COM301P- Dr.Sivaselvan

OS Lab Assignment 6

Animesh Kumar (CED18I065)

Very Important Note:

Q1, 2, 3, 9 and 10 were *already* submitted as part of the *Assignment 5*(to which they originally belonged to). Therefore to save the time of both of us, I haven't put them here. Thanks.

- Question 4: Generation of Prime Numbers upto a limit supplied as Command Line Parameter (Multithreaded)

(The interpretation of the question is that the program outputs all the prime numbers less than or equal to the supplied parameter via command line.)

  - Algorithm: (1 parameter in input via cmd prompt)
    - The main thread creates new threads for each and every number smaller than or equal to the number entered.
    - These new threads parallely invoke the `_Bool is_prime(int n)`, which returns 1 if the number is prime,else returns 0.
    - If the returned value is 1, that number is printed on stdout.
    - After printing the new threads are joined in the main thread.
    - After joining all the threads, the main thread terminates successfully.
  - C Program:

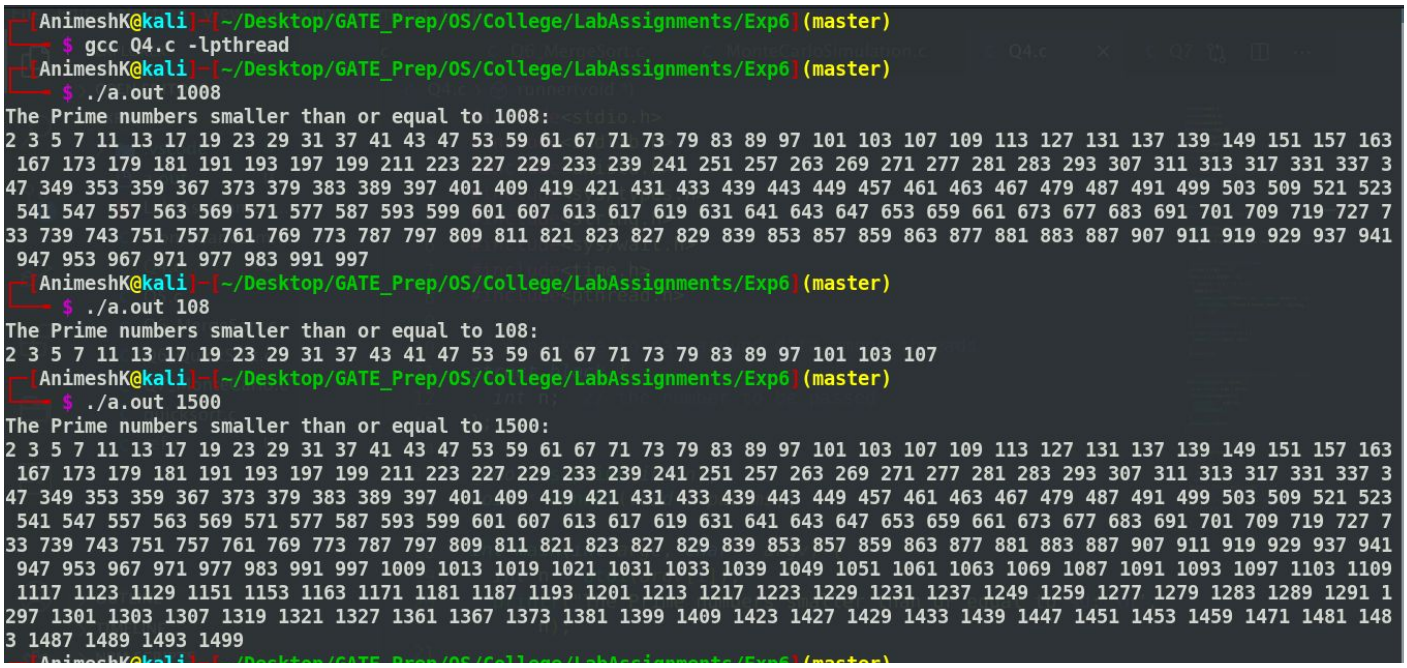
```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>
```

```c
#include<sys/wait.h>
#include<time.h>
#include<pthread.h>


// A package to pas around data among threads
struct block {
 int n;  // the number to be passed
};


_Bool is_prime(int n);
void * runner(void * params);


int main(int argc, char** argv) {
 if (argc < 2) {  // wrong input
   fprintf(stderr, "Too few arguments.\nEnter a number N, such that we print all the prime numbers smaller than or equal to N\n");
   exit(1);
 }
 int n = atoi(argv[1]);
 printf("The Prime numbers smaller than or equal to %d: \n", n);


 // Creating 1 thread for each number <= n
 pthread_t tid[n + 1];
```

```
 struct block args[n + 1];
 for (int i = 2; i <= n; i++) {
   args[i].n = i;
   if(pthread_create(&tid[i], NULL, runner, &args[i]) != 0)
     fprintf(stderr, "Thread Creation failed...\nExiting...\n");
 }
 // joining those threads
 for (int i = 2; i <= n; i++) {
   pthread_join(tid[i], NULL);
 }
 printf("\n");
}


// The thread routine which prints the number if it is prime
void * runner(void * params) {
 struct block * args = params;
 if(is_prime(args->n)) {
   printf("%d ", args->n);
   fflush(stdout);
 }
 pthread_exit(NULL);
}
```

```c
// Returns 1 if 'n' is a prime number else returns 0
_Bool is_prime(int n) {
 if (n <= 1)
   return 0;
 if (n <= 3)
   return 1;
 if ((n % 2 == 0) || (n % 3 == 0))
   return 0;
 for (int i = 5; i * i <= n; i = i + 6) {
   if (n % i == 0 || n % (i + 2) == 0)
     return 0;
 }
 return 1;
}
```

- Output:



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

- Algorithm:

The main threads shall create 3 threads `runner1`, `runner2` and `runner3` to calculate Mean, mode and median of the input integers.

