OPERATING SYSTEMS PRACTICE (COM301P)

Name: Vinayak Sethi **Roll No:** COE18B061

Assignment 6

Using Multithreading

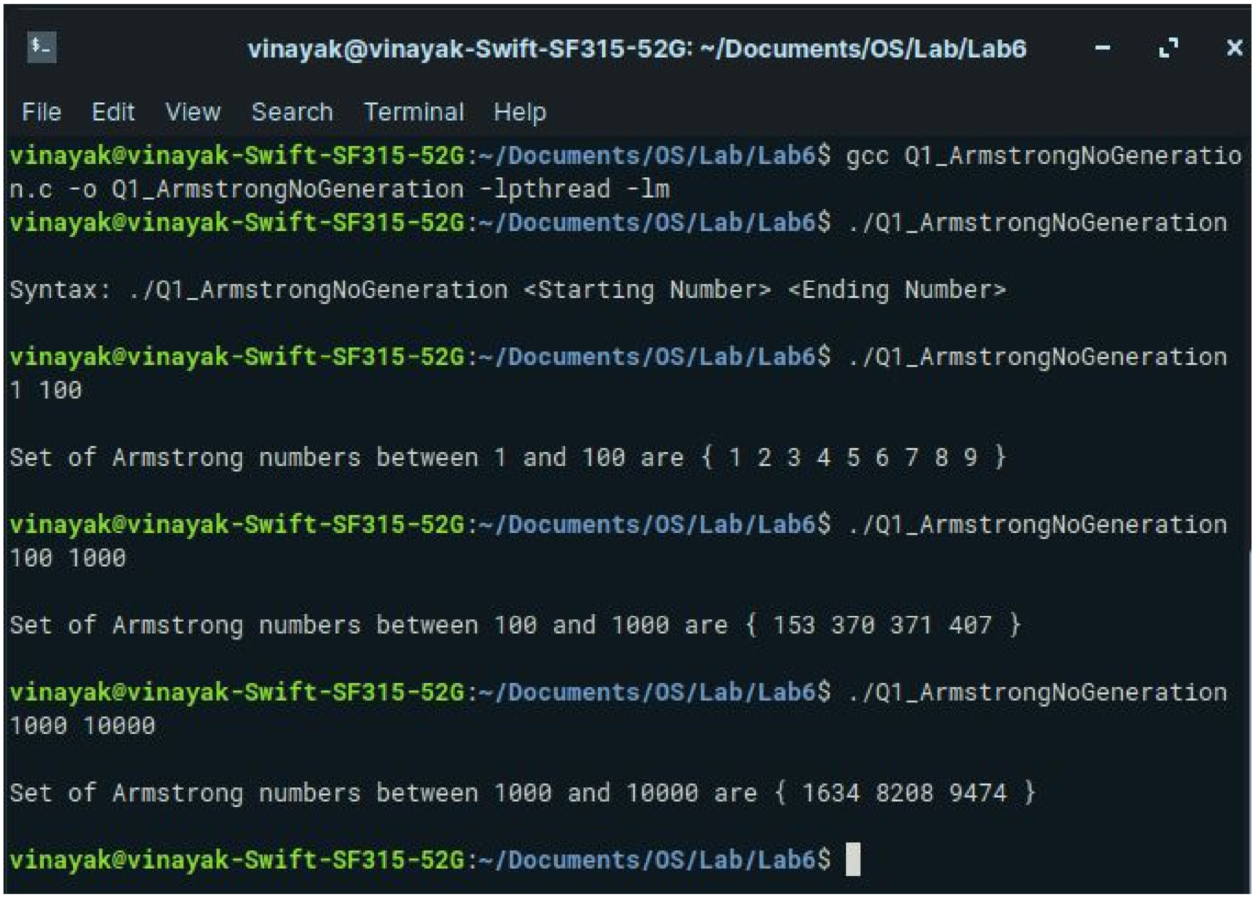
(1) Armstrong number generation within a range.

Filename: Q1_ArmstrongNoGeneration.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#include<math.h>

#define MAX 10000

int start, end;
int store_sum[MAX];

void *runner(void *num);

int main(int argc, char *argv[])
{
   //Validate the usage of command line arguments
   if(argc < 3)
   {
       printf("\nSyntax: %s <Starting Number> <Ending Number>\n\n",
argv[0]);
       exit(EXIT_FAILURE);
   }

   start = atoi(argv[1]);
```

```
    end = atoi(argv[2]);

    printf("\nSet of Armstrong numbers between %d and %d are { ", start, end);

    for(int i=start; i<=end; i++)
    {
        int *val = (int *)malloc(sizeof(int));
        *val = i;

        pthread_t tid; //returns the thread id of thread created
        pthread_attr_t attr; //to define thread attributes
        pthread_attr_init(&attr); //initializes the thread attributes

        pthread_create(&tid, &attr, runner, val); //creates a new thread
        pthread_join(tid, NULL); //wait for termination of the thread
    }

    for(int i=0; i<=end-start; i++)
        if(store_sum[i] == i+start)
            printf("%d ", store_sum[i]);

    printf("\b }\n\n");
    return 0;
}

void *runner(void *num)
{
    int *val = (int *)num;
    int temp = *val;
    int i = 0;
    int sum = 0;

    while(temp != 0)
    {
        i++; // counts the number of digit in the number
        temp = temp/10;
    }
    temp = *val;
    while(temp != 0)
    {
        sum = sum + pow(temp%10, i);
        temp = temp/10;
    }

    store_sum[*val - start] = sum; //stores the value of sum

    pthread_exit(0); //to exit a thread
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q1_ArmstrongNoGeneratio
n.c -o Q1_ArmstrongNoGeneration -lpthread -lm
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q1_ArmstrongNoGeneration

Syntax: ./Q1_ArmstrongNoGeneration <Starting Number> <Ending Number>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q1_ArmstrongNoGeneration
1 100

Set of Armstrong numbers between 1 and 100 are { 1 2 3 4 5 6 7 8 9 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q1_ArmstrongNoGeneration
100 1000

Set of Armstrong numbers between 100 and 1000 are { 153 370 371 407 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q1_ArmstrongNoGeneration
1000 10000

Set of Armstrong numbers between 1000 and 10000 are { 1634 8208 9474 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

Explanation:

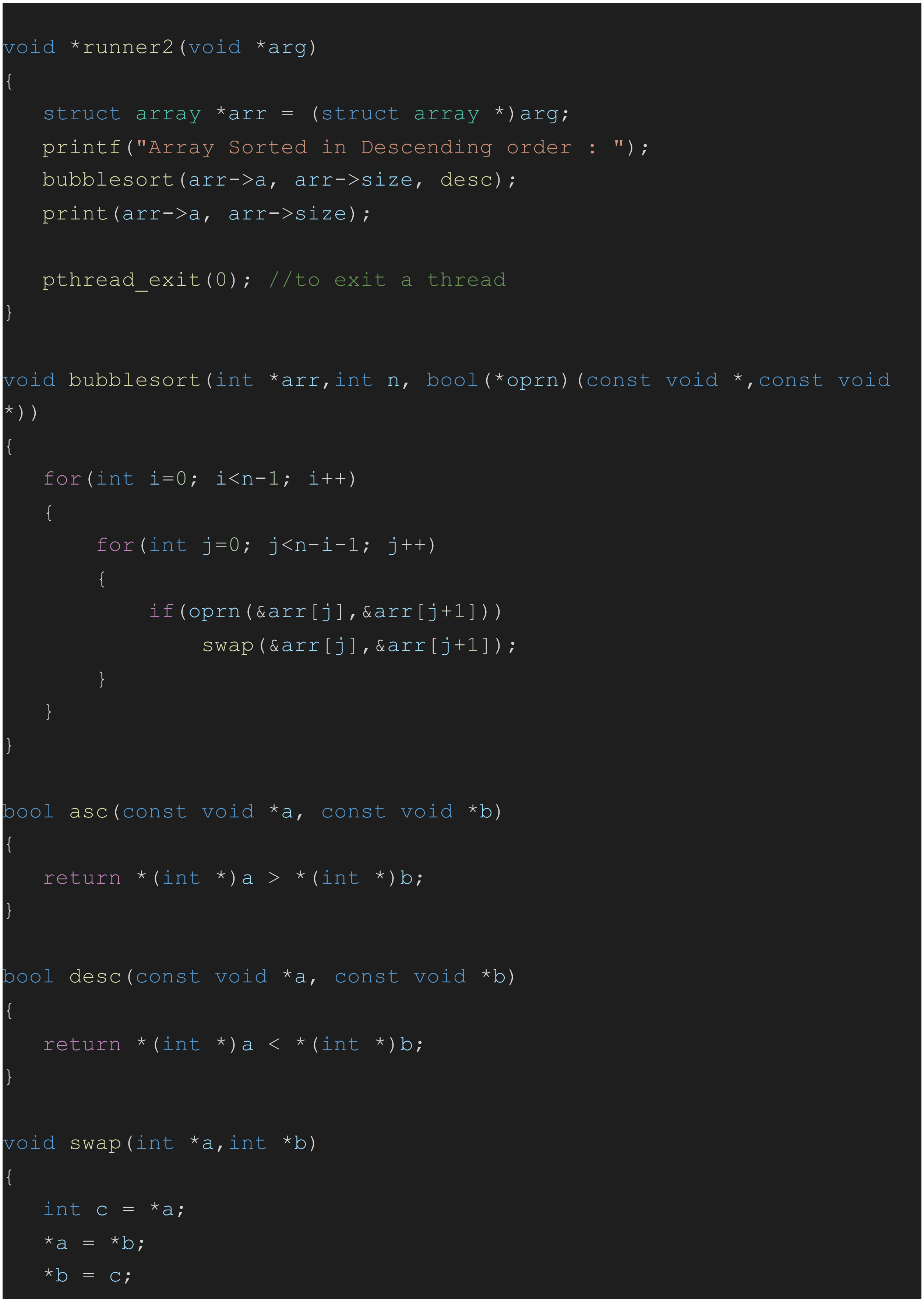
For each number lying in the range of <Starting Number> and <Ending Number> we create a thread. The thread stores the sum of the digits in the given number raised to the power of the number of digits in that number. We compare this sum with the original number inside the main() function. A match means the number is an Armstrong number.

(2) Ascending Order sort and Descending order sort.

Filename: Q2_Sort.c

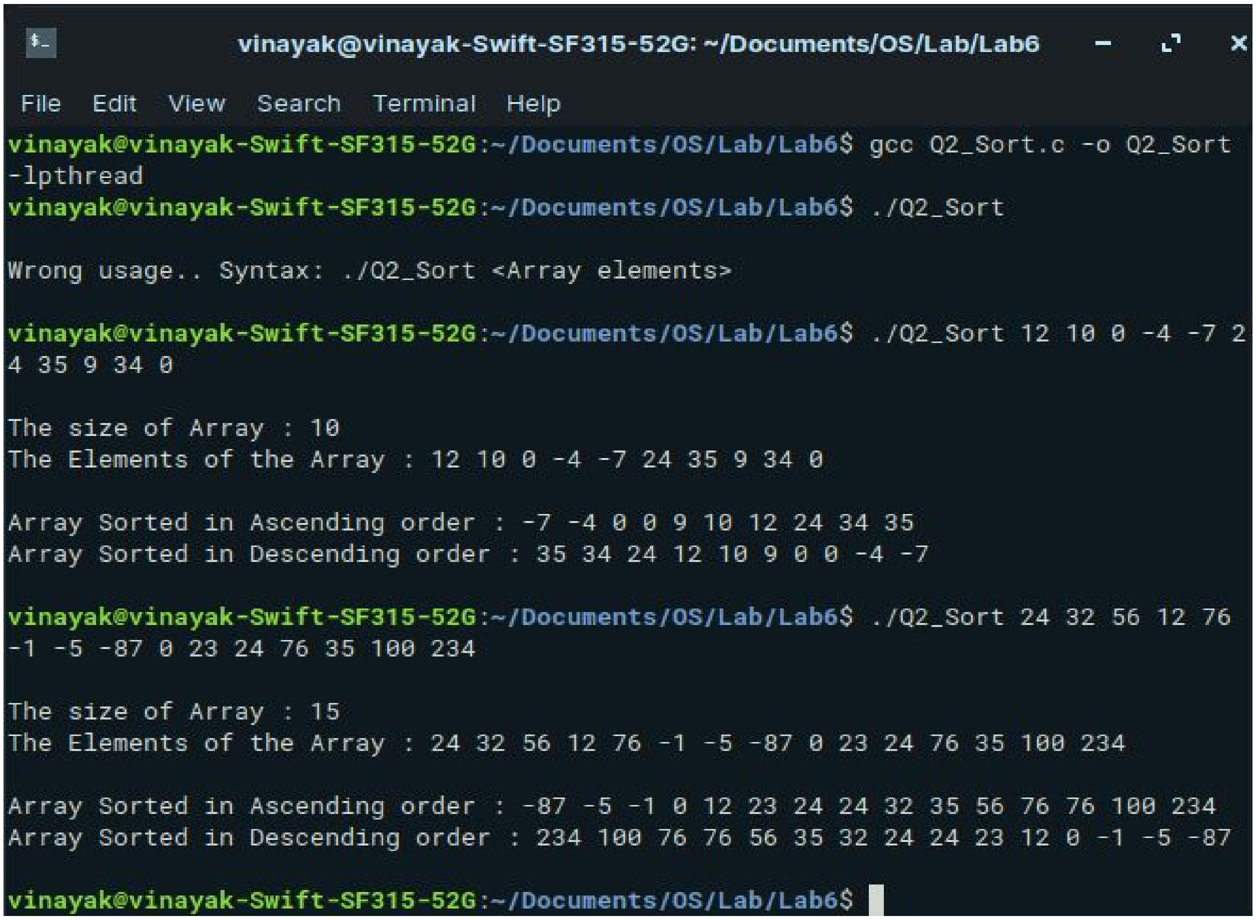
```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>
#include<pthread.h>

struct array
{
   int a[100];
   int size;
};

void *runner1(void *arg); //for ascending order sort
void *runner2(void *arg); //for descending order sort
void bubblesort(int *arr, int n, bool(*oprn)(const void *,const void
*)); //using function pointer
bool asc(const void *a, const void *b);
bool desc(const void *a, const void *b);
void swap(int *a,int *b);
void print(int *arr,int n);

int main(int argc, char *argv[])
{
   //Validate correct number of arguments
   if(argc < 2)
   {
       printf("\nWrong usage.. Syntax: %s <Array elements> \n\n",
argv[0]);
```

```c
        exit(EXIT_FAILURE);
    }

    struct array *arr = (struct array *)malloc(sizeof(struct array));
    arr->size = argc - 1;

    printf("\nThe size of Array : %d", arr->size);
    printf("\nThe Elements of the Array : ");
    for(int i=1; i<argc; i++)
    {
        arr->a[i-1] = atoi(argv[i]);
    }
    print(arr->a, arr->size);

    pthread_t tid1, tid2; //returns the thread id of thread created
    pthread_attr_t attr; //to define thread attributes
    pthread_attr_init(&attr); //initializes the thread attributes

    pthread_create(&tid1, &attr, runner1, arr); //creates a new thread for ascending sort
    pthread_create(&tid2, &attr, runner2, arr); //creates a new thread for descending sort

    pthread_join(tid1, NULL); //wait for termination of the thread1
    pthread_join(tid2, NULL); //wait for termination of the thread2

    printf("\n");
    return 0;
}

void *runner1(void *arg)
{
    struct array *arr = (struct array *)arg;
    printf("\nArray Sorted in Ascending order : ");
    bubblesort(arr->a, arr->size,asc);
    print(arr->a, arr->size);

    pthread_exit(0); //to exit a thread
}
```



```
void *runner2(void *arg)
{
   struct array *arr = (struct array *)arg;
   printf("Array Sorted in Descending order : ");
   bubblesort(arr->a, arr->size, desc);
   print(arr->a, arr->size);

   pthread_exit(0); //to exit a thread
}

void bubblesort(int *arr,int n, bool(*oprn)(const void *,const void
*))
{
   for(int i=0; i<n-1; i++)
   {
       for(int j=0; j<n-i-1; j++)
       {
           if(oprn(&arr[j],&arr[j+1]))
               swap(&arr[j],&arr[j+1]);
       }
   }
}

bool asc(const void *a, const void *b)
{
   return *(int *)a > *(int *)b;
}

bool desc(const void *a, const void *b)
{
   return *(int *)a < *(int *)b;
}

void swap(int *a,int *b)
{
   int c = *a;
   *a = *b;
   *b = c;
}

void print(int *arr,int n)
{
   for(int i=0; i<n; i++)
   {
       printf("%d ",arr[i]);
   }
   printf("\n");
}
```

