**Question 1**

#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;

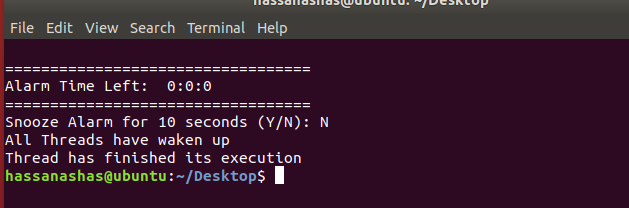
pthread\_create(&t1, NULL, Thread\_Fun, NULL);

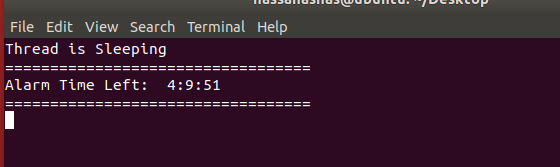
obj.Timer();

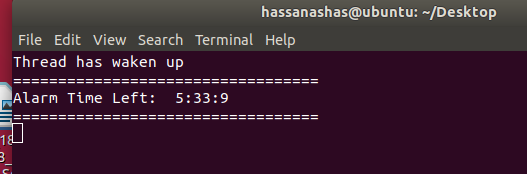
pthread\_join(t1, NULL);

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())) {

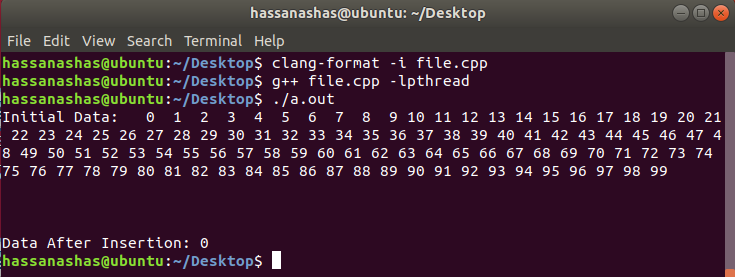
this->enqueue(q.peek());

q.dequeue();

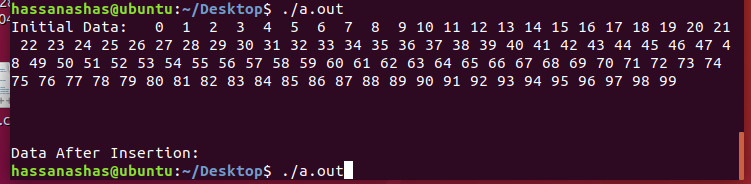
}

}

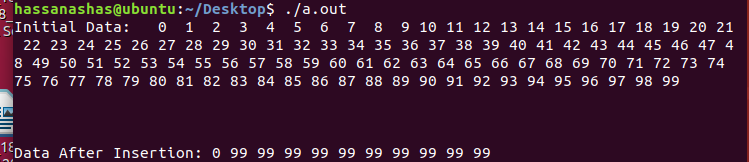
* Initially, the text file contains all the numbers from 0 to 99 (i.e. 100 numbers in total)
* Output of program on first try,



* 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);

}

}

