# 1. Problem Statement

Simulate a set of processors that handle one of 4 types of jobs of various lengths and priorities.  Create a log that displays the state of the processor at all times during the simulation, and make a recommendation on the number of processors to use based on the data in the log.

# 2. Requirements

# 2.1 Assumptions

* The user will interact with the program on the command line (the only interaction being an indication of the number of processors)

# 2.2 Specifications

## Job Information

### Job Arrival

Jobs arrive at semi-random times based on the table below.  This "job stream" should be set up prior to the processor starting.

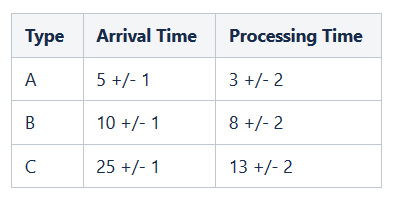
### Job Processing

Jobs have a semi-random processing time specific to the job type.  These processing times are specified in the table below.  The job must process for that amount of time for it to be considered complete.

### Job Types

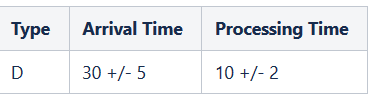
#### Regular Jobs

Regular jobs are all of equal priority, sorted based on processing time.  They should be handled in the order of shortest processing time first.  ***Note:*** these items do not displace any item currently being processed.



#### Highest Priority Jobs

High priority jobs should displace any item that is currently "active" in an open processor.  The item that has been displaced will be added to the heap as if it was a new item.  The remaining time to complete the item will be tracked (it doesn't restart).  If the item that is currently in the processor is a high priority job, it should not be displaced.



## Processor Information

### Number of Processors

The user will specify the number of processors at the beginning of the program.  Each processor will run exactly the same way.  The purpose of the program is to determine what number of processors is appropriate for this system.

### Processor Behavior

#### Processor Time

The processors operate on a time system that represents each potential operation by the processor.  This system should run for 10,000 units of time, where 10,000 potential actions could be taken in that time.

#### Processor Queue/Heap

The queue/heap for the processor determines what item should be processed next.  The item that is at the top of the heap should be added to the next open processor.  If the item is a high priority item, it should be placed into the processor immediately.

#### Order of Operations

1. Remove completed items from the processor
2. Add items from Job Stream to the Heap
3. Check for high priority jobs at top of the Heap.  Add to processor if it is a high priority job
   1. Add any removed items back to the Heap
4. Add regular jobs to an open processor
5. Process any items in a processor for a time of 1
6. Increment the system clock and go to the top of this list

## Log Information

### Log Setup

The log will be in a file called log.txt

The log will contain the following information.  Items will be separated by semicolons.

[Time Units])    [Job completion actions]; [Job arrival actions]; [Interrupt Actions]   
; [Adding to Processor Actions]; [Processing Actions]

### Job Information

All jobs will have the following information available

* Overall Job Number (count of all jobs)
* Job type number (count of jobs of the same type)
* Processor end time (what system time the job will complete at)
* Remaining time

### Time Units

Each line of the log represents a time unit.  The time unit will be indicated by the system time (ex: 3000)

### Job Completion Actions

Displays:

* Any jobs that are done at the beginning of the processor cycle.  Contains job type and job type number

Example

Job B 30 Completed

### Job Arrival Actions

Displays:

* job type
* overall job number
* job type number
* processing time information

Example

Arrival Job C: Overall Job 40, Job C 4, Processing Time 13;

### Interrupt Actions

Displays:

* The job interrupted
* Total number of jobs interrupted
* Processor the high priority job goes into
* Interrupted job information

Example

Interrupt Job A 1535, Total Interrupted Jobs: 157, New high priority job goes into Processor 2, Job A 1535 Added to heap with processing time 1;

### Adding to Processor Actions

Displays:

* Heap Empty if no item in heap to add
* Job information and the processor it is added to
* Time of job completion

Example 1

Heap Empty

Example 2

Begin Processing Job B 3 in CPU 2, end time 50

### Processing Actions and Processor State

Displays:

* Active or inactive message
* Job name

Example:

CPU 1: Job A 3; CPU 2: Idle Time

## Final Report Metrics

* Final Queue size: <value every x time units>
* Average queue size: <value>
* Average time in queue: <value> time units.
* Idle time: <value> time units.
* Total number of jobs A arrived: <value>
* Total number of jobs B arrived: <value>
* Total number of jobs C arrived: <value>
* Total number of jobs D arrived: <value>
* Total wait time in queue: <value> time units.
* Maximum jobs in queue: <value>
* Jobs interrupted <value>
* Total jobs completed <value>
* Number of processor(s) used <value>
* Total number of time units the processors(s) run <value>
* Total time processor(s) spent processing is <value> time units

The final evaluation will be based on the values above, and an evaluation of whether the queue is growing or constant over time. These values will be at the bottom of the log

# 3. UML Diagram 1

# 

# 4. Decomposition Diagram

|  |  |  |
| --- | --- | --- |
| Input | Process | Outputs |
| Number of processors | Construct Job Stack | Jobs started |
|  | Construct Heap | Jobs completed |
|  | Assign Jobs | Jobs Interrupted |
|  | Process Jobs | Jobs arriving |
|  |  | Final report |

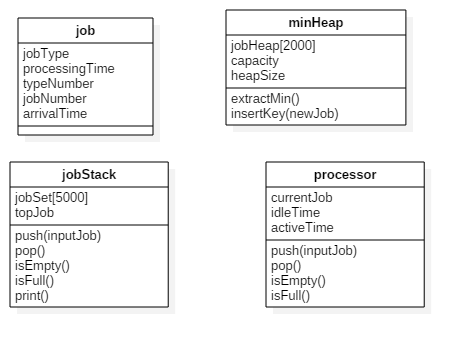
# 5. Test Strategies

* Job Stack Test – test that the job stack is created correctly
* Heap Test – test that the heap is created correctly
* Valid Inputs
* Invalid Inputs
* Many Processors Test – how many processors for no queue to build up?

# 6. Test Plan 1

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Job Stack Test | 1 | Job Stack Test - Just A jobs |  |  |  |  |
| Heap Test | 2 | Heap Test - Just A jobs |  |  |  |  |
| Job Stack Test | 3 | Job Stack Test - A and B Jobs |  |  |  |  |
| Heap Test | 4 | Heap Test - A and B Jobs |  |  |  |  |
| Job Stack Test | 5 | Job Stack Test - A, B, and C Jobs |  |  |  |  |
| Heap Test | 6 | Heap Test - A, B, and C Jobs |  |  |  |  |
| Valid Input | 7 | Process A, B, and C Jobs |  |  |  |  |
| Valid Input | 8 | Test as many processors as needed for no queue - A, B and C Jobs |  |  |  |  |
| Invalid Input | 9 | Negative number of processors |  |  |  |  |
| Job Stack Test | 10 | Job Stack Test - All Jobs |  |  |  |  |
| Heap Test | 11 | Heap Test - All Jobs |  |  |  |  |
| Valid Input | 12 | Process All Job types - 1 processors |  |  |  |  |
| Valid Input | 13 | Process All Job types - 2 processors |  |  |  |  |
| Valid Input | 14 | Process All Job types - 3 processors |  |  |  |  |
| Valid Input | 15 | Test as many processors as needed for no queue - All jobs |  |  |  |  |

