* **I have copied assignments as it is from the three files.**
* **In Assignment 3, I have been given 40/60 marks. Two of my questions (Q2 and Q3) were correct whereas first question was very ambiguous but I did whatever I could understand**
* **In Assignment 4, I have been given no marks yet**
* **In Assignment 5, I have been given 2/3 marks even though I did all the questions and all of them were correct.**

**Assignment 3 (40/60 Marks)**

**Question 1 (Ambiguous Question)**

#include <iostream>

#include <pthread.h>

#include <stdlib.h>

#include <string>

#include <unistd.h>

using namespace std;

pthread\_cond\_t c = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

int done;

// THREAD Function

void \*Thread\_Fun(void \*arg) {

while (done == 0) // To avoid spurious wake up, done is used.

pthread\_cond\_wait(&c, &m); // Thread will wait for signal until time of sleep is completed or the alarm is over.

cout << "Thread has finished its execution" << endl;

pthread\_exit(NULL);

}

class Alarm\_Clock { // alarm Class

int time, hours, mins, secs, alarm;

friend void \*Thread\_Fun(void \*);

public:

Alarm\_Clock(int);

void GoToSleep(int);

void PrintTime();

void Timer();

};

Alarm\_Clock::Alarm\_Clock(int t) { // Constructor

done = 0;

alarm = t;

secs = t;

mins = t / 60;

secs = t % 60;

hours = mins / 60;

mins = mins % 60;

}

void Alarm\_Clock::GoToSleep(int ForHowLong) {

time = ForHowLong; // Go To Sleep Function...

}

// Function to Print alarm clock...

void Alarm\_Clock::Timer() {

while (alarm > 0) {

system("clear");

if (time <= 0 && done == 0) { // If Thread sleep time is completed.

pthread\_cond\_signal(&c);

done = 1;

cout << "Thread has been waken up" << endl;

}

else if (done == 0)

cout << "Thread is Sleeping" << endl;

else

cout << "Thread has waken up" << endl;

PrintTime();

alarm--;

secs--;

time--;

if (secs < 0) {

secs = 60;

mins--;

}

if (mins < 0) {

mins = 60;

hours--;

}

sleep(1);

if (alarm <= 0) // If Alarm is over, Snooze option will appear.

{

system("clear");

cout << endl;

PrintTime();

char y;

cout << "Snooze Alarm for 10 seconds (Y/N): ";

cin >> y;

switch (y)

{

case 'y':

case 'Y':

alarm = alarm + 10;

secs = secs + 10;

break; // In case any other key is pressed, program will come out

}

}

}

done = 1;

pthread\_cond\_broadcast(&c); // Wake up all Sleeping Threads

cout << "All Threads have waken up" << endl;

}

void Alarm\_Clock::PrintTime() { // Function to Alarm Clock

cout << "==================================\n"

<< "Alarm Time Left: " << hours << ":" << mins << ":" << secs << endl

<< "==================================\n";

}

int main() {

cout << "Enter Time of Alarm (in secs): ";

int x, y;

cin >> x;

Alarm\_Clock obj(x);

cout << "Enter Time of Sleep for Thread (in secs): ";

cin >> y;

obj.GoToSleep(y);

pthread\_t t1;

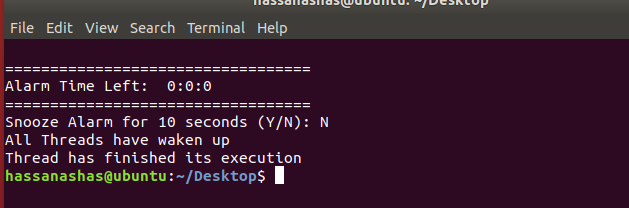
pthread\_create(&t1, NULL, Thread\_Fun, NULL);

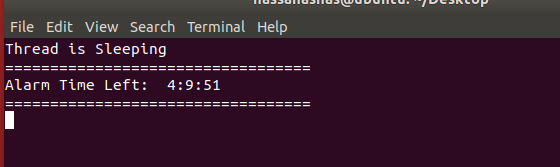
obj.Timer();

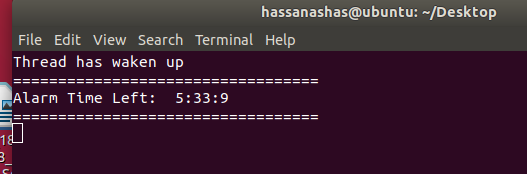
pthread\_join(t1, NULL);

return 0;

}







**Question 2**

#include <fstream>

#include <iomanip>

#include <iostream>

#include <pthread.h>

using namespace std;

struct node {

int data;

node \*next;

};

class Queue {

node \*front;

node \*rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int);

void display();

int peek();

};

int global\_variable;

Queue obj;

bool has\_finished; // To check 1st thread's status.

void \*read\_values(void \*arg);

void \*store\_values(void \*arg);

int main() {

global\_variable = -1; // Initializing it...

pthread\_t reading\_thread, storing\_thread;

has\_finished = false;

cout << "Initial Data: ";

for (int i = 0; i < 100; i++)

cout << setw(3) << i;

cout << endl << endl;

pthread\_create(&reading\_thread, NULL, read\_values, NULL);

pthread\_create(&storing\_thread, NULL, store\_values, NULL);

pthread\_join(reading\_thread, NULL);

pthread\_join(storing\_thread, NULL);

cout << endl << endl << "Data After Insertion: ";

obj.display();

cout << endl;

return 0;

}

void \*read\_values(void \*arg) {

fstream obj1;

obj1.open("Hello.txt");

if (!obj1) {

cout << "File doesn't exist";

exit(1);

}

while (1) {

obj1 >> global\_variable;

if (obj1.eof()) {

break;

}

}

has\_finished = true;

}

void \*store\_values(void \*arg) {

while (has\_finished == false) {

if (global\_variable != -1)

obj.enqueue(global\_variable);

}

}

Queue::Queue() {

front = NULL;

rear = NULL;

}

bool Queue::isEmpty() {

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int d) {

node \*temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty()) {

front = temp;

rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue() {

if (isEmpty()) {

return;

}

node \*temp = front;

front = front->next;

free(temp);

}

void Queue::display() {

if (isEmpty()) {

return;

}

Queue q;

while (!(this->isEmpty())) {

int temp = this->peek();

q.enqueue(temp);

this->dequeue();

cout << temp << " ";

}

while (!(q.isEmpty())) {

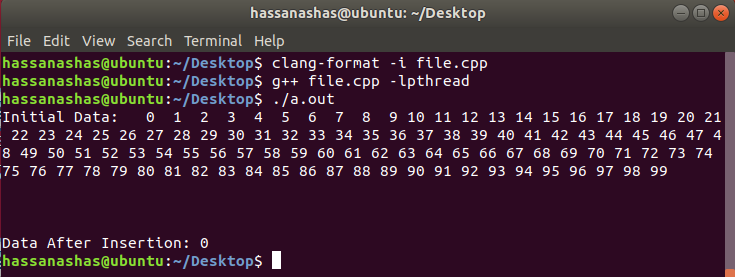
this->enqueue(q.peek());

q.dequeue();

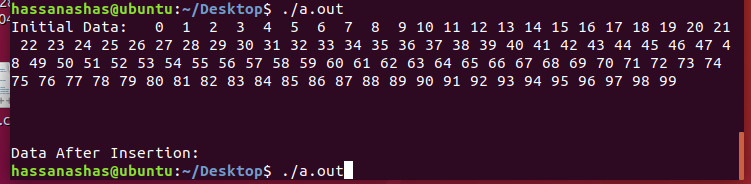
}

}

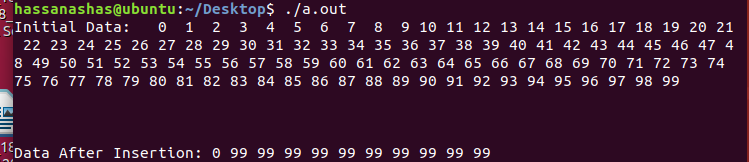
* Initially, the text file contains all the numbers from 0 to 99 (i.e. 100 numbers in total)
* Output of program on first try,



* Output of program on second try,



* Output of program on third try,



* Hence, it can be concluded that almost all the data goes on missing in the queue.

