Alexandria University

Faculty Of Engineering

Computer & Systems Engineering Department

جامعة الاسكندرية
كلية الهندسة

❖ **OS lab #2 report**

❖ **The report includes the following:**

- ❖ A description of the overall organization of my code.
- ❖ Main functions of my code.
- ❖ how to compile and run my code.
- ❖ Sample runs.
- ❖ A comparison between the three methods of matrix multiplication

---

❖ **Student info:**

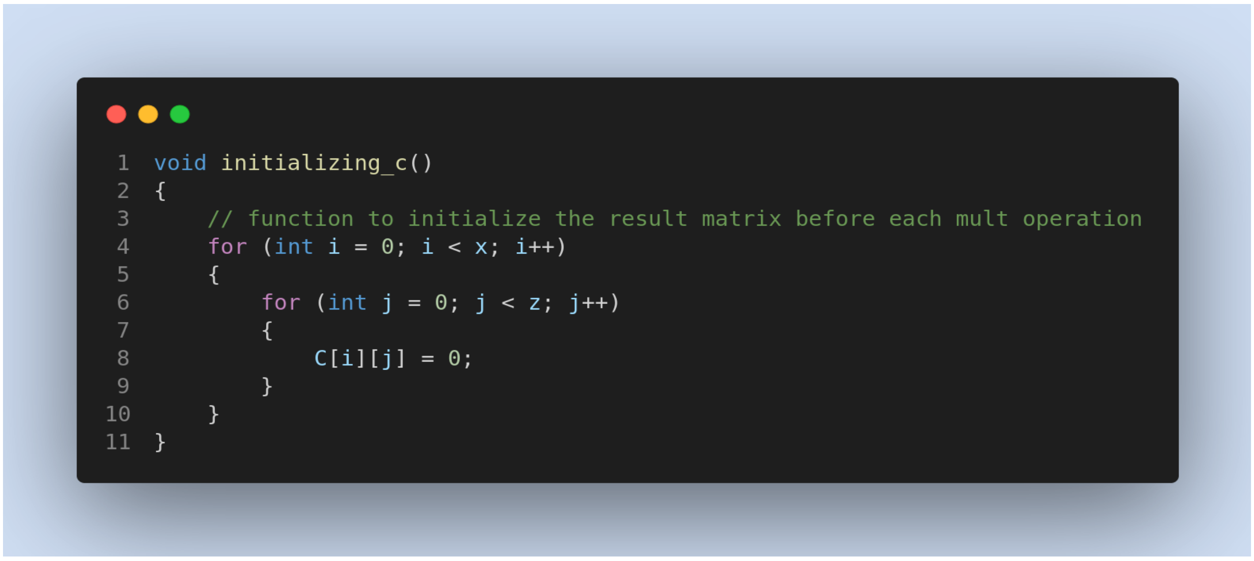
Adel Mahmoud Mohamed Abdelrahman	20010769

A description of the over all organization of my code:

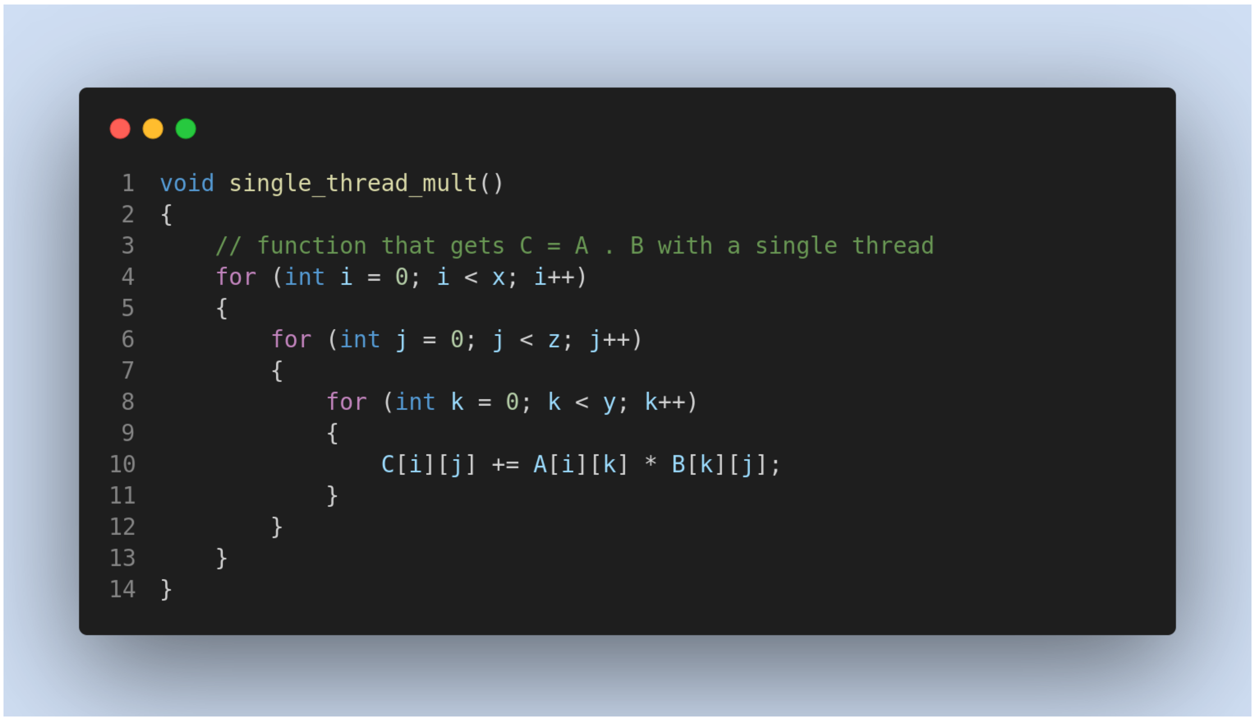
- The code is broken into functions each with a specific responsibility besides the main function.
- There's a function that gets the whole matrix multiplication in the one thread.
- Another function that gets it by making a thread to get each row in the result matrix
- A third function to get it by making a thread to calculate an element in the result marix
- matMult() function where all the kinds of matrix multiplication take place by calling the functions described above.
- The user enters the input as text files by passing the names as arguments when running the executable file(You'll see it in the **How to compile and run section**).
- The output matrix will be provided in the file names as desired and it will be in a file with prefix **c** if not provided.

The main functions of my code:

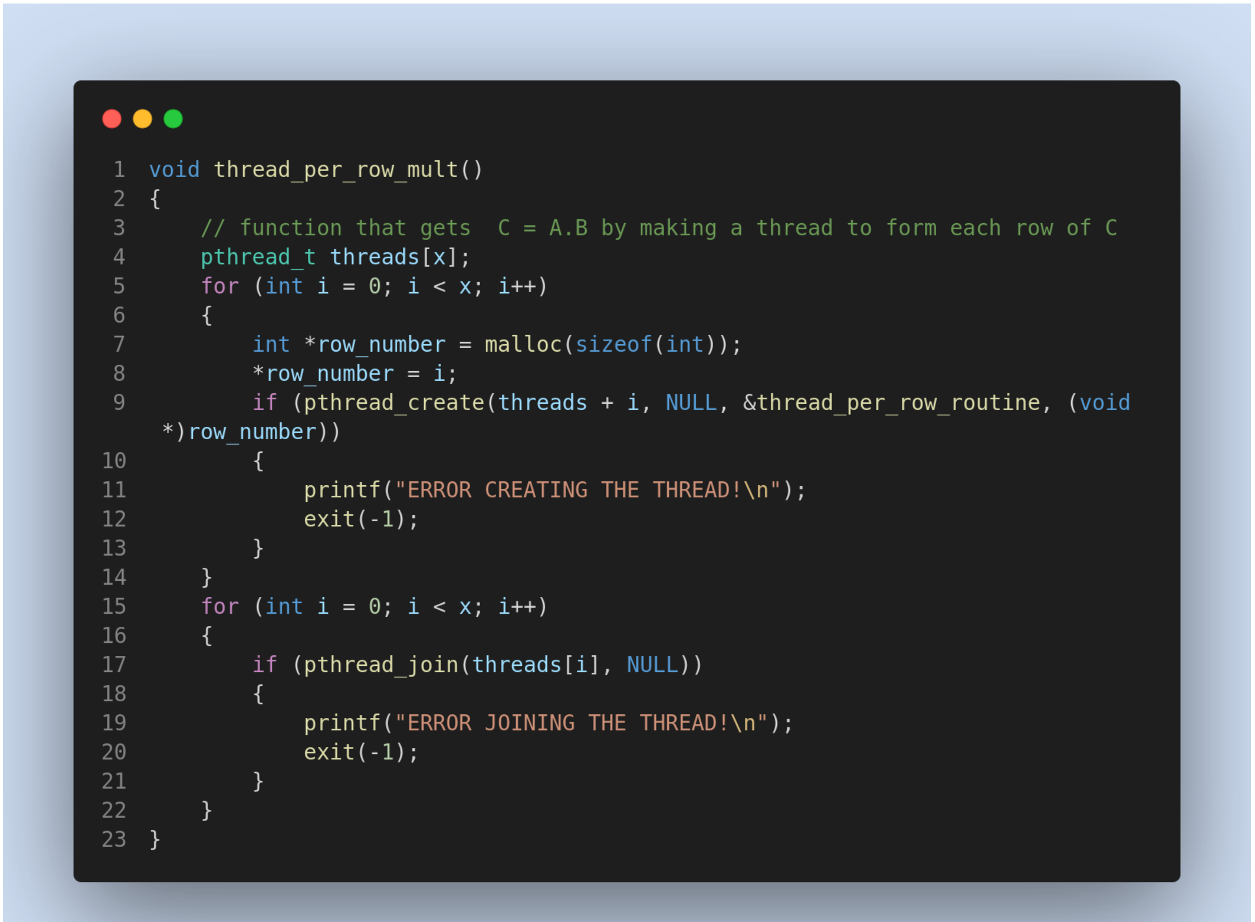
- Each function is clearly commented at its beginning to indicate what its purpose is.
- There's another function called readB() that's the same as readA() but it reads the B out of the its file.



