Operating System

COM301P

Programming Assignment

Lab - 5

(Multithreading)

By :

Sai Kaushik S

CED18I044

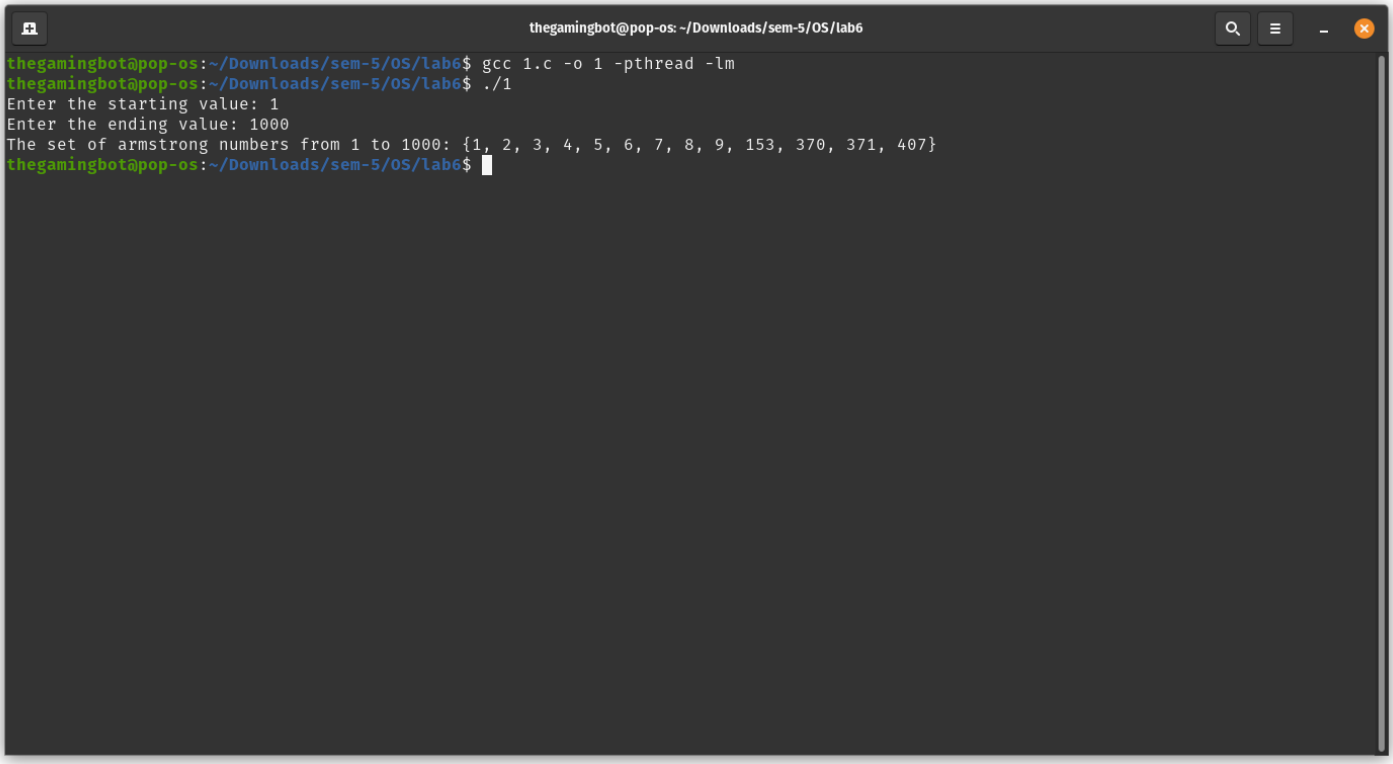
Question 1: Generate Armstrong number generation within a range.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>

#define MAX 1024

int low, high;
int sums[MAX];

// Function definitions
void* runner( void* params );

// Main driver function
int main(){
   // Scan the starting and ending point of the range
   printf("Enter the starting value: ");
   scanf("%d", &low);
   printf("Enter the ending value: ");
   scanf("%d", &high);

