Operating Systems–1: Assignment 2, Report

Name : Ahmik Virani

Roll No. : ES22BTECH11001

Part 1 : Low Level Design

The first part of the code consists of include files, they are listed below

```
#include<unistd.h>
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
#include<fcntl.h>
#include<sys/shm.h>
#include<sys/stat.h>
#include<sys/mman.h>
#include<stdbool.h>
#include<math.h>
#include<pthread.h>
```

Next we have global variables:

Here N is the number below(and equal to) which we are checking for vampire numbers

M is the number of threads

array is allocated memory in the main function

```
int N, M;
int *array;
```

Next we have the function to check if a number is vampire or not

It takes input x, returns true if number is vampire, false if not vampire

```
bool isVampireNumber(int x)
```

```
bool isVampireNumber(int x)
{
    int ct = 0;                     //counts he number of digits in the number x
                                    //This is initialized to zero

    int digitCounter[10];           //An array to store how many digits of each number are present
                                    //Example : digitCounter[0] stores how many zeros are present in x

    for(int i =0; i<10; i++)        //A loop to initialize the values in the digitCounter array
    {
        digitCounter[i] = 0;        //All the values are initialized to zero
    }

    int dummy = x;                  //An integer variable called dummy
                                    //This stores the value of x initially
                                    //This will be used to count the number of digits in x

    /*
    We run a loop on the dummy variable to count the number of digits in x

    loop initialization :
        we initialized the value of dummy to x

    loop moderation :
        The loop will store the units digit in the dummy
        After which, the loop will divide the dummy by 10, as we counted the units digit

    loop termination :
        The loop continues till all the digits are counted and stored
        Say finally the loop runs for dummy = 2
        dummy = dummy / 10 causes dummy to be updated to zero
        This means that the loop will now end
    */
    while(dummy != 0)
    {
        int j = dummy % 10;         //We store the units digit of the dummy variable in a local varaible called j
        digitCounter[j]++;          //Then we increase the digitCounter array indexed at j by 1
                                    //For example the number was 1260
                                    //j will store 0
                                    //digitCounter[0] = 1 (after incrementation)

        dummy = dummy / 10;         //As we have now stored the value of units digit, we divide by 10
                                    //This division by 10 ensures that we are now excluding the last digit which we just counted and stored

        ct ++;                      //We then increment ct by 1

    }
```

```
/*
The ct variable stores the number of digits in x
We know that if the number of digits is odd or less than equal to 2, then the number is not a vampire number
Thus, if the number of digits is odd or less than or equal to 2, then we return false and exit the function
*/
if(ct % 2 != 0 || ct <= 2)
{
    return false;               //Return false if the number of digits is odd or less than or equal to 2
}

ct = ct/2;                      //We know that we need numbers that are of half the size of input number x
                                //Which means we need number of size ct/2
                                //We update the ct value to ct/2

/*
This is a loop to check the divisors of the input x, int the range of 10 ^ (ct - 1) to 10 ^ ct

loop initialization:
    The limits are used because we need only the divisors which are of half the size of the original number
    For example if x = 1260, initially ct = 4, it updates to 2 later on. We need divisors in the range 10 to 99, i.e 10 ^ 1 to 10 ^ 2

loop moderation:
    For each divisor, we check the other pair of divisor as well, example if 21 is divisor of 1260, we even check 60
    We count what digits are present in both of them and store them togetether in an array
    If all the digits in this array match with the original array then we are done, the number is a vampire number
    If not, then we continue till we check all divisors

loop termination
    Either we return true when one of the divisor pair satisfies the condition for vampire numbers
    Or we loop through all the numbers that are half the size of the input number and less than or equal to half of it(The maximum vaulue of divisor)
*/
for(int i = pow(10, ct - 1); i < pow(10, ct) && i <= x / 2; i++)
{
    if (x % i == 0)                                         //If a number i is a divisor of x, then we will check it
                                                            //If not, then me move on to the next value of i
    {
        int dummyDigitCt[10];                               //An array that will store the digits present in the divisor pair
        for(int j =0; j<10; j++)                            //We run a loop to initialize the value at all the indexes of the array dummyDigitCt
        {
            dummyDigitCt[j] = 0;                            //We initialize all the values of this array to 0
        }

        int temp = i;                                       //We create a temporary variable to check the digits of i, which is one of the divisor of x

        /*
        A loop to check the number of digits in i

        loop initialization:
            The value of temp is initialized to i

        loop moderation :
            The loop will store the units digit in the temp
            After which, the loop will divide the temp by 10, as we stored the units digit in the array

        loop termination
            after the value of temp reaches zero(it will reach zero because it is an integer value), the loop will end
        */
```

```
        while(temp != 0)
        {
            int j = temp % 10;      //We store the units digit of temp as a local variable called j
            dummyDigitCt[j]++;      //We will update the dummyDigitaCt array at index j by 1

            temp = temp / 10;       //We will next divide it by 10
        }

        int count = 0;              //This counts how many values are same in the matrices digitCounter and dummyDigitCt

        /*
        This loop runs over all the indexes of digitCounter and dummyDigitCt and compares their corresponding values at each index

        loop initialization:
            loop starts at index 0

        loop moderation:
            loop will check each corresponding value, if they are equal, loop will increment counter variable by 1

        loop termination:
            once all the indexes are checked, loop will end
        */
        for(int j = 0; j<10; j++)
        {
            if(dummyDigitCt[j] == digitCounter[j])              //Compare the values at the index j for both the arrays
                count++;                                        //If they are equal, increment counter by 1
        }

        if(count == 10 && (i%10!=0 || (x / i) % 10 != 0))      //If count equals 10 and both of the fangs are not multiple of 10, the return true
        {
            return true;
        }
    }
}

return false;
}
```

