Parallel Programming

Homework 3

Student Name : ABID ALI
Student ID : 2019380141
Student Email : abiduu354@gmail.com / **3616990795@qq.com**
Lecture : Professor Tianhai Zhao
Submission : 10/27/2022
Submitted Email : zhaoth@nwpu.edu.cn

Contents

1. Please read the Chapter 4 of An Introduction to Parallel Programming.

Answer1**:**

The chapter four was about shared-memory, threads and pthreads. Reading this chapter taught me about that in Pthreads programs. How all the threads have access to global variables, while local variables usually are private to the thread running the function. When multiple threads are executing, the order in which the statements are executed by the different threads is usually nondeterministic. When nondeterminism results from multiple threads attempting to access a shared resource, such as a shared variable or a shared file, at least one of the accesses is an update, and the accesses can result in an error, we have a **race condition**. One of our most important tasks in writing shared-memory programs is identifying and correcting race conditions. A **critical section** is a block of code that updates a shared resource that can only be updated by one thread at a time, so the execution of code in a critical section should, effectively, be executed as serial code. How to perform error checking and various other approaches that can be taken towards threads. I have learned about matrix-vector multiplication, Busy-waiting, mutexes, producer-consumer synchronization and semaphores. I have acquired valuable knowledges on Barriers and Condition variables, read-write locks, caches, cache coherence and false sharing. Alongside, I have learned about thread-safety as well. After going through and learning the contents of this chapter I have applied those in the tasks that were assigned to me.

2. We assumed the matrix-vector multiplication program(*mat_vect_pthread.c*) where *m*, the number of rows were evenly divisible by *t*, the number of threads. Please read, compile and run *mat_vect_pthread*, and then modify the code with following requirements:

1. How do the formulas for the assignments change if *m* cannot evenly divisible by *t*? Give your strategy and modify the code.

Answer2(1)**:**

Matrix-Vector Multiplication

To define multiplication between a matrix **A** and a vector **x** (i.e., the matrix-vector product), we need to view the vector as a column matrix. We define the matrix-vector product only for the case when the number of columns in **A** equals the number of rows in xx. So, if **A** is an m×n matrix (i.e., with n columns), then the product **Ax** is defined for n×1 column vectors **x**. If we let **Ax=b**, then **b** is an m×1 column vector. In other words, the number of rows in **A** (which can be anything) determines the number of rows in the product **b**.

The general formula for a matrix-vector product is

$$A\mathbf{x} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n \end{bmatrix}.$$

Although it may look confusing at first, the process of matrix-vector multiplication is actually quite simple. One takes the dot product of **x** with each of the rows of **A**. (This is why the number of columns in AA has to equal the number of components in xx.) The first component of the matrix-vector product is the dot product of **x** with the first row of **A**, etc. In fact, if **A** has only one row, the matrix-vector product is really a dot product in disguise.

For example, if

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix}$$

and $\mathbf{x} = (2, 1, 0)$, then

$$\begin{aligned} A\mathbf{x} &= \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 1 - 1 \cdot 1 + 0 \cdot 2 \\ 2 \cdot 0 - 1 \cdot 3 + 0 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -3 \end{bmatrix}. \end{aligned}$$

For the given program the code will remain unchanged if the column is not evenly divisible by the number of threads(t). However, if the number of rows isn't evenly divisible by the number of threads then we need make a decision who will be responsible for the rows left over after integer division of "**m**" the number of rows by "**t**" the number of threads.

For example, if m(row)= 10 and t(thread) = 4, then there are two extra rows.

Since 10 % 4 = 2(Remainder).

The 2 remainder is the extra rows.

Clearly, there are a number of possibilities here. One is to give the first two threads each an extra row. So, I'd assign three rows to threads 0 and 1 and two rows to threads 2 and 3.

