
Operating System Assignment-6

1. Generate Armstrong number generation within a range.

Approach:

Setting a variable MAX_THREADS which is the number of threads used at a time. For every iteration from lower bound to upper bound (input from user), MAX_THREADS number of threads are created and runner function is called. Check for armstrong number happens in runner function and prints if the number is armstrong

In total $(\text{upperbound} - \text{lowerbound}) * \text{MAX_THREADS}$ number of threads are created in which MAX_THREADS number of threads are executed in parallel.

Code:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
int sum; /* this data is shared by the thread(s) */
#define MAX_THREADS 10
void *runner (void *param); /* the thread */
int power(int a, int b) {
    int pow = 1;
    while(b) {
        pow *= a;
        b--;
    }
    return pow;
}

int main(int argc, char *argv[])
{
    if(argc != 3) {printf("invalid input\n"); exit(0);}
}
```

```

int lnum=atoi(argv[1]),hnum=atoi(argv[2]);
//int sum[1000]={0};
printf("The armstrong numbers in the givrn range:\n");
for(int i=lnum;i<hnum;i+=MAX_THREADS)
{
pthread_t tid[MAX_THREADS];
pthread_attr_t attr[MAX_THREADS];
int temp[MAX_THREADS];
for(int j=0;j<MAX_THREADS;++j)
{
pthread_attr_init (&attr[j]);
temp[j]=i+j;
pthread_create(&tid[j], &attr[j], runner, &temp[j]);
}
for(int j=0;j<MAX_THREADS;++j)
{
pthread_join(tid[j], NULL);
}
}
return 0;
}
void *runner( void *param)
{

int *num= (int *) param;

int c=0,temp=*num;
while(temp){
temp=temp/10;
c++;
}
temp=*num;
sum=0;
while(temp){
int x=temp%10;
sum+=power(x,c);
temp=temp/10;

```

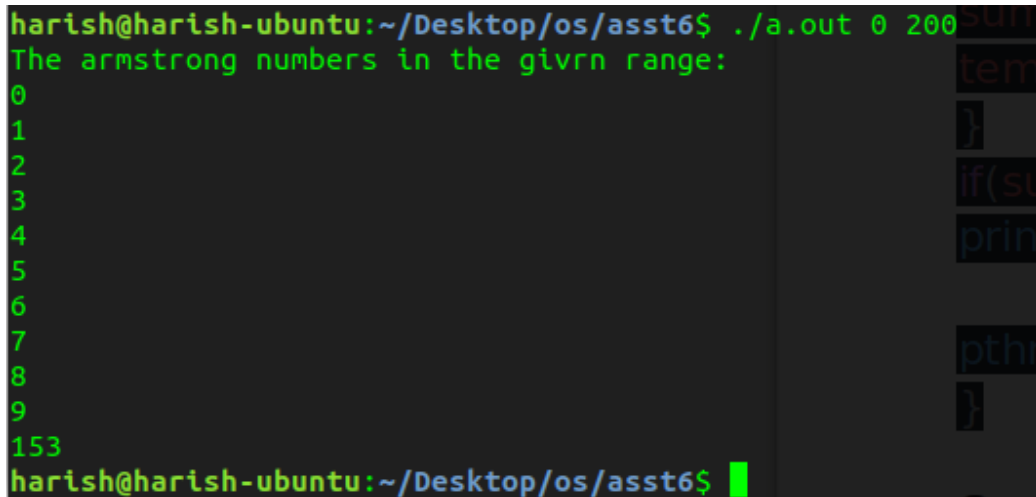
```

}
if(sum==*num)
printf("%d\n", *num);

pthread_exit(0);
}

```

Output:



```

harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 0 200
The armstrong numbers in the givrn range:
0
1
2
3
4
5
6
7
8
9
153
harish@harish-ubuntu:~/Desktop/os/asst6$

```

2. Ascending Order sort and Descending order sort.

Approach:

Two threads are created, where one takes care of ascending order sort and other takes care of descending sort. Order is passed as argument to runner function. Based on order the comparison happens. Two copies of same array is sent to two threads to store and print ascending and descending parallelly.

Code:

```

#include <pthread.h>
#include <stdio.h>
#include<stdlib.h>
#include<string.h>
int sum; /* this data is shared by the thread(s) */
void *runner (void *param); /* the thread */

```

```

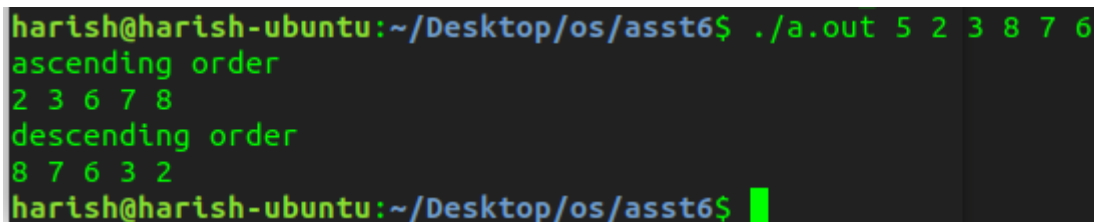
int num;
int arr[2][30];
int main(int argc, char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=num+2) {printf("invalid input\n");exit(0);}
    for(int i=0;i<num;++i)
    {
        arr[0][i]=atoi(argv[i+2]);
        arr[1][i]=atoi(argv[i+2]);
    }
    int order[]={1,0};//1-asc 0-desc
    pthread_t tid[2];
    pthread_attr_t attr[2];
    pthread_attr_init(&attr[0]);
    pthread_attr_init(&attr[1]);
    pthread_create(&tid[0],&attr[0],runner,&order[0]);//ascending sort
    pthread_create(&tid[1],&attr[1],runner,&order[1]);//descending sort
    pthread_join(tid[0],NULL);
    pthread_join(tid[1],NULL);
    return 0;
}

void *runner (void *param)
{
    int *order=(int *)param;
    for (int i = 1; i < num; i++)
    {
        int key = arr[*order][i];
        int j = i - 1;
        while (j >= 0 && ((1-*order)*(key-arr[*order][j])+
(*order)*(arr[*order][j]-key))>0)
        {
            arr[*order][j + 1] = arr[*order][j];
            j = j - 1;
        }
    }
}

```

```
arr[*order][j + 1] = key;
}
char ord[15];
if(*order==1)
strcpy(ord, "ascending");
else
strcpy(ord, "descending");
printf("%s order\n", ord);
for(int i=0; i<num; ++i)
printf("%d ", arr[*order][i]);
printf("\n");
pthread_exit(0);
}
```

Output:



```
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 5 2 3 8 7 6
ascending order
2 3 6 7 8
descending order
8 7 6 3 2
harish@harish-ubuntu:~/Desktop/os/asst6$
```

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)

Approach:

Setting a variable MAX_THREADS which is the number of threads used at a time. For every iteration, MAX_THREADS number of threads are created and runner function is called. The runner function divides the array into MAX_THREADS number of parts and each part is searched parallelly by the threads created by binary search function.

Code:

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

```

#include<unistd.h>
#include<sys/wait.h>
#include<pthread.h>
#include<math.h>
#define MAX_THREAD 4
void *runner(void *param);
int arr[30],start=0,end,key,num;
int part=0;

void binary_search (int start, int end, int key,int part){
int mid = (end + start)/2;
if(start > end){
return;
}
if(start == end){
if(arr[mid] == key){
printf("Key found at address: %d by thread %d\n", mid,part);
}
return;
}

if(start+1 == end){
binary_search(start, start, key,part);
binary_search(end, end, key,part);
}
if(arr[mid] == key){
printf("Key found at address: %d by thread %d\n", mid,part);
}
pid_t pid = fork();
if(pid == 0){
binary_search(start, mid-1, key,part);
}
else{
binary_search(mid+1, end, key,part);
}
}

```