Output:



```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q2_Sort.c -o Q2_Sort
-lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q2_Sort

Wrong usage.. Syntax: ./Q2_Sort <Array elements>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q2_Sort 12 10 0 -4 -7 2
4 35 9 34 0

The size of Array : 10
The Elements of the Array : 12 10 0 -4 -7 24 35 9 34 0

Array Sorted in Ascending order : -7 -4 0 0 9 10 12 24 34 35
Array Sorted in Descending order : 35 34 24 12 10 9 0 0 -4 -7

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q2_Sort 24 32 56 12 76
-1 -5 -87 0 23 24 76 35 100 234

The size of Array : 15
The Elements of the Array : 24 32 56 12 76 -1 -5 -87 0 23 24 76 35 100 234

Array Sorted in Ascending order : -87 -5 -1 0 12 23 24 24 32 35 56 76 76 100 234
Array Sorted in Descending order : 234 100 76 76 56 35 32 24 24 23 12 0 -1 -5 -87

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

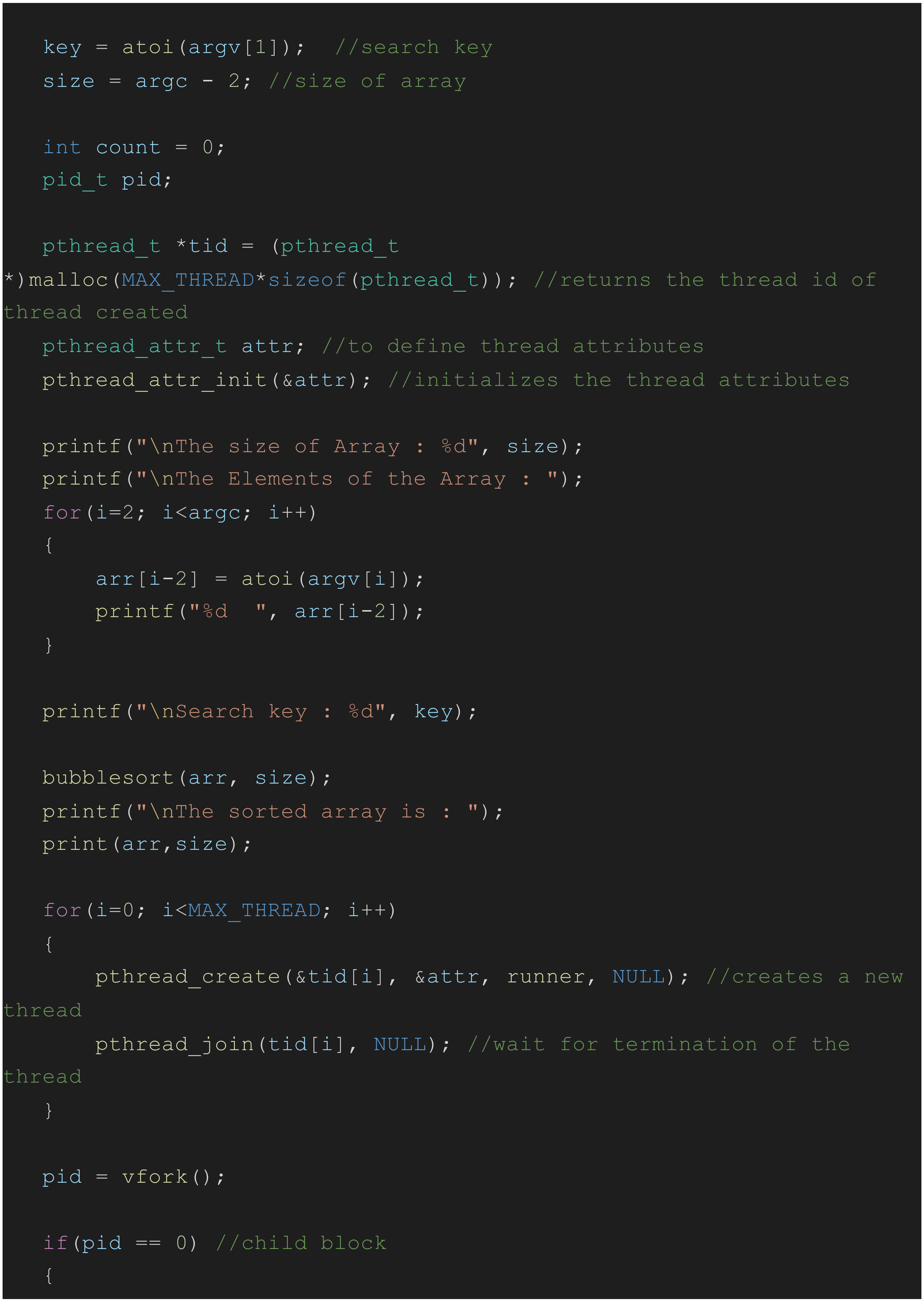
Explanation:

We implemented the bubble sort using the function pointer for both ascending and descending sort. Here we created 2 threads, one for sorting the given array in ascending order using bubble sort and another thread will sort the array in descending order using bubble sort.

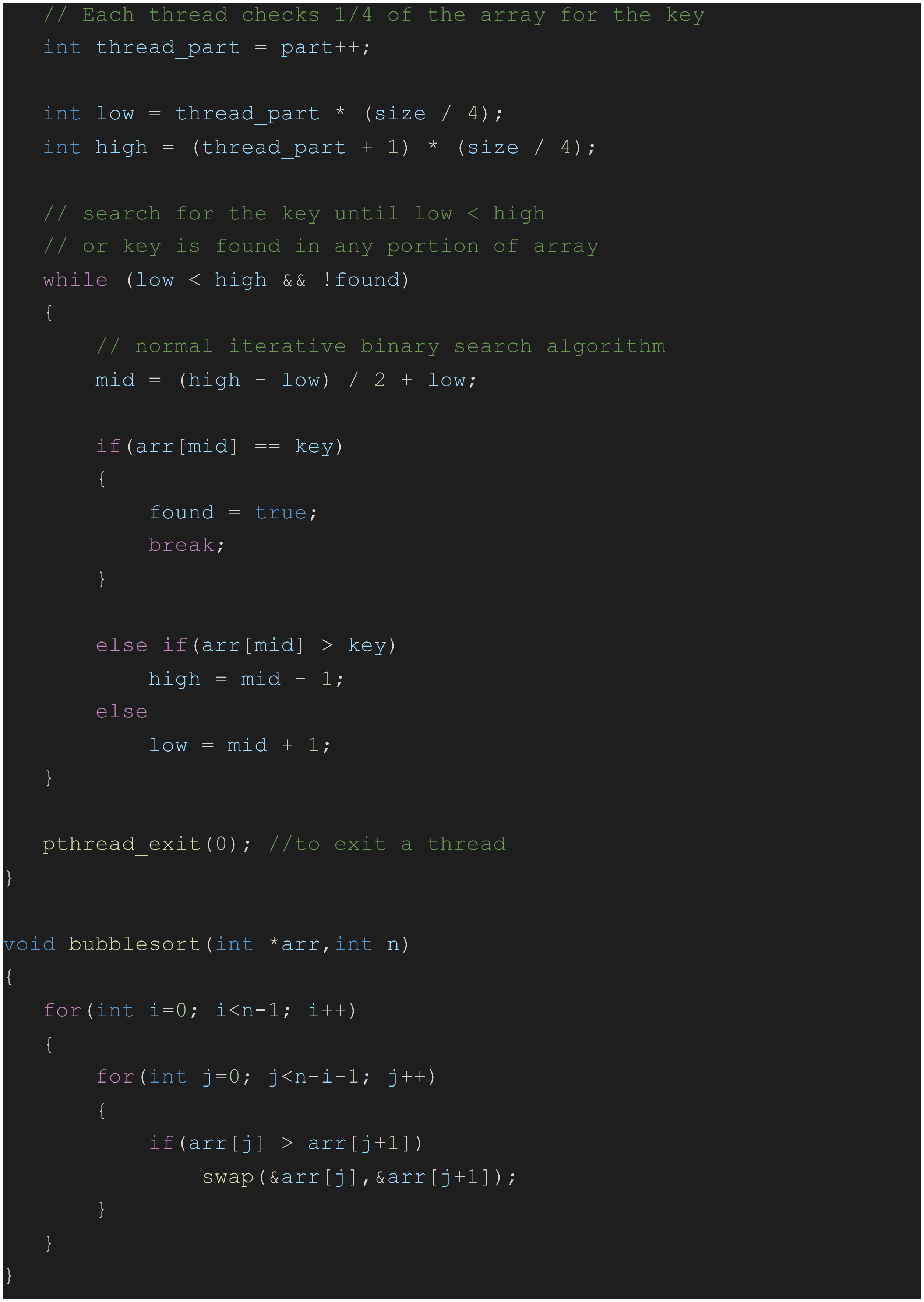
(3) Implement a multithreaded version of binary search. By default, you can implement a search for the first occurrence and later extend to support multiple occurrence (duplicated elements search as well)

Filename: Q3_BinarySearch.c

```
#include<stdio.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdbool.h>
#include<pthread.h>

#define MAX_THREAD 4 //maximum number of threads

int mid = -1, i, key, size, part = 0, arr[10];
bool found = false;

void *runner(void *arg);
void bubblesort(int *arr, int n);
void swap(int *a,int *b);
void print(int *arr,int n);

int main( int argc, char *argv[])
{
   //Validate correct number of arguments
   if(argc < 3)
   {
       printf("\nWrong usage.. Syntax: %s <Key> <Array elements>
\n\n", argv[0]);
       exit(EXIT_FAILURE);
   }
```



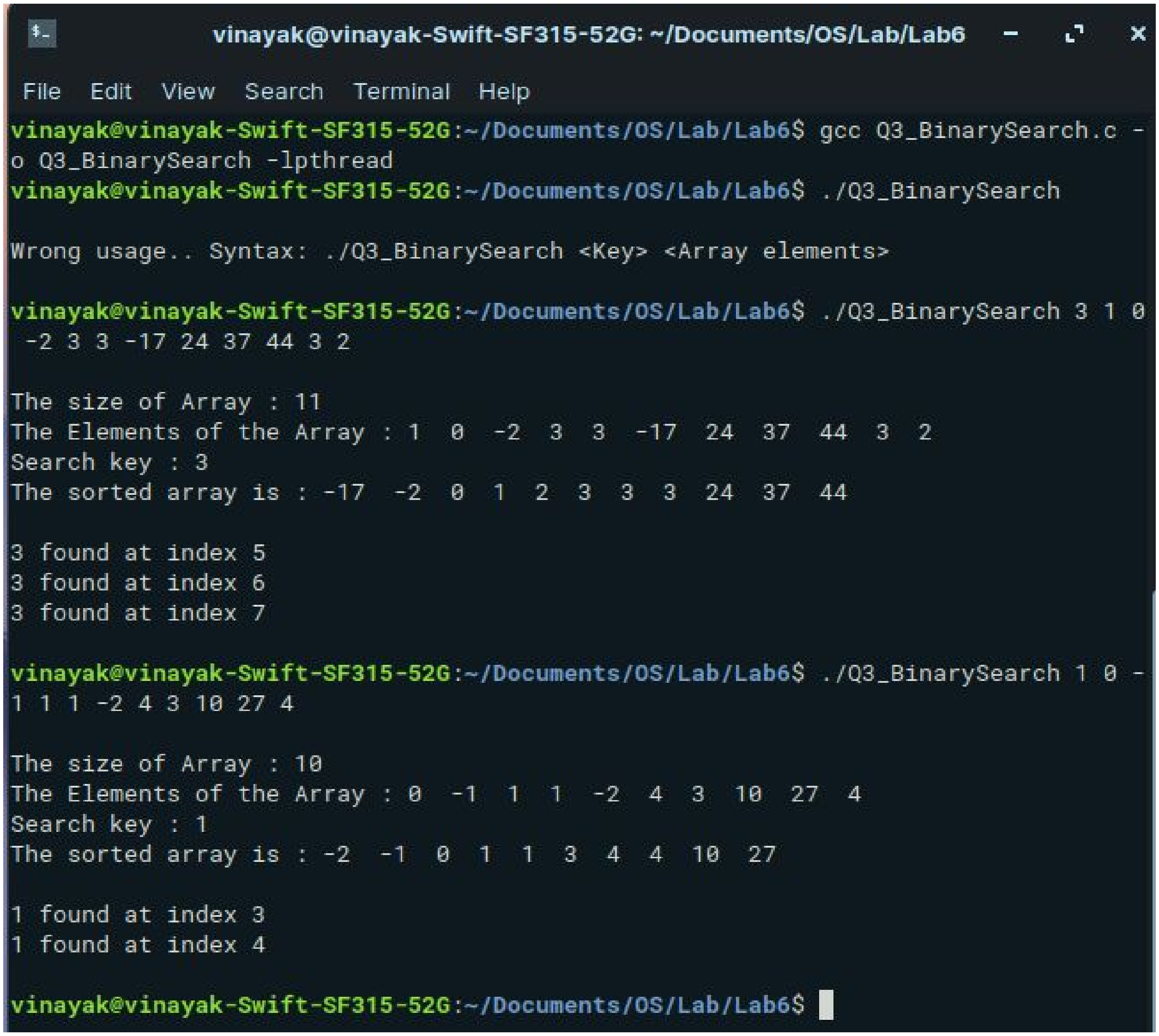
```
   key = atoi(argv[1]);  //search key
   size = argc - 2; //size of array

   int count = 0;
   pid_t pid;

   pthread_t *tid = (pthread_t
*)malloc(MAX_THREAD*sizeof(pthread_t)); //returns the thread id of
thread created
   pthread_attr_t attr; //to define thread attributes
   pthread_attr_init(&attr); //initializes the thread attributes

   printf("\nThe size of Array : %d", size);
   printf("\nThe Elements of the Array : ");
   for(i=2; i<argc; i++)
   {
       arr[i-2] = atoi(argv[i]);
       printf("%d  ", arr[i-2]);
   }

   printf("\nSearch key : %d", key);

   bubblesort(arr, size);
   printf("\nThe sorted array is : ");
   print(arr,size);

   for(i=0; i<MAX_THREAD; i++)
   {
       pthread_create(&tid[i], &attr, runner, NULL); //creates a new
thread
       pthread_join(tid[i], NULL); //wait for termination of the
thread
   }

   pid = vfork();

   if(pid == 0) //child block
   {
       i = mid - 1;
       while(i>=0 && arr[i] == key) //search for other positions where key can be present i.e. first half of mid
       {
           printf("\n%d found at index %d\n", key, i);
           i--;
       }

       if(found == false) //search for first appearance of key in array
       {
           printf("\n%d is not found in the array\n", key);
           exit(1);
       }
       else
           printf("\n%d found at index %d\n", key, mid);

       exit(0);
   }

   else //parent block
   {
       wait(NULL);

       i = mid + 1;
       while(i<size && arr[i] == key) //search for other positions where key can be present i.e. later half of mid
       {
           printf("%d found at index %d\n", key, i);
           i++;
       }
   }

   printf("\n");
   return 0;
}

void *runner(void *arg)
{
```



```
    // Each thread checks 1/4 of the array for the key
    int thread_part = part++;

    int low = thread_part * (size / 4);
    int high = (thread_part + 1) * (size / 4);

    // search for the key until low < high
    // or key is found in any portion of array
    while (low < high && !found)
    {
        // normal iterative binary search algorithm
        mid = (high - low) / 2 + low;

        if(arr[mid] == key)
        {
            found = true;
            break;
        }

        else if(arr[mid] > key)
            high = mid - 1;
        else
            low = mid + 1;
    }

    pthread_exit(0); //to exit a thread
}

void bubblesort(int *arr,int n)
{
    for(int i=0; i<n-1; i++)
    {
        for(int j=0; j<n-i-1; j++)
        {
            if(arr[j] > arr[j+1])
                swap(&arr[j],&arr[j+1]);
        }
    }
}
```

```c
void swap(int *a,int *b)
{
    int c = *a;
    *a = *b;
    *b = c;
}

void print(int *arr,int n)
{
    for(int i=0; i<n; i++)
    {
        printf("%d  ",arr[i]);
    }
    printf("\n");
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q3_BinarySearch.c -
o Q3_BinarySearch -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q3_BinarySearch

Wrong usage.. Syntax: ./Q3_BinarySearch <Key> <Array elements>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q3_BinarySearch 3 1 0
 -2 3 3 -17 24 37 44 3 2

The size of Array : 11
The Elements of the Array : 1  0  -2  3  3  -17  24  37  44  3  2
Search key : 3
The sorted array is : -17  -2  0  1  2  3  3  3  24  37  44

3 found at index 5
3 found at index 6
3 found at index 7

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q3_BinarySearch 1 0 -
1 1 1 -2 4 3 10 27 4

The size of Array : 10
The Elements of the Array : 0  -1  1  1  -2  4  3  10  27  4
Search key : 1
The sorted array is : -2  -1  0  1  1  3  4  4  10  27

1 found at index 3
1 found at index 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