- The `runner1`
  - finds sum of the integers using `runner_sum` which recursively calculates sum using divide and conquer for each half of the array.
  - Then divides by the size of array to print the sum
- The `runner2`
  - finds the frequency of all the integers using an array of `struct number`

- The integers which are having the max frequency are the modes of the integers.
- if all the integers have frequency equal to 1, there is no mode of this data.

- The `runner3`
  - if odd sized array, the middle element is the median
  - if an even sized array then the average of the middle element and the element just right to it, will be the median.

The main thread keeps waiting for all the threads to complete. The main thread terminates successfully.

C Program:

```c
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<pthread.h>


#define INDEX_NOT_FOUND -1


// A data package to pass arguments around threads

struct block {

 int * arr;  // the array

 int start;   // the size of array 'arr'
```

```
 int end;
 int sum_val; // sum of the array from 'start' to 'end'
};


// A data package which will be used to calculate MODE
struct number {
 int self;     // the number itself
 int frequency; // the frequency of the number
};


void* runner1(void* params);
void* runner2(void* params);
void* runner3(void* params);
void* runner_sum(void * params);
void PrintMode(struct number * Numbers, int size);
int get_index(int n, struct number * Numbers, int size);
void Initialize(struct number * Numbers, int size);
void asc_sort(int * arr, int size);
void my_swap(int* x, int* y);


int main() {
 int size;
 printf("Enter the number of elements: ");
```

```
scanf("%d", &size);

int arr[size];

for (int i = 0; i < size; i++) {

  scanf("%d", &arr[i]);

}

struct block args;

args.arr = arr;

args.start = 0;

args.end = size - 1;


void* (*runners[3])(void * params) = {runner1, runner2, runner3};

pthread_t tid[3];

pthread_attr_t attr;

pthread_attr_init(&attr);

for (int i = 0; i < 3; i++) {

  pthread_create(&tid[i], &attr, (*runners[i]), &args);

}

for (int i = 0; i < 3; i++) {

  pthread_join(tid[i], NULL);

}

return 0;

}
```

```c
/******************** Mean Calculation ************************/

// This thread calculates the mean

void* runner1(void* params) {

 struct block * args = params;

 pthread_t tid;

 pthread_create(&tid, NULL, runner_sum, args); // calculate the sum first

 pthread_join(tid, NULL);

 printf("Mean: %f\n", (float)args->sum_val / (float)(args->end + 1));

 pthread_exit(NULL);

}


// this thread routine finds sum of the array using divide and conquer

void* runner_sum(void * params) {

 struct block * array = params;

 if (array->start == array->end) {

   array->sum_val = array->arr[array->start];

   pthread_exit(NULL);

 }

 struct block array_left, array_right;

 int mid = (array->start + array->end) / 2;

 array_left.arr = array->arr;

 array_right.arr = array->arr;

 array_left.start = array->start;
```

```
 array_left.end = mid;
 array_right.start = mid + 1;
 array_right.end = array->end;

 // Find the sum of left and right parts of the array
 pthread_t tid1, tid2;
 pthread_create(&tid1, NULL, runner_sum, &array_left);
 pthread_create(&tid2, NULL, runner_sum, &array_right);
 pthread_join(tid1, NULL);
 pthread_join(tid2, NULL);
 array->sum_val = array_left.sum_val + array_right.sum_val;
 pthread_exit(NULL);
}

/*************************************************************/



/******************** Mode Calculation ************************/

// this calculates the mode of the numbers
void* runner2(void* params) {
 struct block * args = params;
 // Frequency calculation
```

```
 struct number Numbers[args->end + 1];
 Initialize(Numbers, args->end + 1);
 int j = 0;
 for (int i = args->start; i <= args->end; i++) {
   int id = get_index(args->arr[i], Numbers, args->end + 1);
   if (id == INDEX_NOT_FOUND) {
     Numbers[j].self = args->arr[i];
     Numbers[j].frequency = Numbers[j].frequency + 1;
     j++;
   }
   else
     Numbers[id].frequency = Numbers[id].frequency + 1;
 }
 PrintMode(Numbers, args->end + 1);
 pthread_exit(NULL);
}


// Initializing the Numbers array
void Initialize(struct number * Numbers, int size) {
 for (int i = 0; i < size; i++) {
   Numbers[i].self = -1;
   Numbers[i].frequency = 0;
 }
```

```
}


// Returns mode of the input Numbers and their frequencies
void PrintMode(struct number * Numbers, int size) {
 int max_frequency = 0;
 for (int i = 0; i < size; i++) {
   max_frequency = Numbers[i].frequency > max_frequency ?
Numbers[i].frequency : max_frequency;
 }
 printf("Mode (there may be 0 or more modes possible): ");
 if (max_frequency == 1)
   printf("There is no mode of the entered data.\n");
 else { // One or more than one mode exist
   for (int i = 0; i < size; i++) {
     if (Numbers[i].frequency == max_frequency)
       printf("%d ", Numbers[i].self);
   }
   printf("\n");
 }
}


// Returns the index of the N if it is found in Numbers[]
int get_index(int n, struct number * Numbers, int size) {
```

```
 for (int i = 0; i < size; i++) {
   if (Numbers[i].self == n)
     return i;
 }
 return INDEX_NOT_FOUND;
}


/*************************************************************/


/****************** Median Calculation ************************/


// this thread routine calculates median of the integers
void* runner3(void* params) {
 struct block * args = params;
 int size = args->end + 1;
 asc_sort(args->arr, size);
 if (size % 2 == 0) {
   float median = (float)(args->arr[size / 2] + args->arr[size / 2 - 1]) /
(float)2;
   printf("\nMedian: %f\n", median);
 }
 else  // for odd length array
   printf("\nMedian: %d\n", args->arr[size / 2]);
```

```
 pthread_exit(NULL);
}


// Sorting in ascending order
void asc_sort(int * arr, int size) {
 for (int i = 0; i < size - 1; i++) {
   for (int j = 0; j < size - i - 1; j++) {
     if (arr[j] > arr[j + 1])
       my_swap(&arr[j], &arr[j + 1]);
   }
 }
}


// swap two integers
void my_swap(int* x, int* y) {
 int temp = *x;
 *x = *y;
 *y = temp;
}


/*****************************************************************/
```