   // Create a thread
   for(int i = low; i <= high; i++){
       pthread_t tid;
       pthread_create(&tid, NULL, runner, &i);
       pthread_join(tid, NULL);
   }

   // Print the output
   printf("The set of armstrong numbers from %d to %d: {", low, high);
   for(int i = 0; i <= high - low; i++)
       // If the sum equals the number
       if(sums[i] == i + low)
           // Print the number
           printf("%d, ", i + 1);

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

// Runner function for the pthread
void* runner(void* params){
   // Get the number
   int* val = (int*) params;
   int temp = *val;
```

```c
    int i = 0;
    int sum = 0;

    // Count the number of digits in the digits
    while(temp){
        i++;
        temp = temp/10;
    }

    temp = *val;
    // Get the sum of the power of the digits
    while(temp){
        sum += pow( temp%10, i );
        temp = temp/10;
    }
    // Store it in the array
    sums[*val - low] = sum;
    // Exit the pthread
    pthread_exit(0);
}
```

Output :



Question 2: Ascending Order sort and Descending order sort.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <stdbool.h>

// Struct for the data
struct variables{
   int* a;
   int n;
};

// Function definitions
void* runner (void* args);
void bubbleSort(int* arr, int n, bool fun(const void*, const void*));
void print(int* arr, int n);
bool asc(const void* a, const void* b);
bool desc(const void* a, const void* b);
void swap(int* a, int* b);

/// Main driver function
int main(){
   struct variables data[2];
   // Scan the number of elements
   printf("Enter the number of elements: ");
   scanf("%d", &data[0].n);
   data[1].n = data[0].n;
   data[0].a = malloc(data[0].n * sizeof(int));
   data[1].a = malloc(data[1].n * sizeof(int));
   // Scan the numbers
   for (int i = 0; i < data[0].n; i++){
       printf("Enter the data at index %d: ", i + 1);
       scanf("%d", &data[0].a[i]);
       data[1].a[i] = data[0].a[i];
   }
   // Create a thread for the ascending sort
   pthread_t tid;
   pthread_create(&tid, NULL, runner, &data[0]);

   // Print the descending sort in the main thread
   printf("\nDescending sort: ");
   bubbleSort(data[1].a, data[1].n, desc);
   print(data[1].a, data[1].n);

   pthread_join(tid, NULL);
   return 0;
```

```c
}

// Runner function for the pthresad
void* runner (void* args){
   struct variables* data = (struct variables *) args;
   printf("\nAscending sort: ");
   // Print the ascending sort
   bubbleSort(data->a, data->n, asc);
   print(data->a, data->n);
   // Exit the threads
   pthread_exit(0);
}

// A function for bubble sorting the array
void bubbleSort(int* arr, int n, bool fun(const void*, const void*)){
   // Loop through the array
   for (int i = 0; i < n - 1; i++)
       for (int j = 0; j < n - i - 1; j++)
           // Call the function, if it returns true
           if (fun(&arr[j], &arr[j+1]))
               // Swap the numbers
               swap(&arr[j], &arr[j+1]);
}

// A function to print the array
void print(int* arr, int n){
   // Loop through the array
   for(int i = 0; i < n; i++)
       printf("%d ", arr[i]);
   printf("\n");
}

// A function for ascending sort
bool asc(const void* a, const void* b){
   return *(int*)a > *(int*)b;
}

// A function for descending sort
bool desc(const void* a, const void* b){
   return *(int*)a < *(int*)b;
}

// A function to swap two variables
void swap(int* a, int* b){
   int c = *a;
   *a = *b;
   *b = c;
}
```

Output :


thegamingbot@pop-os: ~/Downloads/sem-5/OS/lab6
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ gcc 2.c -o 2 -pthread
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./2
Enter the number of elements: 4
Enter the data at index 1: 8
Enter the data at index 2: 4
Enter the data at index 3: 2
Enter the data at index 4: 10
Descending sort: 10 8 4 2
Ascending sort: 2 4 8 10
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$

Question 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 occurrences (duplicated elements search as well)

