

OPERATING SYSTEMS - COM301P

LAB ASSIGNMENT - 6

MULTI-THREADING

VIKNESH RAJARAMON

COE18B060

1) Generate Armstrong number generation within a range.

LOGIC:

- We create two structures - **thread_node** and **node**.
- **thread_node** structure stores the **lower bound** and the **upper bound of the range**.
- **node** structure stores the **current number** and the **sum of digits raised to the power of the number of digits**.
- We create a thread for every number in the range..
- **get_sum** function calculates the **sum of digits raised to the power of the number of digits** and stores it in struct node. In the join section, we check if the number is equal to the sum.

C CODE:

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

struct thread_node
{
    int lower;
    int upper;
};

struct node
{
    int num;
    int sum_of_digits;
};

void *get_sum(void *param)
{
    struct node *data=param;
    int sum,x,digits;
    x=data->num;
    sum=0;
    digits=(int) log10(x)+1;
    while(x)
    {
        int r=x%10;
        x/=10;
        sum+=pow(r,digits);
    }
    data->sum_of_digits=sum;
}

void *Armstrong(void *ptr)
{
    pthread_t tid;
    pthread_attr_t attr;
    struct thread_node *data=ptr;
```

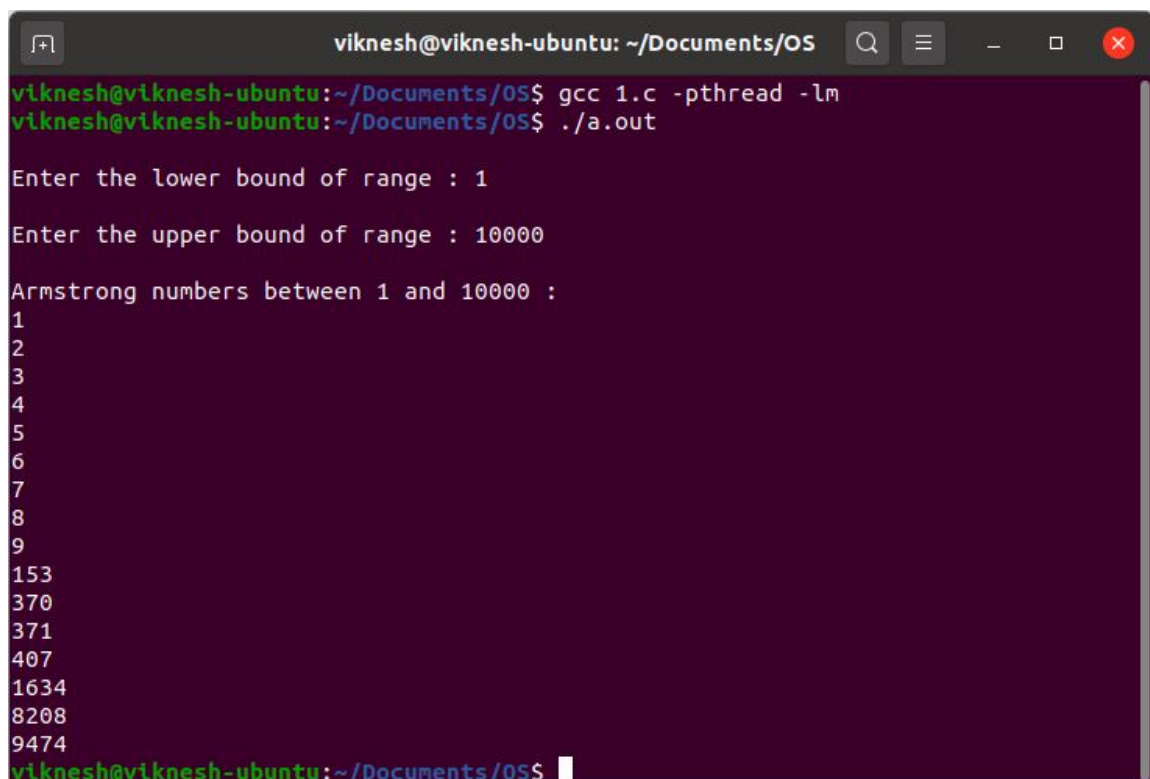
```

