// jobSatck.h

#pragma once

#include<iostream>

struct job { //general structure of a job

char jobType;

int processingTime;

int typeNumber;

int jobNumber;

int arrivalTime;

int waitTime = 0;

};

class jobStack {

private:

job jobSet[5000]; //We can change this number later if we need to

int jobCount = 0;

public:

jobStack();

void push(job inputJob);

job pop();

bool isEmpty();

bool isFull();

void print();

int peekArrivalTime();

void quickSort(int low, int high);

void swap(job\* a, job\* b);

int partition(int low, int high);

int jobCount() {

return jobCount;

}

~jobStack();

};

//MainHeap.h

#pragma once

#include<iostream>

#include "jobStack.h"

class MinHeap {

job\* heapArray;

int capacity;

int heap\_size;

public:

MinHeap(int capacity);

void MinHeapify(int); //to heapify a subtree with root at given index

int parent(int i) { return (i - 1) / 2; } //to get index of parent of node at index i

int left(int i) { return (2 \* i + 1); } //to get index of left child of node at index i

int tight(int i) { return (2 \* i + 2); } //to get index of right child of node at index i

job extractMin(); //to extract the root which is the minium element

job getMin() { return heapArray[0]; } //returns the minimum key from min heap

//inserts a new key 'k'

void insertKey(job k);

bool isEmpty();

void swap(job\* x, job\* y);

void print();

int size() { return heap\_size; }

void addWaitTime();

int getRemainingWaitTime();

};

//processor.h

#pragma once

#include "jobStack.h"

class Processor {

private:

job currentJob;

int jobCount = 0;

public:

Processor();

void push(job inputJobs);

job pop();

bool isEmpty();

bool isFull();

bool isComplete();

void processJobOne();

job peekJob() { return currentJob; }

~Processor();

};

//JobStack.cpp

#include "jobStack.h"

jobStack::jobStack(){}

//This function adds a new job to the top of the stack\*/

void jobStack::push(job inputJob)

{

jobcount++;

jobSet[jobcount - 1] = inputJob;

}

//Removes the object on the top of the stack and returns it\*/

job jobStack::pop()

{

job tempJob;

tempJob = jobSet[jobcount - 1];

jobSet[jobcount - 1] = { ' ',0, 0,0,0 };

jobcount--;

return tempJob;

}

//Returns true if there are no objects in the stack\*/

bool jobStack::isEmpty()

{

if (jobcount == 0) {

return true;

}

else {

return false;

}

}

//Returns true if there are 5000 objects in the stack\*/

bool jobStack::isFull()

{

if (jobcount == 5000) {

return true;

}

else {

return false;

}

}

//Prints out the objects in the stack\*/

void jobStack::print()

{

for (int arr\_i = jobcount - 1; arr\_i >= 0; arr\_i--) {

std::cout << "Type " << jobSet[arr\_i].jobType << " "

<< "Arrival Time " << jobSet[arr\_i].arrivalTime << " "

<< "Job Number " << jobSet[arr\_i].jobNumber << " "

<< "Type Number " << jobSet[arr\_i].typeNumber << " "

<< "Processing Time " << jobSet[arr\_i].processingTime << std::endl;

}

}

//Returns the arrival time of the top job object in the stack\*/

int jobStack::peekArrivalTime()

{

return jobSet[jobcount - 1].arrivalTime;

}

// The main function that implements QuickSort

void jobStack::quickSort(int low, int high)

{

if (low < high)

{

int pi = partition(low, high);

quickSort(low, pi - 1);

quickSort(pi + 1, high);

}

}

// A utility function to swap two elements

void jobStack::swap(job\* a, job\* b)

{

job t = \*a;

\*a = \*b;

\*b = t;

}

int jobStack::partition(int low, int high)

{

job pivot = jobSet[high];

int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (jobSet[j].arrivalTime >= pivot.arrivalTime)

{

i++;

swap(&jobSet[i], &jobSet[j]);

}

}

swap(&jobSet[i + 1], &jobSet[high]);

return (i + 1);

}

jobStack::~jobStack(){}

//minHeap.cpp

#include "minHeap.h"

// Constructor: Builds a heap from a given array a[] of given size

MinHeap::MinHeap(int cap)