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Struct for the data
struct data{
   int* a;
   int n;
   int idx;
   int x;
};

// Function definitions
void* runner(void* params);
int binarysearch(int *a, int n, int x);
void bubbleSort(int *arr, int n);
void swap(int *a, int *b);
void print(int *arr, int n);

/// Main driver function
int main(){
   struct data argv;
   // Get the number of elements
   printf("Enter the number elements: ");
   fflush(stdin);
   scanf("%d", &argv.n);
   argv.a = (int *) malloc(sizeof(int) * argv.n);
   // Get the elements of the array
   for (int i = 0; i < argv.n; i++){
       printf("Enter number %d: ", i + 1);
       fflush(stdin);
       scanf("%d", &argv.a[i]);
   }
   // Get the search element
   printf("\nEnter the search key: ");
   fflush(stdin);
   scanf("%d", &argv.x);
   // Sort the array
   bubbleSort(argv.a, argv.n);
   printf("\nThe sorted array\n");
   fflush(stdin);
```

```
    print(argv.a, argv.n);
    // Binary search the array for the search element
    argv.idx = binarysearch(argv.a, argv.n, argv.x);
    // If the return value is -1
    if(argv.idx == -1){
        // Exit the program
        printf("\n%d not found in the array.\n", argv.x);
        exit(0);
    }
    printf("\n%d found at index %d.\n", argv.x, argv.idx);
    // Create a new thread to search the left part
    pthread_t tid;
    pthread_create(&tid, NULL, runner, &argv);
    pthread_join(tid, NULL);
    // Search the right part in the main thread
    int i = argv.idx + 1;
    while (argv.a[i] == argv.x && i < argv.n){
        printf("%d found at index %d.\n", argv.x, i);
        i++;
    }
}

// Runner function for the thread
void* runner(void* params){
    // Search the left part of the array
    struct data* argv = (struct data *) params;
    int i = argv->idx - 1;
    while (argv->a[i] == argv->x && i > -1){
        printf("%d found at index %d.\n", argv->x, i);
        i--;
    }
    // Exit the thread
    pthread_exit(NULL);
}

// A function to implement binary search
int binarysearch(int *a, int n, int x){
    int l = 0, r = n - 1;
    while (l <= r){
        // Check the mid of the range
        int m = l + (r - l) / 2;
        if (a[m] == x)
            return m;
        // If the mid is less than search element
        else if (a[m] < x)
            // Update the start of the range
            l = m + 1;
        // If the mig is greater than search element
```

```
        else
            //Update the end of the range
            r = m - 1;
    }
    // If element not found
    return -1;
}

// A function to implement bubble sort
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]);
}

// A function to swap the elements
void swap(int *a, int *b){
    int c = *a;
    *a = *b;
    *b = c;
}

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

Output :


thegamingbot@pop-os: ~/Downloads/sem-5/OS/lab6
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ gcc 3.c -o 3 -pthread
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./3
Enter the number elements: 7
Enter number 1: 2
Enter number 2: 6
Enter number 3: 4
Enter number 4: 8
Enter number 5: 3
Enter number 6: 6
Enter number 7: 6
Enter the search key: 6
The sorted array
Index 0: 2
Index 1: 3
Index 2: 4
Index 3: 6
Index 4: 6
Index 5: 6
Index 6: 8
6 found at index 3.
6 found at index 4.
6 found at index 5.
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$

Question 4: Generation of Prime Numbers upto a limit supplied as Command Line Parameter.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <math.h>

#define MAX_THREADS 5

// Struct for the data
struct range
{
   int low;
   int high;
};

// Function definitions
void* prime(void* params);