```

int main(int argc, char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=num+3) {printf("invalid input\n");exit(0);}
    for(int i=0;i<num;++i)
    {
        arr[i]=atoi(argv[i+2]);
    }
    key=atoi(argv[num+2]);
    end=num-1;
    pthread_t threads[MAX_THREAD];

    if(num/MAX_THREAD<=1)
    {
        printf("more number of threads!");
        exit(0);
    }
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_create(&threads[i], NULL,runner, (void*)NULL);
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_join(threads[i], NULL);
    return 0;
}

void* runner(void* param)
{
    // Each thread checks 1/MAX_THREADS of the array for the key
    int thread_part = part++;
    int mid;
    int low = thread_part * (num / MAX_THREAD);
    int high = (thread_part + 1) * (num / MAX_THREAD);

    binary_search(low,high,key,part);
    pthread_exit(0);
}

```

Output:

```
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 6 9 4 2 7 6 1 4
Key found at address: 1 by thread 1
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 6 9 4 2 7 6 1 6
Key found at address: 4 by thread 2
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 8 9 4 2 7 6 1 7 9 9
Key found at address: 0 by thread 1
Key found at address: 0 by thread 1
Key found at address: 7 by thread 2
Key found at address: 7 by thread 2
```

Approach 2:

A structure is declared which has start,end,array and search key as members. The main function calls the binary search code, which searches for key by dividing the array using mid calculation. Each of 2 halves is given to 2 different threads and recursion happens using the passed structure variable as a parameter. Once element is found it prints the position.

Code:

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

struct arguments{
int arr[20];
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]);

    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);
    }

    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;
    }
}

```

```

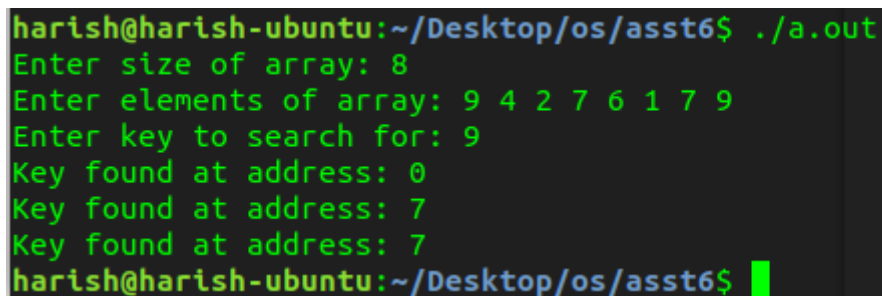
}
if(args->start == args->end){
if(args->arr[mid] == args->key){
printf("Key found at address: %d\n", mid);
}
return NULL;
}
if(args->start+1 == args->end){

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] == args->key){
printf("Key found at address: %d\n", mid);
}
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++;
}

```

Output:



```

harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out
Enter size of array: 8
Enter elements of array: 9 4 2 7 6 1 7 9
Enter key to search for: 9
Key found at address: 0
Key found at address: 7
Key found at address: 7
harish@harish-ubuntu:~/Desktop/os/asst6$

```

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

Approach:

Setting a variable MAX_THREADS which is the number of threads used at a time. For every iteration, MAX_THREADS number of threads are created and runner function is called. From i=2 to the number specified by user, this happens. The runner code checks if the number passed as argument is prime and prints the number if it is a prime.

Code:

```
#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
#define MAX_THREADS 4
void *runner(void* param);
int num;
int main(int argc, char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=2) {printf("invalid input\n");exit(0);}

    for(int i=2;i<num;i+=MAX_THREADS)
    {
        pthread_t tid[MAX_THREADS];
        int temp[MAX_THREADS];
        for(int j=0;j<MAX_THREADS&& j<num-i;++j)
            temp[j]=i+j;
        for(int j=0;j<MAX_THREADS&& j<num-i;j++)
        {
            pthread_create(&tid[j], NULL, runner, &temp[j]);
        }
        for(int j=0;j<MAX_THREADS&& j<num-i;j++)
            pthread_join(tid[j], NULL);
    }
}
```

```

return 0;
}

void *runner(void* param)
{
int *n=(int*)param;
int flag=1;
for(int i=2;i<*n;++i)
{
if(*n%i==0)
{
flag=0;
pthread_exit(0);
}
}
if(flag==1)
printf("%d\n", *n);
pthread_exit(0);
}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 25
2
3
5
7
13
11
17
19
23
harish@harish-ubuntu:~/Desktop/os/asst6$

