MANIPAL INSTITUTE OF TECHNOLOGY
MANIPAL
A Constituent Institution of Manipal University

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

A Comparative Study of Multithreading and Multiprocessing in Various Searching and Sorting Algorithms

Project Report

Submitted by
Anuradha Sharma- 200905292
Nittala Sri Manish- 200905252
Sarveshwari Keralikar- 200905249
Shubham Suryank- 200905334
Aashna Sharan - 200905184

ACKNOWLEDGEMENT

We would like to show our gratitude towards the Department of Computer Science Engineering for giving us the opportunity to work on this project. This project has helped us develop a deeper understanding of the concepts of operating systems.

We would also like to thank our teacher Ms. Suchitra Shetty for guiding us throughout the project and for providing us insightful ideas to move forward. We would also like to thank Ms. Suma D for guiding us during our laboratory sessions.

We would also extend our thanks to our fellow classmates who have consistently motivated and helped us by exchanging interesting ideas.

TABLE OF CONTENTS

1.Objective

The objective of this project is to draw a comparison between multithreading and multiprocessing environments by implementing various searching and sorting techniques in both these environments. In this project we use two searching algorithms linear search, binary search and one sorting technique i.e bubble sort.

The efficiency of algorithms can be measured in terms of execution time (complexity) and amount of memory required. In order to reduce the running time of a given algorithm, techniques like multithreading and multiprocessing are being applied, leading to evolution of the architecture of computers. We compare the performance of these sorting and searching algorithms with respect to time and speed in relation with multiple threads and multiple processes.

Keywords

Multithreading, Multiprocessing, Linear Search, Binary Search, Bubble Sort

Programming Fundamentals Used

1)Language used is **C language.**

2) The operating system used is Linux.

2. INTRODUCTION

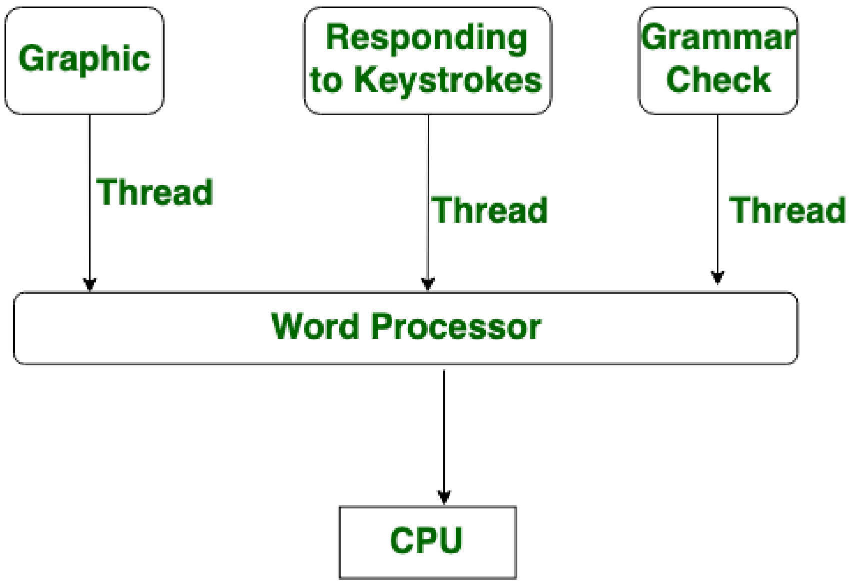
2.1 Basic concepts of Operating Systems

2.1.1 Multithreading

Multithreading is a system in which multiple threads are created of a process for increasing the computing speed of the system. In multithreading , many threads of a process are executed simultaneously .Multithreading allows a process to divide itself into a number of individual threads.

These threads share with each other code, data and other resources. Each thread has its own stack and a set of registers. A common address space is shared among the threads of a process.

Each thread belongs to exactly one process and no thread can exist outside a process.Each thread is independent and has its own path of execution with enabled inter thread communication. Thread is the path followed while executing a program. Each thread has its own program counter, stack and register. A thread is a light weight process.



