NAME: ADWAIT S PURAO

UID: 2021300101

BATCH: B2

Experiment no.: 4

Theory:

What is a Thread?

A thread is a single sequence stream within a process. Because threads have some of the properties of processes, they are sometimes called *lightweight processes*.

Multithreading in C refers to the use of many threads inside a single process. Each thread serves a separate function. Multithreading operates concurrently which means numerous jobs may be executed simultaneously. Multithreading also minimizes the consumption of resources of the CPU. Multitasking can be divided into two types: process based and thread based. By saying Multithreading, it means that there are at least two or more than two threads in the process that runs simultaneously. For understanding multithreading in c, first, we need to learn about what is thread and what is a process. Lets us see these topics for better understanding.

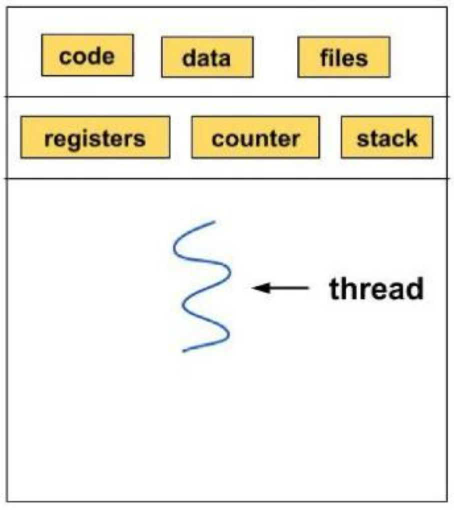
What are Thread and Processes?

- Threads: A thread is a basic unit of execution of any process. A program comprises many processes and all the processes comprise much simpler units known as threads. So, the thread can be referred to as the basic unit of a process or it is the simpler unit that tother makes the CPU utilization. To know more about threads in multithreading, you can visit Threads in Operating System.

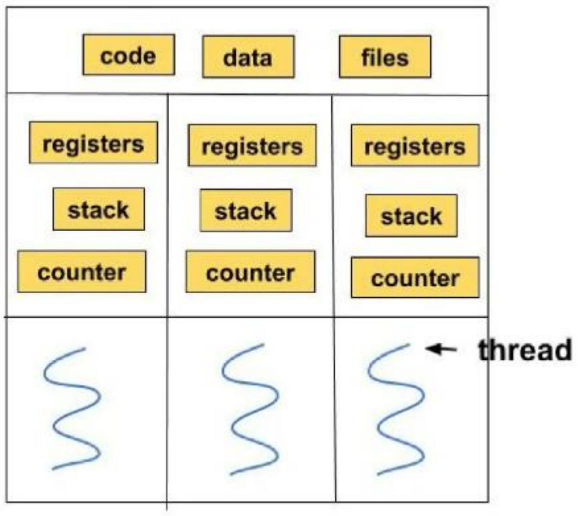
A thread comprises the following thing:-

1. A thread ID: It is a unique ID of the thread that is created during the creation of the thread and it remains for the whole life of that particular thread.
2. Program Counter: It is a value that loads into the hardware.
3. A registered set: It is a set of general registers.
4. A stack: It is the memory of that particular thread.

Apart from these, if two threads are working inside the same process, they share their code, their data section, and also share the other resources operating system like file open and signals. A traditional process that is also called a heavyweight process can control a single thread. But, if we talk about a multi-thread of control, it is capable of opening and executing more than one task at the same time. That is why threads become useful and how using threads makes the system much more efficient. Let us see how multithreading in c affects a process by using the diagram given below.



Single-threaded process



Multi-threaded process

Here, the difference between single and multithreading in c is depicted. In the first diagram, it is a **single-threaded process**. So, the whole block content shown in the diagram like code, data, etc is considered a single process and the process contains only one thread. It means this process will perform only one task at a time. But in contrast to this, the second diagram is a multithreading process. There are also tasks represented by blocks like **code, stack, data, files,** etc but it is having multi-threads over there and each of these threads has there own stack and registers. In this case, this process can perform multiple tasks at a time and is hence called the Multithreading process.

There are **two** types of Thread:

1. **User-level thread**: As the name suggests, it is at the **user level**. Its information is not shared with the kernel.
2. **Kernel level thread**: It is the type of thread that is operated by the **operating system** of the system and the kernel from which the thread belongs.

Process The process can be defined as the series of actions that are done to execute a program. When a program is run, it is not directly gets executed. It gets divided into some steps of small actions and these actions get executed one by one in a systematic way which ultimately leads to the execution of a process. When a process is divided into simpler actions, it is known as the **" clone or child process"** and the main process is the **"parent"** process. All the processes consume some space in the memory and these spaces are not interchanged with other processes.

There are some stages of a process that it gets before execution.

1. **NEW-** this is the state where a new process is created.
2. **READY-** this is the state when a process is ready and waiting for the allocation of the processor.
3. **RUNNING-** this is the state when the process is running.
4. **WAITING-** this is the state when the process is waiting for some event.
5. **TERMINATED-** this is the state when the process gets executed.

