OPERATING SYSTEMS PRACTICE
(ASSIGNMENT 6)

Name – Mridul Harish
Roll number – CED18I034

Question 1 : Generate Armstrong number generation within a range.

Code :-

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

void *generate(void *number);

int main(int argc, char const *argv[])
{
	pthread_t tid;
	pthread_attr_t attr;
	if(argc < 2)
	{
		printf("This is not allowed\n");
		return 0;
	}

	int n = atoi(argv[1]);
	for(int i = 1;i <= n; i = i+1)
	{
		int *number = (int*)malloc(2*sizeof(int));
		int original_number = i; int remainder; int result=0;
		while(original_number != 0)
		{
			remainder = original_number % 10;
			result += remainder * remainder * remainder;
			original_number /= 10;
		}

		number[0] = i;
```

```
                number[1] = result;
                pthread_attr_init(&attr);
                pthread_create(&tid, NULL, generate, (void *)number);
                pthread_join(tid, NULL);
                free(number);
        }
        return 0;
}

void *generate(void *number)
{
        int *parameter = (int *)number;
        if(parameter[0] == parameter[1])
        {
                printf("%d\n", parameter[0]);
        }
        pthread_exit(0);
}
```

Output :-

Question 2 : Ascending Order sort and Descending order sort.

Code :-

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

```
void *generate(void *number);

int main(int argc, char const *argv[])
{
	pthread_t tid;
	pthread_attr_t attr;

	if (argc<2)
	{
		printf("This is not allowed\n");
		return 0;
	}

	int n = argc-1;
	int *number = (int *)malloc((n+1)*sizeof(int));

	for(int i = 1; i < argc; i = i+1)
	{
		number[i] = atoi(argv[i]);
	}

	number[0] = n;
	pthread_attr_init(&attr);
	pthread_create(&tid ,NULL, generate, (void *)number);
	pthread_join(tid,NULL);

	n = n+1;
	printf("Ascending order\n");
	for(int i = 1; i < n; i = i+1)
	{
		for(int j = i+1; j < n; j = j+1 )
		{
			if(number[j] < number[i])
			{
				int t = number[i];
				number[i] = number[j];
				number[j] = t;
			}
		}
```

```
	}

	for(int i = 1; i < n; i = i+1)
	{
		printf("%d ",number[i]);
	}

	printf("\n");

	free(number);
	return 0;
}

void *generate(void *number)
{
	int *parameter = (int *)number;
	int n = parameter[0];
	n = n+1;

	printf("Descending order\n");
	for(int i = 1; i < n; i = i+1)
	{
		for (int j = i+1; j < n; j = j+1 )
		{
			if(parameter[i] < parameter[j])
			{
				int t = parameter[i];
				parameter[i] = parameter[j];
				parameter[j] = t;
			}
		}
	}

	for(int i = 1;i < n;i = i+1)
	{
		printf("%d ", parameter[i]);
	}
	printf("\n");
```