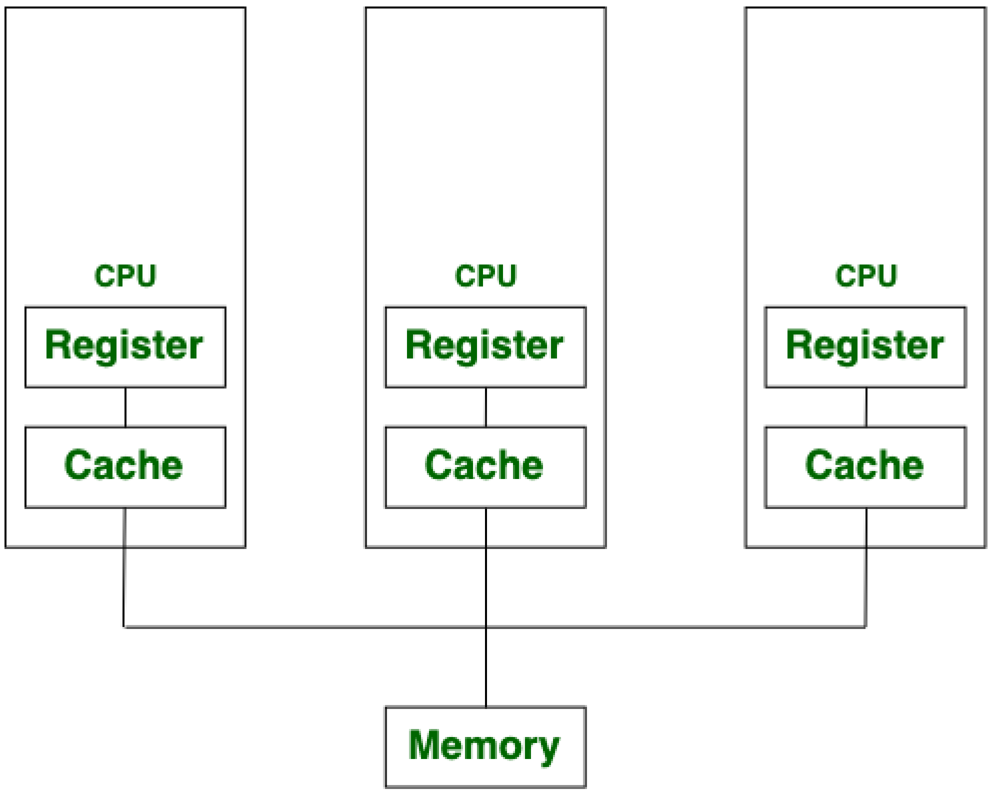
Multithreading

2.1.2 Multiprocessing

Multiprocessing is a system that has more than one or two processors. In multiprocessing, CPU's are added for increasing computing speed of the system. Because of Multiprocessing, there are many processes that can be executed simultaneously.

Each microprocessor has its central processing unit(CPU) on a single tiny chip. The major advantage of a multiprocessor computer is the speed and the ability to manage larger amounts of information.

