MC 214 LAB-5 REPORT

Nisarg Suthar

202003030

[14 Oct, 2021]

EXAMPLE 1:

CODE:

```
/**
*@file Example 1
*@author Nisarg Suthar (202003030)
*@brief Example code for threading
*/

// header files
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

// Hola thread function
void * PrintHello(void * data)
{

      int my_data = *((int*)data);
      printf("\n Hello from new thread - got %d !\n", my_data);
      pthread_exit(NULL);              // exiting from the thread

}

// start of main
int main()
{
      int rc;
      pthread_t thread_id;       // var to store thread-id

      int num = 11;
      int *t = &num;
      rc = pthread_create(&thread_id, NULL, PrintHello, (void*)t);
      // creating a thread

      if(rc)
      {
```

```
        printf("\n ERROR: return code from pthread_create is %d
    \n", rc);  // thread creation failure
        exit(1);        // Exit code 1
    }

    printf("\n Created new thread (%lu)... \n", thread_id);
    // if thread is created successfully
    pthread_exit(NULL);        // exiting from thread
}
```

OUTPUT:

Created new thread (140598609581824) …

Hello from new thread – got 11 !

EXPLANATION:

Here we create a var thread_id which is of the type pthread_t to store the id of the newly created thread. Then we create a new thread using pthread_create command. The printHello function and a pointer to variable t is passed to the thread as arguments. The thread executes the printHello function and the value of variable t is printed by the function. The thread is terminated by the pthread_exit() function which returns a void function.

All the functions related to thread creation, execution and termination are facilitated by the pthread API which has to be included by including the <pthread.h> library.

EXAMPLE 2:

CODE:

```
/**
*@file example2.c
*@author Nisarg Suthar (202003030)
*@brief Example code for multithreading
*/

// header files
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];          // arr to store thread-ids
int counter;

// example function
void* trythis(void* arg)
{
      unsigned long i = 0;
      counter += 1;
      printf("\n Job %d has started in thread %lu \n", counter,
pthread_self());
      for (i = 0; i < (0xFFFFFFFF); i++);
      printf("\n Job %d has finished in thread %lu \n", counter,
pthread_self());
      return NULL;
}

// start of main
int main(void)
{
      int i = 0;

      int error;

      while (i < 2)
      {
            error = pthread_create(&(tid[i]), NULL, &trythis,NULL);
      // creating threads
```

```
           if (error != 0)                  // a non zero value
means failure in thread creation
                printf("\nThread can't be created :
[%s]",strerror(error));
           i++;
     }

     pthread_join(tid[0], NULL);      // wait till thread tid[0]
terminates
     pthread_join(tid[1], NULL);      // wait till thread tid[1]
terminates

     return 0;
}
```

OUTPUT:

Job 2 has started. (140129920218880)

Job 1 has started. (140129928611584)

Job 2 has finished. (140129928611584)

Job 2 has finished. (140129920218880)

EXPLANATION:

This is a classic example of race condition.

Here we are creating two threads. Both the threads execute the tryThis function.

The try this function increments and prints the value of counter variable. Thread 1 increments it from 0 to 1. When thread two runs it increments the value from 1 to 2. When both threads terminate the value of the counter is 2.

Hence what we see is that the threads start the jobs 1 and 2. But at the time of termination both the threads have the value of the counter variable as 2. Hence the program prints job 2 is finished twice though actually one of them was job 1 as seen from the thread ids.

This is corrected by keeping a mutex lock on the critical section of the process.

EXAMPLE 3:

CODE:

```
/**
*@file example3.c
*@author Nisarg Suthar (202003030)
*@brief Example code for mutex
*/

// header files
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];               // var to store thread ids
int counter;                    // counter var
pthread_mutex_t lock;      // var to store mutext object

// try this function
void* trythis(void* arg)
{
     pthread_mutex_lock(&lock);       // critical resources locked

     unsigned long i = 0;
     counter += 1;

     printf("\n Job %d has started\n", counter);

     for (i = 0; i < (0xFFFFFFFF); i++);

     printf("\n Job %d has finished\n", counter);

     pthread_mutex_unlock(&lock);     // critical resources unlocked

     return NULL;
}

// start of main
int main(void)
{
     int i = 0;
     int error;
```

```
      if (pthread_mutex_init(&lock, NULL) != 0)        //
initialising mutex object
      {
            printf("\n mutex init has failed\n");      // prompt if
mutex fails
            return 1;
      }

      while (i < 2)
      {
            error = pthread_create(&(tid[i]),NULL,&trythis, NULL);
      // creating threads

            if (error != 0)
                  printf("\nThread can't be created
:[%s]",strerror(error));   // prompt if thread creation fails

            i++;
      }

      pthread_join(tid[0], NULL);      // waiting for thread with
tid[0] to terminate
      pthread_join(tid[1], NULL);      // waiting for thread with
tid[1] to terminate

      pthread_mutex_destroy(&lock);    // deleting the mutex object

      return 0;
}

// end of program
```

OUTPUT:

Job 1 has started

Job 1 has finished

Job 2 has started

Job 2 has finished

EXPLANATION:

Here we have used a mutex lock to protect the critical section of the program from being modified by two concurrent processes in our case the counter variable.