- Output:

```
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ gcc -lpthread Q5.c
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter the number of elements: 10
12 234 321 12 34 54 65 65 76 76
Mode (there may be 0 or more modes possible): 12 65 76

Median: 65.000000
Mean: 94.900002
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter the number of elements: 11
123 21 100 97 32 123 21 21 65 100 100
Mode (there may be 0 or more modes possible): 21 100

Median: 97
Mean: 73.000000
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter the number of elements: 16
123 21 100 97 32 123 21 21 65 100 100
20 20 20 30 30
Mode (there may be 0 or more modes possible): 21 100 20

Median: 31.000000
Mean: 57.687500
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
```

```
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter the number of elements: 16
12 23 343 2323 5564 766 22543 65 555 777 342 233 6565454 123 3344 545
Mode (there may be 0 or more modes possible): There is no mode of the entered data.

Median: 550.000000
Mean: 412688.250000
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter the number of elements: 4
12 21 12 23
Mode (there may be 0 or more modes possible): 12
Median: 16.500000

Mean: 17.000000
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$
```

- ## Question 6(A): Multithreaded MergeSort
  - ### Algorithm:

    Given an array of n integers as input.

    - The main thread creates two threads.
    - Both of these new threads recursively sort the left half and right half of the input array respectively.
    - After joining above two threads, the main thread calls the usual Merge() routine to merge the sorted halves.
    - After printing the sorted final array, the main thread terminates.
  - ### C Program:

```c
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

#include<pthread.h>



// A package to pass around data among threads

struct block {

 int * arr;  // the array

 int start;  // the start index

 int end;    // the end index
```

```
};


void merge(int arr[], int l, int m, int r);
void * runner(void * params);


int main   () {
 int size;
 printf("Enter Array size: ");
 scanf("%d", &size);
 int arr[size];
 for (int i = 0; i < size; i++)
   scanf("%d", &arr[i]);
  struct block args;
 args.arr = arr;
 args.start = 0;
 args.end = size - 1;
 pthread_t tid;
 pthread_create(&tid, NULL, runner, &args);
 pthread_join(tid, NULL);
 printf("\nSorted array is \n");
 for (int i = 0; i < size; i++)
   printf("%d ", args.arr[i]);
 printf("\n");
```

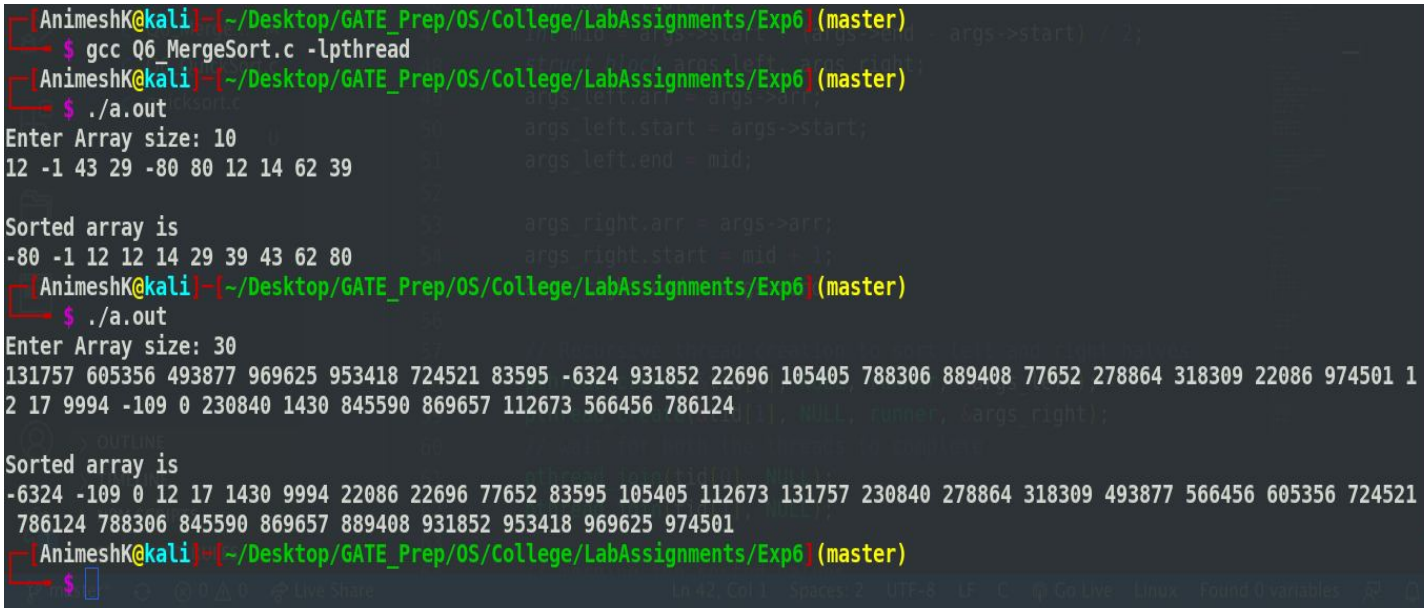
```
 return 0;
}


// This routine recursively implements multithreading version of Mergesort
void *runner(void * params) {
 struct block * args = params;
 if (args->start < args->end) {
   pthread_t tid[2];
   int mid = args->start + (args->end - args->start) / 2;
   struct block args_left, args_right;
   args_left.arr = args->arr;
   args_left.start = args->start;
   args_left.end = mid;
    args_right.arr = args->arr;
   args_right.start = mid + 1;
   args_right.end = args->end;


   // Recursive thread creation to sort left and right halves
   pthread_create(&tid[0], NULL, runner, &args_left);
   pthread_create(&tid[1], NULL, runner, &args_right);
   // wait for both the threads to complete
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
```

```
    // merging the sorted halves
    merge(args->arr, args->start, mid, args->end);
  }
  pthread_exit(NULL);
}



// The usual Merge routine to merge the sorted halves of the array arr[]
void merge(int arr[], int l, int m, int r)
{
  int i, j, k;
  int n1 = m - l + 1;
  int n2 = r - m;
  int L[n1], R[n2];
  for (i = 0; i < n1; i++)
    L[i] = arr[l + i];
  for (j = 0; j < n2; j++)
    R[j] = arr[m + 1 + j];
  i = 0; // 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++;
}
// for remaining elements
while (i < n1) {
  arr[k] = L[i];
  i++;
  k++;
}
// for remaining elements
while (j < n2) {
  arr[k] = R[j];
  j++;
  k++;
```

```
 }

}
```