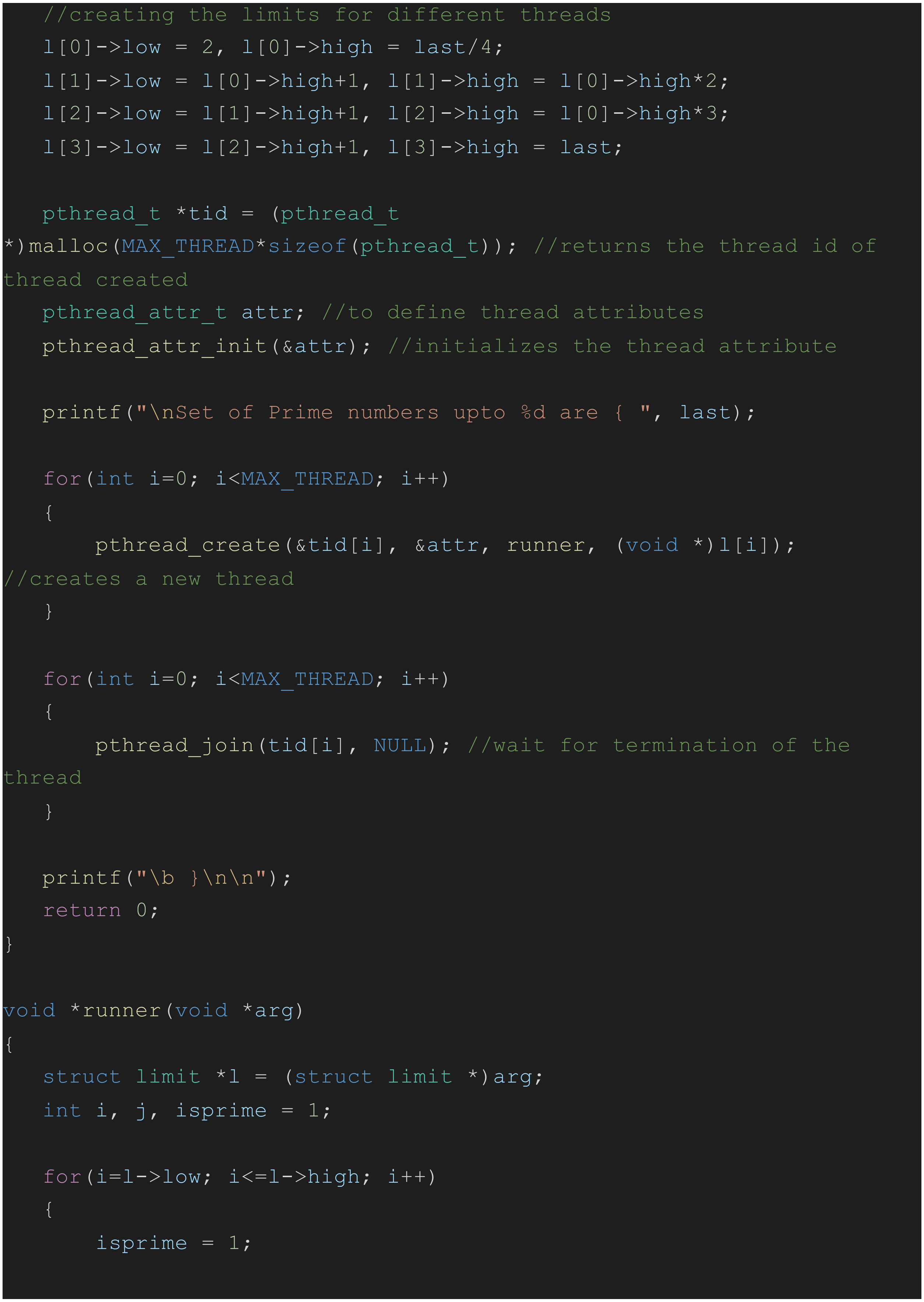
Explanation:

We will first sort the unsorted array, then for binary search we created 4 threads which will search for the key in their respective limits **{low, high},** Once we find the key we will check if there is any duplicate key present in the array, if present we will also search it which we will find using the index at which first occurrence of key is found, as the remaining duplicates will be around the index at which the first occurrence of key is found.

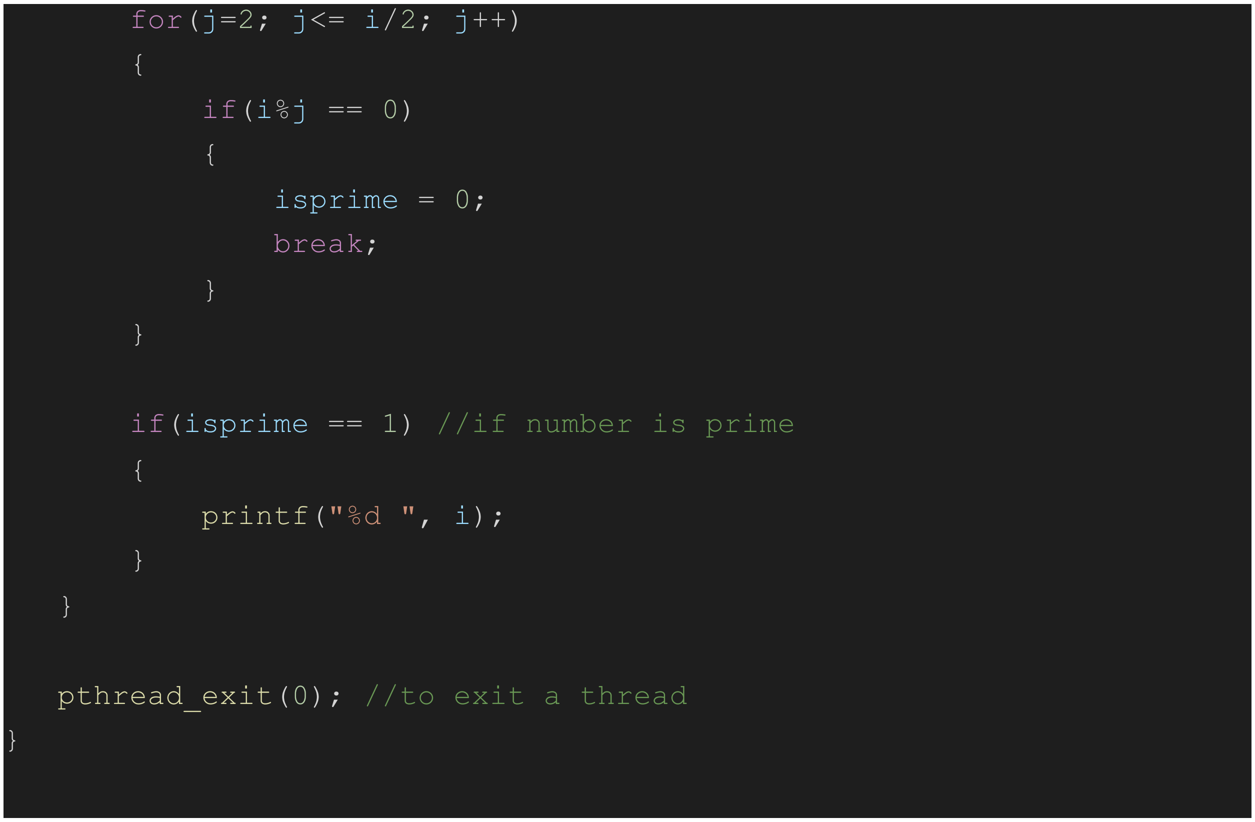
(4) Generation of Prime Numbers upto a limit supplied as Command Line Parameter.

Filename: Q4_PrimeNumbers.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#define MAX_THREAD 4 //maximum number of threads

struct limit
{
   int low;
   int high;
};

void *runner(void *arg);

int main(int argc, char *argv[])
{
   //Validate the correct usage of command line arguments
   if(argc != 2)
   {
       printf("\nSyntax: %s <Upper Limit>\n\n", argv[0]);
       exit(EXIT_FAILURE);
   }

   if(atoi(argv[1]) < 0)
   {
       printf("\nn value should be greater than zero...\n\n");
       exit(EXIT_FAILURE);
   }

   int last = atoi(argv[1]);
   struct limit *l[4];

   for(int i=0; i<4; i++)
       l[i] = (struct limit *)malloc(sizeof(struct limit));
```

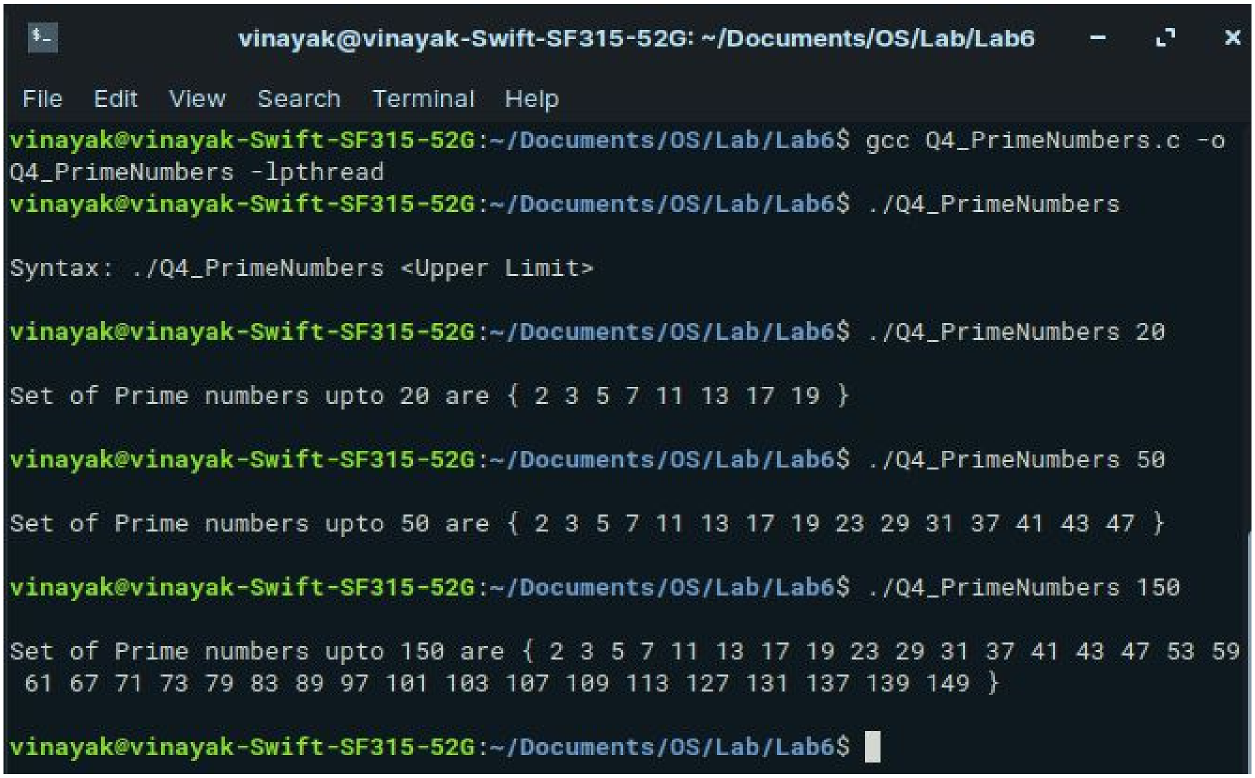


```
    //creating the limits for different threads
    l[0]->low = 2, l[0]->high = last/4;
    l[1]->low = l[0]->high+1, l[1]->high = l[0]->high*2;
    l[2]->low = l[1]->high+1, l[2]->high = l[0]->high*3;
    l[3]->low = l[2]->high+1, l[3]->high = last;

    pthread_t *tid = (pthread_t
*)malloc(MAX_THREAD*sizeof(pthread_t)); //returns the thread id of
thread created
    pthread_attr_t attr; //to define thread attributes
    pthread_attr_init(&attr); //initializes the thread attribute

    printf("\nSet of Prime numbers upto %d are { ", last);

    for(int i=0; i<MAX_THREAD; i++)
    {
        pthread_create(&tid[i], &attr, runner, (void *)l[i]);
//creates a new thread
    }

    for(int i=0; i<MAX_THREAD; i++)
    {
        pthread_join(tid[i], NULL); //wait for termination of the
thread
    }

    printf("\b }\n\n");
    return 0;
}

void *runner(void *arg)
{
    struct limit *l = (struct limit *)arg;
    int i, j, isprime = 1;

    for(i=l->low; i<=l->high; i++)
    {
        isprime = 1;
```



```
        for(j=2; j<= i/2; j++)
        {
            if(i%j == 0)
            {
                isprime = 0;
                break;
            }
        }

        if(isprime == 1) //if number is prime
        {
            printf("%d ", i);
        }
    }

