

Assignment-6

Threading

Subash Mylraj
(CED18I051)

30 October 2020

Question 1: Generate Armstrong number generation within a range.

Code:

```
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
#include<stdlib.h>
#include<math.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int main(int argc, char const *argv[]){

    int n=atoi(argv[1]);

    int i=0, num=0, sum=0, pow_to=0, j=0;
    struct armstrong {
        int numbers[100];
        int n;
    };

    key_t key = ftok("shmfile", 65);
    int shmid = shmget(key, 5*sizeof(struct armstrong), 0666|IPC_CREAT);
    struct armstrong *armstg = shmat(shmid, 0, 0);

    for(i=0; i<n; i=i+1){

        sum = 0;
        num = i;
        pow_to = 0;
        for(; num>0; num/=10)
            pow_to+=1;
        num = i;

        pid_t pid = fork();
        if(pid == 0){
            while (num>0){
                int digit = num%10;
                sum = sum + pow(digit, pow_to);
                num = num/10;
            }
            if(sum == i){
                armstg->numbers[armstg->n] = i;
                armstg->n += 1;
            }
            exit(0);
        }
    }
```

```

}

wait(NULL);

printf("Armstrong numbers:\n");
for(int i=0; i<armstg->n; i++)
    printf("%d\n", armstg->numbers[i]);

shmdt(armstg);
shmctl(shmid,IPC_RMID,NULL);

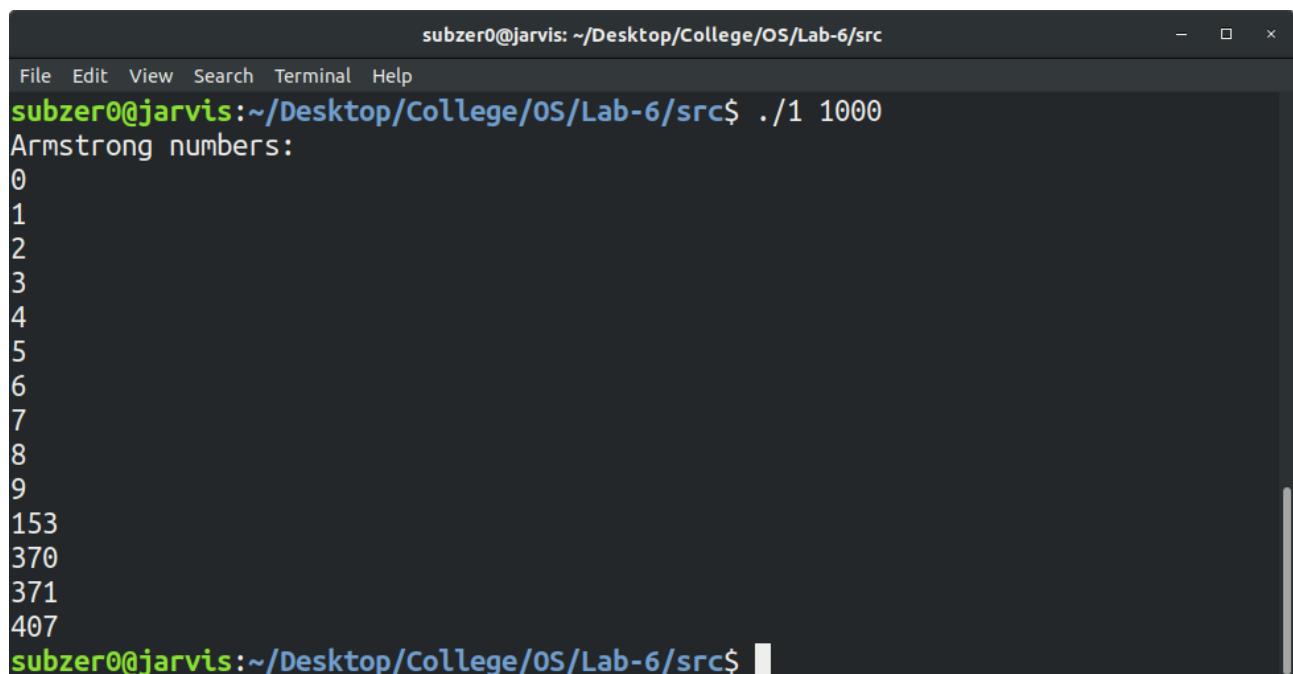
return 0;
}

```

Explanation:

This program finds all the armstrong numbers from 0 to range (used input). Shared memory was used to store all the armstrong numbers in the range. A structure was defined to store the array and the iterator for that array.