struct node *temp=(struct node *)malloc(sizeof(struct node));
for(int i=data->lower;i<=data->upper;++i)
{
    temp->num=i;
    temp->sum_of_digits=0;
    pthread_create(&tid,&attr,get_sum,temp);
    pthread_join(tid,NULL);
    if(temp->num==temp->sum_of_digits)
    {
        printf("%d\n",temp->num);
    }
}
}

int main()
{
    pthread_t tid;
    pthread_attr_t attr;
    int a,b;
    int sum=0,x=0;
    int digits=0;
    printf("\nEnter the lower bound of range : ");
    scanf("%d",&a);
    printf("\nEnter the upper bound of range : ");
    scanf("%d",&b);
    printf("\nArmstrong numbers between %d and %d : \n",a,b);
    struct thread_node *data=(struct thread_node *)malloc(sizeof(struct thread_node));
    data->lower=a;
    data->upper=b;
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,Armstrong,data);
    pthread_join(tid,NULL);
    return 0;
}

```

OUTPUT:



```

viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 1.c -pthread -lm
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the lower bound of range : 1

Enter the upper bound of range : 10000

Armstrong numbers between 1 and 10000 :
1
2
3
4
5
6
7
8
9
153
370
371
407
1634
8208
9474
viknesh@viknesh-ubuntu:~/Documents/OS$

```

2) Ascending Order sort and Descending order sort.

LOGIC:

- We get the number of elements from the user and randomly generate the array.
- We create two threads - **Thread 1** and **Thread 2**.
- **Thread 1** sorts the array in ascending order whereas **Thread 2** sorts the array in descending order.

C CODE:

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

struct node
{
    int num;
    int *array;
};

int ascending(const void *a,const void *b)
{
    return (*(int *)a - *(int *)b);
}

int descending(const void *a,const void *b)
{
    return (*(int *)b - *(int *)a);
}

void *asc_sort(void *param)
{
    struct node *data=param;
    qsort(data->array,data->num,sizeof(int),ascending);
}

void *des_sort(void *param)
{
    struct node *data=param;
    qsort(data->array,data->num,sizeof(int),descending);
}

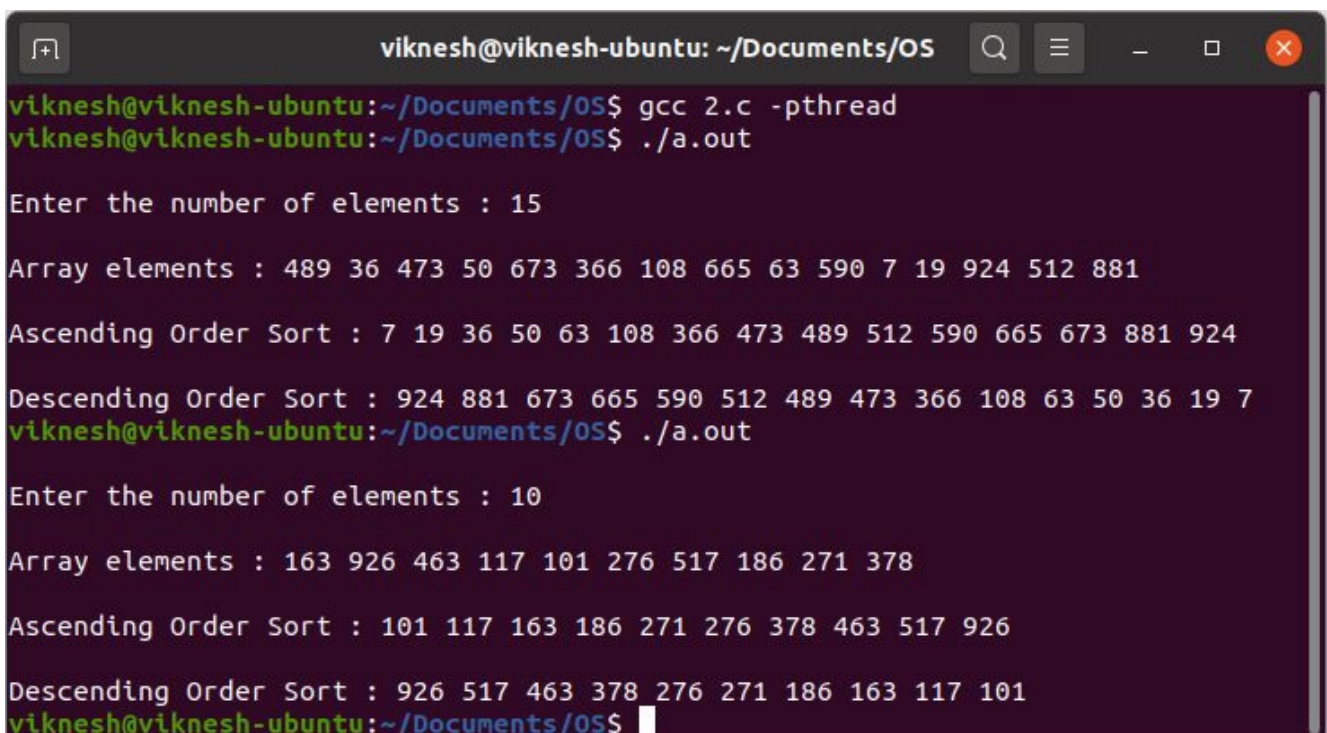
int main()
{
    srand(time(0));
    int n;
    printf("\nEnter the number of elements : ");
    scanf("%d",&n);
    int a[n];
    printf("\nArray elements : ");
    for(int i=0;i<n;++i)
    {
        a[i]=rand()%1000 + 1;
        printf("%d ",a[i]);
    }
}
```

```

}
printf("\n");
pthread_t tid[2];
pthread_attr_t attr[2];
pthread_attr_init(&attr[0]);
pthread_attr_init(&attr[1]);
struct node *data1=(struct node *)malloc(sizeof(struct node));
struct node *data2=(struct node *)malloc(sizeof(struct node));
data1->num=data2->num=n;
data1->array=(int *)malloc(sizeof(int)*n);
data2->array=(int *)malloc(sizeof(int)*n);
for(int i=0;i<n;++i)
{
    data1->array[i]=a[i];
    data2->array[i]=a[i];
}
pthread_create(&tid[0],&attr[0],asc_sort,data1);
pthread_create(&tid[1],&attr[1],des_sort,data2);
pthread_join(tid[0],NULL);
pthread_join(tid[1],NULL);
printf("\nAscending Order Sort : ");
for(int i=0;i<n;++i)
{
    printf("%d ",data1->array[i]);
}
printf("\n");
printf("\nDescending Order Sort : ");
for(int i=0;i<n;++i)
{
    printf("%d ",data2->array[i]);
}
printf("\n");
return 0;
}