The mutex prevents the second process from updating the value of the counter variable and hence when the first process finishes the value of the counter remains 1.

When the first process unlocks the critical section, the second thread increases counter by 1 and hence when the second process finishes the value of the counter is 2.

In the output what we get is that Job 2 is finished is not repeated twice. The processes are "synchronised" and we get the correct output.

The mutex object needs to be destroyed at the end of the program.

Exercise 1: Given an even sized array, compute summation of adjacent elements using threads, and store it into an (n/2) sized array. (Example below: 2+3=5, 4+5=9, 6+7=13)

CODE:

```
/**
*@file exercise1.c
*@author Nisarg Suthar (202003030)
*@brief Adding adacent elements of an array using threads
*/

// Assumptions:
// Here an even sized array is given.
// The size of the array is assumed to be 6.

# define ARRAY_SIZE 6

// Header files
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h>

int counter = 0;                          // counter variable
int array[ARRAY_SIZE];        // input array
int sum_arr[ARRAY_SIZE/2];    // output array
pthread_t tid[3];                         // thread-ids

void* adj_add(void *data)                 // function to sum adjacent
elements
{
      int i = *((int*)data);
      //printf("\n %d \t %d \t counter = %d \t i = %d", array[2*i],
array[2*i+1], counter, i);
      sum_arr[i] = array[2*i] + array[2*i + 1];
      pthread_exit(NULL);           // thread termination
}

// start of main
int main(int argc, char *argv[])
{
      int error;
      int count[3] = {0};           // to pass as an argument so that the
function knows where to store the sum

      printf("\n Enter the array elements : \n");          // taking user
input
```

```
	for(int i = 0; i < 6; ++i)
	{
		printf(" Element %d : ", i+1);
		scanf("%d", &array[i]);
		//printf("%d", array[i]);
	}
	printf("\n");

	for(counter; counter < (ARRAY_SIZE/2); ++counter)		//
creating 3 threads
	{
		count[counter] = counter;
		error = pthread_create(&tid[counter], NULL, &adj_add,
count+counter);
		if(error != 0)							//
exception handling if thread creation fails
		{
			printf("\n Thread Creation Failed : %s",
strerror(error));
		}
	}

	pthread_join(tid[0], NULL);		// joining the threads one after
the other
	pthread_join(tid[1], NULL);
	pthread_join(tid[2], NULL);

	printf("\n The sum of adjacent entries are : \n");		//
displaying the output

	while(counter--)
	{
		printf(" Sum of %d and %d entries : %d\n",(3-counter-1)*2, (3-
counter-1)*2+1, sum_arr[3-counter-1]);
	}

	return 0;
}
// end of program
```

OUTPUT:

Enter the array elements :
Element 1 : 2
Element 2 : 3
Element 3 : 4
Element 4 : 5
Element 5 : 6
Element 6 : 7

The sum of adjacent entries are :
Sum of 0 and 1 entries : 5
Sum of 2 and 3 entries : 9
Sum of 4 and 5 entries : 13

EXPLANATION:

In this program the input array is of length 6. We need to find the sum of the adjacent entries and store them into an array of length 3.

For that we create 3 threads each summing one of the 3 pairs of numbers. As an argument to the thread, we pass the pointer to the function add_adj and the position at which the thread should store the sum in the form of a void pointer.

The threads sum the adjacent numbers and finally we print the resultant array as the output.

Exercise 2: In your bank account your current balance is 500 Rs. You have 2 functions:

1. **Credit -> (reads amount, credits 50 Rs., prints final amount)**
2. **Debit -> (reads amount, debits 50 Rs., prints final amount)**

Implement a C program to safely complete both the transactions if both transactions are taking place concurrently.