```
        pthread_exit(0);
}
```

Output :-

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 occurrence (duplicated elements search as well).

Code :-

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

int a[100]; int position = -1;

struct data
{
        int begin;
        int end;
        int m;
```

```
};

void *runner(void *param);
void print(int x);

int main(int argc, char *argv[])
{
	if(argc < 4)
	{
		printf("This is not allowed\n");
	}
	else
	{
		int n = argc - 2;
		//printf("n = %d",n);
		for(int j = 0; j < n; j = j+1)
		{
			a[j] = atoi(argv[j+2]);
		}

		for(int k = 0; k < n; k = k+1)
		{
			for(int l = 0; l < n-k-1; l = l+1)
			{
				if(a[l] > a[l+1])
				{
					int temp = a[l];
					a[l] = a[l+1];
					a[l+1] = temp;
				}
			}
		}

		printf("The sorted sequence is :-\n");
		for(int o = 0; o < n; o = o+1)
		{
			printf("%d ", a[o]);
		}
		printf("\n");
```

```
//printf("n = %d\n",n);

struct data param[4];
param[0].begin = 0;
param[0].end = n/4 - 1;
param[0].m = atoi(argv[1]);
param[1].begin = n/4;
param[1].end = n/2 -1;
param[1].m = atoi(argv[1]);
param[2].begin = n/2;
param[2].end = 3*n/4 - 1;
param[2].m = atoi(argv[1]);
param[3].begin = 3*n/4;
param[3].end = n-1;
param[3].m = atoi(argv[1]);

pthread_t tid[4];
pthread_attr_t attr;
pthread_attr_init(&attr);

for(int p = 0; p < 4; p = p+1)
{
	pthread_create(&tid[p], &attr, runner, &param[p]);
}

for(int q = 0; q < 4; q = q+1)
{
	pthread_join(tid[q], NULL);
}

if(position > -1)
{
	print(atoi(argv[1]));
}
else
{
	printf("\n%d not found\n", atoi(argv[1]));
}
```

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

void *runner(void *param)
{
        struct data *dm = param;
        int begin = dm -> begin;
        int end = dm -> end;
        int m = dm -> m;
        //printf("begin = %d, end = %d, m = %d\n",begin, end, m);
        if(begin < end)
        {
                int mid = (begin + end)/2;

                if(a[mid] = m)
                {
                        position = mid;
                }
                else
                {
                        pthread_t tid;
                        pthread_attr_t attr;
                        pthread_attr_init(&attr);

                        struct data param;
                        param.m = m;

                        if(m < a[mid])
                        {
                                param.begin = begin;
                                param.end = mid;
                        }
                        else if(m > a[mid])
                        {
                                param.begin = mid + 1;
                                param.end = end;
                        }
```

```
                pthread_create(&tid, &attr, runner, &param);
                pthread_join(tid, NULL);
        }
    }
    pthread_exit(0);
}

void print(int x)
{
    printf("%d is found at indice = ", x);

    int left_end = 0;
    while(left_end >= 0)
    {
        if(a[position - left_end] == x)
        {
            printf("%d ", position - left_end);
            left_end = -1;
        }
        else
        {
            left_end = left_end - 1;
        }
    }