**Question 3**

#include <fstream>

#include <unistd.h>

#include <iostream>

#include <time.h>

#include <cstdlib>

#include <ctime>

#include <ctime>

#include <pthread.h>

using namespace std;

void Mergesort(int[], int, int, int);

void Merge(int[], int, int, int, int);

void Selection\_Sort(int \*, int);

void Insertion\_Sort(int \*, int);

void quicksort(int[], int, int);

int partition(int[], int, int);

void \*MergeSort(void \*arg);

void \*SelectionSort(void \*arg);

void \*QuickSort(void \*arg);

void \*InsertionSort(void \*arg);

const time\_t ctt = time(0);

int arr\_size;

int rand\_number();

int main() {

srand(time(0));

cout << "Enter size of your array: ";

cin >> arr\_size;

int \*arr = new int[arr\_size];

for (int i = 0; i < arr\_size; i++)

arr[i] = rand() % 10000; // Only taking in numbers from 0-10000

fstream obj1;

obj1.open("Unsorted\_00.txt", ios::out | ios::trunc);

for (int i = 1; i <= arr\_size / 4; i++)

obj1 << rand\_number() << " ";

obj1.close();

obj1.open("Unsorted\_01.txt", ios::out | ios::trunc);

for (int i = (arr\_size / 4) + 1; i <= arr\_size / 2; i++)

obj1 << rand\_number() << " ";

obj1.close();

obj1.open("Unsorted\_02.txt", ios::out | ios::trunc);

for (int i = (arr\_size / 2) + 1; i <= (arr\_size / 4) + (arr\_size / 2); i++)

obj1 << rand\_number() << " ";

obj1.close();

obj1.open("Unsorted\_03.txt", ios::out | ios::trunc);

for (int i = (arr\_size / 4) + (arr\_size / 2) + 1; i <= arr\_size; i++)

obj1 << rand\_number() << " ";

obj1.close();

pthread\_t merge, selection, quick, insertion;

pthread\_create(&merge, NULL, MergeSort, NULL);

pthread\_create(&selection, NULL, SelectionSort, NULL);

pthread\_create(&insertion, NULL, InsertionSort, NULL);

pthread\_create(&quick, NULL, QuickSort, NULL);

pthread\_join(merge, NULL);

pthread\_join(selection, NULL);

pthread\_join(insertion, NULL);

pthread\_join(quick, NULL);

fstream in, out;

int temp;

out.open("Sorted.txt", ios::out | ios::trunc);

in.open("merge.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> temp;

out << temp << " ";

if (in.eof())

break;

}

in.close();

in.open("selection.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> temp;

out << temp << " ";

if (in.eof())

break;

}

in.close();

in.open("insertion.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> temp;

out << temp << " ";

if (in.eof())

break;

}

in.close();

in.open("quick.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> temp;

out << temp << " ";

if (in.eof())

break;

}

in.close();

out.close();

system("pause");

return 0;

}

int rand\_number()

{

return rand() % 100000; // Numbers are kept within the 1000000 range.

}

void \*MergeSort(void \*arg)

{

clock\_t start, end;

double cpu\_time\_used;

start = clock();

cout << "Thread ID for Merge Sort: " << pthread\_self() << endl

<< "Algorithm Name: Merge Sort" << endl

<< "Start Time for Merge Sort: " << asctime(localtime(&ctt));

fstream in;

int index = 0;

int \*arr = new int[arr\_size / 4];

in.open("Unsorted\_00.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> arr[index++];

if (in.eof())

break;

}

in.close();

Mergesort(arr, 0, (arr\_size / 4) - 1, arr\_size / 4);

index = 0;

in.open("merge.txt", ios::out | ios::trunc);

for (int i = 0; i < arr\_size / 4; i++)

in << arr[index++] << " ";

in.close();

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

cout << "End Time for Merge Sort: " << asctime(localtime(&ctt))

<< "Time consumed for Merge Sort: " << cpu\_time\_used << " seconds." << endl << endl;

}

void \*QuickSort(void \*arg)

{

clock\_t start, end;

double cpu\_time\_used;

sleep(1);

start = clock();

cout << "Thread ID for Quick Sort: " << pthread\_self() << endl

<< "Algorithm Name: Quick Sort" << endl

<< "Start Time for Quick Sort: " << asctime(localtime(&ctt));

fstream in;

int index = 0;

int \*arr = new int[arr\_size / 4];

in.open("Unsorted\_03.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> arr[index++];

if (in.eof())

break;

}

in.close();

quicksort(arr, 0, (arr\_size / 4) - 1);

index = 0;

in.open("quick.txt", ios::out | ios::trunc);

for (int i = 0; i < arr\_size / 4; i++)

in << arr[index++] << " ";

in.close();

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

cout << "End Time for Quick Sort: " << asctime(localtime(&ctt))

<< "Time consumed for Quick Sort: " << cpu\_time\_used << " seconds." << endl << endl;

}

void \*SelectionSort(void \*arg)

{

clock\_t start, end;

double cpu\_time\_used;

sleep(2);

start = clock();

cout << "Thread ID for Selection Sort: " << pthread\_self() << endl

<< "Algorithm Name: Selection Sort" << endl

<< "Start Time for Selection Sort: " << asctime(localtime(&ctt));

fstream in;

int index = 0;

int \*arr = new int[arr\_size / 4];

in.open("Unsorted\_01.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> arr[index++];

if (in.eof())

break;

}

in.close();

Selection\_Sort(arr, arr\_size / 4);

index = 0;

in.open("selection.txt", ios::out | ios::trunc);

for (int i = 0; i < arr\_size / 4; i++)

in << arr[index++] << " ";

in.close();

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

cout << "End Time for Selection Sort: " << asctime(localtime(&ctt))

<< "Time consumed for Selection Sort: " << cpu\_time\_used << " seconds." << endl << endl;

}

void \*InsertionSort(void \*arg)

{

clock\_t start, end;

double cpu\_time\_used;

sleep(3);

start = clock();

cout << "Thread ID for Insertion Sort: " << pthread\_self() << endl

<< "Algorithm Name: Insertion Sort" << endl

<< "Start Time for Insertion Sort: " << asctime(localtime(&ctt));

fstream in;

int index = 0;

int \*arr = new int[arr\_size / 4];

in.open("Unsorted\_02.txt");

if (!in)