// Main driver function
int main(int argc , char *argv[])
{
   // Usage: ./<outout_binary> <a_number>
   if(argc != 2){
       printf("Usage %s [number] \n", argv[0]);
       exit(0);
   }
   // Convert the string to int
   int n = atoi(argv[1]);
   struct range lh[MAX_THREADS];
   // Split the given int into MAX_THREADS pieces
   for (int i = 0; i < MAX_THREADS; i++){
       lh[i].low = i*(n/MAX_THREADS);
       lh[i].high = (i+1)*(n/MAX_THREADS);
   }
   printf("The set of %d prime numbers are { ", n);
   pthread_t tid[MAX_THREADS];
   // Create MAX_THREADS number of threads
   for(int i = 0; i < MAX_THREADS; i++)
       pthread_create(&tid[i], NULL, prime, &lh[i]);
   // Wait until all the threads exit
   for(int i = 0; i < MAX_THREADS; i++)
       pthread_join(tid[i], NULL);
   printf("\b\b }\n");
}
```

```c
// A function to generate the prime numbers
void* prime(void* params){
   int flag;
   struct range* lh = (struct range*) params;
   // Loop from the start to end
   for (int i = lh->low; i <= lh->high; i++){
       // 1 and 0 are not primes
       if (i == 1 || i == 0)
           continue;
       flag = 1;
       // Loop till square root of the number
       for (int j = 2; j <= sqrt(i); ++j){
           // If there exits a perfect divisor
           if (i % j == 0){
               flag = 0;
               // Break the loop
               break;
           }
       }
       // If the number is a prime
       if (flag == 1)
           printf("%d, ", i);
   }
   // Exit the thread
   pthread_exit(0);
}
```

Output :


thegamingbot@pop-os: ~/Downloads/sem-5/OS/lab6
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ gcc 4.c -o 4 -pthread -lm
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./4
Usage ./4 [number]
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./4 30
The set of 30 prime numbers are { 7, 11, 2, 3, 5, 13, 17, 19, 23, 29 }
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$

Question 5: Computation of Mean, Median, Mode for an array of integers.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

// Struct for the data
struct block{
   int *val;
   int n;
};

struct mode{
   int a;
   int count;
};

// Function definitions
void* mean(void * p);
void* median(void * p);
void* mode(void * p);
int removeDuplicates(int arr[], int n);
void bubbleSort(void* params);
void print(void* params);
void swap(int* x, int* y);

// Main driver function
int main(){
   struct block arr;
   // Scan the length of the array
   printf("Enter size of array : ");
   scanf("%d", &arr.n);
   // Scan the elements
   arr.val = (int*)malloc(sizeof(int)*arr.n);
   for(int i=0; i<arr.n; i++){
       printf("Enter the number %d: ", i + 1);
       scanf("%d", &arr.val[i]);
   }
   // Sort the array
   bubbleSort(&arr);
   print(&arr);
   pthread_t tid[2];
   // Create a thread for computation of the median
   pthread_create(&tid[0], NULL, median, &arr);
```

```
    // Create a thread for the computation of the mode
    pthread_create(&tid[1], NULL, mode, &arr);
    // Compute the mean in the main thread
    mean(&arr);
    //Wait for the threads to complete
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
}

// A function to compute the mean of the given array
void* mean(void* params){
    struct block *temp = (struct block*) params;
    float mean = 0;
    // Loop through the array
    for(int i=0;i<temp->n;i++)
        // Add the elements
        mean = mean + temp->val[i];
    // Compute the mean
    mean = mean / temp->n;
    printf("\nMean: %f\n", mean);
}

// A function to cimpute the median of a given array
void* median(void* params){
    struct block *temp = (struct block*) params;
    int med;
    // If the length of array is odd
    if(temp->n%2 == 1)
        // Get the middle element
        med = temp->val[(temp->n-1)/2];
    // If the length of array is even
    else
        // Get the average of the two middle elements
        med = (temp->val[temp->n/2] + temp->val[temp->n/2 - 1]) / 2;
    printf("\nMedian: %d\n", med);
    // Exit the thread
    pthread_exit(0);
}

// Get the mode of the given array
void* mode(void* params){
    struct block *temp = (struct block*) params;
    struct block mod;
    mod.n = temp->n;
    mod.val = (int*)malloc(sizeof(int)*mod.n);
    // Duplicate the array
    for (int i = 0; i < temp->n; i++)
        mod.val[i] = temp->val[i];
```

```
    // Remove the duplicate
    mod.n = removeDuplicates(mod.val , mod.n);
    int max = 0;
    int count[mod.n];
    // Inititalize the array
    for (int i = 0; i < mod.n; i++)
        count[i] = 0;
    // Get the frequency of the elements
    for(int i = 0 ; i < mod.n ; i++)
        for (int j = 0 ; j < temp->n ; j++){
            if(mod.val[i] == temp->val[j])
                count[i]++;
            if(count[i] > max)
                max = count[i];
        }
    printf("\nMode : ");
    // Print the elements with highest frequency
    for(int i = 0; i < mod.n ; i++)
        if(count[i] == max)
            printf(" %d " , mod.val[i]);
    printf("\n");
    // Exit the thread
    pthread_exit(0);
}

int removeDuplicates(int arr[], int n){
    // Return, if array is empty or contains a single element
    if (n==0 || n==1)
        return n;
    int temp[n];
    // Start traversing elements
    int j = 0;
    for (int i=0; i<n-1; i++)
        // If current element is not equal to next element then store that current
element
        if (arr[i] != arr[i+1])
            temp[j++] = arr[i];
    // Store the last element as whether it is unique or repeated, it hasn't stored
previously
    temp[j++] = arr[n-1];
    // Modify original array
    for (int i=0; i<j; i++)
        arr[i] = temp[i];
    for (int i = j; i < n; i++)
        arr[i] = 0;
    return j;
}
```

```c
// A function to sort the array
void bubbleSort(void* params){
   struct block *temp = (struct block*) params;
   for (int i = 0; i < temp->n - 1; i++)
       for (int j = 0; j < temp->n - i - 1; j++)
           if (temp->val[j] > temp->val[j+1])
               swap(&temp->val[j], &temp->val[j+1]);
}

// A function to display the array
void print(void* params){
   struct block *temp = (struct block*) params;
   printf("Sorted array : ");
   for(int i = 0; i < temp->n; i++)
       printf("%d ", temp->val[i]);
   printf("\n");
}

// A function to swap two numbers
void swap(int* x, int* y){
   int c = *x;
   *x = *y;
   *y = c;
}
```

