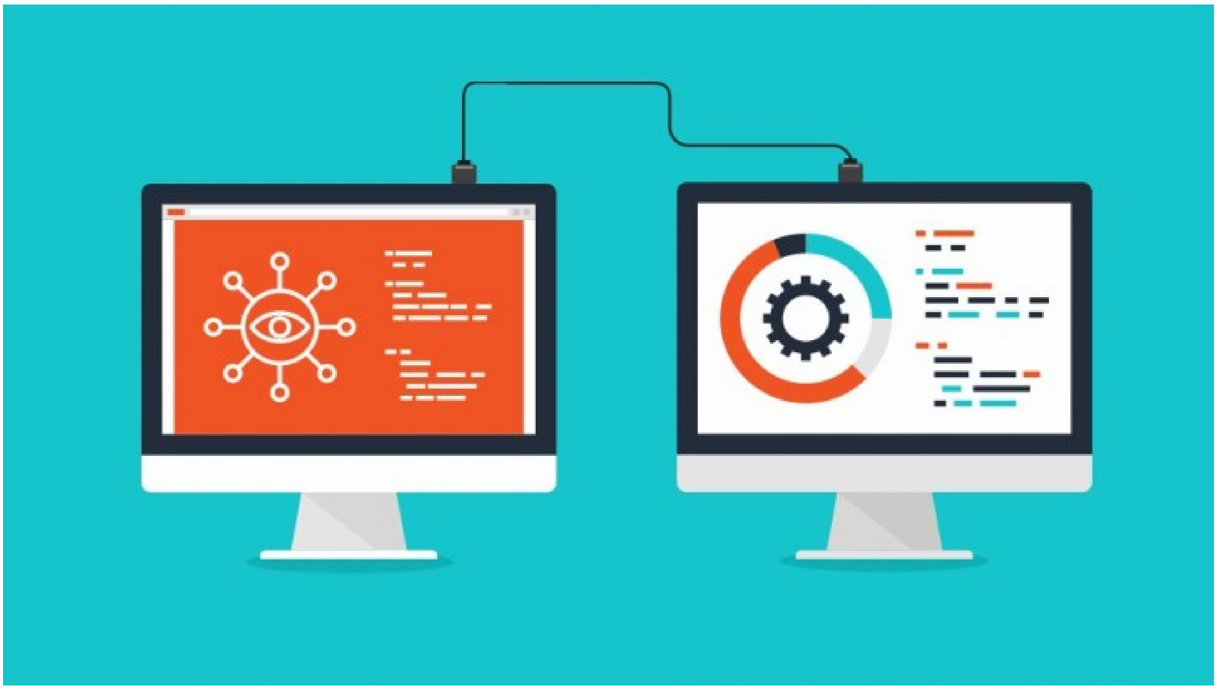