The algorithm will be as followed:

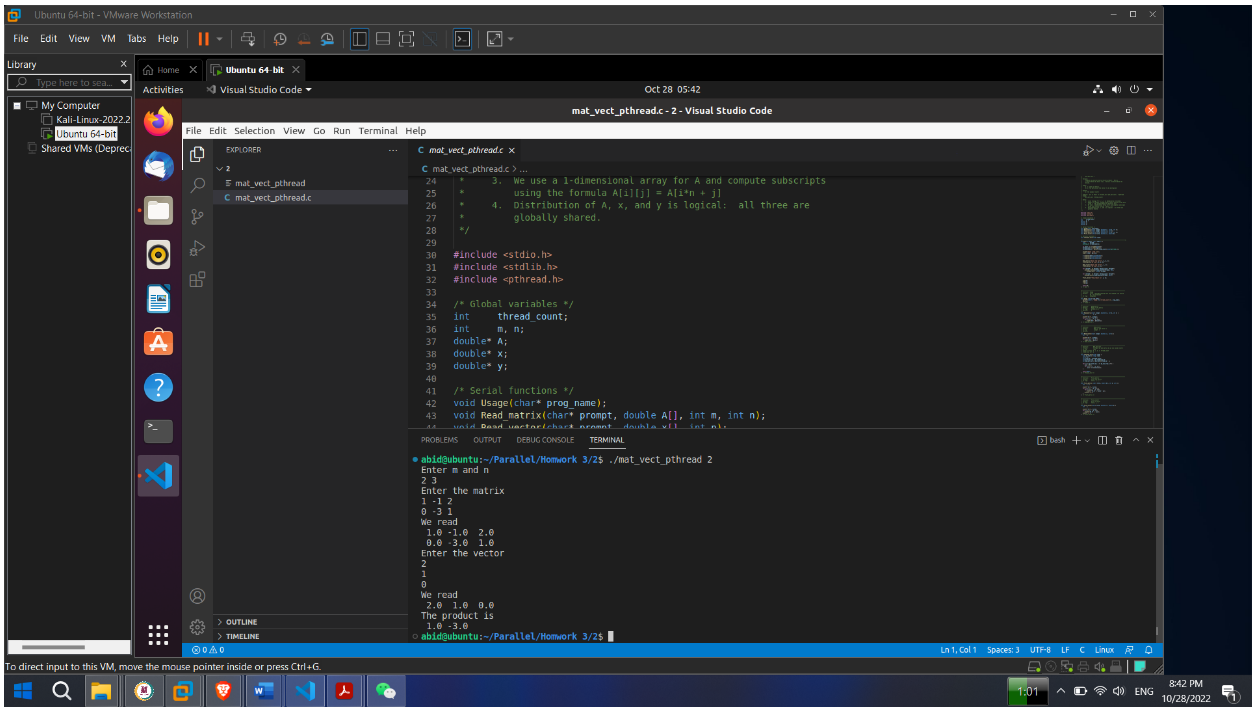
```
int quotient = m/t; /* Every thread gets at least m/t rows */
int remainder = m % t;
if (my_rank < remainder) { /* I get m/t + 1 rows */
local_m = quotient + 1;
my_first_row = my_rank*local_m;
my_last_row = my_first_row + local_m - 1;
} else { /* We got m/t rows */
local_m = quotient;
/* Each of the threads 0, 1, . . . , remainder - 1 gets */
/* An extra row. So add in these rows to get my_first_row */
my_first_row = my_rank*local_m + remainder;
my_last_row = my_first_row + local_m - 1;
}
```