{

cout << "File doesn't exist";

getchar();

exit(0);

}

while (1)

{

in >> arr[index++];

if (in.eof())

break;

}

in.close();

Insertion\_Sort(arr, arr\_size / 4);

index = 0;

in.open("insertion.txt", ios::out | ios::trunc);

for (int i = 0; i < arr\_size / 4; i++)

in << arr[index++] << " ";

in.close();

end = clock();

cpu\_time\_used = ((double)(end - start)) / CLOCKS\_PER\_SEC;

cout << "End Time for Insertion Sort: " << asctime(localtime(&ctt))

<< "Time consumed for Insertion Sort: " << cpu\_time\_used << " seconds." << endl << endl;

}

void Merge(int arr[], int low, int high, int mid, int s)

{

int i, j, k, \*b;

b = new int[s];

i = low;

k = low;

j = mid + 1;

while (i <= mid && j <= high)

{

if (arr[i] < arr[j])

{

b[k] = arr[i];

k++;

i++;

}

else

{

b[k] = arr[j];

k++;

j++;

}

}

while (i <= mid)

{

b[k] = arr[i];

k++;

i++;

}

while (j <= high)

{

b[k] = arr[j];

k++;

j++;

}

for (i = low; i < k; i++)

{

arr[i] = b[i];

}

}

void Mergesort(int arr[], int low, int high, int size)

{

if (low < high) {

int mid = (low + high) / 2;

Mergesort(arr, low, mid, size);

Mergesort(arr, mid + 1, high, size);

Merge(arr, low, high, mid, size);

}

}

void Selection\_Sort(int \*arr, int n)

{

int min, temp;

for (int i = 0; i < n - 1; i++)

{

min = i;

for (int j = i + 1; j < n; j++)

if (arr[j] < arr[min])

min = j;

temp = arr[i];

arr[i] = arr[min];

arr[min] = temp;

}

}

void Insertion\_Sort(int \*arr, int n)

{

int j, temp;

for (int i = 1; i < n; i++)

{

temp = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > temp)

{

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = temp;

}

}

int partition(int arr[], int l, int u)

{

int v, i, j;

v = arr[l];

i = l;

j = u + 1;

do {

do {

i++;

} while (arr[i] < v && i <= u);

do {

j--;

} while (v < arr[j]);

if (i < j) {

int temp;

temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

} while (i < j);

arr[l] = arr[j];

arr[j] = v;

return j;

}

void quicksort(int arr[], int l, int u)

{

int j = 0;

if (l < u) {

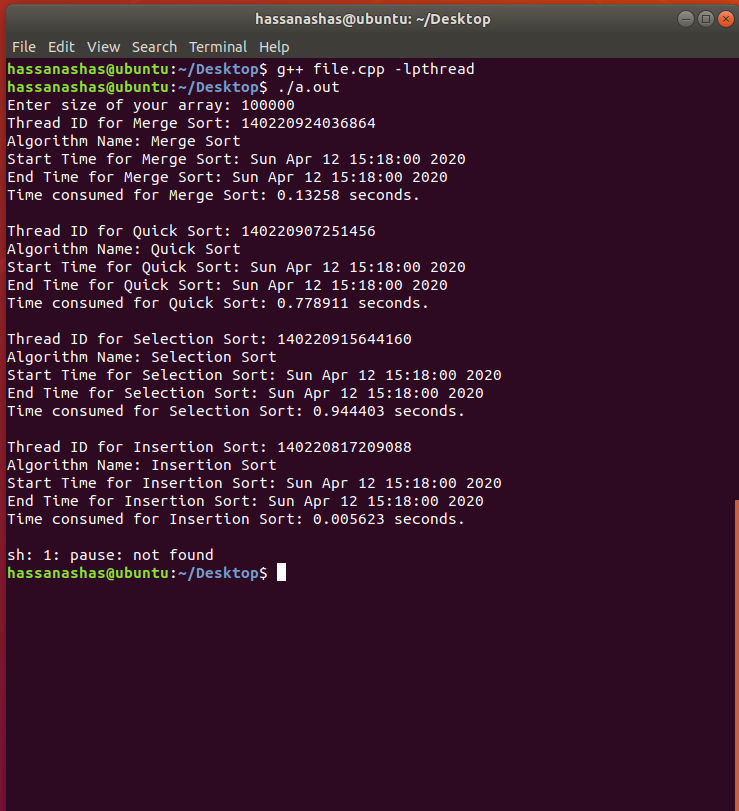
j = partition(arr, l, u);

quicksort(arr, l, j - 1);

quicksort(arr, j + 1, u);

}

}



**Assignment 4 (No Marks given)**

**Question 1**

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Round\_Robin;

struct node

{

int data;

node\* next;

};

struct Process\_Type // To store average times for all the processes.

{

string process\_name;

float avg\_turnaround\_time;

float avg\_waiting\_time;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int priority;

public:

friend class Priority\_Scheduling;

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; priority = 0; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

void set\_p(int &a) { priority = a; }

int get\_priority() { return priority; }

void Display();

void Display\_Before();

};

class Round\_Robin

{

public:

void Calculate(Queue &);

void algorithm();

};

int total\_time = 0;

int Process::count = 0;

int time\_slice, num;

Process\_Type p\_type[4];

Process \*obj = NULL;

Queue Ready\_Queue;

float avg\_turnaround\_time;

float avg\_waiting\_time;

void input();

int main()

{

input();

Round\_Robin rr;

// Round Robin Algorithm is used as a driver algorithm for the implementation of question.

rr.algorithm();

// Printing Before Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << endl << "------------------------------------------------------------------" << endl;

for (int i = 0; i < num; i++)

obj[i].Display\_Before();

// Printing After Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Turnaround" << setw(15) << "Waiting Time" << endl

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

for (int i = 1; i < num; ++i)

{

for (int j = 0; j < num - i; ++j)

if (obj[j].getid() > obj[j + 1].getid())

swap(obj[j], obj[j + 1]);

}

for (int i = 0; i < num; i++)

obj[i].Display();

cout << endl << endl;

cout << "Average Turnaround Time: " << avg\_turnaround\_time << endl

<< "Average Waiting Time: " << avg\_waiting\_time << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes: ";

cin >> num;

} while (num < 1);

obj = new Process[num];

for (int i = 0; i < num; i++)

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

// cout << "Enter Priority: "; cin >> priority;

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Round\_Robin::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!Ready\_Queue.isEmpty())

{

int temp;

temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

if (obj[temp].get\_at() > total\_time)

Ready\_Queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

//obj[temp].waiting\_time = obj[temp].turnaround\_time - obj[temp].burst\_time;

}

else

{

total\_time = total\_time + time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

Ready\_Queue.enqueue(temp);

}

}

}

for (int i = 0; i < num; i++)

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Round\_Robin::algorithm()

{

for (int i = 0; i < num; i++)

(Ready\_Queue).enqueue(i); // All the processes pushed into the Ready Queue...

