OS Scheduling Algorithm

Varun Subramanya

01705578

**Scheduling Algorithms:**

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. These algorithms are either non-preemptive or preemptive. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

**First Come First Serve:**

* Jobs are executed on first come, first serve basis.
* It is a non-preemptive, pre-emptive scheduling algorithm.
* Easy to understand and implement.
* Its implementation is based on FIFO queue.
* Poor in performance as average wait time is high.

**SJF Algorithm**

* This is also known as **shortest job first**, or SJF
* This is a non-preemptive, pre-emptive scheduling algorithm.
* Best approach to minimize waiting time.
* Easy to implement in Batch systems where required CPU time is known in advance.
* Impossible to implement in interactive systems where required CPU time is not known.
* The processer should know in advance how much time process will take.

**Round Robin**

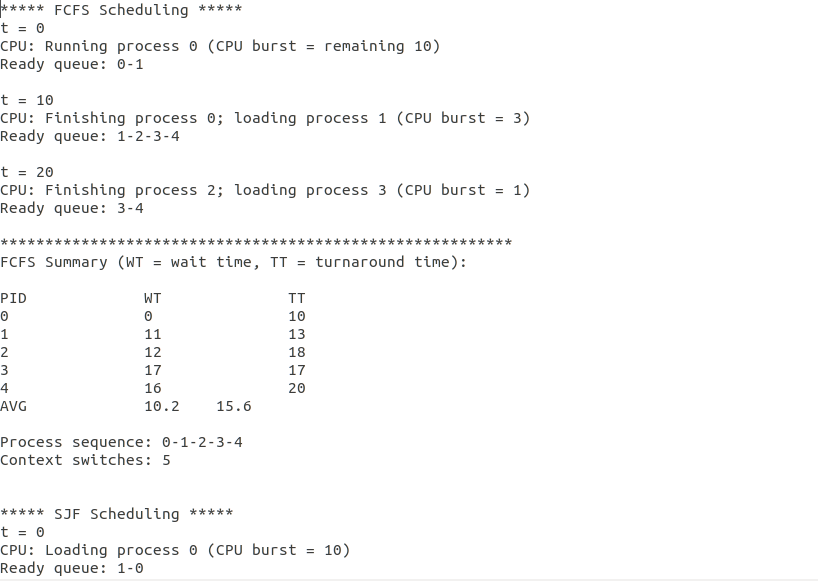
* Round Robin is the preemptive process scheduling algorithm.
* Each process is provided a fix time to execute, it is called a **quantum**.
* Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
* Context switching is used to save states of preempted processes.

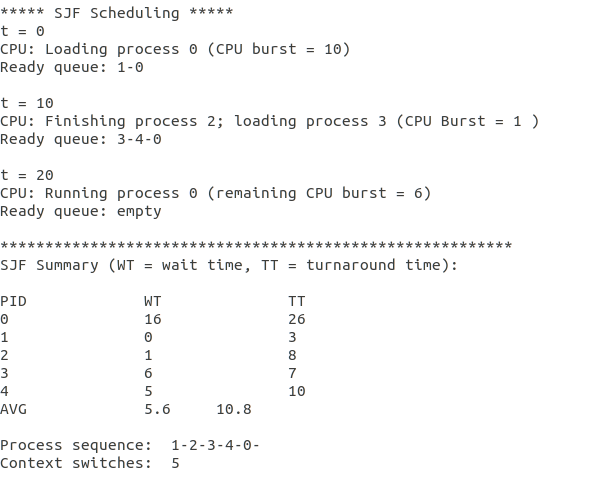
**STCF**:

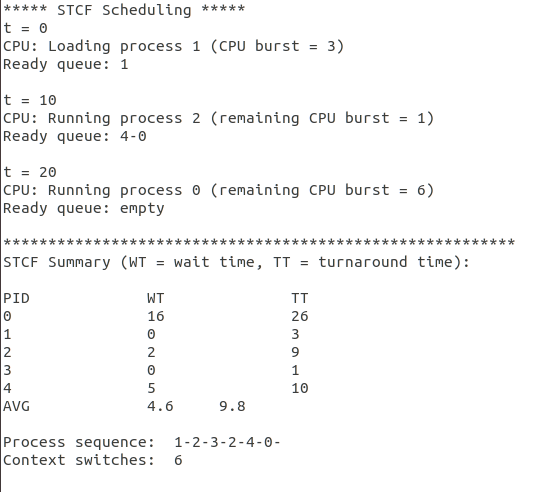
In this scheduling algorithm, the [process](https://en.wikipedia.org/wiki/Process_(computing)) with the smallest amount of time remaining until completion is selected to execute. Since the currently executing process is the one with the shortest amount of time remaining by definition, and since that time should only reduce as execution progresses, processes will always run until they complete or a new process is added that requires a smaller amount of time.

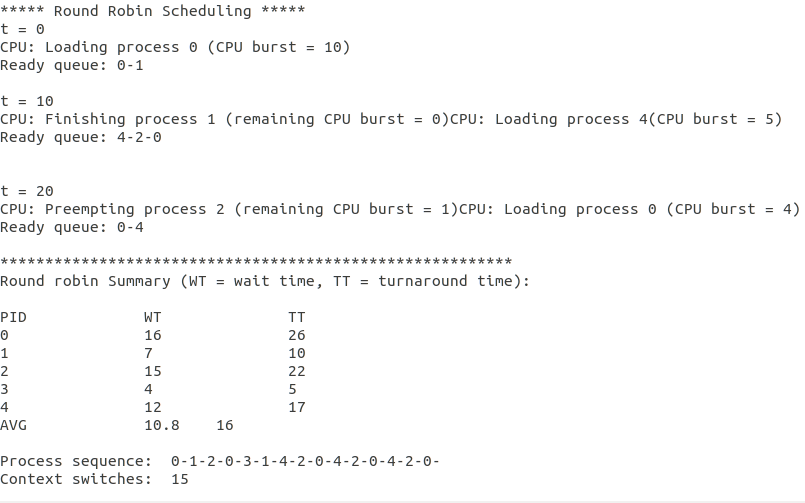
Shortest remaining time is advantageous because short processes are handled very quickly. The system also requires very little overhead since it only makes a decision when a process completes or a new process is added, and when a new process is added the algorithm only needs to compare the currently executing process with the new process, ignoring all other processes currently waiting to execute