2. Make tests and timing the execution time when you increase the *n* with different *t*, and analyze the results.

Answer2(2)**:**

For this program, ‘n’ will be representing the columns. Changing columns will not leave any impact on the code, it will rather remain unchanged. As long the rows are evenly divisible by (t) the program will keep performing. This are the tests that I have conducted:

i) t = 2
   n = 3



```
mat_vect_pthread.c - 2 - Visual Studio Code

 *     3.  We use a 1-dimensional array for A and compute subscripts
 *         using the formula A[i][j] = A[i*n + j]
 *     4.  Distribution of A, x, and y is logical:  all three are
 *         globally shared.
 */

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

/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;

/* Serial functions */
void Usage(char* prog_name);
void Read_matrix(char* prompt, double A[], int m, int n);
```

```
abid@ubuntu:~/Parallel/Homwork 3/2$ ./mat_vect_pthread 2
Enter m and n
2 3
Enter the matrix
1 -1 2
0 -3 1
We read
 1.0 -1.0  2.0
 0.0 -3.0  1.0
Enter the vector
2
1
0
We read
 2.0  1.0  0.0
The product is
 1.0 -3.0
abid@ubuntu:~/Parallel/Homwork 3/2$
```

The matrix that was used:

1 -1 2

0 -3 1

The vector that was used:

2

1

0

For example, if

$$A = \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix}$$

and $\mathbf{x} = (2, 1, 0)$, then

$$\begin{aligned} A\mathbf{x} &= \begin{bmatrix} 1 & -1 & 2 \\ 0 & -3 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 1 \\ 0 \end{bmatrix} \\ &= \begin{bmatrix} 2 \cdot 1 - 1 \cdot 1 + 0 \cdot 2 \\ 2 \cdot 0 - 1 \cdot 3 + 0 \cdot 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 \\ -3 \end{bmatrix}. \end{aligned}$$