{

heap\_size = 0;

capacity = cap;

heapArray = new job[cap];

}

// Inserts a new key 'k'

void MinHeap::insertKey(job k)

{

if (heap\_size == capacity)

{

std::cout << "\nOverflow: Could not insertKey\n";

return;

}

// First insert the new key at the end

heap\_size++;

int i = heap\_size - 1;

heapArray[i] = k;

// Fix the min heap property if it is violated

while (i != 0 && (heapArray[parent(i)].processingTime > heapArray[i].processingTime || heapArray[i].jobType == 'D'))

{

swap(&heapArray[i], &heapArray[parent(i)]);

i = parent(i);

}

//fixes problem where root isn't swapped out

if (i == 0) {

if (heap\_size > 1 && (heapArray[1].processingTime < heapArray[0].processingTime || heapArray[1].jobType == 'D')) {

swap(&heapArray[1], &heapArray[0]);

}

if (heap\_size > 2 && (heapArray[2].processingTime < heapArray[0].processingTime || heapArray[2].jobType == 'D')) {

swap(&heapArray[2], &heapArray[0]);

}

}

}

//Returns true if there are no objects in the heap

bool MinHeap::isEmpty()

{

if (heap\_size == 0) {

return true;

}

else {

return false;

}

}

// A utility function to swap two elements

void MinHeap::swap(job\* x, job\* y)

{

job temp = \*x;

\*x = \*y;

\*y = temp;

}

//Prints out the objects in the Heap, starting at the root

void MinHeap::print()

{

for (int arr\_i = 0; arr\_i < heap\_size; arr\_i++) {

std::cout << "Type " << heapArray[arr\_i].jobType << " "

<< "Arrival Time " << heapArray[arr\_i].arrivalTime << " "

<< "Job Number " << heapArray[arr\_i].jobNumber << " "

<< "Type Number " << heapArray[arr\_i].typeNumber << " "

<< "Processing Time " << heapArray[arr\_i].processingTime << std::endl;

}

}

//Increases the wait time in all job objects in the queue

void MinHeap::addWaitTime()

{

if (heap\_size > 0) {

for (int arr\_i = 0; arr\_i < heap\_size; arr\_i++) {

heapArray[arr\_i].waitTime++;

}

}

}

//Returns the sum of the wait times of all job objects

int MinHeap::getRemainingWaitTime()

{

int totalWait = 0;

if (heap\_size > 0) {

for (int arr\_i = 0; arr\_i < heap\_size; arr\_i++) {

totalWait = totalWait + heapArray[arr\_i].waitTime;

}

}

return totalWait;

}

// Method to remove minimum element (or root) from min heap

job MinHeap::extractMin()

{

if (heap\_size <= 0) {

return { ' ',0,0,0,0 };

}

if (heap\_size == 1)

{

heap\_size--;

return heapArray[0];

}

// Store the minimum value, and remove it from heap

job root = heapArray[0];

heapArray[0] = heapArray[heap\_size - 1];

heap\_size--;

MinHeapify(0);

return root;

}

// A recursive method to heapify a subtree with root at given index

void MinHeap::MinHeapify(int i)

{

int l = left(i);

int r = right(i);

int smallest = i;

if (l < heap\_size && heapArray[l].processingTime < heapArray[i].processingTime)

smallest = l;

if (r < heap\_size && heapArray[r].processingTime < heapArray[smallest].processingTime)

smallest = r;

if (smallest != i)

{

swap(&heapArray[i], &heapArray[smallest]);

MinHeapify(smallest);

}

}

//processor.cpp

#include "Processor.h"

Processor::Processor()

{

}

//Adds an item to the stack

void Processor::push(job inputJob)

{

if (jobCount == 0) {

currentJob = inputJob;

jobCount++;

}

}

//Removes an item from the stack

job Processor::pop()

{

job tempJob = currentJob;

currentJob = {};

jobCount--;

return tempJob;

}

//Returns true when the stack is empty

bool Processor::isEmpty()

{

if (jobCount == 0) {

return true;

}

else {

return false;

}

}

//Returns true when the stack is full(max of 1)

bool Processor::isFull()