    int right_end = 1;
    while(right_end >= 1)
    {
        if(a[position + right_end] == x)
        {
            printf("%d ", position - right_end);
            right_end =  -1;
        }
        else
        {
            right_end = right_end - 1;
        }
    }
```

}

Output :-

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

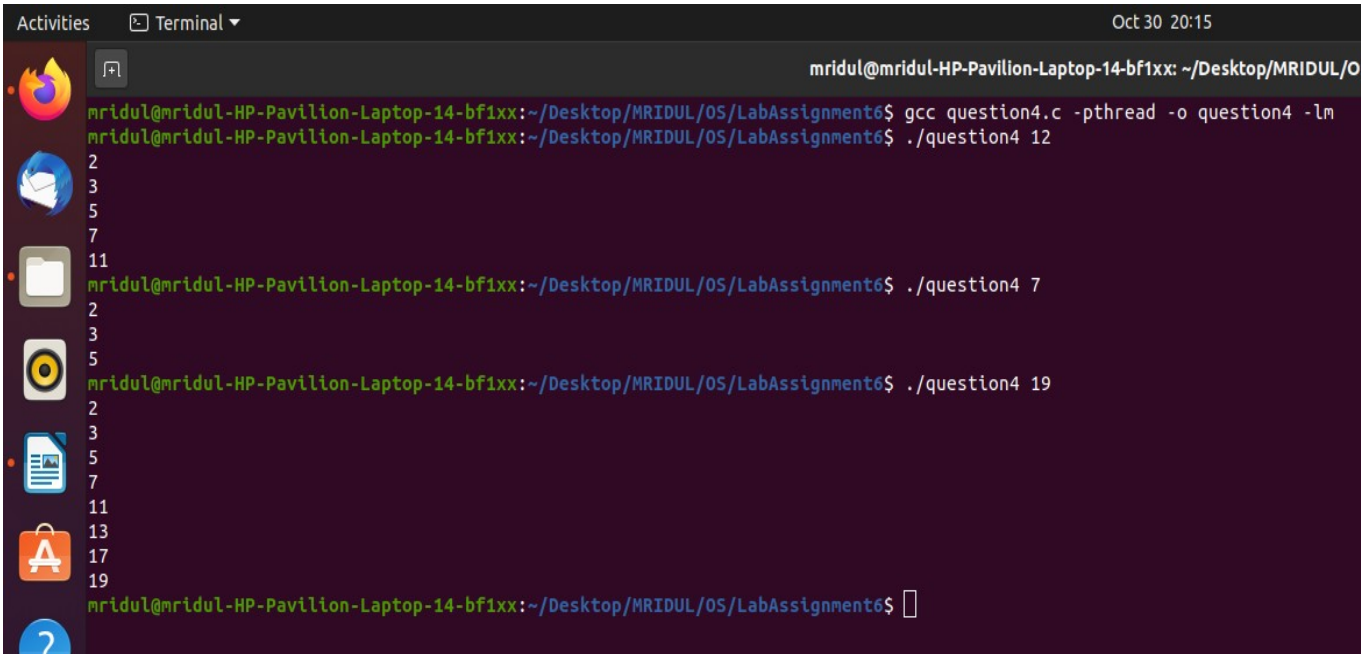
Code :-

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<pthread.h>

void *runner(void *param);

int main(int argc, const char *argv[])
{
	if(argc != 2)
	{
		printf("This is not allowed\n");
	}
	else
	{
```

```
        int count = atoi(argv[1]);
        pthread_t tid[count*2];
        pthread_attr_t attr;
        pthread_attr_init(&attr);

        for(int i = 1; i <= count; i = i+1)
        {
            pthread_create(&tid[i], &attr, runner, &i);
            pthread_join(tid[i], NULL);
        }
    }
    return 0;
}

void *runner(void *param)
{
    int n = *((int *)param);
    int j; int flag = 1;

    if(n <= 1)
    {
        flag = 0;
    }
    else if(n == 2)
    {
        flag = 1;
    }
    else
    {
        for(int j = 2; j <= sqrt(n); j = j+1)
        {
            if(n%j == 0)
            {
                flag = 0;
                break;
            }
        }
    }
```

```
        if(flag == 1)
        {
                printf("%d\n", n);
        }
}
```

