Name: Subhendu Maji   Roll number: 18ETCS002121

Laboratory 3

Title of the Laboratory Exercise: Programs based on multithreaded programming

1. Introduction and Purpose of Experiment

   Multithreading is the ability of a processor or a single core in a multi-core processor to execute multiple threads concurrently, supported by the operating system. By solving students will be able to manipulate multiple threads in a program.

2. Aim and Objectives

   Aim

   - To develop programs using multiple threads.

   Objectives

   At the end of this lab, the student will be able to

   - Identify multiple tasks
   - Use threads constructs for creating threads
   - Apply threads for different/multiple tasks

3. Experimental Procedure

   i. Analyse the problem statement ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code iii. Implement the algorithm in C language iv. Compile the C program

   v. Test the implemented program vi. Document the Results vii. Analyse and discuss the outcomes of your experiment

4. Questions

   Create multithreaded programs to implement the following

   a) Display “Hello World” message by 3 different threads
   b) Display thread IDs by each thread

c) Create three threads;

 Thread1 add marks out of 10 from subject1 to subject5 of student 1, Thread2 adds from Subject1 to subject 5 of student 2 and Thread3 takes the sum from the Thread1 and Thread2 and decides who scored more marks. Display the total marks of the student (highest score) in parent process.
Instructions: Use a global variable to keep track of the sum from Thread1 and Thread2 and update it. Thread3 needs to wait until both Thread1 and Thread2 are done. Finally, parent process needs to wait until Thread3 is done. Get the return value from Thread3, and print the return value.

5. Calculations/Computations/Algorithms

Algorithm of a.

```
STEP 1: Start
STEP 2: define NUM_THREADS ← 3
STEP 3: ret_code ← 0, malloc threads array
STEP 4: for i=0 to NUM_THREADS, do
     4.1: ret_code ← create thread that executes callback function (pthread_create)
     4.2: if ret_code is positive, then display error message and exit.
STEP 5: Stop
```

```
void* callback(void* args);
STEP 1: Start
STEP 2: display "hello world"
STEP 3: Stop
```

Algorithm of b.

```
STEP 1: Start
STEP 2: define NUM_THREADS ← 3
STEP 3: ret_code ← 0, malloc threads array
STEP 4: for i=0 to NUM_THREADS, do
    4.1: ret_code ← create thread that executes callback function (pthread_create)
    4.2: if ret_code is positive, then display error message and exit.
STEP 5: Stop
```

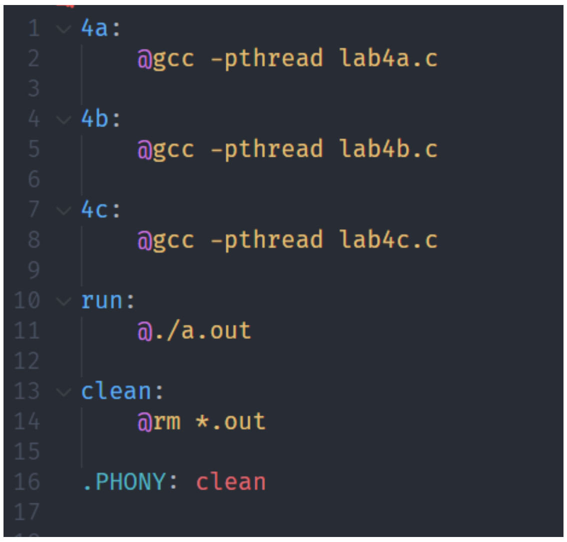
```
void* callback(void* args);
STEP 1: Start
STEP 2: display "hello world : id = " pthread_self()
STEP 3: Stop
```

Algorithm of c.

```
STEP 1: Start
STEP 2: define student1 marks = [], student2 marks = []
STEP 3: sums = malloc (2* sizeof(int))
STEP 4: create thread1, thread2, thread3
STEP 5: pthread_create (&thread1 , NULL, sum,  student1)
STEP 6: pthread_create (&thread2 , NULL, sum,  student2)
STEP 7: pthread_join (&thread1 ,&sums[0])
STEP 8: pthread_join (&thread2 ,&sums[1])
STEP 9: pthread_create(&thread3 ,NULL, compare, sums)
STEP 10: pthread_join (thread3 ,result)
STEP 11: print result
STEP 12: Stop
```