# 8. UML Diagram 2



# 9. Algorithm

## Program Main

### Create Job Stack

createJobStack on a instance of jobStack called inputJob

### Create Heap

create instance of minHeap called jobHeap

#### Create Log file

Create an output file stream with handle logFile and filename log.txt

### Create array of processors

Prompt user for the number of processors to test

* How many processors do you want to use in this test?
  + Valid inputs >0
    - Create an array of processor objects called CPU of the size the user input
  + Invalid Inputs
    - Output "Invalid Input"
    - Reprompt

### Program Loop

For all time values in the program (1-10000)

#### De-load Completed Jobs

If the CPU objects have an active job present (isEmpty = false)

* if isComplete = true
  + call completeJob on that CPU
  + if time>500, Log Job Information
    - Ex. Job B 30 Completed;

#### Add to heap

While time = the time of the top object in the stack

* pop the object off the stack
* Add to the Heap (call insertKey with object as parameter)
* Log information
  + Ex. Arrival Job C: Overall Job 40, Job C 4, Processing Time 13;

#### Interrupt for Type D jobs

* For all the cpu's
  + If the top object in the heap is higher priority than was is in the processor and is of type D
    - Remove object from top of jobHeap (call extractMin)
    - Pop the current item from the processor
    - Push the item from the jobHeap
    - return the object currently in the cpu to the heap
      * heap that item up
    - Log interrupt and reheaping of object
      * Ex. Interrupt Job A 1535, Total Interrupted Jobs: 157, New high priority job goes into Processor 2, Job A 1535 Added to heap with processing time 1;

#### Add to CPU

* For all cpu's
  + if isEmpty is true
    - Call extract on item at top of heap to the cpu (call push with the job as the parameter)
      * Log action
        + Ex. Begin Processing Job B 3 in CPU 2, end time 50
    - If no item at top of heap
      * Log empty heap
        + Heap Empty

#### Process

* For all cpu's
  + if isEmpty is false
    - Call processOne on that job
  + if isEmpty is true
    - Log idle time
      * CPU<value> Idle Time

### Create Report

Create report (to screen and log) with this structure

* Final Queue size: <value every x time units>
* Average queue size: <value>
* Average time in queue: <value> time units.
* Idle time: <value> time units.
* Total number of jobs A arrived: <value>
* Total number of jobs B arrived: <value>
* Total number of jobs C arrived: <value>
* Total number of jobs D arrived: <value>
* Total wait time in queue: <value> time units.
* Maximum jobs in queue: <value>
* Jobs interrupted <value>
* Total jobs completed <value>
* Number of processor(s) used <value>
* Total number of time units the processors(s) run <value>
* Total time processor(s) spent processing is <value> time units

## Report Data Structure

Name: programReportData

Description: contains data values used in final evaluation of performance

* int finalQueueSize
* float avgQueueSize
* float avgTimeInQueue
* int idleTime
* int totalA\_Arrived
* int totalB\_Arrived
* int totalC\_arrived
* int totalD\_arrived
* int totalWaitTime
* int maxQueueLength
* int jobsInterrupted
* int totalComplete
* int numProcessors
* int totalTime
* int totalActiveTime

## Creating the Job Stack

### Method: createJobStack

* Description: Creates the job stack
* Precondition: a jobStack object
* Postcondition: returns the job stack
* for time values 1 to 10000
  + create a jobStack object called temp
  + A type jobs
    - if time%5 = 0
      * Push a job of type A with these attributes
        + Arrival time of current loop iterator + (4+rand()%3)
        + Processing time of 1+rand()%5
  + B type Jobs
    - if time%10 = 0
      * Push a job of type B with these attributes
        + Arrival time of current loop iterator + (9+rand()%3)
        + Processing time of 6+rand()%5
  + C type Jobs
    - if time%25 = 0
      * Push a job of type C with these attributes
        + Arrival time of current loop iterator + (25+rand()%3)
        + Processing time of 11+rand()%5
  + D type Jobs
    - if time%30 = 0
      * Push a job of type D with these attributes
        + Arrival time of current loop iterator + (25+rand()%11)
        + Processing time of 8+rand()%5
* For all the values in the temp jobStack
  + Pop them off of the tempJobStack
  + Push them on to the input jobStack

## Job Stream/Stack

### Data Structures

#### **Job**

Data Elements

* Char - jobType
* Int - processingTime
* Int - typeNumber
* Int - jobNumber
* Int - arrivalTime

**Private Data Elements**

* Array of jobs with 5000 elements
* topJob - an integer tracking the index of the top job

### Methods

**Push**

* Description: Adds an item to the stack
* Precondition: An existing array of Job
* Postcondition:  Item added to top of the stack

**Pop**

* Description: Removes an item from the stack
* Precondition: An existing array of Job
* Postcondition:  Item is removed from the stack and returned

**isEmpty**

* Description: Returns true when the stack is empty
* Precondition: A job stack
* Postcondition: True if empty, false if not

## Processor Class

**Private Data Elements**

* Job object currentJob
* int idleTime
* int activeTime

### Methods

**Push**

* Description: Adds an item to the stack
* Precondition: Job item as input
* Postcondition:  Item added to top of the stack

**Pop**

* Description: Removes an item from the stack
* Precondition: item in the cpu stack
* Postcondition:  Item is removed from the stack and returned

**isEmpty**

* Description: Returns true when the stack is empty
* Precondition: A job stack
* Postcondition: True if empty, false if not

**isFull**

* Description: Returns true when the stack is full (max of 1)
* Precondition: A job stack
* Postcondition: True if full, false if not

## Heap Structure

Stores the priority queue of objects for processing

**Private Data Values**

* job heapArray[2000] - a heap for storing job objects
* int capacity - maximum size
* int heap\_size - current size of the heap