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

Calculate(Ready\_Queue);

cout << endl << left << setw(30) << " " << "ROUND ROBIN ALGORITHM" << endl << endl;

for (int i = 0; i < num; i++)

{

avg\_turnaround\_time += obj[i].get\_t();

avg\_waiting\_time += obj[i].get\_w();

}

avg\_turnaround\_time = avg\_turnaround\_time / num;

avg\_waiting\_time = avg\_waiting\_time / num;

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{

front = temp;

rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

cout << "Nothing to delete";

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

**Sample Input Used:**

Enter Number of Processes: 3

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 5

Process: 1

Enter Arrival Time: 0

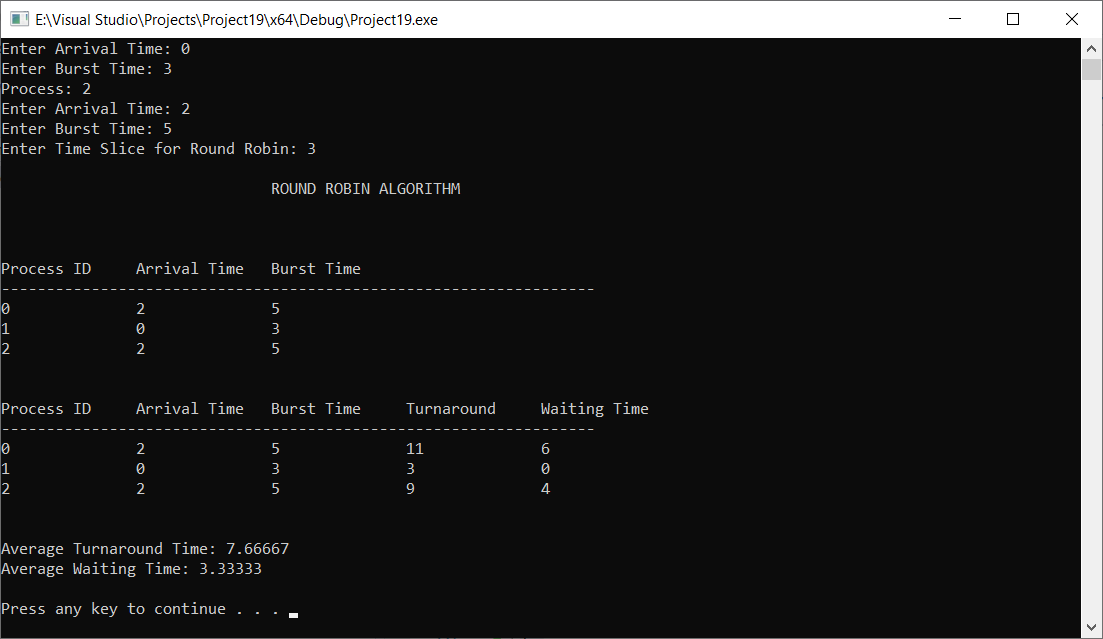
Enter Burst Time: 3

Process: 2

Enter Arrival Time: 2

Enter Burst Time: 5

Enter Time Slice for Round Robin: 3



**Question 2**

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Round\_Robin;

class FCFS;

class SJF;

class Priority\_Scheduling;

struct node

{

int data;

node\* next;

};

struct Process\_Type // To store average times for all the processes.

{

string process\_name;

float avg\_turnaround\_time;

float avg\_waiting\_time;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

int priority;

bool has\_finished; // To check whether process has finished execution or not

public:

friend class Priority\_Scheduling;

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; has\_finished = false; priority = 0; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

bool get\_finished() { return has\_finished; }

void set\_finished(bool a) { has\_finished = a; }

int get\_priority() { return priority; }

void Display();

void Display\_Before();

Process(const Process &);

};

class Round\_Robin

{

public:

void Calculate(Queue &);

void algorithm();

};

class FCFS

{

public:

void Calculate(Queue &);

void algorithm();

void sort();

};

class SJF

{

int total\_time;

public:

void Calculate(Queue &);

void algorithm();

void sort();

bool check();

};

class Priority\_Scheduling

{

int total\_time;

public:

void sort();

bool check();

void priority\_sort();

void Calculate(Queue &);

void algorithm();

};

int Process::count = 0;

int time\_slice, num;

Process\_Type p\_type[4];

Process \*obj = NULL;

Queue Ready\_Queue[4];

void input();

int main()

{

input();

Round\_Robin rr;

Priority\_Scheduling ps;

SJF sjf;

FCFS fcfs;

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Priority" << endl

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

for (int i = 0; i < num; i++)

obj[i].Display\_Before();

fcfs.algorithm();

sjf.algorithm();

for (int i = 0; i < num; i++)

obj[i].set\_finished(true);

ps.algorithm();

rr.algorithm();

int min = 0;

for (int i = 0; i < 4; i++)

{

if (p\_type[min].avg\_waiting\_time > p\_type[i].avg\_waiting\_time)

min = i;

}

cout << endl << endl;

cout << "Best Algorithm for given Input is: " << p\_type[min].process\_name << endl;

cout << "Minimum Average Waiting Time: " << p\_type[min].avg\_waiting\_time << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes: ";

cin >> num;

} while (num < 1);

obj = new Process[num];

for (int i = 0; i < num; i++)

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << priority << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

cout << "Enter Priority: "; cin >> priority;

}

// --------------------FCFS CLASS ------------------------------------

void FCFS::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_at() > obj[j + 1].get\_at())

swap(obj[j], obj[j + 1]);

}

}

void FCFS::algorithm()

{

sort();

for (int i = 0; i < num; i++)

Ready\_Queue[0].enqueue(i); // Inserting Process IDs into the Queue for FCFS

Calculate(Ready\_Queue[0]);

for (int i = 0; i < num; i++)

{

p\_type[0].avg\_turnaround\_time += obj[i].get\_t();

p\_type[0].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "FIRST COME FIRST SERVE ALGORITHM" << endl << endl;

p\_type[0].process\_name = "FIRST COME FIRST SERVE ALGORITHM";

p\_type[0].avg\_turnaround\_time = p\_type[0].avg\_turnaround\_time / num;

p\_type[0].avg\_waiting\_time = p\_type[0].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[0].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[0].avg\_waiting\_time << endl;

}

void FCFS::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!(Ready\_Queue.isEmpty()))

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj->set\_w(b);

}

}

// --------------------SJF CLASS ------------------------------------

void SJF::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_sbt() > obj[j + 1].get\_sbt())

swap(obj[j], obj[j + 1]);

}

}

bool SJF::check()

{

for (int i = 0; i < num; i++)

if (obj[i].get\_finished() == false)

return false;

return true;

}

void SJF::algorithm()

{

total\_time = 0;

sort();

while (!check())

{

for (int i = 0; i < num; i++)

{

if (obj[i].get\_at() <= total\_time) // Initially, only those processes are inserted whose arrival time is zero.

if (obj[i].get\_finished() == false)

Ready\_Queue[1].enqueue(i); // Inserting Process IDs into the Queue for SJF

}

if (!(Ready\_Queue[1].isEmpty()))

Calculate(Ready\_Queue[1]);

}

for (int i = 0; i < num; i++)

{

p\_type[1].avg\_turnaround\_time += obj[i].get\_t();

p\_type[1].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "SHORT JOB FIRST ALGORITHM" << endl << endl;

p\_type[1].process\_name = "SHORT JOB FIRST ALGORITHM";

p\_type[1].avg\_turnaround\_time = p\_type[1].avg\_turnaround\_time / num;

p\_type[1].avg\_waiting\_time = p\_type[1].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[1].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[1].avg\_waiting\_time << endl;