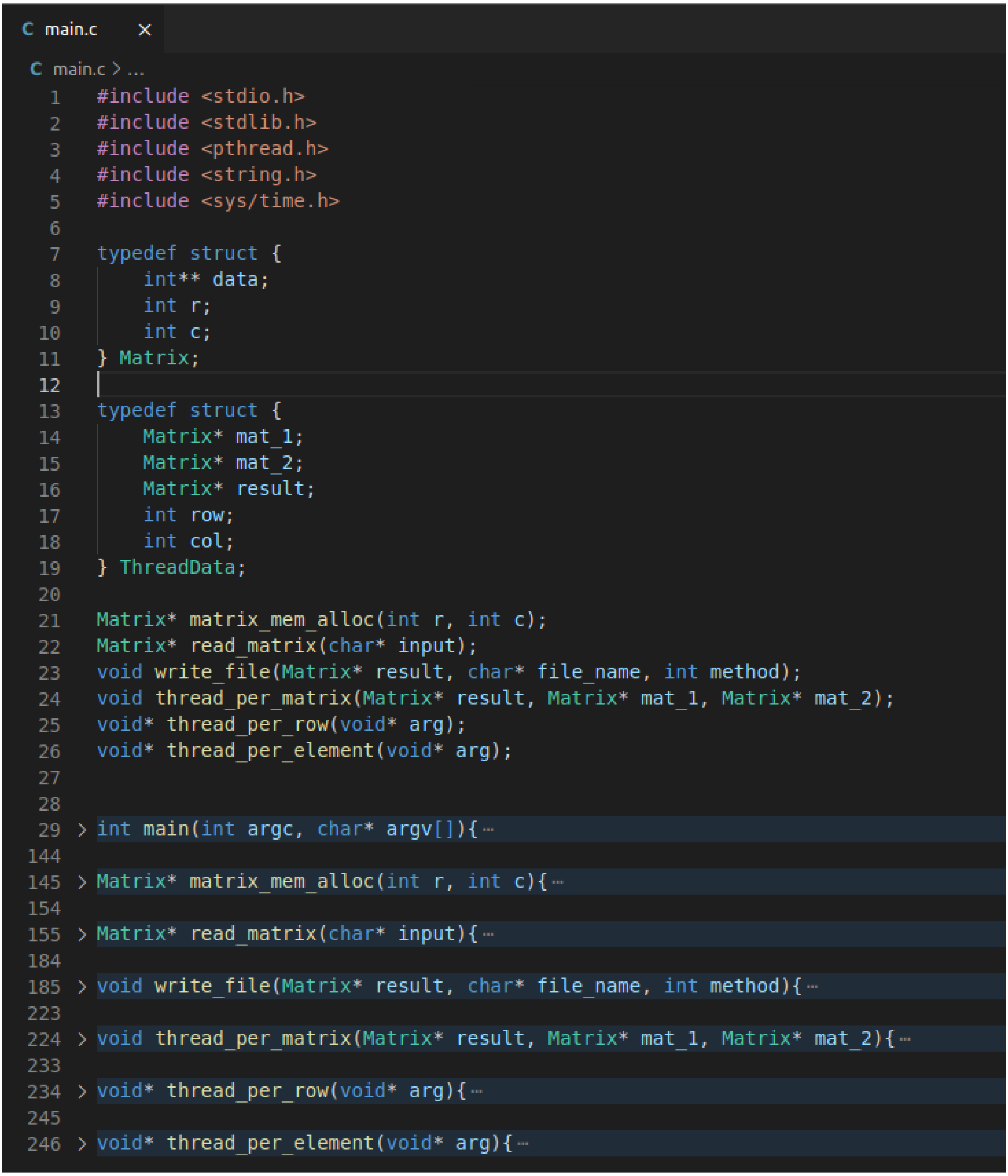
Lab 2