Output :-



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

Code :-

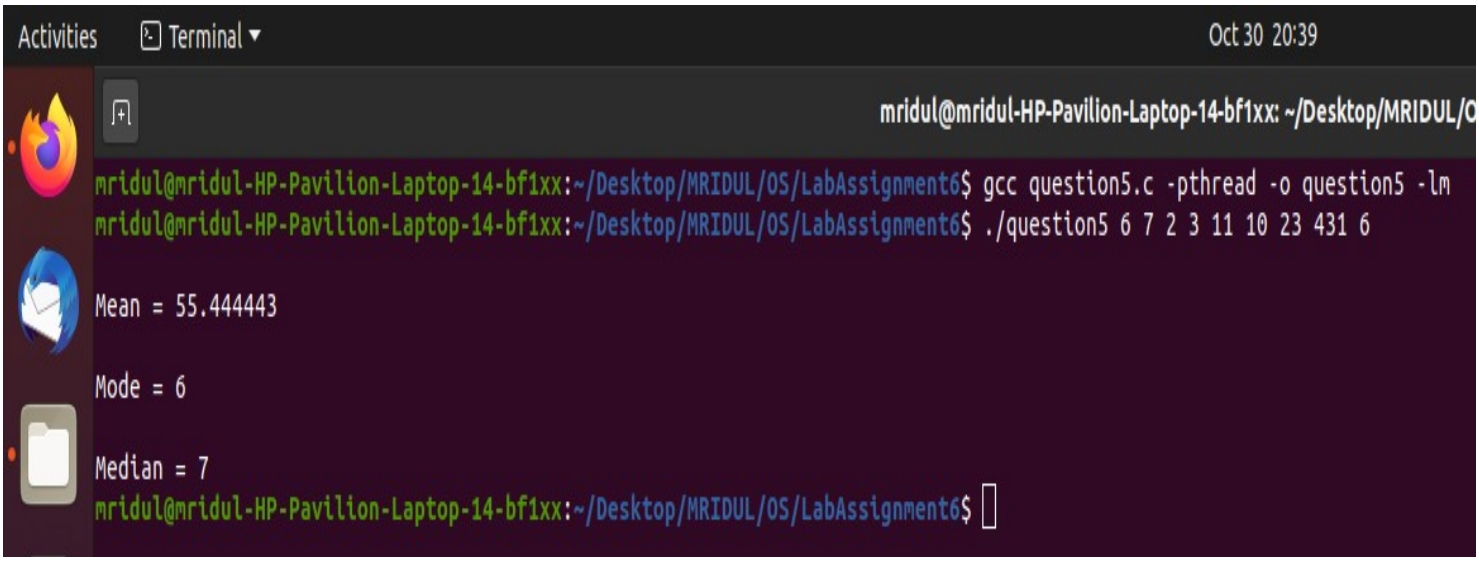
```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<pthread.h>

#define MAX 512
```

```
int a[MAX];
int n;

int cmpfunc(const void *a, const void *b);
void *mean(void *param);
void *median(void *param);
void *mode(void *param);

int main(int argc, char const *argv[])
{
	if(argc < 2)
	{
		printf("This is not allowed\n");
	}

	n = argc - 1;
	for(int i = 0; i < n; i = i+1)
	{
		a[i] = atoi(argv[i+1]);
	}

	pthread_t tid[3];
	pthread_attr_t attr;
	pthread_attr_init(&attr);

	pthread_create(&tid[0], &attr, mean, NULL);
	pthread_create(&tid[1], &attr, median, NULL);
	pthread_create(&tid[2], &attr, mode, NULL);

	pthread_join(tid[0], NULL);
	pthread_join(tid[1], NULL);
	pthread_join(tid[2], NULL);

	return 0;
}

int cmpfunc(const void *a, const void *b)
{
```

```
	return (*(int *)a - *(int *)b);
}

void *mean(void *param)
{
	int temp[n];
	for(int i = 0; i < n; i = i+1)
	{
		temp[i] = a[i];
	}

	float mean;

	for(int i = 0; i < n; i = i+1)
	{
		mean = mean + temp[i];
	}

	mean = mean/n;

	printf("\nMean = %f\n", mean);
	pthread_exit(NULL);
}

void *median(void *param)
{
	int temp[n];
	for(int i = 0; i < n; i = i+1)
	{
		temp[i] = a[i];
	}

	qsort(temp, n, sizeof(int), cmpfunc);

	int median;
	if(n%2 == 1)
	{
		median = temp[(n - 1)/2];
	}
```

```
	else
	{
		median = (temp[n/2] + temp[n/2 - 1])/2;
	}

	printf("\nMedian = %d\n", median);
	pthread_exit(NULL);
}

void *mode(void *param)
{
	int temp[n];
	for(int i = 0; i < n; i = i+1)
	{
		temp[i] = a[i];
	}

	int mode;
	int max_count;
	for(int i = 0; i < n; i = i+1)
	{
		int count = 0;
		for(int j = 0; j < n; j = j+1)
		{
			if(temp[j] == temp[i])
			{
				count = count + 1;
			}
		}
		if(count > mode)
		{
			max_count = count;
			mode = temp[i];
		}
	}

	printf("\nMode = %d\n", mode);
	pthread_exit(NULL);
}
```

