EECS 2011 Assignment 2

System for Deterministic Modeling of CPU Scheduling Algorithms

Date: November 8, 2017

Course: EECS 2011

|  |  |  |  |
| --- | --- | --- | --- |
| No | Name | Student ID | EECS ID |
| 1 | Haseeb Ahmad | 213579891 | ahmadhas |
| 2 | Jason Skinner | 215115678 | skinner1 |

Report

CPU scheduling is nothing but picking up the process from the ready state and sending it to the CPU. The short-term scheduler and the dispatcher (sometimes part of the scheduler) do this task from ready state to running state. The process can move from running to termination (process completed), or to wait state or ready state (ready for the new process). Now there are many ways to assess the situation of the process. For example, if the process the priority queue algorithm is used, then we set the priority of the process and send to queue. Whichever has the higher priority, it’s state change from to ready to be running or waiting to running (if the process has already been started.)

For this assignment, first we needed to create processes and define their states. We made a processes class, which defines what each process has and the states are defined using ENUM. The Test class creates an ArrayList of the processes to test for the algorithms.

The First-come, First-serve algorithm goes through all the processes until all the processes are done. It will print the wait time and start time and increment time and process time for each process CPU Time. The process gets removed as it outputs the data of the process.

The Shortest-Job First gets the index of the process which has the shortest time, then it will go through all the processes and checks if the process time is less than the time of the shortest process time, then increment the time and process time, it will at the end remove that process and again gets the index of the process which has the shortest time.

In the Priority Scheduling, we first had to create a queue, in which has a process is added if the priority of that process is greater than all other, also we had to check for whether it is done and whether it is CPU or IO process as well.

Round-Robin Scheduling goes through all the processes and whatever the process is it will execute that process for a particular amount of time and after the execution, it will pre-empt it and onto the next process.

Multi-Level Scheduling combines all the above scheduling and gives the priority to the highest-level process. For example, the highest-level process is System Process (using FCFS) and then Intermediate (SJF) and then Round Robin and finally lowest-level process is using priority. In this case System process will be executed first and then the next and so on. The advantage in this is that for various types of processes, we can apply various types of scheduling.

Multi-Level Feedback Scheduling is very similar to Multi-Level Scheduling, but the difference is that the process at the lowest-level can be moved to a higher priority and will not be executing for a long time. The implementation of it can be done by checking which process has important and which is not and then move the priority of that process and which algorithm to use.

Testing was done with random processes of random lengths. The processes were created once the Algorithm class was initiated. Each time the algorithm run a new set of processes are created and analysed. These processes were tested and results were compared by calculations by hand.

NonpreemptiveFCFS.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

//runs the algorithms on the processes and ouputs the Wait times etc for each process

//based on First Come first serve

public class NonpreemptiveFCFS implements Algorithms {

List<Processes> processes = new ArrayList<Processes>();

long totalWaitTime = 0;

int numberOfProcesses = 0;

public void runAlgorithm() {

long time = 0;

long processTime;

Test.createProcesses();

processes = Test.getProcesses();

while (!Test.allProcessDone()) {

processTime = 0;

System.out.println("Wait time was: " +waitTime(time));

System.out.println("StartTime: " + time);

while (processTime < processes.get(0).getTotalTime()) {

processTime++;

time++;

}

System.out.println("EndTime: " + time);

System.out.println("");

processes.remove(0);

}

System.out.println("");

System.out.println("Average Wait Time was: " + averageWait());

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(0).getQueuedTime());

processes.get(0).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

NonpreemptiveFCFS start = new NonpreemptiveFCFS();

start.runAlgorithm();

}

}

NonpreemptiveSJF.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

//runs the algorithms on the processes and ouputs the Wait times etc for each process

// based on Shorted Job First

public class NonpreemptiveSJF implements Algorithms {

List<Processes> processes = new ArrayList<Processes>();

long totalWaitTime = 0;

int shortestJobIndex = 0;

int numberOfProcesses = 0;

public void runAlgorithm() {

long time = 0;

long processTime;

Test.createProcesses();

processes = Test.getProcesses();

while (!Test.allProcessDone()) {

processTime = 0;

System.out.println(shortestJobIndex);

System.out.println(processes.get(shortestJobIndex).getTotalTime());