{

if (jobCount == 1) {

return true;

}

else {

return false;

}

}

//Returns true when job in the processor has 0 time remaining

bool Processor::isComplete()

{

if (currentJob.processingTime == 0) {

return true;

}

else {

return false;

}

}

//Decreases the processing time in an active job in a processor

void Processor::processJobOne() {

currentJob.processingTime--;

}

Processor::~Processor()

{

}

//main.cpp

#include "jobStack.h"

#include "minHeap.h"

#include <random>

#include "Processor.h"

#include <fstream>

#include <iostream>

using namespace std;

//Description: Creates the job stack

//Precondition: a jobStack object

//Returns the job stack

void createJobStack(jobStack& inputJobs)

{

jobStack tempJobs;

int jobCount = 0;

job tempA = { 'A', 0,0,0,0,0 };

job tempB = { 'B', 0,0,0,0,0 };

job tempC = { 'C', 0,0,0,0,0 };

for (int time = 0; time < 10000; time++) {

//FIFO

//create A Type jobs

if (time % 5 == 0) {

tempA.arrivalTime = time + 4 + rand() % 3;

if (tempA.arrivalTime < 10000) {

jobCount++;

tempA.jobNumber = jobCount;

tempA.processingTime = 1 + rand() % 5;

tempA.typeNumber++;

tempJobs.push(tempA);

}

}

//create type B jobs

if (time % 10 == 0) {

tempB.arrivalTime = time + 9 + rand() % 3;

if (tempB.arrivalTime < 10000) {

jobCount++;

tempB.jobNumber = jobCount;

tempB.processingTime = 6 + rand() % 5;

tempB.typeNumber++;

tempJobs.push(tempB);

}

}

//create type C jobs

if (time % 25 == 0) {

tempC.arrivalTime = time + 24 + rand() % 3;

if (tempC.arrivalTime < 10000) {

jobCount++;

tempC.jobNumber = jobCount;

tempC.processingTime = 11 + rand() % 5;

tempC.typeNumber++;

tempJobs.push(tempC);

}

}

while (tempJobs.isEmpty() != true) {

inputJobs.push(tempJobs.pop());

}

inputJobs.quickSort(0, inputJobs.jobCount() - 1);

}

}

int main(){

jobStack inputJobs;

MinHeap jobHeap(5000);

ofstream logFile;

logFile.open("log.txt");

float heapAvg = 0;

int maxQueue = 0;

long int idleCount = 0;

int jobsCompleted = 0;

int aCount = 0, bCount = 0, cCount = 0;

int activeCycle = 0, totalProcessingTime = 0;

int jobsInterrupted = 0;

int totalWaitTime = 0;

float averageWaitTime = 0;

int userCPU;

createJobStack(inputJobs);

cout << "Welcome to the Test Processor Program!" << endl;

do {

cout << "How many processors would you like to use in this test?" << endl;

cin >> userCPU;

if (userCPU < 0) {

cout << "Invalid Input" << endl;

}

} while (userCPU < 1);

Processor\* CPU = new Processor[userCPU];

//main program loop

for (int time = 0; time < 10000; time++) {

if (time >= 500) {

logFile << time << ") ";

}

//complete job

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

if (CPU[i\_cpu].isComplete() && CPU[i\_cpu].isFull()) {

job tempJob;

tempJob = CPU[i\_cpu].pop();

if (time >= 500) {

logFile << "Job " << tempJob.jobType << " " << tempJob.typeNumber << " Completed; ";

jobsCompleted++;

}

}

}

//add to the heap

while (inputJobs.peekArrivalTime() == time) {

job tempJob = inputJobs.pop();

jobHeap.insertKey(tempJob);

if (time >= 500) {

logFile << "Arrival Job " << tempJob.jobType << ": Overall Job " << tempJob.jobNumber

<< ", Job " << tempJob.jobType << " " << tempJob.typeNumber

<< ", Processing Time " << tempJob.processingTime << "; ";

//count job types

switch (tempJob.jobType)

{

case 'A':

aCount++;

break;

case 'B':

bCount++;

break;

case 'C':

cCount++;

break;

}

}

}

//Add to CPU

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

if (jobHeap.isEmpty()) {

if (time >= 500) {

logFile << "Heap Empty; ";