}

void SJF::Calculate(Queue &Ready\_Queue)

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

obj[temp].set\_finished(true);

while (!(Ready\_Queue.isEmpty()))

Ready\_Queue.dequeue();

}

// --------------------Priority CLASS ------------------------------------

void Priority\_Scheduling::sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - i; ++j)

if (obj[j].get\_at() > obj[j + 1].get\_at())

swap(obj[j], obj[j + 1]);

}

}

void Priority\_Scheduling::priority\_sort()

{

int j;

for (int i = 1; i < num; ++i)

{

for (j = 0; j < num - 1; ++j)

{

if (obj[j].priority > obj[j + 1].priority)

swap(obj[j], obj[j + 1]);

}

}

}

bool Priority\_Scheduling::check()

{

for (int i = 0; i < num; i++)

if (obj[i].get\_finished() == false)

return false;

return true;

}

void Priority\_Scheduling::algorithm()

{

total\_time = 0;

cout << endl;

sort();

priority\_sort();

while (!check())

{

for (int i = 0; i < num; i++)

{

if (obj[i].get\_at() <= total\_time) // Initially, only those processes are inserted whose arrival time is zero.

if (obj[i].get\_finished() == false)

Ready\_Queue[2].enqueue(i); // Inserting Process IDs into the Queue for Priority\_Scheduling

}

if (!(Ready\_Queue[2].isEmpty()))

Calculate(Ready\_Queue[2]);

}

for (int i = 0; i < num; i++)

{

p\_type[2].avg\_turnaround\_time += obj[i].get\_t();

p\_type[2].avg\_waiting\_time += obj[i].get\_w();

}

p\_type[2].process\_name = "PRIORITY SCHEDULING ALGORITHM";

cout << endl << left << setw(30) << " " << "PRIORITY SCHEDULING ALGORITHM" << endl << endl;

p\_type[2].avg\_turnaround\_time = p\_type[2].avg\_turnaround\_time / num;

p\_type[2].avg\_waiting\_time = p\_type[2].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[2].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[2].avg\_waiting\_time << endl;

}

void Priority\_Scheduling::Calculate(Queue &Ready\_Queue)

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

obj[temp].set\_finished(true);

while (!(Ready\_Queue.isEmpty()))

Ready\_Queue.dequeue();

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Round\_Robin::Calculate(Queue &Ready\_Queue)

{

int total\_time = 0;

while (!Ready\_Queue.isEmpty())

{

int temp;

temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

if (obj[temp].get\_at() > total\_time)

Ready\_Queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

//obj[temp].waiting\_time = obj[temp].turnaround\_time - obj[temp].burst\_time;

}

else

{

total\_time = total\_time + time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

Ready\_Queue.enqueue(temp);

}

}

}

for (int i = 0; i < num; i++)

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Round\_Robin::algorithm()

{

for (int i = 0; i < num; i++)

(Ready\_Queue[3]).enqueue(i); // All the processes pushed into the Ready Queue...

Calculate(Ready\_Queue[3]);

for (int i = 0; i < num; i++)

{

p\_type[3].avg\_turnaround\_time += obj[i].get\_t();

p\_type[3].avg\_waiting\_time += obj[i].get\_w();

}

cout << endl << left << setw(30) << " " << "ROUND ROBIN ALGORITHM" << endl << endl;

p\_type[3].process\_name = "ROUND ROBIN ALGORITHM";

p\_type[3].avg\_turnaround\_time = p\_type[3].avg\_turnaround\_time / num;

p\_type[3].avg\_waiting\_time = p\_type[3].avg\_waiting\_time / num;

cout << "Average Turnaround Time: " << p\_type[3].avg\_turnaround\_time << endl

<< "Average Waiting Time: " << p\_type[3].avg\_waiting\_time << endl;

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{

front = temp;

rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

cout << "Nothing to delete";

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

Process::Process(const Process &Process\_obj)

{

this->arrival\_time = Process\_obj.arrival\_time;

this->burst\_time = Process\_obj.burst\_time;

this->completion\_time = Process\_obj.completion\_time;

this->id = Process\_obj.id;

this->stored\_burst\_time = Process\_obj.stored\_burst\_time;

this->turnaround\_time = Process\_obj.turnaround\_time;

this->waiting\_time = Process\_obj.waiting\_time;

this->has\_finished = Process\_obj.has\_finished;