Why Multithreading in C?

Multithreading in the C concept can be used to improve the working of an application through **parallelism** . Suppose, if we are using a browser and we are working on many tabs in a browser window. Then, each of the tabs works parallelly and can be referenced as **Thread** . Suppose, we are using MS Excel, then one thread controls text formatting, and one thread processes input. So, multithreading in C makes it easy to perform more than tasks simultaneously. Thread is much faster to create. The switching of context between the threads is faster. Also, making communication between the threads is faster and the termination of threads is also easy.

How to Write Multithreading Programs in C?

In C language, there is not any built-in support for multithreading applications but it can do multithreading depending upon the operating system. The standard library used for implementing the concept of multithreading in C is known as **<threads.h>** but it is not possible to implement it using any known compiler yet. If we want to use multithreading in C then we must use some platform-specific implementations like the "POSIX" threads library by using the header file **pthread.h** . This is also known as "pthreads". We can create a POSIX thread in the following ways:

```
#include <pthread.h>
pthread_create (thread, attr, start_routine, arg)
```

Here, for making the thread executable, 'pthread_create makes a new thread. In this way, the multithreading in C can be done as many times as you want in your code. Following are the parameters and their description that we used above.

- thread: it is a unique identifier that is returned by the subprocess.
- attr: it is an opaque attribute that is used when we want to set thread attributes.
- start_routine: the thread will execute a routine when start_routine is created.
- arg: The argument that passes to the start_routine. If no argument is passed then, NULL will be used.

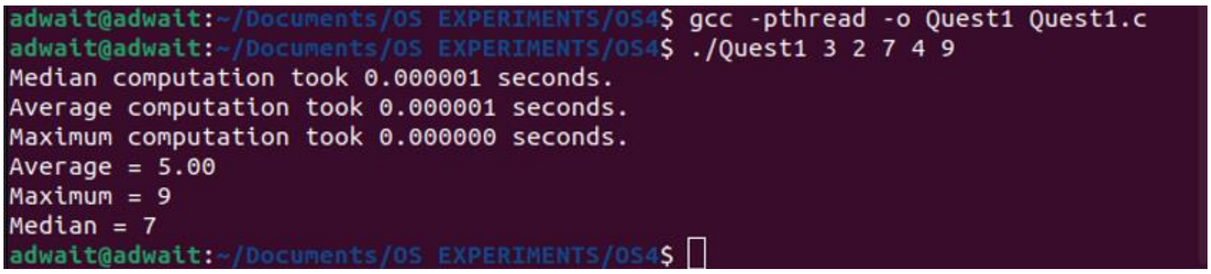
Question 1:

Write a multithreaded program that calculates various statistical values for a list of numbers. This program will be passed a series of numbers on the command line and will then create three separate worker threads. One thread will determine the average of the numbers, the second will determine the maximum value, and the third will determine the median value. (Using Pthread library) also record and display time taken by each thread

Code:

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

#define MAX_NUMBERS 100

struct stats {
    int* numbers;
    int count;
    double average;
    int maximum;
    int median;
};

void* compute_average(void* arg) {
    struct stats* data = (struct stats*)arg;
    double sum = 0.0;
    clock_t start_time = clock();
    for (int i = 0; i < data->count; i++) {
        sum += data->numbers[i];
    }
    data->average = sum / data->count;
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Average computation took %f seconds.\n", elapsed_time);
    pthread_exit(NULL);
}

void* compute_maximum(void* arg) {
    struct stats* data = (struct stats*)arg;
    int maximum = data->numbers[0];
    clock_t start_time = clock();
    for (int i = 1; i < data->count; i++) {
        if (data->numbers[i] > maximum) {
            maximum = data->numbers[i];
        }
    }
    data->maximum = maximum;
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Maximum computation took %f seconds.\n", elapsed_time);
    pthread_exit(NULL);
}

void* compute_median(void* arg) {
    struct stats* data = (struct stats*)arg;
    int n = data->count;
    int* numbers = data->numbers;
    clock_t start_time = clock();
    for (int i = 0; i < n; i++) {
        int count = 0;
        for (int j = 0; j < n; j++) {
            if (numbers[j] < numbers[i]) {
                count++;
            } else if (numbers[j] == numbers[i] && j < i) {
                count++;
            }
        }
        if (count == (n + 1) / 2) {
            data->median = numbers[i];
            break;
        }
    }
    clock_t end_time = clock();
    double elapsed_time = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Median computation took %f seconds.\n", elapsed_time);
    pthread_exit(NULL);
}

int main(int argc, char* argv[]) {
    if (argc <= 1 || argc > MAX_NUMBERS + 1) {
        printf("Usage: %s <number1> <number2> ... <number%d>\n",
argv[0], MAX_NUMBERS);
        return 1;
    }
    int count = argc - 1;
    int* numbers = malloc(count * sizeof(int));
    if (numbers == NULL) {
        printf("Error: Failed to allocate memory for numbers.\n");
        return 1;
    }
    for (int i = 0; i < count; i++) {
        numbers[i] = atoi(argv[i+1]);
    }
    struct stats data = { numbers, count, 0.0, 0, 0 };
    pthread_t threads[3];
    pthread_create(&threads[0], NULL, compute_average, (void*)&data);
    pthread_create(&threads[1], NULL, compute_maximum, (void*)&data);
    pthread_create(&threads[2], NULL, compute_median, (void*)&data);
    for (int i = 0; i < 3; i++) {
        pthread_join(threads[i], NULL);
    }
    printf("Average = %.2f\n", data.average);
    printf("Maximum = %d\n", data.maximum);
    printf("Median = %d\n", data.median);
    free(numbers);
    return 0;
}
```

Output:



```
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ gcc -pthread -o Quest1 Quest1.c
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ ./Quest1 3 2 7 4 9
Median computation took 0.000001 seconds.
Average computation took 0.000001 seconds.
Maximum computation took 0.000000 seconds.
Average = 5.00
Maximum = 9
Median = 7
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ 
```

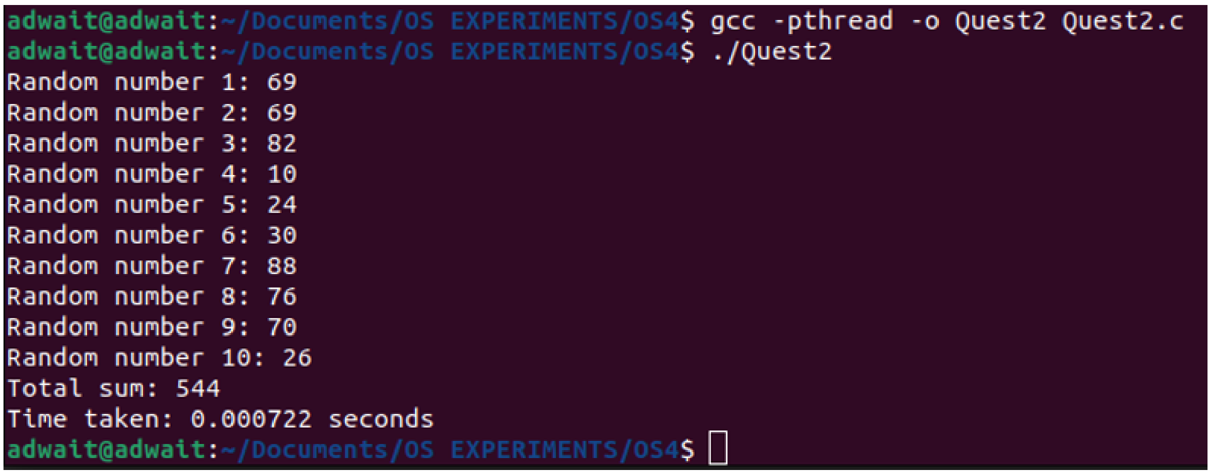
Question 2:

Write a program that creates a single thread that generates random numbers between 1 and 100. The thread should run for a specified number of iterations, printing each generated number to the screen. After the specified number of iterations, the thread should stop and the main program should print the sum of all the generated numbers also record and display the time taken by each thread

Code:

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

#define NUM_ITERATIONS 10

void* generate_random_numbers(void* arg);

int main() {
    pthread_t thread;
    srand(time(NULL));
    int sum = 0;
    clock_t start_time, end_time;
    start_time = clock();
    pthread_create(&thread, NULL, generate_random_numbers,
(void*)&sum);
    pthread_join(thread, NULL);
    end_time = clock();
    double time_taken = ((double)(end_time - start_time)) /
CLOCKS_PER_SEC;
    printf("Total sum: %d\n", sum);
    printf("Time taken: %f seconds\n", time_taken);
    return 0;
}

void* generate_random_numbers(void* arg) {
    int* sum = (int*)arg;
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        int rand_num = rand() % 100 + 1;
        printf("Random number %d: %d\n", i+1, rand_num);
        *sum += rand_num;
    }
    pthread_exit(NULL);
}
```

Output:



```
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ gcc -pthread -o Quest2 Quest2.c
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ ./Quest2
Random number 1: 69
Random number 2: 69
Random number 3: 82
Random number 4: 10
Random number 5: 24
Random number 6: 30
Random number 7: 88
Random number 8: 76
Random number 9: 70
Random number 10: 26
Total sum: 544
Time taken: 0.000722 seconds
adwait@adwait:~/Documents/OS EXPERIMENTS/OS4$ 
```

Conclusion:

In this experiment we learnt about threads and processes and their use for concurrent programming. I learnt about the pthread_create and pthread_join functions , When the **pthread_create** function is called, it creates a new thread that runs concurrently with the calling thread. The new thread starts executing the specified thread function, which can access and modify the same memory space as the parent thread. The parent thread can wait for the new thread to finish using the **pthread_join** function, or it can continue executing while the new thread runs in the background.

However, multi-threaded programming can also be more complex and error-prone, so it's important to be careful and use proper synchronization techniques to avoid race conditions and other problems.