    pthread_exit(0); //to exit a thread
}
```

Output:



vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q4_PrimeNumbers.c -o
Q4_PrimeNumbers -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q4_PrimeNumbers

Syntax: ./Q4_PrimeNumbers <Upper Limit>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q4_PrimeNumbers 20

Set of Prime numbers upto 20 are { 2 3 5 7 11 13 17 19 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q4_PrimeNumbers 50

Set of Prime numbers upto 50 are { 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q4_PrimeNumbers 150

Set of Prime numbers upto 150 are { 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

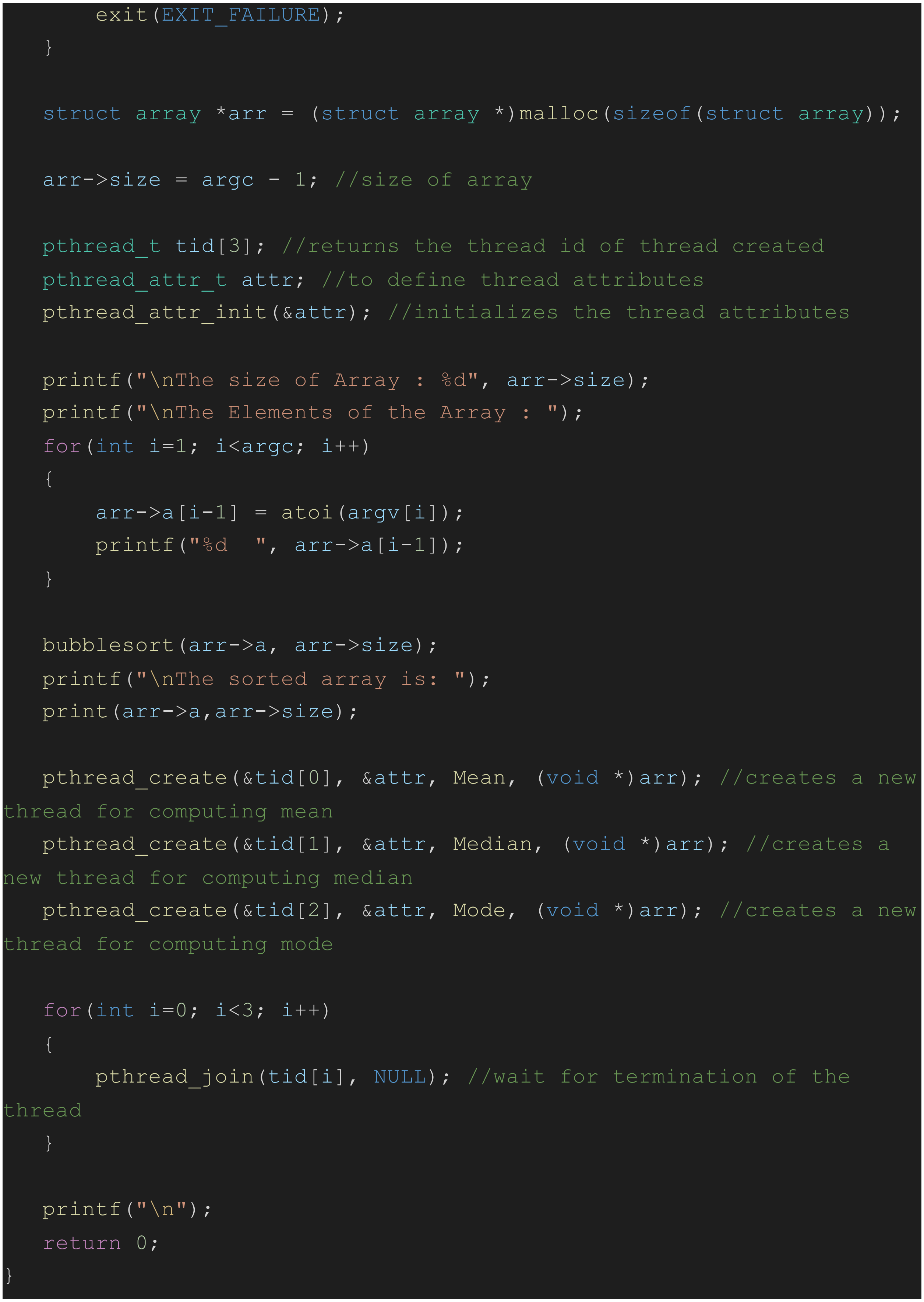
Explanation:

Here we created 4 threads and each thread will generate the prime numbers in their given range **{low, high}** which helps to provide parallelism to the process of generating prime numbers. Prime number generation is done using the logic if a number has only 2 factors (1 and number itself) then it is a prime number, else not.

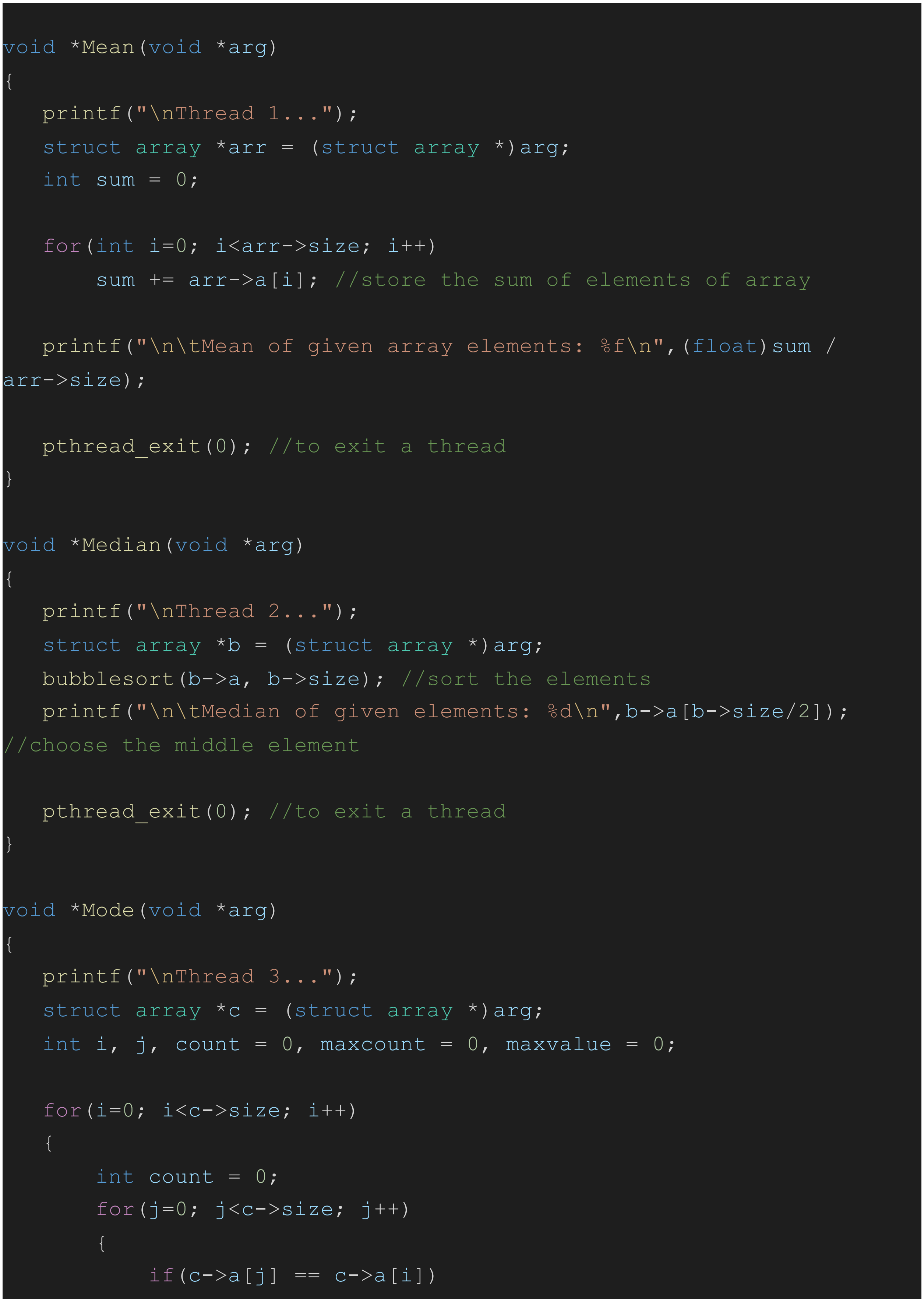
(5) Computation of Mean, Median, Mode for an array of integers.

Filename: Q5_MeanMedianMode.c

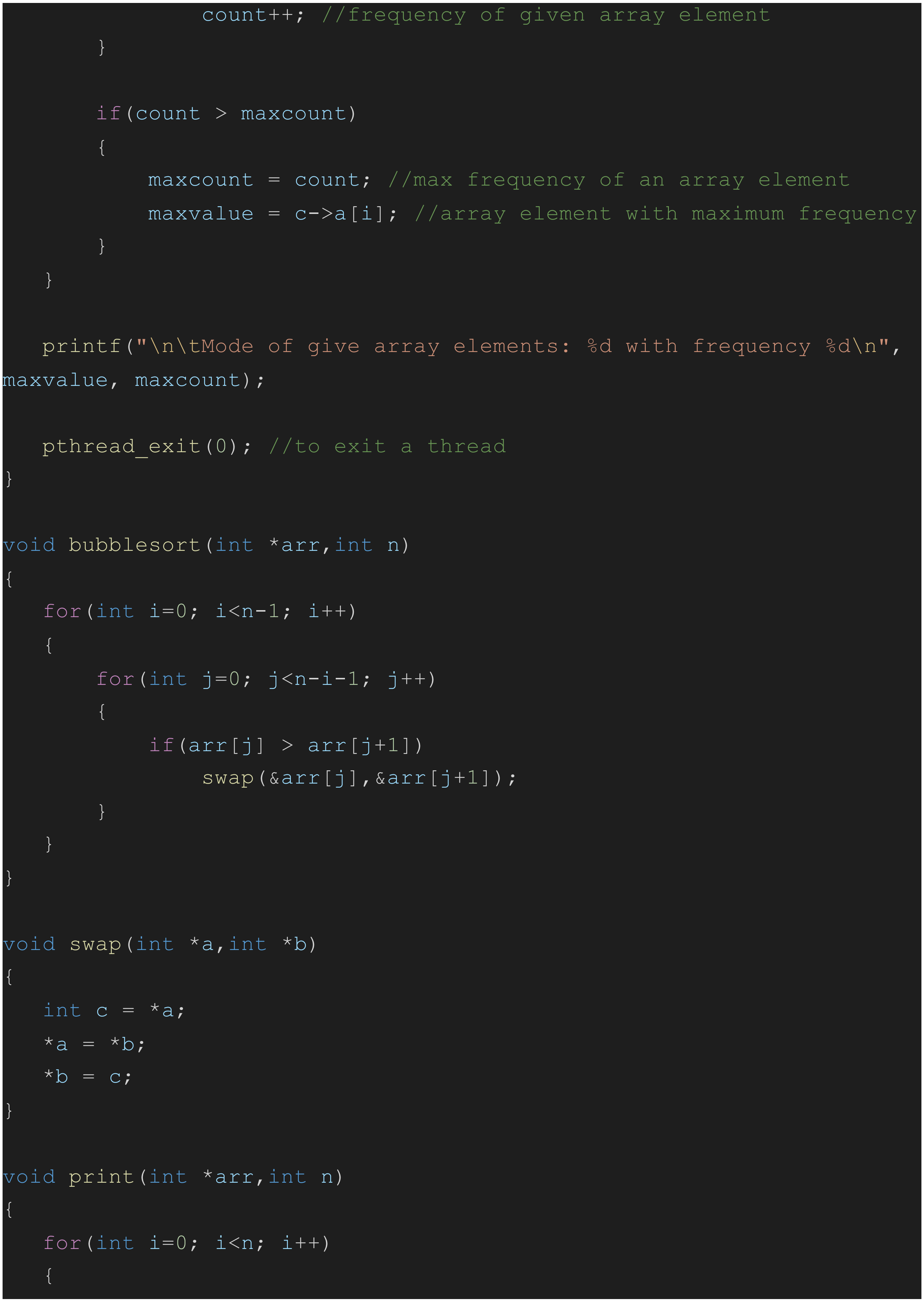
```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

struct array
{
   int a[100];
   int size;
};

void *Mean(void *arg);
void *Median(void *arg);
void *Mode(void *arg);
void bubblesort(int *arr, int n);
void swap(int *a,int *b);
void print(int *arr,int n);

int main(int argc, char *argv[])
{
   //Validate correct number of arguments
   if(argc < 2)
   {
       printf("\nWrong usage.. Syntax: %s <Array elements> \n\n",
argv[0]);
```



```
        exit(EXIT_FAILURE);
    }

    struct array *arr = (struct array *)malloc(sizeof(struct array));

    arr->size = argc - 1; //size of array

    pthread_t tid[3]; //returns the thread id of thread created
    pthread_attr_t attr; //to define thread attributes
    pthread_attr_init(&attr); //initializes the thread attributes

    printf("\nThe size of Array : %d", arr->size);
    printf("\nThe Elements of the Array : ");
    for(int i=1; i<argc; i++)
    {
        arr->a[i-1] = atoi(argv[i]);
        printf("%d  ", arr->a[i-1]);
    }

    bubblesort(arr->a, arr->size);
    printf("\nThe sorted array is: ");
    print(arr->a,arr->size);

    pthread_create(&tid[0], &attr, Mean, (void *)arr); //creates a new
thread for computing mean
    pthread_create(&tid[1], &attr, Median, (void *)arr); //creates a
new thread for computing median
    pthread_create(&tid[2], &attr, Mode, (void *)arr); //creates a new
thread for computing mode

    for(int i=0; i<3; i++)
    {
        pthread_join(tid[i], NULL); //wait for termination of the
thread
    }

    printf("\n");
    return 0;
}
```

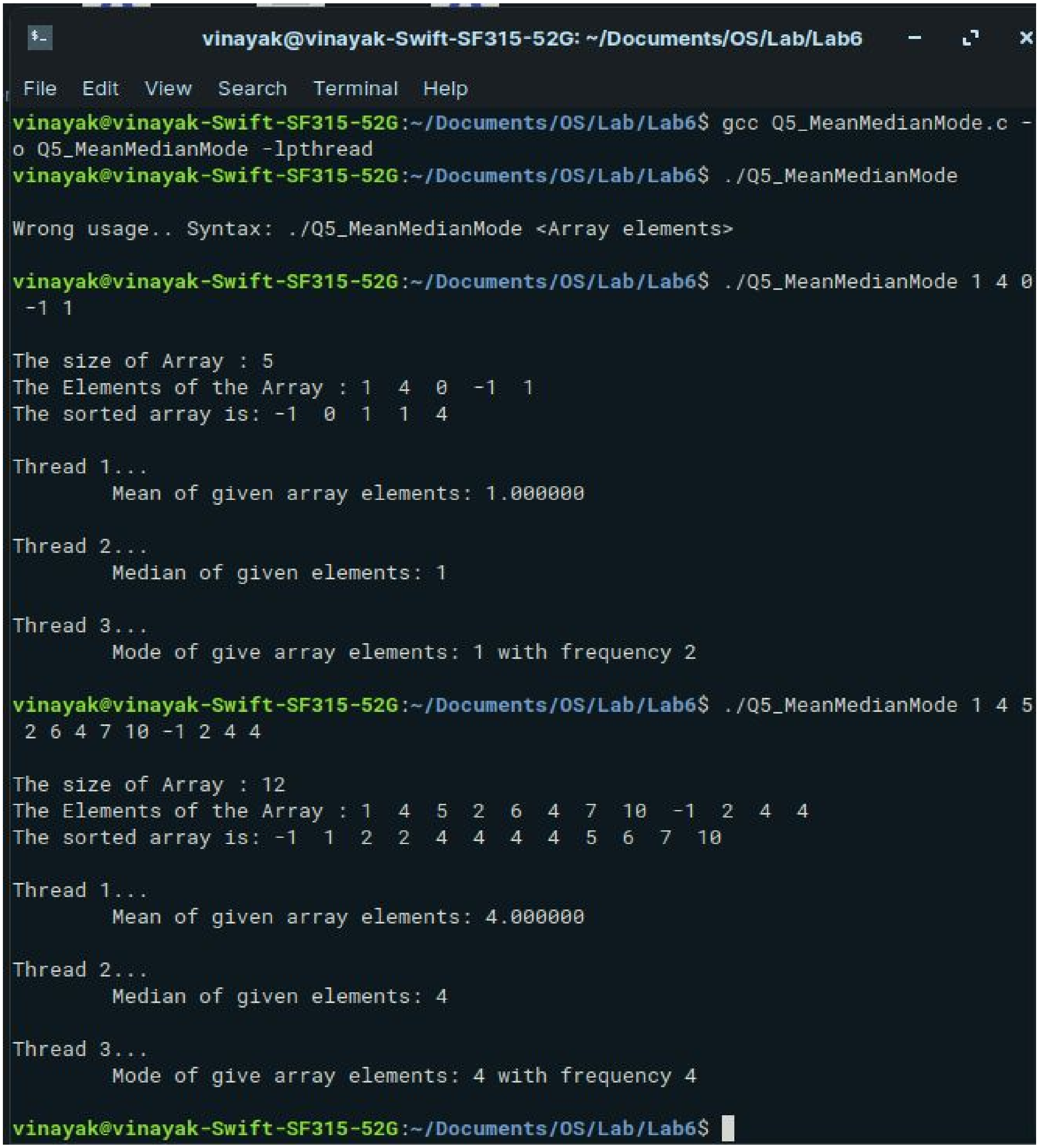


```
void *Mean(void *arg)
{
   printf("\nThread 1...");
   struct array *arr = (struct array *)arg;
   int sum = 0;

   for(int i=0; i<arr->size; i++)
       sum += arr->a[i]; //store the sum of elements of array

   printf("\n\tMean of given array elements: %f\n",(float)sum /
arr->size);

   pthread_exit(0); //to exit a thread
}

void *Median(void *arg)
{
   printf("\nThread 2...");
   struct array *b = (struct array *)arg;
   bubblesort(b->a, b->size); //sort the elements
   printf("\n\tMedian of given elements: %d\n",b->a[b->size/2]);
//choose the middle element

   pthread_exit(0); //to exit a thread
}

void *Mode(void *arg)
{
   printf("\nThread 3...");
   struct array *c = (struct array *)arg;
   int i, j, count = 0, maxcount = 0, maxvalue = 0;

   for(i=0; i<c->size; i++)
   {
       int count = 0;
       for(j=0; j<c->size; j++)
       {
           if(c->a[j] == c->a[i])
```



```
               count++; //frequency of given array element
       }

       if(count > maxcount)
       {
           maxcount = count; //max frequency of an array element
           maxvalue = c->a[i]; //array element with maximum frequency
       }
   }

   printf("\n\tMode of give array elements: %d with frequency %d\n",
maxvalue, maxcount);

   pthread_exit(0); //to exit a thread
}

void bubblesort(int *arr,int n)
{
   for(int i=0; i<n-1; i++)
   {
       for(int j=0; j<n-i-1; j++)
       {
           if(arr[j] > arr[j+1])
               swap(&arr[j],&arr[j+1]);
       }
   }
}

