/\*----------------------------------------------------------------------------

Meryl Mabin

CSE 410, section 001

proj03 - Scheduling

---------------------------------------------------------------------------\*/

#include <iostream>

#include <fstream>

#include <string.h>

#include <queue>

#include <vector>

#include <algorithm>

using namespace std;

struct process

{

int ID,

arrivalTime,

serviceTime,

remainTime,

alreadyRun;

};

struct responseRatio

{

int Tr,

Ts,

ID;

float R;

};

bool processSort( process left, process right );

bool timeSort( responseRatio left, responseRatio right );

int main( int Argc, char \*Arg[] )

{

int timeSlot,

procNum = 0,

noProc = 0,

index = 0,

quantum = 0,

tempQ;

float floatTr,

floatTs;

ifstream InStream; // Input stream

ofstream OutStream; // Output stream

string InFile, // Name of input file

OutFile, // Name of output file

checker,

strQuantum;

process readProc,

thisProc,

shortestProc;

vector<process> processList, // List of processes read from file

readyProcs;

vector<responseRatio> timesList; // Response Ratio

queue<process> RRqueue; // Queue for Round Robin

// Arg[0] = ScheSim

// Arg[1] = input file

// Arg[2] = output file

// Arg[3] = option

// Arg[4] = argument (time quantum)

// Not enough arguments

if ( Argc < 4 )

{

cout << "Please make sure you've typed in all the arguments." << endl;

}

// Too many arguments

else if ( Argc > 5 )

{

cout << "Too many arguments." << endl;

}

// Valid input

else

{

// Check to make sure the fourth arg is one of the right options

string checker = Arg[3];

if ( checker != "-FCFS" && checker != "-SPN" && checker != "-SRT" &&

checker != "-RR" )

{

cout << checker << endl;

cout << "Invalid option entered." << endl;

}

// Make sure there's a number for Round Robin

else if ( (checker == "-RR") && (Argc != 5) )

{

cout << "Please enter a number for Round Robin.\n";

}

// Arguments all okay

else

{

// Open files

InFile = Arg[1];

InStream.open( InFile.c\_str() );

OutFile = Arg[2];

OutStream.open( OutFile.c\_str() );

// Read the input file into an array of processes

while ( true )

{

procNum++;

readProc.ID = procNum;

readProc.alreadyRun = 0;

InStream >> readProc.arrivalTime >> readProc.serviceTime;

readProc.remainTime = readProc.serviceTime;

if ( InStream.eof() ) break;

processList.push\_back(readProc);

}

/\*

\*\*\*\* Options \*\*\*\*

\*/

// If FCFS, do first-come-first-served

if ( checker == "-FCFS" )

{

timeSlot = 0;

for ( unsigned int i=0; i<processList.size(); i++ )

{

thisProc = processList[i];

// if it's too early for the next process, just increment time

while ( timeSlot < processList[i].arrivalTime )

{

OutStream << timeSlot << endl;

timeSlot++;

}

// run the current process to completion, output into file

while ( processList[i].remainTime > 0 )

{

OutStream << timeSlot << " " << processList[i].ID << endl;

timeSlot++;

processList[i].remainTime -= 1;

}

// Figure out turnaround time, etc.

if (processList[i].remainTime == 0)

{

responseRatio procTime;

procTime.Tr = timeSlot - processList[i].arrivalTime;

floatTr = procTime.Tr;

procTime.ID = processList[i].ID;

procTime.Ts = processList[i].serviceTime;

floatTs = procTime.Ts;

procTime.R = (floatTr / floatTs);

timesList.push\_back(procTime);

}

}

}

// If SPN, do shortest process next

else if ( checker == "-SPN" )

{

timeSlot = 0;

noProc = 1;

while ( !processList.empty() )

{

// Add any processes that are ready at this time to the Ready queue

while ( processList.front().arrivalTime <= timeSlot )

{

noProc = 0;

readyProcs.push\_back( processList.front() );

processList.erase( processList.begin() );

if ( processList.empty() ) break;

}

// Loop through the Ready queue and run processes to completion

while ( !readyProcs.empty() )

{

shortestProc = readyProcs.front();

index = 0;

// Find the shortest process in the Ready queue

for ( unsigned i=0; i<readyProcs.size(); i++ )

{

if ( readyProcs[i].serviceTime < shortestProc.serviceTime )

{

shortestProc = readyProcs[i];

index = i;

}

}

// Run the process to completion

while ( shortestProc.remainTime > 0 )

{

OutStream << timeSlot << " " << shortestProc.ID << endl;

timeSlot++;

shortestProc.remainTime -= 1;

}

// Figure out turnaround time, etc.

if (shortestProc.remainTime == 0)

{

responseRatio procTime;

procTime.Tr = timeSlot - shortestProc.arrivalTime;

floatTr = procTime.Tr;

procTime.ID = shortestProc.ID;

procTime.Ts = shortestProc.serviceTime;

floatTs = procTime.Ts;

procTime.R = (floatTr / floatTs);

timesList.push\_back(procTime);

}

// Process is done running, so pop it

readyProcs.erase( readyProcs.begin() + index );