Matrix Multiplication (Multi-Threading)



Name : Pancee Wahid
ID: 18010467

Code Overview



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <sys/time.h>

typedef struct {
    int** data;
    int r;
    int c;
} Matrix;

typedef struct {
    Matrix* mat_1;
    Matrix* mat_2;
    Matrix* result;
    int row;
    int col;
} ThreadData;

Matrix* matrix_mem_alloc(int r, int c);
Matrix* read_matrix(char* input);
void write_file(Matrix* result, char* file_name, int method);
void thread_per_matrix(Matrix* result, Matrix* mat_1, Matrix* mat_2);
void* thread_per_row(void* arg);
void* thread_per_element(void* arg);


int main(int argc, char* argv[]){…

Matrix* matrix_mem_alloc(int r, int c){…

Matrix* read_matrix(char* input){…

void write_file(Matrix* result, char* file_name, int method){…

void thread_per_matrix(Matrix* result, Matrix* mat_1, Matrix* mat_2){…

void* thread_per_row(void* arg){…

void* thread_per_element(void* arg){…
```

Structs:

→ Matrix

→ data: a double pointer to the matrix

→ r: number of columns of the matrix   → c: number of rows of the matrix

→ThreadData

→mat_1: the first matrix double pointer   →mat_2: the second matrix double pointer

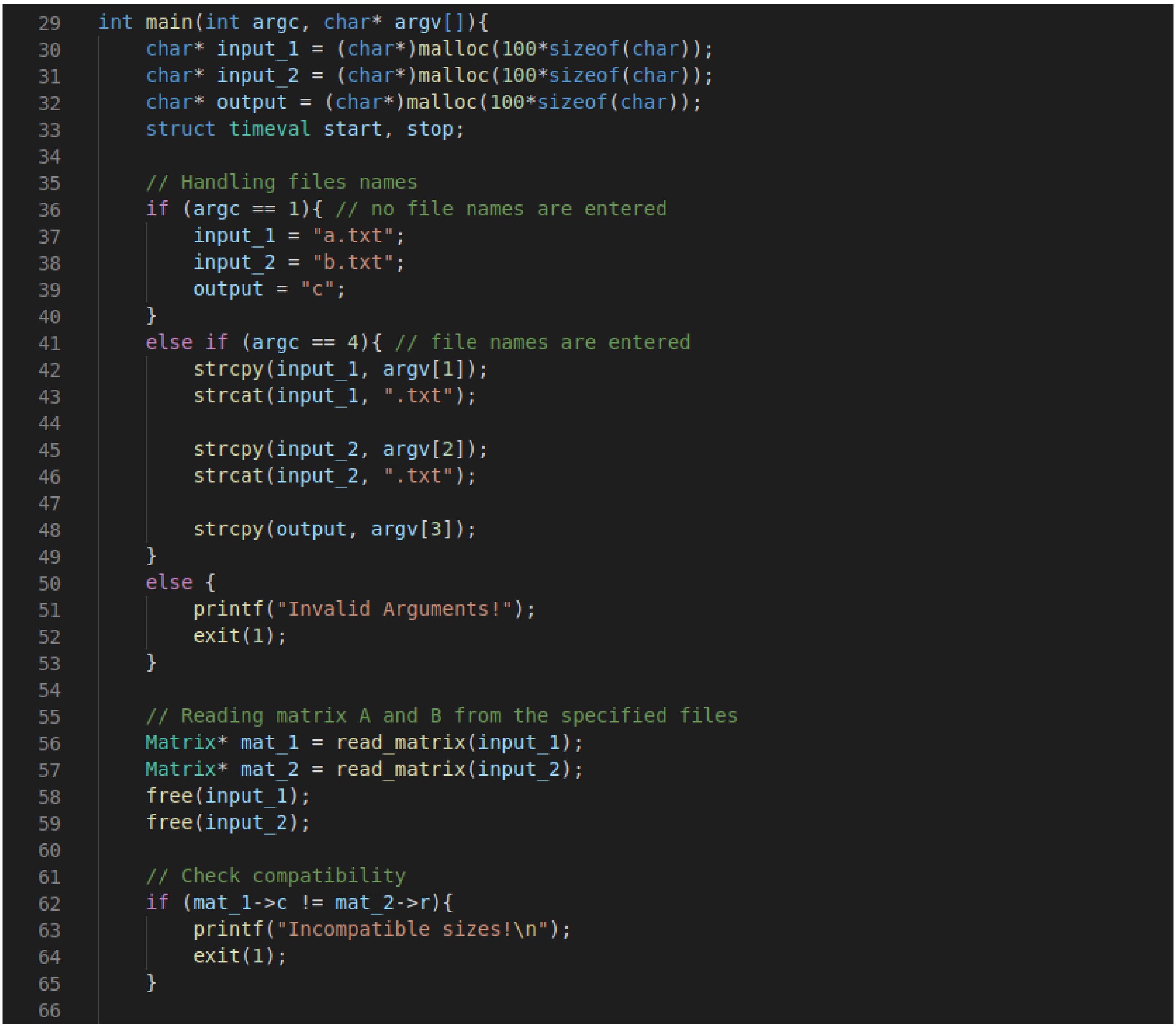
→result: double pointer to the matrix to store the result in

→row: row to be used (used in thread per row and thread per element methods)

