# Department of Computing

**CS 330: Operating Systems**

**BESE: 9AB**

# Lab 8: ****Programming Threads****

**CLO4(Develop programs to interact with OS components through its API)**

# Date: 9-10-2017

**Time: 10:00AM – 01:00 PM**

**&**

**02:00 PM – 05:00 PM**

**Instructor: Mr. Taufeeq Ur Rehman**

**Ahmed Hassan Ismail – BESE-9B – 237897**

**Lab 8: Programming Threads**

**Introduction**

The purpose of this lab is to learn threads programmatically in more depth and learn to ensure safe global memory access.

**Objectives**

Objective of this lab is to enable students use threads in programs and synchronization primitives.

**Tools/Software Requirement**

Linux OS installed on laptops or systems.

**Description**

In order to understand the concept of threads and role of synchronization primitives to ensure safe global variable access you are supposed to Write a program using pthreads, which calculates the sum of elements in a hard-coded integer array in parallel using 4 threads. The program must divide the work between 4 threads which run simultaneously. For simplicity, you can assume that the size of the array is 100. Note that the integer array must be declared as a global data structure.

**Tasks**

1) Initially code your solution so that the sum of elements is maintained in a global shared variable. Each thread modifies the same shared variable as it sums up elements from the array. Use a suitable synchronization primitive to ensure safe access to the global variable.

2) Now modify your solution so that it does not require any synchronization primitives. An idea here is that each thread must first find its own local sum first, and later the main () function can use these local sums to determine the global sum for the entire array.

3) Which approach do you think would run faster, and why?

**Deliverables**

Submit the document containing code and screenshot for both of your programs, along with the answer to the last question. You are supposed to strictly follow submission guidelines.

Hard code array 1 to 100

Task 1:

Global variable Sum

1:25 thread 1

26:50 thread 2

51:75 thread 3

76:100 thread 4

Sum: 1+26+51+75 + 2+ 27+ 52+77….

Synchronization primitives

Task 2:

1:25 thread 1- -> compute local sum1

26:50 thread 2 -> compute local sum2

51:75 thread 3 -> compute local sum3

76:100 thread 4 -> compute local sum4

Total sum= sum1+ sum2+ sum3+ sum4

Without Synchronization primitives

Task 3:

Time analysis

**Ahmed Hassan Ismail – BESE-9B – 237897**

**My Tasks**

Task 1

# Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

void \*print\_message\_function(void \*ptr);

int calculate\_sum(int ary[], int start\_index, int end\_index);

int arry[100];

int sum = 0;

int start\_index = 0;

int end\_index = 0;

int main()

{

pthread\_t thread1, thread2, thread3, thread4;

const char \*message1 = "Thread 1";

const char \*message2 = "Thread 2";

const char \*message3 = "Thread 3";

const char \*message4 = "Thread 4";

int iret1, iret2, iret3, iret4;

for (int i = 0; i<100; i++)

{

arry[i] = i;

printf("%d ", arry[i]);

}

printf("\n\n");

/\* Here We go - Create independent threads each of which will execute function \*/

iret1 = pthread\_create(&thread1, NULL, print\_message\_function, (void\*)message1);

if (iret1)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret1);

exit(EXIT\_FAILURE);

}

iret2 = pthread\_create(&thread2, NULL, print\_message\_function, (void\*)message2);

if (iret2)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret2);

exit(EXIT\_FAILURE);

}

iret3 = pthread\_create(&thread3, NULL, print\_message\_function, (void\*)message3);

if (iret3)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret3);

exit(EXIT\_FAILURE);

}

iret4 = pthread\_create(&thread4, NULL, print\_message\_function, (void\*)message4);

if (iret4)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret4);

exit(EXIT\_FAILURE);

}

pthread\_join(thread1, NULL);

pthread\_join(thread2, NULL);

pthread\_join(thread3, NULL);

pthread\_join(thread4, NULL);

exit(EXIT\_SUCCESS);

system("pause");

return 0;

}

void \*print\_message\_function(void \*ptr)

{

char \*message;

message = (char \*)ptr;

printf("%s \n", message);

if (message == "Thread 1"){

start\_index = 0;

end\_index = 25;

}

else if (message == "Thread 2"){

start\_index = 25;

end\_index = 50;

}

else if (message == "Thread 3"){

start\_index = 50;

end\_index = 75;

}

else if (message == "Thread 4"){

start\_index = 75;

end\_index = 100;

}

pthread\_mutex\_lock(&mutex1);

sum = calculate\_sum(arry, start\_index, end\_index);

pthread\_mutex\_unlock(&mutex1);

printf("Start Index: %d\n",start\_index);

printf("End Index: %d\n", end\_index);

printf("Sum = : %d\n", sum);

return (int\*)sum;

}

int calculate\_sum(int ary[], int start\_index, int end\_index)