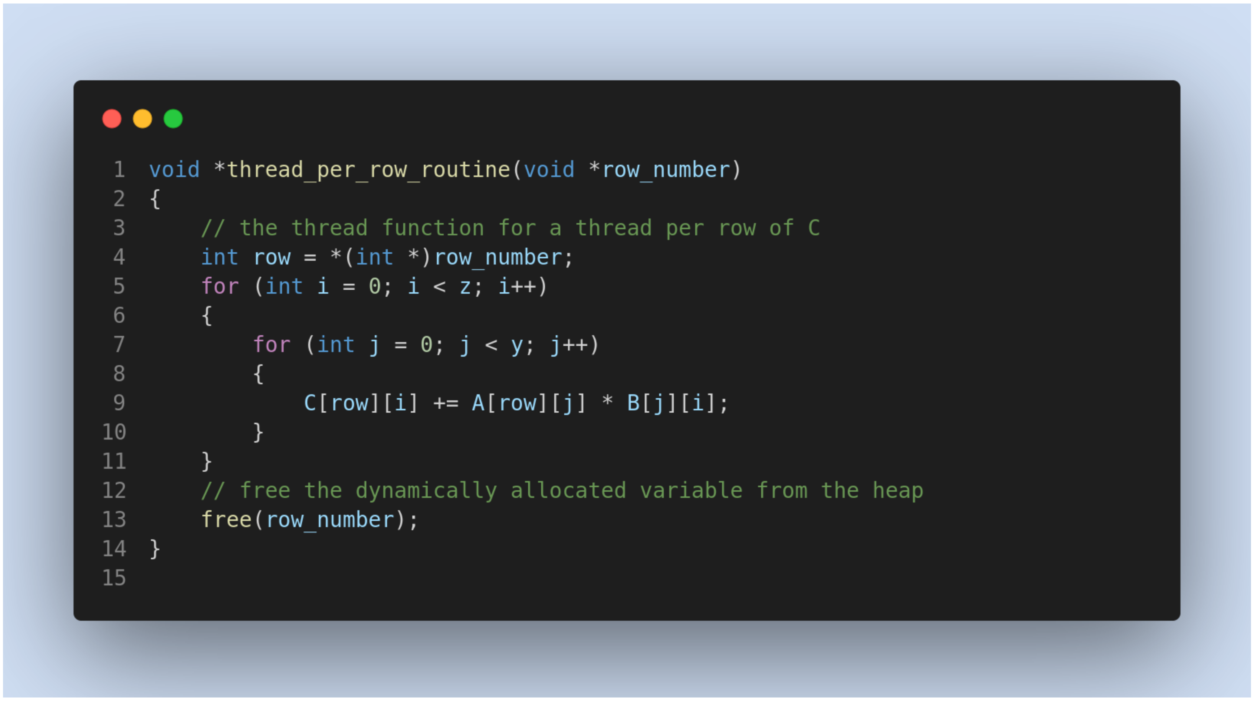
```
void initializing_c()
{
    // function to initialize the result matrix before each mult operation
    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < z; j++)
        {
            C[i][j] = 0;
        }
    }
}
```



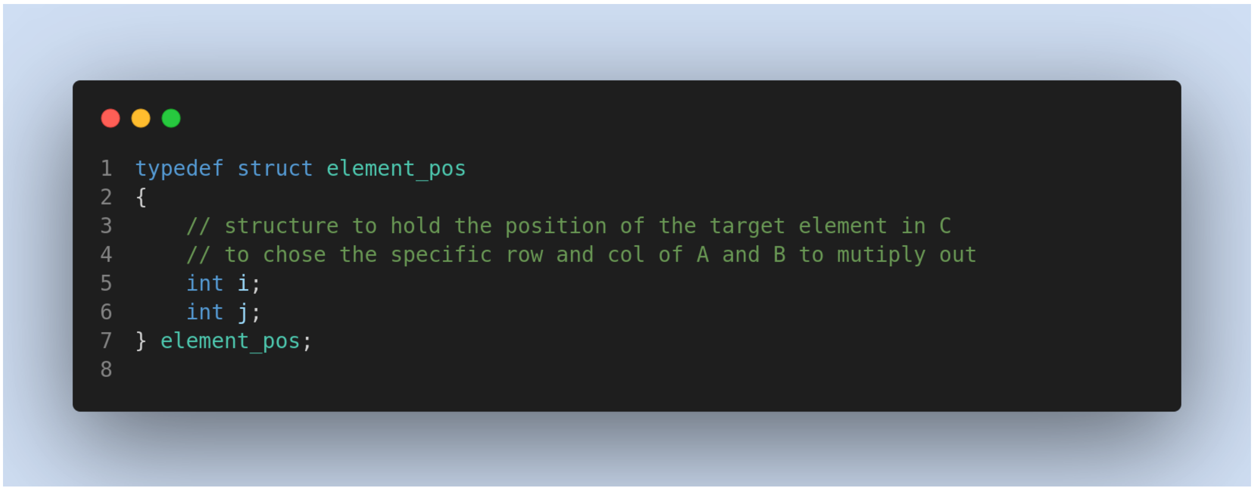
```c
void single_thread_mult()
{
    // function that gets C = A . B with a single thread
    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < z; j++)
        {
            for (int k = 0; k < y; k++)
            {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
}
```



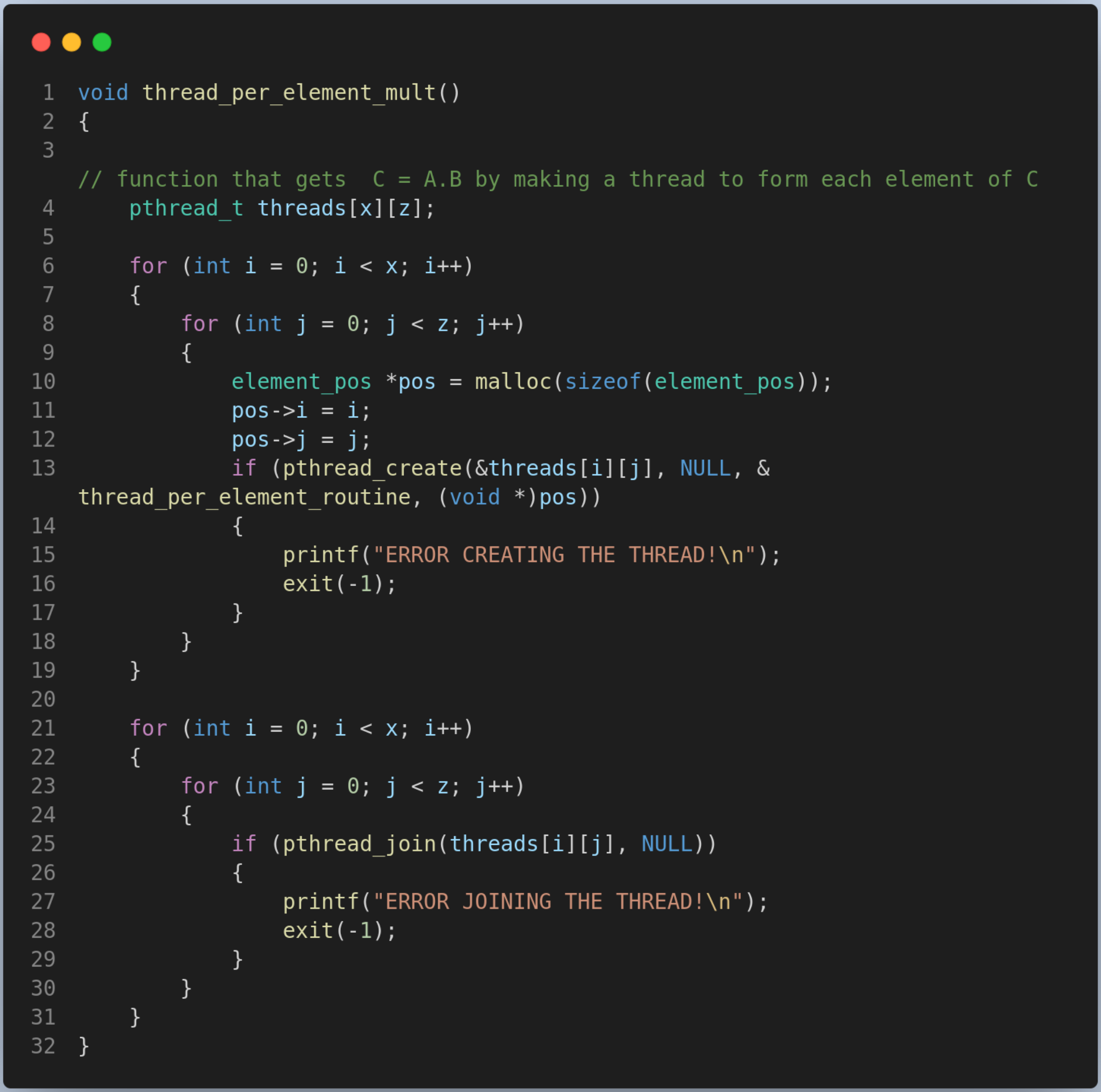
```c
void thread_per_row_mult()
{
    // function that gets  C = A.B by making a thread to form each row of C
    pthread_t threads[x];
    for (int i = 0; i < x; i++)
    {
        int *row_number = malloc(sizeof(int));
        *row_number = i;
        if (pthread_create(threads + i, NULL, &thread_per_row_routine, (void
 *)row_number))
        {
            printf("ERROR CREATING THE THREAD!\n");
            exit(-1);
        }
    }
    for (int i = 0; i < x; i++)
    {
        if (pthread_join(threads[i], NULL))
        {
            printf("ERROR JOINING THE THREAD!\n");
            exit(-1);
        }
    }
}
```