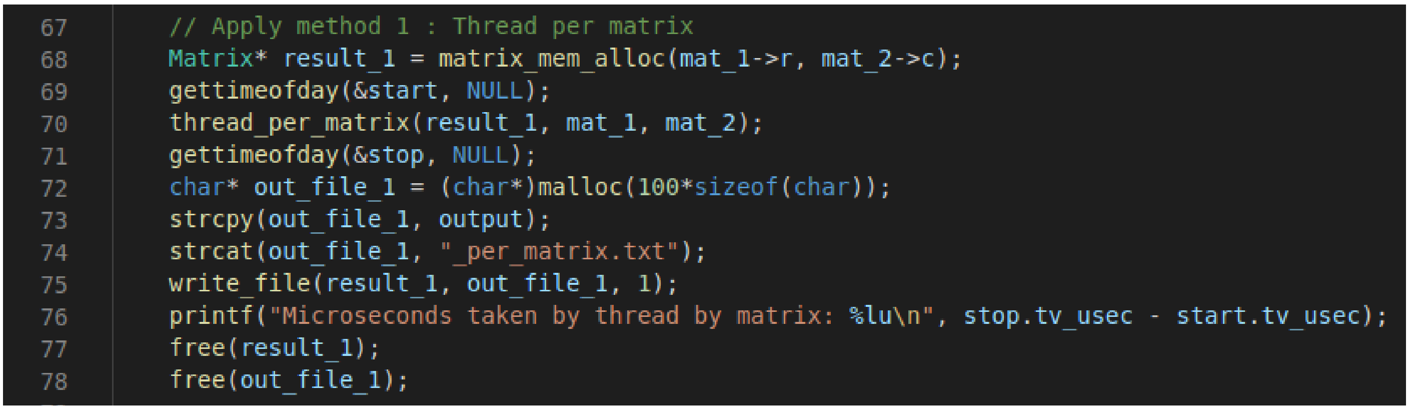
→col: col to be used (used in thread per element method)

Functions

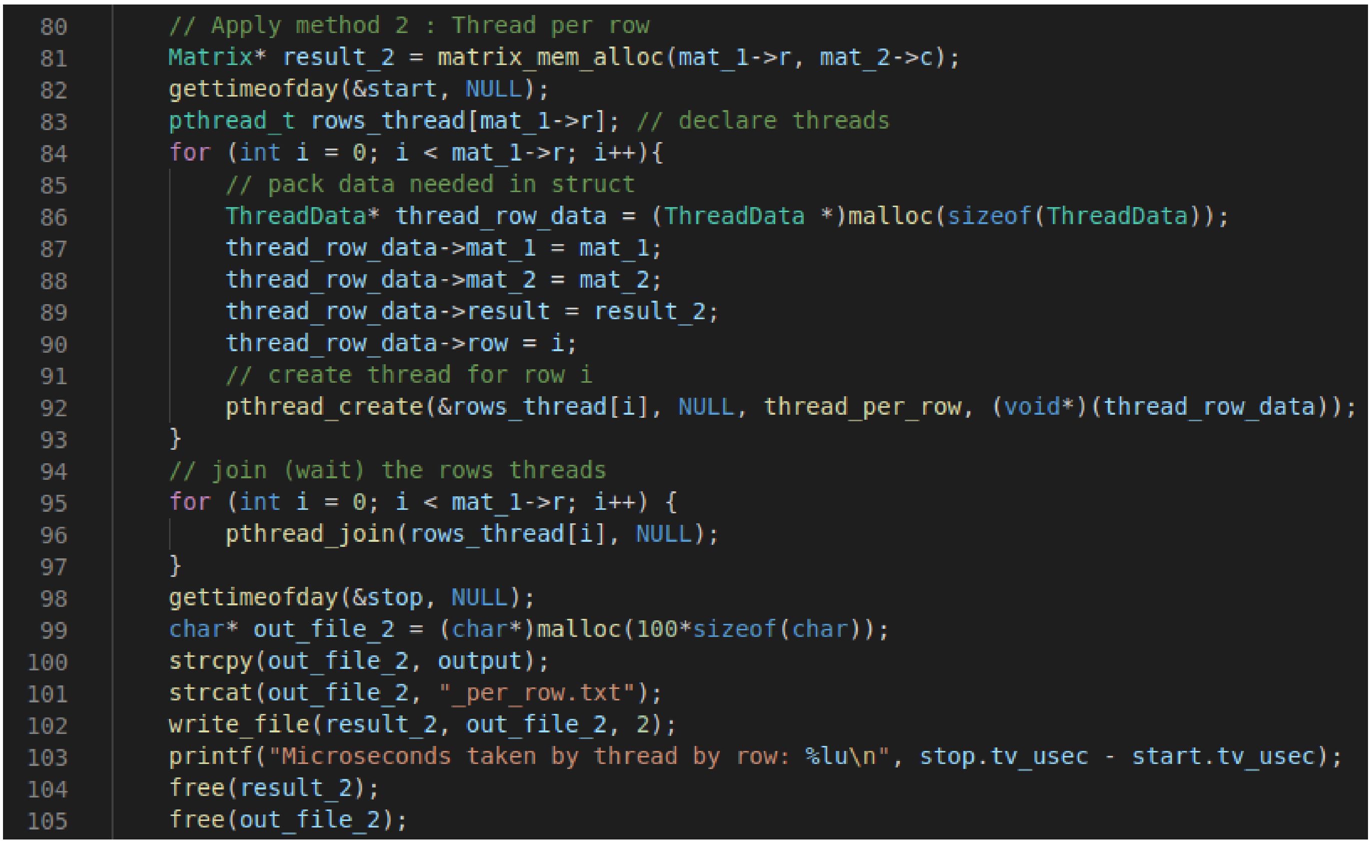
Main Function



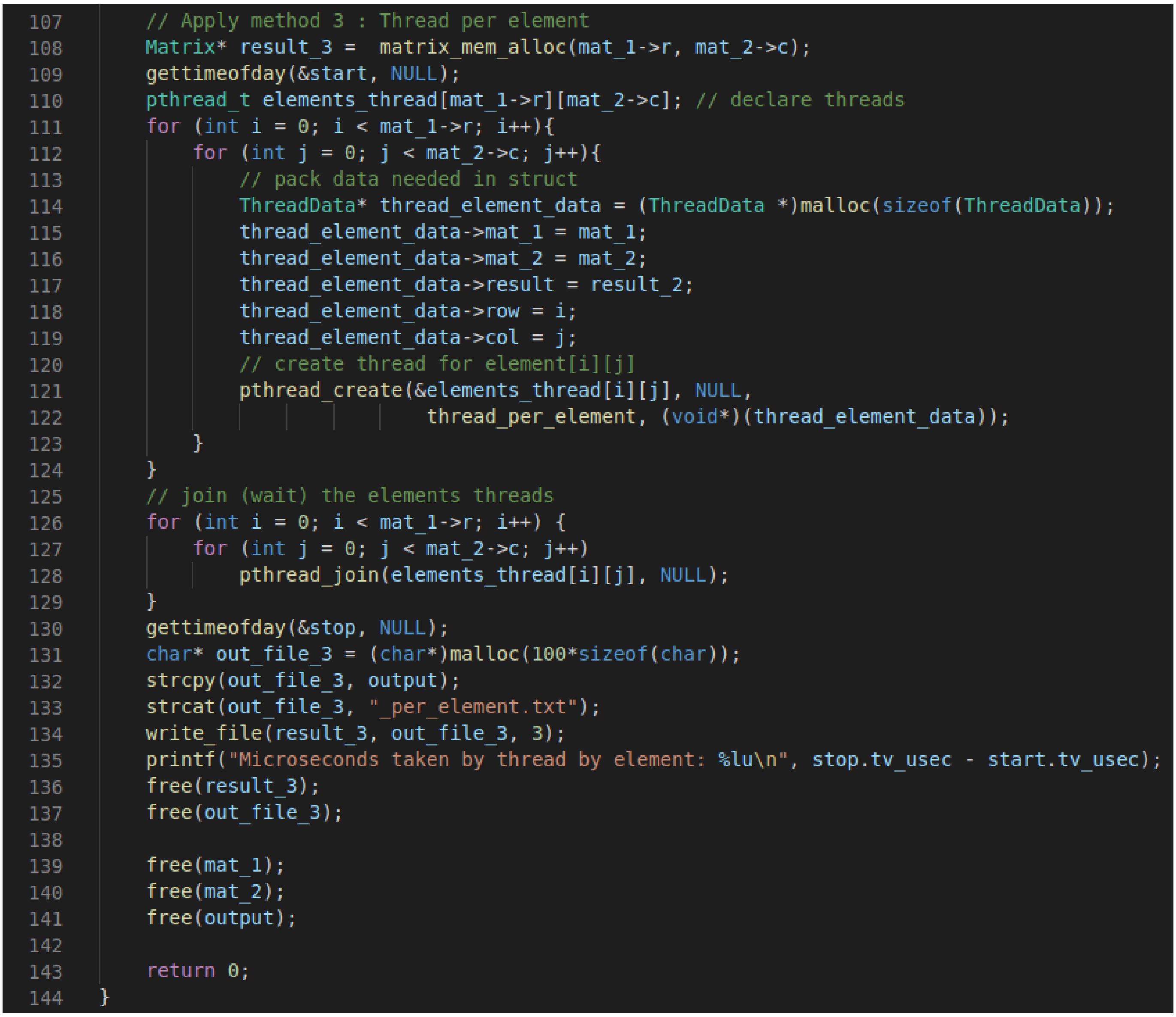
```c
int main(int argc, char* argv[]){
    char* input_1 = (char*)malloc(100*sizeof(char));
    char* input_2 = (char*)malloc(100*sizeof(char));
    char* output = (char*)malloc(100*sizeof(char));
    struct timeval start, stop;