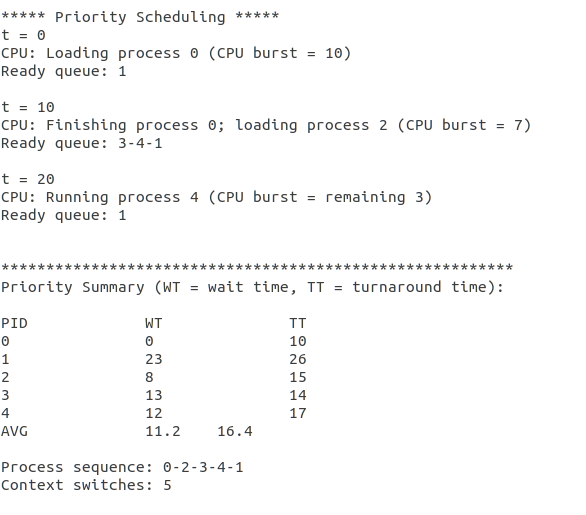
**Program outputs:**

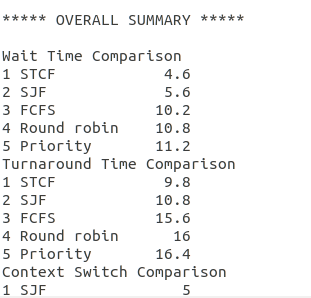












**Appendix:**

**Code :**

/\*

Varun Subramanya

OS\_Program\_3.cpp

\*/

#include "sched\_sim.hpp"

using namespace std;

int main(int argc, char\* argv[]) {

// array for other alg stats

float stats[6];

int context\_switch = 0;

Q\_struct Queue;

Summary Sum\_Timings;

int numProcesses = 0;

vector<Process> processFCFS;

vector<Process> processSJF;

vector<Process> processSTCF;

vector<Process> processRoundRobin;

vector<Process> processPriority;

// -- get user's input arguments

char \*inputFile = argv[1];

char \*outputFile = argv[2];

int SnapshotStep = atoi(argv[3]); // time interval between printing snapshots

// variables to hold process info

int CPUburst, priority, arrivalTime;

string line;

// create a new file stream to loop through file

ifstream input;

input.open(inputFile, fstream::in);

// loop through the file and get text line by line

while (getline(input, line)) {

istringstream iss(line);

if (!(iss >> CPUburst >> priority >> arrivalTime)) {

// if we're here something went wrong

break;

}

// create the new object and push it

processFCFS.push\_back(Process(numProcesses, CPUburst, priority, arrivalTime));

processSJF.push\_back(Process(numProcesses, CPUburst, priority, arrivalTime));

processSTCF.push\_back(Process(numProcesses, CPUburst, priority, arrivalTime));

processRoundRobin.push\_back(Process(numProcesses, CPUburst, priority, arrivalTime));

processPriority.push\_back(Process(numProcesses, CPUburst, priority, arrivalTime));

// increment the counter

numProcesses++;

}

int Buffer\_Burst[numProcesses]; // temporary copy from burst timmings

for(int t=0; t<numProcesses; t++) Buffer\_Burst[t] = processFCFS[t].CPUburst; // coping in the temp buffer

// ---- Scheduling ALgorithms ----------

runFCFS(processFCFS, SnapshotStep, numProcesses, outputFile, stats); //FCFS Algorithm

SJF(&Queue, processSJF, numProcesses, outputFile, Buffer\_Burst, SnapshotStep, &Sum\_Timings); // SJF Algorithm

STCF(&Queue, processSTCF, numProcesses, outputFile, Buffer\_Burst, SnapshotStep, &Sum\_Timings); // STCF Algorithm

RoundRobin(&Queue, processRoundRobin, numProcesses, outputFile, Buffer\_Burst, SnapshotStep, &Sum\_Timings); // RR Algorithm

runPriority(processPriority, SnapshotStep, numProcesses, outputFile, stats);

// -------------------------------------

Performace\_report(&Sum\_Timings, outputFile, stats);

return 0;

}

Sched\_sim.cpp

/\*

Varun Subramanya

\*/

#include "sched\_sim.hpp"

#define DELTA\_Q\_SIZE 140

using namespace std;

// push to the tail in the queue

void push( Q\_struct \*queue, int data) {

queue->Q\_space[queue->tail++] = data;

}

// pop from the head of the queue

int pop(Q\_struct \*queue) {

int x = queue->Q\_space[queue->head++];

return x;

}

// find the process in the queue which has the minimum burst time

int MinBurst(Q\_struct \*Q, vector<Process> &Processes){

int minB = 1000; // random maximum burst time number to get started

int minB\_Position = 0;

for(int Q\_location = Q->head; Q\_location <= Q->tail; Q\_location++) {

if (Processes[Q->Q\_space[Q\_location]].CPUburst < minB ) {

minB = Processes[Q->Q\_space[Q\_location]].CPUburst;

minB\_Position = Q\_location;

}

}

return Q->Q\_space[minB\_Position]; // return PID with lowest Burst time

}

// check if a new process arrived based on the arrival time

int Arrived\_Flag(int time, int Total\_Processes\_Num, vector<Process> &Processes){

int arrived\_PID = 0;

for(int t = 0; t < Total\_Processes\_Num; t++){

if (time == Processes[t].arrivalTime ){

arrived\_PID=t;

break;