Output:



```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./1 1000
Armstrong numbers:
0
1
2
3
4
5
6
7
8
9
153
370
371
407
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

Question 2: Ascending Order sort and Descending order sort.

Code:

```

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

struct arguments

```

```

{
    int arr[10];
    int n;
    int opt;
};

void* sort (void *arg){

    struct arguments* args = (struct arguments*)arg;
    int opt = args->opt;
    if(args->opt){
        printf("Descending Array:  ");
    }
    else{
        printf("Ascending Array:  ");
    }

    for(int i=0; i<args->n; i++){
        for(int j=1; j<args->n-i; j++){
            if(((opt*(args->arr[j] < args->arr[j-1])) + ((1-opt)*(args->arr[j] > args->arr[j-1])))){
                args->arr[j] += args->arr[j-1];
                args->arr[j-1] = args->arr[j] - args->arr[j-1];
                args->arr[j] -= args->arr[j-1];
            }
        }
    }

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

    pthread_exit(NULL);
}

int main (){

    struct arguments arg_asc, arg_dsc;

    printf("Enter size of array: ");
    scanf("%d", &arg_asc.n);

    printf("Enter elements of array: ");
    for(int i=0; i<arg_asc.n; i++)
        scanf("%d", &arg_asc.arr[i]);

    printf("Original Array:  ");
    for(int i=0; i<arg_asc.n; i++)
        printf("%d ", arg_asc.arr[i]);
    printf("\n");

    arg_dsc = arg_asc;

    struct arguments *asc_ptr=&arg_asc, *dsc_ptr=&arg_dsc;
    pthread_t asc, dsc;

    arg_asc.opt = 1;
    pthread_create(&asc, NULL, sort, asc_ptr);
    arg_dsc.opt = 0;
    pthread_create(&dsc, NULL, sort, dsc_ptr);

    pthread_join(asc, NULL);
    pthread_join(dsc, NULL);

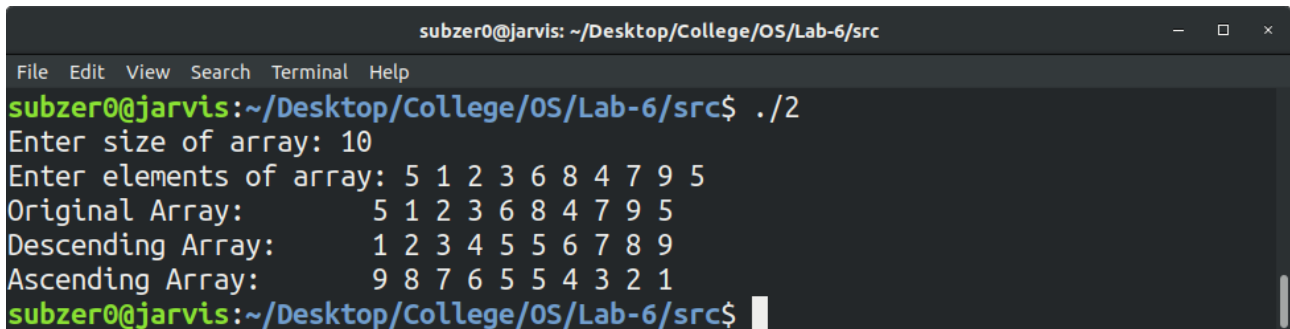
    return 0;
}

```

Explanation:

Ascending and Descending sorts are done in 2 separate threads. There is only one sort function which accepts an additional argument for the type of sort. The conditional statement within the sort makes sure that both ascending and descending sorting can be performed. The conditional statement resembles to the expression of a multiplexor.

Output:



```
subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./2
Enter size of array: 10
Enter elements of array: 5 1 2 3 6 8 4 7 9 5
Original Array:      5 1 2 3 6 8 4 7 9 5
Descending Array:    1 2 3 4 5 5 6 7 8 9
Ascending Array:      9 8 7 6 5 5 4 3 2 1
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$
```

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 element search as well)

Code:

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

struct arguments{
    int arr[20][2];
    int start;
    int end;
    int key;
};

pthread_t threads[100];
int itr=0;

void* binary_search (void *);

int main (){

    struct arguments *args = (struct arguments *)malloc(sizeof(struct arguments));
    int n;
    printf("Enter size of array: ");
    scanf("%d", &n);

    printf("Enter elements of array: ");
    for(int i=0; i<n; i++)
        scanf("%d", &args->arr[i][0]);
```

```

printf("Enter key to search for: ");
scanf("%d", &args->key);

args->end = n;
args->start = 0;

binary_search(args);

for(int i=0; i<itr; i++){
    pthread_join(threads[i], NULL);
}

for(int i=0; i<n; i++)
    if(args->arr[i][1] == 1)
        printf("Key found at address: %d\n", i);

return 0;
}

void* binary_search (void* a){

    struct arguments* args = (struct arguments*) a;
    int mid = (args->end + args->start)/2;
    int end = args->end, start = args->start;

    if(args->start > args->end){
        return NULL;
    }
    if(args->start == args->end){
        if(args->arr[mid][0] == args->key){
            args->arr[mid][1] = 1;
        }
        return NULL;
    }
    if(args->start+1 == args->end){

        args->end=args->start;
        pthread_create(&threads[itr], NULL, binary_search, args);itr++;

        args->start=args->end=end;
        pthread_create(&threads[itr], NULL, binary_search, args);itr++;
        args->start=start; args->end=end;
    }
    if(args->arr[mid][0] == args->key){
        args->arr[mid][1] = 1;
    }

    args->end = mid-1;
    binary_search(args);
    pthread_create(&threads[itr], NULL, binary_search, args);itr++;

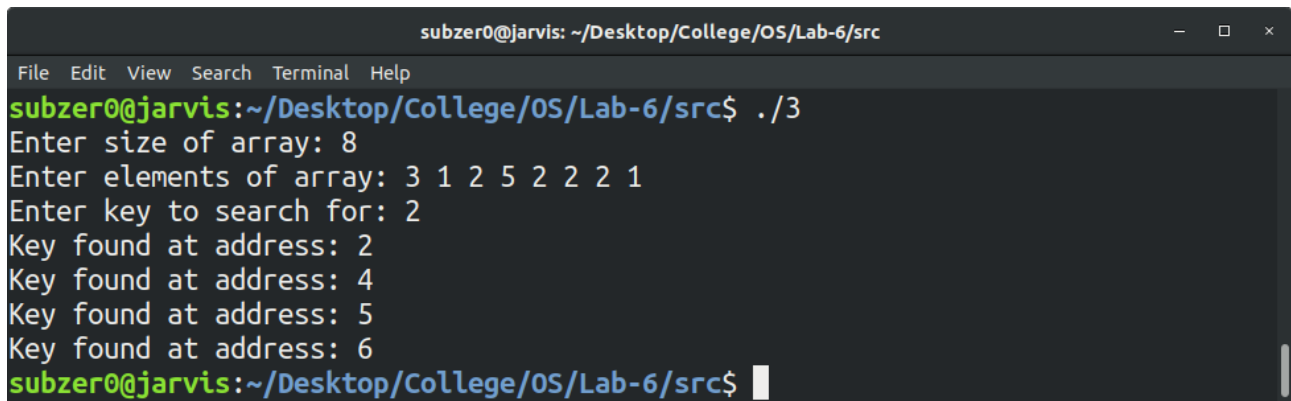
    args->start = mid+1;
    args->end = end;
    binary_search(args);
    pthread_create(&threads[itr], NULL, binary_search, args);itr++;
}

```

Explanation:

This code implements a search algorithm (non-sorted array) by initially checking if the middle element is the key. If not, then both the left and the right sub-halves are checked in the same recursively. The checking for the left and right sub-halves is done using threading. Hence this is a parallelized implementation. This is slightly similar to that of binary search in the fact that the middle element is checked in every iteration.

