, when the program used a burst time (e.g., (2.5)), then it set “null” for that burst time.):

```
public static int[] firstNotNull(List<HashMap<String, ArrayList<int[]>>> processArray){
    for(int index =0;index<processArray.size();index++){
        for(int i=0;i<processArray.get(index).get("P"+(index+1)).size();i++){
            if(processArray.get((index)).get("P" + (index + 1)).get(i)!=null){
                int[] arr = new int[2];
                arr[0] = index;
                arr[1] = i;
                return arr;
            }
        }
    }
    return null;
}
```

The other method is “getProcIndex” to determine the index of my processArray to use. I set “null” value to processes after that method is used:

```
public static int getProcIndex(List<HashMap<String, ArrayList<int[]>>> processArray,int index){
    for(int i=0;i<processArray.get(index).get("P"+(index+1)).size();i++){
        if(processArray.get((index)).get("P" + (index + 1)).get(i)!=null){
            return i;
        }
    }

    return 0;
}
```

After I created these kind of arrays or methods, there must be a loop until processArray is null. Actually, everything is here, inside a while loop. I control if there is a burst time which is not null **and** its CPU time is smaller than or equal to the current “time”, that was zero in the beginning. If a process is under that “if statement”, then the current time and burst time is going to change. If not, I increase “index” to find another process that providing this condition; so, to save them from loop. Here is my “while” loop and “if” statement to do all of these:

```
while(firstNotNull(processArray)!=null){

    if(returnTimes[index]<=time && firstNotNull(processArray)!=null){

        int i = getProcIndex(processArray,index);

        int[] procceses = processArray.get(index).get("P" + (index + 1)).get(i);
        processArray.get((index)).get("P" + (index + 1)).set(i, null);

        cpuTimes[index] += procceses[0];
        int burst;
        if(procceses[0] < 0) burst =0;
        else burst = procceses[0];

        returnTimes[index] = time + burst + procceses[1];
        System.out.print(time+"\t"+"P"+(index+1)+"\t"+"("+procceses[0]+","+(procceses[1]==0 ? -1:procceses[1])+")"+"\t");
        if(firstNotNull(processArray)!=null) {
            time += burst ;

        }
```

If, there is no process that provides these conditions, then I must increase the “time” until a process is found:

```
else {

    if(index!=processArray.size()-1){

        index++;
    }
    else {
        time++;
        index = 0;
    }
}
```

After all of these transactions, I should calculate waiting times and turnaround times for each process, and the average of them:

```
double awg = 0;
double att = 0;
for(int i=0; i<processArray.size(); i++){
    System.out.println("Waiting time (P"+(i+1)+ ") :\t "+ (returnTimes[i] - cpuTimes[i]));
    System.out.println("Turnaround time (P"+(i+1)+ ") :\t " + (returnTimes[i]));
    System.out.println();
    awg += (returnTimes[i] - cpuTimes[i]);
    att += (returnTimes[i]);
}
System.out.println("AVERAGE WAITING TIME :\t" +(awg/processArray.size()));
System.out.println("AVERAGE TURNAROUND TIME :\t" +(att/processArray.size()));
```

My output for my example text file (in the begging of this document) is:

```
Time    PID     Burst   Return time
0       P1      (2,5)   7
2       P2      (4,2)   8
6       P3      (3,4)   13
9       P1      (3,1)   13
12      P2      (6,10)  28
18      P1      (4,6)   28
22      P3      (1,2)   25
25      P3      (8,3)   36
33      P1      (2,-1)  35
35      P2      (1,-1)  36
36      P3      (4,-1)  40

Total time :     36

Waiting time (P1) :      24
Turnaround time (P1) :   35

Waiting time (P2) :      25
Turnaround time (P2) :   36

Waiting time (P3) :      24
Turnaround time (P3) :   40

AVERAGE WAITING TIME :  24.333333333333332
AVERAGE TURNAROUND TIME :       37.0
```