```

OUTPUT:



```

viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 2.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 15

Array elements : 489 36 473 50 673 366 108 665 63 590 7 19 924 512 881

Ascending Order Sort : 7 19 36 50 63 108 366 473 489 512 590 665 673 881 924

Descending Order Sort : 924 881 673 665 590 512 489 473 366 108 63 50 36 19 7
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 10

Array elements : 163 926 463 117 101 276 517 186 271 378

Ascending Order Sort : 101 117 163 186 271 276 378 463 517 926

Descending Order Sort : 926 517 463 378 276 271 186 163 117 101
viknesh@viknesh-ubuntu:~/Documents/OS$

```

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 (duplicates elements search as well)

LOGIC:

- We get the number of elements from the user and randomly generate the array.
- We create two threads - **Thread 1** and **Thread 2**.
- **Thread 1 sorts the first half of the array and finds the first occurrence and last occurrence of the search element.**
- **Thread 2 sorts the second half of the array and finds the first occurrence and last occurrence of the search element.**
- **Number of occurrences** is calculated as the **difference between last and first occurrence**.
- The number of occurrences of the search element in both halves are displayed separately.

C CODE:

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

struct node
{
    int num;
    int *array;
    int x;
    int count;
};

int sort(const void *a,const void *b)
{
    return (*(int *)a - *(int *)b);
}

int first_occurrence(int array[],int lower,int upper,int x)
{
    if(lower>upper)
    {
        return lower;
    }
    int mid=lower+(upper-lower)/2;
    if(array[mid]>=x)
    {
        return first_occurrence(array,lower,mid-1,x);
    }
    return first_occurrence(array,mid+1,upper,x);
}

int last_occurrence(int array[],int lower,int upper,int x)
{
    if(lower>upper)
    {
        return lower;
    }
```