Output :

Question 6: Implement Merge Sort and Quick Sort in a multithreaded fashion.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Function definitions
void merge(int* arr, int l, int m, int r);
void* mergeSort(void* lr);
void printArray(int* arr, int n);

int* arr;

// Main driver function
int main(){
   // Scan the length of the array
   int n;
   printf("Enter the number of elements: ");
   scanf("%d", &n);
   // Scan the array
   arr = (int*) malloc(sizeof(int) * n);
   for(int i = 0; i < n; i++){
       printf("Enter number %d: ", i + 1);
       scanf("%d", &arr[i]);
   }
   // Merge sort the array
   int lr[] = {0, n - 1};
   mergeSort(lr);
   // Print the sorted array
   printf("\nSorted array is: ");
   printArray(arr, n);
   return 0;
}

//A function to merge two arrays
void merge(int* arr, int l, int m, int r){
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   int L[n1], R[n2];
   // Copy the array elements to a temp array
   for (i = 0;i < n1;i++)
       L[i] = arr[l + i];
   for (j = 0;j < n2;j++)
```

```
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    // Loop till the overflow
    while (i < n1 && j < n2){
        // If L < R
        if (L[i] <= R[j])
            // Copy L
            arr[k++] = L[i++];
        // If L > R
        else
            // Copy R
            arr[k++] = R[j++];
    }

    // Copy the remaining elements
    while (i < n1)
        arr[k++] = L[i++];
    while (j < n2)
        arr[k++] = R[j++];
}

// A function to merge sort an array
void* mergeSort(void* params){
    int* lr = (int*)params;
    int l = lr[0];
    int r = lr[1];
    // If there exits atleast one element in the range l...r
    if (l < r){
        // Split the array at the mid
        int m = l + (r - l)/ 2;
        int llr[] = {l, m};
        int rlr[] = {m + 1, r};
        // Create a thread to the first half of the array
        pthread_t tid;
        pthread_create(&tid, NULL, mergeSort, llr);
        // Sort the second half in the main thread
        mergeSort(rlr);
        pthread_join(tid, NULL);
        // Merge the two halfs
        merge(arr, l, m, r);
    }
}