The processes do not share any address space



Multiprocessing

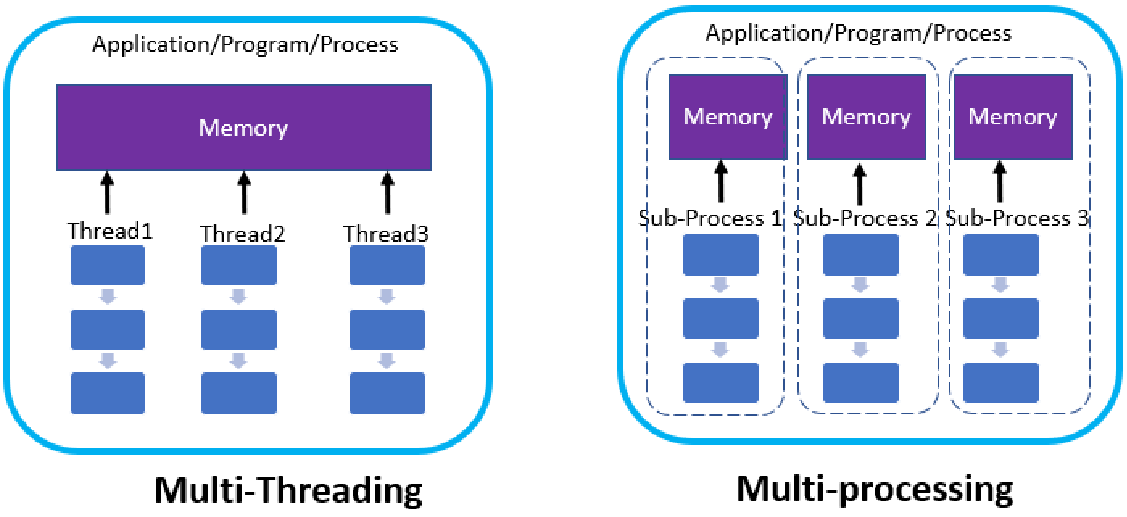


Fig 2.1 Comparison between multiprocessing and multithreading

Multithreading vs Multiprocessing

Multithreading	Multiprocessing
Many threads are created of a single process .	CPU'S ARE ADDED FOR INCREASING COMPUTING POWER
MANY THREADS OF A PROCESS ARE EXECUTED SIMULTANEOUSLY	MANY PROCESSES ARE EXECUTED SIMULTNEOUSLY
A cOMMON ADDRESS SPACE IS SHARED BY ALL THE THREADS	EVERY PROCESS OWNED A SEPARATE ADDRESS SPACE
CREATION IS ACCORDING TO ECONOMICAL	PROCESS CREATION IS TIME-CONSUMING

2.2 Algorithms

An algorithm is a sequence of steps that help solve computational problems. Algorithms act as an exact list of instructions that conduct specified step by step in software-based routines.

Algorithms play an important role in solving computational problems. An algorithm is a well-defined computational procedure that accepts input and produces an output.

2.2.1 Searching Algorithms

Searching Algorithms are used to locate a specific data in a collection of data that is provided.

2.2.1.1 Linear Search

Linear search is a search algorithm that starts at one end and goes through each element of a list until the desired element is found.

If the element is not found the search continues till the end of the data set.

Approach:

1. Start from the leftmost element of arr[] and one by one compare a with each element of arr[].
2. If a matches with an element, return the array index.
3. If a doesn't match with any of the elements return -1.

2.2.1.2 Binary Search

It is a searching algorithm which finds the position of the desired element within a sorted array.

Approach:

1. Begin with the middle element of the whole array as a search key.

2. If the value of the search key is equal to the item then return the index of the search key.
3. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half.
4. Otherwise, narrow it to the upper half.
5. Do this repeatedly until the interval is empty.

	Best Case time complexity	Average case time complexity	Worst case time complexity
Linear Search	O(1)	O(n)	O(n)
Binary Search	O(1)	O(log n)	O(log n)

Here n is the number of elements in an array.

2.2.2 Sorting Algorithms

These algorithms are used in order to arrange a collection of data-items in either ascending or descending order.

2.2.2.1 Bubble Sort

This algorithm repeatedly swaps the adjacent elements if they are not in the right order. This is suitable only for small data sets.

Approach:

1. Run a nested for loop to transverse the input array using two variables i and j, such that 0 < i< n-1 and 0 < j < n-1

2. If arr[j] is greater than arr[j+1] then swap these adjacent elements, else move on.

3. This whole process is repeated until no swaps have occurred on the last pass.

	Best Case	Average Case	Worst Case
Bubble Sort	O(n)	O(n^2)	O(n^2)