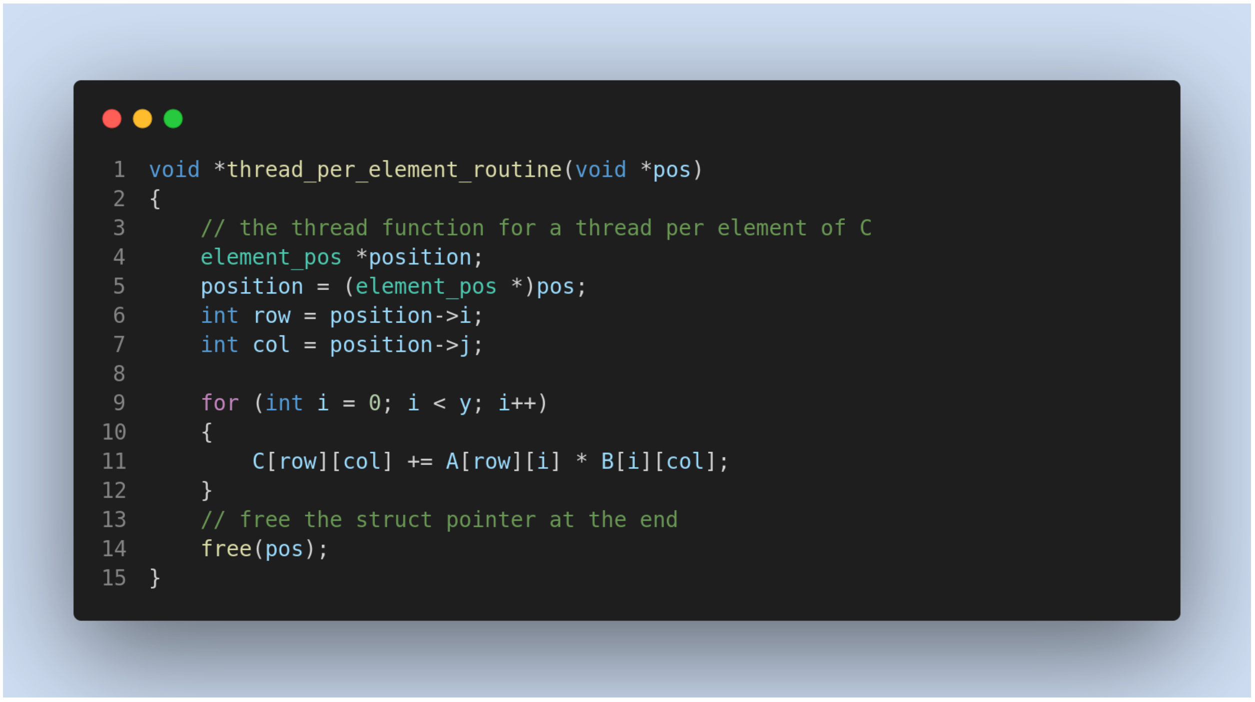
```c
void *thread_per_row_routine(void *row_number)
{
    // the thread function for a thread per row of C
    int row = *(int *)row_number;
    for (int i = 0; i < z; i++)
    {
        for (int j = 0; j < y; j++)
        {
            C[row][i] += A[row][j] * B[j][i];
        }
    }
    // free the dynamically allocated variable from the heap
    free(row_number);
}

```



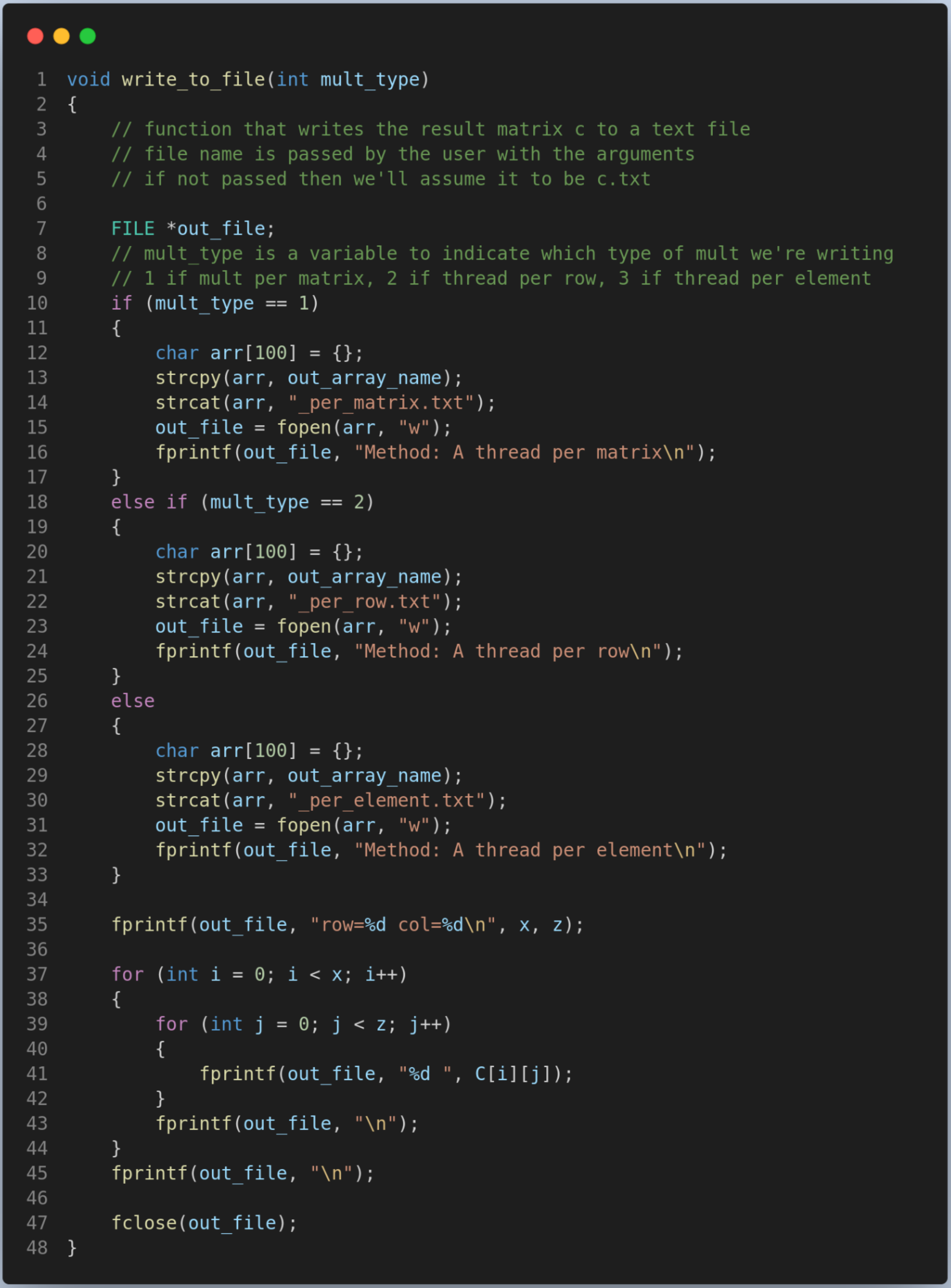
```c
typedef struct element_pos
{
    // structure to hold the position of the target element in C
    // to chose the specific row and col of A and B to mutiply out
    int i;
    int j;
} element_pos;

```



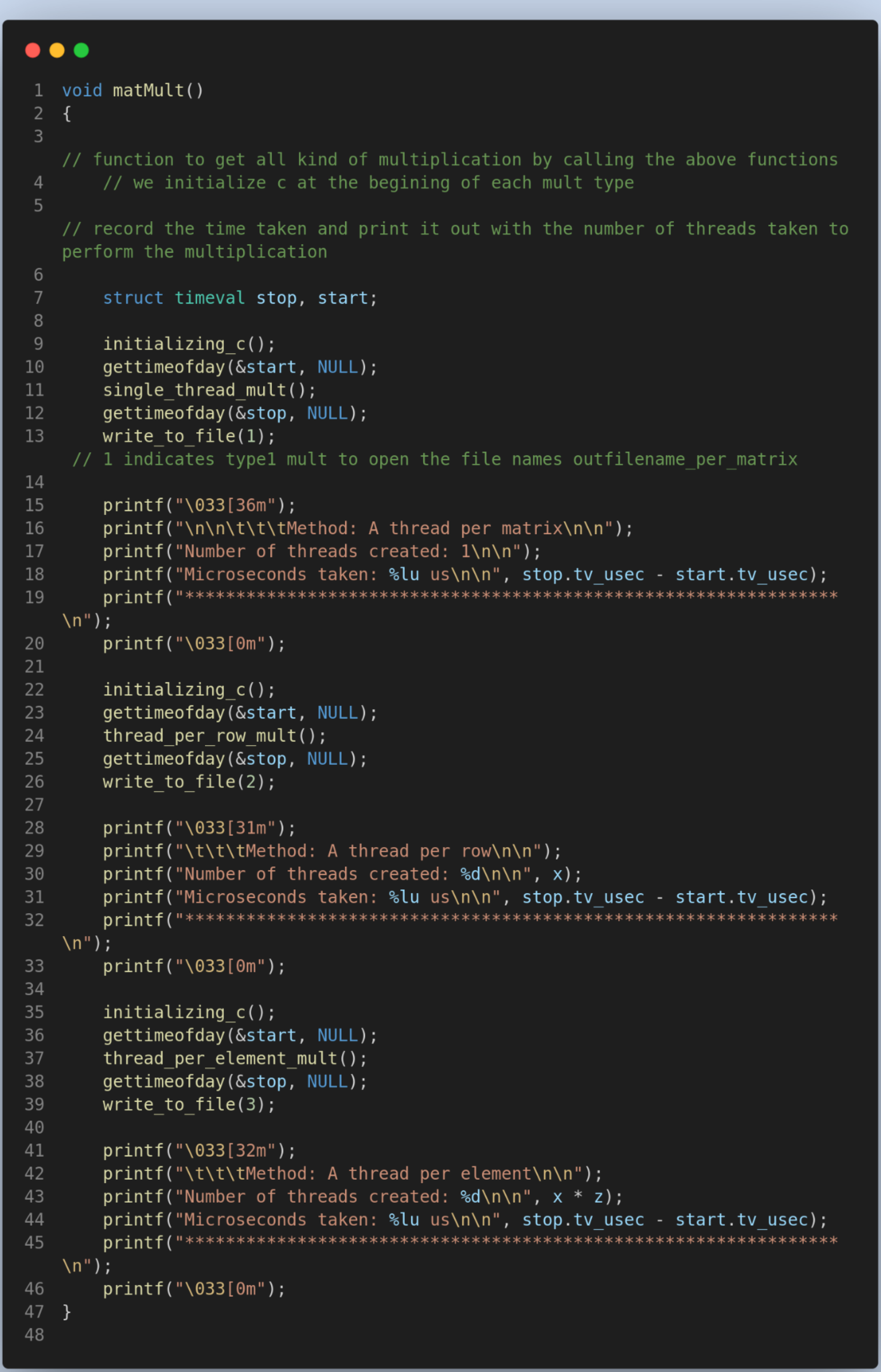
```c
void thread_per_element_mult()
{

// function that gets  C = A.B by making a thread to form each element of C
    pthread_t threads[x][z];

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < z; j++)
        {
            element_pos *pos = malloc(sizeof(element_pos));
            pos->i = i;
            pos->j = j;
            if (pthread_create(&threads[i][j], NULL, &
thread_per_element_routine, (void *)pos))
            {
                printf("ERROR CREATING THE THREAD!\n");
                exit(-1);
            }
        }
    }

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < z; j++)
        {
            if (pthread_join(threads[i][j], NULL))
            {
                printf("ERROR JOINING THE THREAD!\n");
                exit(-1);
            }
        }
    }
}
```