System.out.println("Wait time was: " + waitTime(time));

System.out.println("StartTime: " + time);

while (processTime < processes.get(shortestJobIndex).getTotalTime()) {

processTime++;

time++;

}

System.out.println("EndTime: " + time);

System.out.println("");

processes.remove(shortestJobIndex);

if (!Test.allProcessDone()) {

shortestJob();

}

}

System.out.println("");

System.out.println("Average Wait Time was: " + averageWait());

}

public void shortestJob() {

shortestJobIndex = 0;

int shortTime = processes.get(0).getTotalTime();

for (int i = 1; i < processes.size(); i++) {

if (processes.get(i).getTotalTime() < shortTime) {

shortTime = processes.get(i).getTotalTime();

shortestJobIndex = i;

}

}

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(shortestJobIndex).getQueuedTime());

processes.get(shortestJobIndex).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

NonpreemptiveSJF start = new NonpreemptiveSJF();

start.runAlgorithm();

}

}

NonpreemptivePriority.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

//runs the algorithms on the processes and ouputs the Wait times etc for each process

//based on Priority

public class NonpreemptivePriority implements Algorithms{

List<Processes> processes = new ArrayList<Processes>();

long totalWaitTime = 0;

int priorityJobIndex = 0;

int numberOfProcesses = 0;

public void runAlgorithm() {

long time = 0;

long processTime;

Test.createProcesses();

processes = Test.getProcesses();

while (!Test.allProcessDone()) {

processTime = 0;

System.out.println(priorityJobIndex);

System.out.println(processes.get(priorityJobIndex).getProcessPriority());

System.out.println("Wait time was: " + waitTime(time));

System.out.println("StartTime: " + time);

while (processTime < processes.get(priorityJobIndex).getTotalTime()) {

processTime++;

time++;

}

System.out.println("EndTime: " + time);

System.out.println("");

processes.remove(priorityJobIndex);

if (!Test.allProcessDone()) {

shortestPriorityJob();

}

}

System.out.println("");

System.out.println("Average Wait Time was: " + averageWait());

}

public void shortestPriorityJob() {

priorityJobIndex = 0;

int shortPriority = processes.get(0).getProcessPriority();

for (int i = 1; i < processes.size(); i++) {

if (processes.get(i).getProcessPriority() > shortPriority) {

shortPriority = processes.get(i).getProcessPriority();

priorityJobIndex = i;

}

}

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(priorityJobIndex).getQueuedTime());

processes.get(priorityJobIndex).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

NonpreemptivePriority start = new NonpreemptivePriority();

start.runAlgorithm();

}

}

PreemptivePriority.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class PreemptivePriority implements Algorithms {

List<Processes> processes = new ArrayList<Processes>();

List<PriorityJobs> jobs = new ArrayList<PriorityJobs>();

long totalWaitTime = 0;

int priorityJobIndex = 0;

int numberOfProcesses = 0;

int IO\_Counter = 0;

long time = 0;

long IOtime = 0;

long processTime;

int CPUprocessID;

int CPUID;

int IOprocessID;

int IOID;

public void runAlgorithm() {

Test.createProcesses();

processes = Test.getProcesses();

CPUprocessID = 0;

CPUID = 0;

IOprocessID = -1;

IOID = 0;

List<Processes> queueCPU = new ArrayList<Processes>();

List<Processes> queueIO = new ArrayList<Processes>();

int i = 0;

Processes p = processes.get(0);

while (0 < p.getCPU\_time().size()) {

processTime = 0;

p = processes.get(0);

System.out.println("SIZE : " + p.getCPU\_time().size());

queueCPU.add(p);

while (queueCPU.size() > 0) {

for (int j = 0; j < queueCPU.get(0).getCPU\_time().size(); j++) {

if (!(queueCPU.get(0).getCPU\_time().get(j).isDone())

&& queueCPU.get(0).getCPU\_time().get(j).getStr() == PType.CPU\_time) {

i = j;

break;

}

}

queueCPU.get(0).getCPU\_time().get(i).setTime(-1);

time++;

if (queueCPU.get(0).getCPU\_time().get(i).getTime()==0) {

queueCPU.get(0).getCPU\_time().get(i).setDone(true);

break;

// send to IO

}

break;