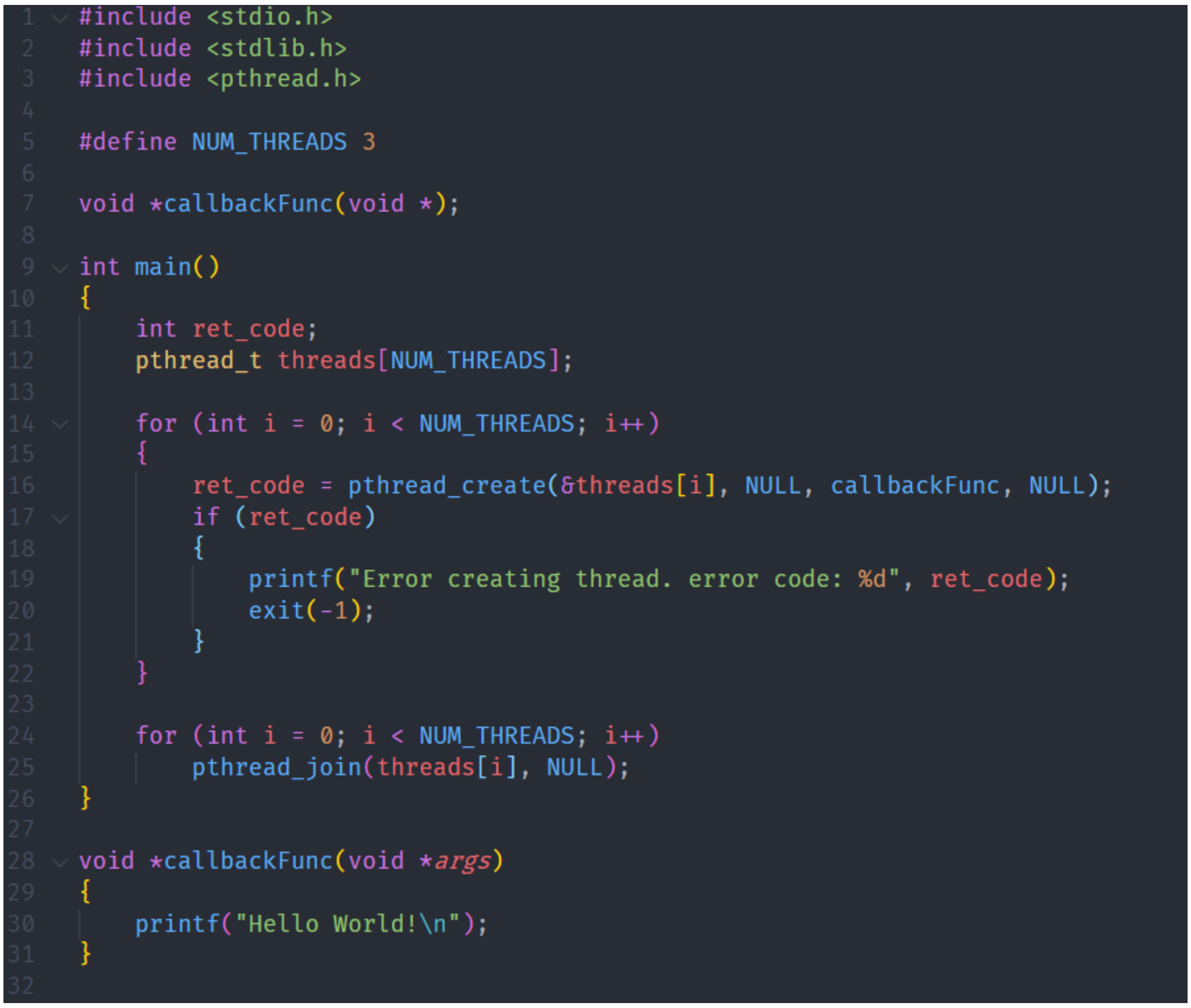
```
Void *sum (void *arg)

STEP 1: define *arr

STEP 2: for i = 0 to 5

      2.1 res = res+arr[i]

STEP 3: return res


Void *compare(void *arg)

STEP 1: define totals

STEP 2: if totals[0] >= totals[1]

      2.1 return totals[0]

STEP 3: else

      3.1 return totals[1]
```