- Output:



Question 6(B): Multithreaded QuickSort

- Algorithm:

Given an array of n integers as input.

- The main thread uses the last element of the array as the pivot element.
- The `PlacePivotCorrectly()` routine partitions the array appropriately and fixes the final location of the pivot element after partitioning the elements lesser than or equal to the pivot on left side and the elements greater than the pivot on the right side of the `corrected_index` returned.
- The main thread creates two new threads which recursive QuickSort the left array and right array respectively.
- After joining above two threads, the final sorted array is printed.

- The main thread terminates successfully.

- C Program:

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


// A package to pass around data among threads
struct block {
 int * arr;  // the array
 int start;  // the start index
 int end;    // the end index
};


void swap_ints(int* x, int* y);
int PlacePivotCorrectly(int * arr, int start, int end);
void * runner(void * params);


int main() {
 int size;
```

```
 printf("Enter Array size: ");
 scanf("%d", &size);
 int arr[size];
 for (int i = 0; i < size; i++)
   scanf("%d", &arr[i]);
  struct block args;
 args.arr = arr;
 args.start = 0;
 args.end = size - 1;
 pthread_t tid;
 pthread_create(&tid, NULL, runner, &args);
 pthread_join(tid, NULL);
 printf("\nSorted array is \n");
 for (int i = 0; i < size; i++)
   printf("%d ", args.arr[i]);
 printf("\n");
 return 0;
}


// This routine recursively implements multithreading version of Quicksort
void *runner(void * params) {
 struct block * args = params;
 if (args->start < args->end) {
```

```
   pthread_t tid[2];
   int corrected_index = PlacePivotCorrectly(args->arr, args->start,
args->end);


   struct block args_left, args_right;
   args_left.arr = args->arr;
   args_left.start = args->start;
   args_left.end = corrected_index - 1;
    args_right.arr = args->arr;
   args_right.start = corrected_index + 1;
   args_right.end = args->end;


   // Recursive thread creation to sort left and right halves
   pthread_create(&tid[0], NULL, runner, &args_left);
   pthread_create(&tid[1], NULL, runner, &args_right);
   // wait for both the threads to complete
   pthread_join(tid[0], NULL);
   pthread_join(tid[1], NULL);
 }
 pthread_exit(NULL);
}
```

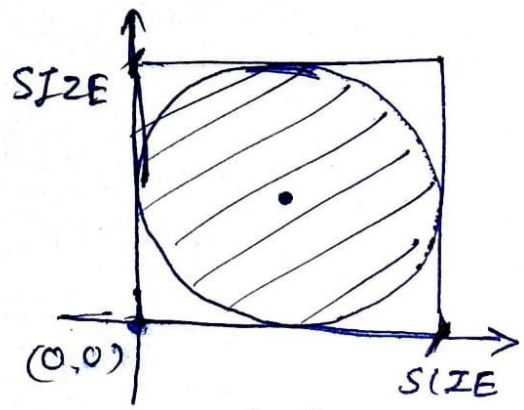
```
// Places the pivot element appropriately by partitioning the array and
return the new pivot index

int PlacePivotCorrectly(int * arr, int start, int end) {

 int i = start - 1;

 int j = start;

 int pivot_id = end;

 int pivot = arr[pivot_id];

 while(j <= pivot_id - 1) {

   if(arr[j] < pivot) {

     i++;

     swap_ints(&arr[i], &arr[j]);

   }

   j++;