}

System.out.println(0);

processes.remove(0);

i++;

if (Test.allProcessDone()) {

break;

}

}

System.out.println("");

// System.out.println("Average Wait Time was: " + averageWait());

}

public void checkJobDone() {

int done = 0;

for (int i = 0; i < processes.get(priorityJobIndex).getCPU\_time().size(); i++) {

if (!(processes.get(priorityJobIndex).getCPU\_time().get(i).isDone())) {

done = 1;

break;

}

}

if (done != 0) {

processes.get(priorityJobIndex).setState(State.Ready);

} else {

processes.get(priorityJobIndex).setState(State.Terminated);

}

}

public int getNextPriorityReady() {

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

if (processes.get(jobs.get(i).getJobNo()).getState() == State.Ready) {

return jobs.get(i).getJobNo();

}

}

return 0;

}

public void shortestPriorityJob() {

jobs.clear();

System.out.println(processes.size());

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

jobs.add(new PriorityJobs(processes.get(i).getProcessPriority(), i));

}

Collections.sort(jobs);

priorityJobIndex = jobs.get(0).getJobNo();

jobs.remove(0);

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(priorityJobIndex).getQueuedTime());

processes.get(priorityJobIndex).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

PreemptivePriority start = new PreemptivePriority();

start.runAlgorithm();

}

}

PreemptiveRoundRobin.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

public class PreemptiveRoundRobin implements Algorithms {

List<Processes> processes = new ArrayList<Processes>();

long totalWaitTime = 0;

int numberOfProcesses = 0;

public void runAlgorithm(int burstTime) {

long time = 0;

long processTime;

Test.createProcesses();

processes = Test.getProcesses();

while (!Test.allProcessDone()) {

processTime = 0;

System.out.println("Wait time was: " +waitTime(time));

System.out.println("StartTime: " + time);

while (processTime < processes.get(0).getTotalTime()) {

processTime++;

time++;

}

System.out.println("EndTime: " + time);

System.out.println("");

processes.remove(0);

}

System.out.println("");

System.out.println("Average Wait Time was: " + averageWait());

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(0).getQueuedTime());

processes.get(0).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

PreemptiveRoundRobin start = new PreemptiveRoundRobin();

start.runAlgorithm(4);

}

@Override

public void runAlgorithm() {

; // Intentionally left blank

}

}

PreemptiveSJF.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

public class PreemptiveSJF implements Algorithms {

List<Processes> processes = new ArrayList<Processes>();

long totalWaitTime = 0;

int shortestJobIndex = 0;

int numberOfProcesses = 0;

public void runAlgorithm() {

long time = 0;

long processTime;

Test.createProcesses();

processes = Test.getProcesses();

while (!Test.allProcessDone()) {

processTime = 0;

System.out.println(shortestJobIndex);

System.out.println(processes.get(shortestJobIndex).getTotalTime());

System.out.println("Wait time was: " + waitTime(time));

System.out.println("StartTime: " + time);

while (processTime < processes.get(shortestJobIndex).getTotalTime()) {

processTime++;

time++;

}

System.out.println("EndTime: " + time);

System.out.println("");

processes.remove(shortestJobIndex);

if (!Test.allProcessDone()) {

shortestJob();

}

}

System.out.println("");

System.out.println("Average Wait Time was: " + averageWait());

}

public void shortestJob() {

shortestJobIndex = 0;

int shortTime = processes.get(0).getTotalTime();

for (int i = 1; i < processes.size(); i++) {

if (processes.get(i).getTotalTime() < shortTime) {

shortTime = processes.get(i).getTotalTime();

shortestJobIndex = i;

}

}

}

@Override

public long waitTime(long time) {

long waitTime = (time - processes.get(shortestJobIndex).getQueuedTime());

processes.get(shortestJobIndex).setWaitTime(waitTime);

totalWaitTime += waitTime;

numberOfProcesses++;

return waitTime;

}

@Override

public long averageWait() {

return (long) (totalWaitTime / numberOfProcesses);

}

public static void main(String[] args) {

PreemptiveSJF start = new PreemptiveSJF();

start.runAlgorithm();

}

}

Processes.java

package eecs2011a2;

import java.util.ArrayList;

import java.util.List;

