```cpp
/*

 Khaled Sharafaddin
 Kristina Reddy
 Qi Zhang
 Program Name: Project OS3

 Purpose: The purpose of this project is to simulate the workflow of the CPU scheduling based on the Round Robin scheduling algorithm.

 Input:
    1) Input from keyboard:
        - Input filename
        - The total simulation time (in integer seconds)
        - The quantum size (in integer milliseconds; usually between 10 and 100)
        - The number of processes allowed in the system (degree of multiprogramming - how many jobs are in the system)
    2) An input text file contains incomingQueue jobs. The first line is an integer that represents the total number of lines (jobs) in the file. Each subsequent line has four integers: start time of the job, PID, the probability of I/O requests, and the job length.

 Output:
    1) Output on console:
        - Prompt to enter input filename, simulation time, quantum size and the degree of multiprogramming
        - Error messages while file not found
        - Throughput (number of jobs completed during the simulation)
        - Number of jobs still in system
        - Number of jobs skipped
        - Average job length excluding I/O time
        - Average turnaround time
        - Average waiting time per process
        - Percentage of time CPU is busy (CPU utilization)

 - We have abided by the Wheaton College honor code in this work.
*/

#include<iostream>
#include<fstream>
#include<stdlib.h>
#include<time.h>
#include<queue>
#include<math.h>
using namespace std;


const int PENALTYOUTCPU = 4;  // Constant value indicating the penalty of incomplete job being swapped out of the CPU

// Define a class to hold all the relevant values for each job
class jobs
{
public:
    int jobStartTime;
    int jobPID;
    int jobProbIORequest;
    int jobLength;
    int jobLengthOriginal;
    bool firstEnter; // Initial false in main()
    int ioLength;  // Initial 0 in main()

};

//Define a class to hold the three queues;
```

```
class queues
{
public:
    queue<jobs> incomingQueue;
    queue<jobs> readyQueue;
    queue<jobs> ioQueue;

};

int randomNumber(int start, int endtime)
{
    /* This function generates the random number within the limits of start and endtime
     - Pre-condition: two integers start and endtime are given
     - Post-condition: random number is generated between the limits
     - Return: random number
     */

    // A variable to hold the random number
    int randomNumber;

    // Generate the random number within the limits of start and endtime
    srand(time(NULL));
    randomNumber = rand() % endtime + start;

    return randomNumber;
}

void io(queues &simulationqueues, time_t &ioTimeStart, int &ioJobLength){
    /* This function simulates the IO
     - Pre-condition: Two queues (readyQueue, ioQueue) of type jobs are defined and stored in simulationqueues
                      ioTimeStart and ioJobLength are declared
     - Post-condition: Two queues, ioTimeStart, and ioJoblength are modified
     - Return: None
     */

    jobs inIO;  // Declare a variable of type jobs to hold the job that is in the IO
    bool enterIO = true;  // A boolean to hold if the first job of the IOqueue is allowed to enter the IO

    // Check if the previous IO job has finished or not
    if(ioTimeStart != 0 ){
        // If the IO is currently busy, then set the boolean enterIO to false to prevent the job entering IO; otherwise, next the first job from the ioQueue can enter the IO
        if ((ioJobLength < (time(NULL) * 1000 - ioTimeStart))) {
            enterIO = false;
        } else {
            enterIO = true;
        }
    }

    // If the ioQueue is not empty, and the previous job has left the IO, then get the job from the front of the ioQueue
    if(!simulationqueues.ioQueue.empty() && enterIO){

        // Get a job from the front of the ioQueue
        inIO = simulationqueues.ioQueue.front();

        // Start the clock for current job in IO
        ioTimeStart = time(NULL) * 1000; // Change seconds to milliseconds

        // Pop the job after finished the IO
        simulationqueues.ioQueue.pop();

        // Check to see if the jobLength is greater than 0, then push the job into the
```

```
readyQueue
        if (inIO.jobLength > 0) {
            // Generate initial random value for new process and put it at the end of the readyQueue queue
            ioJobLength = randomNumber(5,25);
            inIO.ioLength += ioJobLength;
            simulationqueues.readyQueue.push(inIO);
        }
    }
}

int cpu(int quantum, int &throughput, int &jobsInSystem, int systemTimeAt, int simulationTime, queues &simulationqueues, int &totalJobLength, int &totalWaitTime, int &totalTurnaround, int &cpubusy, bool &jobLeftInCPU)
{
    /* This function simulates the CPU
     - Pre-condition: quantum is defined as an integer from user input
                      throughput is an integer indicating the amount of job finished
                      jobsInSystem is an integer indicating how many jobs are in the system
