

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) Create two threads;
 - Thread1 adds marks (out of 10) of student1 from subject1 to subject5, Thread2 adds marks of student2 from subject1 to subject 5. Main process takes the sum returned from the Thread1 and Thread2, decides who scored more marks and displays student with its highest score.

5. Calculations/Computations/Algorithms

caculate_total()

```
1. std::accumulate(subject_marks.begin(), subject_marks.end(), 0)
```

create_worker_threads()

```
1. for i = 0 to NUM_THREADS
2.     thread_data[i].thread_idx = i
3.     thread_data[i].student = students.at(i)
4.     pthread_create(&thrs[i], NULL, thread_worker_func, &thread_data[i])
```

join_worker_threads()

```
1. for i = 0 to NUM_THREADS
2.     pthread_join(thrs[i], NULL)
```

get_max_marks()

```
1. create_worker_threads()
2. calculate total() in each worker_thread
3. join_worker_threads()
4. print the max marks
```

6. Presentation of Results

```
#include <iostream>
#include <vector>
#include <numeric>

#include <unistd.h>
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>

// program parameters
#define TOTAL_SUBJECTS 10
#define NUM_STUDENTS 10

class Student {
public:
    Student(int nsubjects, bool rand_init);
    unsigned short get_mark_of_subject(int idx);
    unsigned int get_total();
    unsigned int get_id() { return this -> id; };

    virtual std::ostream& dump(std::ostream& out);
private:
    unsigned int id;
    std::vector<unsigned short> subject_marks;
};

Student::Student(int nsubjects = TOTAL_SUBJECTS, bool rand_init = true) : subject_marks(nsubjects), id(rand()) {
    if (rand_init) {
        for (auto& subject : subject_marks) {
            subject = rand() % 10;
        }
    }
}

std::ostream& Student::dump(std::ostream& out) {
    out << "{ ID: " << this -> id << ", Marks : [ ";
    for (auto mark : this -> subject_marks) {
        out << mark << ", ";
    }
    out << "], }";

    return out;
}

std::ostream& operator<<(std::ostream& out, Student student) {
    return student.dump(out);
}
```

```
}

unsigned short Student::get_mark_of_subject(int idx) {
    if (idx >= subject_marks.size()) {
        throw "idx out of range";
    }

    return subject_marks.at(idx);
}

unsigned int Student::get_total() {
    return std::accumulate(this -> subject_marks.begin(), this -
> subject_marks.end(), 0);
}

struct thread_data {
    unsigned int thread_idx;
    Student student;
    unsigned int total_marks;
};

void* my_thread_fun(void* vargp) {
    thread_data* this_data = (thread_data*) vargp;

    std::cout.sync_with_stdio(true);

    std::cout << "[ Thread ID : " << pthread_self() << " ]" << std::endl;
    std::cout << "Total Marks for Student ID : " << this_data -
> student.get_id() << " is : " << this_data -> student.get_total() << std::endl;

    this_data -> total_marks = this_data -> student.get_total();

    std::cout << std::endl;

    return NULL;
}

void* say_hello(void* vargp) {
    std::cout << "Hello World from Thread : " << *(pthread_t*)vargp << std::endl;

    return NULL;
}

#define NUM_THREADS NUM_STUDENTS

thread_data thr_data[NUM_THREADS];

int main(int, char**) {
```

```
std::cout.sync_with_stdio(true);

pthread_t thread_id[3];

// Say Hello from 3 different threads
for (short i = 0 ; i < 3 ; i++) {
    if ((pthread_create(&thread_id[i], NULL, say_hello, &thread_id[i])) < 0) {
        std::cerr << "Error creating thread" << std::endl;

        return EXIT_FAILURE;
    }
}

for (short i = 0 ; i < 3 ; i++) {
    pthread_join(thread_id[i], NULL);
}

std::cout << std::endl;

std::cout << "[ Running with " << NUM_THREADS << " Threads ]\n" << std::endl;
// create the students and randomly initialize them

std::vector<Student> students(NUM_THREADS);

for (auto student: students) {
    std::cout << student << std::endl;
}
std::cout << std::endl;

// create the threads
pthread_t thrs[NUM_THREADS];

for (short i = 0 ; i < NUM_THREADS ; i++) {
    thr_data[i].thread_idx = i;
    thr_data[i].student = students.at(i);

    int rc;
    if (rc = pthread_create(&thrs[i], NULL, my_thread_fun, &thr_data[i])) {
        std::cerr << "Error : Cannot create thread " << thrs[i] << std::endl;
        return EXIT_FAILURE;
    }
}

// block until all threads are completed
for (short i = 0 ; i < NUM_THREADS ; i++) {
    pthread_join(thrs[i], NULL);
}

// check who has the highest marks
```

```

int max_idx = 0;
int max_marks = thr_data[0].total_marks;

for (int i = 0 ; i < NUM_THREADS ; i++) {
    if (thr_data[i].total_marks > max_marks) {
        max_marks = thr_data[i].total_marks;
        max_idx = i;
    }
}

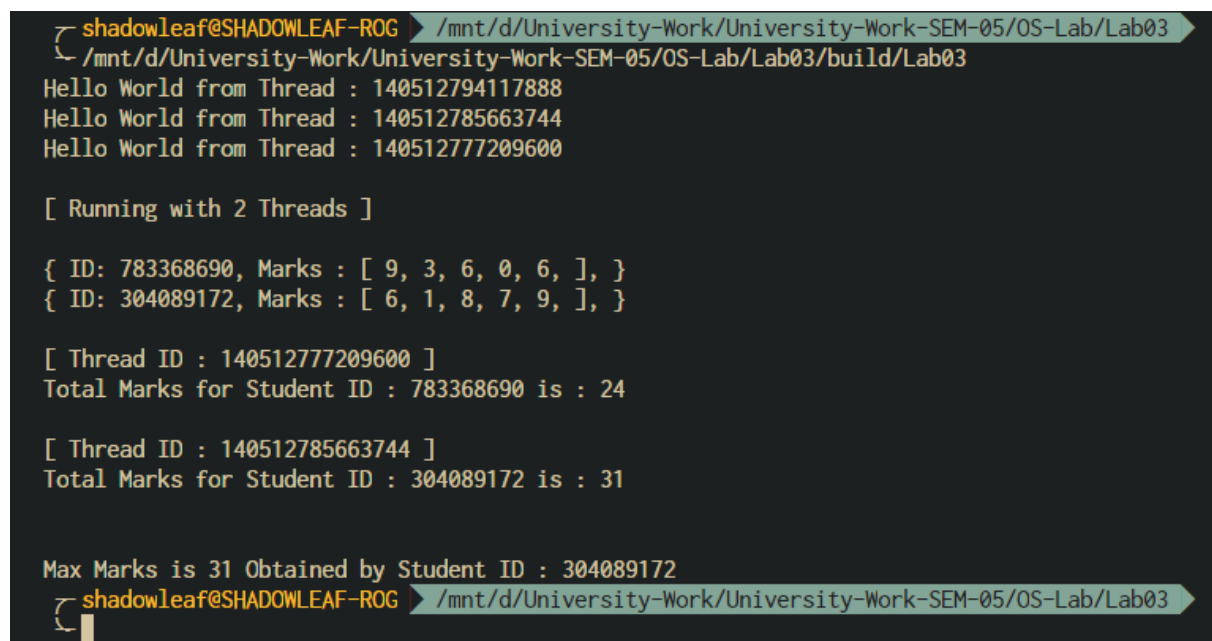
std::cout << "\nMax Marks is " << max_marks << " Obtained by Student ID : " << thr_data[max_idx].student.get_id() << std::endl;

pthread_exit(NULL);

return 0;
}