void swap(int *a,int *b)
{
   int c = *a;
   *a = *b;
   *b = c;
}

void print(int *arr,int n)
{
   for(int i=0; i<n; i++)
   {
       printf("%d  ",arr[i]);
   }
   printf("\n");
}
```

Output:



vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q5_MeanMedianMode.c -
o Q5_MeanMedianMode -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q5_MeanMedianMode

Wrong usage.. Syntax: ./Q5_MeanMedianMode <Array elements>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q5_MeanMedianMode 1 4 0
 -1 1

The size of Array : 5
The Elements of the Array : 1  4  0  -1  1
The sorted array is: -1  0  1  1  4

Thread 1...
        Mean of given array elements: 1.000000

Thread 2...
        Median of given elements: 1

Thread 3...
        Mode of give array elements: 1 with frequency 2

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q5_MeanMedianMode 1 4 5
 2 6 4 7 10 -1 2 4 4

The size of Array : 12
The Elements of the Array : 1  4  5  2  6  4  7  10  -1  2  4  4
The sorted array is: -1  1  2  2  4  4  4  4  5  6  7  10

Thread 1...
        Mean of given array elements: 4.000000

Thread 2...
        Median of given elements: 4

Thread 3...
        Mode of give array elements: 4 with frequency 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

Explanation:

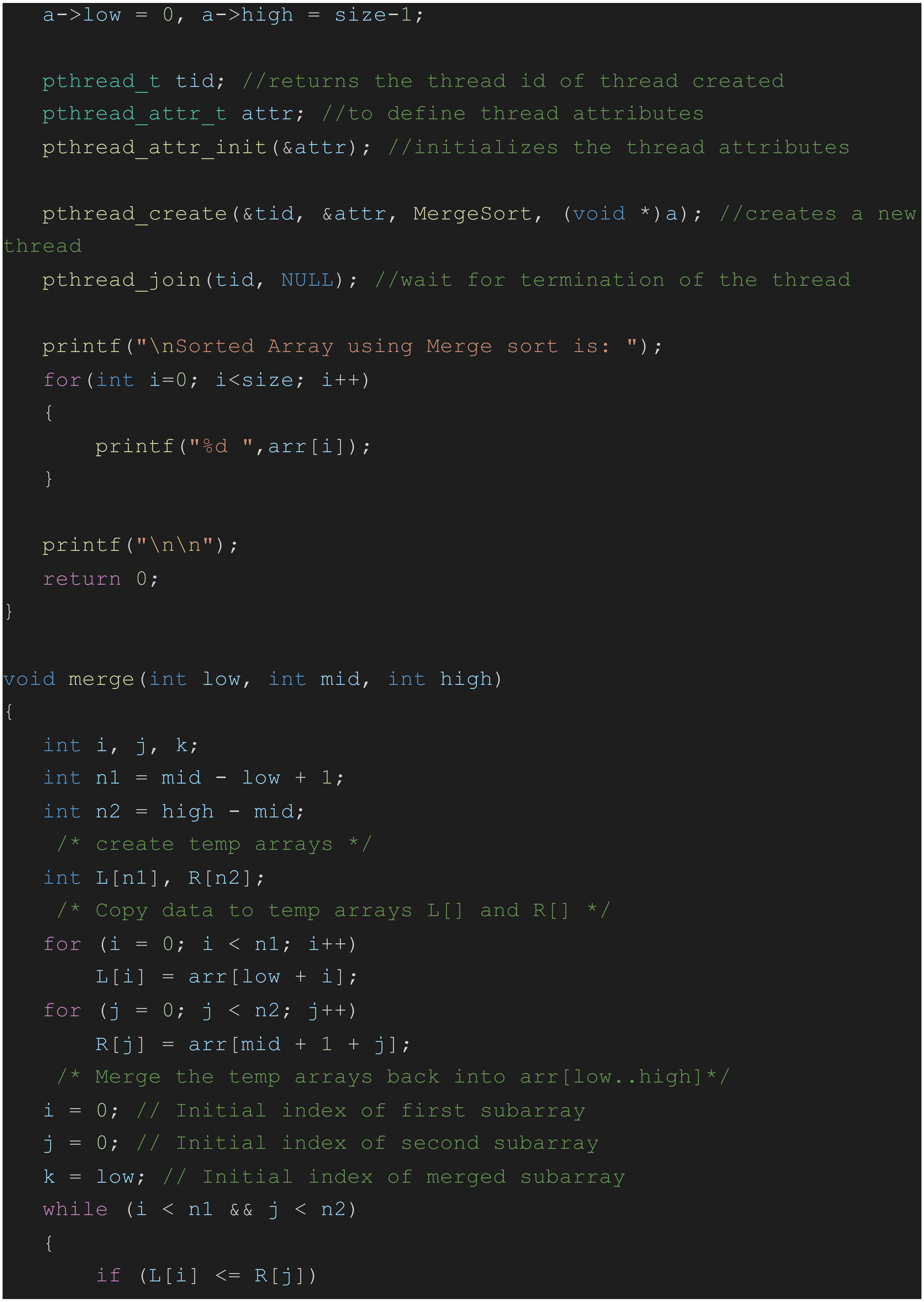
For the computation of mean, median and mode we created 3 threads, one for each process to provide parallelism.
Mean is the average of the given elements of the array.

Median is the middle element of the sorted array. For sorting we used bubble sort.
Mode is the element with the highest number of occurrences in the array.

(6) Implement Merge Sort and Quick Sort in a multithreaded fashion.

Filename: Q6_MergeSort.c

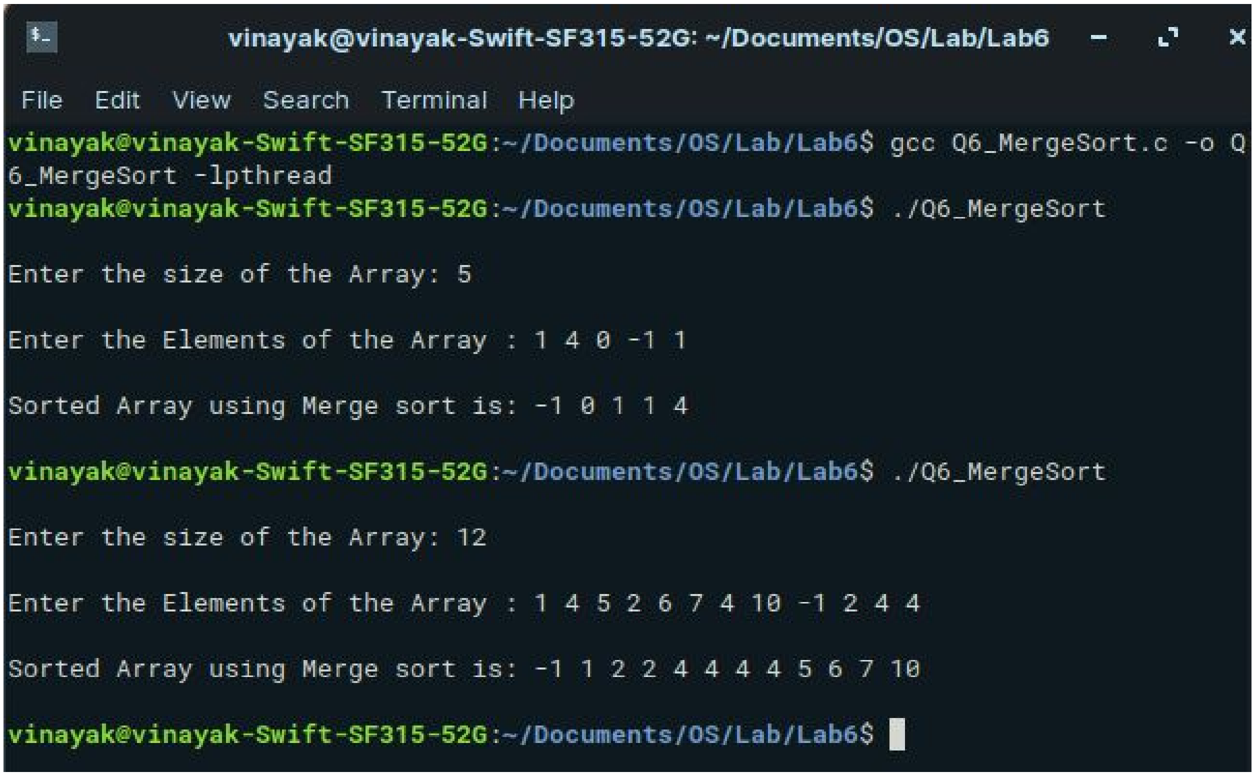
```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#define MAX_THREAD 4
int arr[100];

struct limit
{
   int low;
   int high;
};

void merge(int low, int mid, int high);
void *MergeSort(void *arg);

int main()
{
   struct limit *a = (struct limit *)malloc(sizeof(struct limit));

   int size;
   printf("\nEnter the size of the Array: ");
   scanf("%d", &size);

   printf("\nEnter the Elements of the Array : ");
   for(int i=0; i<size; i++)
   {
       scanf("%d", &arr[i]);
   }
```



```
   a->low = 0, a->high = size-1;

   pthread_t tid; //returns the thread id of thread created
   pthread_attr_t attr; //to define thread attributes
   pthread_attr_init(&attr); //initializes the thread attributes

   pthread_create(&tid, &attr, MergeSort, (void *)a); //creates a new
thread
   pthread_join(tid, NULL); //wait for termination of the thread

   printf("\nSorted Array using Merge sort is: ");
   for(int i=0; i<size; i++)
   {
       printf("%d ",arr[i]);
   }

   printf("\n\n");
   return 0;
}

void merge(int low, int mid, int high)
{
   int i, j, k;
   int n1 = mid - low + 1;
   int n2 = high - mid;
    /* create temp arrays */
   int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
       L[i] = arr[low + i];
   for (j = 0; j < n2; j++)
       R[j] = arr[mid + 1 + j];
    /* Merge the temp arrays back into arr[low..high]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = low; // Initial index of merged subarray
   while (i < n1 && j < n2)
   {
       if (L[i] <= R[j])
```

```
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
     /* Copy the remaining elements of L[], if there are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
     /* Copy the remaining elements of R[], if there are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void *MergeSort(void *arg)
{
    struct limit *a = (struct limit *)arg;
    int i;

    struct limit *left = (struct limit *)malloc(sizeof(struct limit));
//for left partition
    struct limit *right = (struct limit *)malloc(sizeof(struct
limit)); //for right partition

    if(a->low < a->high)
    {
        int mid = a->low + (a->high - a->low) / 2;

        pthread_t tid[2]; //returns the thread id of thread created
        pthread_attr_t attr; //to define thread attributes
        pthread_attr_init(&attr); //initializes the thread attributes

        left->low = a->low, left->high = mid;
        right->low = mid+1, right->high = a->high;

        pthread_create(&tid[0], &attr, MergeSort, (void
*)left);//creates a new thread for left part
        pthread_create(&tid[1], &attr, MergeSort, (void
*)right);//creates a new thread for right part

        for(i=0; i<2; i++)
        {
            pthread_join(tid[i], NULL); //wait for termination of the
thread
        }

        merge(left->low, mid, right->high);
    }

    pthread_exit(0); //to exit a thread
}
```

Output:



```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q6_MergeSort.c -o Q
6_MergeSort -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q6_MergeSort

Enter the size of the Array: 5

Enter the Elements of the Array : 1 4 0 -1 1

Sorted Array using Merge sort is: -1 0 1 1 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q6_MergeSort

Enter the size of the Array: 12

Enter the Elements of the Array : 1 4 5 2 6 7 4 10 -1 2 4 4

Sorted Array using Merge sort is: -1 1 2 2 4 4 4 4 5 6 7 10

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

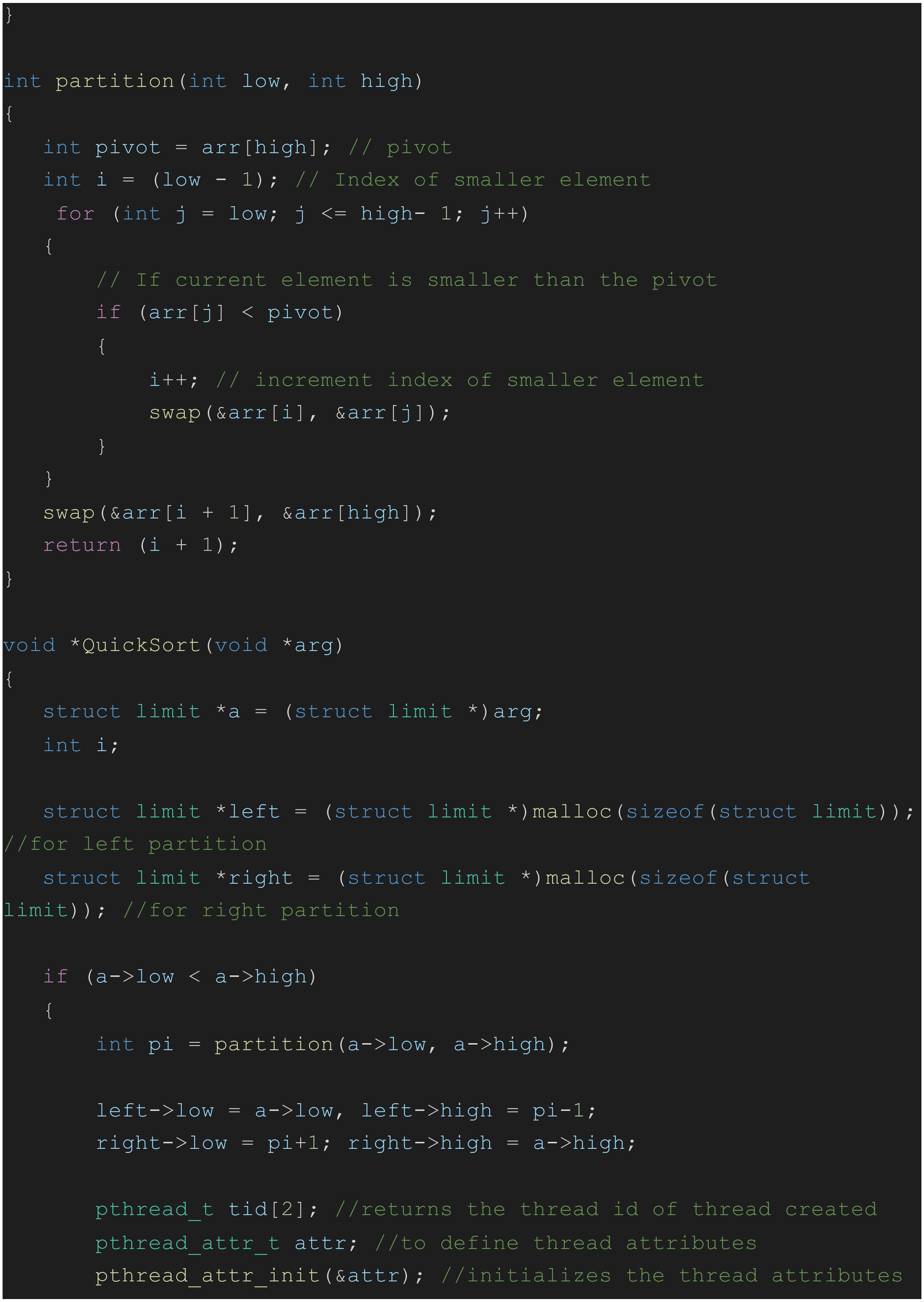
Explanation:

For the merge sort we implemented the threading for the partitions.
We created 2 threads, one for the left partition and one for the right partition which helps to provide parallelism. After completion of both the processes we will merge both the threads.

Filename: Q6_QuickSort.c

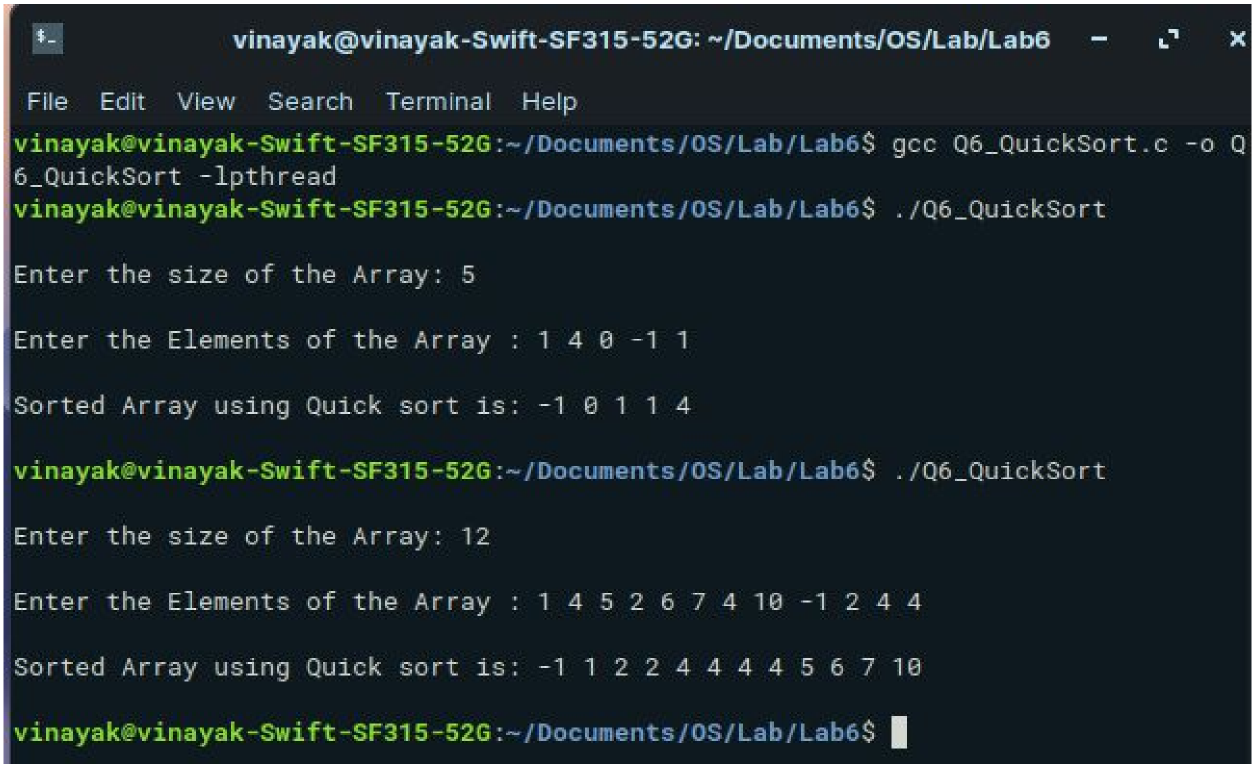
```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#define MAX_THREAD 4
int arr[100];

struct limit
{
   int low;
```

```
    int high;
};

int partition(int low, int high);
void *QuickSort(void *arg);
void swap(int *a, int *b);

int main()
{
    struct limit *a = (struct limit *)malloc(sizeof(struct limit));

    int size;
    printf("\nEnter the size of the Array: ");
    scanf("%d", &size);

    printf("\nEnter the Elements of the Array : ");
    for(int i=0; i<size; i++)
    {
        scanf("%d", &arr[i]);
    }

    a->low = 0, a->high = size-1;

    pthread_t tid; //returns the thread id of thread created
    pthread_attr_t attr; //to define thread attributes
    pthread_attr_init(&attr); //initializes the thread attributes

    pthread_create(&tid, &attr, QuickSort, (void *)a); //creates a new
thread
    pthread_join(tid, NULL); //wait for termination of the thread

    printf("\nSorted Array using Quick sort is: ");
    for(int i=0; i<size; i++)
    {
        printf("%d ",arr[i]);
    }

    printf("\n\n");
    return 0;
```



```
}

int partition(int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element
     for (int j = low; j <= high- 1; j++)
    {
        // If current element is smaller than the pivot
        if (arr[j] < pivot)
        {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

void *QuickSort(void *arg)
{
    struct limit *a = (struct limit *)arg;
    int i;

    struct limit *left = (struct limit *)malloc(sizeof(struct limit));
//for left partition
    struct limit *right = (struct limit *)malloc(sizeof(struct
limit)); //for right partition

    if (a->low < a->high)
    {
        int pi = partition(a->low, a->high);

        left->low = a->low, left->high = pi-1;
        right->low = pi+1; right->high = a->high;

        pthread_t tid[2]; //returns the thread id of thread created
        pthread_attr_t attr; //to define thread attributes
        pthread_attr_init(&attr); //initializes the thread attributes
        pthread_create(&tid[0], &attr, QuickSort, (void
*)left);//creates a new thread for left part
        pthread_create(&tid[1], &attr, QuickSort, (void
*)right);//creates a new thread for right part

        for(i=0; i<2; i++)
        {
            pthread_join(tid[i], NULL); //wait for termination of the
thread
        }
    }

    pthread_exit(0); //to exit a thread
}

void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q6_QuickSort.c -o Q
6_QuickSort -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q6_QuickSort

Enter the size of the Array: 5

Enter the Elements of the Array : 1 4 0 -1 1

Sorted Array using Quick sort is: -1 0 1 1 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q6_QuickSort

Enter the size of the Array: 12

Enter the Elements of the Array : 1 4 5 2 6 7 4 10 -1 2 4 4

Sorted Array using Quick sort is: -1 1 2 2 4 4 4 4 5 6 7 10

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$
```