The functioning of the vampire function is described here:

The function takes in a number, call it x,

It first counts the number of digits in a variable called ct and also stores the number of each digit in an array called digitCounter

example is x was 11223333,

then ct = 8

digitCounter[1] = 2

digitCounter[2] = 2

digitCounter[3] = 4

rest all are zero

Lets say ct was odd, then it is not vampire number, so we return false from the function.

If number of digits is not odd, then we check all the divisors of the given number that are half the size of the original number, in the above example it would be of size 4, i.e. we would check numbers from 10^3 to 10 ^4- 1.

Then we check the divisor pairs, i.e., if 'i' is divisor of x, then we calculate 'x / i'.

If all the numbers in 'i' and 'x/i' equal to the original number, then it is a vampire number.

Example : x = 1260

say i = 21 then x / i = 60

clearly all the digits match, combining 21 and 60 we have one 1, one 2, one 6, one 0, and in 1260 also we have the same, thus it is a vampire number.

Please note that both the fangs of the vampire number cannot be divisible by 10 together, so the additional condition was added.

```
if(count == 10 && (i%10!=0 || (x / i) % 10 != 0))
```

Next we have the runner function, where the thread begins its function.

This is where the major difficulty arose.

In this we had to assign values to each thread to increase efficiency

How i did this was, I assigned the values in a modular fashion, i.e. for example say N = 10 and M = 3, then

```
for(int i = k; i <= N; i += M)
```

thread 1 - 1, 4, 7, 10

thread 2 - 2, 5, 8

thread 3- 3, 6, 9

```
void *runner(void *param)
```

One important thing to keep in mind was that the local buffer size is not N/M but N/M + 1, because C interprets N/M(in above example) as 3, but clearly thread 1 stores 4 values, so size should be N/M + 1

```
int localBuffer[N/M + 1];
```

Keeping the above conditions in mind, the code is as follows

```c
void *runner(void *param)
{
    int k = (*(int *)param);            //A variable which corresponds to the threadNumber array value in the main function

    int localBuffer[N/M + 1];           //A local buffer to store the values of vampire numbers

    int z = 0;                          //A variable to calculate the number of vampire numbers in the given numbers this thread checks

    /*
    The following loop will check if each number is vampire or not and then store it in the local buffer

    loop initialization:
        The loop will start at the variable value of k and moluarly loop over all values till N

    loop moderation:
        If a number is vampire, it will be stored in the local buffer

    loop termination:
        The loop will terminate if all the values are checked
    */
    for(int i = k; i <= N; i += M)
    {
        if( isVampireNumber(i) )            //Function call to check if a number is vampire or not
        {
            localBuffer[z] = i;             //If the number is vampire, store it in the local buffer
            z++;                            //Increment the value of z by 1
        }
    }

    /*
    The following loop will store all the values in the global array at the corresponding value of k

    loop initialization:
        We will start at zero, and go up to z, i.e all the vampire numbers that were present in this thread

    loop moderation:
        For each vampire number, the (K + 1) number will be stored in the global array

    loop termination:
        The loop will terminate if all the values are added
    */
```

```c
    for (int i = 0; i < z; i++)
    {
        int value = localBuffer[i];     //Store the value of the local buffer into a temporary variable
        array[value] = k+1;             //We add the value of index localBuffer[i], and store the value as threadNumber + 1
    }

    pthread_exit(0);                        //Exit the thread
}
```