Output:

A terminal window titled 'subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src' with a menu bar (File, Edit, View, Search, Terminal, Help). The prompt is 'subzer0@jarvis:~/Desktop/College/OS/Lab-6/src\$./3'. The output shows: 'Enter size of array: 8', 'Enter elements of array: 3 1 2 5 2 2 2 1', 'Enter key to search for: 2', and four lines of 'Key found at address: 2', '4', '5', and '6' respectively. The prompt returns to 'subzer0@jarvis:~/Desktop/College/OS/Lab-6/src\$'.

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

Code:

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

void* test_prime(void*);
void* runner (void* args);

int main(int argc, char const *argv[]){

    int range = atoi(argv[1]);

    pthread_t runner_thread;
    pthread_create(&runner_thread, NULL, runner, &range);

    pthread_join(runner_thread, NULL);
    printf("\n");
}

void* runner (void* args){
    int range = *(int *)args;

    pthread_t tid[range];
    int thread_args[range];
    printf("Prime numbers from 1 to %d: 1 2 ", range);
    for(int i=2; i<range; i++){
        thread_args[i] = i;
        pthread_create(&tid[i], NULL, test_prime, &thread_args[i]);
    }

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

void* test_prime(void* args){

    int num = *(int *)args;
    int flag = 0;
```

```

for(int i=2; i<=num/2+1; i++){
    if(num%i == 0){
        flag = 1;
        break;
    }
}

if(flag == 0){
    printf("%d ", num);
}
num += 1;

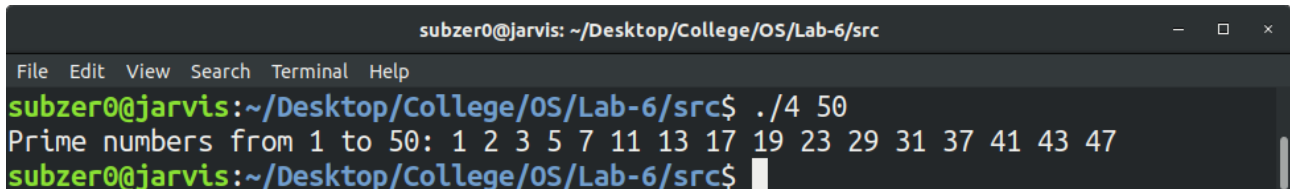
return NULL;
}

```

Explanation:

The code above, iterates from 3 through range (user input). Every number in this range is checked for a prime number. The check for each of these numbers is done by a separate thread. Hence this code is parallelizable as possible.

Output:



```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./4 50
Prime numbers from 1 to 50: 1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

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

Code:

```

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

struct mmm {    // stands for mean median mode
    int arr[100], n, median, mode;
    float mean;
};

void* mean (void*);
void* median (void*);
void* mode (void*);

int main(int argc, char const *argv[]){

    // int array[100]={0}, n=0, i=0, j=0, temp=0;
    struct mmm args;

    args.n = atoi(argv[1]);
    for(int i=0; i<args.n; i++){
        args.arr[i] = atoi(argv[i+2]);
    }
}