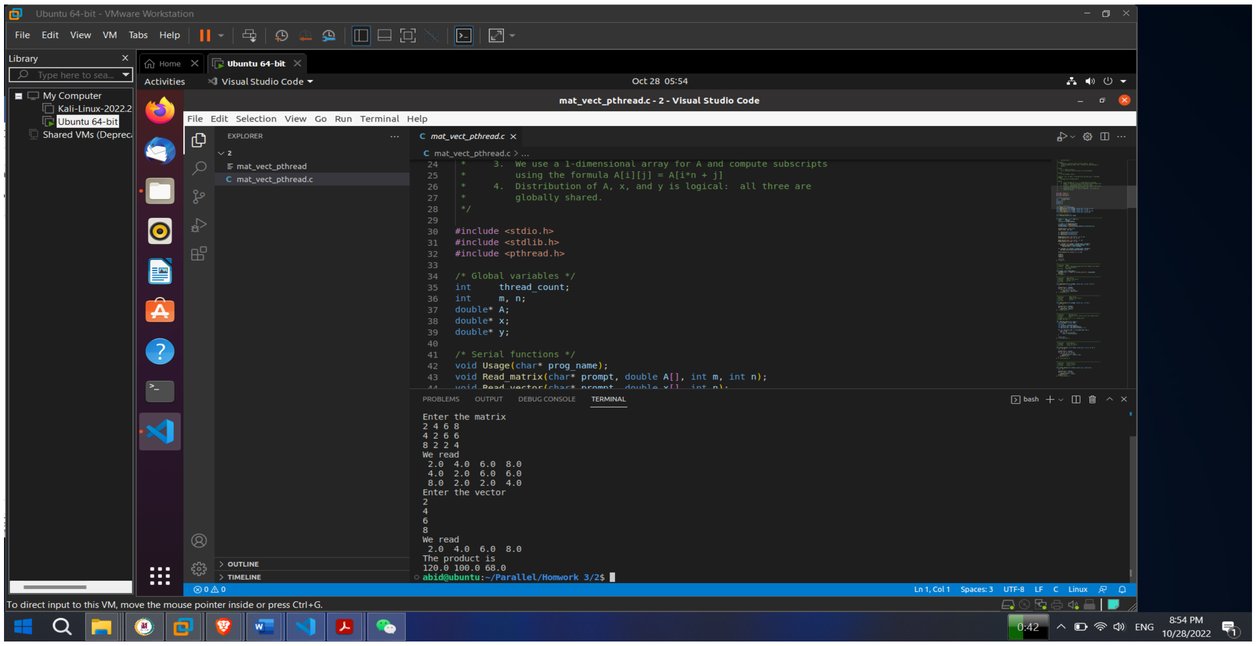
Makefile

Compile: gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread

./mat_vect_pthread 2

ii) t = 3

n = 4



```
 *     3.  We use a 1-dimensional array for A and compute subscripts
 *         using the formula A[i][j] = A[i*n + j]
 *     4.  Distribution of A, x, and y is logical:  all three are
 *         globally shared.
 */

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

/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;

/* Serial functions */
void Usage(char* prog_name);
void Read_matrix(char* prompt, double A[], int m, int n);
```

```
Enter the matrix
2 4 6 8
4 2 6 6
8 2 2 4
We read
 2.0  4.0  6.0  8.0
 4.0  2.0  6.0  6.0
 8.0  2.0  2.0  4.0
Enter the vector
2
4
6
8
We read
 2.0  4.0  6.0  8.0
The product is
120.0 100.0 68.0
abid@ubuntu:~/Parallel/Homwork 3/2$ 
```

The matrix that was used:

```
2 4 6 8
4 2 6 6
8 2 2 4
```

The vector that was used:

```
2
4
6
8
```

Makefile

Compile: gcc -g -Wall -o pth_mat_vect pth_mat_vect.c -lpthread

```
./mat_vect_pthread 3
```

Analyze:

Here we can see that, changing `n` with different `t` doesn't change the way the code works and I am getting the result I should be getting.

3. Modify the mutex version of the π calculation program(*pth_pi_mutex.c*) so that it uses a semaphore instead of a mutex. How does the performance of this version compare with the mutex version?

Answer:

In an Operating System, there are a number of processes that are ready to be executed at a particular instant of time. These processes require various resources for their execution. So, for this, we have shared resources in our system that can be shared between these processes. But one thing that should be kept in mind is that the resources are shared but it should not be used simultaneously by all the processes. For example, if the system is having a printer, then this printer is shared with all the processes but at a time, only one process can use the printer. No two processes should be allowed to use the printer at the same instant of time. This is called **Process Synchronization**.

Process synchronization, two methods are used. They are:

1. Mutex
2. Semaphore

Mutex:

Mutex or Mutual Exclusion Object is used to give access to a resource to only one process at a time. The mutex object allows all the processes to use the same resource but at a time, only one process is allowed to use the resource. Mutex uses the lock-based technique to handle the critical section problem.

Whenever a process requests for a resource from the system, then the system will create a mutex object with a unique name or ID. So, whenever the process wants to use that resource, then the process occupies a lock on the object. After locking, the process uses the resource and finally releases the mutex object. After that, other processes can create the mutex object in the same manner and use it.

By locking the object, that particular resource is allocated to that particular process and no other process can take that resource. So, in the critical section, no other processes are allowed to use the shared resource. In this way, the process synchronization can be achieved with the help of a mutex object.

Semaphore:

Semaphore is an integer variable S, that is initialized with the number of resources present in the system and is used for process synchronization. It uses two functions to change the value of S i.e. wait() and signal(). Both these functions are used to modify the value of semaphore but the functions allow only one process to change the value at a particular time i.e. no two processes can change the value of semaphore simultaneously. There are two categories of semaphores i.e. Counting semaphores and Binary semaphores.