Output :-



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

Code :-

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<pthread.h>

#define MAX 512
int q_sort[MAX];
int m_sort[MAX];

struct data
{
        int begin;
        int end;
};
```

```
int partition(int begin, int end);
void *quicksort(void *arg1);
void merge_array(int begin, int mid, int end);
void *mergesort(void *arg2);

int main(int argc, char *argv[])
{
	if(argc < 2)
	{
		printf("This is not allowed\n");
	}
	else
	{
		int n = argc - 1;
		for(int i = 0; i < n; i = i+1)
		{
			q_sort[i] = atoi(argv[i + 1]);
			m_sort[i] = atoi(argv[i + 1]);
		}

		struct data param;
		param.begin = 0;
		param.end = n - 1;

		pthread_t tid[2];
		pthread_attr_t attr;
		pthread_attr_init(&attr);

		pthread_create(&tid[0], &attr, mergesort, &param);
		pthread_create(&tid[1], &attr, quicksort, &param);

		pthread_join(tid[0], NULL);
		pthread_join(tid[1], NULL);

		printf("Sorted array using Merge sort is : ");
		for(int i = 0; i < n; i = i+1)
		{
			printf("%d ", m_sort[i]);
```

```
            }

            printf("\n");

            printf("Sorted array using Quick sort is : ");
            for(int i = 0; i < n; i = i+1)
            {
                  printf("%d ", q_sort[i]);
            }

            printf("\n");
      }
}

int partition(int begin, int end)
{
      int i = begin; int j = end;
      int p = begin;
      int val = p;

      while(i < j)
      {
            while(q_sort[p] >= q_sort[i] && i < end)
            {
                  i = i+1;
            }

            while(q_sort[p] < q_sort[j] && j > begin)
            {
                  if(j == p+1 && i <=p)
                  {
                        j = p-1;
                  }
                  else
                  {
                        j = j-1;
                  }
            }
```

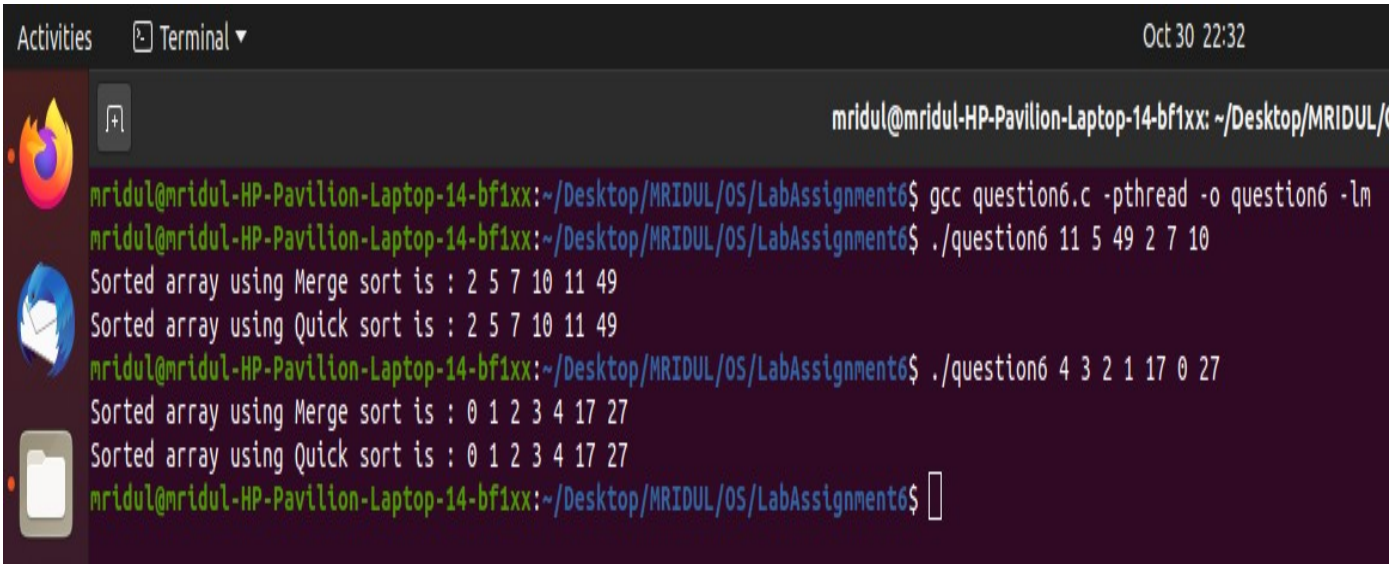
```
            if(i < j)
            {
                int temp = q_sort[i];
                q_sort[i] = q_sort[j];
                q_sort[j] = temp;
            }
        }

