Lab 2 OS
Matrix Multiplication (Multi-Threading)

Name: Yousef Mohamed Hassan Elmedany.
ID: 20012293.

Code Organization:

At the beginning of my code, I defined two structs that I used then in my code to hold the data that will be needed during the multiplication as arguments. The first struct's data type objects are sent to the function that is used to compute the multiplication by assigning a thread to each row. The second struct's data type objects are sent as a parameters that hold the required data to operate the function that computes the multiplication by assigning a thread per each element of the resulting matrix. My code then begins from the main function by first checking the arguments sent on execution which are the names of files to read and write into and the code behaves according to the number of arguments sent. After that, i read the two matrices to be multiplied from the stated filenames and then allocate two 2d arrays to hold their data depending on the number of rows and columns for each matrix. I then dynamically allocate 3 "2d" matrices whose sizes are row of 1st matrix * column of 2nd matrix to hold the results of multiplication done by the three methods. After that, I then pass the data needed by each method to 3 different functions each is responsible for one of the methods and after their completion, the results are now assigned in the three matrices that I allocated to hold the results. I placed the three functions one after another in my code starting with the basic computation "a thread per matrix" then "a thread per row" and finally "a thread per element". And after he completion of each, I create 3 txt files with the name passes at the beginning as an argument representing the output name, and each file is written by one of the three methods. After writing results into the files, I de-allocate the memory spaces used during the code execution that I allocated manually (to optimize the space in memory) that contained the arrays storing matrices. At the end, I print out to the console the number of threads created per each method and its execution time in seconds and microseconds to provide a comparison among the three methods.

Main Functions:

in my code, I declared and implemented 5 different main functions that are used to perform different operation. The functions are:

- int thread_per_matrix():
  this function is called to handle the multiplication using the normal method which is assigning a single thread to compute the results of multiplication.
- int thread_per_row():
  this function is called to apply the multiplication of the two matrices by creating and assigning a thread per each row for the first matrix to obtain the corresponding row in the resulting matrix.
- int thread_per_element():
  this function is use to compute the matrix multiplication by creating and assigning a thread to compute each element in the resulting matrix. This method creates number of threads equal to the size of the resulting matrices (row of first matrix*column of second matrix).
- void* multiply_by_row():
  this function is passed as an argument to the function pthread_create() which creates a new thread and this thread performs the logic inside this function and it is passed within "thread_per_row" function when it creates a new thread. The function multiplies the row of the first matrix passed to it with the whole second matrix to obtain the corresponding row in the resulting matrix.
- void* multiply_by_element():
  this function is passed as an argument to the function pthread_create() which creates a new thread and this thread performs the logic inside this function and it is passed within "thread_per_element" function when it creates a new thread. The function multiplies the passed row of the first matrix with the passed column of the second matrix to obtain the corresponding element in the resulting matrix.

How to compile and run my code?

You can simply compile and run the code by just executing the makefile supplied within the code either in your terminal or within your IDE and then execute "./matMultp a b c", where a is the name of the text file to read the first matrix from, b is the name of the text file to read the second matrix from and c is the name of the file to save the resulting matrices in.

Note that: you have to write and save the matrices to be multiplied inside the two files you want to read from then before running the code.

Sample Runs:

1.

matrix A

```
row=10 col=5
1       2       3       4       5
6       7       8       9       10
11      12      13      14      15
16      17      18      19      20
21      22      23      24      25
26      27      28      29      30
31      32      33      34      35
36      37      38      39      40
41      42      43      44      45
46      47      48      49      50
```

matrix B

```
row=5 col=10
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38      39      40
41      42      43      44      45      46      47      48      49      50
```

```
Method: A thread per element
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300
```

```
Method: A thread per matrix
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300
```

```
Method: A thread per row
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 3600 3740 3880 4020 4160 4300
3565 3730 3895 4060 4225 4390 4555 4720 4885 5050
4090 4280 4470 4660 4850 5040 5230 5420 5610 5800
4615 4830 5045 5260 5475 5690 5905 6120 6335 6550
5140 5380 5620 5860 6100 6340 6580 6820 7060 7300
```

```
yousef@yousef-TP:~/Desktop/C files/Lab2_OS$ ./lab2 a b c
Method 1: A thread per matrix
Number of threads created = 1
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 6

Method 2: A thread per row
Number of threads created = 10
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 949

Method 3: A thread per element
Number of threads created = 100
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 4949
```

2.

matrix A

```
row=3 col=5
1	-2	3	4	5
1	2	-3	4	5
-1	2	3	4	5
```

matrix B

```
row=5 col=4
-1	2	3	4
1	-2	3	4
1	2	-3	4
1	2	3	-4
-1	-2	-3	-4
```

```
Method: A thread per element
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

```
Method: A thread per matrix
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

```
Method: A thread per row
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

```
yousef@yousef-TP:~/Desktop/C files/Lab2_OS$ ./lab2 a b c
Method 1: A thread per matrix
Number of threads created = 1
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 1

Method 2: A thread per row
Number of threads created = 3
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 284

Method 3: A thread per element
Number of threads created = 12
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 746
```

3.

matrix A

```
row=5 col=5
1       2       3       4       5
6       7       8       9       10
11      12      13      14      15
16      17      18      19      20
21      22      23      24      25
```

matrix B

```
row=5 col=4
1       2       3       4
5       6       7       8
9       10      11      12
13      14      15      16
17      18      19      20
```

```
Method: A thread per element
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420
```

```
Method: A thread per matrix
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420
```

```
Method: A thread per row
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420
```

```
yousef@yousef-TP:~/Desktop/C files/Lab2_OS$ ./lab2 a b c
Method 1: A thread per matrix
Number of threads created = 1
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 2

Method 2: A thread per row
Number of threads created = 5
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 1096

Method 3: A thread per element
Number of threads created = 20
Time taken for this method in Seconds = 0
Time taken for this method in Microseconds= 5583
```

Comparison among the three methods of Multiplication:

the results of the sample runs show that the least execution time taken was that by the method that creates a thread per matrix and the second least time was that which creates a thread per each row and the method that has the highest execution time was that which creates a thread per each element.

But why the first method is performing better than the others?

The reason is that creating and handling threads requires extra overhead and lots of computation, so, it's not always the ideal solution to go for multi-threading, and there's always a tradeoff.

In designing multi-threaded programs, we should always consider many things, including the threading overhead, the size of the problem, and the level of concurrency. There's no rule for that, we just make our own analysis and take our decision accordingly.