6. Presentation of Results



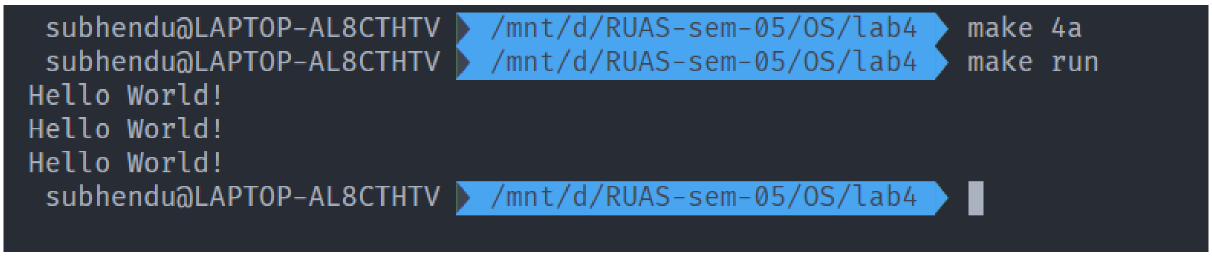
```
4a:
	@gcc -pthread lab4a.c

4b:
	@gcc -pthread lab4b.c

4c:
	@gcc -pthread lab4c.c

run:
	@./a.out

clean:
	@rm *.out

.PHONY: clean
```

Figure 8 makefile



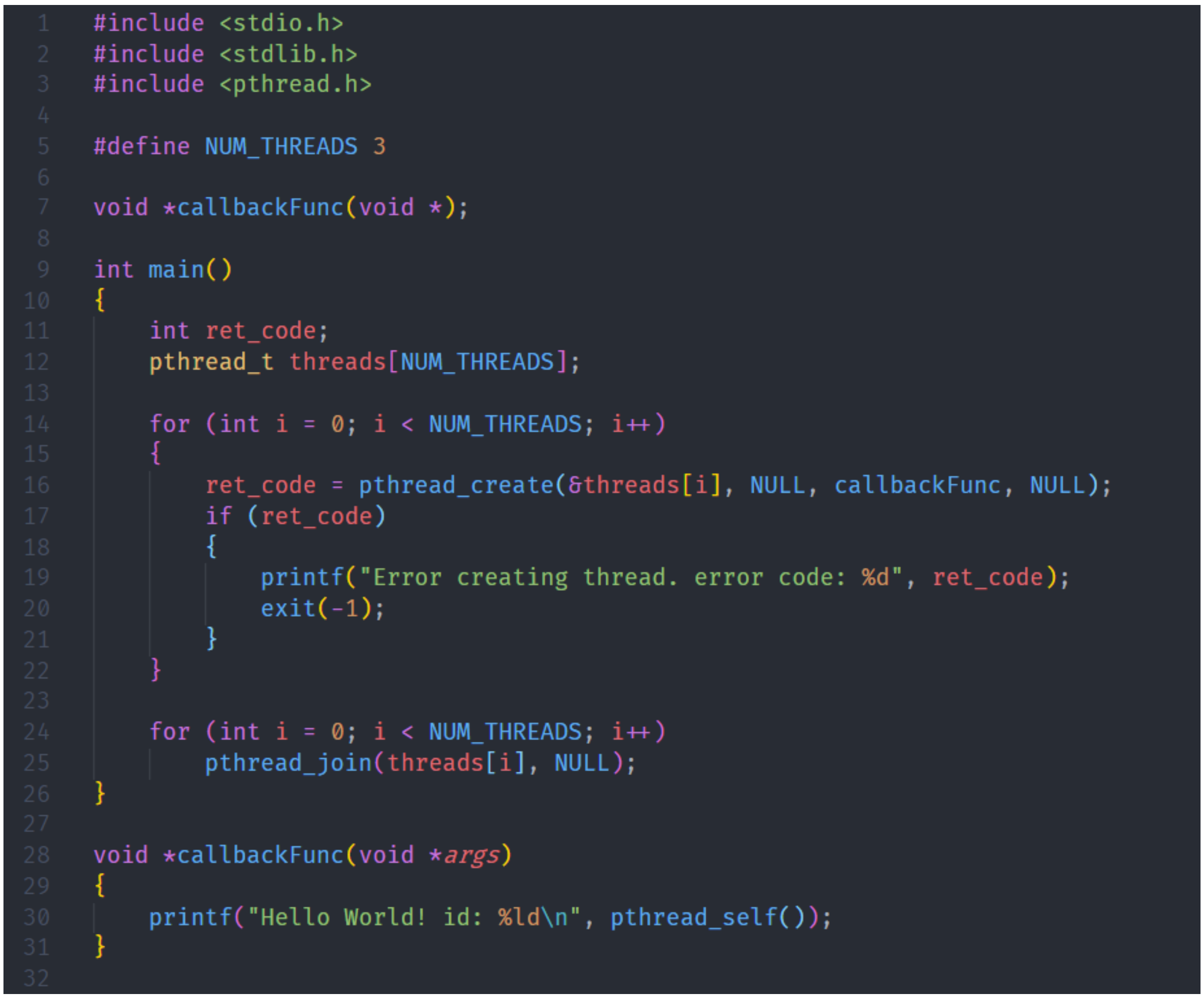
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 3