```

    }
    int mid=lower+(upper-lower)/2;
    if(array[mid]>x)
    {
        return last_occurrence(array,lower,mid-1,x);
    }
    return last_occurrence(array,mid+1,upper,x);
}

void *binary_search(void *ptr)
{
    struct node *data=ptr;
    qsort(data->array,data->num,sizeof(int),sort);
    int last_pos=last_occurrence(data->array,0,data->num-1,data->x);
    int first_pos=first_occurrence(data->array,0,data->num-1,data->x);
    data->count=last_pos-first_pos;
}

int main()
{
    srand(time(0));
    int n,x;
    printf("\nEnter the number of elements : ");
    scanf("%d",&n);
    int array[n];
    printf("\nUnsorted Array elements : ");
    for(int i=0;i<n;++i)
    {
        array[i]=rand()%10+1;
        printf("%d ",array[i]);
    }
    printf("\n\nEnter the element to search for : ");
    scanf("%d",&x);
    pthread_t tid[2];
    struct node arg[2];
    for(int i=0;i<2;++i)
    {
        arg[i].num=n/2+((n%2)*i);
        arg[i].array=array+(i*n/2);
        arg[i].x=x;
        arg[i].count=0;
    }
    for(int i=0;i<2;++i)
    {
        pthread_create(&tid[i],NULL,binary_search,&arg[i]);
    }
    for(int i=0;i<2;++i)
    {
        pthread_join(tid[i],NULL);
        printf("\nOccurrence of %d in Half %d = %d\n",x,i+1,arg[i].count);
    }
    return 0;
}

```

OUTPUT:

```
viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 3.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 10

Unsorted Array elements : 3 3 1 4 5 1 4 8 9 4

Enter the element to search for : 4

Occurrence of 4 in Half 1 = 1

Occurrence of 4 in Half 2 = 2
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 35

Unsorted Array elements : 8 3 1 5 8 8 3 1 7 4 7 8 7 7 2 9 5 7 9 3 5 6 1 10 9 9 8
3 6 1 2 3 3 4 9

Enter the element to search for : 8

Occurrence of 8 in Half 1 = 4

Occurrence of 8 in Half 2 = 1
viknesh@viknesh-ubuntu:~/Documents/OS$
```

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

LOGIC:

- We get the limit from the user through the command line.
- We create four threads - **Thread 1, Thread 2, Thread 3** and **Thread 4**.
- We also use a **static variable** and **keep incrementing the value** so as to **check whether the current value is a prime number or not**.
- If the number is a prime number, we **keep the value as 1** in that **index**.
- We get the input string from the user and **write the string to the fd_parent pipe in the parent process**.
- We **read the string from the fd_parent pipe** and **reverse the string in the child process**.
- Once the string is reversed, we **write the string back to the parent process using the fd_child pipe**.
- Finally, when all threads have finished execution, we join all the threads and display the prime numbers.

C CODE:

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

#define MAX_THREADS 4

static int x=0;
```



```

struct thread_node
{
    int *prime_array;
    int num;
};

void *prime(void *ptr)
{
    struct thread_node *data=ptr;
    int i=x;
    ++x;
    int flag;
    while(i<data->num)
    {
        flag=0;
        for(int j=2;j<=i/2;++j)
        {
            if(i%j==0)
            {
                flag=1;
                break;
            }
        }
        if((flag==0) && (i>1))
        {
            data->prime_array[i]=1;
        }
        i+=MAX_THREADS;
    }
}

int main(int argc,char *argv[])
{
    if(argc!=2)
    {
        printf("\nTry ./a.out <number>\n");
        exit(0);
    }
    int n=atoi(argv[1]);
    pthread_t tid[MAX_THREADS];
    pthread_attr_t attr[MAX_THREADS];
    struct thread_node *data=(struct thread_node *)malloc(sizeof(struct thread_node));
    data->prime_array=(int *)malloc(sizeof(int)*n);
    data->num=n;
    struct node *ptr[n];
    for(int i=0;i<MAX_THREADS;++i)
    {
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i],&attr[i],prime,data);
    }
    for(int i=0;i<MAX_THREADS;++i)
    {
        pthread_join(tid[i],NULL);
    }
    printf("\nPrime Numbers\n\n");
    for(int i=0;i<n;++i)
    {

```

```

        if(data->prime_array[i]==1)
        {
            printf("%d ",i);
        }
    }
    printf("\n");
    return 0;
}

```

OUTPUT:

```

viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 4.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 10

Prime Numbers

2 3 5 7
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 100

Prime Numbers

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 200

Prime Numbers

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 1000

Prime Numbers

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 897 907 911 919 929 937 941 947 953 967 971 977 983 991 997
viknesh@viknesh-ubuntu:~/Documents/OS$

```

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

LOGIC:

- We get the number of elements from the user and randomly generate the array.
- Next, we sort the array.
- We create three threads - **Thread 1**, **Thread 2** and **Thread 3**.
- **Thread 1** calculates the **mean** of the array.
- **Thread 2** calculates the **median** of the array.
- **Thread 3** calculates the **mode** of the array.