}

}

return arrived\_PID;

}

// allocating memory for the queue and initializing the queue

void Queue\_Init(Q\_struct \*Q, int Total\_Processes\_Num, vector<Process> &Processes){

Q->head = 0;

Q->tail = 0;

Q->Q\_space = (int \*)malloc(sizeof(int) \* (Total\_Processes\_Num + DELTA\_Q\_SIZE));

for(int t = Q->head; t < Total\_Processes\_Num; t++){

if(Processes[t].arrivalTime == 0) {

push(Q, Processes[t].pid);

}}

}

// sorting the process IDs in the queue with the corresponding bursts

void SortQ(Q\_struct \*queue, vector<Process> &Processes){

int swap = 0;

int b\_swap = 0;

int size = queue->tail-queue->head;

int Burst[size+10];

if (size > 1) { // do sort if you have more than one element in the queue

for(int k = queue->head; k<queue->tail; k++ ){

Burst[k] = Processes[queue->Q\_space[k]].CPUburst;

}

// sorting operation

for (int r = queue->head; r<queue->tail-1; r++){

for (int j = queue->head; j<queue->tail-1; j++){

if( Burst[j] > Burst[j+1] ){

// sort Burst time

b\_swap = Burst[j];

Burst[j] = Burst[j+1];

Burst[j+1] = b\_swap;

// sort PID

swap = queue->Q\_space[j];

queue->Q\_space[j] = queue->Q\_space[j+1];

queue->Q\_space[j+1] =swap;

}}}

} // end of size check

}

// print the content of the current queue

void Q\_print(Q\_struct \*queue, vector<Process> &Processes, char \*outputFile){

ofstream output;

output.open(outputFile, fstream::app);

output << "Ready queue: ";

for (int a = queue->head; a<queue->tail; a++){

if (a == queue->head || a == queue->tail) {

output << queue->Q\_space[a];

} else {

output << "-" << queue->Q\_space[a];

}

}

output << endl;

}

// get the sum of all the processes burst time

int sum\_processes\_burst(int numLine, vector<Process> &Processes){

int sum\_line\_countr = numLine;

int sum\_countr = 0;

int sum\_cpu\_burst = 0;

while(sum\_line\_countr > 0) {

sum\_cpu\_burst = sum\_cpu\_burst + Processes[sum\_countr].CPUburst;

sum\_countr++;

sum\_line\_countr--;

}

return sum\_cpu\_burst;

}

// get the number of the remaining processes

int number\_process\_with\_bursts(int number\_lines, vector<Process> &Processes){

int process\_counter = 0;

for(int i = 0; i < number\_lines ; i++){

if(Processes[i].CPUburst > 0) process\_counter++;

}

return process\_counter;

}

// count the waiting cycle for each process

void TimeLogger(int total\_Processes, int curr\_PID, vector<Process> &Processes){

int loopCount;

for(loopCount = 0; loopCount < total\_Processes; loopCount++ ){

if(( Processes[curr\_PID].pid != loopCount )&&(Processes[loopCount].CPUburst>0)){

Processes[loopCount].waitingTime++;

}}

}

bool sortByArrivalTime(const Process &lhs, const Process &rhs) {

return lhs.arrivalTime < rhs.arrivalTime;

}

bool sortByCPUburst(const Process &lhs, const Process &rhs) {

return lhs.CPUburst < rhs.CPUburst;

}

bool sortByPriority(const Process &lhs, const Process &rhs) {

return lhs.priority < rhs.priority;

}

bool sortByValue(const Comparison &lhs, const Comparison &rhs) {

return lhs.value < rhs.value;

}

void runFCFS(vector<Process> processes, int timeStep, int numProc, char \*outputFile, float stats[]) {

ofstream output;

output.open(outputFile, fstream::out);

// variable to hold the end time of the previous process

int lastEnd = 0;

// variable to keep track of the process number

int procNum = 0;

// simulation time

int time = 0;

// variables for algorithm summary

float AvgWaitTime = 0;

float AvgTurnArndTime = 0;

// vector to hold pids for ready queue

vector<int> readyQueue;

// calculate start and end times for FCFS

for (int i = 0; i < numProc; i++) {

if (i == 0) {

processes[i].startTime = processes[i].arrivalTime;

processes[i].endTime = processes[i].CPUburst;

lastEnd = processes[i].endTime;

} else {

processes[i].startTime = lastEnd + 1;

processes[i].endTime = processes[i].startTime + processes[i].CPUburst - 1;

lastEnd = processes[i].endTime;

}

// calculate turn around time

processes[i].turnAroundtime = processes[i].endTime - processes[i].arrivalTime;

}

// print the header for the FCFS algorithm

output << "\*\*\*\*\* FCFS Scheduling \*\*\*\*\*" << endl;

// run the simulation

while (1) {

// if all processes have been evaluated, break out of the loop

if (procNum == numProc) {

break;

}

// loop through the remaining processes to see what needs to be put on the ready queue

for (int i = 0; i < numProc; i++) {

// if they have arrived place them in the ready queue for sorting

if (time == processes[i].arrivalTime) {

readyQueue.push\_back(i);

}

// if the process hasn't arrived yet and the end time has not elapsed, add it to ready queue

else if ((procNum != i) && (time < processes[i].endTime) && (time >= processes[i].arrivalTime)) {

// add it to the ready queue

readyQueue.push\_back(i);

}

}

if (!(time % timeStep)) {

// print the time

output << "t = " << time << endl;

// print out start of a new process

if (time == processes[procNum].startTime - 1) {

output << "CPU: Loading process " << procNum << " (CPU burst = " << processes[procNum].CPUburst << ")" << endl;