void *callbackFunc(void *);

int main()
{
    int ret_code;
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++)
    {
        ret_code = pthread_create(&threads[i], NULL, callbackFunc, NULL);
        if (ret_code)
        {
            printf("Error creating thread. error code: %d", ret_code);
            exit(-1);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
}

void *callbackFunc(void *args)
{
    printf("Hello World!\n");
}

```

Figure 9 Source Code of lab4a



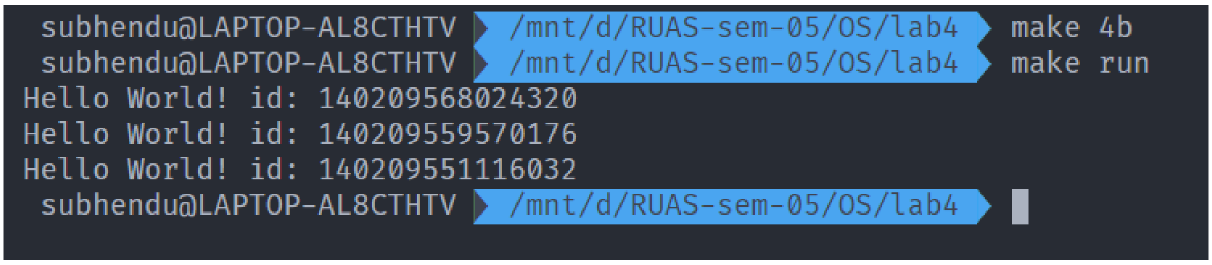
```
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make 4a
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make run
Hello World!
Hello World!
Hello World!
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4 
```

Figure 10 Execution of a



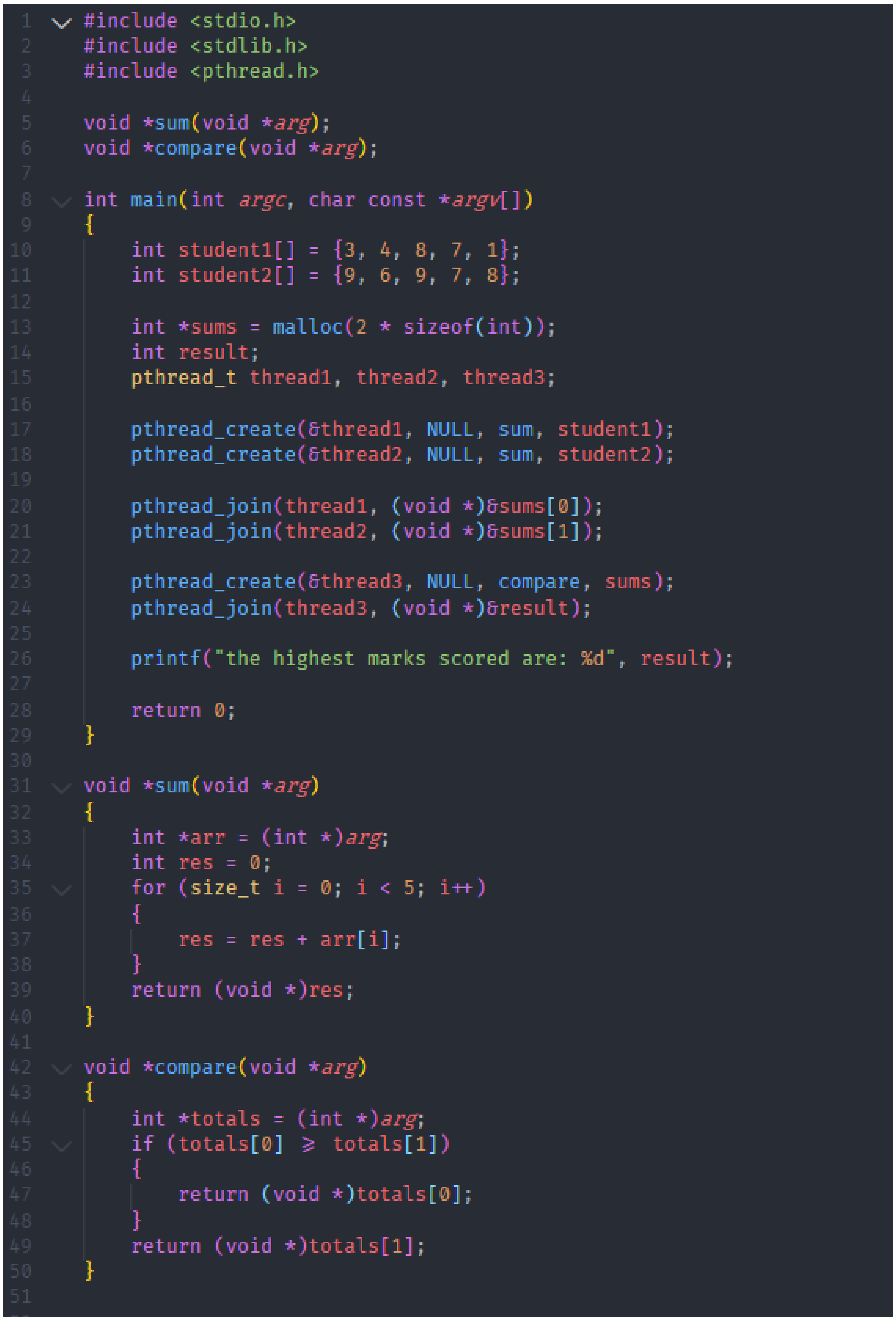
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 3

void *callbackFunc(void *);

int main()
{
    int ret_code;
    pthread_t threads[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++)
    {
        ret_code = pthread_create(&threads[i], NULL, callbackFunc, NULL);
        if (ret_code)
        {
            printf("Error creating thread. error code: %d", ret_code);
            exit(-1);
        }
    }

    for (int i = 0; i < NUM_THREADS; i++)
        pthread_join(threads[i], NULL);
}

void *callbackFunc(void *args)
{
    printf("Hello World! id: %ld\n", pthread_self());
}
```