```

Approach:

Code:

Output:

6. Implemen

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

Approach:

Three threads are created where each one calculates one of mean, median or mode according to the value passed to the runner function. Runner function uses switch case

to decide which one of three to perform. And value of mean, median and mode is printed in runner itself.

Code:

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

void *runner(void* param);
int num,arr[20];

void swap(int *p,int *q)
{
    int t;
    t=*p;
    *p=*q;
    *q=t;
}

void sort(int a[],int n)
{
    int i,j,temp;

    for(i = 0;i < n-1;i++) {
        for(j = 0;j < n-i-1;j++) {
            if(a[j] > a[j+1])
                swap(&a[j],&a[j+1]);
        }
    }
}

int main(int argc,char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=num+2) {printf("invalid input\n");exit(0);}
    for(int i=0;i<num;++i)
```

```

{
arr[i]=atoi(argv[i+2]);
}
sort(arr,num);
pthread_t tid[3];
int temp[]={0,1,2};
pthread_create(&tid[0],NULL,runner,&temp[0]);
pthread_create(&tid[1],NULL,runner,&temp[1]);
pthread_create(&tid[2],NULL,runner,&temp[2]);

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

return 0;

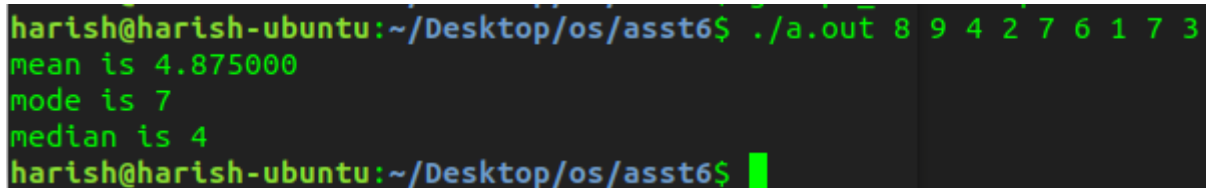
}

void *runner(void * param)
{
int *n=(int*)param;
switch(*n)
{
case 0: {float sum=0;
for(int i=0;i<num;++i)
sum+=arr[i];
float mean=sum/num;
printf("mean is %f\n",mean);
break;}
case 1: {int median=arr[(num+1) / 2 - 1];
printf("median is %d\n",median);
break;}
case 2: {int t=arr[num-1]+1;
int *count=(int *)malloc(t*sizeof(int));
for (int i = 0; i < t; i++)
count[i] = 0;
for (int i = 0; i < num; i++)

```

```
count[arr[i]]++;  
int mode = 0;  
int k = count[0];  
for (int i = 1; i < t; i++)  
{  
    if (count[i] > k)  
    {  
        k = count[i];  
        mode = i;  
    }  
}  
printf("mode is %d\n",mode);  
break;}  
default: break;  
}  
pthread_exit(0);  
}
```

Output:



```
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 8 9 4 2 7 6 1 7 3  
mean is 4.875000  
mode is 7  
median is 4  
harish@harish-ubuntu:~/Desktop/os/asst6$
```

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

Merge sort:

Approach:

The merge sort code is parallelized by:

- During the divide phase of merge sort, each half of the divided array is assigned to one thread.
- Runner function calls the mergesort code again using the values passed by the structure variable as parameters.
- During the conquer phase, threads get joined.

Code:

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

void *runner(void* param);
int num,arr[20];
struct passing
{
int b,e;
};

void merge(int arr[], int l, int m, int r)
{
int i, j, k;
int n1 = m - l + 1;
int n2 = r - m;
/* 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[l + i];
for (j = 0; j < n2; j++)
R[j] = arr[m + 1 + j];
/* Merge the temp arrays back into arr[l..r]*/
i = 0; // Initial index of first subarray
j = 0; // Initial index of second subarray
k = l; // 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++;
}
}
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
if (l < r) {
// Same as (l+r)/2, but avoids overflow for
// large l and h
int m = l + (r - l) / 2;

struct passing p,q;
p.b=l;p.e=m;
q.b=m+1;q.e=r;
pthread_t tid[2];
pthread_create(&tid[0],NULL,runner,&p);
pthread_create(&tid[1],NULL,runner,&q);
pthread_join(tid[0],NULL);
pthread_join(tid[1],NULL);
merge(arr, l, m, r);

```

```

}
}
/* UTILITY FUNCTIONS */
/* Function to print an array */
void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main(int argc, char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=num+2) {printf("invalid input\n");exit(0);}
    for(int i=0;i<num;++i)
    {
        arr[i]=atoi(argv[i+2]);
    }

    printf("Given array is \n");
    printArray(arr, num);

    mergeSort(arr, 0, num - 1);
    printf("\nSorted array is \n");
    printArray(arr, num);
    return 0;
}

void *runner(void* param)
{
    struct passing *p=(struct passing*)param;
    mergeSort(arr,p->b,p->e);
}

```

Output:

```
harish@harish-ubuntu:~/Desktop/os/asst6$ gcc p6_merge.c -pthread
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 8 9 4 2 7 6 1 7 3
Given array is
9 4 2 7 6 1 7 3
Sorted array is
1 2 3 4 6 7 7 9
number of threads used:14
harish@harish-ubuntu:~/Desktop/os/asst6$
```

Quick sort:

Approach:

The quick sort code is parallelized by:

- During the divide phase of quick sort, each half of divided array is assigned to one thread.
- Runner function calls the quicksort code again using the values passed by the structure variable as parameters.
- During the conquer phase threads get joined.

Code:

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

void *runner(void* param);
int num, arr[20], k, tcount=0;
struct passing
{
int b,e,k;
};

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 passing 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);

    }
}

int main(int argc,char *argv[])
{
    num=atoi(argv[1]);
    if(argc!=num+2) {printf("invalid input\n");exit(0);}
}

```

```

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

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

printf("\nSorted array:\n");

for(int i=0;i<k;++i)
printf("%d ",arr[i]);
printf("\n");
printf("Number of threads used:%d\n",tcount);
return 0;
}

void *runner(void* param)
{
struct passing* p=(struct passing*)param;
//printf("%d ",++tcount);
++tcount;
quicksort(arr,p->b,p->e,p->k);
pthread_exit(0);
}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst6$ gcc p6_quick.c -pthread
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 8 9 4 2 7 6 1 7 3