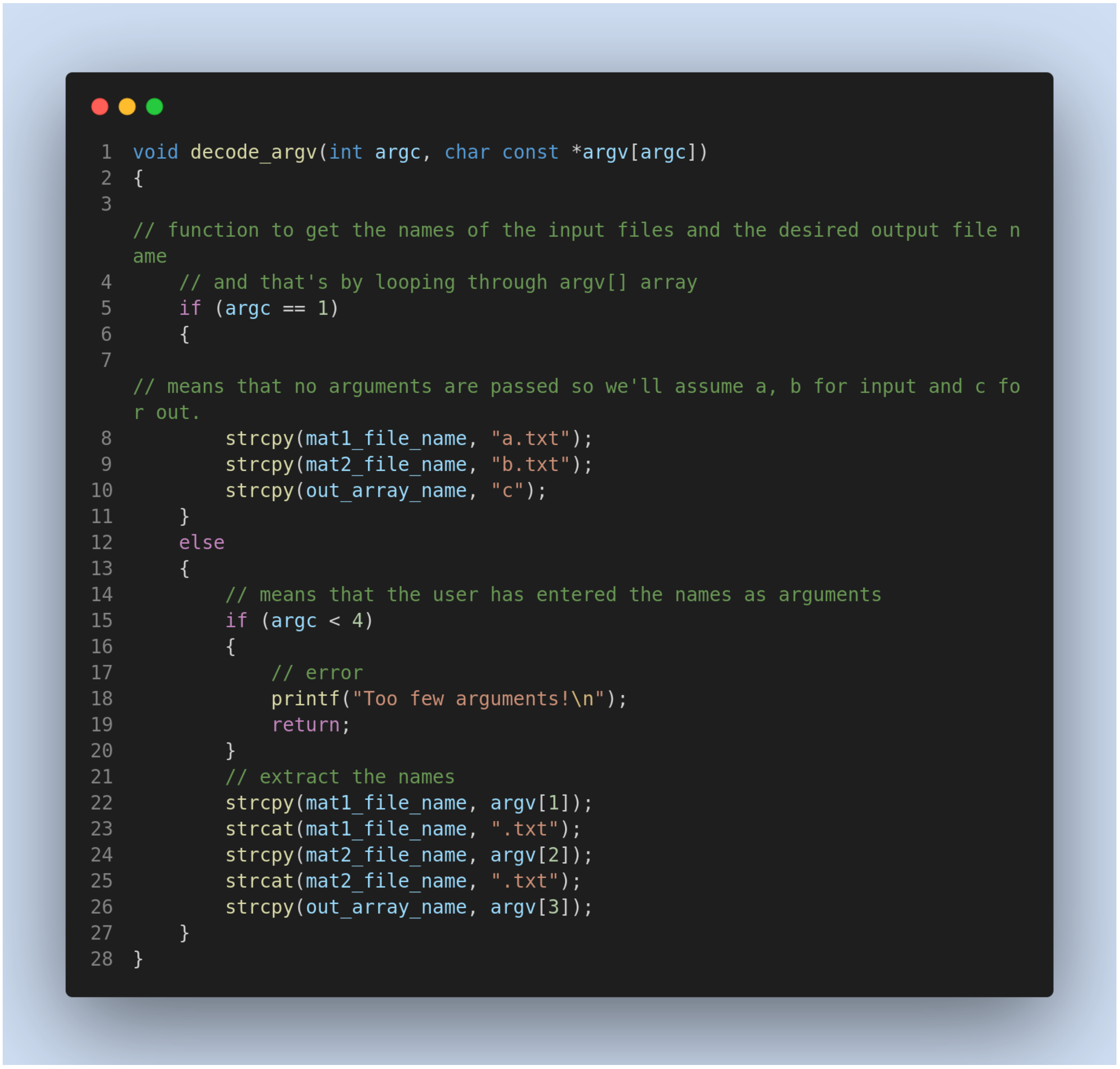
```c
void *thread_per_element_routine(void *pos)
{
    // the thread function for a thread per element of C
    element_pos *position;
    position = (element_pos *)pos;
    int row = position->i;
    int col = position->j;

    for (int i = 0; i < y; i++)
    {
        C[row][col] += A[row][i] * B[i][col];
    }
    // free the struct pointer at the end
    free(pos);
}
```



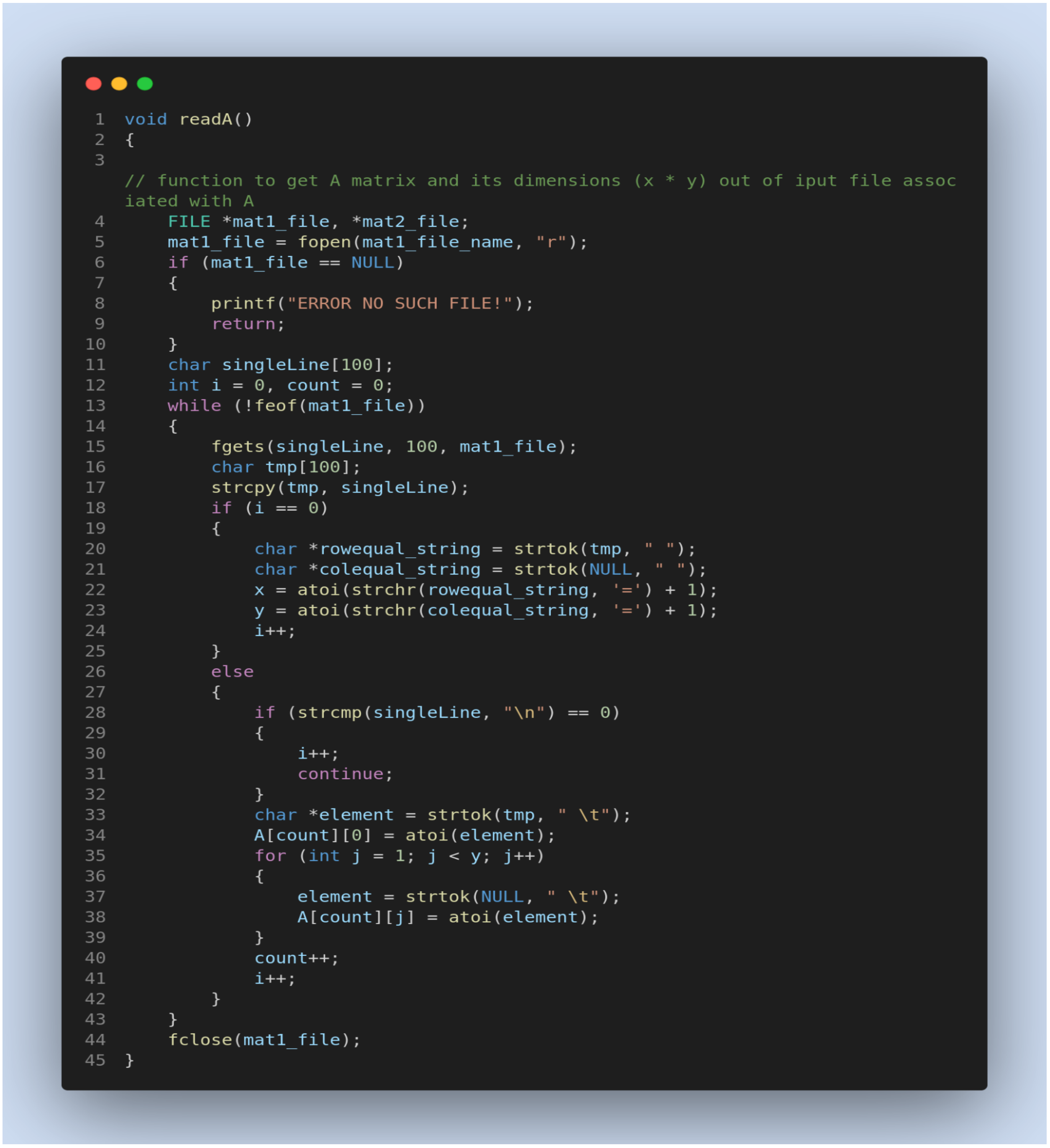
```c
void write_to_file(int mult_type)
{
    // function that writes the result matrix c to a text file
    // file name is passed by the user with the arguments
    // if not passed then we'll assume it to be c.txt

    FILE *out_file;
    // mult_type is a variable to indicate which type of mult we're writing
    // 1 if mult per matrix, 2 if thread per row, 3 if thread per element
    if (mult_type == 1)
    {
        char arr[100] = {};
        strcpy(arr, out_array_name);
        strcat(arr, "_per_matrix.txt");
        out_file = fopen(arr, "w");
        fprintf(out_file, "Method: A thread per matrix\n");
    }
    else if (mult_type == 2)
    {
        char arr[100] = {};
        strcpy(arr, out_array_name);
        strcat(arr, "_per_row.txt");
        out_file = fopen(arr, "w");
        fprintf(out_file, "Method: A thread per row\n");
    }
    else
    {
        char arr[100] = {};
        strcpy(arr, out_array_name);
        strcat(arr, "_per_element.txt");
        out_file = fopen(arr, "w");
        fprintf(out_file, "Method: A thread per element\n");
    }

    fprintf(out_file, "row=%d col=%d\n", x, z);

    for (int i = 0; i < x; i++)
    {
        for (int j = 0; j < z; j++)
        {
            fprintf(out_file, "%d ", C[i][j]);
        }
        fprintf(out_file, "\n");
    }
    fprintf(out_file, "\n");

    fclose(out_file);
}
```