        int temp = q_sort[p];
        q_sort[p] = q_sort[j];
        q_sort[j] = temp;
        val = j;

        return val;
}

void *quicksort(void *arg1)
{
        struct data *temp = arg1;
        int begin = temp -> begin;
        int end = temp -> end;

        if(begin < end)
        {
            int j = partition(begin, end);

            struct data left;
            left.begin = begin;
            left. end = j-1;

            struct data right;
            right.begin = j+1;
            right.end = end;

            pthread_t tid[2];
            pthread_attr_t attr;
            pthread_attr_init(&attr);

            pthread_create(&tid[0], &attr, quicksort, &left);
```

```
            pthread_create(&tid[1], &attr, quicksort, &right);

            pthread_join(tid[0], NULL);
            pthread_join(tid[1], NULL);
        }
        pthread_exit(0);
}

void merge_array(int begin, int mid, int end)
{
        int n1 = mid - begin + 1;
        int n2 = end - mid;

        int L[n1]; int R[n2];

        for(int i = 0; i < n1; i = i+1)
        {
                L[i] = m_sort[begin + i];
        }
        for(int j = 0; j < n2; j = j+1)
        {
                R[j] = m_sort[mid + 1 + j];
        }

        int i = 0; int j = 0; int k = begin;
        while(i < n1 && j < n2)
        {
                if(L[i] <= R[j])
                {
                        m_sort[k] = L[i];
                        i = i+1;
                }
                else
                {
                        m_sort[k] = R[j];
                        j = j+1;
                }
                k = k+1;
        }
```

```
        while(i < n1)
        {
                m_sort[k] = L[i];
                i = i+1;
                k = k+1;
        }
        while(j < n2)
        {
                m_sort[k] = R[j];
                j = j+1;
                k = k+1;
        }
}

void *mergesort(void *arg2)
{
        struct data *temp = arg2;
        int begin = temp -> begin;
        int end = temp -> end;

        if(begin < end)
        {
                int mid = (begin + end)/2;

                struct data left;
                left.begin = begin;
                left. end = mid;

                struct data right;
                right.begin = mid+1;
                right.end = end;

                pthread_t tid[2];
                pthread_attr_t attr;
                pthread_attr_init(&attr);

                pthread_create(&tid[0], &attr, mergesort, &left);
                pthread_create(&tid[1], &attr, mergesort, &right);
```

```
            pthread_join(tid[0], NULL);
            pthread_join(tid[1], NULL);