                      systemTimeAt is an integer indicating the current system time
                      simulationTime is an integer indicating the total simulation time
                      Three queues (incomingQueue, readyQueue, ioQueue) of type jobs are defined
                      Total job length holding the total length of jobs finished
                      Total wait time for all the completed jobs
                      Total turnaround time for all the completed jobs
                      CPU busy time indicating the amount of time all jobs spent in the CPU
                      A boolean indicating whether the CPU is currently busy or not
     - Post-condition: throughput and jobsInSystem are passed by reference
                       three queues are modified
                       totalJobLength, totalWaitTime, totalTurnaround, cpubusy, jobLeftInCPU are passed by reference
     - Return: Current system time
     */

    // Check to see if the readyQueue is empty or not. If not empty, then put a job from the readyQueue into the CPU
    if(!simulationqueues.readyQueue.empty()){
        // For Debugging
        // cout << "Sys Time Before: " << systemTimeAt << endl;

        // Declare a variable of type jobs to hold the job that is in the CPU, and get the job from the front of the readyQueue
        jobs inCPU;
        inCPU = simulationqueues.readyQueue.front();
        simulationqueues.readyQueue.pop();

        // Check to see if the job has already been into the CPU or not. If so, set the current system time to the start time of this job and change the boolean value to true
        if (inCPU.firstEnter == false && (inCPU.jobStartTime >= systemTimeAt)){
            systemTimeAt = inCPU.jobStartTime;
            inCPU.firstEnter = true;
        }

        int rrandomNumber = randomNumber(1,100);  //sets a new variable to hold the random number to determine if it goes into IO
        int iorandomNumber = randomNumber(0,quantum);  // Sets a new variable to hold when the job goes into the ioQueue

        // If the current job length would not exceed the simulation time, then
        if((systemTimeAt + quantum) <= simulationTime )
        {
            //if the probability of the joblength is greater than or equal to that of the calculated random number then the job goes into the ioQueue
            if(inCPU.jobProbIORequest >= rrandomNumber)
```

```cpp
            {
                // Push this job into the ioQueue
                simulationqueues.ioQueue.push(inCPU);

                // Decrement the current job length, and increment CPU busy time and current system time
                inCPU.jobLength -= iorandomNumber;
                cpubusy += iorandomNumber;
                systemTimeAt += iorandomNumber;

                // For Debugging
                // cout << "CPU -> IO 1" << endl;
            }
            else{
                if (inCPU.jobLength > quantum){
                    // Decrement the current job length, and increment CPU busy time and current system time
                    inCPU.jobLength = inCPU.jobLength - quantum;
                    cpubusy += quantum;
                    systemTimeAt = systemTimeAt + quantum + PENALTYOUTCPU;

                    // Push this job into the readyQueue
                    simulationqueues.readyQueue.push(inCPU);

                    // For Debugging
                    // cout << "readyQueue -> CPU"  << endl;
                }

                else{
                    // As this job is finished, increment the throughput, CPU busy time, and current system time
                    throughput++;
                    cpubusy += inCPU.jobLength;
                    systemTimeAt = systemTimeAt + inCPU.jobLength + PENALTYOUTCPU;

                    // Increment the totalJobLength, totalWaitTime, and totalTurnaround
                    totalJobLength += inCPU.jobLengthOriginal;
                    totalWaitTime += systemTimeAt - inCPU.jobLengthOriginal;
                    totalTurnaround += systemTimeAt - inCPU.jobStartTime + inCPU.ioLength;

                    // If the incomingQueue is not empty, then push the first job to the readyQueue, and remove it from the incomingQueue
                    if(!simulationqueues.incomingQueue.empty())
                    {
                        simulationqueues.readyQueue.push(simulationqueues.incomingQueue.front());
                        simulationqueues.incomingQueue.pop();

                        // For Debugging
                        // cout << "incomingQueue -> readyQueue"  << endl;
                    }
                }
            }
        }
        // If the current job length would exceed the simulation time, then
        else
        {
            //if the probability of the joblength is greater than or equal to that of the calculated random number, and the time entering the IO would not exceed the total simulation time, then the job goes into the ioQueue