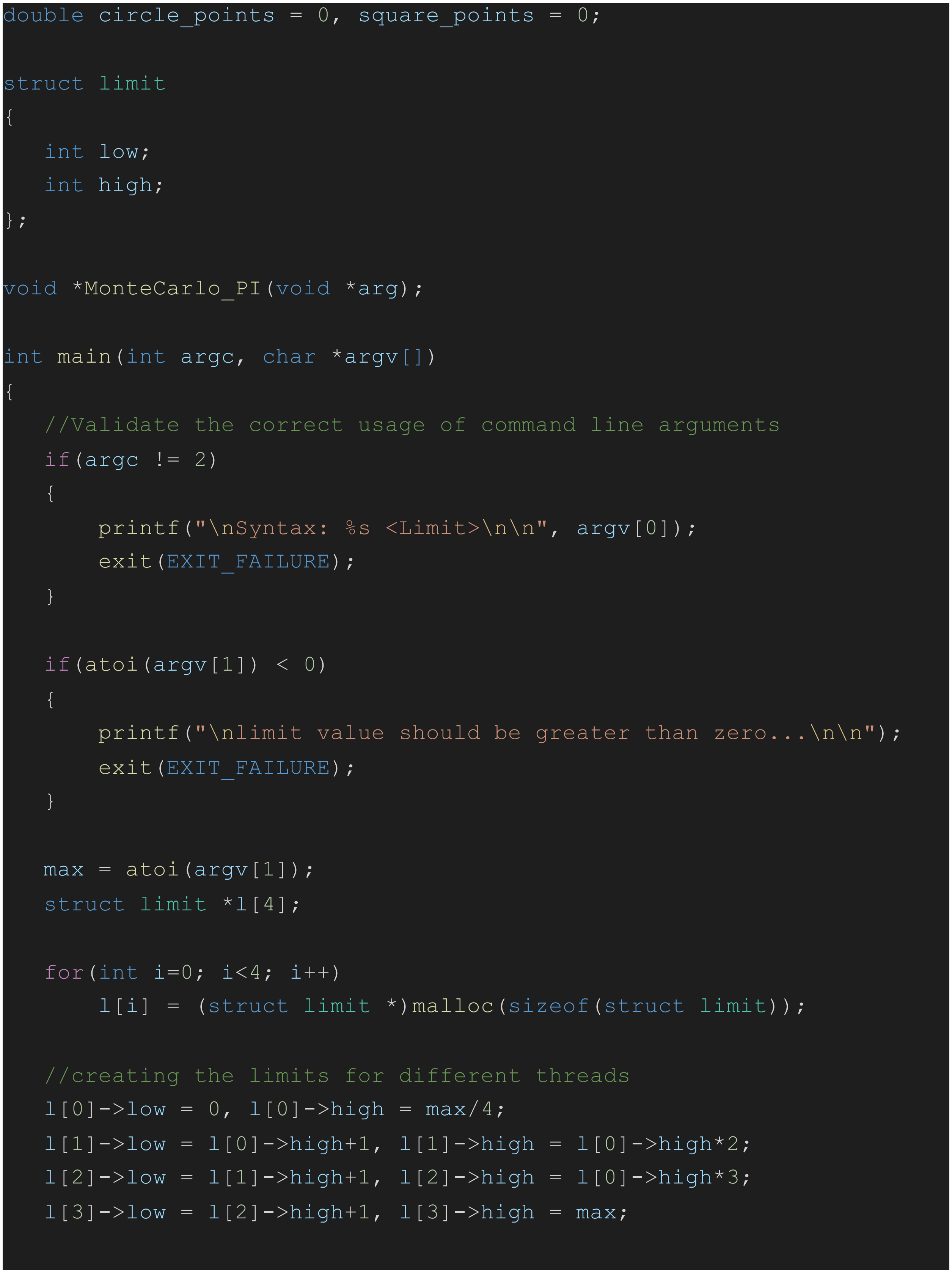
Explanation:

Multithreading applied here is similar to merge sort, here 2 threads are created for left partition and another for right partition across pivot point. And then after completion of task, we join the threads, which is basically waiting for termination of threads.

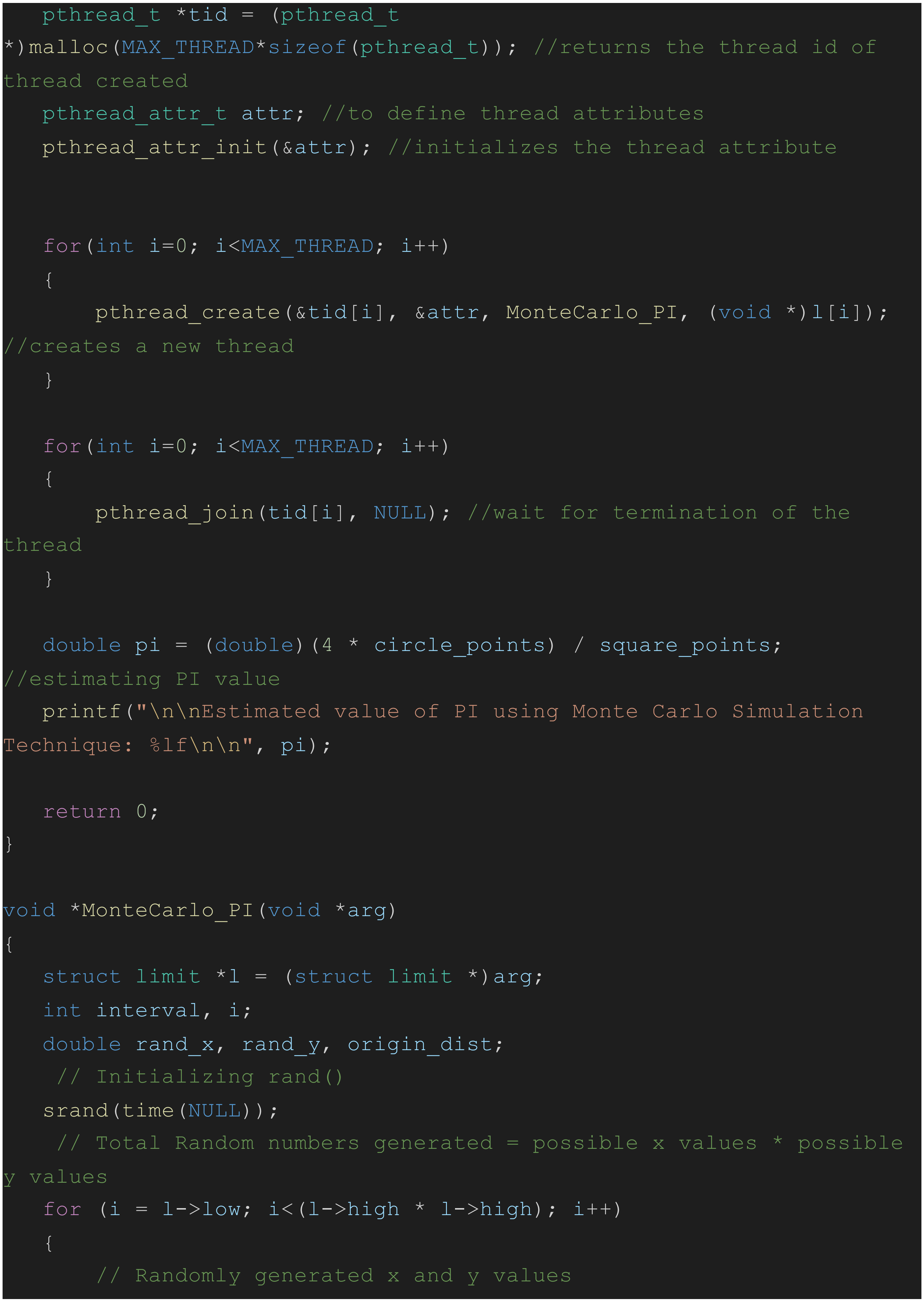
(7) Estimation of PI Value using Monte Carlo simulation technique (refer the internet for the method..) using threads.

Filename: Q7_MonteCarlo_EstimatePI.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

#define MAX_THREAD 4 //maximum number of threads

int max;
```

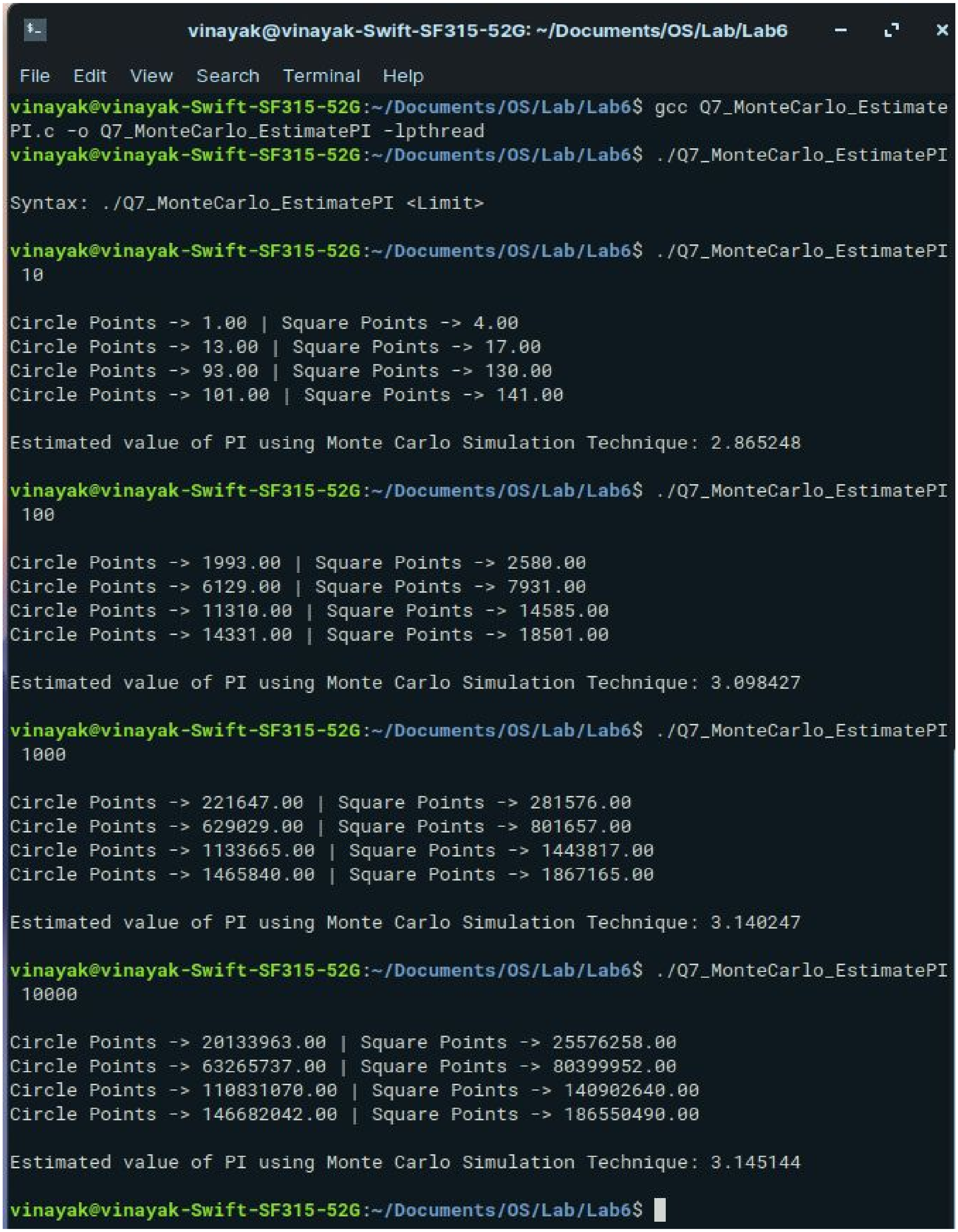


```c
double circle_points = 0, square_points = 0;

struct limit
{
   int low;
   int high;
};

void *MonteCarlo_PI(void *arg);

int main(int argc, char *argv[])
{
   //Validate the correct usage of command line arguments
   if(argc != 2)
   {
       printf("\nSyntax: %s <Limit>\n\n", argv[0]);
       exit(EXIT_FAILURE);
   }

   if(atoi(argv[1]) < 0)
   {
       printf("\nlimit value should be greater than zero...\n\n");
       exit(EXIT_FAILURE);
   }

   max = atoi(argv[1]);
   struct limit *l[4];

   for(int i=0; i<4; i++)
       l[i] = (struct limit *)malloc(sizeof(struct limit));

   //creating the limits for different threads
   l[0]->low = 0, l[0]->high = max/4;
   l[1]->low = l[0]->high+1, l[1]->high = l[0]->high*2;
   l[2]->low = l[1]->high+1, l[2]->high = l[0]->high*3;
   l[3]->low = l[2]->high+1, l[3]->high = max;
```



```
   pthread_t *tid = (pthread_t
*)malloc(MAX_THREAD*sizeof(pthread_t)); //returns the thread id of
thread created
   pthread_attr_t attr; //to define thread attributes
   pthread_attr_init(&attr); //initializes the thread attribute


   for(int i=0; i<MAX_THREAD; i++)
   {
       pthread_create(&tid[i], &attr, MonteCarlo_PI, (void *)l[i]);
//creates a new thread
   }

   for(int i=0; i<MAX_THREAD; i++)
   {
       pthread_join(tid[i], NULL); //wait for termination of the
thread
   }

   double pi = (double)(4 * circle_points) / square_points;
//estimating PI value
   printf("\n\nEstimated value of PI using Monte Carlo Simulation
Technique: %lf\n\n", pi);

   return 0;
}

void *MonteCarlo_PI(void *arg)
{
   struct limit *l = (struct limit *)arg;
   int interval, i;
   double rand_x, rand_y, origin_dist;
    // Initializing rand()
   srand(time(NULL));
    // Total Random numbers generated = possible x values * possible
y values
   for (i = l->low; i<(l->high * l->high); i++)
   {
       // Randomly generated x and y values
       rand_x = (double)(rand() % (l->high + 1)) / l->high;
       rand_y = (double)(rand() % (l->high + 1)) / l->high;

        // Distance between (x, y) from the origin
       origin_dist = rand_x * rand_x + rand_y * rand_y;

        // Checking if (x, y) lies inside the define
       // circle with R=1
       if (origin_dist <= 1)
           circle_points++;
        // Total number of points generated
       square_points++;
   }

   printf("\nCircle Points -> %.2f | Square Points -> %.2f",
circle_points, square_points);

   pthread_exit(0); //to exit a thread
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q7_MonteCarlo_Estimate
PI.c -o Q7_MonteCarlo_EstimatePI -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q7_MonteCarlo_EstimatePI

Syntax: ./Q7_MonteCarlo_EstimatePI <Limit>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q7_MonteCarlo_EstimatePI
 10

Circle Points -> 1.00 | Square Points -> 4.00
Circle Points -> 13.00 | Square Points -> 17.00
Circle Points -> 93.00 | Square Points -> 130.00
Circle Points -> 101.00 | Square Points -> 141.00

Estimated value of PI using Monte Carlo Simulation Technique: 2.865248

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q7_MonteCarlo_EstimatePI
 100

Circle Points -> 1993.00 | Square Points -> 2580.00
Circle Points -> 6129.00 | Square Points -> 7931.00
Circle Points -> 11310.00 | Square Points -> 14585.00
Circle Points -> 14331.00 | Square Points -> 18501.00

Estimated value of PI using Monte Carlo Simulation Technique: 3.098427

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q7_MonteCarlo_EstimatePI
 1000

Circle Points -> 221647.00 | Square Points -> 281576.00
Circle Points -> 629029.00 | Square Points -> 801657.00
Circle Points -> 1133665.00 | Square Points -> 1443817.00
Circle Points -> 1465840.00 | Square Points -> 1867165.00

Estimated value of PI using Monte Carlo Simulation Technique: 3.140247

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q7_MonteCarlo_EstimatePI
 10000

Circle Points -> 20133963.00 | Square Points -> 25576258.00
Circle Points -> 63265737.00 | Square Points -> 80399952.00
Circle Points -> 110831070.00 | Square Points -> 140902640.00
Circle Points -> 146682042.00 | Square Points -> 186550490.00