Public Methods

**extractMin**

* Returns the minimum element

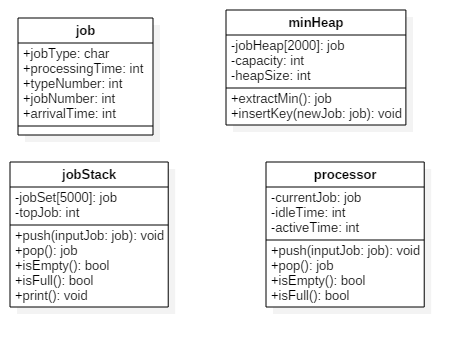
**insertKey**

* inserts a job into the heap

# 10. Test Plan 2

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Job Stack Test | 1 | Job Stack Test - Just A jobs | Just A jobs | Jobs in semi-random order |  |  |
| Heap Test | 2 | Heap Test - Just A jobs | Just A jobs | Jobs in heap order (minimum at top) |  |  |
| Job Stack Test | 3 | Job Stack Test - A and B Jobs | A and B Jobs | Jobs in semi-random order |  |  |
| Heap Test | 4 | Heap Test - A and B Jobs | A and B Jobs | Jobs in heap order (minimum at top) |  |  |
| Job Stack Test | 5 | Job Stack Test - A, B, and C Jobs | A, B, and C Jobs | Jobs in semi-random order |  |  |
| Heap Test | 6 | Heap Test - A, B, and C Jobs | A, B, and C Jobs | Jobs in heap order (minimum at top) |  |  |
| Valid Input | 7 | Process A, B, and C Jobs | A, B, and C Jobs 2 Processors | Log - should see short jobs processed more quickly, probably some queue build up at this level |  |  |
| Valid Input | 8 | Test as many processors as needed for no queue - A, B and C Jobs | A, B, and C Jobs As many processors as needed | Log - should see short jobs processed more quickly, no queue build up |  |  |
| Invalid Input | 9 | Negative number of processors | -1 processor | Invalid Input |  |  |
| Job Stack Test | 10 | Job Stack Test - All Jobs | All Jobs | Jobs in semi-random order |  |  |
| Heap Test | 11 | Heap Test - All Jobs | All Jobs | Jobs in heap order (D jobs at top) |  |  |
| Valid Input | 12 | Process All Job types - 1 processors | All Jobs 1 Processor | Log - should see short jobs processed more quickly, D jobs interrupting, should be a long queue |  |  |
| Valid Input | 13 | Process All Job types - 2 processors | All Jobs 2 Processors | Log - should see short jobs processed more quickly, D jobs interrupting, queue should be shorter than 1 processor |  |  |
| Valid Input | 14 | Process All Job types - 3 processors | All Jobs 3 Processors | Log - should see short jobs processed more quickly, D jobs interrupting, queue should be shorter than 2 processors |  |  |
| Valid Input | 15 | Test as many processors as needed for no queue - All jobs | All Jobs As many processors as needed | Log - should see short jobs processed more quickly, no queue build up |  |  |

# 11. UML Diagram 3



# 12. Code

// assignment 4.cpp : Tests a systems capacity to handle 4 types of jobs in a user defined setup of processors

// Programmer: Mark Ferrall

// Date: 4/23/2018

#include "stdafx.h"

#include "jobStack.h"

#include "minHeap.h"

#include <random>

#include "Processor.h"

#include <fstream>

#include <iostream>

using namespace std;

void createJobStack(jobStack &inputJobs);

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, dCount = 0;

int activeCycle = 0, totalProcessingTime = 0;

int jobsInterrupted = 0;

int totalWaitTime = 0;

float averageWaitTime = 0;

int userCPU;

createJobStack(inputJobs);

//inputJobs.print();

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;

case 'D':

dCount++;

break;

}

}

}

//interrupt for type D jobs

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

if (CPU[i\_cpu].isEmpty() || jobHeap.size() == 0) {

continue;

}

job temp = jobHeap.getMin();

if (temp.jobType == 'D') {

job displacedJob;

job Djob;

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

Djob = jobHeap.extractMin();

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

jobHeap.insertKey(displacedJob);

if (time >= 500) {

jobsInterrupted++;

logFile << "Interrupt Job " << displacedJob.jobType << " " << displacedJob.typeNumber

<< ", Total Interrupted Jobs: " << jobsInterrupted << ", New high priority job goes into Processor " << i\_cpu + 1

<< ", Job " << displacedJob.jobType << " " << displacedJob.typeNumber

<< "Added to heap with processing time " << displacedJob.processingTime << ";";

}

}

else {

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;

}

}

//should print blank, comment out

//inputJobs.print();

//jobHeap.print();

//final report

int totalJobsArrived = aCount + bCount + cCount + dCount;

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 number of jobs D arrived: " << dCount << 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 number of jobs D arrived: " << dCount << 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();

//jobHeap.print();

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

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

system("pause");

return 0;

}

//Description: Creates the job stack

//Precondition : a jobStack object

//Postcondition : 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 };

job tempD = { 'D', 0,0,0,0, 0 };

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

//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);

}

}

//create type D jobs

if (time % 30 == 0) {

tempD.arrivalTime = time + 24 + rand() % 11;

if (tempD.arrivalTime < 10000) {

jobCount++;

tempD.jobNumber = jobCount;

tempD.processingTime = 8 + rand() % 5;

tempD.typeNumber++;

tempJobs.push(tempD);

}

}

}

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

inputJobs.push(tempJobs.pop());

}

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

}

#pragma once

#include <iostream>

struct job {

char jobType;

int processingTime;

int typeNumber;

int jobNumber;

int arrivalTime;

int waitTime = 0;

};

class jobStack

{

private:

job jobSet[5000];

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 getCount() { return jobCount; }

~jobStack();

};

#pragma once

#include <iostream>

#include "jobStack.h"

// A class for Min Heap

class MinHeap

{

job \*heaparray; // pointer to array of elements in heap

int capacity; // maximum possible size of min heap

int heap\_size; // Current number of elements in min heap

public:

// Constructor

MinHeap(int capacity);

// to heapify a subtree with root at given index

void MinHeapify(int);

// to get index of parent of node at index i

int parent(int i) { return (i - 1) / 2; }

// to get index of left child of node at index i

int left(int i) { return (2 \* i + 1); }

// to get index of right child of node at index i

int right(int i) { return (2 \* i + 2); }

// to extract the root which is the minimum element

job extractMin();

// Returns the minimum key (key at root) from min heap

job getMin() { return heaparray[0]; }

// 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();

};

#pragma once

#include "jobStack.h"

class Processor

{

private:

job currentJob;

int jobCount = 0;

public:

Processor();

void push(job inputJob);

job pop();

bool isEmpty();

bool isFull();

bool isComplete();

void processJobOne();

job peekJob() { return currentJob; }

~Processor();

};

#include "stdafx.h"

#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

low --> Starting index,

high --> Ending index \*/

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

{

if (low < high)

{

/\* pi is partitioning index, arr[p] is now

at right place \*/

int pi = partition(low, high);

// Separately sort elements before

// partition and after partition

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;

}

/\* This function takes last element as pivot, places

the pivot element at its correct position in sorted

array, and places all smaller (smaller than pivot)

to left of pivot and all greater elements to right

of pivot \*/

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

{

job pivot = jobSet[high]; // pivot

int i = (low - 1); // Index of smaller element

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

{

// If current element is smaller than or

// equal to pivot

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

{

i++; // increment index of smaller element

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

}

}

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

return (i + 1);

}

jobStack::~jobStack()

{

}

#include "stdafx.h"

#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

Used for reporting purposes\*/

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

// This method assumes that the subtrees are already heapified

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);

}

}

#include "stdafx.h"

#include "Processor.h"

Processor::Processor()

{

}

//Description: Adds an item to the stack

//Precondition : Job item as input

//Postcondition : Item added to top of the stack

void Processor::push(job inputJob)

{

if (jobCount == 0) {

currentJob = inputJob;

jobCount++;

}

}

//Description: Removes an item from the stack

//Precondition : item in the cpu stack

//Postcondition : Item is removed from the stack and returned

job Processor::pop()