} // end while readyProcs.size() > 0

// If there were no processes ready, output this into the file

if (noProc)

{

OutStream << timeSlot << "no process\n";

timeSlot++;

}

}

}

// If SRT, do shortest remaining time

else if ( checker == "-SRT" )

{

timeSlot = 0;

noProc = 1;

while ( !processList.empty() )

{

while ( processList.front().arrivalTime <= timeSlot )

{

noProc = 0;

readyProcs.push\_back( processList.front() );

processList.erase( processList.begin() );

if ( processList.empty() ) break;

}

// Loop through the Ready queue and run processes to completion

while ( !readyProcs.empty() )

{

shortestProc = readyProcs.front();

index = 0;

// Find the shortest process in the Ready queue

for ( unsigned i=0; i<readyProcs.size(); i++ )

{

if ( readyProcs[i].remainTime < shortestProc.remainTime )

{

shortestProc = readyProcs[i];

index = i;

}

}

// Run shortest process for one time quantum

OutStream << timeSlot << " " << shortestProc.ID << endl;

timeSlot++;

shortestProc.remainTime -= 1;

// If remaining time is 0, erase it from Ready

if ( shortestProc.remainTime == 0 )

{

readyProcs.erase( readyProcs.begin() + index );

// Figure out turnaround time, etc.

responseRatio procTime;

procTime.Tr = timeSlot - shortestProc.arrivalTime;

floatTr = procTime.Tr;

procTime.ID = shortestProc.ID;

procTime.Ts = shortestProc.serviceTime;

floatTs = procTime.Ts;

procTime.R = (floatTr / floatTs);

timesList.push\_back(procTime);

}

else

{

// Remove it from the queue and then add it back

readyProcs.erase( readyProcs.begin() + index );

readyProcs.push\_back( shortestProc );

}

// At this time add any processes that have become ready

while ( (!processList.empty()) &&

(processList.front().arrivalTime <= timeSlot) )

{

noProc = 0;

readyProcs.push\_back( processList.front() );

processList.erase( processList.begin() );

if ( processList.empty() ) break;

}

} // end while readyProcs.size() > 0

// If there were no processes ready, output this into the file

if (noProc)

{

OutStream << timeSlot << " no process\n";

timeSlot++;

}

}

}

// If RR, do round-robin

else if ( checker == "-RR" )

{

bool flag = 0;

timeSlot = 0;

quantum = atoi( Arg[4] );

tempQ = quantum;

noProc = 1;

// while there still exist processes...

while ( !processList.empty() )

{

while ( processList.front().arrivalTime <= timeSlot )

{

// put processes that are ready into the ready queue

noProc = 0;

RRqueue.push( processList.front() );

processList.erase( processList.begin() );

if ( processList.empty() ) break;

}

while ( !RRqueue.empty() )

{

// Run current process for one time quantum

OutStream << timeSlot << " " << RRqueue.front().ID << endl;

timeSlot++;

tempQ--;

flag = 0;

RRqueue.front().remainTime -= 1;

// if the process is done, pop it from the queue

if ( RRqueue.front().remainTime == 0 )

{

// Figure out turnaround time, etc.

responseRatio procTime;

procTime.Tr = timeSlot - RRqueue.front().arrivalTime;

floatTr = procTime.Tr;

procTime.ID = RRqueue.front().ID;

procTime.Ts = RRqueue.front().serviceTime;

floatTs = procTime.Ts;

procTime.R = (floatTr / floatTs);

timesList.push\_back(procTime);

// pop from the queue, reset variables

RRqueue.pop();

flag = 1;

tempQ = quantum;

}

// check for a new process

while ( (processList.front().arrivalTime <= timeSlot) &&

!processList.empty() )

{

// put processes that are ready into the ready queue

noProc = 0;

RRqueue.push( processList.front() );

processList.erase( processList.begin() );

if ( processList.empty() ) break;

}

// if time quantum is up, switch to next process by

// moving front of the queue to the back

if ( (tempQ == 0) && (RRqueue.front().remainTime != 0) )

{

if ( flag == 0)

{

thisProc = RRqueue.front();

RRqueue.pop();

RRqueue.push( thisProc );

}

// reset tempQ

tempQ = quantum;

}

} // end while !RRqueue.empty()

// if no processes were ready, output this

if ( noProc && RRqueue.empty() )

{

OutStream << timeSlot << " no process\n";

timeSlot++;

}

}

}

InStream.close();

OutStream.close();

// Output turnaround time, service time, response time

sort( timesList.begin(), timesList.end(), timeSort );

for ( unsigned i=0; i<timesList.size(); i++ )

{

cout << "----------------------------------------------" << endl;

cout << "[ID: " << timesList[i].ID << "]\n";

cout << "Turnaround time: " << timesList[i].Tr << endl;

cout << "Service time: " << timesList[i].Ts << endl;

cout << "Response time: " << timesList[i].R << endl;

cout << "----------------------------------------------" << endl;

cout << endl;

}

}

}

}

bool processSort( process left, process right )

{

return (left.remainTime < right.remainTime);

}

bool timeSort( responseRatio left, responseRatio right )

{

return (left.ID < right.ID);

}