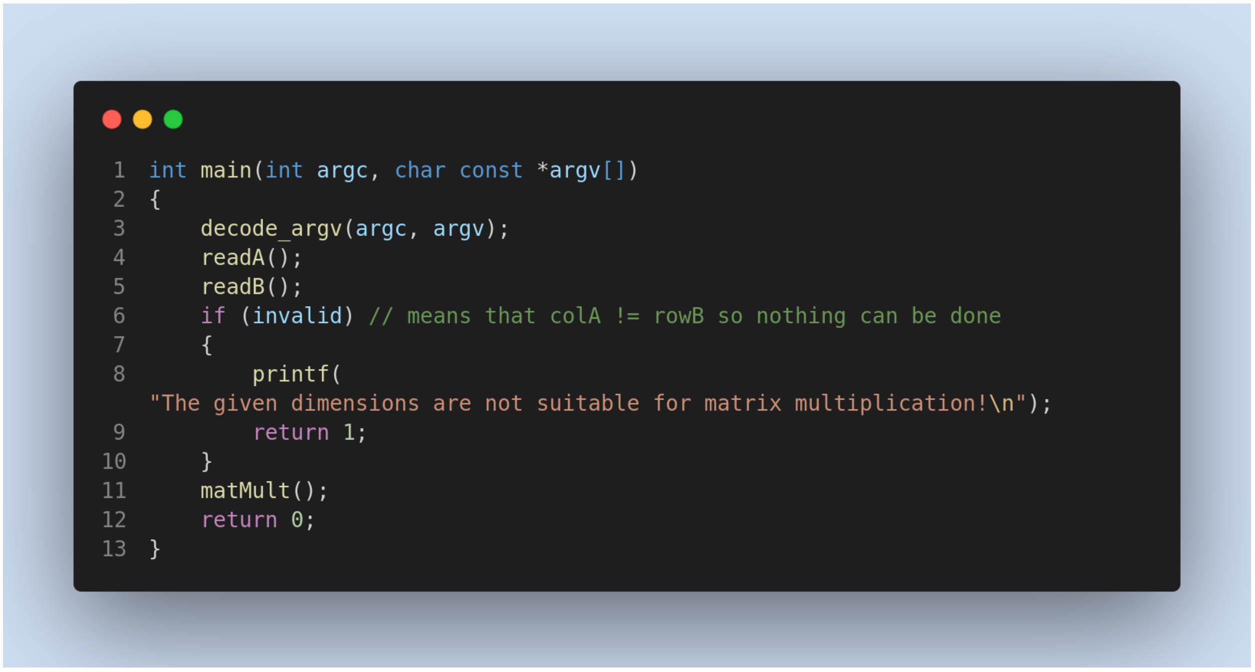
```c
void matMult()
{

// function to get all kind of multiplication by calling the above functions
    // we initialize c at the begining of each mult type

// record the time taken and print it out with the number of threads taken to perform the multiplication

    struct timeval stop, start;

    initializing_c();
    gettimeofday(&start, NULL);
    single_thread_mult();
    gettimeofday(&stop, NULL);
    write_to_file(1);
 // 1 indicates type1 mult to open the file names outfilename_per_matrix

    printf("\033[36m");
    printf("\n\n\t\t\tMethod: A thread per matrix\n\n");
    printf("Number of threads created: 1\n\n");
    printf("Microseconds taken: %lu us\n\n", stop.tv_usec - start.tv_usec);
    printf("****************************************************************\n");
    printf("\033[0m");

    initializing_c();
    gettimeofday(&start, NULL);
    thread_per_row_mult();
    gettimeofday(&stop, NULL);
    write_to_file(2);

    printf("\033[31m");
    printf("\t\t\tMethod: A thread per row\n\n");
    printf("Number of threads created: %d\n\n", x);
    printf("Microseconds taken: %lu us\n\n", stop.tv_usec - start.tv_usec);
    printf("****************************************************************\n");
    printf("\033[0m");

    initializing_c();
    gettimeofday(&start, NULL);
    thread_per_element_mult();
    gettimeofday(&stop, NULL);
    write_to_file(3);

    printf("\033[32m");
    printf("\t\t\tMethod: A thread per element\n\n");
    printf("Number of threads created: %d\n\n", x * z);
    printf("Microseconds taken: %lu us\n\n", stop.tv_usec - start.tv_usec);
    printf("****************************************************************\n");
    printf("\033[0m");
}
```



```c
void decode_argv(int argc, char const *argv[argc])
{
// function to get the names of the input files and the desired output file name
    // and that's by looping through argv[] array
    if (argc == 1)
    {
// means that no arguments are passed so we'll assume a, b for input and c for out.
        strcpy(mat1_file_name, "a.txt");
        strcpy(mat2_file_name, "b.txt");
        strcpy(out_array_name, "c");
    }
    else
    {
        // means that the user has entered the names as arguments
        if (argc < 4)
        {
            // error
            printf("Too few arguments!\n");
            return;
        }
        // extract the names
        strcpy(mat1_file_name, argv[1]);
        strcat(mat1_file_name, ".txt");
        strcpy(mat2_file_name, argv[2]);
        strcat(mat2_file_name, ".txt");
        strcpy(out_array_name, argv[3]);
    }
}
```



```c
void readA()
{

// function to get A matrix and its dimensions (x * y) out of iput file assoc
iated with A
    FILE *mat1_file, *mat2_file;
    mat1_file = fopen(mat1_file_name, "r");
    if (mat1_file == NULL)
    {
        printf("ERROR NO SUCH FILE!");
        return;
    }
    char singleLine[100];
    int i = 0, count = 0;
    while (!feof(mat1_file))
    {
        fgets(singleLine, 100, mat1_file);
        char tmp[100];
        strcpy(tmp, singleLine);
        if (i == 0)
        {
            char *rowequal_string = strtok(tmp, " ");
            char *colequal_string = strtok(NULL, " ");
            x = atoi(strchr(rowequal_string, '=') + 1);
            y = atoi(strchr(colequal_string, '=') + 1);
            i++;
        }
        else
        {
            if (strcmp(singleLine, "\n") == 0)
            {
                i++;
                continue;
            }
            char *element = strtok(tmp, " \t");
            A[count][0] = atoi(element);
            for (int j = 1; j < y; j++)
            {
                element = strtok(NULL, " \t");
                A[count][j] = atoi(element);
            }
            count++;
            i++;
        }
    }
    fclose(mat1_file);
}
```

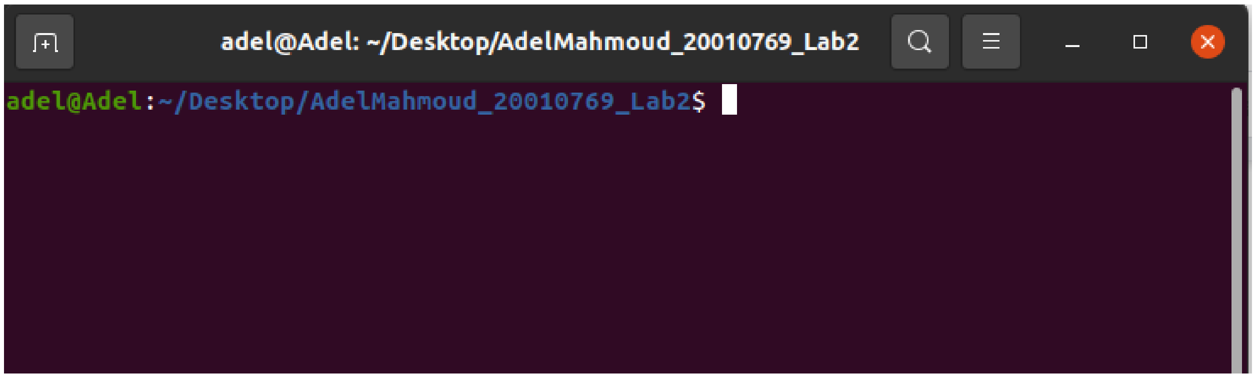


```c
int main(int argc, char const *argv[])
{
    decode_argv(argc, argv);
    readA();
    readB();
    if (invalid) // means that colA != rowB so nothing can be done
    {
        printf(
"The given dimensions are not suitable for matrix multiplication!\n");
        return 1;
    }
    matMult();
    return 0;
}
```

How to compile and run my code:

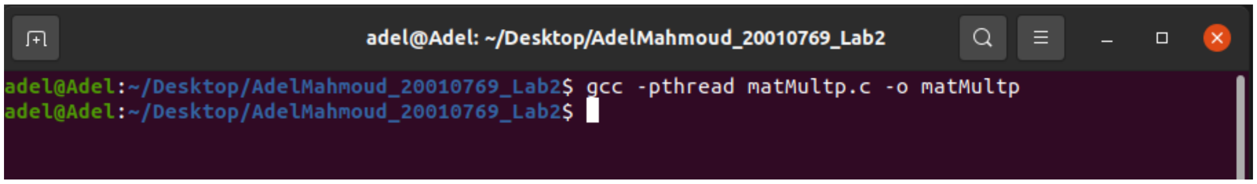
It's pretty easy, follow the steps and you'll get there:

1. Open the terminal on the directory witch contains the source code.



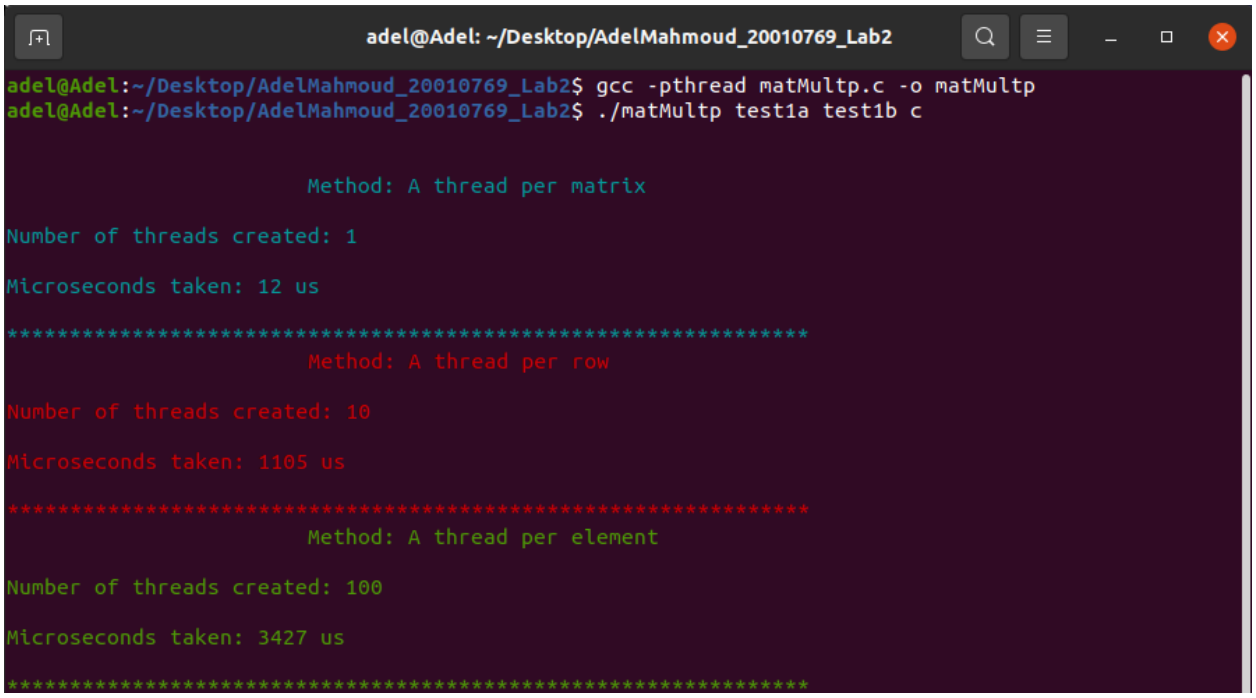
```
adel@Adel:~/Desktop/AdelMahmoud_20010769_Lab2$ 
```

2. I've named the c file as matMultp.c so you'll use this name to compile the c file by invoking this command



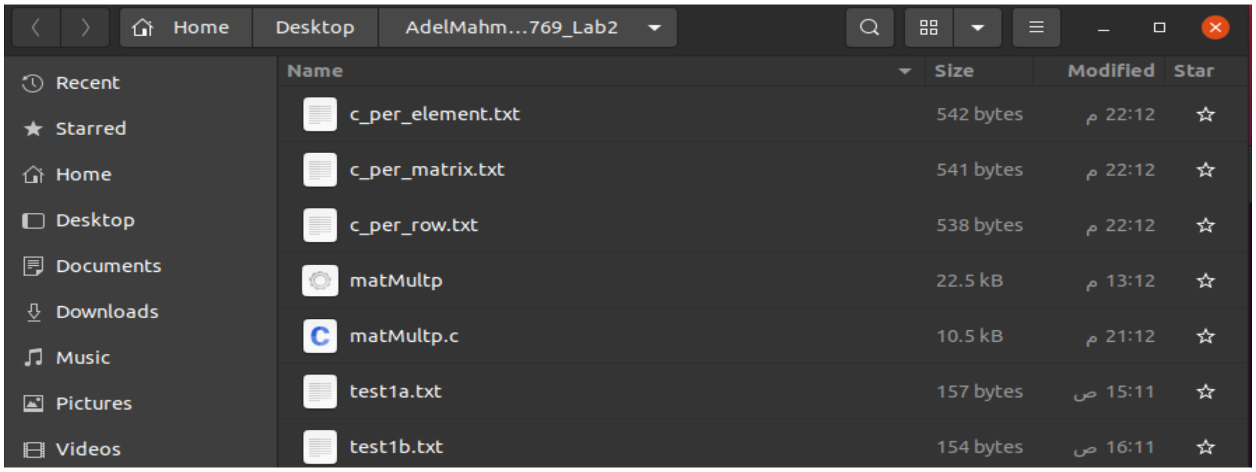
```
adel@Adel:~/Desktop/AdelMahmoud_20010769_Lab2$ gcc -pthread matMultp.c -o matMultp
adel@Adel:~/Desktop/AdelMahmoud_20010769_Lab2$ 
```

3. Great job ! now the code is successfully compiled and it's compile-time-error free now we want to execute it, the executable file is named matMultp and of course there are the input files which contain the matrices and the desired output file name here are the two paths you may take:
   - Either to not specify any name and the default names then will be a, b and c for the file names.
   - Or you can specify the specific names by passing them after **./matMultp()**



```
adel@Adel:~/Desktop/AdelMahmoud_20010769_Lab2$ gcc -pthread matMultp.c -o matMultp
adel@Adel:~/Desktop/AdelMahmoud_20010769_Lab2$ ./matMultp test1a test1b c


                        Method: A thread per matrix

Number of threads created: 1

Microseconds taken: 12 us

*****************************************************************
                        Method: A thread per row

Number of threads created: 10

Microseconds taken: 1105 us

*****************************************************************
                        Method: A thread per element

Number of threads created: 100

Microseconds taken: 3427 us

*****************************************************************
```

Here you can see that the program has run successfully because the text files named **test1a.txt** and **test1b**.txt are found in the code directory, and you can see also that the desired output file is c so you'll find you're results in 3 different text files with a prefix c and here's the code directory:



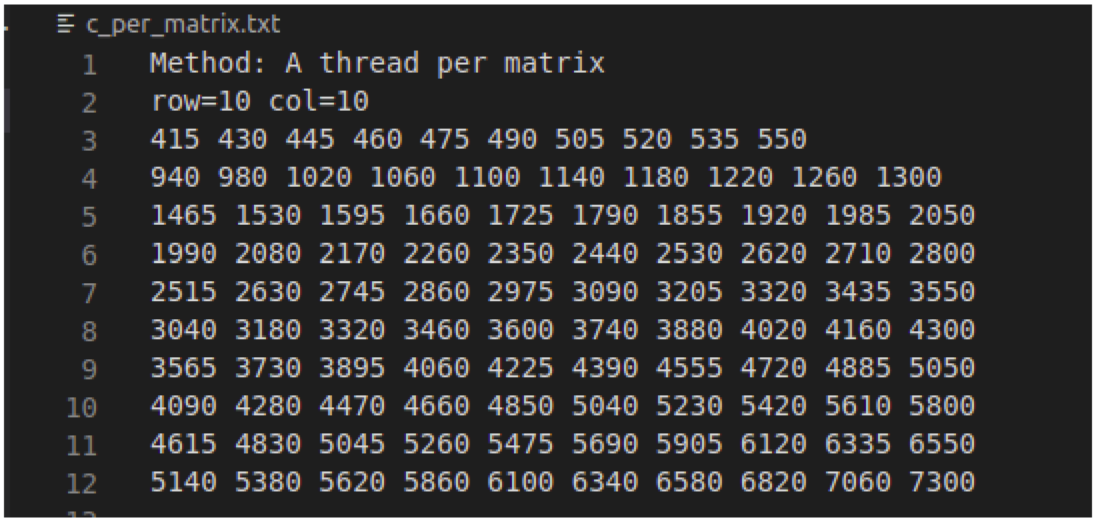
Name	Size	Modified	Star
c_per_element.txt	542 bytes	22:12 م	☆
c_per_matrix.txt	541 bytes	22:12 م	☆
c_per_row.txt	538 bytes	22:12 م	☆
matMultp	22.5 kB	13:12 م	☆
matMultp.c	10.5 kB	21:12 م	☆
test1a.txt	157 bytes	15:11 ص	☆
test1b.txt	154 bytes	16:11 ص	☆

NOTE:

- Note that the format in which the input must be written in the input files provided must be like this:

```
row=3 col=5
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
```

- And the output format in c-prefixed files you'll find it like this, here's a shot form one file and the rest are in the same format:



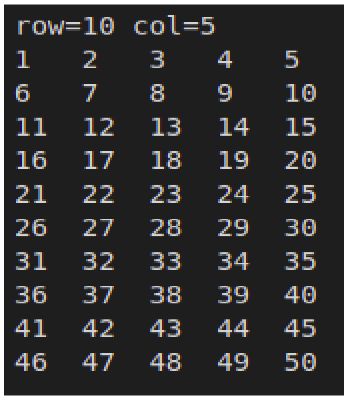
```
c_per_matrix.txt
Method: A thread per matrix
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
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
```

-

- **Note that if the input files names are not found then an error message will show and the program is terminated**

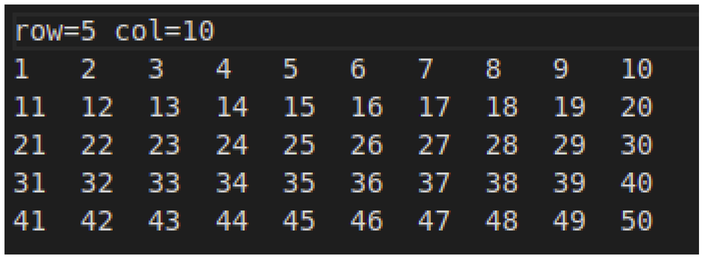
Sample runs:

- **Test 1:**
  The matrix A is:



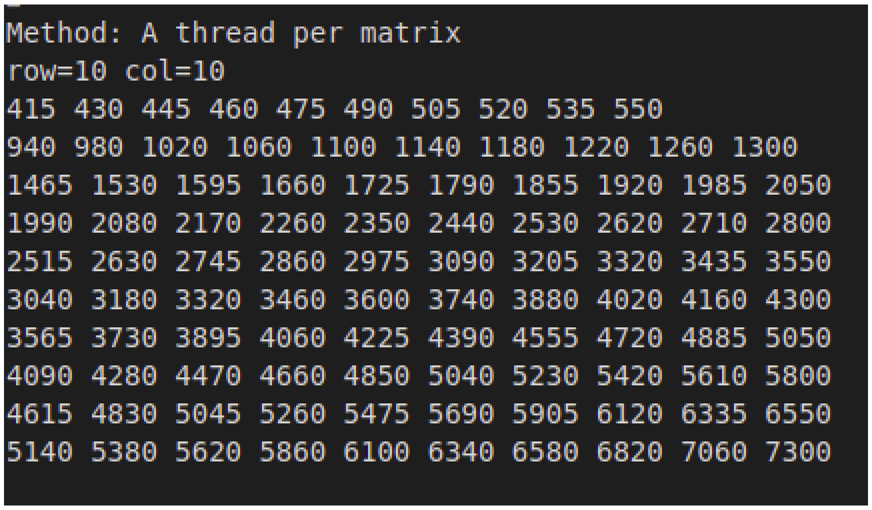
```
row=10 col=5
1    2    3    4    5
6    7    8    9    10
11   12   13   14   15
16   17   18   19   20
21   22   23   24   25
26   27   28   29   30
31   32   33   34   35
36   37   38   39   40
41   42   43   44   45
46   47   48   49   50
```