}

**SAMPLE INPUT**

Enter Number of Processes: 4

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 5

Enter Priority: 5

Process: 1

Enter Arrival Time: 0

Enter Burst Time: 3

Enter Priority: 4

Process: 2

Enter Arrival Time: 3

Enter Burst Time: 5

Enter Priority: 2

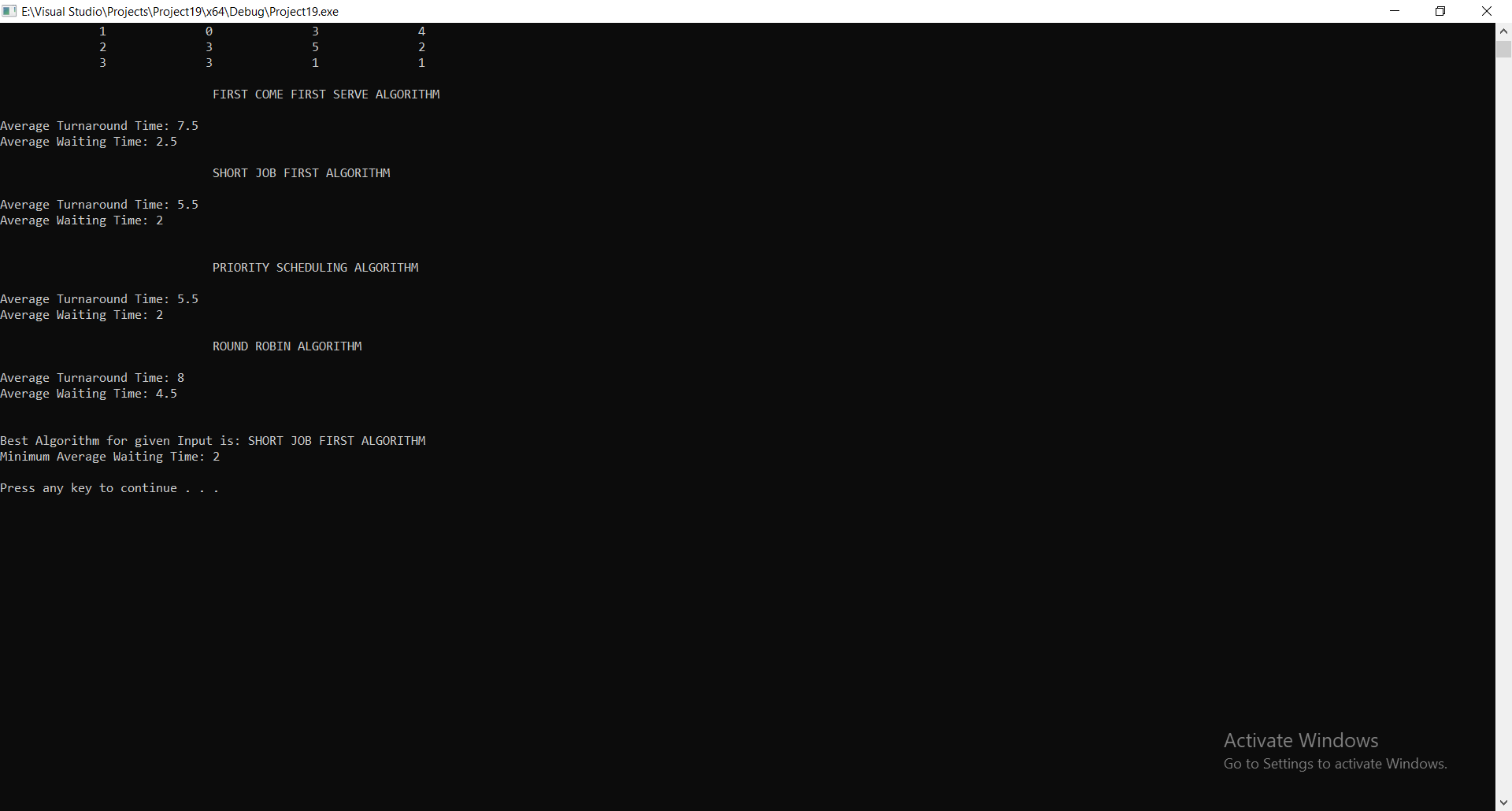
Process: 3

Enter Arrival Time: 3

Enter Burst Time: 1

Enter Priority: 1

Enter Time Slice for Round Robin: 2



**Question 3**

#include <iostream>

#include <string>

#include <iomanip>

using namespace std;

class Process;

class Implementation\_Class;

class FCFS;

struct node

{

int data;

node\* next;

};

class Queue

{

node\* front;

node\* rear;

public:

Queue();

bool isEmpty();

void dequeue();

void enqueue(int &);

int peek();

};

class Process

{

int id;

static int count; // Count will maintain the record of total processes created.

int burst\_time;

int stored\_burst\_time; // To keep the original value of burst time intact.

int arrival\_time;

int completion\_time;

int turnaround\_time;

int waiting\_time;

public:

Process() :id(count) { completion\_time = turnaround\_time = waiting\_time = 0; count++; }

// Every process will have a unique Process ID as assigned by count...

void input();

// Set and Get Functions.

int get\_t() { return turnaround\_time; }

int get\_w() { return waiting\_time; }

int getid() { return id; }

int get\_bt() { return burst\_time; }

int get\_ct() { return completion\_time; }

int get\_at() { return arrival\_time; }

int get\_sbt() { return stored\_burst\_time; }

void set\_bt(int &a) { burst\_time = a; }

void set\_at(int &a) { arrival\_time = a; }

void set\_ct(int &a) { completion\_time = a; }

void set\_t(int &a) { turnaround\_time = a; }

void set\_w(int &a) { waiting\_time = a; }

void Display();

void Display\_Before();

};

class Implementation\_Class

{

int total\_time = 0;

public:

void Calculate\_1st(Queue &);

void Calculate\_2nd(Queue &);

void FCFS\_Algorithm(Queue &);

void algorithm();

};

int Process::count = 0;

int time\_slice, num, process\_time;

int num1, num2, num3;

Process \*obj = NULL;

Queue Ready\_Queue[3];

void input();

int main()

{

input();

cout << "Enter Time Slice for Round Robin: "; cin >> time\_slice;

Implementation\_Class r1;

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Priority" << endl

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

for (int i = 0; i < num; i++)

obj[i].Display\_Before();

// Round Robin Algorithm is used as a driver algorithm for the implementation of question.

// Printing Before Calculation

// Printing After Calculation

cout << endl << endl << setw(15) << "Process ID" << setw(15) << "Arrival Time" << setw(15) << "Burst Time" << setw(15) << "Turnaround" << setw(15) << "Waiting Time" << endl

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

for (int i = 1; i < num; ++i)

{

for (int j = 0; j < num - i; ++j)

if (obj[j].getid() > obj[j + 1].getid())

swap(obj[j], obj[j + 1]);

}

for (int i = 0; i < num; i++)

obj[i].Display();

cout << endl << endl;

cout << endl;

system("pause");

return 0;

}

void input()

{

do

{

cout << "Enter Number of Processes for 1st Queue (Priority 3): ";

cin >> num1;

} while (num1 < 1);

num = 0;

num += num1;

do

{

cout << "Enter Number of Processes for 2nd Queue (Priority 2): ";

cin >> num2;

} while (num2 < 1);

num += num2;

do

{

cout << "Enter Number of Processes for 3rd Queue (Priority 1): ";

cin >> num3;

} while (num3 < 1);

num += num3;

obj = new Process[num];

cout << "\nEnter Processes for Queue 1 (Priority 3), \n";

int i = 0;

for (i = 0; i < num1; i++)

obj[i].input();

cout << "\nEnter Processes for Queue 2 (Priority 2), \n";

for (i; i < num1 + num2; i++)

obj[i].input();

cout << "\nEnter Processes for Queue 3 (Priority 1), \n";

for (i; i < num; i++)

obj[i].input();

}

void Process::Display()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << setw(15) << turnaround\_time << setw(15) << waiting\_time;

cout << endl;

}

void Process::Display\_Before()

{

cout << setw(15) << id << setw(15) << arrival\_time << setw(15) << stored\_burst\_time << endl;

}

void Process::input()

{

cout << "Process: " << id << endl

<< "Enter Arrival Time: "; cin >> arrival\_time;

cout << "Enter Burst Time: "; cin >> burst\_time;

stored\_burst\_time = burst\_time;

}

// --------------------ROUND ROBIN CLASS ------------------------------------

void Implementation\_Class::Calculate\_1st(Queue &first\_queue)

{

int time\_interval = 0;

while (!first\_queue.isEmpty())

{

int temp;

temp = first\_queue.peek();

first\_queue.dequeue();

if (obj[temp].get\_at() > total\_time)

first\_queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

process\_time += obj[temp].get\_bt();

time\_interval += obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

}

else

{

total\_time = total\_time + time\_slice;

process\_time += time\_slice;

time\_interval += time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

first\_queue.enqueue(temp);

}

}

if (time\_interval >= 10) // If 10 seconds have passed, push the process from 3rd to 2nd queue

{

if (!Ready\_Queue[2].isEmpty())

{

int temp1 = Ready\_Queue[2].peek();

Ready\_Queue[1].enqueue(temp1);

Ready\_Queue[2].dequeue();

time\_interval -= 10;

}

}

if (process\_time > 3 \* time\_slice) // if allocated time is passed then return from function..

return;

}

for (int i = 0; i < num; i++)

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Implementation\_Class::Calculate\_2nd(Queue &second\_queue)

{

int time\_interval = 0;

while (!second\_queue.isEmpty())

{

int temp;

temp = second\_queue.peek();

second\_queue.dequeue();

if (obj[temp].get\_at() > total\_time)

second\_queue.enqueue(temp);

else

{

if (obj[temp].get\_bt() <= time\_slice)

{

total\_time = total\_time + obj[temp].get\_bt();

process\_time += obj[temp].get\_bt();

time\_interval += obj[temp].get\_bt();

obj[temp].set\_ct(total\_time);

int b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

}

else

{

total\_time = total\_time + time\_slice;

process\_time += time\_slice;

time\_interval += time\_slice;

int b = obj[temp].get\_bt() - time\_slice;

obj[temp].set\_bt(b);

second\_queue.enqueue(temp);