 }

 swap_ints(&arr[i + 1], &arr[pivot_id]);

 return i + 1;  // this is new pivot_id

}



// swaps two integers

void swap_ints(int* x, int* y) {

 int temp = *x;

 *x = *y;

 *y = temp;
```

```
}
```

- Output:

```
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ gcc Q6_QuickSort.c -lpthread
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter Array size: 10
12 -1 43 29 -80 80 12 14 62 39

Sorted array is
-80 -1 12 12 14 29 39 43 62 80
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter Array size: 30
131757 605356 493877 969625 953418 724521 83595 -6324 931852 22696 105405 788306 889408 77652 278864 318309 22086 974501 1
2 17 9994 -109 0 230840 1430 845590 869657 112673 566456 786124

Sorted array is
-6324 -109 0 12 17 1430 9994 22086 22696 77652 83595 105405 112673 131757 230840 278864 318309 493877 566456 605356 724521
 786124 788306 845590 869657 889408 931852 953418 969625 974501
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
```

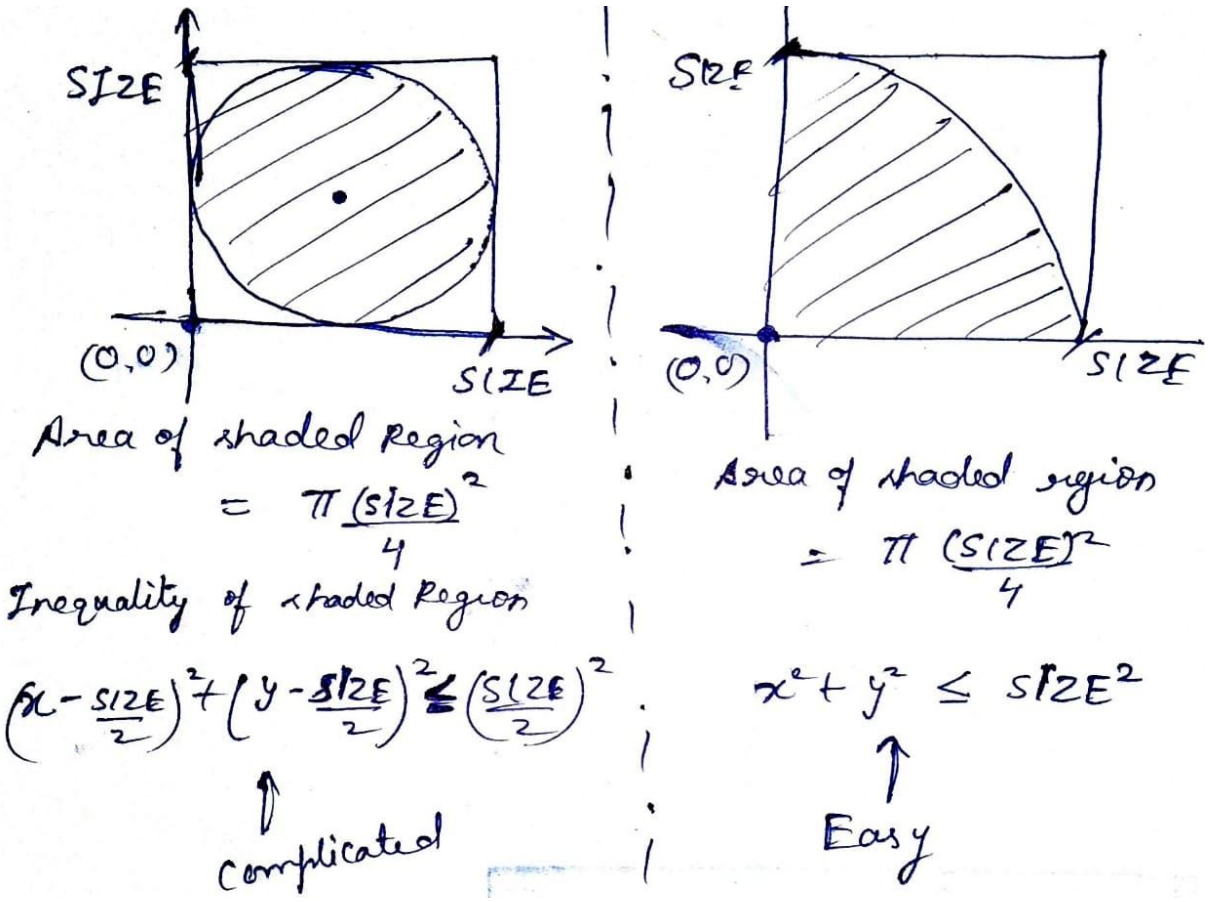
- Question 7: Estimation of π Value using Monte Carlo simulation technique using threads.
  - Algorithm:



- If we have a square of side length SIZE and a circle of diameter SIZE. If the Area of the Square is S and the area of the circle is C, then the ***π*** value will be:

  4*C / S , because C = ***π*** (SIZE/2)^2 and S = SIZE^2

- But the setup we are using is this:



- **So to save the time of calculation and code simplicity I am choosing the right diagram which is equivalent to the left diagram. The routine `BelongsToCircle()` works on the basis of the right diagram.**
- For the illusion of the Area, we use points randomly distributed across the whole square.
- The main thread creates **10000** threads after initialising a mutex object lock.
- Each and every new thread chooses a random point on the X-Y 2D plane :

- Locks the mutex and enters the critical section
- Choose a random float number less than or equal to SIZE for x coordinate.
- Choose another random float number less than or equal to SIZE for y coordinate.
- if the new point lies inside the circle(checked using `BelongsToCircle()`), increment the `ScoreCircle` .
- Increment the `ScoreSquare`
- Unlocks the mutex and exits from critical section
- Thread terminates

■ After joining all the threads the main thread destroys the mutex object lock and terminates successfully.

○ C Program:

```c
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

#include<pthread.h>



#define SIZE 10

#define NUMTHREADS 10000



// A package to pass around data among threads

struct block {

 int * score_circle; // pointer to ScoreCircle
```

```
 int * score_square; // pointer to ScoreSquare
};


_Bool BelongsToCircle(float x, float y);
void * runner(void * params);


pthread_mutex_t lock;  // A mutex object


int main() {
 int ScoreCircle = 0; // # points in Circle
 int ScoreSquare = 0; // # points in Square
 pthread_mutex_init(&lock, NULL); // Initialise mutex object
 pthread_t tid[NUMTHREADS];
 struct block args;
 args.score_circle = &ScoreCircle;
 args.score_square = &ScoreSquare;
 srand(time(0));
  // Creating NUMTHREADS which land NUMTHREADS points
 for (int i = 0; i < NUMTHREADS; i++)
   pthread_create(&tid[i], NULL, runner, &args);
 for (int i = 0; i < NUMTHREADS; i++)
   pthread_join(tid[i], NULL);
 pthread_mutex_destroy(&lock); // Destroy the mutex object
```

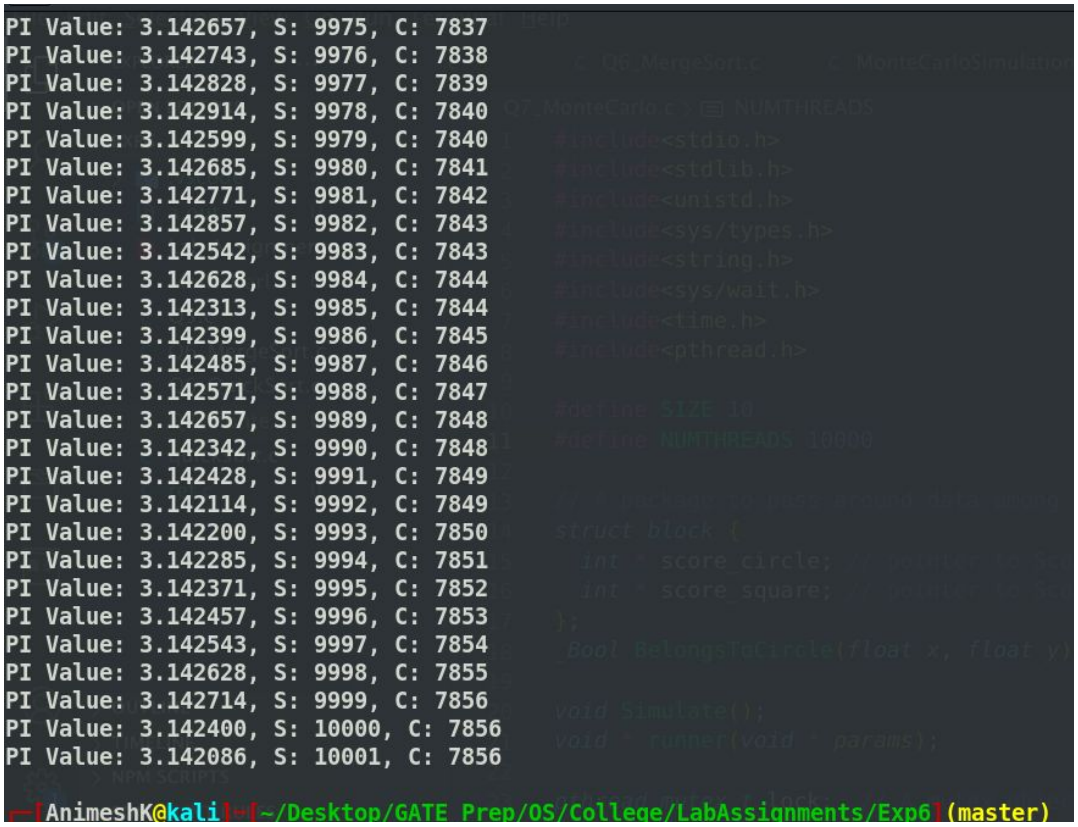
```
 printf("\n");

 return 0;

}


// The thread routine which lands one point on the SIZE X SIZE square

void * runner(void * params) {

 struct block * args = params;

 // Entering into the critical section

 pthread_mutex_lock(&lock);  // lock the mutex

  float x = ((float)rand()/(float)(RAND_MAX)) * SIZE;

 float y = ((float)rand()/(float)(RAND_MAX)) * SIZE;

 if (BelongsToCircle(x, y))

   *(args->score_circle) += 1;

 *(args->score_square) += 1;

 float PI = ((float)(*(args->score_circle) * 4)) /
((float)(*(args->score_square)));

 printf("PI Value: %f, S: %d, C: %d\n", PI, *args->score_square,
*args->score_circle);


 pthread_mutex_unlock(&lock);

 pthread_exit(NULL);

}


// Returns 1 if (x, y) belongs to the circle
```

```
_Bool BelongsToCircle(float x, float y) {

 float radius = x * x + y * y;

 if (radius <= SIZE * SIZE)

   return 1;  // Belongs to circle

 return 0; // Belongs to square

}
```

- Output:

S: means number of points on Square and C: means number of points on the circle.