            if(inCPU.jobProbIORequest >= rrandomNumber && iorandomNumber < (simulationTime - systemTimeAt))
            {
                // Push this job into the ioQueue
                simulationqueues.ioQueue.push(inCPU);

                // Decrement the current job length, and increment CPU busy time and current system time
                inCPU.jobLength -= iorandomNumber;
                cpubusy += iorandomNumber;
                systemTimeAt += iorandomNumber;

                // For Debugging
                // cout << "CPU -> IO 2" << endl;
            }
            else{
                // If the job length left is smaller than or equal to the simulation time left, then this job is able to finish within the total simulation time
                if (inCPU.jobLength <= (simulationTime - systemTimeAt)) {
                    // As this job is finished, increment the throughput, CPU busy time, and current system time
                    throughput++;
                    cpubusy += inCPU.jobLength;
                    systemTimeAt = systemTimeAt + inCPU.jobLength + PENALTYOUTCPU;

                    // Increment the totalJobLength, totalWaitTime, and totalTurnaround
                    totalJobLength += inCPU.jobLengthOriginal;
                    totalWaitTime += systemTimeAt - inCPU.jobLengthOriginal;
                    totalTurnaround += systemTimeAt - inCPU.jobStartTime + inCPU.ioLength;
                }
                // The job length left is greater than the simulation time, then this job will not be able to finish within the simulation time, and will remain in the CPU
                else {
                    // Set the systemTimeAt to the simulation time and set the boolean jobLeftInCPU to true to indicate current job has not finished yet
                    systemTimeAt = simulationTime;
                    jobLeftInCPU = true;

                    // Increment CPU busy time
                    cpubusy += (simulationTime - systemTimeAt);

                    // For Debugging
                    // cout << "Simulation done" << endl;
                }
            }
        }

        // For Debugging
        // cout << "-------- cpu -------\n"
        //      << inCPU.jobStartTime << "\t"
        //      << inCPU.jobPID << "\t"
        //      << inCPU.jobProbIORequest << "\t"
        //      << inCPU.jobLength << "\n"
        //      << "-------- cpu -------\n";
        // cout << "Sys Time After: " << systemTimeAt << "\n\n\n\n" << endl;
    }

    return systemTimeAt;
};

int nbjobsstillinsystem(queues simulationqueues, bool &jobLeftInCPU)
{
    /* This function calculates the total number of jobs still in the system
     - Pre-condition: Two queues of type jobs called readyQueue and ioQueue that are stored in simulationqueues
                      A boolean indicating whether the CPU is currently busy or not
     - Post-condition: Both queues will be empty, and jobLeftInCPU is passed by reference
     - Return: total number of jobs still in the system
     */

```

```cpp
    int jobsInSystem = 0;  // the variable to determine the total number of jobs still in the system

    // Loop through the readyQueue to find how many jobs are still in it
    while(!simulationqueues.readyQueue.empty())
    {
        jobsInSystem++;
        simulationqueues.readyQueue.pop();
    }

    // Loop through the ioQueue to find how many jobs are still in it
    while(!simulationqueues.ioQueue.empty())
    {
        jobsInSystem++;
        simulationqueues.ioQueue.pop();
    }

    // Check if there is a job in the CPU, and if so increment the value
    if (jobLeftInCPU){
        jobsInSystem++;
    }

    return jobsInSystem;
}

int totaljobsSkipped(queues simulationqueues)
{
    /* This function calculates the amount of job skipped
     - Pre-condition: a queue of type jobs called incomingQueue that is stored in simulationqueues
     - Post-condition: The incomingQueue will become empty
     - Return: number of job skipped
     */

    int jobsSkipped = 0 ;  // total number of job being skipped

    // Loop through the incomingQueue to see how many jobs are still left in the queue
    while(!simulationqueues.incomingQueue.empty())
    {
        jobsSkipped++;
        simulationqueues.incomingQueue.pop();
    }
    return jobsSkipped;
}

void gousie() {
    /* This function contains an ASCII art of a ghost
     - Pre-condition: None
     - Post-condition: Ghost in cout
     - Return: None
     */
    cout << " .-." << endl
         << "(o o) boo!" << endl
         << "| O \\ " << endl
         << " \\   \\ " << endl
         << "  `~~~' " << endl
         << "HAPPY HALLOWEEN!!!" << endl;
}

int main()
{
    ifstream file;  // variable to hold the input file
    string filename;    // variable to hold the filename
    double simulationTime;   // Variable to hold the simulation time from the user input
    int quantumSize;    // Variable to hold the quantum size from the user input
```

```