3. Methodology

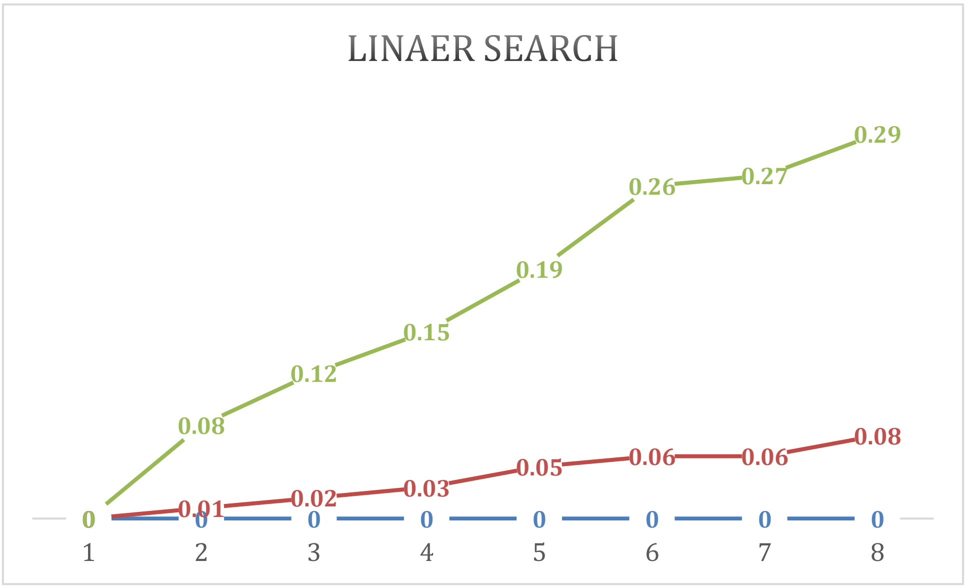
3.1 Linear Search:

Multithreading: We use the pthread library and then create two threads, each of them will search for a key value. If it is found we set the flag to 1 and display the output along with the time taken.

Multiprocessing: Two processes are creates which help in finding a particular value in a list. This is done by checking every one of its elements one at a time until we find the desired item.

Array Size	Running Time of Linear Search Dual Thread(ms)	Running Time of Linear Search Dual-core(ms)
30	**0.17**	**0.01**
60	**0.81**	**0.02**

120	1.13	0.03
240	1.17	0.05
500	1.11	0.06
1000	1.59	0.11

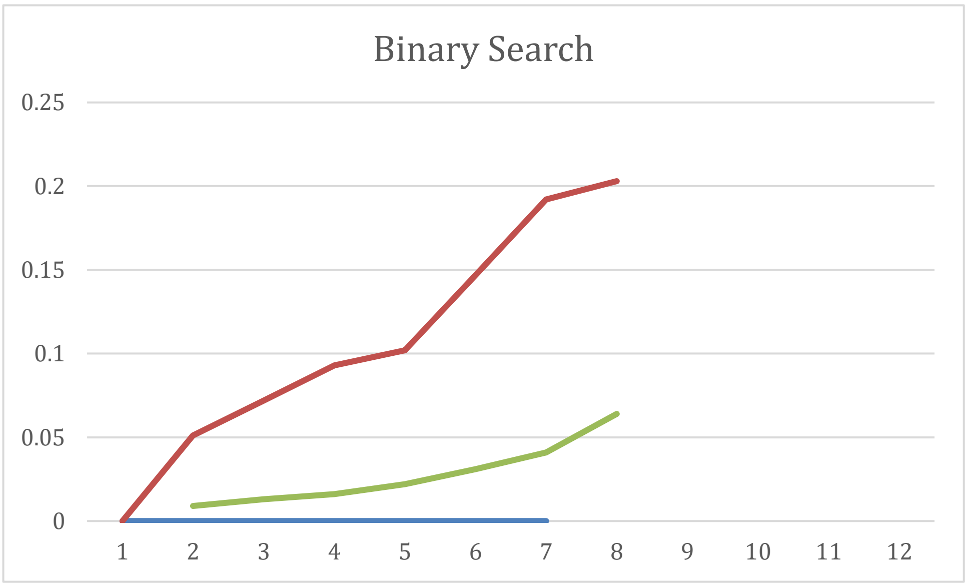


3.2 Binary Search

Multithreading: Using the pthread library we create two threads, each of which will perform binary search individually. If the desired key is found we set the flag variable to 1 and display the output.