}

// otherwise print the status of the current process or next process to be started

if (time == processes[procNum].endTime && procNum != numProc - 1) {

output << "CPU: Finishing process " << procNum << "; loading process " << procNum + 1 << " (CPU burst = " << processes[procNum + 1].CPUburst << ")" << endl;

}

else if (time >= processes[procNum].startTime) {

if (processes[procNum].endTime - time == 0) {

output << "CPU: Finishing process " << procNum << endl;

} else {

output << "CPU: Running process " << procNum << " (CPU burst = remaining " << processes[procNum].endTime - time << ")" << endl;

}

}

}

if (!(time % timeStep)) {

// print ready queue contents

if (!readyQueue.empty()) {

output << "Ready queue: ";

} else {

output << "Ready queue: empty";

output << endl;

}

// loop through the ready queue

for (int j = 0; j < int(readyQueue.size()); j++) {

if (j + 1 != int(readyQueue.size())) {

output << readyQueue[j] << '-';

}

else {

output << readyQueue[j] << endl;

}

}

output << endl;

}

// reset the ready queue for next iteration

readyQueue.clear();

// if the process is finished evaluate the next process

if (time == processes[procNum].endTime) {

procNum++;

}

// increment the time

time++;

}

// print out algorithm summary

output << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

output << "FCFS Summary (WT = wait time, TT = turnaround time):" << endl;

output << "\nPID\t\tWT\t\tTT" << endl;

for (int i = 0; i < numProc; i++) {

AvgWaitTime = (processes[i].startTime - processes[i].arrivalTime - 1) + AvgWaitTime;

AvgTurnArndTime = processes[i].turnAroundtime + AvgTurnArndTime;

output << processes[i].pid << "\t\t" << processes[i].startTime - processes[i].arrivalTime << "\t\t" << processes[i].turnAroundtime << endl;

}

// calculate the averages

AvgTurnArndTime = AvgTurnArndTime / numProc;

stats[0] = AvgTurnArndTime;

AvgWaitTime = AvgWaitTime / numProc;

stats[1] = AvgWaitTime;

// print out the averages

output << "AVG\t\t" << AvgWaitTime << "\t" << AvgTurnArndTime << "\n" << endl;

output << "Process sequence: ";

// loop through the process vector and print out id numbers, since its FCFS arrival order is the only thing that matters

stats[2] = numProc;

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

if (j + 1 != numProc) {

output << processes[j].pid << '-';

} else {

output << processes[j].pid << endl;

}

}

output << "Context switches: " << numProc << endl;

output.close();

}

void SJF (Q\_struct \* queue, vector<Process> &Processes, int total\_Processes, char \*outputFile, int int\_bursts[], int snapStep, Summary \*Sum\_Timings){

ofstream output;

output.open(outputFile, fstream::app);

output << "\n\n\*\*\*\*\* SJF Scheduling \*\*\*\*\*" << endl;

vector<int> context;

int curr\_process = 0, Arrived\_PID = 0,shortest\_PID, X\_time = 0;

int Total\_Burst = sum\_processes\_burst(total\_Processes, Processes);

Queue\_Init(queue, total\_Processes, Processes);

for(X\_time = 0; X\_time <= Total\_Burst; X\_time++ ){ // check for the last process burst time ot be 0

SortQ(queue, Processes); // sort the queue for any execution cycle

if (Processes[curr\_process].CPUburst == 0 ){ // check if the execution of the current process ended

context.push\_back(curr\_process);

SortQ(queue, Processes);

if(X\_time%snapStep == 0){

output << "t = " << X\_time << endl;

output << "CPU: Finishing process " << curr\_process;

}

if(X\_time%snapStep == 0){

if (X\_time ==Total\_Burst) output << " "<< endl;

else output << "; loading process " << Processes[queue->Q\_space[queue->head]].pid << " (CPU Burst = " << Processes[queue->Q\_space[queue->head]].CPUburst << " )" <<endl;

if (queue->head == queue->tail) output << "Ready queue: empty" << endl;

else Q\_print(queue, Processes, outputFile);

output << endl;

}

if (queue->head != queue->tail) shortest\_PID = pop(queue); // POP the shortest Process

curr\_process = Processes[shortest\_PID].pid;

//}

} else {

// condition to manage the queue print outs regarding the snapshot interval value given by the user

if(X\_time%snapStep == 0){

output << "t = " << X\_time << endl;

}

if (X\_time > 0){ // if we running after the first cycle

if(X\_time%snapStep == 0) output << "CPU: Running process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

} else {

if(X\_time%snapStep == 0) {

output << "CPU: Loading process " << curr\_process << " (CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

Q\_print(queue, Processes, outputFile); // print the content of the queue

}

SortQ(queue, Processes); // sort the queue initially

curr\_process = Processes[pop(queue)].pid; // pop a process from the queue and do context switch with the current running process

}

// if new processs arrives we push it in the queue

if( (Arrived\_PID = Arrived\_Flag(X\_time, total\_Processes, Processes)) > 0 ) push(queue, Arrived\_PID);

SortQ(queue, Processes);

if(X\_time%snapStep == 0) { // sort the queue after pushing a new one in the qu

if ((queue->head == queue->tail)&&(X\_time>0)) output << "Ready queue: empty" << endl;

else { if(X\_time !=0)Q\_print(queue, Processes, outputFile); }

output << endl;

}

}

Processes[curr\_process].CPUburst--; // process the burst time

TimeLogger(total\_Processes, curr\_process, Processes); // calculating the waiting cycles for later use to calculate the waiting and turn around time

}

output << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

output << "SJF Summary (WT = wait time, TT = turnaround time):" << endl;

Statistic\_Logs(Processes, context, total\_Processes, outputFile, int\_bursts ,context.size(), &Sum\_Timings->SJF\_alg); // calculating the waiting and turn around time for the currnent algorithm