C CODE:

```

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

```

```

#include<pthread.h>
#include<unistd.h>
#include<time.h>

struct node
{
    int num;
    int *array;
};

int sort(const void *a,const void *b)
{
    return (*(int *)a - *(int *)b);
}

void *cal_mean(void *param)
{
    struct node *data=param;
    int sum=0;
    int n=data->num;
    for(int i=0;i<n;++i)
    {
        sum+=data->array[i];
    }
    float mean=((float)sum)/n;
    printf("\nMean = %f\n",mean);
}

void *cal_median(void *param)
{
    struct node *data=param;
    int n=data->num;
    int mid=n/2;
    float median;
    if(n%2!=0)
    {
        median=data->array[mid];
    }
    else
    {
        median=((float)data->array[mid-1]+(float)data->array[mid])/2;
    }
    printf("\nMedian = %f\n",median);
}

void *cal_mode(void *param)
{
    struct node *data=param;
    int n=data->num;
    int max_count=0,mode=0,count=0;
    for(int i=0;i<n;++i)
    {
        count=0;
        for(int j=0;j<n;++j)
        {
            if(data->array[i]==data->array[j])
            {

```

```

        ++count;
    }
}
if(count>max_count)
{
    max_count=count;
    mode=data->array[i];
}
}
printf("\nMode = %d\n",mode);
}

int main()
{
    srand(time(0));
    int n;
    printf("\nEnter the number of elements : ");
    scanf("%d",&n);
    int a[n];
    for(int i=0;i<n;++i)
    {
        a[i]=rand()%10 + 1;
    }
    qsort(a,n,sizeof(int),sort);
    printf("\nArray elements : ");
    for(int i=0;i<n;++i)
    {
        printf("%d ",a[i]);
    }
    printf("\n");
    pthread_t tid[3];
    pthread_attr_t attr[3];
    for(int i=0;i<3;++i)
    {
        pthread_attr_init(&attr[i]);
    }
    struct node *data=(struct node *)malloc(sizeof(struct node));
    data->num=n;
    data->array=(int *)malloc(sizeof(int)*n);
    for(int i=0;i<n;++i)
    {
        data->array[i]=a[i];
    }
    pthread_create(&tid[0],&attr[0],cal_mean,data);
    pthread_create(&tid[1],&attr[1],cal_median,data);
    pthread_create(&tid[2],&attr[2],cal_mode,data);
    for(int i=0;i<3;++i)
    {
        pthread_join(tid[i],NULL);
    }
    return 0;
}

```

OUTPUT:

```
viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 5.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 20

Array elements : 1 1 1 1 3 3 3 4 4 4 4 4 6 7 7 8 8 8 8 10

Mean = 4.750000
Median = 4.000000
Mode = 4
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the number of elements : 30

Array elements : 1 1 1 2 3 3 3 3 3 3 3 4 4 4 4 4 5 5 5 5 5 5 5 6 6 7 8 10 10 10

Mean = 4.600000
Median = 4.000000
Mode = 3
viknesh@viknesh-ubuntu:~/Documents/OS$
```

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

MERGE SORT

LOGIC:

- We get the number of elements from the user and randomly generate the array.
- We split the array into N parts, N being the number of threads.
- We perform Merge sort for each part separately and merge all the parts at the end.