Then we finally have the main function

This is where I did all the file operations, allocated memory to the array(which was defined globally), made threads.

```c
int main()
```

```c
int main()
{
    FILE *fileIn = fopen("InFile.txt", "r");          //Open the input file named "InFile.txt" in read only mode

    if(fileIn == NULL)                                 //Check if the file in NULL
    {
        perror("error opening file\n");                //If it is NULL, which means it does not exist, throwing an error message
        return 1;                                      //A value more than zero is returned to show that something went wrong in the execution
    }

    /*
    The following two lines scan the values of N and M from the input file
    The first value scanned is N
    The second value scanned is M
    */
    fscanf(fileIn, "%d", &N);                          //value N is scanned as an integer from the input file
    fscanf(fileIn, "%d", &M);                          //value M is scanned as an integer from the input file

    fclose(fileIn);                                    //Once all the input values are taken, we close the input file

    pthread_t tid[M];                                  //Creating 'M' thread identifiers, one for each thread

    pthread_attr_t attr[M];                            //creating 'M' set of thread attribures, one for each thread

    array = (int *)calloc(N + 1, sizeof(int));             //Dynamically allocating all values in an array of size N to zero

    int threadNumber[M];                               //An array to store the thread number, example the first thread will have its value in threadNumber array as zero, thread 2 will have 1 and so on

    /*
    Loop for assigning the thread number as well as initializing its attributes to default
    We are alos creating a thread, and using the function runner for execution
    The function runner will take input the address of thread number

    loop initialization:
        We start at thread one, whose corresponding threadNumber is 0

    loop moderation:
        We assign the threadNumber for each thread
        We initialize its attributes to the default ones
        we create a thread for it

    loop termination
        the loop ends once we have created all M threads
    */
    for(int i = 0; i<M; i++)
    {
        threadNumber[i] = i;                                        //Assigning threadNumber
        pthread_attr_init(&attr[i]);                                //Assigning default attributes
        pthread_create(&tid[i], &attr[i], runner, &threadNumber[i]);    //Creating a thread
    }
```

```c
    /*
    The following loop is to wait for each thread to exit

    loop initialization:
        We start by waiting for thread 1, whose corresponding threadNumber is 0, to exit

    loop moderation:
        We wait for wach thread to exit one by one before the next thread starts execution

    loop termination:
        the loop ends once all the threads have finished execution
    */
    for(int i = 0; i<M; i++)
    {
        pthread_join(tid[i],NULL);                     //Wait for a thread to exit
    }

    int totalVampireNumbers = 0;                                   //A variable to store the total number of vampire numbers before N

    FILE *file = fopen("OutFile.txt", "w");            //Open the output file in write mode
    if(file == NULL)                                   //If file is NULL, i.e. does not exist show an error message
    {
        perror("Error opening file\n");                //Show the error messahe
        return 2;                                      //Return a positive value more than zero to tell the OS that an error occured
    }

    /*
    The following loop prints the data found by th thread in the output file

    loop initialization:
        We start at the local variable i = 0

    loop moderation:
        for each value, we print the value and the thread that found it

    loop termination:
        once all the numbers have been traversed, the loop ends
    */
    for (int i = 0; i < N + 1; i++)
    {
        if (array[i]!=0)
        {
            totalVampireNumbers++;                                              //If the array does not store 0, we increment the total variable
            fprintf(file, "%d: Found by Thread %d\n", i,array[i]);      //We print the data in the output file
        }
    }

    fprintf(file, "Total Vampire numbers: %d\n", totalVampireNumbers);                //We print the total number of vampire numbers in the output file

    fclose(file);                                                   //We close the output file

    return 0;
}
```

How threads were created :

First declared thread ids and attributes for each thread

```
pthread_t tid[M];

pthread_attr_t attr[M];
```

Introduced a thread number array to identify each thread

```
int threadNumber[M];
```