}

break;

}

else if (CPU[i\_cpu].isEmpty()) {

job tempJob = jobHeap.extractMin();

totalWaitTime = totalWaitTime + tempJob.waitTime;

tempJob.waitTime = 0; //reset in case it is displaced in interrupt

CPU[i\_cpu].push(tempJob);

if (time >= 500) {

logFile << "Begin Processing Job " << tempJob.jobType << " " << tempJob.typeNumber

<< " in CPU " << i\_cpu + 1 << " , end time " << time + tempJob.processingTime << "; ";

}

}

}

//Process

bool activeJob = false;

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

if (CPU[i\_cpu].isEmpty()) {

if (time >= 500) {

logFile << "CPU " << i\_cpu + 1 << ": Idle Time ";

idleCount++;

}

}

else {

job tempJob = CPU[i\_cpu].peekJob();

CPU[i\_cpu].processJobOne();

activeJob = true;

if (time >= 500) {

logFile << "CPU " << i\_cpu + 1 << ": Job " << tempJob.jobType << " " << tempJob.jobNumber << "; ";

totalProcessingTime++;

}

}

}

jobHeap.addWaitTime();

if (activeJob == true && time >= 500)

activeCycle++;

if (time >= 500) {

heapAvg = heapAvg + (jobHeap.size() - heapAvg) / time;

if (jobHeap.size() > maxQueue)

maxQueue = jobHeap.size();

logFile << endl;

}

}

//final report

int totalJobsArrived = aCount + bCount + cCount;

totalWaitTime = totalWaitTime + jobHeap.getRemainingWaitTime();

averageWaitTime = totalWaitTime / totalJobsArrived;

logFile << endl << "Performance Metrics - Calculated from cycle 500 on" << endl;

logFile << "Final Queue Size: " << jobHeap.size() << endl;

logFile << "Average queue size: " << heapAvg << endl;

logFile << "Average time in queue: " << averageWaitTime << " time units" << endl;

logFile << "Idle time: " << idleCount << " time units." << endl;

logFile << "Total Jobs Arrived: " << totalJobsArrived << endl;

logFile << "Total number of jobs A arrived: " << aCount << endl;

logFile << "Total number of jobs B arrived: " << bCount << endl;

logFile << "Total number of jobs C arrived: " << cCount << endl;

logFile << "Total wait time in queue: " << totalWaitTime << " time units." << endl;

logFile << "Maximum jobs in queue: " << maxQueue << endl;

logFile << "Jobs interrupted " << jobsInterrupted << endl;

logFile << "Total jobs completed: " << jobsCompleted << endl;

logFile << "Number of processor(s) used: " << userCPU << endl;

logFile << "Total number of time units the processors(s) run: " << activeCycle << endl;

logFile << "Total time processor(s) spent processing is: " << totalProcessingTime << " time units" << endl;

cout << endl << "Performance Metrics - Calculated from cycle 500 on" << endl;

cout << "Final Queue Size: " << jobHeap.size() << endl;

cout << "Average queue size: " << heapAvg << endl;

cout << "Average time in queue: " << averageWaitTime << " time units" << endl;

cout << "Idle time: " << idleCount << " time units." << endl;

cout << "Total Jobs Arrived: " << totalJobsArrived << endl;

cout << "Total number of jobs A arrived: " << aCount << endl;

cout << "Total number of jobs B arrived: " << bCount << endl;

cout << "Total number of jobs C arrived: " << cCount << endl;

cout << "Total wait time in queue: " << totalWaitTime << " time units." << endl;

cout << "Maximum jobs in queue: " << maxQueue << endl;

cout << "Jobs interrupted " << jobsInterrupted << endl;

cout << "Total jobs completed: " << jobsCompleted << endl;

cout << "Number of processor(s) used: " << userCPU << endl;

cout << "Total number of time units the processors(s) run: " << activeCycle << endl;

cout << "Total time processor(s) spent processing is: " << totalProcessingTime << " time units" << endl;

logFile.close();

cout << "Report information is in the file log.txt" << endl;

cout << "Thank you, have a nice day!" << endl;

system("pause");

return 0;

}