//A function to print the array
void printArray(int* arr, int n){
```

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

Output :

Code:

```
// Include the required libraries
#include<stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Function declaration
void swap(int* x, int* y);
void* quickSort(void* params);
int partition (int* arr, int low, int high);
void print(int* arr, int size);

int* arr;

// Main function
int main(){
    int n;
    // Taking the size of array from user
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    // Allocating memory for arr
    arr = (int*) malloc(sizeof(int) * n);
```

```c
    // Taking the input elemets from user
    for(int i = 0; i < n; i++){
        printf("Enter number %d: ", i + 1);
        scanf("%d", &arr[i]);
    }
    // Print the given array
    printf("Given array is: ");
    print(arr, n);
    // Storing the low and high in lr[]
    int lr[] = {0, n - 1};
    quickSort(lr);
    // Printing the sorted array
    printf("\nSorted array is: ");
    print(arr, n);
    return 0;
}

// This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot
int partition (int* arr, int low, int high){
    // Pivot
    int pivot = arr[high];
    // Index of smaller element
    int i = (low - 1);

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

// The main function that implements QuickSort arr[] --> Array to be sorted, low --> Starting index, high --> Ending index
void* quickSort(void* params){
    // Storing the data from params to lr
    int* lr = (int*)params;
    int low = lr[0];
    int high = lr[1];
    if (low < high){
        // pi is partitioning index, arr[p] is now at right place
```

```
        int pi = partition(arr, low, high);
        int llr[] = {low, pi-1};
        int rlr[] = {pi + 1, high};
        pthread_t tid;
        // Separately sort elements before partition and after partition
        // Creating the thread
        pthread_create(&tid, NULL, quickSort, llr);
        // Calling the quickSort function recursively
        quickSort(rlr);
        // Joining the thread
        pthread_join(tid, NULL);
    }
}

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

// Swap two variables
void swap(int* x, int* y){
    int t = *x;
    *x = *y;
    *y = t;
}
```

Output :

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

Code:

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

// Max number of threads
#define MAX_THREADS 4

// Function declaration
void* runner(void* params);
double randomGen();
// Declaring the count globally
long int count = 0;

// Main function
int main(){
   int n;
   // Take input from user for the number of points
   printf("Enter number of points: ");
   scanf("%d", &n);

   pthread_t tid[MAX_THREADS];

   // Creating the thread (MAX_THREADS = 4) to generate n number of points
   // Each thread will generate certain number of points based on the total count of
threads (Here it is 4)
   for(int i = 0; i < MAX_THREADS; i++){
       int lr[] = {i * n / MAX_THREADS, (i + 1) * n / MAX_THREADS};
       pthread_create(&tid[i], NULL, runner, lr);
   }
   // Joining the threads
   for(int i = 0; i < MAX_THREADS; i++)
       pthread_join(tid[i], NULL);
   // Finding the value the pi
   double pi = 4 * (double)count / n;
   printf("The estimated value of pi: %f\n", pi);
}

// Runner function
void* runner(void* params){
   // Storing the data from params to temp
   int* temp = (int*)params;
   // Generating the random coordinates (random points)
   for(int i = temp[0]; i < temp[1]; i++){
```

```
        double x = randomGen();
        double y = randomGen();
        // If the generated coordinates satisfy the equation of the circle so it means
the points life on or within the circle so increment the count
        double r = x * x + y * y;
        if (r <= 1)
            count++;
    }
}

double randomGen(){
    return ((double)rand() / (double)RAND_MAX);
}
```

Output :

Question 8: Computation of a Matrix Inverse using Determinant, Cofactor threads, etc.