{

job tempJob = currentJob;

currentJob = {};

jobCount--;

return tempJob;

}

//Description: Returns true when the stack is empty

//Precondition : A processor

//Postcondition : True if empty, false if not

bool Processor::isEmpty()

{

if (jobCount == 0) {

return true;

}

else {

return false;

}

}

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

//Precondition : A processor

//Postcondition : True if full, false if not

bool Processor::isFull()

{

if (jobCount == 1) {

return true;

}

else {

return false;

}

}

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

//Precondition : An active job in a processor

//Postcondition : True if complete, false if not

bool Processor::isComplete()

{

if (currentJob.processingTime==0) {

return true;

}

else {

return false;

}

}

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

//Precondition : An active job in a processor

//Postcondition : Decreases the job amount 1

void Processor::processJobOne(){

currentJob.processingTime--;

}

Processor::~Processor()

{

}

# 13. Updated Algorithm

## Program Main

### Create Job Stack

createJobStack on a instance of jobStack called inputJob

### Create Heap

create instance of minHeap called jobHeap

#### Create Log file

Create an output file stream with handle logFile and filename log.txt

### Create array of processors

Prompt user for the number of processors to test

* How many processors do you want to use in this test?
  + Valid inputs >0
    - Create an array of processor objects called CPU of the size the user input
  + Invalid Inputs
    - Output "Invalid Input"
    - Reprompt

### Program Loop

For all time values in the program (1-10000)

#### De-load Completed Jobs

For all CPU objects

If the CPU objects have an active job present (~~isEmpty = false )~~ (isComplete = true and isFull=false)

* if isComplete = true
  + call ~~completeJob~~ pop on that CPU
  + if time>500, Log Job Information
    - Ex. Job B 30 Completed;

#### Add to heap

While time = the time of the top object in the stack

* pop the object off the stack
* Add to the Heap (call insertKey with object as parameter)
* If time >500
  + Log information
    - Ex. Arrival Job C: Overall Job 40, Job C 4, Processing Time 13;
  + Increment a count of the job type that arrived

#### Interrupt for Type D jobs

* For all the cpu's
  + ~~If the top object in the heap is higher priority than was is in the processor and is of type D~~
  + If isEmpty on the CPU is false or the heap size is 0, skip to the next CPU
  + Call getMin on the heap, and if the jobType is D
    - Remove object from top of jobHeap (call extractMin)
    - Pop the current item from the processor
    - Push the item from the jobHeap
    - return the object currently in the cpu to the heap
      * Insert using insertKey
    - If time >500
      * Log interrupt and reheaping of object
        + Ex. Interrupt Job A 1535, Total Interrupted Jobs: 157, New high priority job goes into Processor 2, Job A 1535 Added to heap with processing time 1;

#### Add to CPU

* For all cpu's
  + if isEmpty is true
    - Call extractMin on item at top of heap to the cpu (call push with the job as the parameter)
      * If time >500, Log action
        + Ex. Begin Processing Job B 3 in CPU 2, end time 50
    - If no item at top of heap
      * Log empty heap
        + Heap Empty

#### Process

* For all cpu's
  + if isEmpty is false
    - Call processJobOne on that job
  + if isEmpty is true
    - Log idle time
      * CPU<value> Idle Time
      * Increase a count of total Idle time

### Create Report

Create report (to screen and log) with this structure

* Final Queue size: <value every x time units>
* Average queue size: <value>
* Average time in queue: <value> time units.
* Idle time: <value> time units.
* Total number of jobs A arrived: <value>
* Total number of jobs B arrived: <value>
* Total number of jobs C arrived: <value>
* Total number of jobs D arrived: <value>
* Total wait time in queue: <value> time units.
* Maximum jobs in queue: <value>
* Jobs interrupted <value>
* Total jobs completed <value>
* Number of processor(s) used <value>
* Total number of time units the processors(s) run <value>
* Total time processor(s) spent processing is <value> time units

## ~~Report Data Structure~~

~~Name: programReportData~~

~~Description: contains data values used in final evaluation of performance~~

* ~~int finalQueueSize~~
* ~~float avgQueueSize~~
* ~~float avgTimeInQueue~~
* ~~int idleTime~~
* ~~int totalA\_Arrived~~
* ~~int totalB\_Arrived~~
* ~~int totalC\_arrived~~
* ~~int totalD\_arrived~~
* ~~int totalWaitTime~~
* ~~int maxQueueLength~~
* ~~int jobsInterrupted~~
* ~~int totalComplete~~
* ~~int numProcessors~~
* ~~int totalTime~~
* ~~int totalActiveTime~~

## Creating the Job Stack

### Method: createJobStack

* Description: Creates the job stack
* Precondition: a jobStack object
* Postcondition: returns the job stack
* for time values 1 to 10000
  + create a jobStack object called temp
  + A type jobs
    - if time%5 = 0
      * Push a job of type A with these attributes
        + Arrival time of current loop iterator + (4+rand()%3)
        + Processing time of 1+rand()%5
  + B type Jobs
    - if time%10 = 0
      * Push a job of type B with these attributes
        + Arrival time of current loop iterator + (9+rand()%3)
        + Processing time of 6+rand()%5
  + C type Jobs
    - if time%25 = 0
      * Push a job of type C with these attributes
        + Arrival time of current loop iterator + (24+rand()%3)
        + Processing time of 11+rand()%5
  + D type Jobs
    - if time%30 = 0
      * Push a job of type D with these attributes
        + Arrival time of current loop iterator + (24+rand()%11)
        + Processing time of 8+rand()%5
* For all the values in the temp jobStack
  + Pop them off of the tempJobStack
  + Push them on to the input jobStack
* Call quicksort on the stack

## Job Stream/Stack

### Data Structures

#### **Job**

Data Elements

* Char - jobType
* Int - processingTime
* Int - typeNumber
* Int - jobNumber
* Int – arrivalTime
* Int - waitTime

**Private Data Elements**

* Array of jobs with 5000 elements
* topJob - an integer tracking the index of the top job

### Methods

**Push**

* Description: Adds an item to the stack
* Precondition: An existing array of Job
* Postcondition:  Item added to top of the stack
* Increment jobCount and add the inputJob at location jobCount - 1

**Pop**

* Description: Removes an item from the stack
* Precondition: An existing array of Job
* Postcondition:  Item is removed from the stack and returned
* Return the item at location jobCount -1
* Decrement jobCount
* Set the object at the previous top location equal to 0’s and spaces

**isEmpty**

* Description: Returns true when the stack is empty
* Precondition: A job stack
* Postcondition: True if empty, false if not

**isFull**

* Returns true if jobCount = 5000, false otherwise

**Print**

* Prints out all the objects in the jobStack
* Format:
  + Type:[], Arrival Time:[], Job Number: [], Type Number: [], Processing Time: []

**PeekArrivalTime**

* Return the arrival time of the object at location jobCount -1

**QuickSort**

* Description: A sort function
* Precondition: two integer locations, high and low (initial condition is the first and last active location in the array)
* PostCondition: A sorted array
* If low is less than high
  + Set a partition index variable equal to the result of partition(low, high)
  + Call quicksort from low to partition index – 1
  + Call quicksort from partition index + 1 to high