//creates the processes and runs different methods on them

public class Processes {

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

private List<TimePType> CPU\_time = new ArrayList<TimePType>();

private int processPriority = 1;

private State state = State.Ready;

/\*\*\*\*\*\* Needed for calculations \*\*\*\*\*\*/

private long queuedTime;

private long waitTime;

private long currentIOWait;

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public Processes(int priority, State state, long queuedTime) {

this.processPriority = priority;

this.state = state;

this.queuedTime = queuedTime;

}

public long getWaitTime() {

return waitTime;

}

public void setWaitTime(long waitTime) {

this.waitTime = waitTime;

}

public long getQueuedTime() {

return queuedTime;

}

public void setState(State state) {

this.state = state;

}

public State getState() {

return state;

}

public List<TimePType> getCPU\_time() {

return CPU\_time;

}

public TimePType removeFirst() {

TimePType temp = CPU\_time.get(0);

CPU\_time.remove(0);

return temp;

}

public int getTotalTime() {

int time = 0;

for (int i = 0; i < CPU\_time.size(); i++) {

time += CPU\_time.get(i).getTime();

}

return time;

}

public int getTotalCPUTime() {

int time = 0;

for (int i = 0; i < CPU\_time.size(); i++) {

if (CPU\_time.get(i).getStr() == PType.CPU\_time) {

time += CPU\_time.get(i).getTime();

}

}

return time;

}

public int getTotalIOTime() {

int time = 0;

for (int i = 0; i < CPU\_time.size(); i++) {

if (CPU\_time.get(i).getStr() == PType.IO\_time) {

time += CPU\_time.get(i).getTime();

}

}

return time;

}

public TimePType removeLast() {

TimePType temp = CPU\_time.get(CPU\_time.size() - 1);

CPU\_time.remove(CPU\_time.size() - 1);

return temp;

}

public void addLast(TimePType timeStr) {

CPU\_time.add(timeStr);

}

public void run() {

state = State.Running;

}

public boolean isRunning() {

return (state == State.Running);

}

public int getProcessPriority() {

return processPriority;

}

public void setProcessPriority(int processPriority) {

this.processPriority = processPriority;

}

public static void main(String[] args) {

Test.createProcesses();

}

public long getCurrentIOWait() {

return currentIOWait;

}

public void setCurrentIOWait(long currentIOWait) {

this.currentIOWait = currentIOWait;

}

public boolean doneCPU() {

for (int i =0 ; i<CPU\_time.size(); i++) {

if (!CPU\_time.get(i).isDone()){

return false;

}

}

return true;

}

}

PriorityJobs.java

package eecs2011a2;

// uses the processes to evaluate the priority and puts them in order

public class PriorityJobs implements Comparable<Object>{

private int priority;

private int jobNo;

public PriorityJobs(int priority,int jobNo){

this.setPriority(priority);

this.setJobNo(jobNo);

}

public int getPriority() {

return priority;

}

public void setPriority(int priority) {

this.priority = priority;

}

public int getJobNo() {

return jobNo;

}

public void setJobNo(int jobNo) {

this.jobNo = jobNo;

}

public String toString(){

return "P: = " + priority + " J: " + jobNo;

}

@Override

public int compareTo(Object o) {

PriorityJobs oth = (PriorityJobs) o;

return oth.priority - this.priority;

}

}

PType.java

package eecs2011a2;

// used for testing giving each process some types

public enum PType {

*CPU\_time*,

*IO\_time*

}

State.java

package eecs2011a2;

//used for testing giving the process the running state

public enum State {

*Running*, *New*, *Ready*, *Waiting\_for\_IO*, *Paused*, *Terminated*

}

Test.java

package eecs2011a2;

//creates the test processes randomly

import java.util.ArrayList;

import java.util.List;

public class Test {

static List<Processes> processes = new ArrayList<Processes>();

public static void createProcesses() {

for (int j = 0; j < 5; j++) {

processes.add(new Processes((int) (Math.random() \* 99) + 1, State.New, 10 \* j)); PType p;

System.out.println("Start of Process " + j);

for (int i = 0; i < (Math.random() \* 5) + 1; i++) {

p = PType.CPU\_time;