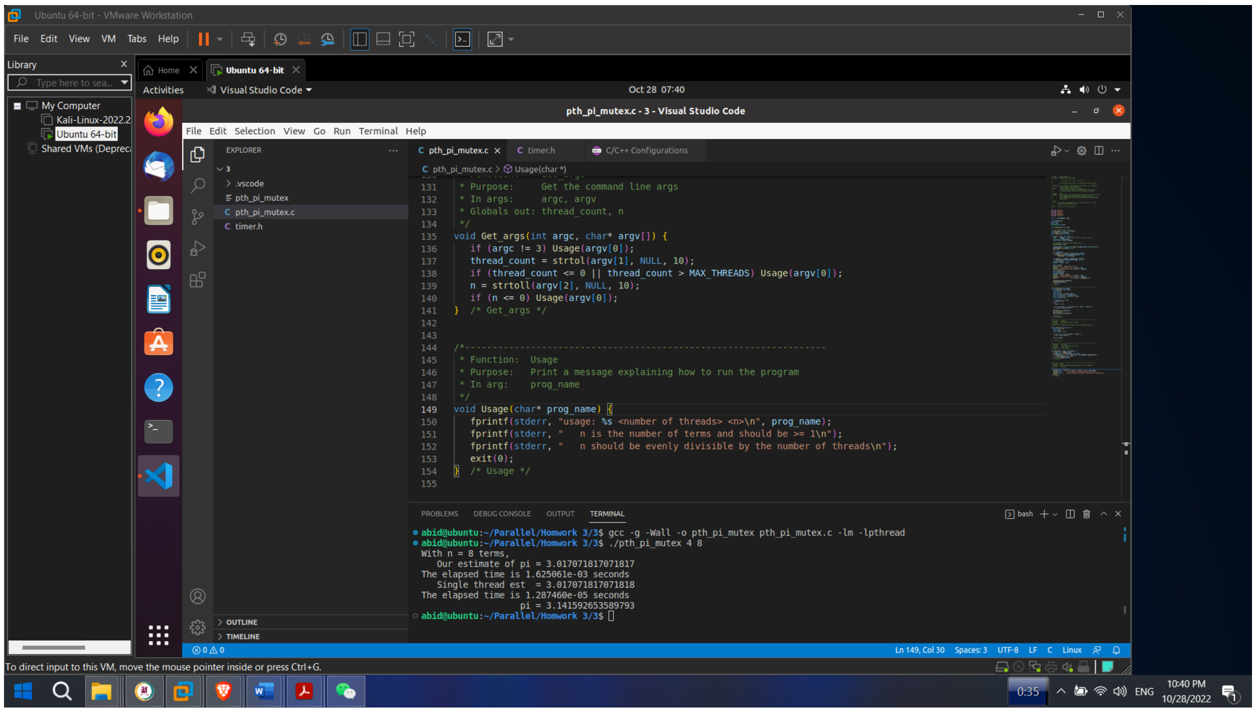
In Counting semaphores, firstly, the semaphore variable is initialized with the number of resources available. After that, whenever a process needs some resource, then the wait() function is called and the value of the semaphore variable is decreased by one. The process then uses the resource and after using the resource, the signal() function is called and the value of the semaphore variable is increased by one. So, when the value of the semaphore variable goes to 0 i.e all the resources are taken by the process and there is no resource left to be used, then if some other process wants to use resources then that process has to wait for its turn. In this way, we achieve the process synchronization.

In Binary semaphores, the value of the semaphore variable will be 0 or 1. Initially, the value of semaphore variable is set to 1 and if some process wants to use some resource then the wait() function is called and the value of the semaphore is changed to 0 from 1. The process then uses the resource and when it releases the resource then the signal() function is called and the value of the semaphore variable is increased to 1. If at a particular instant of time, the value of the semaphore variable is 0 and some other process wants to use the same resource then it has to wait for the release of the resource by the previous process. In this way, process synchronization can be achieved. It is similar to mutex but here locking is not performed.