    // Handling files names
    if (argc == 1){ // no file names are entered
        input_1 = "a.txt";
        input_2 = "b.txt";
        output = "c";
    }
    else if (argc == 4){ // file names are entered
        strcpy(input_1, argv[1]);
        strcat(input_1, ".txt");

        strcpy(input_2, argv[2]);
        strcat(input_2, ".txt");

        strcpy(output, argv[3]);
    }
    else {
        printf("Invalid Arguments!");
        exit(1);
    }

    // Reading matrix A and B from the specified files
    Matrix* mat_1 = read_matrix(input_1);
    Matrix* mat_2 = read_matrix(input_2);
    free(input_1);
    free(input_2);

    // Check compatibility
    if (mat_1->c != mat_2->r){
        printf("Incompatible sizes!\n");
        exit(1);
    }

```



```c
    // Apply method 1 : Thread per matrix
    Matrix* result_1 = matrix_mem_alloc(mat_1->r, mat_2->c);
    gettimeofday(&start, NULL);
    thread_per_matrix(result_1, mat_1, mat_2);
    gettimeofday(&stop, NULL);
    char* out_file_1 = (char*)malloc(100*sizeof(char));
    strcpy(out_file_1, output);
    strcat(out_file_1, "_per_matrix.txt");
    write_file(result_1, out_file_1, 1);
    printf("Microseconds taken by thread by matrix: %lu\n", stop.tv_usec - start.tv_usec);
    free(result_1);
    free(out_file_1);
```



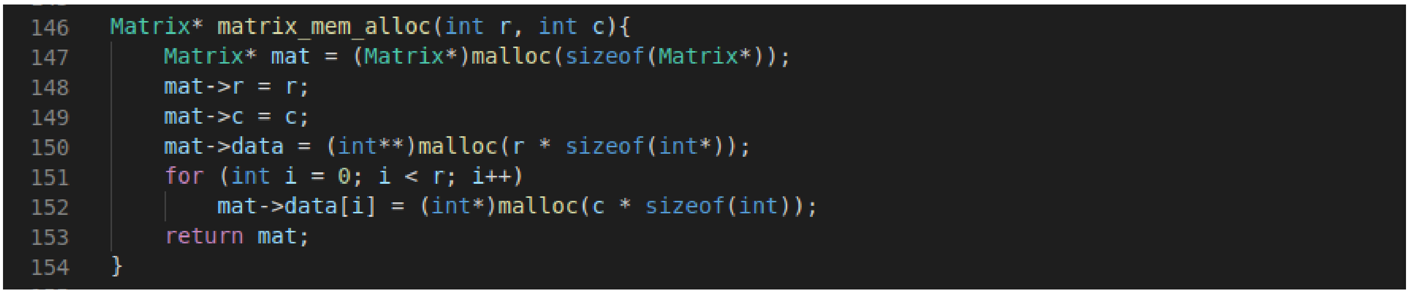
```c
    // Apply method 2 : Thread per row
    Matrix* result_2 = matrix_mem_alloc(mat_1->r, mat_2->c);
    gettimeofday(&start, NULL);
    pthread_t rows_thread[mat_1->r]; // declare threads
    for (int i = 0; i < mat_1->r; i++){
        // pack data needed in struct
        ThreadData* thread_row_data = (ThreadData *)malloc(sizeof(ThreadData));
        thread_row_data->mat_1 = mat_1;
        thread_row_data->mat_2 = mat_2;
        thread_row_data->result = result_2;
        thread_row_data->row = i;
        // create thread for row i
        pthread_create(&rows_thread[i], NULL, thread_per_row, (void*)(thread_row_data));
    }
    // join (wait) the rows threads
    for (int i = 0; i < mat_1->r; i++) {
        pthread_join(rows_thread[i], NULL);
    }
    gettimeofday(&stop, NULL);
    char* out_file_2 = (char*)malloc(100*sizeof(char));
    strcpy(out_file_2, output);
    strcat(out_file_2, "_per_row.txt");
    write_file(result_2, out_file_2, 2);
    printf("Microseconds taken by thread by row: %lu\n", stop.tv_usec - start.tv_usec);
    free(result_2);
    free(out_file_2);
```



```c
    // Apply method 3 : Thread per element
    Matrix* result_3 =  matrix_mem_alloc(mat_1->r, mat_2->c);
    gettimeofday(&start, NULL);
    pthread_t elements_thread[mat_1->r][mat_2->c]; // declare threads
    for (int i = 0; i < mat_1->r; i++){
        for (int j = 0; j < mat_2->c; j++){
            // pack data needed in struct
            ThreadData* thread_element_data = (ThreadData *)malloc(sizeof(ThreadData));
            thread_element_data->mat_1 = mat_1;
            thread_element_data->mat_2 = mat_2;
            thread_element_data->result = result_2;
            thread_element_data->row = i;
            thread_element_data->col = j;
            // create thread for element[i][j]
            pthread_create(&elements_thread[i][j], NULL,
                            thread_per_element, (void*)(thread_element_data));
        }
    }
    // join (wait) the elements threads
    for (int i = 0; i < mat_1->r; i++) {
        for (int j = 0; j < mat_2->c; j++)
            pthread_join(elements_thread[i][j], NULL);
    }
    gettimeofday(&stop, NULL);
    char* out_file_3 = (char*)malloc(100*sizeof(char));
    strcpy(out_file_3, output);
    strcat(out_file_3, "_per_element.txt");
    write_file(result_3, out_file_3, 3);
    printf("Microseconds taken by thread by element: %lu\n", stop.tv_usec - start.tv_usec);
    free(result_3);
    free(out_file_3);

    free(mat_1);
    free(mat_2);
    free(output);