**Swap**

* Swap items at two specified indices

**Partition**

## This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot

* Set a job object pivot equal to the high location
* Set an integer i equal to low-1
* For the items between low and high locations
  + If the item at the current location has an arrival time greater than the arrival time at the pivot location
    - Increment i
    - Swap objects at i and the current location
* Swap the objects at location i+1 and high
* Return i+1

## Processor Class

**Private Data Elements**

* Job object currentJob
* Int jobCount -initialize to 0
* ~~int idleTime~~
* ~~int activeTime~~

### Methods

**Push**

* Description: Adds an item to the stack
* Precondition: Job item as input
* Postcondition:  Item added to top of the stack

**Pop**

* Description: Removes an item from the stack
* Precondition: item in the cpu stack
* Postcondition:  Item is removed from the stack and returned

**isEmpty**

* Description: Returns true when the stack is empty
* Precondition: A job stack
* Postcondition: True if empty, false if not

**isFull**

* Description: Returns true when the stack is full (max of 1)
* Precondition: A job stack
* Postcondition: True if full, false if not

**isComplete**

* Description: Returns true when job in the processor has 0 time remaining
* Precondition : An active job in a processor
* Postcondition : True if complete, false if not

**processJobOne**

* Description: Decreases the processing time in an active job in a processor
* Precondition : An active job in a processor
* Postcondition : Decreases the job amount 1

## Heap Structure

Stores the priority queue of objects for processing

**Private Data Values**

* job heapArray~~[2000]~~ - a heap for storing job objects
* int capacity - maximum size
* int heap\_size - current size of the heap

Public Methods

**extractMin**

* Removes and Returns the minimum element

**insertKey**

* inserts a job into the heap

**MinHeap**

* Constructor, takes an argument that creates an array of that integer input size

**MinHeapify**

* heapify a subtree with root at given index
* If either the left or right child node has a processing time less than that of the parent node
  + Swap the child and parent node

**Parent**

* Returns the index of an items parent node

**Left**

* Returns the index of an items left child node

**Right**

* Returns the index of an items right child node node

**isEmpty**

* Returns true if the heap size is 0

**Swap**

* Swaps the location of two items in the array

**Print**

* Prints the contents of all items in the heap in this format:
  + Type:[], Arrival Time:[], Job Number: [], Type Number: [], Processing Time: []

**size**

* Returns the heap size

**AddWaitTime**

* Increments the wait time value in all objects in the heap

**GetRemainingWaitTime()**

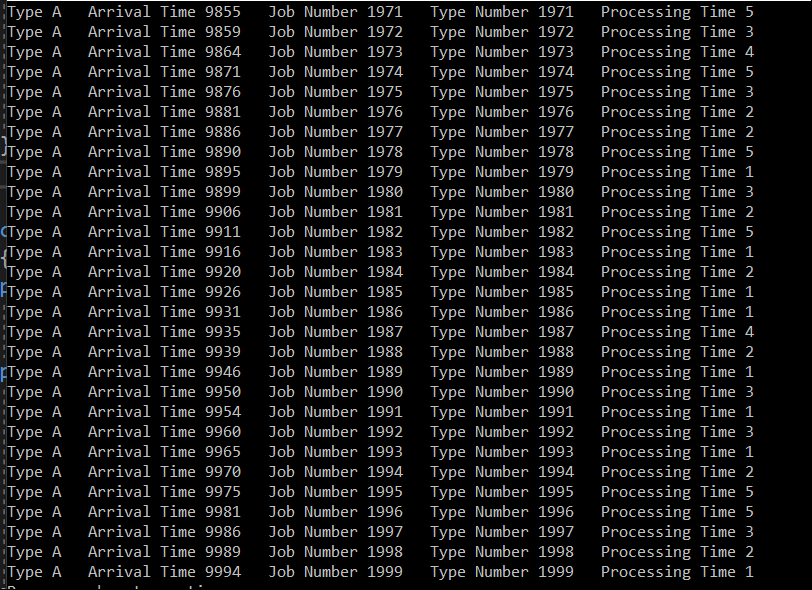
* Returns the sum of all the remaining wait times in the heap
* Used for reporting

# 14. Test Plan 3

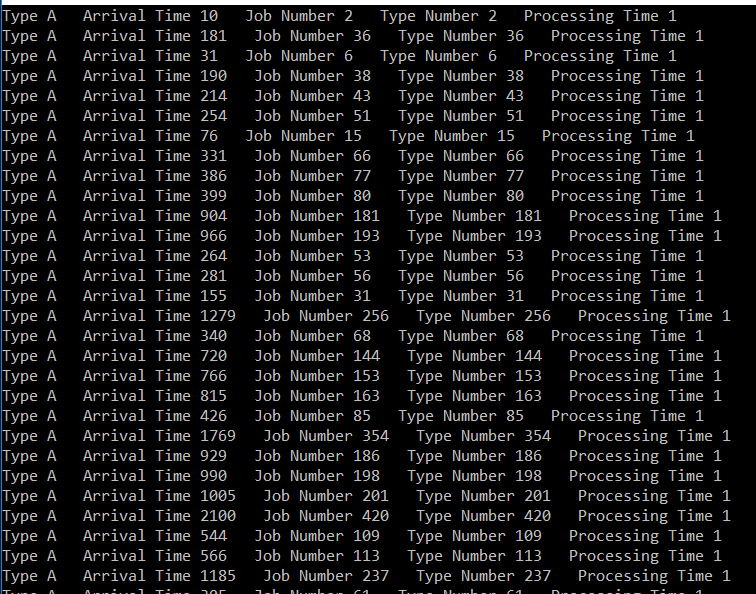
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Test Number | Description | Input | Expected Output | Actual Output | Pass/Fail |
| Job Stack Test | 1 | Job Stack Test - Just A jobs | Just A jobs | Jobs in semi-random order | Jobs in semi-random order | Pass |
| Heap Test | 2 | Heap Test - Just A jobs | Just A jobs | Jobs in heap order (minimum at top) | Jobs in heap order (minimum at top) | Pass |
| Job Stack Test | 3 | Job Stack Test - A and B Jobs | A and B Jobs | Jobs in semi-random order | Jobs in semi-random order | Pass |
| Heap Test | 4 | Heap Test - A and B Jobs | A and B Jobs | Jobs in heap order (minimum at top) | Jobs in heap order (minimum at top) | Pass |
| Job Stack Test | 5 | Job Stack Test - A, B, and C Jobs | A, B, and C Jobs | Jobs in semi-random order | Jobs in semi-random order | Pass |
| Heap Test | 6 | Heap Test - A, B, and C Jobs | A, B, and C Jobs | Jobs in heap order (minimum at top) | Jobs in heap order (minimum at top) | Pass |
| Valid Input | 7 | Process A, B, and C Jobs | A, B, and C Jobs 2 Processors | Log - should see short jobs processed more quickly, probably some queue build up at this level | Log - short jobs are prioritized, no queue build up at 2 | Pass |
| Valid Input | 8 | Test as many processors as needed for no queue - A, B and C Jobs | A, B, and C Jobs As many processors as needed | Log - All Jobs should process immediately | Log - All Jobs should process immediately | Pass |
| Invalid Input | 9 | Negative number of processors | -1 processor | Invalid Input | Invalid Input | Pass |
| Job Stack Test | 10 | Job Stack Test - All Jobs | All Jobs | Jobs in semi-random order | Jobs in semi-random order | Pass |
| Heap Test | 11 | Heap Test - All Jobs | All Jobs | Jobs in heap order (D jobs at top) | Jobs in heap order (D jobs at top) | Pass |
| Valid Input | 12 | Process All Job types - 1 processors | All Jobs 1 Processor | Log - should see short jobs processed more quickly, D jobs interrupting, should be a long queue | Log - should see short jobs processed more quickly, D jobs interrupting, should be a long queue | Pass |
| Valid Input | 13 | Process All Job types - 2 processors | All Jobs 2 Processors | Log - should see short jobs processed more quickly, D jobs interrupting, queue should be shorter than 1 processor | Log - should see short jobs processed more quickly, D jobs interrupting, queue should be shorter than 1 processor | Pass |
| Valid Input | 14 | Process All Job types - 3 processors | All Jobs 3 Processors | Log - should see short jobs processed more quickly, D jobs interrupting, queue should be shorter than 2 processors | Log - should see short jobs processed more quickly, D jobs interrupting, no queue at end of run | Pass |
| Valid Input | 15 | Test as many processors as needed for no queue - All jobs | All Jobs As many processors as needed | Log - should see short jobs processed more quickly, no queue build up | Log - should see short jobs processed more quickly, no queue build up - takes 5 processors | Pass |
| Valid Input | 16 | Process A, B, and C Jobs | A, B, and C Jobs 1 Processor | Log - should see short jobs processed more quickly, Significant Queueing | Log - should see short jobs processed more quickly, Significant Queueing | Pass |
| Valid Input - Half time queue test | 17 | Process all jobs, stop at 5000 to examine if queue is static or growing with 2 processors | All Jobs, time only 5000 | Log - same as full for 2 processors, looking to see if queue is near the 180 mark of test 13 (no increase in queue), or if it is close to half | Queue is close to half of test 13, implies that the queue would continue to grow | Pass |