Multiprocessing: We create two processes which perform binary search . They compare the target value with the middle element if it does not match the half in which the target value does not lie is eliminated and this process is continued until the value is found.

Array Size	Running Time of Linear Search Dual Thread(ms)	Running Time of Linear Search Dual-core(ms)
30	**0.57**	**0.0090**
60	**0.60**	**0.0130**
120	**0.58**	**0.0147**
240	**0.46**	**0.0198**
500	**0.49**	**0.0422**
1000	**0.53**	**0.0672**

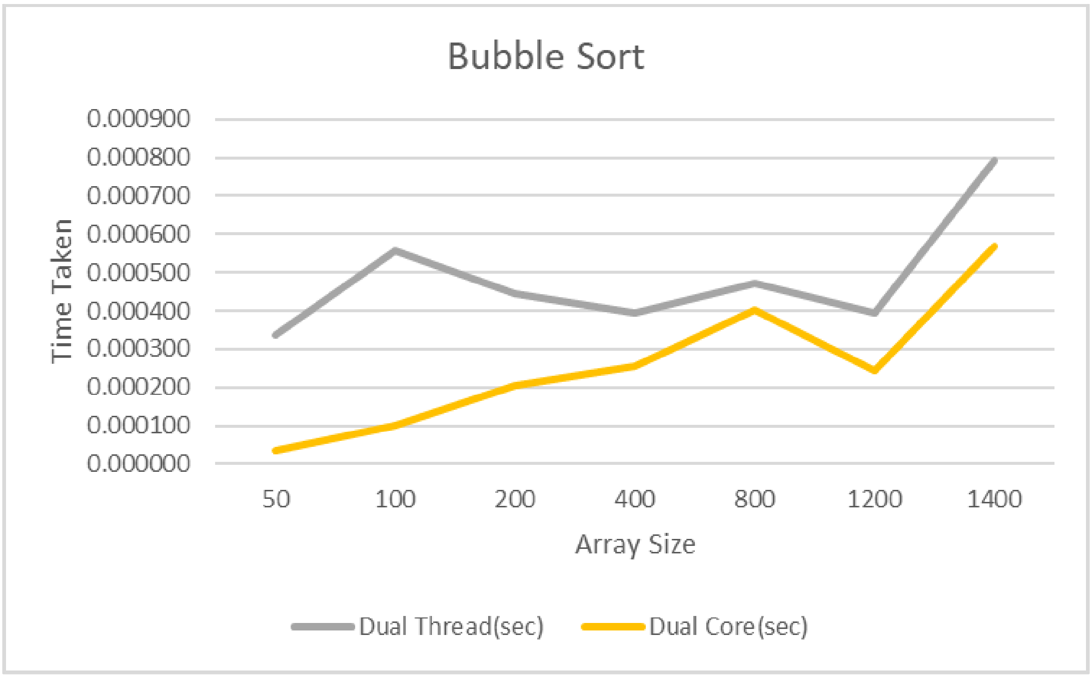


3.3 Bubble Sort

Multithreading: Using the pthread library to create two threads, each of which contains alternate elements from the list. The two threads perform bubble sort and the sorted elements from the threads are combined.

Multiprocessing: We create two processes which perform bubble sort. The sorted elements in both the processes are then combined.

Array Size	Running Time of Linear Search Dual Thread(ms)	Running Time of Linear Search Dual-core(ms)
30	**0.01**	**0.065**
60	**0.03**	**0.058**
120	**0.12**	**0.057**
240	**0.44**	**0.059**
500	**0.17**	**0.063**
1000	**0.59**	**0.060**



4. Result

In the Linear search algorithm , we can clearly see that using the multiprocess code is better than the multi thread code. The multi process code takes less time as compared to the multithreading.

- In the binary search algorithm we find that the multiprocess code performs better than the multithread code. The number of processes increase as the array size increases.