CODE:

```
/**
*@file exercise2.c
*@author Nisarg Suthar (202003030)
*@brief Concurrent transactions
*/

// Header files
#include<stdio.h>
#include<unistd.h>
#include<pthread.h>
#include<stdlib.h>
#include<string.h>

int counter = 0;                        // counter variable
float bank_bal = 500;                   // bank balance
pthread_t tid[2];                            // thread-ids
pthread_mutex_t lock;                   // mutex lock object

void* credit_amount(void *data)             // function to credit to
the account
{
     pthread_mutex_lock(&lock);             // locking the critical
section of code

     float i = *((float*)data);
     printf("\n Current Acc Balance : Rs %6.2f.", bank_bal);
               // reading the current balance
     bank_bal += i;
                          // adding the credit amount
     printf("\n Acc Balance after credit : Rs %6.2f.\n", bank_bal);
          // printing the balance after creditting

     pthread_mutex_unlock(&lock);          // unlocking the
critical section
```

```
      pthread_exit(NULL);                          // terminating the
thread
}

void* debit_amount(void *data)              // function debit amount
from the account
{
      pthread_mutex_lock(&lock);            // locking the critcal
section of program

      float i = *((float*)data);
      printf("\n Current Acc Balance : Rs %6.2f.", bank_bal);
                  // reading current balance
      bank_bal -= i;
                              // subtracting the debit amount
      printf("\n Acc Balance after debit : Rs %6.2f.\n", bank_bal);
            // printing the balance after debitting

      pthread_mutex_unlock(&lock);          // unlocking the
critical section
      pthread_exit(NULL);                          // terminating the
thread
}

// start of main
int main(int argc, char *argv[])
{
      float credit = 0;
      float debit = 0;

      void *c = &credit;
      void *d = &debit;

      printf("\n Enter the amount to be credited to the acount : ");
            // credit and debit amounts as input
      scanf("%f", &credit);
      printf(" Enter the amount to be debited from the acount : ");
      scanf("%f", &debit);

      if(pthread_mutex_init(&lock, NULL) != 0)         // creating
the mutex lock object
      {
            printf("\n Mutex Initialization Failed !");
            return 1;
      }
```

```
	if(pthread_create(&tid[0], NULL, &credit_amount, c) != 0)
	// creating a thread for creditting
	{
		printf("\n Thread Creation Failed !");
	}

	if(pthread_create(&tid[1], NULL, &debit_amount, d) != 0)
	// creating a thread for debitting
	{
		printf("\n thread Creation Failed !");
	}

	pthread_join(tid[0], NULL);		// joining the two
threads
	pthread_join(tid[1], NULL);

	pthread_mutex_destroy(&lock);		// deleting the mutex
object

	return 0;
}
// end of program
```

OUTPUT:

Enter the amount to be credited to the acount : 250
Enter the amount to be debited from the acount : 100

Current Acc Balance : Rs 500.00.
Acc Balance after debit : Rs 400.00.

Current Acc Balance : Rs 400.00.
Acc Balance after credit : Rs 650.00.

EXPLANATION:

Here we are handling a situation where we are updating the account balance for two concurrent transactions.

We create a thread for each transaction. Then to keep things "synchronised" we need to use the mutex lock to prevent one thread from accessing the account balance while another thread is executing.

The functions credit_amount and debit_amount are made to credit the amount to and debit the amount from the account. These are passed as arguments to the thread.

❖ **Simulate given problem for Shortest remaining time first and RR. Compare the average waiting time for Shortest remaining time first and RR and explain the reason.**

Process:	A	B	C	D	E	F	G
Arrival time:	2	4	5	7	9	15	16
Service time:	3	2	1	4	2	6	8
Priority:	0	0	0	0	0	0	0

FOR SHORTEST REMAINING TIME FIRST SCHEDULING:

Process	Arrival	Service	Priority	Started	Completion	turnaround time (TAT)	waiting time (WAT)
A	2	3	0	2	5	3	0
C	5	1	0	5	6	1	0
B	4	2	0	6	8	4	2
E	9	2	0	9	11	2	0
D	7	4	0	8	14	7	3
F	15	6	0	15	21	6	0
G	16	8	0	21	29	13	5
Avg:	-	**3.71429**	-	-	-	**5.14286**	**1.42857**

FOR ROUND ROBIN SCHEDULING:

Process	Arrival	Service	Priority	Started	Completion	turnaround time (TAT)	waiting time (WAT)
A	2	3	0	2	6	4	1
C	5	1	0	6	7	2	1
B	4	2	0	4	8	4	2
E	9	2	0	9	12	3	1
D	7	4	0	8	14	7	3
F	15	6	0	15	26	11	5
G	16	8	0	16	29	13	5
Avg:	-	**3.71429**	-	-	-	**6.28571**	**2.57143**

Here, as one can observe, the average turnaround time as well as the average waiting time is less for the SRTF scheduling compared to RR scheduling.

This is because in Round Robin scheduling, each process in the ready queue is given a slice of CPU time also called a time quantum. Now when the ready queue contains processes which have larger CPU burst-times as compared to the time quantum, the process is interrupted and another process is given the CPU. This results in increase in average turnaround time for processes which have are CPU-bound. Also, the average waiting time also increase in the turnaround time is a direct consequence of increase in waiting time since the service time remains same. Also, repetitive context switching also causes increase in the average turnaround and waiting times.

On the other hand, in Shortest Remaining Time First scheduling, the process with the process with the shortest next CPU burst is scheduled first. This results in early completion of the processes which have a smaller CPU burst resulting in lesser waiting times for the remaining processes and hence a lesser turnaround time overall.