# 15. Screenshots

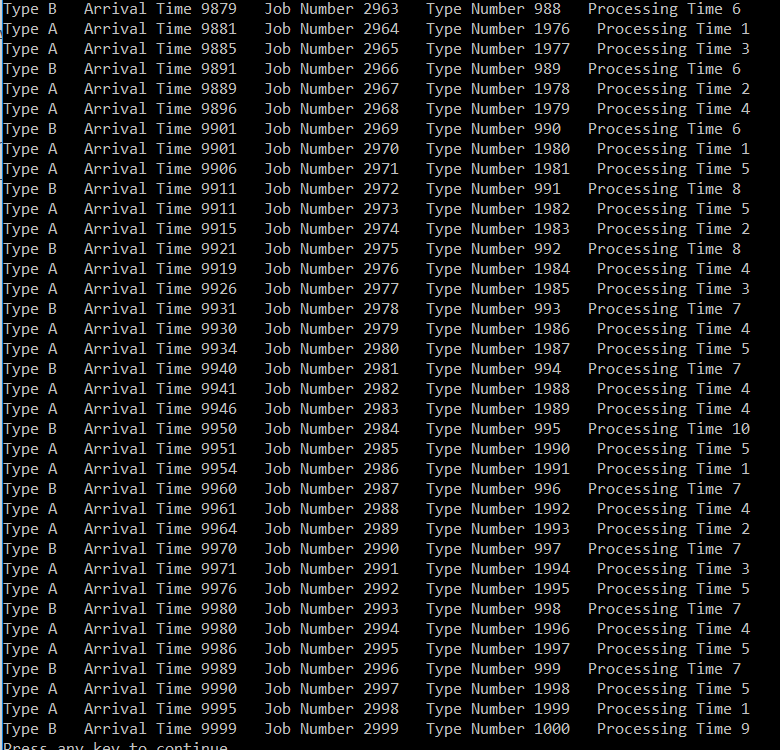
Test case 1



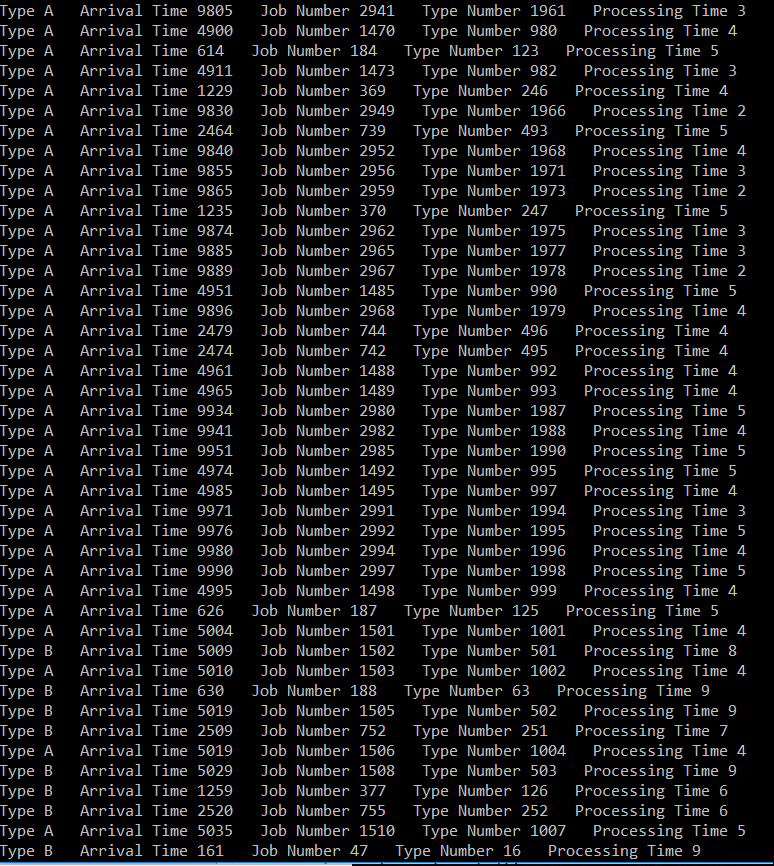
Test Case 2



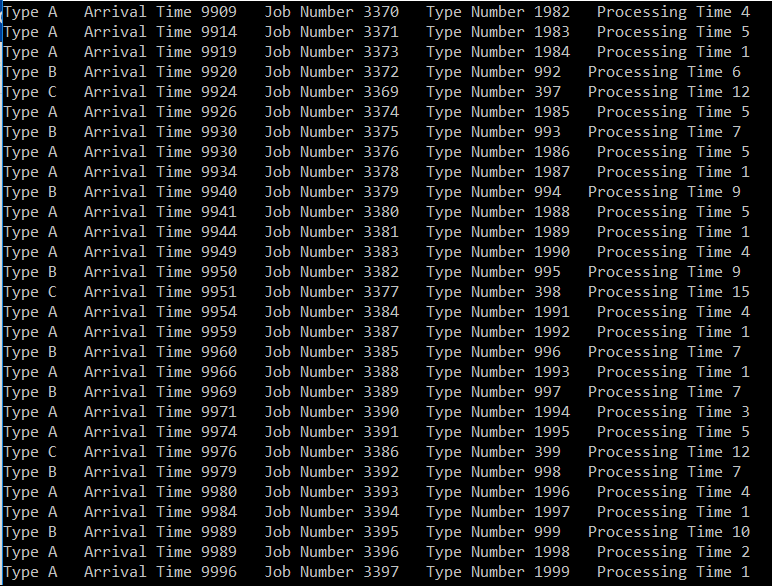
Test Case 3



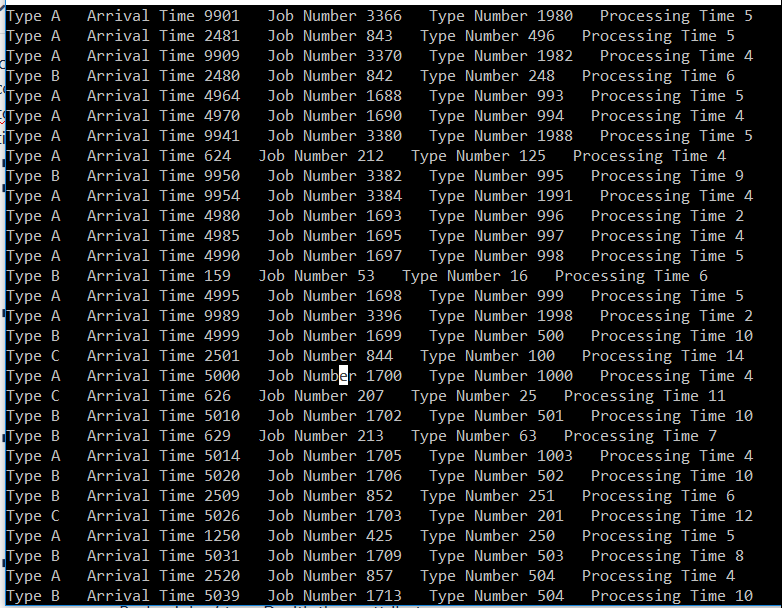
Test Case 4



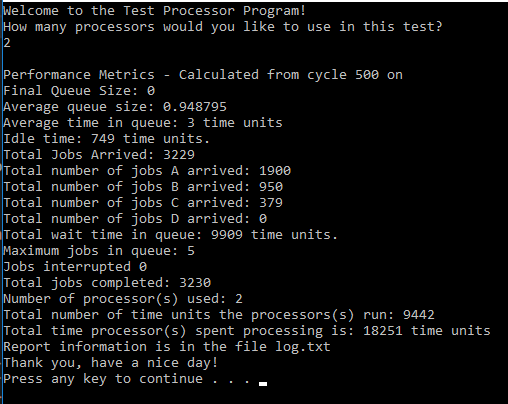
Test Case 5



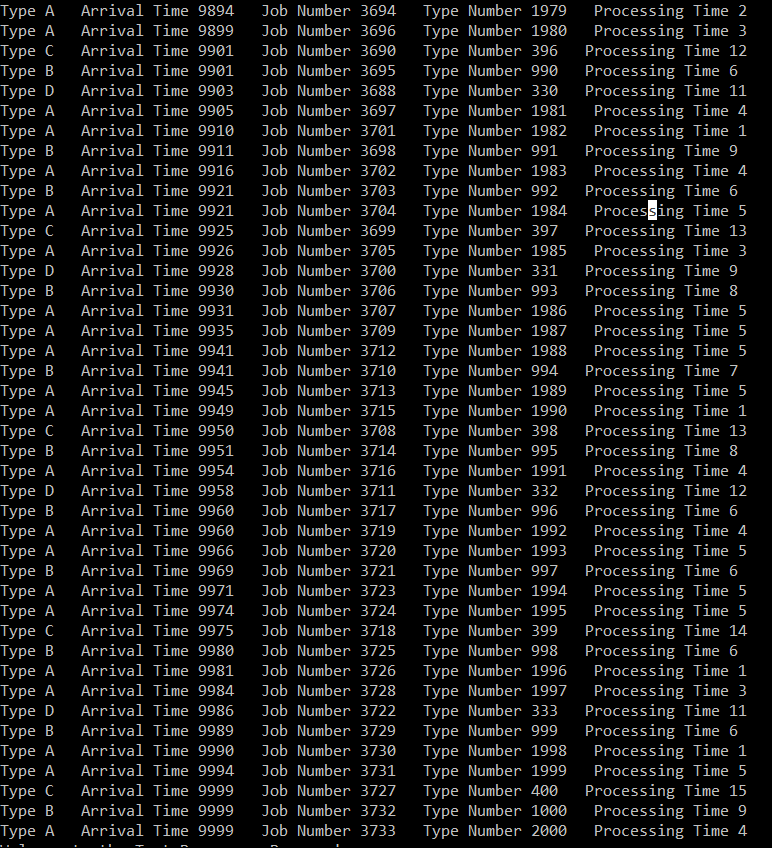
Test Case 6