The matrix B is:



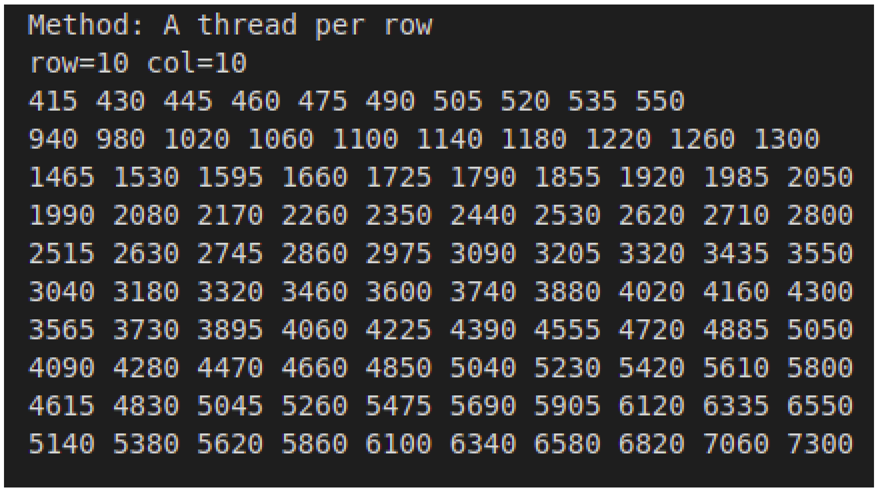
```
row=5 col=10
1    2    3    4    5    6    7    8    9    10
11   12   13   14   15   16   17   18   19   20
21   22   23   24   25   26   27   28   29   30
31   32   33   34   35   36   37   38   39   40
41   42   43   44   45   46   47   48   49   50
```

The result matrix with method one(thread per matrix)



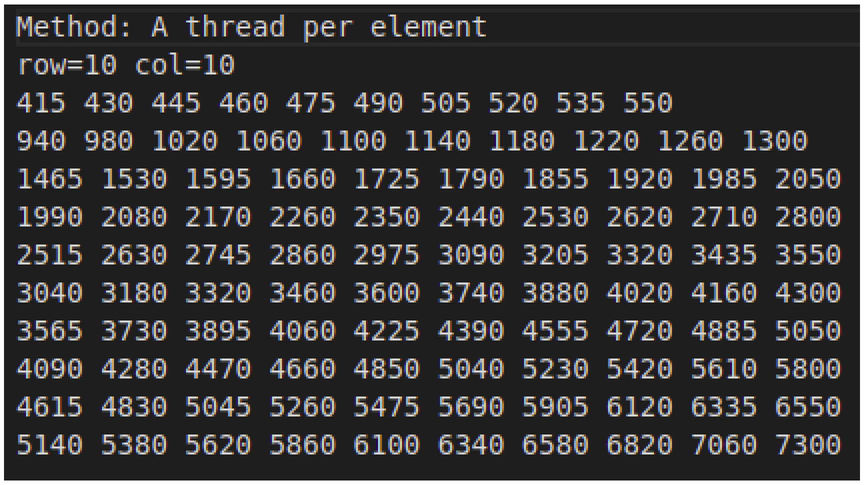
```
Method: A thread per matrix
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 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
```

The result matrix with the second method(thread per row)

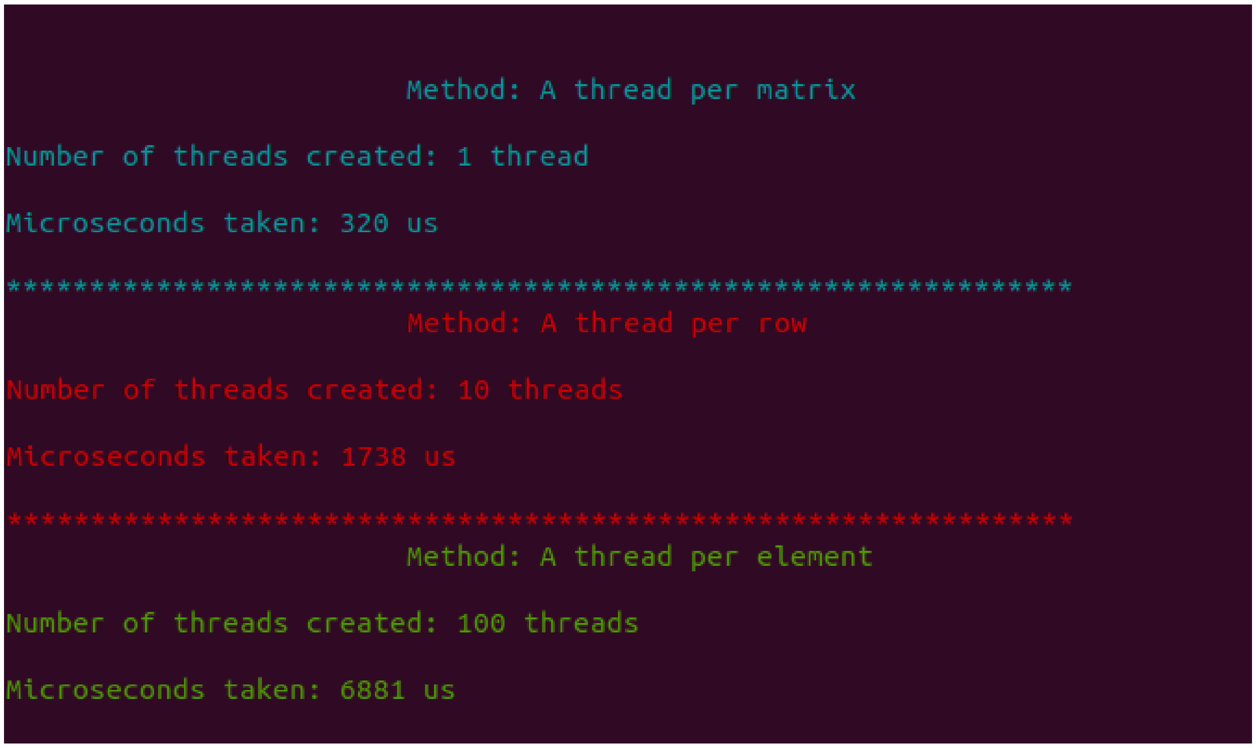


```
Method: A thread per row
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460 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
```

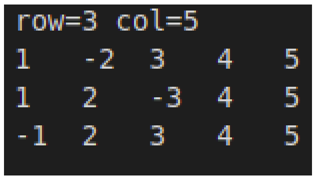
The result matrix with the third method(thread per element)


```
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
```

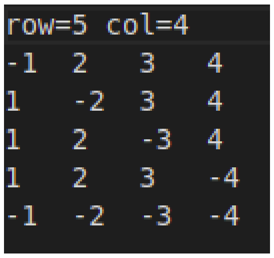
The time taken by each method and the number of threads created for each:


```
                    Method: A thread per matrix

Number of threads created: 1 thread

Microseconds taken: 320 us

*****************************************************************
                    Method: A thread per row

Number of threads created: 10 threads

Microseconds taken: 1738 us

*****************************************************************
                    Method: A thread per element

Number of threads created: 100 threads

Microseconds taken: 6881 us
```

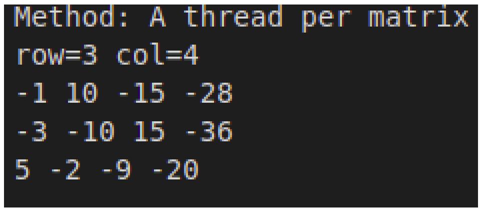
- **Test 2:**
  The matrix A is:


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

The matrix B is:

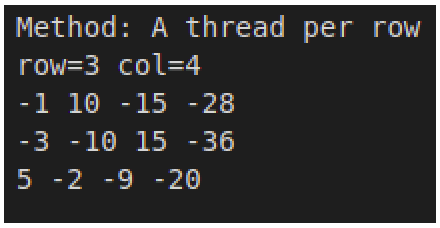

```
row=5 col=4
-1  2   3   4
1   -2  3   4
1   2   -3  4
1   2   3   -4
-1  -2  -3  -4
```

The result matrix using thread per matrix method:



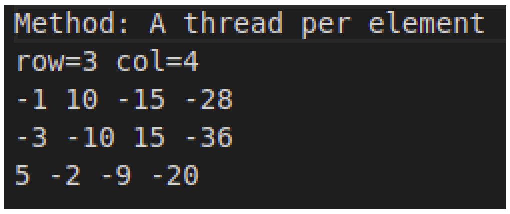
```
Method: A thread per matrix
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

The resule matrix using thread per row method:



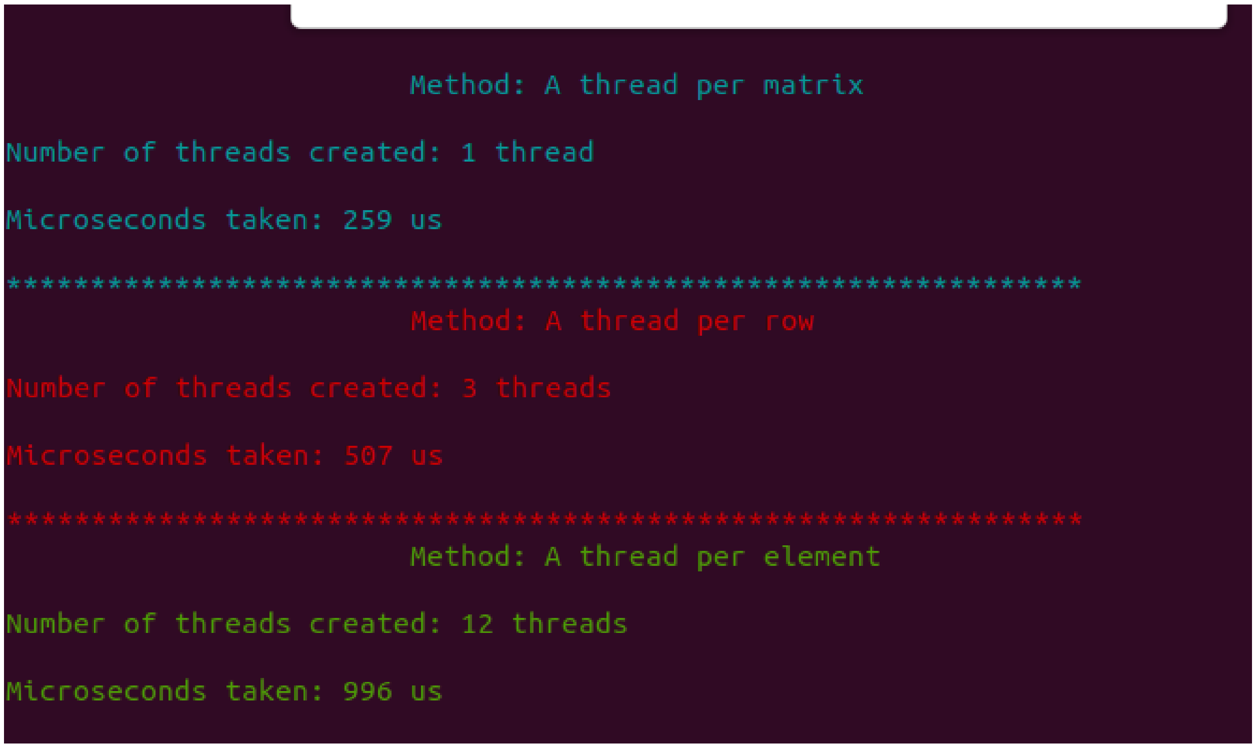
```
Method: A thread per row
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

The resule matrix using thread per element method:



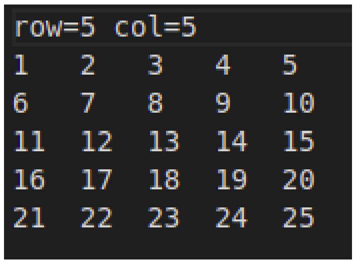
```
Method: A thread per element
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

The time taken by each method and the number of threads created for each:



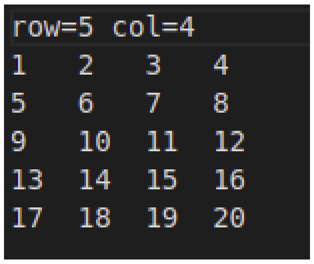
```
                        Method: A thread per matrix

Number of threads created: 1 thread

Microseconds taken: 259 us

****************************************************************
                        Method: A thread per row

Number of threads created: 3 threads

Microseconds taken: 507 us

****************************************************************
                        Method: A thread per element

Number of threads created: 12 threads

Microseconds taken: 996 us
```

- Test 3:
  The matrix A is:



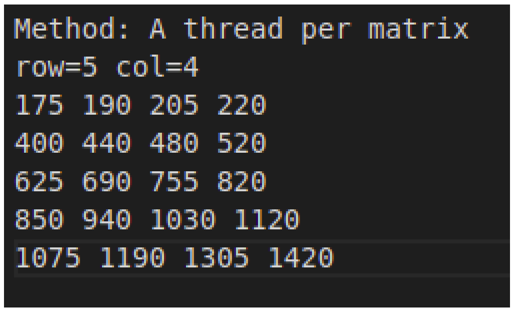
```
row=5 col=5
1   2   3   4   5
6   7   8   9   10
11  12  13  14  15
16  17  18  19  20
21  22  23  24  25
```

The matrix B is:



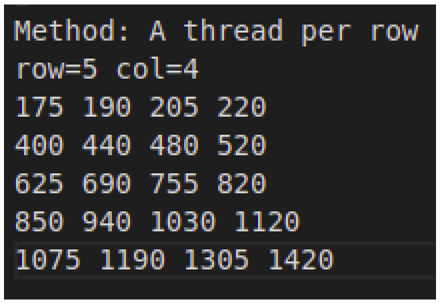
```
row=5 col=4
1   2   3   4
5   6   7   8
9   10  11  12
13  14  15  16
17  18  19  20
```

The result matrix using thread per matrix method:



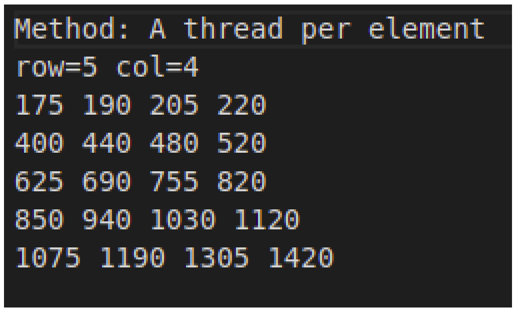
```
Method: A thread per matrix
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420
```

The result matrix using thread per row method:



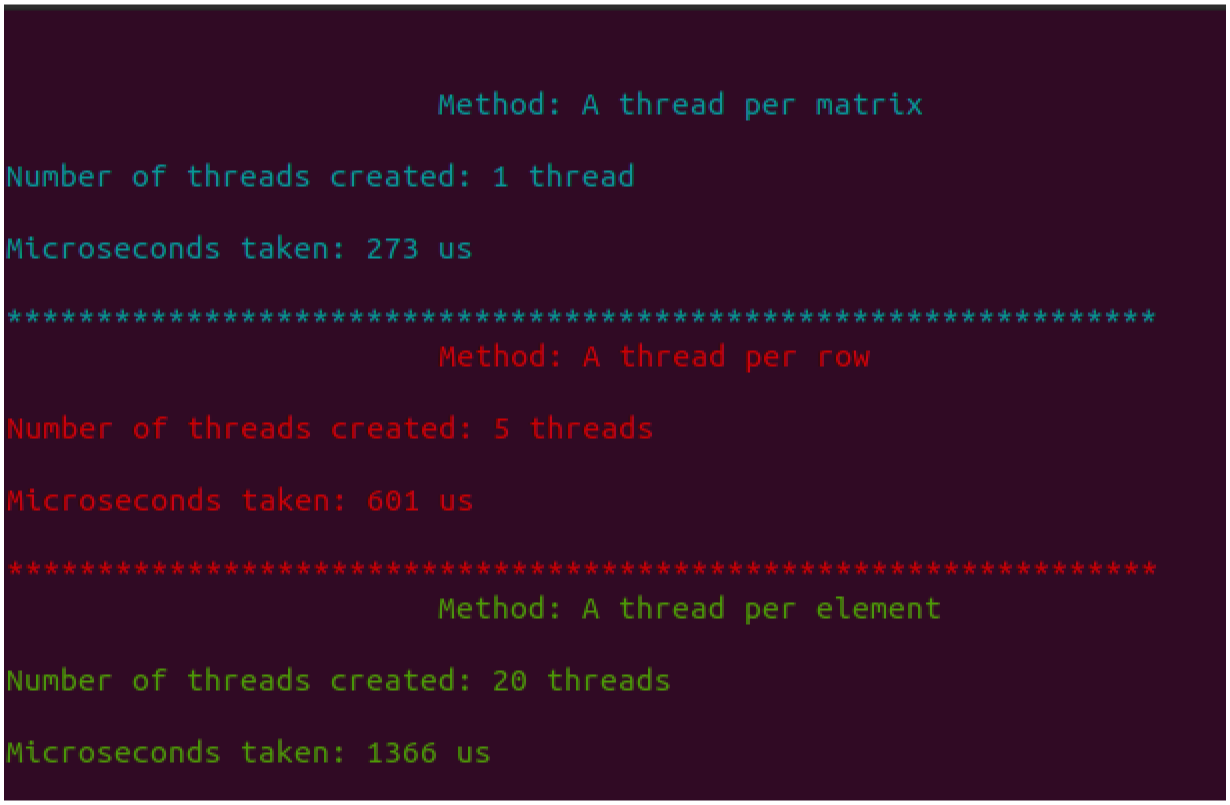
```
Method: A thread per row
row=5 col=4
175 190 205 220
400 440 480 520
625 690 755 820
850 940 1030 1120
1075 1190 1305 1420
```

The result matrix using thread per element method:



```
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
```

The time taken by each method and the number of threads created for each:



```
                        Method: A thread per matrix

Number of threads created: 1 thread

Microseconds taken: 273 us

*****************************************************************
                        Method: A thread per row

Number of threads created: 5 threads

Microseconds taken: 601 us

*****************************************************************
                        Method: A thread per element

Number of threads created: 20 threads

Microseconds taken: 1366 us
```

A comparison between the three types of matrix multiplication:

- There are many factors that will define the goodness of one method over the others like of course the number of cores the user has and other stuff. Because if the code is run on a uniprocessor environment then you'll find out that the threads will execute sequentially so it'll violate the advantages of using threads.
- But provided that the code is run on a multi-processor environment what is the difference then between the methods?
- By noticing the results provided in the Test cases section we'll find that in all the cases the first method(thread per matrix) is the faster than the second method (thread per row) which is faster than the third method (thread per element), and that's because threads require overhead time in creating and managing them so the more threads you have the more overhead you'll get.
- And we can tell that using the multi-threaded process in this context (matrix multiplication) is not the best solution and if you perform it in one take (single thread) it

would have been better because all the threads perform pretty much the same operation on different elements so it’s not the best idea to use threads in this context.

<u>Comparison considering the test cases:</u>

- For the first test case provided in the test cases section above the first method is faster than the second (10 threads) and the slowest was the third method (100 threads) and that’s because of the overhead required to create and handle the threads.
- For the second test case, still the first method is the fastest but for the second(3 threads)and third(12 threads)the number of threads created is near so the time taken by them is also near (still the second faster)
- For the third test case, it’s the same observation as test case one.

So, to cut it short, using threads is great but when the context of use is suitable and when the number of threads created goes along with the number of cores you have to get the best out of parallelism.