And then running a loop, where we initialize the attributes to default and create each thread to work in the runner function.

```
for(int i = 0; i<M; i++)
{
    threadNumber[i] = i;
    pthread_attr_init(&attr[i]);
    pthread_create(&tid[i], &attr[i], runner, &threadNumber[i]);
}
```

And finally waiting for each thread to complete

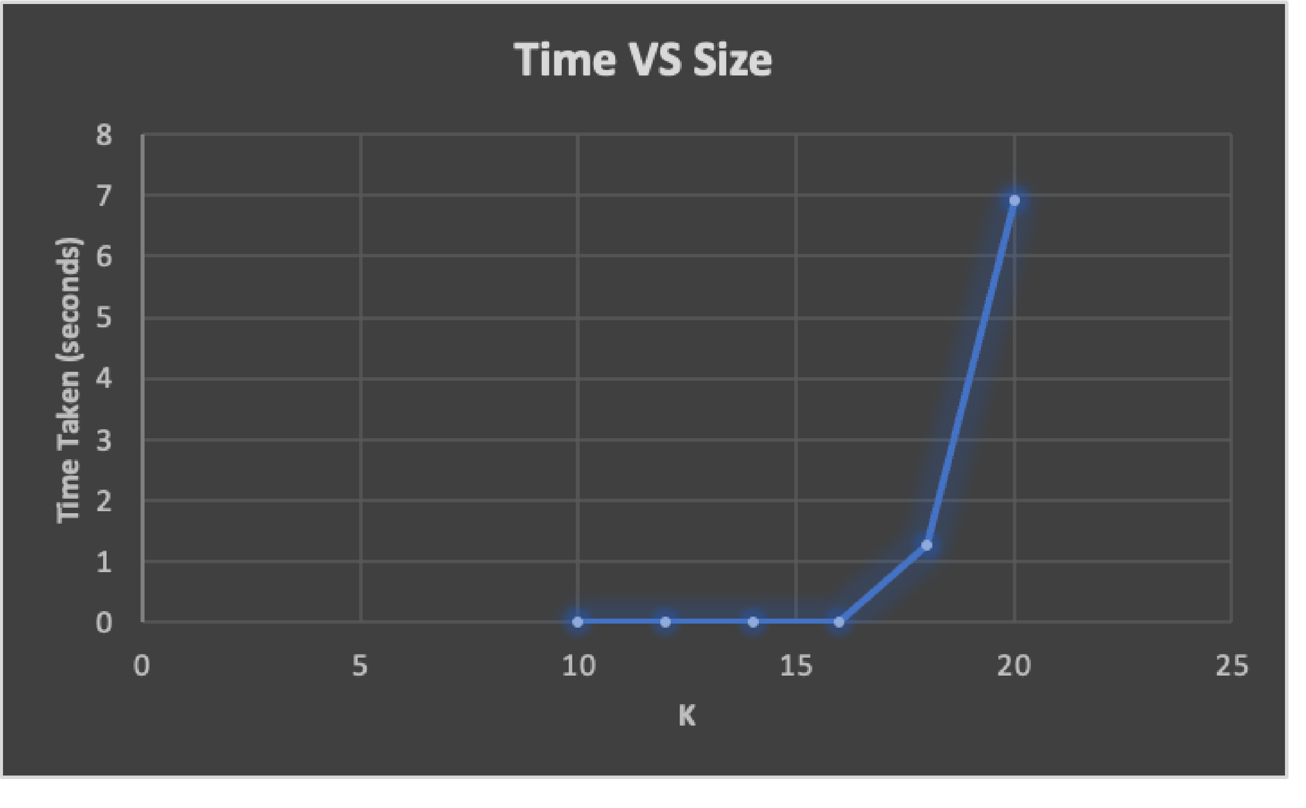
```
for(int i = 0; i<M; i++)
{
    pthread_join(tid[i],NULL);
}
```

Part 2: Performance of the program

1. Time vs Size

Here the number of threads were fixed at 8 and N = 2 ^ K

K	time(seconds)
10	0.004
12	0.006
14	0.016
16	0.023
18	1.26
20	6.911



Clearly we can see that as the size increases, the time taken also increases.