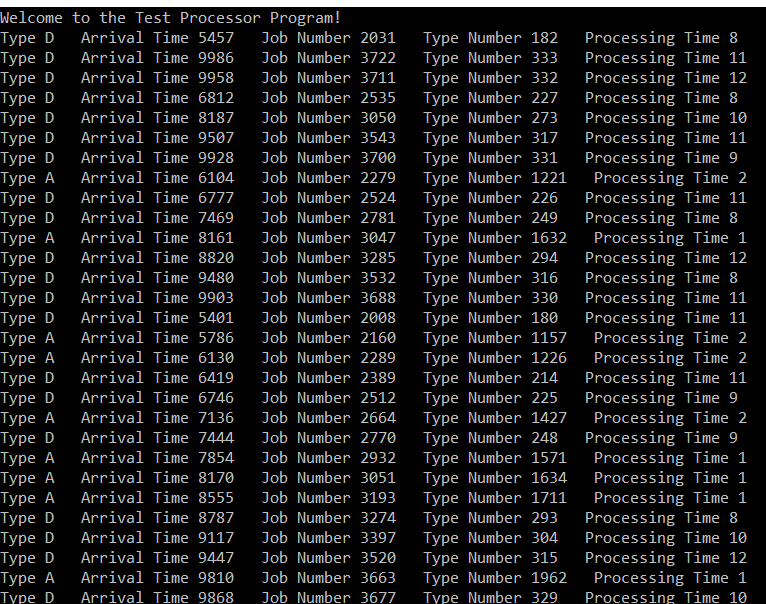
Test Case 7



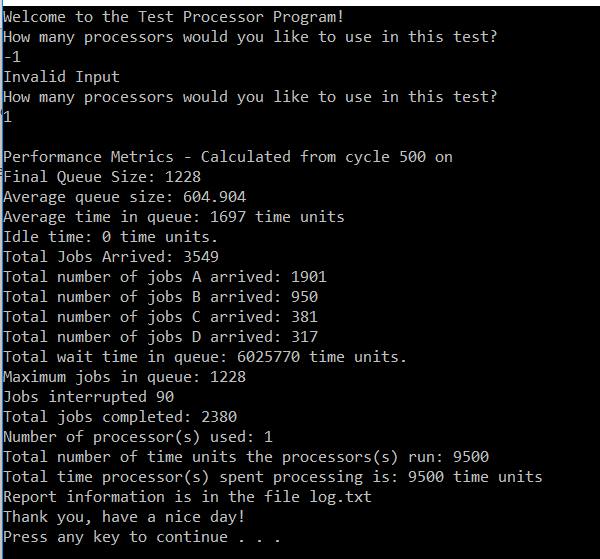
Test Case 10



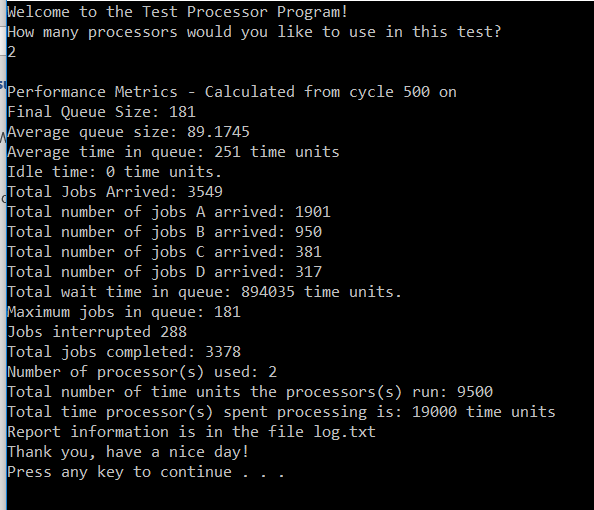
Test Case 11



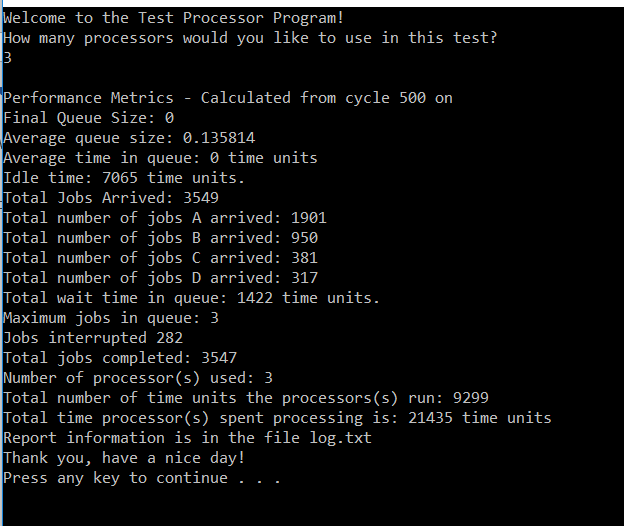
Test Case 9 and 12



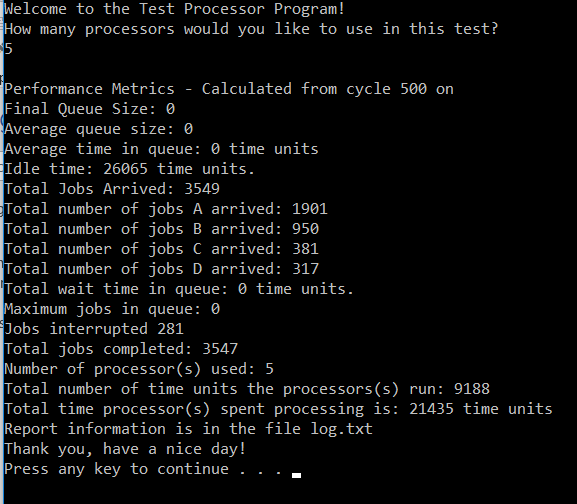
Test Case 13



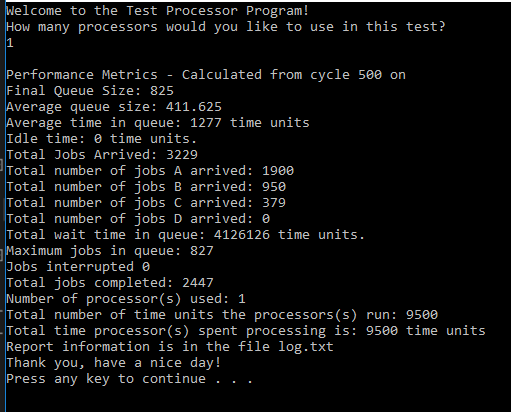
Test Case 14