Difference between Mutex and Semaphore

Till now, we have learned about mutex and semaphore. Most of you might have guessed the difference between these two. Let's have a look into the difference between mutex and semaphore:

- Mutex uses a locking mechanism i.e. if a process wants to use a resource then it locks the resource, uses it and then release it. But on the other hand, semaphore uses a signaling mechanism where wait() and signal() methods are used to show if a process is releasing a resource or taking a resource.
- A mutex is an object but semaphore is an integer variable.
- In semaphore, we have wait() and signal() functions. But in mutex, there is no such function.
- A mutex object allows multiple process threads to access a single shared resource but only one at a time. On the other hand, semaphore allows multiple process threads to access the finite instance of the resource until available.
- In mutex, the lock can be acquired and released by the same process at a time. But the value of the semaphore variable can be modified by any process that needs some resource but only one process can change the value at a time.



```
 * Purpose:     Get the command line args
 * In args:     argc, argv
 * Globals out: thread_count, n
 */
void Get_args(int argc, char* argv[]) {
   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
   n = strtoll(argv[2], NULL, 10);
   if (n <= 0) Usage(argv[0]);
}  /* Get_args */


/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   Print a message explaining how to run the program
 * In arg:    prog_name
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads> <n>\n", prog_name);
   fprintf(stderr, "   n is the number of terms and should be >= 1\n");
   fprintf(stderr, "   n should be evenly divisible by the number of threads\n");
   exit(0);
}  /* Usage */
```

```
abid@ubuntu:~/Parallel/Homwork 3/3$ gcc -g -Wall -o pth_pi_mutex pth_pi_mutex.c -lm -lpthread
abid@ubuntu:~/Parallel/Homwork 3/3$ ./pth_pi_mutex 4 8
With n = 8 terms,
   Our estimate of pi = 3.017071817071817
The elapsed time is 1.625061e-03 seconds
   Single thread est  = 3.017071817071818
The elapsed time is 1.287460e-05 seconds
                   pi = 3.141592653589793
abid@ubuntu:~/Parallel/Homwork 3/3$ 
```

Our estimate of pi = 3.017071817071817

The elapsed time is 1.625061e-03 seconds / 0.001625061 seconds

= 0.00162 seconds

Single thread est = 3.017071817071818

The elapsed time is 1.287460e-05 seconds / 0.00001287460 seconds

= 0.000128 seconds

pi = 3.141592653589793

Our estimation time we got is 0.00162 seconds and single thread estimated time we get is 0.000128 seconds when we use mutex.

timer.h header file for mutex

```
/* File:     timer.h
 *
 * Purpose:  Define a macro that returns the number of seconds that
 *           have elapsed since some point in the past.  The timer
 *           should return times with microsecond accuracy.
 *
 * Note:     The argument passed to the GET_TIME macro should be
 *           a double, *not* a pointer to a double.
 *
 * Example:
 *    #include "timer.h"
 *    . . .
 *    double start, finish, elapsed;
 *    . . .
 *    GET_TIME(start);
 *    . . .
 *    Code to be timed
 *    . . .
 *    GET_TIME(finish);
 *    elapsed = finish - start;
 *    printf("The code to be timed took %e seconds\n", elapsed);
 *
 * IPP:  Section 3.6.1 (pp. 121 and ff.) and Section 6.1.2 (pp. 273 and ff.)
 */
#ifndef _TIMER_H_
#define _TIMER_H_

#include <sys/time.h>

/* The argument now should be a double (not a pointer to a double) */
#define GET_TIME(now) { \
   struct timeval t; \
   gettimeofday(&t, NULL); \
   now = t.tv_sec + t.tv_usec/1000000.0; \
}

#endif
```