Processes.clear();

free(queue->Q\_space); // freeing the memory

}

void STCF (Q\_struct \* queue, vector<Process> &Processes, int total\_Processes, char \*outputFile, int int\_bursts[], int snapStep, Summary \*Sum\_Timings) {

ofstream output;

output.open(outputFile, fstream::app);

output << "\n\*\*\*\*\* STCF Scheduling \*\*\*\*\*" << endl;

vector<int> context;

int curr\_process = 0, Arrived\_PID = 0, popedID = 0, X\_time = 0;

int Total\_Burst = sum\_processes\_burst(total\_Processes, Processes);

Queue\_Init(queue, total\_Processes, Processes); // Store the first data in the Queue

for(X\_time = 0; X\_time <= Total\_Burst; X\_time++){ // check for the last process burst time ot be 0

SortQ(queue, Processes); // Sort the Queue on every cycle

if (X\_time == 0) curr\_process = Processes[pop(queue)].pid; // sort the queue initially

if( (Arrived\_PID = Arrived\_Flag(X\_time, total\_Processes, Processes)) > 0 ){ // something arrived

push(queue, Arrived\_PID); // Push in the queue the incomming process

SortQ(queue, Processes); // sort the queue

if (Processes[curr\_process].CPUburst > Processes[Arrived\_PID].CPUburst){

if(X\_time%snapStep == 0){

output << "t = " << X\_time << endl;

output << "CPU: Preempting process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ");";

}

// condition to manage the queue print outs regarding the snapshot interval value given by the user

if(X\_time%snapStep == 0)output << "loading process " << Processes[queue->Q\_space[queue->head]].pid <<"; (CPU burst = "<< Processes[queue->Q\_space[queue->head]].CPUburst << ")" <<endl;

if(X\_time%snapStep == 0){

if (queue->head == queue->tail)output << "Ready queue: empty" << endl;

else Q\_print(queue, Processes, outputFile);

output << endl;

}

// check remaining burst against to arrived burst

push(queue, curr\_process); // if no , keep the current Process for the next X-cycle

context.push\_back(curr\_process); // pushing process ID in the vector when the process finishes so we can calculate the context switch

int shortest\_PID = pop(queue); // if yes ,POP Process with the Shortest Burst time

curr\_process = shortest\_PID; // contect switch happens here, replacing the current process ID with the shortest PID burst process form the queue

} else {

// condition to manage the queue print outs regarding the snapshot interval value given by the user

if((X\_time%snapStep == 0)&&(Processes[curr\_process].CPUburst >0)){

output << "t = " << X\_time << endl;

if(X\_time >0)output << "CPU: Running process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

if (queue->head == queue->tail)output << "Ready queue: empty" << endl;

else Q\_print(queue, Processes, outputFile);

output << endl;

}

}

} else {

if((X\_time%snapStep == 0)&&(Processes[curr\_process].CPUburst > 0)){

output << "t = " << X\_time << endl;

if(X\_time >0) { // if this is not the first execution cycle

output << "CPU: Running process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

if (queue->head == queue->tail)output << "Ready queue: empty" << endl;

else Q\_print(queue, Processes, outputFile);

}else {

output << "CPU: Loading process " << curr\_process << " (CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

output << "Ready queue: "<< curr\_process <<endl;

}

output << endl;

}

}

if (Processes[curr\_process].CPUburst == 0){ // check if the execution of the current process ended

context.push\_back(curr\_process); // pushing process ID in the vector when the process finishes so we can calculate the context switch

SortQ(queue, Processes); // if the current process ended sort again the queue

if(X\_time % snapStep == 0){

output << "t = " << X\_time << endl; // print the currnet execution cycle

output << "CPU: Finishing process " << curr\_process;

if (X\_time ==Total\_Burst) output << " "<< endl;

else output << "; loading process " << Processes[queue->Q\_space[queue->head]].pid << "; (CPU burst = " << Processes[queue->Q\_space[queue->head]].CPUburst << ");" << endl;

}

if(X\_time%snapStep == 0){

if (queue->head == queue->tail)output << "Ready queue: empty" << endl; // if tail = head means empty queue

else Q\_print(queue, Processes, outputFile); // print the current queue

output << endl;

}

popedID = pop(queue); // POP the shortest Process

curr\_process = Processes[popedID].pid; // loading new process

}

Processes[curr\_process].CPUburst--;

TimeLogger(total\_Processes, curr\_process, Processes); // calculating the waiting cycles of each process

}

output << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

output << "STCF Summary (WT = wait time, TT = turnaround time):" << endl;

Statistic\_Logs(Processes, context, total\_Processes, outputFile, int\_bursts ,context.size(), &Sum\_Timings->STCF\_alg); // calculating the waiting and turn around time for the currnent algorithm

Processes.clear(); // clear the vecotor processes

free(queue->Q\_space);

}

void RoundRobin(Q\_struct \* queue , vector<Process> &Processes, int total\_processes ,char \*outputFile, int int\_bursts[],int snapStep, Summary \*Sum\_Timings ){

ofstream output;

output.open(outputFile, fstream::app);

output << "\n\n\*\*\*\*\* Round Robin Scheduling \*\*\*\*\*";

vector<int> context;

int time\_quanta = 2;

int flag = 0; /////variable that shows if the process is executing or not

int time\_quanta\_countr = 0;

int sum\_cpu\_burst = sum\_processes\_burst(total\_processes, Processes);

int curr\_process = 0;

int Arrived\_PID = 0;

// Store the first data in the Queue

Queue\_Init(queue, total\_processes, Processes); // Store the first data in the Queue

int processes\_in\_queue = 0; ///need to add

for (int i = 0; i <= sum\_cpu\_burst; i++) {

if(i % snapStep == 0){

output<<" "<< endl;

output << "t = " << i << endl;

}

Arrived\_PID = Arrived\_Flag(i, total\_processes, Processes);

if (Arrived\_PID > 0) {

push(queue, Arrived\_PID);