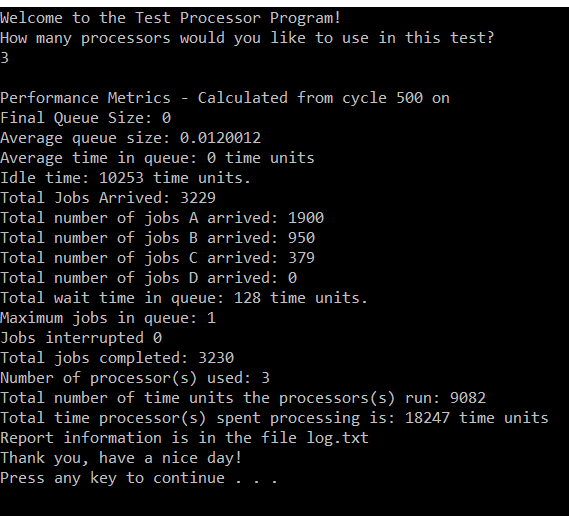
Test Case 15



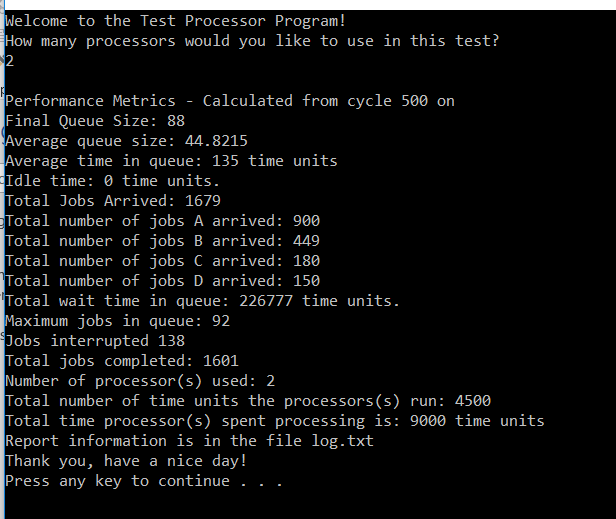
Test Case 16



Test Case 8



Test Case 17



# 16. Error Log

|  |  |  |
| --- | --- | --- |
| Error type | Cause of Error | Solution to error |
| None |  |  |

# 17. Final Status

The program works 100% according to the specifications and assumptions.