- Multiprocessing Bubble sort has a faster running time as the number of processors grows. While multithreading has a slower processing time. In terms of efficiency , multiprocessing is more efficient when the elements are smaller is number.

5. Conclusion

By the observations drawn from the above , we can conclude that multithreading will slow the process of sorting. Multiprocessors increase the number of processes that are being performed simultaneously.
The actual sorting speed is determined by the number of comparisons and memory operations. A single processor has to do all of this one at a time , so the threads simply add overhead from the context switching hence increasing the amount of time taken.
Multiprocessing can speed up the work by a constant factor. Assuming that the computer has a constant number of processes (p), then in the best case scenario the sorting will take place q times faster. This happens because the actual sorting part is now divided across multiple cores. The challenge here is the identification and the splitting of various tasks in different cores.

To conclude, multithreading slows down the algorithms by adding overhead due to context switching where as multiprocessing may speed up the same.

CODE

Multiprocess.c

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <sys/wait.h>
#include "Processes/linearSearchProcess.c"
#include "Processes/binarySearchProcess.c"
#include "Processes/bubbleSortProcess.c"
void bubbleSortRun()
{
   int shmid;
   key_t key = IPC_PRIVATE;
```

```
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");  _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat");  _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startingTime and endingTime for calculating time for bubble sort
    clock_t startingTime, endingTime;
    startingTime = clock();
    // Sort the created array
    bubbleSort(shm_array, length);
    endingTime = clock();
    // Check if array is sorted or not
    isSorted(shm_array, length);
    for (int i = 0; i < length; i++)
    {
        printf("%d ", shm_array[i]);
    }
    printf("\n");
    /* Detach from the shared memory now that we are done using it. */
    if (shmdt(shm_array) == -1)
    {
        perror("shmdt");  _exit(1);
    }
    /* Delete the shared memory segment. */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl");  _exit(1);
    }
    // calculating time taken in applying merge sort
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);  exit(0);
}
void linearSearchRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
    size_t SHM_SIZE = sizeof(int) * length;
    // Create the segment.
    if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
    {
        perror("shmget");   _exit(1);
    }
    // Now we attach the segment to our data space.
    if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
    {
        perror("shmat");  _exit(1);
    }
    // Create a random array of given length
    srand(time(NULL));
    TakingInput(shm_array, length);
    printf("\n");
    // startingTime and endingTime for calculating time for merge sort
    clock_t startingTime, endingTime;
    startingTime = clock();
    int toSearch;
    printf("Enter the key to search:");
    scanf("%d", &toSearch);
    linearSearch(shm_array, length, toSearch);
    endingTime = clock();
    /* Detach from the shared memory now that we are done using it. */
    if (shmdt(shm_array) == -1)
    {
        perror("shmdt");  _exit(1);
    }
    /* Delete the shared memory segment. */
    if (shmctl(shmid, IPC_RMID, NULL) == -1)
    {
        perror("shmctl");  _exit(1);
    }
    // calculating time taken in applying linear search
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC); exit(0);
}
void binarySearchRun()
{
    int shmid;
    key_t key = IPC_PRIVATE;
    int *shm_array;
    int length;
    printf("Enter No of elements of Array:");
    scanf("%d", &length);
    // Calculate segment length
```