}

//// change this location to the place of decreementing the counter

if ( i == 0){

if(i%snapStep == 0){

output << "CPU: Loading process " << queue->head << " (CPU burst = " << Processes[queue->Q\_space[queue->head]].CPUburst << ")" << endl;

Q\_print(queue, Processes, outputFile);

}

}

processes\_in\_queue = number\_process\_with\_bursts(total\_processes, Processes);

if (flag == 1 || processes\_in\_queue > 0 ) {

if (flag == 0 && processes\_in\_queue > 0) {

curr\_process = Processes[pop(queue)].pid;

flag = 1;

time\_quanta\_countr = time\_quanta; //give time quanta to process

}

if (i != 0) {

Processes[curr\_process].CPUburst--;

time\_quanta\_countr = time\_quanta\_countr - 1;

TimeLogger(total\_processes, curr\_process, Processes);

}

if ((i != 0) && (Processes[curr\_process].CPUburst >= 1) && time\_quanta\_countr ==1){

if(i%snapStep == 0){

output << "CPU: Running process " << curr\_process << " (CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

if(queue->head == queue->tail) output << "Ready queue: empty" << endl;

else { Q\_print(queue, Processes, outputFile); }

output << endl;

//Q\_print(queue, Processes, outputFile);

}

}

if (Processes[curr\_process].CPUburst == 0) {

flag = 0;

context.push\_back(curr\_process);

if(i%snapStep == 0){

output << "CPU: Finishing process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ")";

if(queue->head != queue->tail)

output << "CPU: Loading process " << queue->Q\_space[queue->head] << "(CPU burst = " << Processes[queue->Q\_space[queue->head]].CPUburst << ")" << endl;

else output<<" "<< endl;

if(queue->head == queue->tail) output << "Ready queue: empty" << endl;

else { Q\_print(queue, Processes, outputFile); }

output << endl;

//Q\_print(queue, Processes, outputFile);

}

}

if (time\_quanta\_countr == 0 && Processes[curr\_process].CPUburst != 0) {

/// put a if condition if the head and tail of the process is same

if(queue->head != queue->tail){

context.push\_back(curr\_process);

if(i%snapStep == 0){

output << "CPU: Preempting process " << curr\_process << " (remaining CPU burst = " << Processes[curr\_process].CPUburst << ")" ;

output << "CPU: Loading process " << queue->Q\_space[queue->head] << " (CPU burst = " << Processes[queue->Q\_space[queue->head]].CPUburst << ")" << endl;

Q\_print(queue, Processes, outputFile);

}

}

/// changed from else to if with condition other that before condition = queue->head == queue->tail

if(queue->head == queue->tail){

if(i % snapStep == 0){

output << "CPU: Running process " << curr\_process << " (CPU burst = " << Processes[curr\_process].CPUburst << ")" << endl;

//Q\_print(queue, Processes, outputFile);

output << "Ready Queue: ";

output << "empty\n";

output << endl;

}

}

flag = 0;

push(queue, curr\_process);

}

}

}

output << "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

output << "Round robin Summary (WT = wait time, TT = turnaround time):" << endl;

Statistic\_Logs(Processes, context, total\_processes, outputFile, int\_bursts ,context.size(), &Sum\_Timings->RR\_alg);

Processes.clear();

free(queue->Q\_space);

}

void runPriority(vector<Process> processes, int timeStep, int numProc, char \*outputFile, float stats[]) {

ofstream output;

output.open(outputFile, fstream::app);

// print the header for the FCFS algorithm

output << "\n\*\*\*\*\* Priority Scheduling \*\*\*\*\*" << endl;

// variable to keep track of the process number

int procNum = 0;

// simulation time

int time = 0;

// current running process

int CPUProc = -1;

// variable to hold current processes remaining time

int timeRemain = 0;

// vector for context switches

vector<int> context;

// vector to hold contents of ready queue

vector<Process> readyQueue;

// variables for algorithm summary

float AvgWaitTime = 0;

float AvgTurnArndTime = 0;

while (1) {

// check the time step for output

if (!(time % timeStep)) {

output << "t = " << time << endl;

}

// check to see which processes have arrived

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

// if they have arrived place them in the ready queue for sorting

if (time == processes[j].arrivalTime) {

readyQueue.push\_back(processes[j]);

}

}

// sort the list by priority

sort(readyQueue.begin(), readyQueue.end(), sortByPriority);

if (CPUProc != -1 && timeRemain == 0) {

// note the context switch

context.push\_back(CPUProc);

// note the end time

processes[CPUProc].endTime = time;

// calculate turn around time

processes[CPUProc].turnAroundtime = processes[CPUProc].endTime - processes[CPUProc].arrivalTime;

if (numProc == int(context.size())) {

// print out last

if (!(time % timeStep)) {

output << "CPU: Finishing process " << CPUProc << endl;

output << "Ready queue: empty" << endl;

}

break;

}

if (!(time % timeStep)) {

output << "CPU: Finishing process " << CPUProc << "; loading process " << readyQueue[0].pid << " (CPU burst = " << readyQueue[0].CPUburst << ")" << endl;

}

// reset flags and counter

CPUProc = readyQueue[0].pid;

timeRemain = readyQueue[0].CPUburst;

// note the start time

processes[CPUProc].startTime = time;

// take it off the ready queue

readyQueue.erase(readyQueue.begin());

// bring in the next process

++procNum;

} else if (CPUProc != -1 && timeRemain > 0) {

if (!(time % timeStep)) {

output << "CPU: Running process " << CPUProc << " (CPU burst = remaining " << timeRemain << ")" << endl;

}

}

// first iteration

if (CPUProc == -1){

CPUProc = readyQueue[0].pid;

timeRemain = readyQueue[0].CPUburst;

if (!(time % timeStep)) {

if (time == 0) {

output << "CPU: Loading process " << CPUProc << " (CPU burst = " << timeRemain << ")" << endl;