Sorted array:
1 2 3 4 6 7 7 9
Number of threads used:10
harish@harish-ubuntu:~/Desktop/os/asst6$ █

```

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

Approach:

(Pi value generation is explained in comments)

Multithreading is done by:

- Setting a variable MAX_THREADS which is the number of threads used at a time. For every iteration, MAX_THREADS number of threads are created and runner function is called. The function calculates pi value by random value generation.
- In total MAX_THREADS*INTERVAL*INTERVAL number of threads are created and in every iteration MAX_THREADS number of threads run in parallel.

Code:

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

#define MAX_THREADS 4
#define INTERVAL 100
int interval, i;
double rand_x, rand_y, origin_dist, pi;
int circle_points = 0, square_points = 0;

void *runner(void* param);

int main(int argc, char *argv[])
{
    srand(time(NULL));
    for (i = 0; i < (INTERVAL * INTERVAL); i+=MAX_THREADS)
    {
        pthread_t tid[MAX_THREADS];
        //int temp[MAX_THREADS]={0};
        for(int j=0; j<MAX_THREADS; ++j)
            pthread_create(&tid[j], NULL, runner, NULL);
        for(int j=0; j<MAX_THREADS; ++j)
```

```

pthread_join(tid[j], NULL);
}
printf("pi value is %lf\n", pi);
return 0;
}

void* runner(void* param)
{
rand_x = (double)(rand() % (INTERVAL + 1)) / INTERVAL;
rand_y = (double)(rand() % (INTERVAL + 1)) / INTERVAL;
// 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++;
// estimated pi after this iteration
pi = (double)(4 * circle_points) / square_points;
}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst6$ gcc p7_pi.c -pthread
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out
interval is 200
number of threads 10
pi value is 3.132700
harish@harish-ubuntu:~/Desktop/os/asst6$ gcc p7_pi.c -pthread
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out
interval is 300
number of threads 10
pi value is 3.129022
harish@harish-ubuntu:~/Desktop/os/asst6$ █

```

Optional:

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

Approach:

(The code for inverse calculation is explained in comments)

The inverse code is multithreaded in the following way:

- To calculate inverse we need adjoint and for adjoint we require cofactor of every element of matrix.
- Cofactor finding is multithreaded here. For a matrix of order N, N number of threads are created and each thread takes care of cofactor of one element in a row.
- In N iterations N*N threads are created and cofactor of all the elements of matrix are calculated.

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;
// Looping for each element of the matrix
for (int row = 0; row < n; row++)
{
```



```

for (int col = 0; col < n; col++)
{
    // Copying into temporary matrix only those element
    // which are not in given row and column
    if (row != p && col != q)
    {
        temp[i][j++] = A[row][col];
        // Row is filled, so increase row index and
        // reset col index
        if (j == n - 1)
        {
            j = 0;
            i++;
        }
    }
}

/* Recursive function for finding determinant of matrix.
n is current dimension of A[][]. */
int determinant(int A[N][N], int n)
{
    int D = 0; // Initialize result
    // Base case : if matrix contains single element
    if (n == 1)
        return A[0][0];
    int temp[N][N]; // To store cofactors
    int sign = 1; // To store sign multiplier
    // Iterate for each element of first row
    for (int f = 0; f < n; f++)
    {
        // Getting Cofactor of A[0][f]
        getCofactor(A, temp, 0, f, n);
        D += sign * A[0][f] * determinant(temp, n - 1);
        // terms are to be added with alternate sign
        sign = -sign;
    }
}

```

```

return D;
}
// Function to get adjoint of A[N][N] in adj[N][N].
void adjoint(int A[N][N],int adj[N][N])
{
if (N == 1)
{
adj[0][0] = 1;
return;
}
// temp is used to store cofactors of A[][]
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++)
{
// Get cofactor of A[i][j]
p[j].p=i;p[j].q=j;p[j].n=N;
pthread_create(&tid[j],NULL,runner,&p[j]);
//getCofactor(A, temp, i, j, N);
}
for (int j=0; j<N; j++)
{
// sign of adj[j][i] positive if sum of row
// and column indexes is even.
pthread_join(tid[j],NULL);
sign = ((i+j)%2==0)? 1: -1;
// Interchanging rows and columns to get the
// transpose of the cofactor matrix
adj[j][i] = (sign)*(determinant(p[j].temp, N-1));
}

}
}
// Function to calculate and store inverse, returns false if

```

```

// matrix is singular
bool inverse(int A[N][N], float inverse[N][N])
{
    // Find determinant of A[][]
    int det = determinant(A, N);
    if (det == 0)
    {
        printf("Singular matrix, can't find its inverse");
        return false;
    }
    // Find adjoint
    int adj[N][N];
    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;
}