```

7. Analysis and Discussions



```

shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab03
/mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab03/build/Lab03
Hello World from Thread : 140512794117888
Hello World from Thread : 140512785663744
Hello World from Thread : 140512777209600

[ Running with 2 Threads ]

{ ID: 783368690, Marks : [ 9, 3, 6, 0, 6, ], }
{ ID: 304089172, Marks : [ 6, 1, 8, 7, 9, ], }

[ Thread ID : 140512777209600 ]
Total Marks for Student ID : 783368690 is : 24

[ Thread ID : 140512785663744 ]
Total Marks for Student ID : 304089172 is : 31

Max Marks is 31 Obtained by Student ID : 304089172
shadowleaf@SHADOWLEAF-ROG /mnt/d/University-Work/University-Work-SEM-05/OS-Lab/Lab03

```

Figure 0-1 Output

In Figure 0-1 Output 3 different threads with different threads ID's are generated to print Hello World, follows the program output for the student marks calculation, 2 threads are created since 2 students are taken into consideration for now, which can be later changed in the program parameters, each of the thread calculates the total marks of their respective student and stores the result in thr_data, this is the structure we passed as a reference to the thread function to have a sort of multiple arguments, which can be later dereferenced in the thread function to obtain the members of the structure. The

program then waits for all the threads to finish their job and the main thread then selects the student with the maximum marks and displays it to stdout.

8. Conclusions

pthread_create - create a new thread

```
#include <pthread.h>
```

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,  
void *(*start_routine) (void *), void *arg);
```

Compile and link with *-pthread*.

Description:

The **pthread_create()** function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine()*; *arg* is passed as the sole argument of *start_routine()*.

If *attr* is NULL, then the thread is created with default attributes.

The initial value of the new thread's CPU-time clock is 0

pthread_join - join with a terminated thread

```
#include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Compile and link with *-pthread*.

Description:

The **pthread_join()** function waits for the thread specified by *thread* to terminate. If that thread has already terminated, then **pthread_join()** returns immediately. The thread specified by *thread* must be joinable.

If *retval* is not NULL, then **pthread_join()** copies the exit status of the target thread.

9. Comments

1. Limitations of Experiments

The number of threads is assumed to be equal to the number of students, this will become inefficient when the number of students increase, the experiment can be further enhanced to distribute the work among the threads, such as giving a batch job to each thread. Furthermore, the work of the main thread can be reduced by using a critical section variable such as `max_marks`, which can be updated as soon as any of the thread finished.

2. Limitations of Results

The results are limited to the fact that the potential of multithreading wasn't used, the program has to be simulated for a fair amount of threads and students to get a satisfiable result, which can then be compared to single threaded application.

3. Learning happened

The use of POSIX threads was learnt, such as creating and destroying threads, and passing data to and from threads.

4. Recommendations

Load balance the threads for better performance by giving batch jobs and also reduce the work on the main thread.