C CODE:

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

#define MAX_THREADS 4

struct node
{
    int num;
    int *array;
    int split[5];
};

static int i=0;
```

```

void merge(struct node *data,int low,int mid,int high)
{
    int size=high-low+1;
    int array[size];
    int i=low;
    int j=mid+1;
    int k=0;
    while(i<=mid && j<=high)
    {
        if(data->array[i]<data->array[j])
        {
            array[k]=data->array[i];
            ++i;
        }
        else
        {
            array[k]=data->array[j];
            ++j;
        }
        ++k;
    }
    while(i<=mid)
    {
        array[k]=data->array[i];
        ++i;
        ++k;
    }
    while(j<=high)
    {
        array[k]=data->array[j];
        ++j;
        ++k;
    }
    j=low;
    for(int i=0;i<size;++i)
    {
        data->array[j]=array[i];
        ++j;
    }
}

```

```

void MERGE_SORT(struct node *data,int low,int high)
{
    int mid=low+(high-low)/2;
    if(low<high)
    {
        MERGE_SORT(data,low,mid);
        MERGE_SORT(data,mid+1,high);
        merge(data,low,mid,high);
    }
}

```

```

void *merge_sort(void *ptr)
{
    struct node *data=ptr;
    int low=data->split[i]+1;
    int high=data->split[i+1];
}

```

```

    ++i;
    int mid=low+(high-low)/2;
    if(low<high)
    {
        MERGE_SORT(data,low,mid);
        MERGE_SORT(data,mid+1,high);
        merge(data,low,mid,high);
    }
}

int main()
{
    srand(time(0));
    int n;
    printf("\nEnter the size of array : ");
    scanf("%d",&n);
    struct node *data=(struct node *)malloc(sizeof(struct node));
    data->num=n;
    data->array=(int *)malloc(sizeof(int)*n);
    printf("\nArray elements before sorting : ");
    for(int i=0;i<n;++i)
    {
        data->array[i]=rand()%(5*n)+1;
        printf("%d ",data->array[i]);
    }
    printf("\n");
    pthread_t tid[MAX_THREADS];
    pthread_attr_t attr[MAX_THREADS];
    data->split[4]=data->num-1;
    data->split[2]=(data->split[4]-data->split[0])/2;
    data->split[1]=data->split[0]+(data->split[2]-data->split[0])/2;
    data->split[3]=data->split[2]+(data->split[4]-data->split[2])/2;
    data->split[0]=-1;
    for(int i=0;i<MAX_THREADS;++i)
    {
        pthread_attr_init(&attr[i]);
        pthread_create(&tid[i],&attr[i],merge_sort,data);
    }
    for(int i=0;i<MAX_THREADS;++i)
    {
        pthread_join(tid[i],NULL);
    }
    merge(data,0,data->split[1],data->split[2]);
    merge(data,data->split[2]+1,data->split[3],data->split[4]);
    merge(data,0,data->split[2],data->split[4]);
    printf("\nArray elements after sorting : ");
    for(int i=0;i<n;++i)
    {
        printf("%d ",data->array[i]);
    }
    printf("\n");
    return 0;
}

```

OUTPUT:

```
viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 6a.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 10

Array elements before sorting : 31 19 4 18 20 10 3 21 20 38

Array elements after sorting : 3 4 10 18 19 20 20 21 31 38
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 20

Array elements before sorting : 17 23 5 1 9 95 78 89 96 64 98 58 2 14 100 59 80
37 47 63

Array elements after sorting : 1 2 5 9 14 17 23 37 47 58 59 63 64 78 80 89 95 96
98 100
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 28

Array elements before sorting : 68 23 117 133 134 27 9 117 39 87 20 71 49 45 13
127 100 6 13 28 32 20 140 80 93 106 9 55

Array elements after sorting : 6 9 9 13 13 20 20 23 27 28 32 39 45 49 55 68 71 8
0 87 93 100 106 117 117 127 133 134 140
viknesh@viknesh-ubuntu:~/Documents/OS$
```

QUICK SORT

LOGIC:

- We get the number of elements from the user and randomly generate the array.
- We first partition the array with respect to the pivot element.
- We sort the two halves parallelly using threads.

C CODE:

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

struct node
{
    int *array;
    int low;
    int high;
};

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