            merge_array(begin, mid, end);
        }
        pthread_exit(0);
}
```

Output :-



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

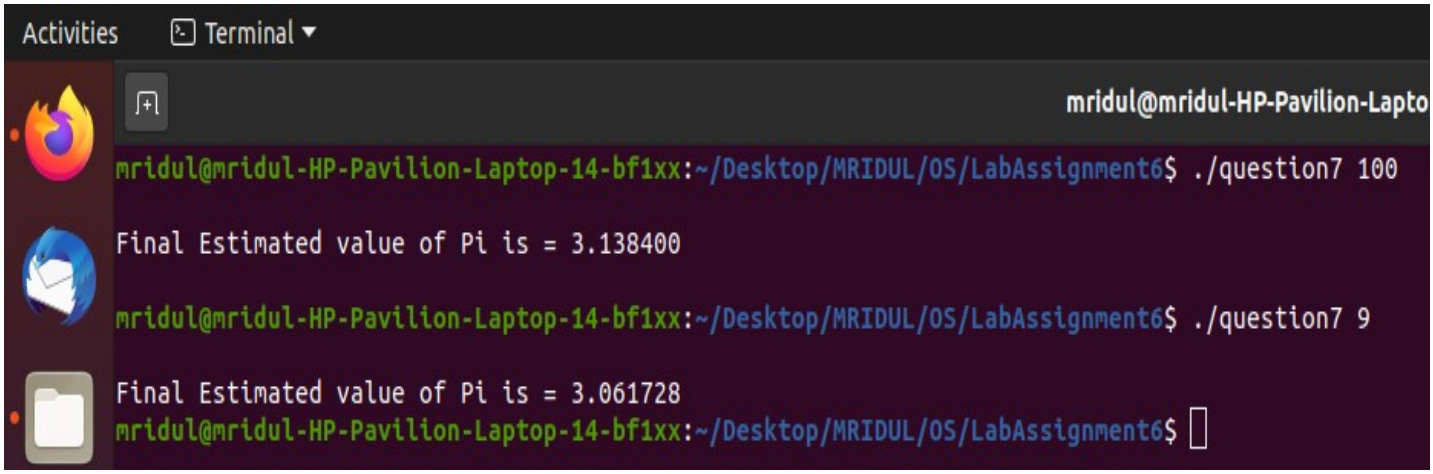
Code :-

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<pthread.h>
```

```
int circle_points = 0;
int square_points = 0;

double rand_x; double rand_y; double original_distance;

int interval;

void *runner(void *parameter);

int main(int argc, char *argv[])
{
	if(argc != 2)
	{
		printf("This is not allowed\n");
	}
	else
	{
		interval = atoi(argv[1]);
		pthread_t tid[interval * interval];
		pthread_attr_t attr;
		pthread_attr_init(&attr);
		srand(time(NULL));

		for(int i = 0; i < (interval * interval); i = i+1)
		{
			pthread_create(&tid[i], &attr, runner, NULL);
		}

		for(int i = 0; i < (interval * interval); i = i+1)
		{
			pthread_join(tid[i], NULL);
		}

		double pi = (double)(4 * circle_points) / square_points;
		printf("\nFinal Estimated value of Pi is = %f\n", pi);
	}
	printf("\n");
	return 0;
}
```

```
void *runner(void *parameter)
{
	rand_x = (double)(rand() % (interval + 1)) / interval;
	rand_y = (double)(rand() % (interval + 1)) / interval;

	original_distance = (rand_x * rand_x) + (rand_y * rand_y);

	if(original_distance <= 1)
	{
		circle_points = circle_points + 1;
	}
	square_points = square_points + 1;

	pthread_exit(NULL);
}
```