Figure 11 Source code of lab4b



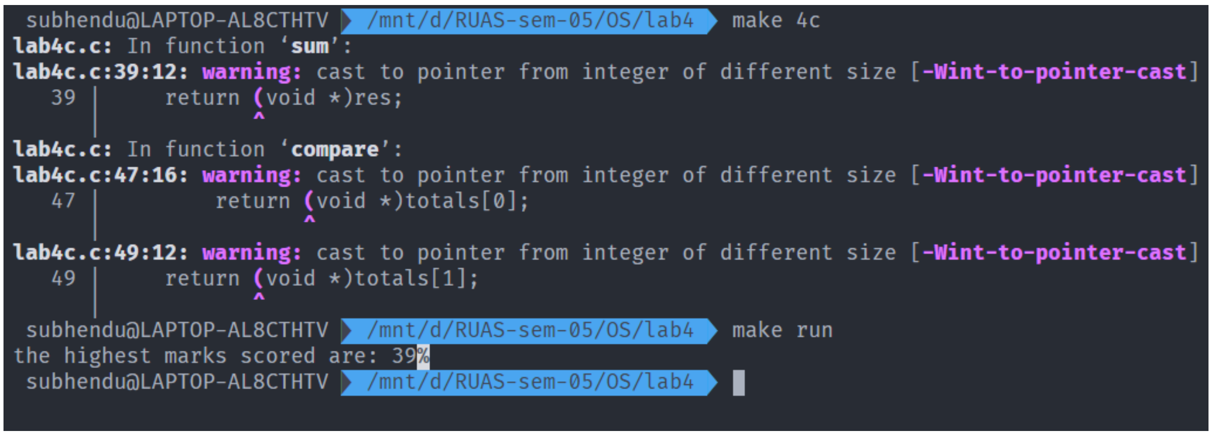
```
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make 4b
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make run
Hello World! id: 140209568024320
Hello World! id: 140209559570176
Hello World! id: 140209551116032
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4 
```