Code:

```c
// Include the required libraries
#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("Enter 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
   for(int i = 0; i < N; i++)
       for(int j = 0; j < N; j++){
           printf("Enter number [%d][%d]: ", i, j);
           scanf("%d", &A[i][j]);
       }
   printf("The matrix: \n");
   for (int i=0; i<N; i++){
       for (int j=0; j<N; j++)
           printf("%6d", 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("\nThe Inverse is :\n");
    if (inverse(A, inv))
        // Print the inverse
        display(inv);
    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]
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++;
                }
            }
        }
    }
```

```
    //Exit the thread
    pthread_exit(0);
}

// 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("%.6f ", A[i][j]);
        printf("\n");
    }
}
```

Output :


thegamingbot@pop-os: ~/Downloads/sem-5/OS/lab6
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ gcc 8.c -o 8 -pthread -lm
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./8
Enter the size of the matrix: 3
Enter number [0][0]: 2
Enter number [0][1]: 6
Enter number [0][2]: 3
Enter number [1][0]: 5
Enter number [1][1]: 7
Enter number [1][2]: 4
Enter number [2][0]: 8
Enter number [2][1]: 3
Enter number [2][2]: 8
The matrix:
2 6 3
5 7 4
8 3 8
The Inverse is :
-0.530120 0.469880 -0.036145
0.096386 0.096386 -0.084337
0.493976 -0.506024 0.192771
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$

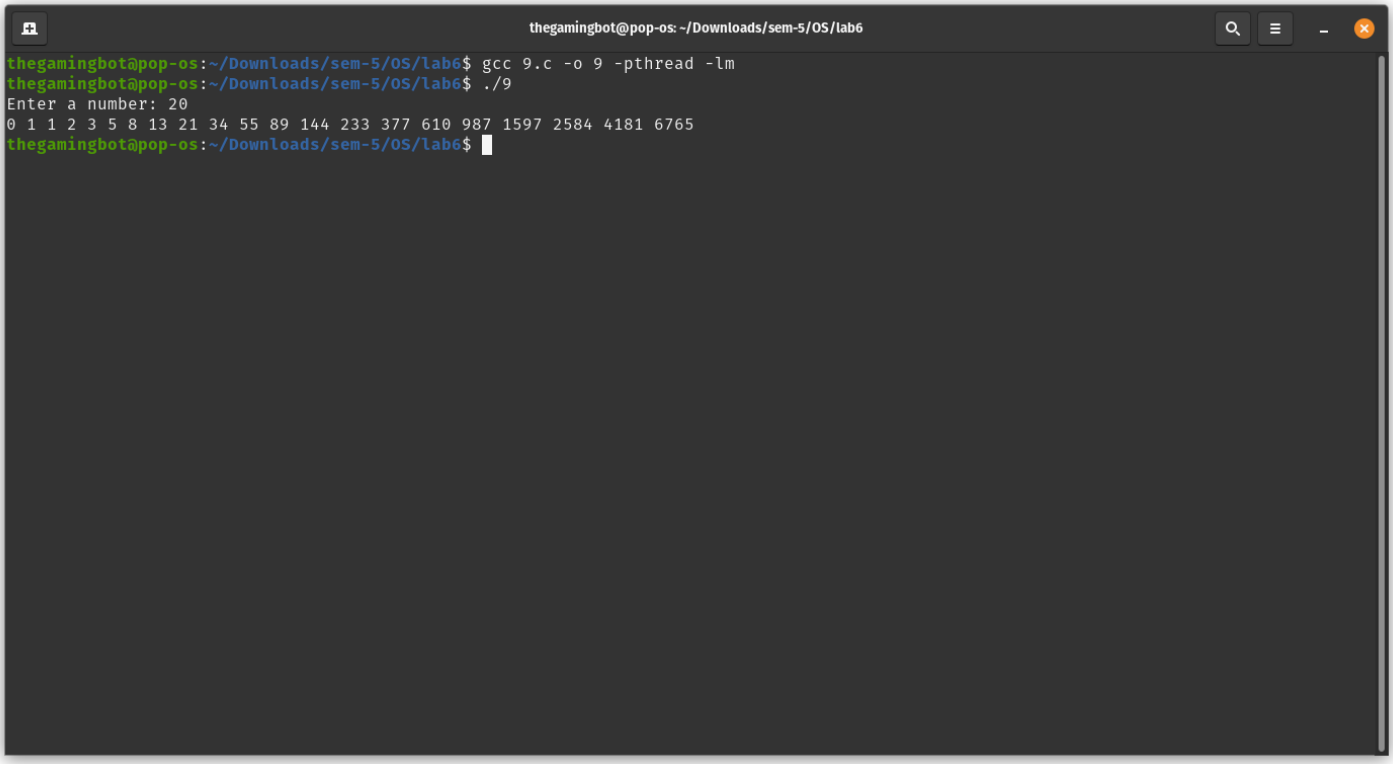
Question 9: Read upon efficient ways of parallelizing the generation of Fibonacci series and apply the logic in a multithreaded fashion to contribute a faster version of fib series generation.