Estimated value of PI using Monte Carlo Simulation Technique: 3.145144

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$
```

Explanation:

Here we have used 4 threads, for generating circle_points and square_points in their limit **{low, high},** and the rest is the implementation of Monte Carlo algorithm to Estimate PI.

The Algorithm

1. Initialize circle_points, square_points and interval to 0.
2. Generate random points x.
3. Generate random points y.
4. Calculate origin_dist = x*x + y*y.
5. If origin_dist <= 1, increment circle_points.
6. Increment square_points.
7. Increment interval.
8. If increment < NO_OF_ITERATIONS, repeat from 2.

After completion of all the threads we calculate
pi = 4*(circle_points/square_points) and TERMINATE.

Optional:

(8) Computation of a Matrix Inverse using Determinant, Cofactor threads, etc.

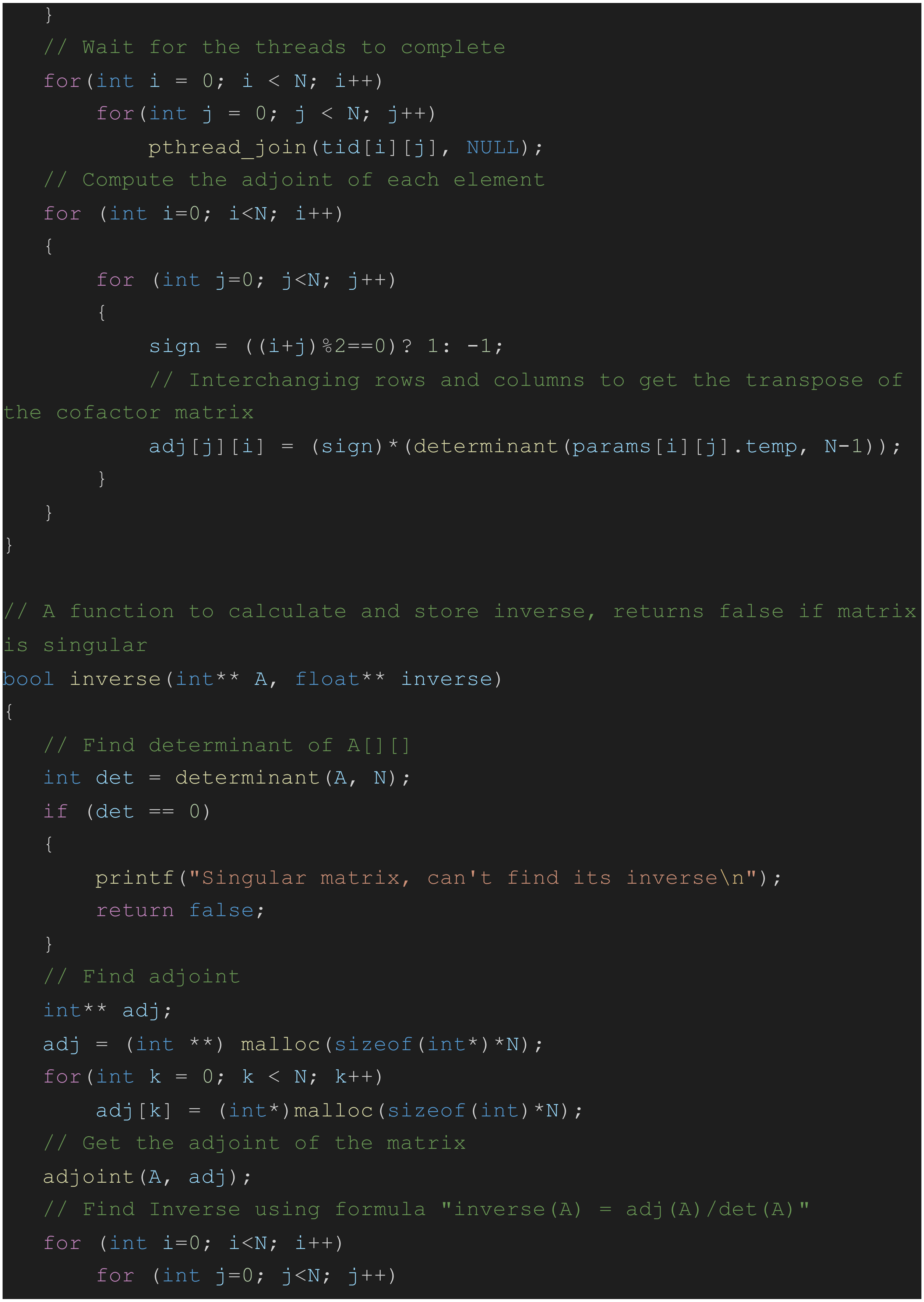
Filename: Q8_MatrixInverse.c

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <pthread.h>

int N = 2;

// Struct for the data
struct data{
   int** arr;
```

```
    int** temp;
    int p;
    int q;
    int n;
};

struct data initData(int** A, int i, int j, int N);
void* getCofactor(void* params);
int determinant(int** A, int n);
void adjoint(int** A,int** adj);
bool inverse(int** A, float** inverse);
void display(float** A);

// Main driver function
int main()
{
    // Scan the size of the matrix
    printf("\nEnter the size of the matrix: ");
    scanf("%d", &N);

    int** A;
    A = (int **) malloc(sizeof(int*)*N);
    for(int k = 0; k < N; k++)
        A[k] = (int*)malloc(sizeof(int)*N);

    // Scan the matrix
    printf("\nEnter the entries for Matrix: ");
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            scanf("%d", &A[i][j]);
        }
    }

    printf("\nMatrix: \n");
    for(int i=0; i<N; i++)
    {
        for(int j=0; j<N; j++)
        {
            printf(" %5d", A[i][j]);
        }
        printf("\n");
    }

    float** inv;
    inv = (float **) malloc(sizeof(float*)*N);
    for(int k = 0; k < N; k++)
        inv[k] = (float*)malloc(sizeof(float)*N);

    // If the inverse exists
    printf("\nInverse of given matrix: \n");
    if (inverse(A, inv))
        // Print the inverse
        display(inv);

    printf("\n");
    return 0;
}

// A function to initialize the structure
struct data initData(int** A, int i, int j, int N)
{
    struct data params;
    params.arr = (int **) malloc(sizeof(int*)*N);
    for(int k = 0; k < N; k++)
        params.arr[k] = (int*)malloc(sizeof(int)*N);
    params.arr = A;
    params.temp = (int **) malloc(sizeof(int*)*N);
    for(int k = 0; k < N; k++)
        params.temp[k] = (int*)malloc(sizeof(int)*N);
    params.p = i;
    params.q = j;
    params.n = N;
    return params;
}

// A function to get the cofactor of A[p][q]
```

```c
void* getCofactor(void* params)
{
   struct data* temp = (struct data* )params;
   int i = 0, j = 0;

   // Looping for each element of the matrix
   for (int row = 0; row < temp->n; row++)
   {
       for (int col = 0; col < temp->n; col++)
       {
           // Copying into temporary matrix
           if (row != temp->p && col != temp->q)
           {
               temp->temp[i][j++] = temp->arr[row][col];
               // Row is filled, so increase row index and reset col index
               if (j == temp->n - 1)
               {
                   j = 0;
                   i++;
               }
           }
       }
   }

   pthread_exit(0); //Exit the thread
}

// A function to return the determinant of the matrix
int determinant(int** A, int n)
{
   int D = 0;
   //If matrix contains single element
   if (n == 1)
       return A[0][0];
   int sign = 1;
   // Create threads for finding cofactor of each element
   pthread_t tid[n];
   struct data params[n];
```

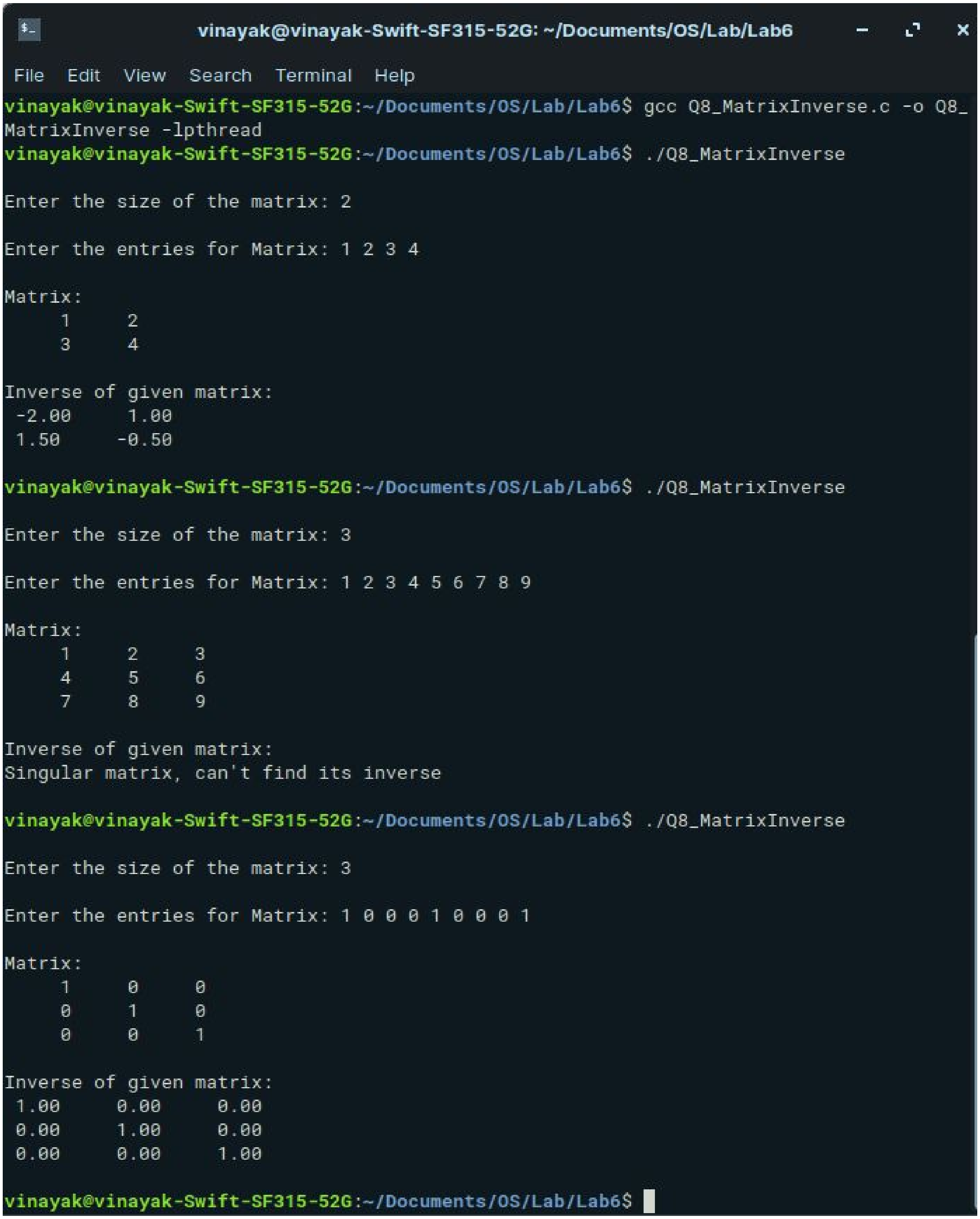
```
    for(int i = 0; i < n; i++)
    {
        params[i] = initData(A, 0, i, n);
        pthread_create(&tid[i], NULL, getCofactor, &params[i]);
    }
    // Wait for the threads to complete
    for(int i = 0; i < n; i++)
        pthread_join(tid[i], NULL);
    // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        D += sign * A[0][f] * determinant(params[f].temp, n - 1);
        // Terms are to be added with alternate sign
        sign = -sign;
    }
    return D;
}

// A function to get adjoint of the given matrix
void adjoint(int** A,int** adj)
{
    if (N == 1)
    {
        adj[0][0] = 1;
        return;
    }
    int sign = 1;
    // Create threads for finding the cofactors
    pthread_t tid[N][N];
    struct data params[N][N];
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N; j++)
        {
            params[i][j] = initData(A, i, j, N);
            pthread_create(&tid[i][j], NULL, getCofactor,
&params[i][j]);
        }
```



```
    }
    // Wait for the threads to complete
    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            pthread_join(tid[i][j], NULL);
    // Compute the adjoint of each element
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            sign = ((i+j)%2==0)? 1: -1;
            // Interchanging rows and columns to get the transpose of
the cofactor matrix
            adj[j][i] = (sign)*(determinant(params[i][j].temp, N-1));
        }
    }
}

// A function to calculate and store inverse, returns false if matrix
is singular
bool inverse(int** A, float** inverse)
{
    // Find determinant of A[][]
    int det = determinant(A, N);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse\n");
        return false;
    }
    // Find adjoint
    int** adj;
    adj = (int **) malloc(sizeof(int*)*N);
    for(int k = 0; k < N; k++)
        adj[k] = (int*)malloc(sizeof(int)*N);
    // Get the adjoint of the matrix
    adjoint(A, adj);
    // Find Inverse using formula "inverse(A) = adj(A)/det(A)"
    for (int i=0; i<N; i++)
        for (int j=0; j<N; j++)
```

```
           inverse[i][j] = adj[i][j]/(float)det;
   return true;
}

// A function to display the matrix.
void display(float** A)
{
   for(int i=0; i<N; i++)
       {
           for(int j=0; j<N; j++)
           {
               printf(" %.2f    ", A[i][j]);
           }
           printf("\n");
       }
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q8_MatrixInverse.c -o Q8_
MatrixInverse -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q8_MatrixInverse

Enter the size of the matrix: 2

Enter the entries for Matrix: 1 2 3 4

Matrix:
     1     2
     3     4

Inverse of given matrix:
 -2.00     1.00
 1.50     -0.50

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q8_MatrixInverse

Enter the size of the matrix: 3

Enter the entries for Matrix: 1 2 3 4 5 6 7 8 9

Matrix:
     1     2     3
     4     5     6
     7     8     9

Inverse of given matrix:
Singular matrix, can't find its inverse

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q8_MatrixInverse

Enter the size of the matrix: 3

Enter the entries for Matrix: 1 0 0 0 1 0 0 0 1

Matrix:
     1     0     0
     0     1     0
     0     0     1

Inverse of given matrix:
 1.00     0.00     0.00
 0.00     1.00     0.00
 0.00     0.00     1.00

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

Explanation:

Here we have used threads to find the cofactors of a given matrix.

Steps to find inverse of a matrix:

Step 1: Calculate the cofactors of the given matrix.
Step 2: Take the transpose of the matrix created from cofactors which we call as adjoint matrix.
Step 3: Calculate the determinant of the given matrix.
Step 4: Inverse of matrix = Adjoint matrix / determinant.

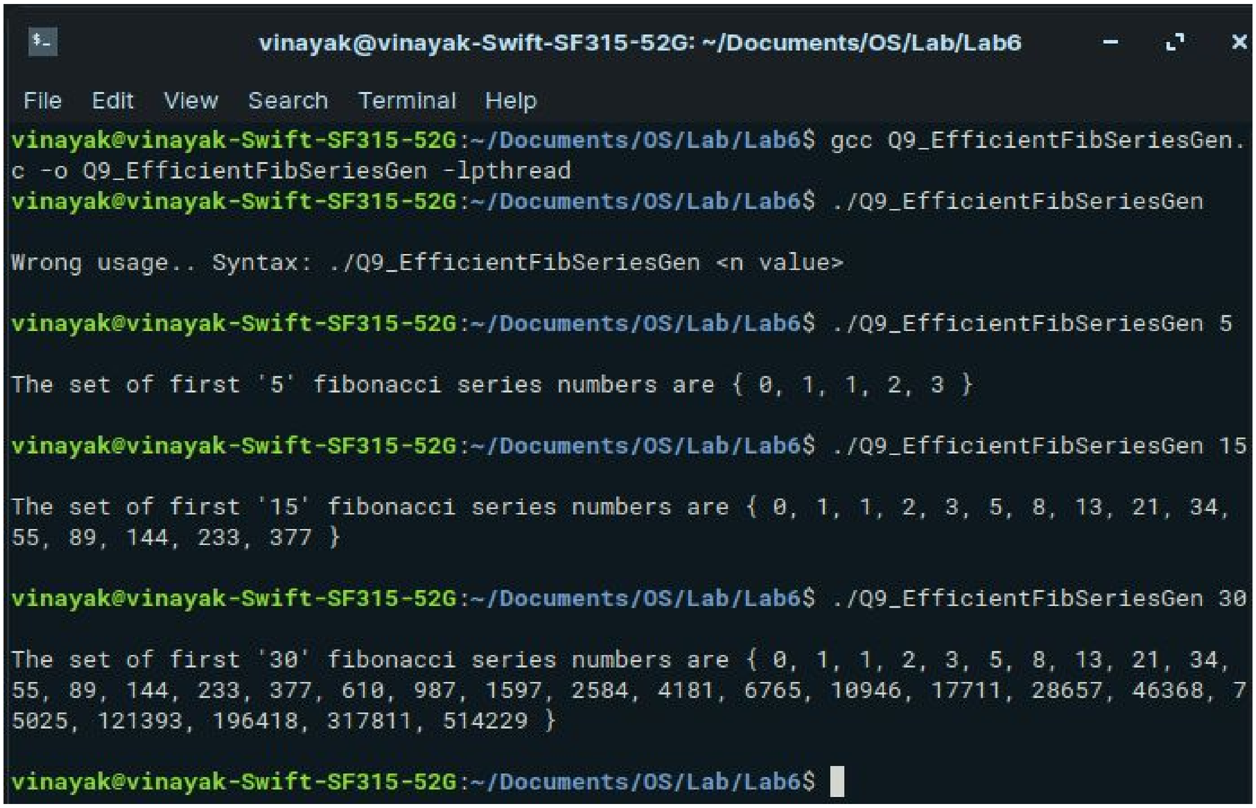
Rest things can be understood using comments in the code.

(9) Read upon efficient ways of parallelizing the generation of Fibonacci series and apply the logic in a multithreaded fashion to contribute a faster version of fib series generation.

Filename: Q9_EfficientFibSeriesGen.c

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>

int *fibseq, i;

void *runner(void *arg);

int main(int argc, char *argv[])
{
   //proper usage of command line arguments
   if(argc != 2)
   {
       printf("\nWrong usage.. Syntax: %s <n value> \n\n", argv[0]);
       exit(EXIT_FAILURE);
   }

   if(atoi(argv[1]) < 0)
   {
       printf("\nn value should be greater than zero\n\n");
       exit(EXIT_SUCCESS);
   }

   int n = atoi(argv[1]); //number of elements to generate
   fibseq = (int *)malloc(n*sizeof(int)); //array to store fibonacci series

   pthread_t *tid = (pthread_t *)malloc(n*sizeof(pthread_t)); //returns the thread id of thread created
   pthread_attr_t attr; //to define thread attributes
   pthread_attr_init(&attr); //initializes the thread attributes

   for(i=0; i<n; i++)
   {
       pthread_create(&tid[i], &attr, runner, (void *) fibseq); //creates a new thread
       pthread_join(tid[i], NULL); //wait for termination of the thread
   }

   printf("\nThe set of first '%d' fibonacci series numbers are { ", n);

   for(i=0; i<n; i++)
   {
       printf("%d, ", fibseq[i]);
   }
   printf("\b\b }\n\n");

   return 0;
}

void *runner(void *arg)
{
   if(i == 0 || i == 1)
   {
       fibseq[i] = i;
       pthread_exit(0); //to exit a thread
   }

   else
   {
       fibseq[i] = fibseq[i-1] + fibseq[i-2];
       pthread_exit(0);
   }
}
```