} else {

output << "CPU: Running process " << CPUProc << " (CPU burst = remaining " << timeRemain << ")" << endl;

}

}

// remove process from ready queue

readyQueue.erase(readyQueue.begin());

}

if (!(time % timeStep)) {

// print out ready queue

if (!readyQueue.empty()) {

output << "Ready queue: ";

} else {

output << "Ready queue: empty";

output << endl;

}

// loop through the ready queue

for (int j = 0; j < int(readyQueue.size()); j++) {

if (j + 1 != int(readyQueue.size())) {

output << readyQueue[j].pid << '-';

}

else {

output << readyQueue[j].pid << endl;

}

}

output << endl;

}

// change the counters

timeRemain--;

time++;

}

// print out algorithm summary

output << "\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*" << endl;

output << "Priority Summary (WT = wait time, TT = turnaround time):" << endl;

output << "\nPID\t\tWT\t\tTT" << endl;

for (int i = 0; i < numProc; i++) {

AvgWaitTime = (processes[i].startTime - processes[i].arrivalTime) + AvgWaitTime;

AvgTurnArndTime = processes[i].turnAroundtime + AvgTurnArndTime;

output << processes[i].pid << "\t\t" << processes[i].startTime - processes[i].arrivalTime << "\t\t" << processes[i].turnAroundtime << endl;

}

// calculate the averages

AvgTurnArndTime = AvgTurnArndTime / numProc;

stats[3] = AvgTurnArndTime;

AvgWaitTime = AvgWaitTime / numProc;

stats[4] = AvgWaitTime;

// print out the averages

output << "AVG\t\t" << AvgWaitTime << "\t" << AvgTurnArndTime << "\n" << endl;

output << "Process sequence: ";

// loop through the process vector and print out id numbers, since its FCFS arrival order is the only thing that matters

for (int j = 0; j < int(context.size()); j++) {

if (j + 1 != numProc) {

output << context[j] << '-';

} else {

output << context[j] << endl;

}

}

stats[5] = int(context.size());

output << "Context switches: " << context.size() << endl << endl;

}

// acquiring timmings for any algorithm calls this function and store it in the performance metrics structure

void Statistic\_Logs(vector<Process> &Processes, vector<int> &Context, int total\_Processes, char \*outputFile, int Burst\_Array[], int context\_Count, P\_metrics \*timmings){

ofstream output;

output.open(outputFile, fstream::app);

int waitc = 0;

float avgWT = 0, avgTT = 0;

output << "\nPID\t\tWT\t\tTT" << endl;

for(int c\_count = 0; c\_count < total\_Processes; c\_count++){

waitc = Processes[c\_count].waitingTime - Processes[c\_count].arrivalTime; // calculating current process waiting time

avgWT = avgWT + waitc; // Average Waiting timming

avgTT = avgTT + waitc + Burst\_Array[c\_count]; // Turn Around timming

output << Processes[c\_count].pid << "\t\t" << waitc << "\t\t" << waitc + Burst\_Array[c\_count] << endl;

}

timmings->WT\_time = avgWT/total\_Processes;

timmings->TT\_time = avgTT/total\_Processes; // store timmings in the sctrucutre

output << "AVG" << "\t\t" << timmings->WT\_time << "\t" << timmings->TT\_time << endl;

output << "\nProcess sequence: ";

// loop through the process vector and print out id numbers, since its FCFS arrival order is the only thing that matters

for (int j = 0; j < int(Context.size()); j++) {

output<< Context[j] << '-';

}

output << endl;

timmings->Context\_c = context\_Count; // contex switch counter stored in the metrics structure

output << "Context switches: " << timmings->Context\_c << endl;

}

void Performace\_report(Summary \* sum\_time, char \*outputFile, float stats[]){

ofstream output;

output.open(outputFile, fstream::app);

// create the vectors to hold the data

vector <Comparison> waitTimeComparison;

vector <Comparison> turnaroundTimeComparison;

vector <Comparison> contextComparison;

Comparison WT;

Comparison TT;

Comparison CS;

// SJF

WT.value = sum\_time->SJF\_alg.WT\_time;

WT.type = "SJF";

CS.value = sum\_time->SJF\_alg.Context\_c;

CS.type = "SJF";

TT.value = sum\_time->SJF\_alg.TT\_time;

TT.type = "SJF";

waitTimeComparison.push\_back(WT);

contextComparison.push\_back(CS);

turnaroundTimeComparison.push\_back(TT);

// STCF

WT.value = sum\_time->STCF\_alg.WT\_time;

WT.type = "STCF";

CS.value = sum\_time->STCF\_alg.Context\_c;

CS.type = "STCF";

TT.value = sum\_time->STCF\_alg.TT\_time;

TT.type = "STCF";

waitTimeComparison.push\_back(WT);

contextComparison.push\_back(CS);

turnaroundTimeComparison.push\_back(TT);

// round robin

WT.value = sum\_time->RR\_alg.WT\_time;

WT.type = "Round robin";

CS.value = sum\_time->RR\_alg.Context\_c;

CS.type = "Round robin";

TT.value = sum\_time->RR\_alg.TT\_time;

TT.type = "Round robin";

waitTimeComparison.push\_back(WT);

contextComparison.push\_back(CS);

turnaroundTimeComparison.push\_back(TT);

// priority

WT.value = stats[4];

WT.type = "Priority";

CS.value = stats[5];

CS.type = "Priority";

TT.value = stats[3];

TT.type = "Priority";

waitTimeComparison.push\_back(WT);

contextComparison.push\_back(CS);

turnaroundTimeComparison.push\_back(TT);

// FCFS

WT.value = stats[1];

WT.type = "FCFS";

CS.value = stats[2];

CS.type = "FCFS";

TT.value = stats[0];