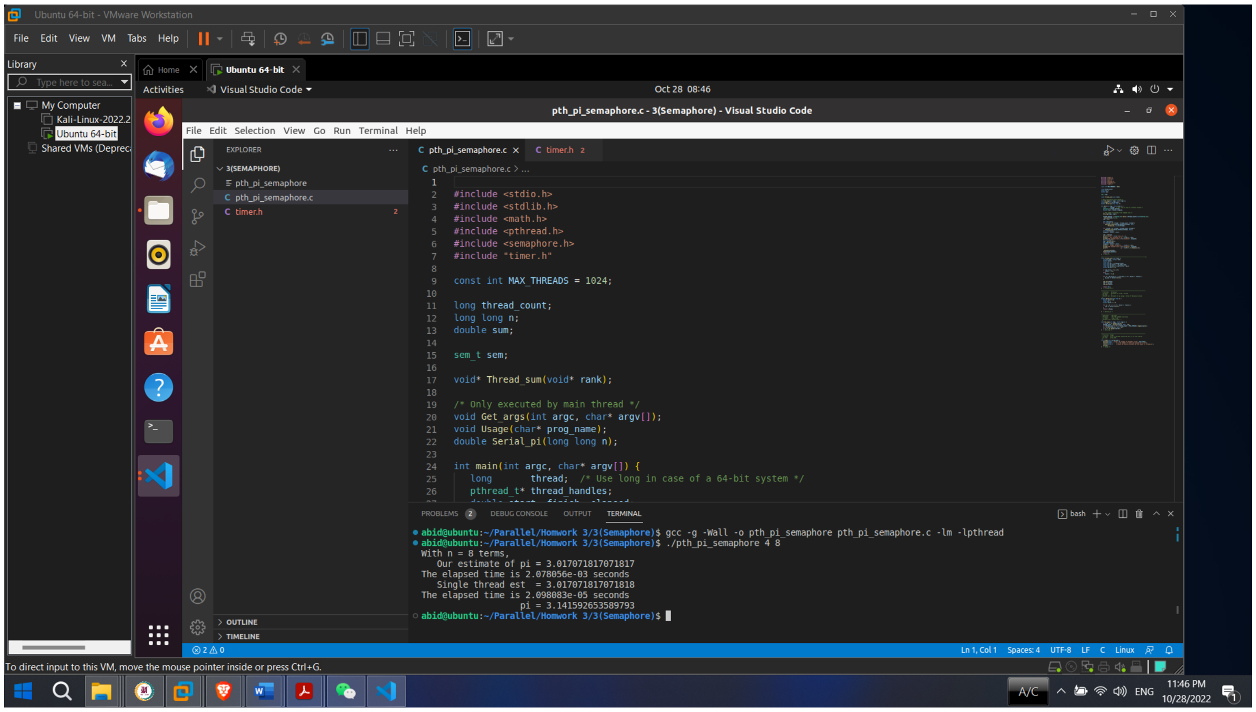
Using semaphore instead of mutex version:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include "timer.h"

const int MAX_THREADS = 1024;

long thread_count;
long long n;
double sum;

sem_t sem;

void* Thread_sum(void* rank);

/* Only executed by main thread */
void Get_args(int argc, char* argv[]);
void Usage(char* prog_name);
double Serial_pi(long long n);

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
   double start, finish, elapsed;

   /* Get number of threads from command line */
   Get_args(argc, argv);

   thread_handles = (pthread_t*) malloc (thread_count*sizeof(pthread_t));
```

```
        sem_init(&sem, 0, 1);
   sum = 0.0;

   GET_TIME(start);
   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
          Thread_sum, (void*)thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
   GET_TIME(finish);
   elapsed = finish - start;

   sum = 4.0*sum;
   printf("With n = %lld terms,\n", n);
   printf("   Our estimate of pi = %.15f\n", sum);
   printf("The elapsed time is %e seconds\n", elapsed);
   GET_TIME(start);
   sum = Serial_pi(n);
   GET_TIME(finish);
   elapsed = finish - start;
   printf("   Single thread est  = %.15f\n", sum);
   printf("The elapsed time is %e seconds\n", elapsed);
   printf("                   pi = %.15f\n", 4.0*atan(1.0));

        sem_destroy(&sem);
   free(thread_handles);
   return 0;
}  /* main */

/*------------------------------------------------------------------*/
void* Thread_sum(void* rank) {
   long my_rank = (long) rank;
   double factor;
   long long i;
   long long my_n = n/thread_count;
   long long my_first_i = my_n*my_rank;
   long long my_last_i = my_first_i + my_n;
   double my_sum = 0.0;
```