```
┌──[AnimeshK@kali]-[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
PI Value: 4.000000, S: 1, C: 1
PI Value: 4.000000, S: 2, C: 2
PI Value: 4.000000, S: 3, C: 3
PI Value: 4.000000, S: 4, C: 4
PI Value: 4.000000, S: 5, C: 5
PI Value: 4.000000, S: 6, C: 6
PI Value: 4.000000, S: 7, C: 7
PI Value: 4.000000, S: 8, C: 8
PI Value: 4.000000, S: 9, C: 9
PI Value: 4.000000, S: 10, C: 10
PI Value: 3.636364, S: 11, C: 10
PI Value: 3.666667, S: 12, C: 11
PI Value: 3.692308, S: 13, C: 12
PI Value: 3.714286, S: 14, C: 13
PI Value: 3.466667, S: 15, C: 13
PI Value: 3.500000, S: 16, C: 14
PI Value: 3.294118, S: 17, C: 14
PI Value: 3.333333, S: 18, C: 15
PI Value: 3.368421, S: 19, C: 16
PI Value: 3.400000, S: 20, C: 17
PI Value: 3.428571, S: 21, C: 18
PI Value: 3.454545, S: 22, C: 19
PI Value: 3.304348, S: 23, C: 19
PI Value: 3.333333, S: 24, C: 20
PI Value: 3.360000, S: 25, C: 21
PI Value: 3.384615, S: 26, C: 22
PI Value: 3.407408, S: 27, C: 23
```



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

    The input to this algorithm is a number n, representing the size of the Matrix. Then an n X n matrix.

    In this program we are using two structures to implement multithreading at two places, one at the get_inverse() and other at the get_transpose().

  - The determinant is calculated, if it is zero, the inverse doesn't exist(Singular Matrix).
  - The main thread calls the get_inverse()
  - The get_inverse() function creates n^2 threads to fill the places of the matrix with their cofactors.
  - To get the Adjoint matrix, we need to transpose the matrix consisting of corresponding cofactors. For this we require get_transpose():

    - get_transpose() creates n^2 threads to transpose the matrix.
- Dividing the Adjoint matrix by the determinant gives us the Inverse matrix finally.
- After printing the inverse matrix and freeing the allocated memory locations the main thread terminates successfully.

C Program:

```
#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<sys/types.h>

#include<string.h>

#include<sys/wait.h>

#include<time.h>

#include<pthread.h>


#define SINGULAR_MATRIX 0

#define INVERSE_EXISTS 1


// A package to pass data to get_inverse()

struct block_inv {

 int i;

 int j;

 float ** Matrix;

 float ** Inter;

 int size;
```

```
};


// A package to pass data to get_transpose()
struct block_trans {
 int i;
 int j;
 float ** src;
 float ** dest;
};


void * runner_trans(void * params);
void * runner_inv(void * params);
int get_inverse(float **Matrix, float **Inverse, int size);
void get_transpose(float ** dest, float ** src, int size);
float get_determinant(float ** Matrix, int size);
float get_cofactor(int x, int y, float ** Matrix, int size);
void PrintMat(float **Matrix, int size);
int sign(int x, int y);



int main() {
 int n;  // size of the matrix
 printf("Enter size n for the n x n square matrix: ");
```

```
scanf("%d", &n);
// Dynamic memory allocation
float ** Matrix = malloc(sizeof(float *) * n);
for (int i = 0; i < n; i++) {
  Matrix[i] = malloc(sizeof(float) * n);
  for (int j = 0; j < n; j++)
    scanf("%f", &Matrix[i][j]);
}

float ** Inverse = malloc(sizeof(float *) * n);
for (int i = 0; i < n; i++)
  Inverse[i] = malloc(sizeof(float) * n);

if(get_inverse(Matrix, Inverse, n) == SINGULAR_MATRIX) {
  fprintf(stderr, "The Inverse is impossible, since the Matrix is
singular.\n");
}
else {
  printf("The Inverse of the input matrix is: \n");
  PrintMat(Inverse, n);
}
// Freeing the allocated memory
for (int i = 0; i < n; i++)
```

```
   free(Matrix[i]);
 free(Matrix);
 for (int i = 0; i < n; i++)
   free(Inverse[i]);
 free(Inverse);
 return 0;
}



/******************** Inverse Creation **********************/



// Returns the inverse(Matrix[][]) in Inverse[]
int get_inverse(float **Matrix, float **Inverse, int size) {
 float determinant = get_determinant(Matrix, size);
 if (determinant == 0)
   return SINGULAR_MATRIX;


 float ** Intermediate;
 pthread_t tid[size*size];
 struct block_inv args[size*size];
 int ptr = 0;
 Intermediate = malloc(sizeof(float *) * size);
```

```
for (int i = 0; i < size; i++) {
  Intermediate[i] = malloc(sizeof(float) * size);
  for (int j = 0; j < size; j++) {
    args[ptr].i = i;
    args[ptr].j = j;
    args[ptr].Inter = Intermediate;
    args[ptr].Matrix = Matrix;
    args[ptr].size = size;
    pthread_create(&tid[ptr], NULL, runner_inv, &args[ptr]);
    ptr++;
  }
}
// Adjoint matrix creation
for(ptr = 0; ptr < size*size; ptr++)
  pthread_join(tid[ptr], NULL);
get_transpose(Inverse, Intermediate, size);
 for(int i = 0; i < size; i++)
  free(Intermediate[i]);
free(Intermediate);
 // Converting Adjoint to inverse
for (int i = 0; i < size; i++) {
  for (int j = 0; j < size; j++)
    Inverse[i][j] = Inverse[i][j] / determinant;
```