// Generic function to display the matrix. We use it to display
// both adjoin and inverse. adjoin is integer matrix and inverse
// is a float.
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 disinv(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");
}
}
// Driver program
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))
disinv(inv);

return 0;
}

void* runner(void* param)
{
struct passing *p=(struct passing*)param;
getCofactor(A, p->temp, p->p,p->q, p->n);
pthread_exit(0);
}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out
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

```

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.

Approach:

Fibonacci series generation must be done in series fashion because $f(n)$ depends on $f(n-1)$ and $f(n-2)$. So to calculate n th term we need $n-1$ th and $n-2$ nd term.

To implement it in multithreaded way I assigned $\text{fib}(n-1)$ and $\text{fib}(n-2)$ calculation to 2 different threads. The sum is made in main and $\text{fib}(n)$ is displayed. This process is done for every number from $n=2$ to $n=\text{num}-1$.

Code:

```

#include<stdio.h>
#include<stdlib.h>
#include<pthread.h>
void *runner(void* param);
int fib[2];
int threadno=-1;
int num;
int main(int argc, char *argv[])

```

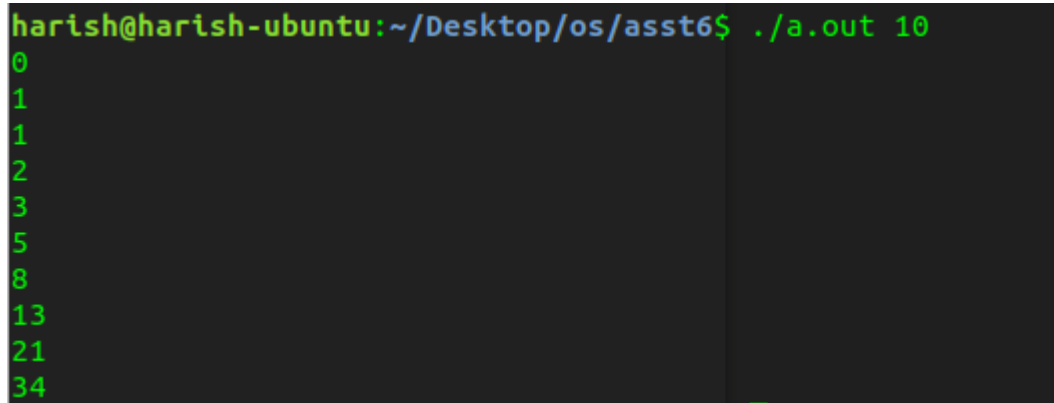
```

{
num=atoi(argv[1]);
if(argc!=2) {printf("invalid input\n");exit(0);}
printf("0\n1\n");
for(int i=2;i<num;++i)
{
pthread_t tid1,tid2;
pthread_attr_t attr;
pthread_attr_init(&attr);
int temp1=i-1,temp2=i-2;
pthread_create(&tid1,NULL,runner,&temp1);
pthread_create(&tid2,NULL,runner,&temp2);
pthread_join(tid1,NULL);
pthread_join(tid2,NULL);
printf("%d\n",fib[0]+fib[1]);
threadno=-1;
}
return 0;
}
void *runner(void* param)
{
int *n=(int*)param;
threadno++;
//printf("%d$", *n);
int a = 0, b = 1, c, i;
if( *n == 0)
fib[threadno]=0;
else
{for (i = 2; i <= *n; i++)
{
c = a + b;
a = b;
b = c;
}
fib[threadno]=b;
}
//printf("%d %d**", threadno, fib[threadno]);

```

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

Output:



```
harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out 10  
0  
1  
1  
2  
3  
5  
8  
13  
21  
34
```

10. Longest common sub sequence generation problem using threads.

Approach:

The program can be consists of 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.

The second part is multithreaded.

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`.

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 subseq(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 subseqs of str1
subseq(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 subseqs
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 subseq(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);
subseq(sb);
}
}
}
}
}

```

Output:

```

harish@harish-ubuntu:~/Desktop/os/asst6$ ./a.out frewtr ttrewuyt
Longest Common Subsequence: rewtr
harish@harish-ubuntu:~/Desktop/os/asst6$

```