# Csc 326 lab 4

## Cpp file

```
#include <iostream>

#include <fstream>

#include <queue>

#include <assert.h>


#if 1

static constexpr size_t kDefaultQueueCapacity = 10;

enum class ProcessState { ARRIVED, SCHEDULED, COMPLETE };

struct Process   // creating struct..

{

   Process* next = nullptr;

          int index = 0;

          char CPU = 0;

          int arrival_time = 0;        //time of arrival

          int actual_start_time = 0;            / time that this process starts

          int service_time = 0;                 //amount of time required to service this event

   ProcessState queueState;      //the queue state of the process

};


static constexpr size_t size = sizeof(Process);


class Queue     // creating class

{

public:

   Queue();

   ~Queue();
```

```
        bool isEmpty();
        void Enqueue(Process& data);
    Process* Dequeue();

private:
        int head = 0;
        int tail = 0;
        int capacity = 0;
        int count = 0;

    Process* front;
    Process* back;
};

Queue::Queue()
{
    front = nullptr;
        back = nullptr;
}

Queue::~Queue()
{
    Process* temp = nullptr;
    while (front)
        {
        temp = front;
        front = front->next;
```

```
        delete temp;
        temp = nullptr;
            }
}
bool Queue::isEmpty() {
            if (count == 0) {
        return true;
            }
            else {
        return false;
            }
}

void Queue::Enqueue(Process& data)
{
            if (!front)
            {
        front = new Process(data);
            }
            else if (!back)
            {
        back = new Process(data);
        front->next = back;
            }
            else
            {
        back->next = new Process(data);
        back = back->next;
```

```
            }
    count++;
}

Process* Queue::Dequeue()
{
            if (count == 0)
            {
        std::cout << "There is nothing in the Queue, thus 0 will be returned" << std::endl;
        return nullptr;
            }

    Process* temp = front;
    front = front->next;
    temp->next = nullptr;
    count--;

    return temp;
}

int main()
{
            char character;
            int arrivalTime, serviceTime;
    Queue eventQueue{};
    std::ifstream infile;
            int processCount = 0;
```

```
//creating txt file to put elements in
infile.open("Jobs.txt");
        if (!infile)
        {
    std::cout << "Unable to open file";
    exit(1);
        }

infile >> processCount;
char* tempCPUID = new char[processCount];

        int index = 0;
while (!infile.eof())
        {
    Process temp{};
    infile.ignore(1, '\n');
    infile >> character;
    infile >> arrivalTime;
    infile >> serviceTime;
    tempCPUID[index] = character;
    temp.index = index;
    temp.CPU = character;
    temp.arrival_time = arrivalTime;
    temp.service_time = serviceTime;
    temp.queueState = ProcessState::SCHEDULED;
    eventQueue.Enqueue(temp);
    ++index;
        }
```

```
infile.close();



//Setup simulation data structure
        int* tempArrivalTime = new int[processCount];
        int* tempServiceTime = new int[processCount];
        int* tempTurnaroundTime = new int[processCount];
        int* tempEndTime = new int[processCount];
        int totalTimeRequired = 0;

//initialize C-arrays
        for (int i = 0; i < processCount; ++i)
        {
    tempArrivalTime[i] = 0;
    tempServiceTime[i] = 0;
    tempTurnaroundTime[i] = 0;
        }

//run simulation
        int simClock = 0;
        bool isBusy = false;
        Process* temp = nullptr;
Process* next = nullptr;



while (!eventQueue.isEmpty() || isBusy) {
    if (!isBusy)
    {
```

```
    delete temp;
    temp = nullptr;
    temp = eventQueue.Dequeue();
    assert(temp);
}
else
{
    ++simClock;
}
if (simClock == temp->service_time + temp->actual_start_time)
{
    temp->queueState = ProcessState::COMPLETE;
}

switch (temp->queueState)
{
case ProcessState::ARRIVED:

    break;
case ProcessState::SCHEDULED:

    isBusy = true;
    temp->queueState = ProcessState::ARRIVED;
    temp->actual_start_time = simClock;

    break;
case ProcessState::COMPLETE:
    isBusy = false;
```

```
        tempEndTime[temp->index] = simClock;
        tempArrivalTime[temp->index] = temp->arrival_time;
        tempServiceTime[temp->index] = temp->service_time;
        tempTurnaroundTime[temp->index] = (simClock - temp->arrival_time);
        break;
    }
        }
float averageWaitingTime = 0.f;
        for (int i = 0; i < processCount; ++i)
        {
    averageWaitingTime += (float)(tempTurnaroundTime[i] - tempServiceTime[i]);
        }

//report results
std::cout << "FCFS(non - preemptive) : " << std::endl;
totalTimeRequired = tempArrivalTime[processCount - 1] + tempTurnaroundTime[processCount - 1];
std::cout << "Total Time required is " << totalTimeRequired << " time units" << std::endl;


averageWaitingTime /= (float)processCount;
std::cout << "Average waiting time is " << averageWaitingTime << " time units" << std::endl;
std::cout << "" <<std::endl;

//report Ghantt Chart
std::cout << "Ghantt Chart" << std::endl;
std::cout << "Time | 0";
        for (int idx = 0; idx < processCount; ++idx) {
    std::cout << " | " << tempEndTime[idx];
```

```
        }
    std::cout << std::endl;
    std::cout << "CPU  | ";
        for (int idx = 0; idx < processCount; ++idx) {
      std::cout << tempCPUID[idx] << " | ";


        }
    std::cout << '-';
    std::cout << std::endl;


    //the report Process
        for (int idx = 0; idx < processCount; ++idx) {
      std::cout << "Process " << tempCPUID[idx] << " :" << std::endl;
      std::cout << "Service time = " << tempServiceTime[idx] << std::endl;
      std::cout << "Turnaround time = " << tempTurnaroundTime[idx] << std::endl;
        }
    delete[] tempCPUID;
    delete[] tempArrivalTime;
    delete[] tempServiceTime;
    delete[] tempTurnaroundTime;


    return 0;
}
 #endif
```

---

**Scheduling.txt**

**5**

**A 0 3**

**B 2 6**

**C 4 4**

**D 6 5**

**E 8 2**

---

**output**