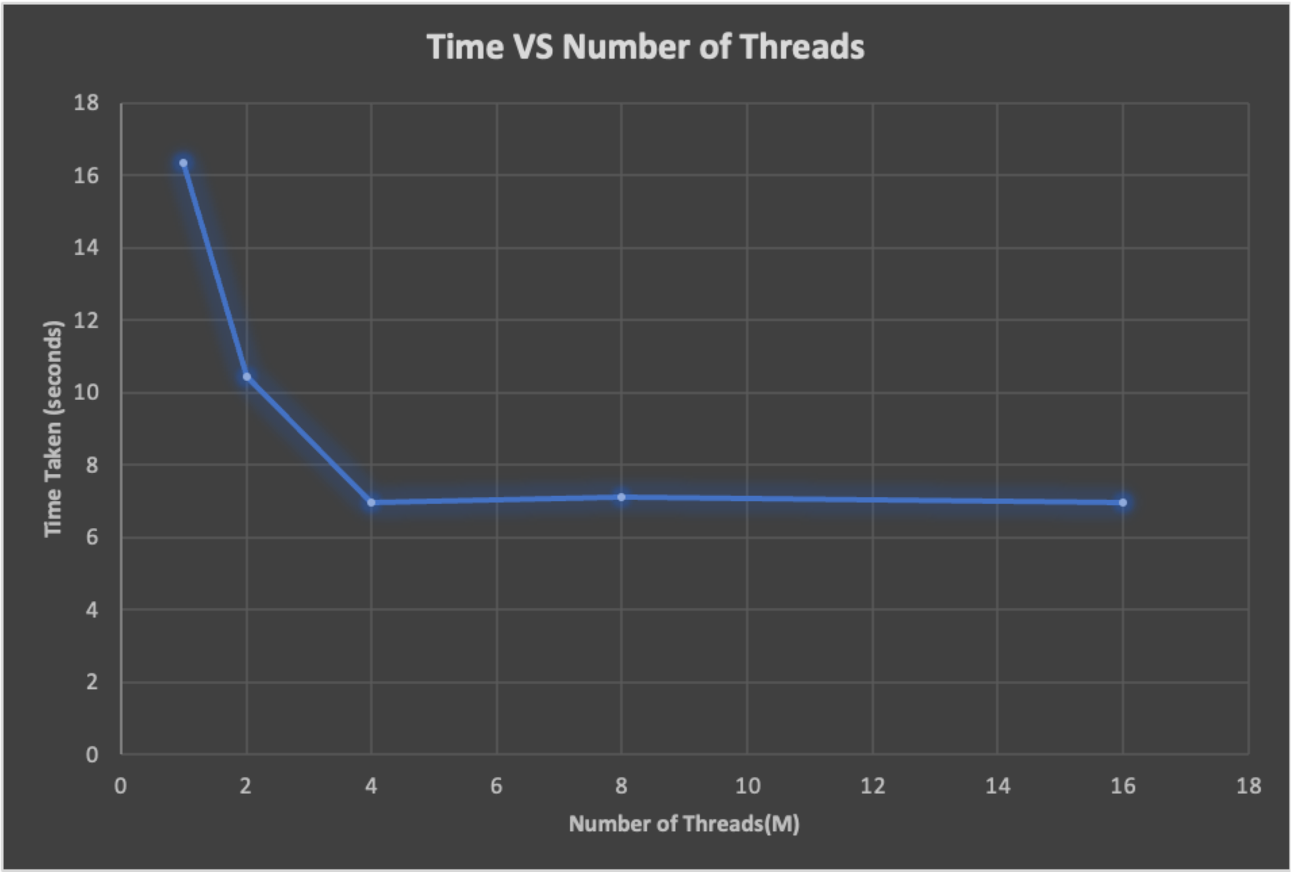
Initially when the size if small, the time taken is almost constant, however when it goes to large values, the time grows exponentially.

Note that this happens because if the size of input increases, then the work done has to be more, and the number of threads are fixed, so each thread has to do more work, and so time increases

2. Time vs Number of Threads

Here the size N was fixed to 10,00,000 (10 lakhs)

	time(seconds)	
1	16.35	
2	10.425	
4	6.94	
8	7.09	
16	6.947	



Clearly we can see that as the number of threads increases, the time taken decreases.

We can see that for small number of threads, it takes more time to run compared to when more threads are working. However after a certain point of time, even if you increase the number of threads, the time taken almost remains constant.

This happens because for the same input size, as you increase the number of threads, the work done is split among each thread nearly equally and so the time taken decreases overall.