Figure 12 execution of b



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

void *sum(void *arg);
void *compare(void *arg);

int main(int argc, char const *argv[])
{
    int student1[] = {3, 4, 8, 7, 1};
    int student2[] = {9, 6, 9, 7, 8};

    int *sums = malloc(2 * sizeof(int));
    int result;
    pthread_t thread1, thread2, thread3;

    pthread_create(&thread1, NULL, sum, student1);
    pthread_create(&thread2, NULL, sum, student2);

    pthread_join(thread1, (void *)&sums[0]);
    pthread_join(thread2, (void *)&sums[1]);

    pthread_create(&thread3, NULL, compare, sums);
    pthread_join(thread3, (void *)&result);

    printf("the highest marks scored are: %d", result);

    return 0;
}

void *sum(void *arg)
{
    int *arr = (int *)arg;
    int res = 0;
    for (size_t i = 0; i < 5; i++)
    {
        res = res + arr[i];
    }
    return (void *)res;
}

void *compare(void *arg)
{
    int *totals = (int *)arg;
    if (totals[0] >= totals[1])
    {
        return (void *)totals[0];
    }
    return (void *)totals[1];
}
```

Figure 13 sourcce code of lab4c



```
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make 4c
lab4c.c: In function ‘sum’:
lab4c.c:39:12: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   39 |     return (void *)res;
      |            ^
lab4c.c: In function ‘compare’:
lab4c.c:47:16: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   47 |         return (void *)totals[0];
      |                ^
lab4c.c:49:12: warning: cast to pointer from integer of different size [-Wint-to-pointer-cast]
   49 |     return (void *)totals[1];
      |            ^
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4  make run
the highest marks scored are: 39%
 subhendu@LAPTOP-AL8CTHTV  /mnt/d/RUAS-sem-05/OS/lab4 
```

Figure 14 execution of c

7. Analysis and Discussions

A thread is the smallest sequence of programming instructions that can be managed independently by the scheduler, which is typically part of the operating system. The implementation of a thread and a process differs from operating system to operating system, but in most cases, a thread is a component of a process. Multiple threads can exist in one single process, executing concurrently and sharing resources such as memory. This is unlike processes, since different processes do not share the same memory.

As mentioned above, threads can be used to execute code within a program concurrently. But care should be taken to avoid race conditions, i.e., a situation that occurs when two threads access a resource simultaneously, which leaves the resource in an unstable state.

In C, threads are created and used using POSIX threads, or more specifically, the pthread library. This library includes code that allows the developer to spawn and run posix threads. The function used to create a thread is called pthread_create(). This thread has the following prototype:

```
pthread_create(pthread_t newThread, pthread_attr_t threadArgs,
void* (*callback)(void*), void* args)
```

The function takes in four arguments. The first, a variable of type pthread_t that defines the thread, the second, a variable of type pthread_attr_t that specifies thread arguments, third, a callback to execute, and fourth, the arguments to be passed to the callback.

To terminate a thread and merge the termination with the main thread, the function pthread_join() is used. It has the following definition:

```
pthread_join(pthread_t thread, void** return_status)
```

the function takes in two arguments. One, the thread to terminate, and the second, a pointer to store the return status or the value returned by the callback which the thread executes. This function makes the main thread wait till the child thread finishes execution.

8. Conclusions

In this experiment, the ways to create a thread, pass arguments to it, obtain the return status etc, was learned. The program that demonstrated these functionalities was written in C using the POSIX thread library.

9. Comments

   1. Limitations of Experiments

      The kernel doesn't know about the existence of threads in a process. So regardless of whether a process has 1000 threads or 1 thread, both of them get the same time slice. Moreover, if a thread uses blocking system calls, the whole process is blocked in the kernel.

   2. Limitations of Results

      The code has not been optimized to handle race conditions and other unfavourable scenarios that occur due to the use of threads.

   3. Learning happened

      The ways to create, run and manage threads in C was learned. The disadvantages were highlighted and the advantages were clearly seen.