}

}

}

if (time\_interval > 10) // If 10 seconds have passed, push the process from 3rd to 2nd queue

{

if (Ready\_Queue[2].isEmpty())

{

int temp1 = Ready\_Queue[2].peek();

Ready\_Queue[1].enqueue(temp1);

Ready\_Queue[2].dequeue();

}

}

if (process\_time > time\_slice) // if allocated time is passed then return from function..

return;

for (int i = 0; i < num; i++)

{

int b = (obj[i].get\_t() - obj[i].get\_sbt());

obj[i].set\_w(b);

}

}

void Implementation\_Class::FCFS\_Algorithm(Queue &Ready\_Queue)

{

while (!(Ready\_Queue.isEmpty()))

{

int temp = Ready\_Queue.peek();

Ready\_Queue.dequeue();

int b = total\_time + obj[temp].get\_sbt();

obj[temp].set\_ct(b);

total\_time += obj[temp].get\_sbt();

b = obj[temp].get\_ct() - obj[temp].get\_at();

obj[temp].set\_t(b);

b = obj[temp].get\_t() - obj[temp].get\_sbt();

obj[temp].set\_w(b);

}

}

void Implementation\_Class::algorithm()

{

total\_time = 0;

int i = 0;

for (i = 0; i < num1; i++)

(Ready\_Queue[0]).enqueue(i);

for (i; i < num1 + num2; i++)

(Ready\_Queue[1]).enqueue(i);

for (i; i < num; i++)

(Ready\_Queue[2]).enqueue(i);

while ((!Ready\_Queue[0].isEmpty()) || (!Ready\_Queue[1].isEmpty()) || (!Ready\_Queue[2].isEmpty()))

{

process\_time = 0;

Calculate\_1st(Ready\_Queue[0]);

process\_time = 0;

Calculate\_2nd(Ready\_Queue[1]);

if ((Ready\_Queue[0].isEmpty()) && (Ready\_Queue[1].isEmpty()))

break;

}

if (!Ready\_Queue[2].isEmpty())

FCFS\_Algorithm(Ready\_Queue[2]);

}

// ------------------- QUEUE CLASS IMPLEMENTATION ---------------------------

Queue::Queue() { front = NULL; rear = NULL; }

bool Queue::isEmpty()

{

if (!front)

return true;

return false;

}

int Queue::peek() { return front->data; }

void Queue::enqueue(int &d)

{

node\* temp = new node;

temp->data = d;

temp->next = NULL;

if (isEmpty())

{ rear = temp;

return;

}

rear->next = temp;

rear = temp;

}

void Queue::dequeue()

{

if (isEmpty())

{

return;

}

node\* temp = front;

front = front->next;

free(temp);

}

**SAMPLE INPUT:**

front = temp;

Enter Number of Processes for 1st Queue (Priority 3): 3

Enter Number of Processes for 2nd Queue (Priority 2): 3

Enter Number of Processes for 3rd Queue (Priority 1): 2

Enter Processes for Queue 1 (Priority 3),

Process: 0

Enter Arrival Time: 2

Enter Burst Time: 3

Process: 1

Enter Arrival Time: 0

Enter Burst Time: 5

Process: 2

Enter Arrival Time: 0

Enter Burst Time: 6

Enter Processes for Queue 2 (Priority 2),

Process: 3

Enter Arrival Time: 0

Enter Burst Time: 3

Process: 4

Enter Arrival Time: 5

Enter Burst Time: 3

Process: 5

Enter Arrival Time: 5

Enter Burst Time: 2

Enter Processes for Queue 3 (Priority 1),

Process: 6

Enter Arrival Time: 2

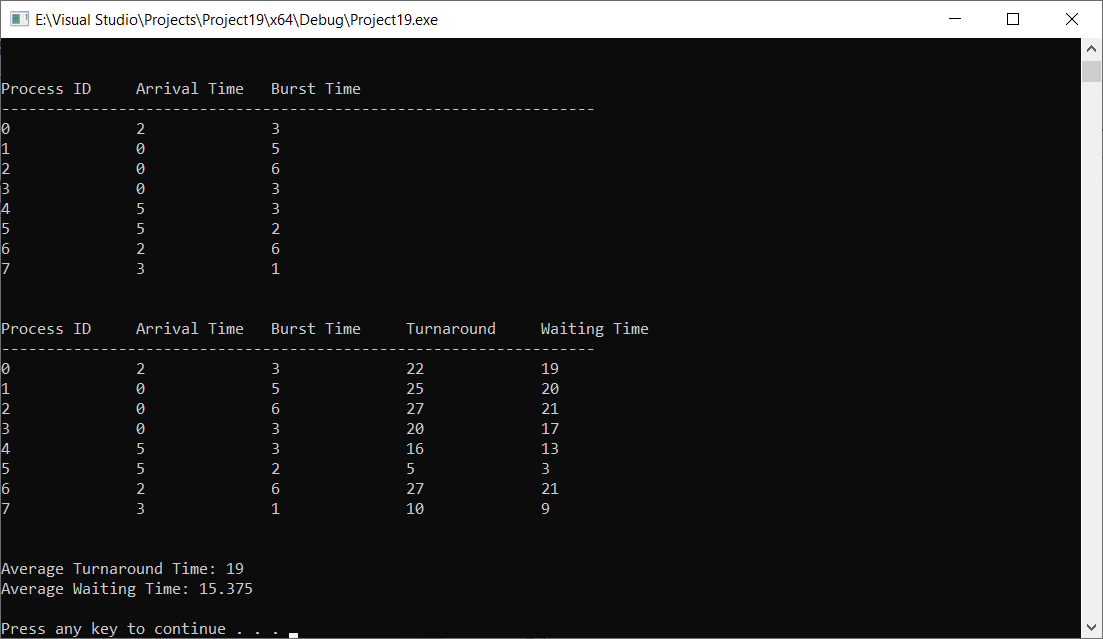
Enter Burst Time: 6

Process: 7

Enter Arrival Time: 3

Enter Burst Time: 1

Enter Time Slice for Round Robin: 2



**Assignment 5 (2/3 Marks)**

**Question 1**

**Mutex**

* Mutex is a mutual exclusion lock. If there is a shared resource between different threads, mutex enforces exclusive access to that variable to only one thread at one time, hence protecting the **critical section** of the memory.
* Mutex doesn’t work between processes and only is applied for threads.
* Only one thread at a time can take the ownership of the mutex. Once it has the ownership, no other thread can access the shared resources or the critical section until that mutex is done with its job. Once that mutex has performed its job, it releases the mutex lock, making it possible for others to take ownership of the mutex.

**Working of the Mutex**

* To begin with, there is an integer in memory that provides the locked state. It can either be one or zero.
* Initially, the locked state is set to zero. As soon as one thread attempts to gain the access of a critical section, it gains the ownership of the mutex and locked state of mutex is turned to one.
* If now a content switch is performed by the scheduler, all other threads that are ready to execute the critical section are awaken. However, since the region is already locked by mutex (because of another thread having the ownership of mutex), all the other threads are denied permission to enter critical section and are put to sleep again.
* Content switch takes place again but again as long as the mutex is locked, no thread is allowed execution.
* As soon as the first thread finishes executing the critical section, it unlocks the mutex, turning the locked state to zero from one. Now, content switch occurs and the first thread to request permission for critical section is given entry into the critical section, with others again placed into sleep due to locked state turning into one.