Code:

```c
// Include the required libraries
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>

// Memoization cache
int cache[40];
// Function declaration
void* fib(void* params);

// Main function
int main(){
   int n;
   // Initialization the cache with -1
   memset(cache, -1, sizeof(cache));
   // Take the input from the user
   printf("Enter a number: ");
   scanf("%d", &n);
   // Call the fibonacci function
   fib(&n);
   // Printing the series
   for(int i = 0; i <= n; i++)
       printf("%d ", cache[i]);
   printf("\n");
}

// Fibonacci function
void* fib(void* params){
   // Storing the data from params to n
   int* ptr = (int*) params;
   int n = *ptr;
   // If fib is not yet computed
   if (cache[n] == -1){
       // If the n(number) is less than 1 store that number itself in the cache
       if (n <= 1)
           cache[n] = n;
       // Else find the fib
       else{
           pthread_t tid;
           // Storing the previous in x
           int x = n - 1;
           // Create threads for each of the number
```

```
            pthread_create(&tid, NULL, fib, &x);
            // Joining those threads
            pthread_join(tid, NULL);
            // Again storing the previous value in x
            x = n - 2;
            // Calling the function recursively
            fib(&x);
            cache[n] = cache[n - 1] + cache[n - 2];
        }
    }
}
```

Output :



Question 10: Longest common subsequence generation problem using threads.

Code:

```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: ");
   fgets(s1,MAX ,stdin);
   strcpy(s1, strtok(s1, "\n"));
   printf("\nEnter the string 2: ");
   fgets(s2,MAX ,stdin);
   strcpy(s2, strtok(s2, "\n"));
   struct length *param = (struct length *)malloc(sizeof(struct length));
   //length of string 1
   param->len1 = strlen(s1);
   //length of string 2
   param->len2 = strlen(s2);
   pthread_t tid;
   // Create a thread
   pthread_create(&tid, NULL, LCS, param);
   // Define the value from thread
   int *max_len;
   // Wait for termination of the thread
   pthread_join(tid, (void *)&max_len);
   printf("\nLength of LCS: %d\n\n", *max_len - 1 );
   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)
       // Exit a thread
       pthread_exit(ret);
   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;
        // Create a thread
        pthread_create(&tid,NULL,LCS,(void *)&arg_1);
        // 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];
        // Create thread
        pthread_create(&tid[0],NULL,LCS,(void *)&arg_1);
        pthread_create(&tid[1],NULL,LCS,(void *)&arg_2);

        // Ge return value from both threads
        int *ret_1, *ret_2;
        // Joining the threads
        pthread_join(tid[0],(void*)&ret_1);
        pthread_join(tid[1],(void*)&ret_2);
        *ret = max(*ret_1,*ret_2);
    }
    // Exit the thread
    pthread_exit(ret);
}
int max(int a, int b){
    if(a > b)
        return a;
    return b;
}
```

Output :


thegamingbot@pop-os: ~/Downloads/sem-5/OS/lab6
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ gcc 10.c -o 10 -pthread -lm
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$ ./10
Enter the string 1: hello this is sai
Enter the string 2: this is s
Length of LCS: 8
thegamingbot@pop-os:~/Downloads/sem-5/OS/lab6$