Output:


vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q9_EfficientFibSeriesGen.
c -o Q9_EfficientFibSeriesGen -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q9_EfficientFibSeriesGen

Wrong usage.. Syntax: ./Q9_EfficientFibSeriesGen <n value>

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q9_EfficientFibSeriesGen 5

The set of first '5' fibonacci series numbers are { 0, 1, 1, 2, 3 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q9_EfficientFibSeriesGen 15

The set of first '15' fibonacci series numbers are { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
55, 89, 144, 233, 377 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q9_EfficientFibSeriesGen 30

The set of first '30' fibonacci series numbers are { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34,
55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 7
5025, 121393, 196418, 317811, 514229 }

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

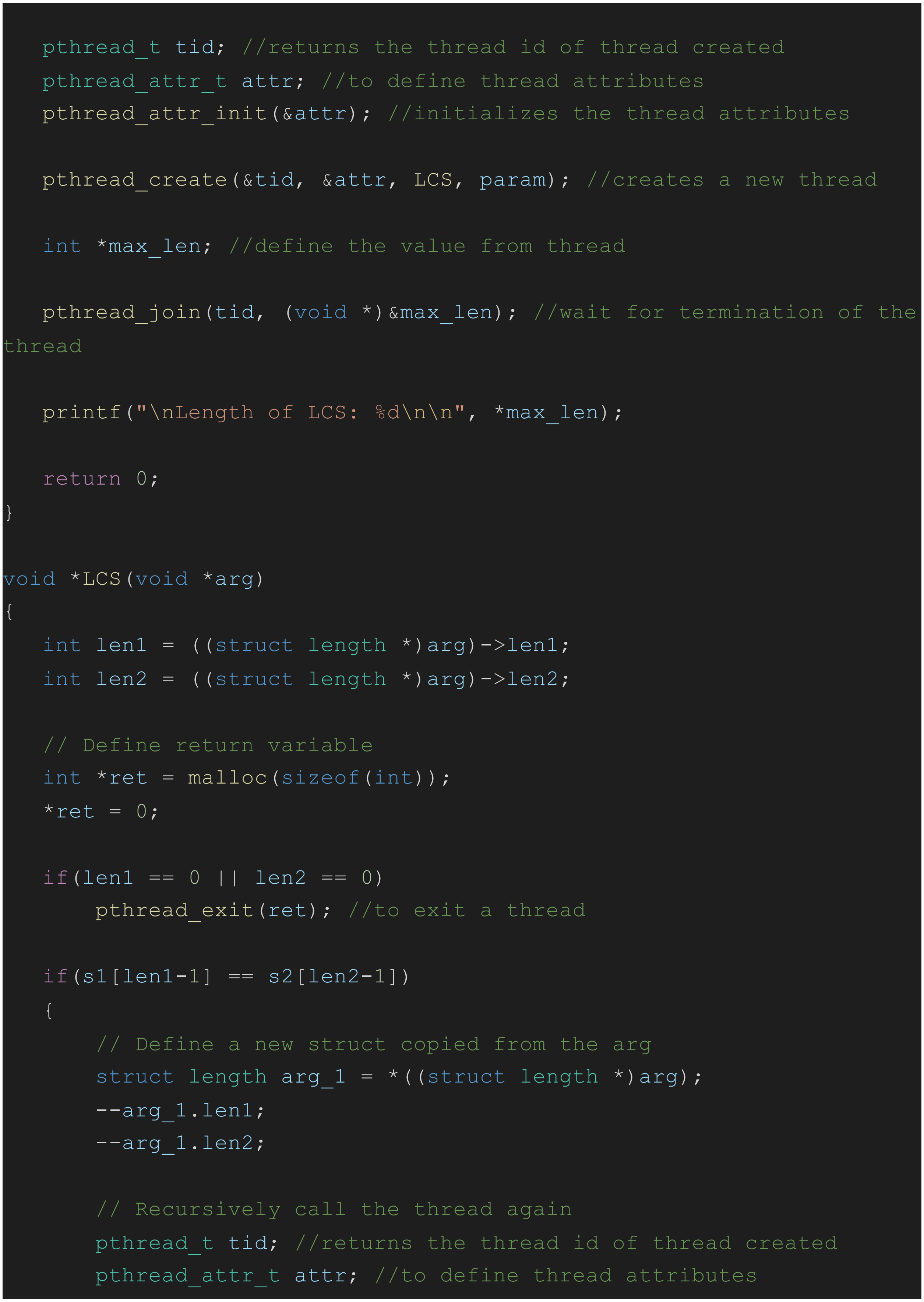
Explanation:

Here we have used one thread per each fibonacci number to provide parallelism and each fibonacci number is stored in the fibseq[] array.
We have tried to implement the recursive function of fibonacci sequence generation using threading.

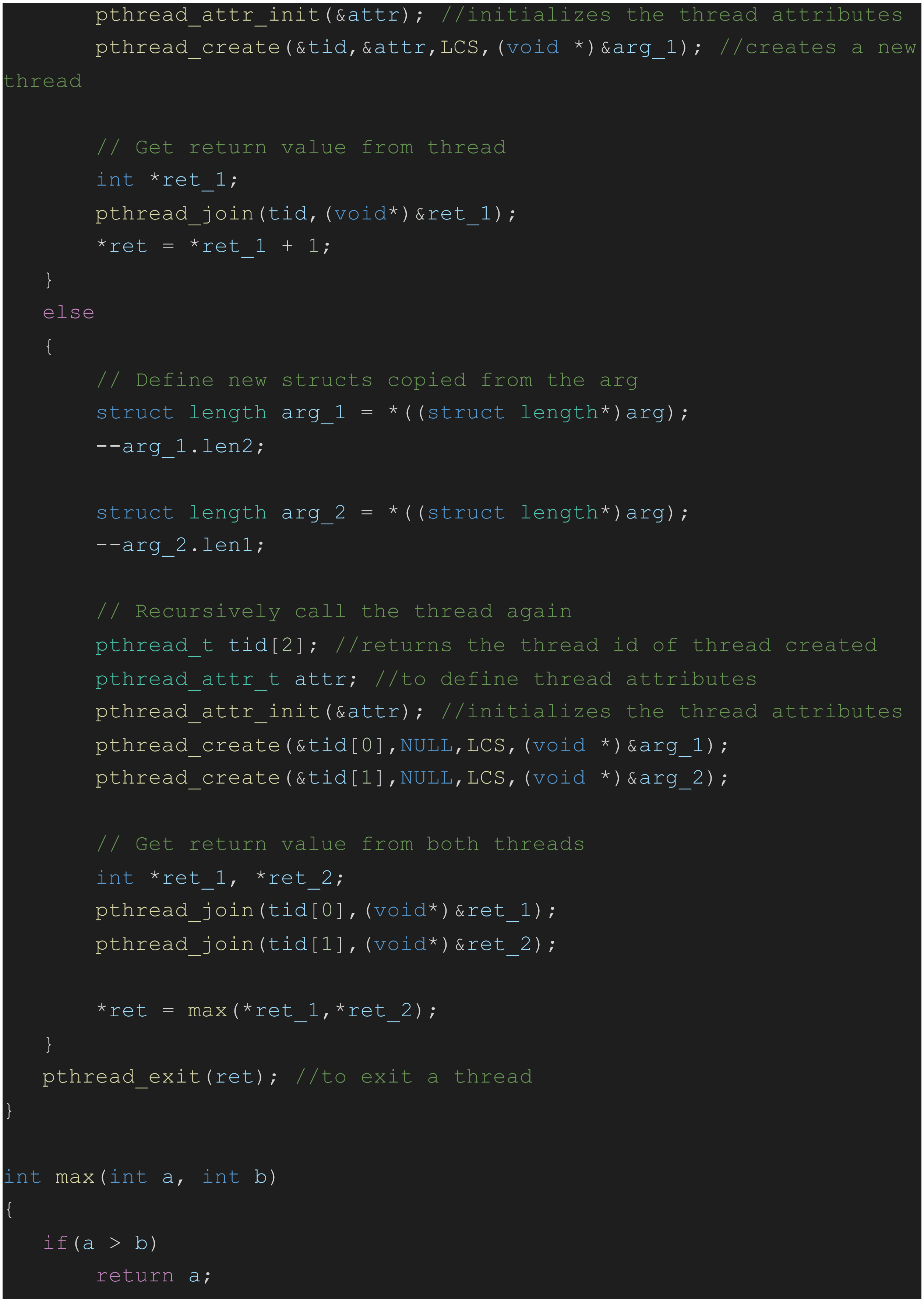
(10) Longest common subsequence generation problem using threads.

Filename: Q10_LongestCommonSubsequence.c

```c
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<pthread.h>

#define MAX 1000

struct length
{
   int len1;
   int len2;
};

char s1[MAX], s2[MAX];

void *LCS(void *arg);
int max(int a, int b);

int main()
{
   printf("\nEnter the string 1: ");
   scanf("%s", s1);

   printf("\nEnter the string 2: ");
   scanf("%s", s2);

   struct length *param = (struct length *)malloc(sizeof(struct
length));

   param->len1 = strlen(s1);  //length of string 1
   param->len2 = strlen(s2);  //length of string 2
```

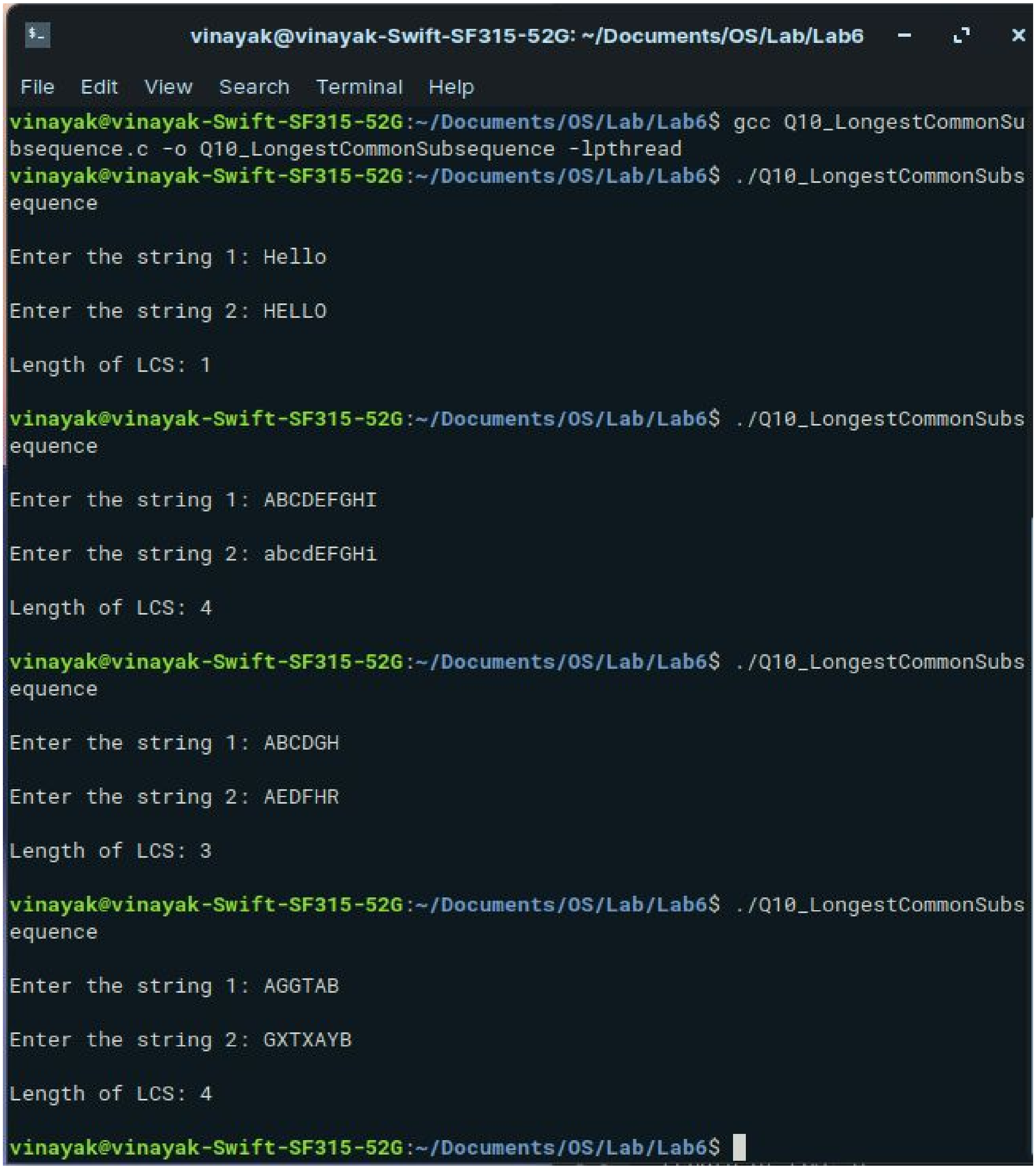


```
    pthread_t tid; //returns the thread id of thread created
    pthread_attr_t attr; //to define thread attributes
    pthread_attr_init(&attr); //initializes the thread attributes

    pthread_create(&tid, &attr, LCS, param); //creates a new thread

    int *max_len; //define the value from thread

    pthread_join(tid, (void *)&max_len); //wait for termination of the
thread

    printf("\nLength of LCS: %d\n\n", *max_len);

    return 0;
}

void *LCS(void *arg)
{
    int len1 = ((struct length *)arg)->len1;
    int len2 = ((struct length *)arg)->len2;

    // Define return variable
    int *ret = malloc(sizeof(int));
    *ret = 0;

    if(len1 == 0 || len2 == 0)
        pthread_exit(ret); //to exit a thread

    if(s1[len1-1] == s2[len2-1])
    {
        // Define a new struct copied from the arg
        struct length arg_1 = *((struct length *)arg);
        --arg_1.len1;
        --arg_1.len2;

        // Recursively call the thread again
        pthread_t tid; //returns the thread id of thread created
        pthread_attr_t attr; //to define thread attributes
```



```
        pthread_attr_init(&attr); //initializes the thread attributes
        pthread_create(&tid,&attr,LCS,(void *)&arg_1); //creates a new thread

        // Get return value from thread
        int *ret_1;
        pthread_join(tid,(void*)&ret_1);
        *ret = *ret_1 + 1;
    }
    else
    {
        // Define new structs copied from the arg
        struct length arg_1 = *((struct length*)arg);
        --arg_1.len2;

        struct length arg_2 = *((struct length*)arg);
        --arg_2.len1;

        // Recursively call the thread again
        pthread_t tid[2]; //returns the thread id of thread created
        pthread_attr_t attr; //to define thread attributes
        pthread_attr_init(&attr); //initializes the thread attributes
        pthread_create(&tid[0],NULL,LCS,(void *)&arg_1);
        pthread_create(&tid[1],NULL,LCS,(void *)&arg_2);

        // Get return value from both threads
        int *ret_1, *ret_2;
        pthread_join(tid[0],(void*)&ret_1);
        pthread_join(tid[1],(void*)&ret_2);

        *ret = max(*ret_1,*ret_2);
    }
    pthread_exit(ret); //to exit a thread
}

int max(int a, int b)
{
    if(a > b)
        return a;
    return b;
}
```

Output:



vinayak@vinayak-Swift-SF315-52G: ~/Documents/OS/Lab/Lab6
File Edit View Search Terminal Help

```
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ gcc Q10_LongestCommonSu
bsequence.c -o Q10_LongestCommonSubsequence -lpthread
vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q10_LongestCommonSubs
equence

Enter the string 1: Hello

Enter the string 2: HELLO

Length of LCS: 1

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q10_LongestCommonSubs
equence

Enter the string 1: ABCDEFGHI

Enter the string 2: abcdEFGHi

Length of LCS: 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q10_LongestCommonSubs
equence

Enter the string 1: ABCDGH

Enter the string 2: AEDFHR

Length of LCS: 3

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ ./Q10_LongestCommonSubs
equence

Enter the string 1: AGGTAB

Enter the string 2: GXTXAYB

Length of LCS: 4

vinayak@vinayak-Swift-SF315-52G:~/Documents/OS/Lab/Lab6$ 
```

Explanation:

We have tried to implement the recursive method to find the Longest Common Subsequence(LCS) using threads.
The logic to implement Recursive LCS is:
if there are no more characters in either of the string, **return 0**.
else if the current characters of both the strings are equal
return 1 + (call for the next characters in both the strings)
else (2 recursive calls will be made)

1. check the current character of string S1 with the next character of string S2
2. check the current character of string S1 with the next character of string S2 and add the maximum value of both the calls.