```
 }
 return INVERSE_EXISTS;
}


// The thread routine to calculate cofactors to create Adjoint
void * runner_inv(void * params) {
 struct block_inv * args = params;
 args->Inter[args->i][args->j] = get_cofactor(args->i, args->j,
args->Matrix, args->size);
 pthread_exit(NULL);
}


/*************************************************************/



/******************** Transpose logic ***********************/



// Returns transpose of 'src' into 'dest'
void get_transpose(float ** dest, float ** src, int size) {
 pthread_t tid[size*size];
 struct block_trans args[size*size];
 int ptr = 0;
```

```
 for (int i = 0; i < size; i++) {
   for (int j = 0; j < size; j++) {
     args[ptr].i = i;
     args[ptr].j = j;
     args[ptr].src = src;
     args[ptr].dest = dest;
     pthread_create(&tid[ptr], NULL, runner_trans, &args[ptr]);
     ptr++;
   }
 }
 for(ptr = 0; ptr < size*size; ptr++)
   pthread_join(tid[ptr], NULL);
}


// The thread routine to swap data to create transpose
void * runner_trans(void * params) {
 struct block_trans * args = params;
 int i = args->i;
 int j = args->j;
 args->dest[i][j] = args->src[j][i];
 pthread_exit(NULL);
}
```

```
/*************************************************************/



/************ Determinant and Cofactor calculators ***********/


// Returns the determinant of square Matrix[][]
float get_determinant(float ** Matrix, int size) {
 if (size == 2) // Base Case
   return ((Matrix[0][0] * Matrix[1][1]) - (Matrix[0][1] * Matrix[1][0]));
 float sum = 0;
 for(int j = 0; j < size; j++)
   sum += (Matrix[0][j] * get_cofactor(0, j, Matrix, size));
 return sum;
}


// Returns the cofactor(x, y) (the signed minor M(x, y))
float get_cofactor(int x, int y, float ** Matrix, int size) {
 float **Cofactor = malloc(sizeof(float *) * size);
 for (int i = 0; i < size; i++)
   Cofactor[i] = malloc(sizeof(float) * size);
 int cof_i = 0;
 int cof_j = 0;
 for (int j = 0; j < size; j++) {
```

```
   if (j != y) {
     cof_i = 0;
     for (int i = 0; i < size; i++) {
       if(i != x) {
         Cofactor[cof_i][cof_j] = Matrix[i][j];
         cof_i++;
       }
     }
     cof_j++;
   }
 }
 float determinant = sign(x, y) * get_determinant(Cofactor, size - 1);
 for (int i = 0; i < size; i++)
   free(Cofactor[i]);
 free(Cofactor);
 return determinant;
}


/*********************************************************************/



/******************** General functions ************************/
```

```
// Returns the appropriate sign

int sign(int x, int y) {

 if ((x + y) % 2 == 0)

   return 1;

 return -1;

}



// Prints the Matrix[][]

void PrintMat(float **Matrix, int size) {

 for (int i = 0; i < size; i++) {

   for (int j = 0; j < size; j++) {

     printf("%f  ", Matrix[i][j]);

   }

   printf("\n");

 }

}

/*************************************************************/
```

- Output:

```
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ gcc Q8_Final.c -lpthread
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter size n for the n x n square matrix: 4
1 3 4 5
2 12 34 21
36 56 67 78
15 200 11 90
The Inverse of the input matrix is:
-0.568735  -0.022967  0.046102  -0.002999
-0.271345  0.022926  0.003461  0.006725
-0.326397  0.056002  0.006017  -0.000149
0.737672  -0.053964  -0.016111  -0.003316
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter size n for the n x n square matrix: 4
-0.568735  -0.022967  0.046102  -0.002999
-0.271345  0.022926  0.003461  0.006725
-0.326397  0.056002  0.006017  -0.000149
0.737672  -0.053964  -0.016111  -0.003316
The Inverse of the input matrix is:
1.000072  3.000119  4.000161  5.000169
2.000605  12.000920  34.001064  21.001270
36.001038  56.000805  67.002365  78.002106
15.003587  200.017136  11.006698  90.011925
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
```

```
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter size n for the n x n square matrix: 3
2 2 2
2 4 2
2 2 4
The Inverse of the input matrix is:
1.500000  -0.500000  -0.500000
-0.500000  0.500000  -0.000000
-0.500000  -0.000000  0.500000
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└─$ ./a.out
Enter size n for the n x n square matrix: 3
1.500000  -0.500000  -0.500000
-0.500000  0.500000  -0.000000
-0.500000  -0.000000  0.500000
The Inverse of the input matrix is:
2.000000  2.000000  2.000000
2.000000  4.000000  2.000000
2.000000  2.000000  4.000000
┌──[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
```

```
┌─[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└──╼ $ ./a.out
Enter size n for the n x n square matrix: 3
1 1 1
1 2 1
1 1 2
The Inverse of the input matrix is:
3.000000  -1.000000  -1.000000
-1.000000  1.000000  -0.000000
-1.000000  -0.000000  1.000000
┌─[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
└──╼ $ ./a.out
Enter size n for the n x n square matrix: 3
3.000000  -1.000000  -1.000000
-1.000000  1.000000  -0.000000
-1.000000  -0.000000  1.000000
The Inverse of the input matrix is:
1.000000  1.000000  1.000000
1.000000  2.000000  1.000000
1.000000  1.000000  2.000000
┌─[AnimeshK@kali]─[~/Desktop/GATE_Prep/OS/College/LabAssignments/Exp6](master)
```

Thanks,

Animesh Kumar

CED18I065