TT.type = "FCFS";

waitTimeComparison.push\_back(WT);

contextComparison.push\_back(CS);

turnaroundTimeComparison.push\_back(TT);

sort(waitTimeComparison.begin(),waitTimeComparison.end(), sortByValue);

sort(contextComparison.begin(),contextComparison.end(), sortByValue);

sort(turnaroundTimeComparison.begin(),turnaroundTimeComparison.end(), sortByValue);

output<< "\n\*\*\*\*\* OVERALL SUMMARY \*\*\*\*\*\n" << endl;

output<< "Wait Time Comparison" << endl;

int setSpace = 0;

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

if (waitTimeComparison[i].type.length() == 11) {

setSpace = 8;

} else if(waitTimeComparison[i].type.length() == 8) {

setSpace = 11;

} else if(waitTimeComparison[i].type.length() == 4) {

setSpace = 15;

} else {

setSpace = 16;

}

output << i + 1 << " " << waitTimeComparison[i].type << setw(setSpace) << waitTimeComparison[i].value << endl;

}

output<<"Turnaround Time Comparison" << endl;

for (int j = 0; j < int(turnaroundTimeComparison.size()); j++) {

if (turnaroundTimeComparison[j].type.length() == 11) {

setSpace = 8;

} else if(turnaroundTimeComparison[j].type.length() == 8) {

setSpace = 11;

} else if(turnaroundTimeComparison[j].type.length() == 4) {

setSpace = 15;

} else {

setSpace = 16;

}

output << j + 1 << " " << turnaroundTimeComparison[j].type << setw(setSpace) << turnaroundTimeComparison[j].value << endl;

}

output<<"Context Switch Comparison" << endl;

for (int k = 0; k < int(contextComparison.size()); k++) {

if (contextComparison[k].type.length() == 11) {

setSpace = 8;

} else if(contextComparison[k].type.length() == 8) {

setSpace = 11;

} else if(contextComparison[k].type.length() == 4) {

setSpace = 15;

} else {

setSpace = 16;

}

output << k + 1 << " " << contextComparison[k].type << setw(setSpace) << contextComparison[k].value << endl;

}

}

Sched\_sim.hpp

/\*

Varun Subramanya

\*/

#ifndef SCHED\_SIM\_HPP

#define SCHED\_SIM\_HPP

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <stdbool.h>

#include <time.h>

#include <iostream>

#include <fstream>

#include <sstream>

#include <string>

#include <vector> // to get the vector class definition

#include <algorithm>

#include <iomanip>

//#define QUEUE\_SIZE 100

using namespace std;

typedef struct {

int \*Q\_space;

int head;

int tail;

} Q\_struct;

typedef struct {

float value;

string type;

} Comparison;

class Process {

public:

// class variables

int pid;

int CPUburst;

int priority;

int arrivalTime;

int startTime;

int endTime;

int waitingTime;

int turnAroundtime;

// class constructor

Process(int id, int CPU, int prior, int arrv) {

pid = id;

CPUburst = CPU;

priority = prior;

arrivalTime = arrv;

startTime = 0;

endTime = 0;

waitingTime = 0;

turnAroundtime = 0;

}

};

// structure for counting turn arround and waiting timmings

typedef struct {

float TT\_time;

float WT\_time;

int Context\_c;

} P\_metrics;

// structure for grouping the performance timmings for each algorithm

typedef struct {

P\_metrics SJF\_alg;

P\_metrics STCF\_alg;

P\_metrics RR\_alg;

P\_metrics FCFS\_alg;

P\_metrics Prior\_alg;

}Summary;

//extern "C" void test(void);

extern Q\_struct Queue;

extern P\_metrics timmings;

extern Summary Sum\_Timings;

// ----- Prototypes -----

void push(Q\_struct \*queue, int data);

int pop(Q\_struct \*queue);

int MinBurst(Q\_struct \*Q, vector<Process> &Processes);

int Arrived\_Flag(int time, int Total\_Processes\_Num, vector<Process> &Processes);

void Queue\_Init(Q\_struct \*Q, int Total\_Processes\_Num, vector<Process> &Processes);

void SortQ(Q\_struct \*queue, vector<Process> &Processes);

void Q\_print(Q\_struct \*queue, vector<Process> &Processes, char \*outputFile);

int sum\_processes\_burst( int numLine , vector<Process> &Processes);

int number\_process\_with\_bursts(int number\_lines, vector<Process> &Processes);

void TimeLogger(int total\_Processes, int curr\_PID, vector<Process> &Processes);

bool sortByArrivalTime(const Process &lhs, const Process &rhs);

bool sortByCPUburst(const Process &lhs, const Process &rhs);

bool sortByPriority(const Process &lhs, const Process &rhs);

bool sortByValue(const Comparison &lhs, const Comparison &rhs);

void runFCFS(vector<Process> processes, int timeStep, int numProc, char \*outputFile, float stats[]);

void STCF(Q\_struct \*queue, vector<Process> &Processes, int total\_Processes, char \*outputFile, int int\_bursts[], int snapStep, Summary \*Sum\_Timings);

void SJF(Q\_struct \* queue, vector<Process> &Processes, int total\_Processes, char \*outputFile, int int\_bursts[], int snapStep, Summary \*Sum\_Timings);

void RoundRobin(Q\_struct \* queue, vector<Process> &Processes, int total\_processes, char \*outputFile, int int\_bursts[], int snapStep, Summary \*Sum\_Timings);

void runPriority(vector<Process> processes, int timeStep, int numProc, char \*outputFile, float stats[]);

void Statistic\_Logs(vector<Process> &Processes, vector<int> &Context, int total\_Processes, char \*outputFile, int Burst\_Array[], int context\_Count, P\_metrics \*timmings);

void Performace\_report(Summary \* sum\_time, char \*outputFile, float stats[]);

#endif