TimePType e = new TimePType(((int) (Math.random() \* 100)), p);

processes.get(j).addLast(e);

p = PType.IO\_time;

e = new TimePType(((int) (Math.random() \* 100)), p);

processes.get(j).addLast(e);

}

for (int i = 0; i < processes.get(j).getCPU\_time().size(); i++) {

System.out.println(processes.get(j).getCPU\_time().get(i));

}

System.out.println("Total Time : " + processes.get(j).getTotalTime());

System.out.println("Priority of : " + processes.get(j).getProcessPriority());

System.out.println("End of Process " + j);

System.out.println("");

}

}

public static List<Processes> getProcesses() {

return processes;

}

public static boolean allProcessDone() {

boolean done = true;

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

if (processes.get(i).getState() != State.Terminated) {

done = false;

break;

}

}

return done;

}

}

TimePType.java

package eecs2011a2;

//igve the time and type for each cpu/io cycle

public class TimePType {

private int time;

private PType str;

private boolean done = false;

public TimePType(int t, PType s) {

time = t;

str = s;

}

public PType getStr() {

return str;

}

public void setStr(PType str) {

this.str = str;

}

public boolean isDone() {

return done;

}

public void setDone(boolean done) {

this.done = done;

}

public int getTime() {

return time;

}

public void setTime(int time) {

this.time = time;

}

public String toString() {

return "time: " + time + " str: " + str;

}

}

Algorithms.java

package eecs2011a2;

public interface Algorithms {

public long waitTime(long time);

public long averageWait();

public void runAlgorithm();

}

TEST\_OUTPUT

Non Preemptive FCFS - We created random processes and the class computed the averages

Start of Process 0

time: 68 str: CPU\_time

time: 54 str: IO\_time

time: 70 str: CPU\_time

time: 53 str: IO\_time

time: 13 str: CPU\_time

time: 97 str: IO\_time

time: 29 str: CPU\_time

time: 39 str: IO\_time

Total Time : 423

Priority of : 11

End of Process 0

Start of Process 1

time: 10 str: CPU\_time

time: 70 str: IO\_time

time: 33 str: CPU\_time

time: 23 str: IO\_time

time: 23 str: CPU\_time

time: 79 str: IO\_time

time: 50 str: CPU\_time

time: 65 str: IO\_time

time: 13 str: CPU\_time

time: 66 str: IO\_time

time: 84 str: CPU\_time

time: 90 str: IO\_time

Total Time : 606

Priority of : 16

End of Process 1

Start of Process 2

time: 17 str: CPU\_time

time: 59 str: IO\_time

time: 47 str: CPU\_time

time: 82 str: IO\_time

time: 72 str: CPU\_time

time: 35 str: IO\_time

time: 47 str: CPU\_time

time: 49 str: IO\_time

Total Time : 408

Priority of : 39

End of Process 2

Start of Process 3

time: 14 str: CPU\_time

time: 90 str: IO\_time

time: 91 str: CPU\_time

time: 43 str: IO\_time

time: 4 str: CPU\_time

time: 15 str: IO\_time

time: 97 str: CPU\_time

time: 32 str: IO\_time

Total Time : 386

Priority of : 94

End of Process 3

Start of Process 4

time: 6 str: CPU\_time

time: 76 str: IO\_time

time: 39 str: CPU\_time

time: 76 str: IO\_time

time: 91 str: CPU\_time

time: 11 str: IO\_time

time: 97 str: CPU\_time

time: 39 str: IO\_time

time: 71 str: CPU\_time

time: 25 str: IO\_time

Total Time : 531

Priority of : 88

End of Process 4

Wait time was: 0

StartTime: 0

EndTime: 423

Wait time was: 413

StartTime: 423

EndTime: 1029

Wait time was: 1009

StartTime: 1029

EndTime: 1437

Wait time was: 1407

StartTime: 1437

EndTime: 1823

Wait time was: 1783

StartTime: 1823

EndTime: 2354

Average Wait Time was: 922

Nonpreemptive Priority - we created some random test cases and produced the output.