```
  size_t SHM_SIZE = sizeof(int) * length;
  // Create the segment.
  if ((shmid = shmget(key, SHM_SIZE, IPC_CREAT | 0666)) < 0)
  {
    perror("shmget");  _exit(1);
  }
  // Now we attach the segment to our data space.
  if ((shm_array = shmat(shmid, NULL, 0)) == (int *)-1)
  {
    perror("shmat");  _exit(1);
  }
  // Create a random array of given length
  srand(time(NULL));
  TakingInput(shm_array, length);
  printf("\n");
  // startingTime and endingTime for calculating time for merge sort
  clock_t startingTime, endingTime;
  startingTime = clock();
  // Sort the created array
  mergeSort(shm_array, 0, length - 1);
  printf("After performing merge sort: ");
  for (int i = 0; i < length; i++)
    printf("%d ", shm_array[i]);
  printf("\n");
  int toSearch;
  printf("Enter the key to search:");
  scanf("%d", &toSearch);
  binarySearch(shm_array, length, toSearch);
  endingTime = clock();
  /* Detach from the shared memory now that we are done using it. */
  if (shmdt(shm_array) == -1)
  {
    perror("shmdt");  _exit(1);
  }
  /* Delete the shared memory segment. */
  if (shmctl(shmid, IPC_RMID, NULL) == -1)
  {
    perror("shmctl");  _exit(1);
  }
  // calculating time taken in applying binary search
  printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC); exit(0);
}
int main()
{
  int choice;
  do
  {
    printf(" 1. Bubble Sort \n  2. Linear Search \n 3. Binary Search \n 4. Exit \n");
    printf("Enter your choice: ");
    scanf("%d", &choice);
```

```
      switch (choice)
      {

              case 1: bubbleSortRun();  break;
              case 2: linearSearchRun();  break;
              case 3: binarySearchRun();  break;
              case 4: exit(0);
              default:  printf("Invalid choice\n");
      }
   } while (choice != 4);
   return 0;
}
```

Multithread.c

```
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>
#include <time.h>
#include "Threads/linearSearchThread.c"
#include "Threads/binarySearchThread.c"
#include "Threads/bubbleSortThread.c"
void bubbleSortRun()
{
	int i;
	int N;
	printf("Enter the number of elements in the array: ");
	scanf("%d", &N);
	srand(time(NULL));
	// Filling the array with random integers
	for (i = 0; i < N; i++)
		arr[i] = rand() % 100;
	printf("Unsorted Array:\n");
	for (i = 0; i < N; i++)
		printf("%d ", arr[i]);
	printf("\n");
	pthread_t sorters[2];
	clock_t startingTime, endingTime;
	startingTime = clock();
	printf("Sorted Array :\n");
	for (i = 0; i < 2; i++)
	{
		pthread_create(&sorters[i], NULL, sortArr, (int *)i);
		pthread_join(sorters[i], NULL);
	}
	endingTime = clock();
	printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
	pthread_exit(NULL);
}
void linearSearchRun()
{
	pthread_t thread[THREAD_MAX];
	clock_t startingTime, endingTime;
	startingTime = clock();
	for (int i = 0; i < THREAD_MAX; i++)
		pthread_create(&thread[i], NULL, ThreadSearch, (void *)NULL); // create multiple threads
	for (int i = 0; i < THREAD_MAX; i++)
		pthread_join(thread[i], NULL); // wait untill all of the threads are completed
	endingTime = clock();
	if (flag == 1)	printf("Key found in the array\n");
	else  printf("Key not found\n");
	printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
```

```c
    pthread_exit(NULL);
}

void binarySearchRun()
{
    pthread_t threads[MAX_THREAD];
    clock_t startingTime, endingTime;
    startingTime = clock();
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_create(&threads[i], NULL, binary_search, (void *)NULL);
    for (int i = 0; i < MAX_THREAD; i++)
        pthread_join(threads[i], NULL); // wait, to join with the main thread
    endingTime = clock();
    if (found == 1)   printf("Key found in the array\n");
    else   printf("Key not found\n");
    printf("Time taken: %f\n", (endingTime - startingTime) / (double)CLOCKS_PER_SEC);
    pthread_exit(NULL);
}
int main()
{
    int choice;
    do
    {
        printf(" 1.  Bubble Sort\n 2. Linear Search\n 3. Binary Search\n 4. Exit \n");
        printf("Enter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
        case 1: bubbleSortRun();break;
        case 2:linearSearchRun(); break;
        case 3: binarySearchRun();break;
        case 4: exit(0);
        default: printf("Invalid choice\n"); break;
        }
    } while (choice != 4);
    return 0;
}
```

REFERENCES

1. Operating System Concepts Ninth Edition by Abraham Silberschatz, Peter Baer Galvin, Greg Gagne
2. Wikipedia
3. GeeksforGeeks
4. TutorialPoint