**Example**

* Question number 5 is a very good example of Mutex.
* Different threads at the same time try to gain access to the shared resource of “bank balance” integer, however only one is given access to it at one time. Once one thread has entered the critical section, locked state turns to one and other threads are put to sleep.
* Now, once the first thread has finished depositing its value into the bank balance, it unlocks the mutex, turning the locked state value to 0.
* Content switch occurs and other threads are awakened. The first thread to ask permission to critical section then gains the ownership of the mutex and the remaining are put to sleep.
* This process repeats until all the threads have finished their execution.
* Implementation of the code is also given below,

**Implementation**

#include <iostream>

#include <pthread.h>

#include <string>

#include <unistd.h>

using namespace std;

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

int bank\_balance = 100; // initially bank balance is set to 100

void \*deposit\_amount(void \*arg) {

int deposited\_amount = \*(int \*)arg;

pthread\_mutex\_lock(&m);

int old\_balance = bank\_balance;

int new\_balance = old\_balance + deposited\_amount;

bank\_balance = new\_balance;

pthread\_mutex\_unlock(&m);

pthread\_exit(NULL);

}

int main() {

pthread\_t brother, dad, cousin;

int bro, d, co;

cout << "Enter amount deposited by Dad: ";

cin >> d;

cout << "Enter amount deposited by Brother: ";

cin >> bro;

cout << "Enter amount deposited by Cousin: ";

cin >> co;

pthread\_create(&dad, NULL, deposit\_amount, &d);

pthread\_create(&brother, NULL, deposit\_amount, &bro);

pthread\_create(&cousin, NULL, deposit\_amount, &co);

pthread\_join(dad, NULL);

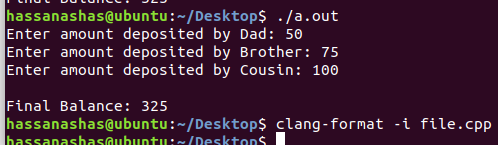
pthread\_join(brother, NULL);

pthread\_join(cousin, NULL);

cout << endl << "Final Balance: " << bank\_balance << endl;

return 0;

}



**Question 2**

#include <iostream>

#include <pthread.h>

#include <semaphore.h>

#include <string>

#include <unistd.h>

using namespace std;

string message;

sem\_t s;

void \*send\_message(void \*arg) {

cout << "Enter Message to Send: ";

getline(cin, message);

cout << "Message Sent to Receiver: " << message << endl;

sem\_post(&s);

pthread\_exit(NULL);

}

void \*receive\_message(void \*arg) {

sem\_wait(&s);

cout << "Message Recieved from Sender: " << message << endl;

pthread\_exit(NULL);

}

int main() {

pthread\_t sender, receiver;

sem\_init(&s, 0, 0);

pthread\_create(&sender, NULL, send\_message, NULL);

pthread\_create(&receiver, NULL, receive\_message, NULL);

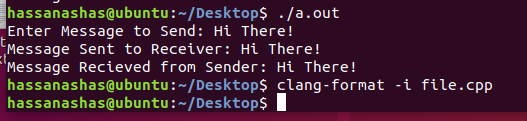
pthread\_join(sender, NULL);

pthread\_join(receiver, NULL);

sem\_destroy(&s);

return 0;

}



**Question 3**

#include <iostream>

#include <pthread.h>

#include <string>

#include <unistd.h>

using namespace std;

string message;

pthread\_cond\_t c = PTHREAD\_COND\_INITIALIZER;

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

int done = 0;

void \*send\_message(void \*arg) {

pthread\_mutex\_lock(&m);

cout << "Enter Message to Send: ";

getline(cin, message);

cout << "Message Sent to Receiver: " << message << endl;

done = 1;

pthread\_cond\_signal(&c);

pthread\_mutex\_unlock(&m);

pthread\_exit(NULL);

}

void \*receive\_message(void \*arg) {

pthread\_mutex\_lock(&m);

while (done == 0)

pthread\_cond\_wait(&c, &m);

cout << "Message Recieved from Sender: " << message << endl;

pthread\_mutex\_unlock(&m);

pthread\_exit(NULL);

}

int main() {

pthread\_t sender, receiver;

pthread\_create(&sender, NULL, send\_message, NULL);

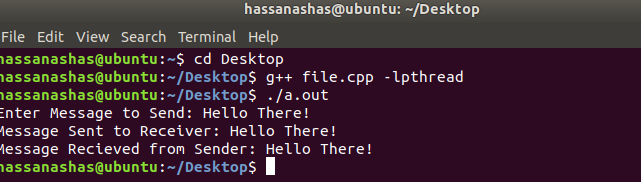
pthread\_create(&receiver, NULL, receive\_message, NULL);

pthread\_join(sender, NULL);

pthread\_join(receiver, NULL);

return 0;

}



**Question 4**

#include <iostream>

#include <pthread.h>

using namespace std;

int var;

int result;

void \*fun(void \*arg)

{

pthread\_detach(pthread\_self());

var++;

result = result + var;

pthread\_exit(NULL);

return NULL;

}

int main()

{

var = 0;

result = 0;

pthread\_t p1;

pthread\_create (&p1, NULL, &fun, NULL);

pthread\_join(p1, NULL);

pthread\_create (&p1, NULL, &fun, NULL);

pthread\_join(p1, NULL);

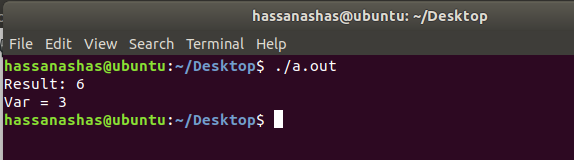
pthread\_create (&p1, NULL, &fun, NULL);

pthread\_join(p1, NULL);

cout << "Result: " << result << endl << "Var = " << var << endl;

return 0;

}



**Question 5**

#include <iostream>

#include <pthread.h>

#include <string>

#include <unistd.h>

using namespace std;

pthread\_mutex\_t m = PTHREAD\_MUTEX\_INITIALIZER;

int bank\_balance = 100; // initially bank balance is set to 100

void \*deposit\_amount(void \*arg) {

int deposited\_amount = \*(int \*)arg;

pthread\_mutex\_lock(&m);

int old\_balance = bank\_balance;

int new\_balance = old\_balance + deposited\_amount;

bank\_balance = new\_balance;

pthread\_mutex\_unlock(&m);

pthread\_exit(NULL);

}

int main() {

pthread\_t brother, dad, cousin;

int bro, d, co;

cout << "Enter amount deposited by Dad: ";

cin >> d;

cout << "Enter amount deposited by Brother: ";

cin >> bro;

cout << "Enter amount deposited by Cousin: ";

cin >> co;

pthread\_create(&dad, NULL, deposit\_amount, &d);

pthread\_create(&brother, NULL, deposit\_amount, &bro);

pthread\_create(&cousin, NULL, deposit\_amount, &co);

pthread\_join(dad, NULL);

pthread\_join(brother, NULL);

pthread\_join(cousin, NULL);

cout << endl << "Final Balance: " << bank\_balance << endl;

return 0;

}