```
  if (my_first_i % 2 == 0)
    factor = 1.0;
  else
    factor = -1.0;

  for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
    my_sum += factor/(2*i+1);
  }

  sem_wait(&sem);
  sum += my_sum;
  sem_post(&sem);

  return NULL;
}  /* Thread_sum */

/*------------------------------------------------------------------
 * Function:   Serial_pi
 * Purpose:    Estimate pi using 1 thread
 * In arg:     n
 * Return val: Estimate of pi using n terms of Maclaurin series
 */
double Serial_pi(long long n) {
  double sum = 0.0;
  long long i;
  double factor = 1.0;

  for (i = 0; i < n; i++, factor = -factor) {
    sum += factor/(2*i+1);
  }
  return 4.0*sum;

}  /* Serial_pi */

/*------------------------------------------------------------------
 * Function:    Get_args
 * Purpose:     Get the command line args
 * In args:     argc, argv
 * Globals out: thread_count, n
 */
```

```
void Get_args(int argc, char* argv[]) {
   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   if (thread_count <= 0 || thread_count > MAX_THREADS) Usage(argv[0]);
   n = strtoll(argv[2], NULL, 10);
   if (n <= 0) Usage(argv[0]);
}  /* Get_args */


/*------------------------------------------------------------------
 * Function:  Usage
 * Purpose:   Print a message explaining how to run the program
 * In arg:    prog_name
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <number of threads> <n>\n", prog_name);
   fprintf(stderr, "   n is the number of terms and should be >= 1\n");
   fprintf(stderr, "   n should be evenly divisible by the number of threads\n");
   exit(0);
}  /* Usage */
```



pth_pi_semaphore.c - 3(Semaphore) - Visual Studio Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <semaphore.h>
#include "timer.h"

const int MAX_THREADS = 1024;

long thread_count;
long long n;
double sum;

sem_t sem;

void* Thread_sum(void* rank);

/* Only executed by main thread */
void Get_args(int argc, char* argv[]);
void Usage(char* prog_name);
double Serial_pi(long long n);

int main(int argc, char* argv[]) {
   long       thread;  /* Use long in case of a 64-bit system */
   pthread_t* thread_handles;
```

```
abid@ubuntu:~/Parallel/Homwork 3/3(Semaphore)$ gcc -g -Wall -o pth_pi_semaphore pth_pi_semaphore.c -lm -lpthread
abid@ubuntu:~/Parallel/Homwork 3/3(Semaphore)$ ./pth_pi_semaphore 4 8
With n = 8 terms,
   Our estimate of pi = 3.017071817071817
The elapsed time is 2.078056e-03 seconds
   Single thread est  = 3.017071817071818
The elapsed time is 2.098083e-05 seconds
                   pi = 3.141592653589793
abid@ubuntu:~/Parallel/Homwork 3/3(Semaphore)$ 
```

Our estimate of pi = 3.017071817071817

The elapsed time is 2.078056e-03 seconds/0.002078056 seconds

= 0.0020 seconds

Single thread est = 3.017071817071818

The elapsed time is 2.098083e-05 seconds/0.00002098083 seconds

= 0.000020 seconds

pi = 3.141592653589793

Our estimation time we got is 0.0020 seconds and single thread estimated time we get is 0.000020 seconds when we use semaphore.

Analyzation:

Here we have used semaphore instead mutex and ran both of programs. But the runtime in both cases appeared to be different.

When we use single thread then the estimated time we got is 0.000128 seconds by using mutex and again when we use single thread then the estimated time we got is 0.000020 seconds by using semaphore.

Therefore, we can say that mutex is faster than semaphore. The performance of mutex is faster than semaphore.