```

        *b=temp;
    }

int partition(struct node *data,int low,int high)
{
    int pivot_element=data->array[high];
    int i=low-1;
    for(int j=low;j<high;++j)
    {
        if(data->array[j]<pivot_element)
        {
            ++i;
            swap(&data->array[i],&data->array[j]);
        }
    }
    swap(&data->array[i+1],&data->array[high]);
    return (i+1);
}

void Quick_Sort(struct node *data,int low,int high)
{
    if(low<high)
    {
        int pivot=partition(data,low,high);
        Quick_Sort(data,low,pivot-1);
        Quick_Sort(data,pivot+1,high);
    }
}

void *quick_sort(void *ptr)
{
    struct node *data=ptr;
    if(data->low<data->high)
    {
        int pivot=partition(data,data->low,data->high);
        pthread_t tid[2];
        struct node *temp=malloc(sizeof(data));
        temp->array=data->array;
        temp->low=pivot+1;
        temp->high=data->high;
        data->high=pivot-1;
        pthread_create(&tid[0],NULL,quick_sort,data);
        pthread_create(&tid[1],NULL,quick_sort,temp);
        pthread_join(tid[0],NULL);
        pthread_join(tid[1],NULL);
    }
}

int main()
{
    srand(time(NULL));
    struct node *data=(struct node *)malloc(sizeof(struct node));
    int n;
    printf("\nEnter the size of array : ");
    scanf("%d",&n);
    data->low=0;
    data->high=n-1;
}

```

```

data->array=(int *)malloc(sizeof(int)*n);
printf("\nArray elements before sorting : ");
for(int i=0;i<n;++i)
{
    data->array[i]=rand()%(5*n)+1;
    printf("%d ",data->array[i]);
}
printf("\n");

pthread_t tid;
pthread_create(&tid,NULL,quick_sort,data);
pthread_join(tid,NULL);
printf("\nArray elements after sorting : ");
for(int i=0;i<n;++i)
{
    printf("%d ",data->array[i]);
}
printf("\n");
return 0;
}

```

OUTPUT:

```

viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 6b.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 10
Array elements before sorting : 38 32 3 45 12 30 6 41 7 15
Array elements after sorting : 3 6 7 12 15 30 32 38 41 45
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 20
Array elements before sorting : 34 82 95 79 50 99 30 51 63 27 63 37 100 67 84
7 69 44 22 12
Array elements after sorting : 7 12 22 27 30 34 37 44 50 51 63 63 67 69 79 82
84 95 99 100
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out