Output :-



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

Code :-

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

```
int N = 2;

struct data
{
    int** arr;
    int** temp;
    int p;
    int q;
    int n;
};

struct data init_data(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);

int main()
{
    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);

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

         printf("\nThe Inverse is :\n");
```

```
        if (inverse(A, inv))
                display(inv);
        return 0;
}

struct data init_data(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;
}

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

        for (int row = 0; row < temp->n; row++)
        {
                for (int col = 0; col < temp->n; col++)
                {
                        if (row != temp->p && col != temp->q)
                        {
                                temp->temp[i][j++] = temp->arr[row][col];

                                if (j == temp->n - 1)
                                {
                                        j = 0;
                                        i++;
                                }
```

```
                    }
                }
        }

    pthread_exit(0);
}


int determinant(int** A, int n)
{
        int D = 0;

        if (n == 1)
                return A[0][0];
        int sign = 1;

    pthread_t tid[n];
    struct data params[n];
    for(int i = 0; i < n; i++)
        {
        params[i] = init_data(A, 0, i, n);
        pthread_create(&tid[i], NULL, getCofactor, &params[i]);
    }

    for(int i = 0; i < n; i++)
        pthread_join(tid[i], NULL);

        for (int f = 0; f < n; f++)
        {

                D += sign * A[0][f] * determinant(params[f].temp, n - 1);

                sign = -sign;
        }
        return D;
}


void adjoint(int** A,int** adj)
```

```
{
        if (N == 1){
                adj[0][0] = 1;
                return;
        }
        int sign = 1;

    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] = init_data(A, i, j, N);
          pthread_create(&tid[i][j], NULL, getCofactor, &params[i][j]);
       }
    }

    for(int i = 0; i < N; i++)
       for(int j = 0; j < N; j++)
          pthread_join(tid[i][j], NULL);

        for (int i=0; i<N; i++)
        {
                for (int j=0; j<N; j++)
                {
                        sign = ((i+j)%2==0)? 1: -1;

                        adj[j][i] = (sign)*(determinant(params[i][j].temp, N-1));
                }
        }
}


bool inverse(int** A, float** inverse)
{

        int det = determinant(A, N);
        if (det == 0)
        {
```

```
            printf("Singular matrix, can't find its inverse\n");
            return false;
        }
        int** adj;
    adj = (int **) malloc(sizeof(int*)*N);
    for(int k = 0; k < N; k++)
       adj[k] = (int*)malloc(sizeof(int)*N);

        adjoint(A, adj);

        for (int i=0; i<N; i++)
            for (int j=0; j<N; j++)
                inverse[i][j] = adj[i][j]/(float)det;
        return true;
}


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 :-

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 :-

```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<string.h>
#include<time.h>
#include<sys/types.h>
#include<sys/wait.h>
#include<unistd.h>
#include<pthread.h>

struct key_value
{
	int key;
	int value;
};

int fib(int n);
void *runner(void *param);

int main()
{
	int range;
	printf("Enter the number of fibonacci numbers to generate : \n");
	scanf("%d", &range);

	struct key_value *generate = (struct
key_value*)malloc(range*sizeof(struct key_value));

	pthread_t tid[range];

	printf("The generated Fibonacci series of %d terms is\n", range);
```

```
	for(int i = 0; i < range; i = i+1)
	{
		generate[i].key = i;
		pthread_create(&tid[i], NULL, runner, &generate[i]);
		pthread_join(tid[i], NULL);
	}

	for(int i = 0; i < range; i = i+1)
	{
		printf("%d\n", generate[i].value);
	}

	return 0;
}

int fib(int n)
{
	if(n == 0 || n ==1)
	{
		return n;
	}
	else
	{
		return fib(n-1) + fib(n-2);
	}
}

void *runner(void *param)
{
	struct key_value *temporary = (struct key_value*)param;
	temporary -> value = fib(temporary -> key);
	pthread_exit(NULL);
}
```

Output :-


Activities
Terminal
Oct 30 23:12
mridul@mridul-HP-Pavilion-Laptop-14-bf1xx: ~/Desktop/MRIDUL/OS
mridul@mridul-HP-Pavilion-Laptop-14-bf1xx:~/Desktop/MRIDUL/OS/LabAssignment6$ gcc question9.c -pthread -o question9 -lm
mridul@mridul-HP-Pavilion-Laptop-14-bf1xx:~/Desktop/MRIDUL/OS/LabAssignment6$ ./question9 6
Enter the number of fibonacci numbers to generate :
5
The generated Fibonacci series of 5 terms is
0
1
1
2
mridul@mridul-HP-Pavilion-Laptop-14-bf1xx:~/Desktop/MRIDUL/OS/LabAssignment6$