{

for (int j = start\_index; j<end\_index; j++){

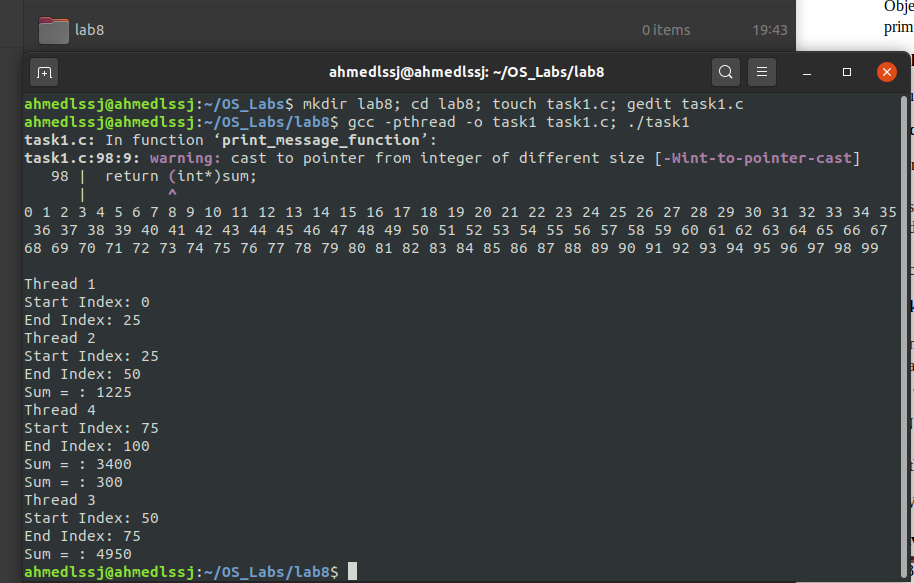
sum = sum + ary[j];

}

return sum;

}

# Screenshot:



Task 2

# Code:

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

pthread\_mutex\_t mutex1 = PTHREAD\_MUTEX\_INITIALIZER;

void \*Add\_function(void \*ptr);

int calculate\_sum(int ary[], int start\_index, int end\_index);

int arry[100];

int sum = 0;

int start\_index = 0;

int end\_index = 0;

int main()

{

pthread\_t thread1, thread2, thread3, thread4;

void \*t1\_sum, \*t2\_sum, \*t3\_sum, \*t4\_sum;

const char \*thread\_name1 = "Thread 1";

const char \*thread\_name2 = "Thread 2";

const char \*thread\_name3 = "Thread 3";

const char \*thread\_name4 = "Thread 4";

int iret1, iret2, iret3, iret4;

for (int i = 0; i<100; i++)

{

arry[i] = i;

printf("%d ", arry[i]);

}

printf("\n\n");

/\* Here We go - Create independent threads each of which will execute function \*/

iret1 = pthread\_create(&thread1, NULL, Add\_function, (void\*)thread\_name1);

if (iret1)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret1);

exit(EXIT\_FAILURE);

}

iret2 = pthread\_create(&thread2, NULL, Add\_function, (void\*)thread\_name2);

if (iret2)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret2);

exit(EXIT\_FAILURE);

}

iret3 = pthread\_create(&thread3, NULL, Add\_function, (void\*)thread\_name3);

if (iret3)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret3);

exit(EXIT\_FAILURE);

}

iret4 = pthread\_create(&thread4, NULL, Add\_function, (void\*)thread\_name4);

if (iret4)

{

fprintf(stderr, "Error - pthread\_create() return code: %d\n", iret4);

exit(EXIT\_FAILURE);

}

pthread\_join(thread1, &t1\_sum);

pthread\_join(thread2, &t2\_sum);

pthread\_join(thread3, &t3\_sum);

pthread\_join(thread4, &t4\_sum);

sum = 0;

sum += (int)t1\_sum;

sum += (int)t2\_sum;

sum += (int)t3\_sum;

sum += (int)t4\_sum;

printf("\n\nSum = %d\n", sum);

exit(EXIT\_SUCCESS);

system("pause");

return 0;

}

void \*Add\_function(void \*ptr)

{

char \*thread\_name;

thread\_name = (char \*)ptr;

printf("%s \n", thread\_name);

if (thread\_name == "Thread 1"){

start\_index = 0;

end\_index = 25;

sum = 0;

sum = calculate\_sum(arry, start\_index, end\_index);

}

else if (thread\_name == "Thread 2"){

start\_index = 25;

end\_index = 50;

sum = 0;

sum = calculate\_sum(arry, start\_index, end\_index);

}

else if (thread\_name == "Thread 3"){

start\_index = 50;

end\_index = 75;

sum = 0;

sum = calculate\_sum(arry, start\_index, end\_index);

}

else if (thread\_name == "Thread 4"){

start\_index = 75;

end\_index = 100;

sum = 0;

sum = calculate\_sum(arry, start\_index, end\_index);

}

printf("Sum = : %d\n", sum);

return (void\*)sum;

}

int calculate\_sum(int ary[], int start\_index, int end\_index)

{

for (int j = start\_index; j<end\_index; j++){

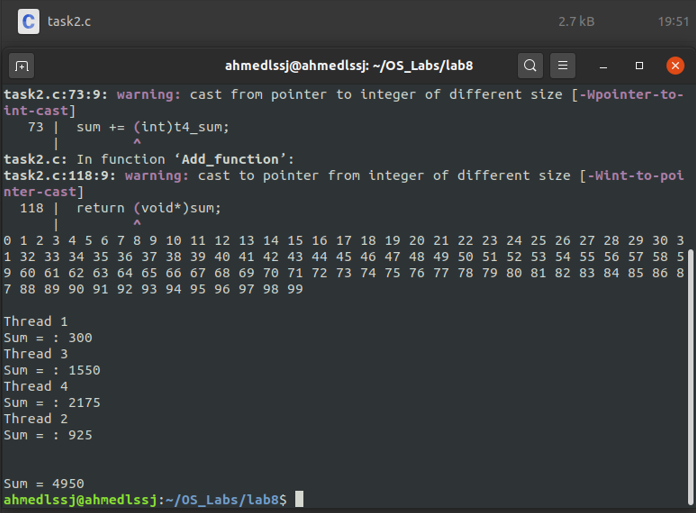
sum = sum + ary[j];

}

return sum;

}

# Screenshot:



Task 3

# Answer

The second approach is faster than the first approach. In the first approach acquiring the lock and synchronizing all the thread takes time whereas, without synchronization, second approach works faster and better.