```

```

}

// Initialize elements of args
args.mean = 0;
args.median = 0;
args.mode = 0;

// Sort the array
for(int i=0; i<args.n; i++){
    for(int j=0; j<args.n-1; j++){
        if(args.arr[j] > args.arr[j+1]){
            int temp = args.arr[j];
            args.arr[j] = args.arr[j+1];
            args.arr[j+1] = temp;
        }
    }
}

struct mmm* args_ptr = &args;
pthread_t tid[3];
pthread_create(&tid[0], NULL, mean, args_ptr);
pthread_create(&tid[1], NULL, median, args_ptr);
pthread_create(&tid[2], NULL, mode, args_ptr);

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

printf("Mean: %f\nMedian: %d\nMode: %d\n", args.mean, args.median, args.mode);

return 0;
}

void* mean (void* args){
    struct mmm* arg = (struct mmm*)args;

    for(int i=0; i<arg->n; i++){
        arg->mean += arg->arr[i];
    }
    arg->mean /= arg->n;
    return NULL;
}

void* median (void* args){
    struct mmm* arg = (struct mmm*)args;
    arg->median = arg->arr[arg->n/2];
    return NULL;
}

void* mode (void* args){
    struct mmm* arg = (struct mmm*)args;

    int max_count = 0, count = 1, curr = arg->arr[0];
    arg->mode = arg->arr[0];
    for(int i=1; i<arg->n; i++){
        if(arg->arr[i] != curr){
            if(count > max_count){
                max_count = count;
                arg->mode = curr;
            }
            curr = arg->arr[i];
            count = 1;
        }
        else{
            count += 1;
        }
    }
}

```



```

}
if(count > max_count){
    max_count = count;
    arg->mode = curr;
}
return NULL;
}

```

Explanation:

Mean, Median and Mode are calculated in 3 separate threads. Since threads accept the input as a void*, a structure has been created with the array, its size, mean, median and mode as its elements. The memory sharing feature of threads is utilised by passing the same address to all the 3 threads. Since the threads will be writing to different addresses within the structure, there will be no race condition possible.

Output:

```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./5 7 50 10 20 50 20 10 50
Mean: 30.000000
Median: 20
Mode: 50
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

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

Code: Merge Sort

```

#include<stdio.h>
#include<sys/shm.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/wait.h>
#include<pthread.h>

struct arguments
{
    int s, e;
};

void* merge_sort(void*);
void merge(int s, int mid, int e);
void copy_arguments (const struct arguments*, struct arguments*);

int array[] = {9, 8, 7, 4, 1, 3, 2, 5}, n = sizeof(array)/sizeof(int);

int main (){

    struct arguments* args = (struct arguments*)malloc(sizeof(struct arguments));

    printf("Input Array: ");
    for(int i=0; i<n; i++){
        printf("%d ", array[i]);
    }

```

```

}
printf("\n");

args->s = 0; args->e = n-1;
merge_sort(args);

printf("Output Array: ");
for(int i=0; i<n; i++){
    printf("%d ", array[i]);
}
printf("\n");

return 0;
}

void merge(int s, int mid, int e){

    int temp[e-s+1];
    int i = 0;
    int l = s;
    int r = mid+1;
    while( l<=mid && r<=e ){

        if(array[l] < array[r]){
            temp[i] = array[l];
            l++; i++;
        }
        else if(array[l] > array[r]){
            temp[i] = array[r];
            r++; i++;
        }
        else if(array[l] == array[r]){
            temp[i] = array[r];
            i++; r++;
            temp[i] = array[l];
            i++; l++;
        }
    }

    while(l <= mid){
        temp[i] = array[l];
        i++; l++;
    }

    while(r <= e){
        temp[i] = array[r];
        i++; r++;
    }

    for(int j=0; j<i; j++){
        array[s+j] = temp[j];
    }
}

void* merge_sort(void* arg){

    struct arguments* args = (struct arguments*)arg;

    if(args->s < args->e){

        int mid = (args->s + args->e)/2;

        pthread_t right_tid, left_tid;

        struct arguments* left_args = (struct arguments*)malloc(sizeof(struct arguments));

```

```

    struct arguments* right_args = (struct arguments*)malloc(sizeof(struct arguments));

    copy_arguments(args, left_args);
    left_args->e = mid;

    copy_arguments(args, right_args);
    right_args->s = mid+1;

    pthread_create(&left_tid, NULL, merge_sort, left_args);
    pthread_create(&right_tid, NULL, merge_sort, right_args);

    pthread_join(left_tid, NULL);
    pthread_join(right_tid, NULL);

    merge(args->s, mid, args->e);
}
}

void copy_arguments (const struct arguments* src, struct arguments* dest){
    dest->s = src->s;
    dest->e = src->e;
}

```

Explanation:

The above code is a multithreading implementation of merge sort. Every division is done by a new thread. Every merge is done by the calling thread when its child threads are complete.

Output:

```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./6_merge
Input Array: 9 8 7 4 1 3 2 5
Output Array: 1 2 3 4 5 7 8 9
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

Code: Quick Sort

```

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

struct arguments{
    int b, e, k;
};

void* runner(void*);
void swap(int*, int*);
void quicksort(int [], int, int, int);
int partition (int [], int, int);

int n, arr[20], k;

int main(int argc, char *argv[]){

    if(argc < 3){
        printf("Invalid input\n");
        exit(0);
    }
}

```

```

}

n=atoi(argv[1]);
if(argc != n+2){
    printf("Invalid input\n");
    exit(0);
}

for(int i=0; i<n; i++){
    arr[i] = atoi(argv[i+2]);
}

k = n;
quicksort(arr, 0, n-1, k);

printf("Sorted array:\n");

for(int i=0; i<k; i++)
    printf("%d ", arr[i]);

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

void* runner(void* param){

    struct arguments* p=(struct arguments*)param;
    quicksort(arr, p->b, p->e, p->k);
    return NULL;
}

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

int partition (int arr[], int low, int high){
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high- 1; j++){
        if (arr[j] < pivot){
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i+1], &arr[high]);
    return i+1;
}

void quicksort(int arr[], int low, int high,int k){

    if (high>low){
        int q = partition(arr, low, high);

        struct arguments p, r;

        p.b = low;
        p.e = q-1;
        p.k = k;

        r.b = q+1;
        r.e = high;
        r.k = k;
    }
}

```

```

pthread_t tid[2];

pthread_create(&tid[0], NULL, runner, &p);
pthread_create(&tid[1], NULL, runner, &r);

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

```

Explanation:

The above code is a multithreading implementation of quick sort.

Output:

```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./6_quick 8 1 3 2 5 7 5 9 8
Sorted array:
1 2 3 5 5 7 8 9
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

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

Code:

```

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

#define ITRS 100000

int num_s=0, num_c=0;
void* runner (void* args);
int whereIsPoint(float x, float y);

int main (){

pthread_t tid[4];

for(int i=0; i<4; i++){
    srand(time(0));
    pthread_create(&tid[i], NULL, runner, NULL);
}

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

float pi = 4.0 * num_c / num_s;

printf("Approximate Value of PI = %f\n", pi);

```

```

    return 0;
}

int whereIsPoint(float x, float y){

    float distance = sqrt((x*x) + (y*y));

    if(distance < 1)
        return 1;
    else
        return 0;
}

void* runner (void* args){

    time_t t;
    float x, y;
    for(int i=0; i<ITRS; i++){
        x = (float)rand() / RAND_MAX;
        y = (float)rand() / RAND_MAX;
        if(whereIsPoint(x, y)){
            num_c += 1;
        }
    }

    num_s += ITRS;
    return NULL;
}

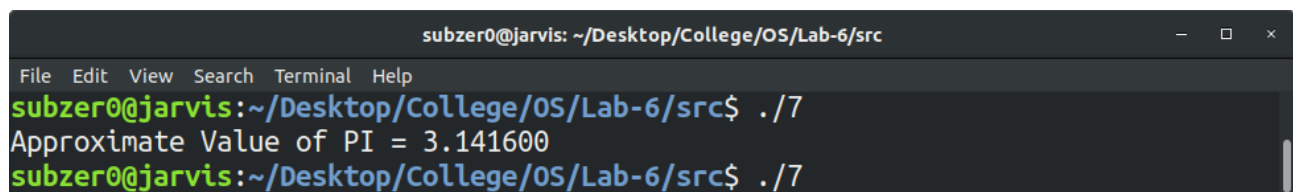
```

Explanation:

Monte Carlo's estimation of PI was done in this code using threading. 4 threads are created. Each thread finds the number of points outside and inside the circle for ITRS number of times. ITRS defines the number of iterations that each thread runs the function. Once all threads are complete, the estimation is completed by putting the values in the formula.

The number of threads have been set to 4 to avoid the CPU from backlogging the active threads and causing the program to run inefficiently.

Output:



```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./7
Approximate Value of PI = 3.141600
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./7

```

Extra Credit Questions

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

Code:

```

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

void *runner(void* param);
#define N 4
int A[N][N] ={{5, -2, 2, 7},
              {1, 0, 0, 3},
              {-3, 1, 5, 0},
              {3, -1, -9, 4}};

int adj[N][N]; // To store adjoint of A[] []
float inv[N][N]; // To store inverse of A[] []

struct passing{
    int temp[N][N],p,q,n;
};

void getCofactor(int A[N][N], int temp[N][N], int p, int q, int n){
    int i = 0, j = 0;

    for (int row = 0; row < n; row++){
        for (int col = 0; col < n; col++){

            if (row != p && col != q){
                temp[i][j++] = A[row][col];

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

int determinant(int A[N][N], int n){

    int D = 0;

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

    int temp[N][N];

    int sign = 1;

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

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

        sign = -sign;
    }

    return D;
}

void adjoint(int A[N][N], int adj[N][N]){

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

```

```

    int sign = 1, temp[N][N];

    for (int i = 0; i < N; i++){
        pthread_t tid[N];
        struct passing p[N];
        for (int j = 0; j < N; j++){
            p[j].p = i;
            p[j].q = j;
            p[j].n = N;
            pthread_create(&tid[j], NULL, runner, &p[j]);
        }
        for (int j = 0; j < N; j++){
            pthread_join(tid[j], NULL);
            sign = ((i + j) % 2 == 0) ? 1 : -1;

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

bool inverse(int A[N][N], float inverse[N][N]){

    int det = determinant(A, N);
    if (det == 0){
        printf("Singular matrix, can't find its inverse");
        return 0;
    }

    int adj[N][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 1;
}

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

void disinvt(float A[N][N]){
    for (int i=0; i<N; i++){
        for (int j=0; j<N; j++)
            printf("%f ", A[i][j]);
        printf("\n");
    }
}

int main(){

    printf("Input matrix is :\n");
    display(A);

    printf("\nThe Adjoint is :\n");
    adjoint(A, adj);
    display(adj);

    printf("\nThe Inverse is :\n");
    if (inverse(A, inv))

```



```

        disinvert(inv);

    return 0;
}

void* runner(void* param){
    struct passing *p=(struct passing*)param;
    getCofactor(A, p->temp, p->p,p->q, p->n);
}

```

Explanation:

This code is a multithreading approach of inverting a matrix. To find the inverse of a matrix, its determinant and adjoint needs to be calculated. In this code, finding the adjoint of the matrix has been parallelized using threading. In order to avoid the creation of a large number of threads, adjoint of every element in a row is calculated parallelly. So the maximum number of threads created will be equal to number of columns. Hence the time complexity is reduced from $O(n^2)$ to $O(n)$.

Output:

```

subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./8
Input matrix is :
5 -2 2 7
1 0 0 3
-3 1 5 0
3 -1 -9 4

The Adjoint is :
-12 76 -60 -36
-56 208 -82 -58
4 4 -2 -10
4 4 20 12

The Inverse is :
-0.136364 0.863636 -0.681818 -0.409091
-0.636364 2.363636 -0.931818 -0.659091
0.045455 0.045455 -0.022727 -0.113636
0.045455 0.045455 0.227273 0.136364
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$

```

Question 9: Estimation of PI Value using Monte carlo simulation technique (refer the internet for the method..) usinreads.

Fibonacci series depends on its previous 2 elements. Hence whatever algo is used, to calculate the nth element, its previous elements have to be calculated. So, the time complexity can never be below $O(n)$.

For example, to calculate $fib(7)$ from a top to bottom approach, we need to find $fib(6)$, $fib(5)$, ... $fib(2)$, $fib(1)$. To calculate $fib(6)$, we need to find $fib(5)$, $fib(4)$, ... $fib(2)$, $fib(1)$. Hence finding all these in a linear sequential manner would lead to a time complexity of about $O(2^n)$.

Although, calculating fibonacci using a bottom up approach will take only $O(n)$.

Example: To calculate $fib(5)$, first calculate $fib(3)$ using $fib(2)$ and $fib(1)$, then calculate $fib(4)$ using $fib(3)$

and $fib(2)$ and then finally calculate $fib(5)$ using $fib(4)$ and $fib(3)$. As it can be observed, this method will take only $O(n)$.

Hence I feel there is no need to parallelize the top to bottom approach requiring more hardware when the solution to the problem is just calculating the n^{th} fibonacci number using a bottom up approach.

Question 10: Longest common sub sequence generation problem using threads.

Code:

```
#include<iostream>
#include<string.h>
#include<pthread.h>
#include<vector>
#include<bits/stdc++.h>

using namespace std;

vector<string> str1_subseq;

void* check_common_seq (void* arg);
void subsequence(string str);

string str1, str2;
vector<string> common;

int main(int argc, char const *argv[]){

    if(strlen(argv[1]) < strlen(argv[2])){
        str1=argv[1];
        str2=argv[2];
    }
    else{
        str1=argv[2];
        str2=argv[1];
    }

    // finds all the subsequences of str1
    subsequence(str1);
    sort(str1_subseq.begin(), str1_subseq.end());
    str1_subseq.erase(std::unique(str1_subseq.begin(), str1_subseq.end()), str1_subseq.end());

    pthread_t threads[100];
    int itr=0;

    // find all common subsequences
    for(int i=0; i<str1_subseq.size(); i++){

        pthread_create(&threads[itr], NULL, check_common_seq, (void*)&str1_subseq[i]);
        pthread_join(threads[itr], NULL);
        itr++;
    }

    // Find the longest common sequence from all common sequences
    string lcs = common[0];
    for(int i=1; i<common.size(); i++){
        if(lcs.length() < common[i].length())
            lcs = common[i];
    }
}
```

```

    if(lcs.length() >= 2){
        cout<<"Longest Common Subsequence: "<<lcs<<endl;
    }
    else{
        cout<<"Longest Common Subsequence of length more than 2 does not exist"<<endl;
    }
    return 0;
}

void* check_common_seq (void* arg){

    string* s = (string*)arg;

    int i, j=0;
    for(i=0; i<(*s).length(); i++){
        int flag=0;
        for(j; j<str2.length(); j++){
            if((*s)[i] == str2[j]){
                j++;
                flag = 1;
                break;
            }
        }
        if(flag == 0){
            break;
        }
    }
    if(i == (*s).length()){
        // cout<<*s<<endl;
        common.push_back(*s);
    }

    pthread_exit(NULL);
}

void subsequence(string str){
    for (int i = 0; i < str.length(); i++)
    {
        for (int j = str.length(); j > i; j--)
        {
            string sub_str = str.substr(i, j);
            str1_subseq.push_back(sub_str);

            for (int k = 1; k < sub_str.length() - 1; k++)
            {
                string sb = sub_str;
                sb.erase(sb.begin() + k);
                subsequence(sb);
            }
        }
    }
}

```

Explanation:

This program uses vectors and stl functions for speedy coding. The program can be viewed as two parts: 1. The first part finds all the subsequences of the first string. 2. The second part checks if any of those subsequences matches with any subsequence of the second string.

In this program, I have multithreaded the second part.

All subsequences of the first string are stored in a vector named str1_subseq. The function check_common_seq takes one argument as a string. This function checks if there exists a subsequence in str2 which matches its argument. If such a subsequence exists, it is appended to a vector named common.

Output:

```
subzer0@jarvis: ~/Desktop/College/OS/Lab-6/src
File Edit View Search Terminal Help
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./10 asdfas fdsa
Longest Common Subsequence: da
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./10 asdfas fdsaass
Longest Common Subsequence: aas
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$ ./10 asdfasaaa fdsaass
Longest Common Subsequence: dsaa
subzer0@jarvis:~/Desktop/College/OS/Lab-6/src$
```