Enter the size of array : 28
Array elements before sorting : 126 126 51 77 127 111 103 7 23 29 92 124 40 90
1 62 23 15 24 25 44 110 123 77 65 101 1 72
Array elements after sorting : 1 1 7 15 23 23 24 25 29 40 44 51 62 65 72 77 77
90 92 101 103 110 111 123 124 126 126 127
viknesh@viknesh-ubuntu:~/Documents/OS$

```

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

LOGIC:

- We get the number of points to be generated (iterations) from the user.
- For each iteration, a thread is created.

- In each iteration, we generate the point (x,y) inside a **square of side 1** and find whether the point lies inside or outside the **circle of diameter 1** and increment the number of points inside the circle accordingly.
- **area of circle/area of square = number of points within the circle/number of points within the square.**
- **$\pi = 4 \times (\text{number of points within the circle} / \text{number of points within the square})$.**

C CODE:

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

int iterations;

struct node
{
    int points_inside_square;
    int points_inside_circle;
};

double pi;

void *monte_carlo(void *ptr)
{
    struct node *data=ptr;
    double x,y,dist;
    for(int i=0;i<iterations;++i)
    {
        x=(double)(rand()%(iterations+1))/iterations;
        y=(double)(rand()%(iterations+1))/iterations;
        dist=x*x+y*y;
        ++data->points_inside_square;
        if(dist<=1)
        {
            ++data->points_inside_circle;
        }
    }
}

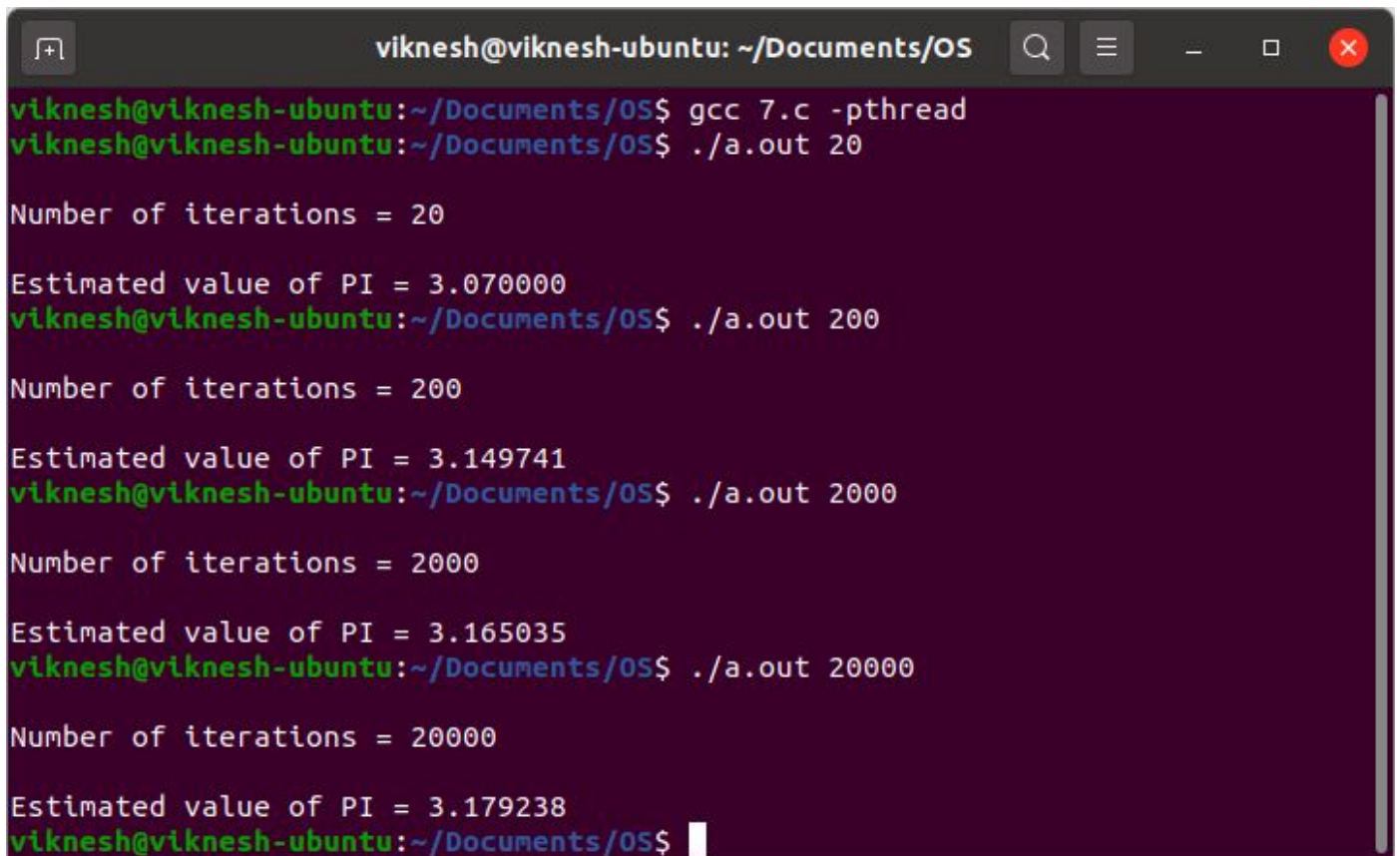
int main(int argc,char *argv[])
{
    srand(time(NULL));
    if(argc!=2)
    {
        printf("\nTry ./a.out <number_of_iterations>\n");
        return(0);
    }
    iterations=atoi(argv[1]);
    struct node *data=(struct node *)malloc(sizeof(struct node));
    data->points_inside_square=0;
    data->points_inside_circle=0;
    pthread_t tid[iterations];
    for(int i=0;i<iterations;++i)
```

```

{
    pthread_create(&tid[i],NULL,monter_carlo,data);
}
for(int i=0;i<iterations;++i)
{
    pthread_join(tid[i],NULL);
}
pi=(double)(4*data->points_inside_circle)/data->points_inside_square;
printf("\nNumber of iterations = %d\n",iterations);
printf("\nEstimated value of PI = %lf\n",pi);
return 0;
}

```

OUTPUT:



```

viknesh@viknesh-ubuntu: ~/Documents/OS
viknesh@viknesh-ubuntu:~/Documents/OS$ gcc 7.c -pthread
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 20

Number of iterations = 20

Estimated value of PI = 3.070000
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 200

Number of iterations = 200

Estimated value of PI = 3.149741
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 2000

Number of iterations = 2000

Estimated value of PI = 3.165035
viknesh@viknesh-ubuntu:~/Documents/OS$ ./a.out 20000

Number of iterations = 20000

Estimated value of PI = 3.179238
viknesh@viknesh-ubuntu:~/Documents/OS$

```