Start of Process 0

time: 63 str: CPU\_time

time: 9 str: IO\_time

time: 10 str: CPU\_time

time: 49 str: IO\_time

time: 82 str: CPU\_time

time: 75 str: IO\_time

time: 72 str: CPU\_time

time: 68 str: IO\_time

Total Time : 428

Priority of : 38

End of Process 0

Start of Process 1

time: 31 str: CPU\_time

time: 17 str: IO\_time

time: 11 str: CPU\_time

time: 44 str: IO\_time

time: 9 str: CPU\_time

time: 4 str: IO\_time

time: 85 str: CPU\_time

time: 31 str: IO\_time

Total Time : 232

Priority of : 85

End of Process 1

Start of Process 2

time: 1 str: CPU\_time

time: 84 str: IO\_time

time: 41 str: CPU\_time

time: 59 str: IO\_time

time: 67 str: CPU\_time

time: 54 str: IO\_time

time: 17 str: CPU\_time

time: 42 str: IO\_time

time: 35 str: CPU\_time

time: 6 str: IO\_time

Total Time : 406

Priority of : 2

End of Process 2

Start of Process 3

time: 61 str: CPU\_time

time: 33 str: IO\_time

time: 28 str: CPU\_time

time: 57 str: IO\_time

Total Time : 179

Priority of : 40

End of Process 3

Start of Process 4

time: 76 str: CPU\_time

time: 90 str: IO\_time

time: 2 str: CPU\_time

time: 8 str: IO\_time

Total Time : 176

Priority of : 8

End of Process 4

0

38

Wait time was: 0

StartTime: 0

EndTime: 428

0

85

Wait time was: 418

StartTime: 428

EndTime: 660

1

40

Wait time was: 630

StartTime: 660

EndTime: 839

1

8

Wait time was: 799

StartTime: 839

EndTime: 1015

0

2

Wait time was: 995

StartTime: 1015

EndTime: 1421

Average Wait Time was: 568

Nonpreemptive SJF - we created some random test cases and produced the output.

Start of Process 0

time: 78 str: CPU\_time

time: 96 str: IO\_time

time: 76 str: CPU\_time

time: 69 str: IO\_time

time: 45 str: CPU\_time

time: 25 str: IO\_time

Total Time : 389

Priority of : 11

End of Process 0

Start of Process 1

time: 72 str: CPU\_time

time: 69 str: IO\_time

time: 49 str: CPU\_time

time: 89 str: IO\_time

time: 15 str: CPU\_time

time: 15 str: IO\_time

time: 86 str: CPU\_time

time: 97 str: IO\_time

Total Time : 492

Priority of : 33

End of Process 1

Start of Process 2

time: 99 str: CPU\_time

time: 44 str: IO\_time

time: 58 str: CPU\_time

time: 84 str: IO\_time

time: 70 str: CPU\_time

time: 36 str: IO\_time

Total Time : 391

Priority of : 86

End of Process 2

Start of Process 3

time: 93 str: CPU\_time

time: 55 str: IO\_time

time: 59 str: CPU\_time

time: 18 str: IO\_time

time: 2 str: CPU\_time

time: 85 str: IO\_time

time: 7 str: CPU\_time

time: 63 str: IO\_time

time: 20 str: CPU\_time

time: 41 str: IO\_time

Total Time : 443

Priority of : 36

End of Process 3

Start of Process 4

time: 57 str: CPU\_time

time: 28 str: IO\_time

time: 93 str: CPU\_time

time: 70 str: IO\_time

time: 41 str: CPU\_time

time: 28 str: IO\_time

time: 27 str: CPU\_time

time: 43 str: IO\_time

Total Time : 387

Priority of : 50

End of Process 4

0

389

Wait time was: 0

StartTime: 0

EndTime: 389

3

387

Wait time was: 349

StartTime: 389

EndTime: 776

1

391

Wait time was: 756

StartTime: 776

EndTime: 1167

1

443

Wait time was: 1137

StartTime: 1167

EndTime: 1610

0

492

Wait time was: 1600

StartTime: 1610

EndTime: 2102

Average Wait Time was: 768

README

When you compile the files make sure that you have included the all the attached files pertaining to the Modeling Algorithm. Each algorithm must be compiled together with: Algorithms.java, Processes.java, PType.java, State.java, Test.java, TimePTpe.java, and PriorityJobs.java

These files are required when compiling the CPU Scheduling Algorithm.

OR

Clone in Eclipse © using https://github.com/jskinner65/EECS2011a2