    int numProcesses;   // Variable to hold the number of processes (degree of multiprogramming) from the user input
    int lines;  // variable to hold the total number of lines (jobs) from the input file

    // Prompt user for input filename and open file
    cout << "Please enter the file name: ";
    cin >> filename;
    file.open(filename.c_str());

    // If the file is open, then
    if(file.is_open())
    {
        // declare variables and initialize to 0
        int throughput = 0;     // Number of finished jobs
        int jobsInSystem = 0;   // Number of jobs in the system
        int systemTimeAt = 0;   // The current system time
        int jobsSkipped = 0;    // Number of jobs skipped
        queues simulationqueues; //declare a variable of type queues

        // Prompt user for simulationTime, quantumSize, and numProcesses
        cout << "What is the desired simulation time (in seconds)? ";
        cin >> simulationTime;
        simulationTime = simulationTime * 1000;     // Change seconds to milliseconds

        cout << "What is the desired quantum size (in milliseconds)? ";
        cin >> quantumSize;

        cout << "What is the number of processes allowed in the system? ";
        cin >> numProcesses;

        // Declare a variable of type jobs to hold the each job being put into the incomingQueue from the file
        jobs nextjob;
        file >> lines;

        // Put all the values from the input file into the incomingQueue
        for(int i=0; i<lines; i++)
        {
            file >> nextjob.jobStartTime
                 >> nextjob.jobPID
                 >> nextjob.jobProbIORequest
                 >> nextjob.jobLength;

            // Initialize the firstEnter, ioLength and jobLengthOriginal
            nextjob.firstEnter = false;
            nextjob.ioLength = 0;
            nextjob.jobLengthOriginal = nextjob.jobLength;

            // Push the job into the incomingQueue
            simulationqueues.incomingQueue.push(nextjob);
        }

        // Push jobs from the incomingQueue into the readyQueue within the degree of multiprogramming, and pop them from the incomingQueue
        for(int j=0; j<numProcesses; j++)
        {
            simulationqueues.readyQueue.push(simulationqueues.incomingQueue.front());
            simulationqueues.incomingQueue.pop();
        }

        // Declare variables for IO
        time_t ioTimeStart = 0;     // Indicating the starting time of the job in IO
        int ioJobLength = 0;        // A variable indicating the previous IO job length

        // Declare variables for CPU
```

```cpp
        int totalJobLength = 0;     // Total job length (for all the completed jobs)
        int totalWaitTime = 0;      // Total waiting time (for all the completed jobs)
        int totalTurnaround = 0;    // Total turnaround time (for all the completed jobs)
        int cpubusy = 0;            // Total time when the CPU is busy

        bool jobLeftInCPU = false;  // A boolean to hold if there is a job in the CPU

        // While not exceed the simulation time, and either the readyQueue or the ioQueue is not empty, then
        while(systemTimeAt < simulationTime && (!simulationqueues.readyQueue.empty() || !simulationqueues.ioQueue.empty()))
        {
            // call the CPU and IO functions to simulate
            systemTimeAt = cpu(quantumSize, throughput, jobsInSystem, systemTimeAt, simulationTime, simulationqueues, totalJobLength, totalWaitTime, totalTurnaround, cpubusy, jobLeftInCPU);
            io(simulationqueues, ioTimeStart, ioJobLength);
        }

        // call functions to calculate jobsInSystem and jobsSkipped
        jobsInSystem = nbjobsstillinsystem(simulationqueues, jobLeftInCPU);
        jobsSkipped = totaljobsSkipped(simulationqueues);

        // Declare a double to hold the CPU utilization, and calculate the value
        double cpuUtilization = (double) cpubusy / simulationTime;

        // Output all the values
        cout << "\nThroughput (number of jobs completed during the simulation): " << throughput << endl
             << "Number of jobs still in system: " << jobsInSystem << endl
             << "Number of jobs skipped: " << jobsSkipped << endl;


        if(throughput == 0)
            {cout<<"Division by Zero is prohibited"<<endl;}

        else{
            printf("Average job length excluding I/O time: %5.2f%s \n", (double) totalJobLength/throughput, " (ms)");
            printf("Average turnaround time: %5.2f%s \n", (double) totalTurnaround / throughput, " (ms)");
            printf("Average waiting time per process: %5.2f%s \n", (double) totalWaitTime / throughput, " (ms)");
        }

        printf("CPU utilization (percentage of time CPU is busy): %5.2f%c \n", cpuUtilization * 100, '%');

        // call this function for Halloween surprise
        gousie();

    }
    // If file cannot be opened, then show the error message
    else
        cout << "sorry not a valid file" << endl;

    return 0;
}
```