    return 0;
}
```

matrix_mem_alloc():

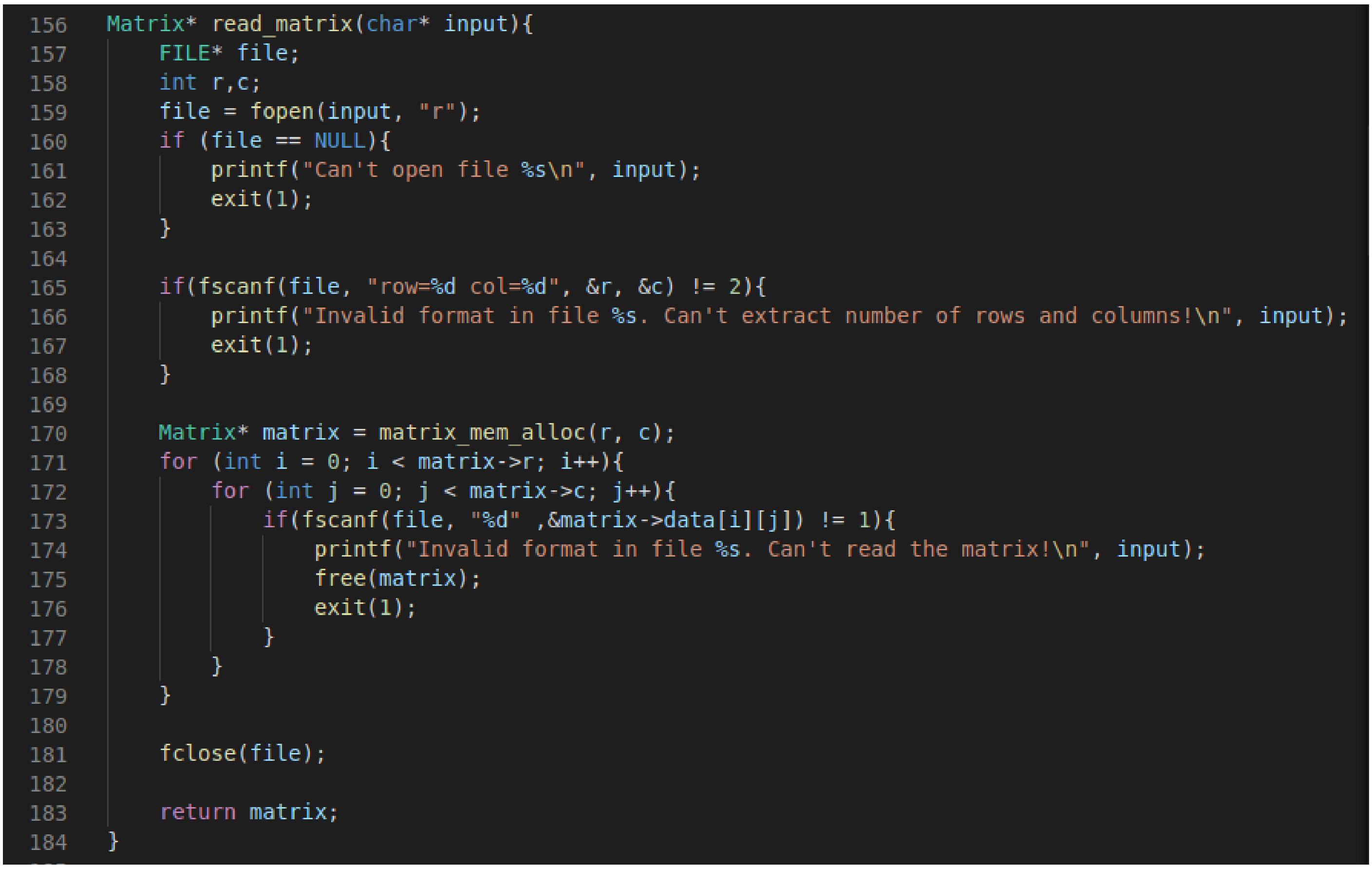
It takes the number of rows and columns of the matrix and allocates memory for them then returns a pointer to the struct Matrix created for the matrix holding a double pointer to it, its number of rows and its number of columns.



```c
Matrix* matrix_mem_alloc(int r, int c){
    Matrix* mat = (Matrix*)malloc(sizeof(Matrix*));
    mat->r = r;
    mat->c = c;
    mat->data = (int**)malloc(r * sizeof(int*));
    for (int i = 0; i < r; i++)
        mat->data[i] = (int*)malloc(c * sizeof(int));
    return mat;
}
```

read_matrix():

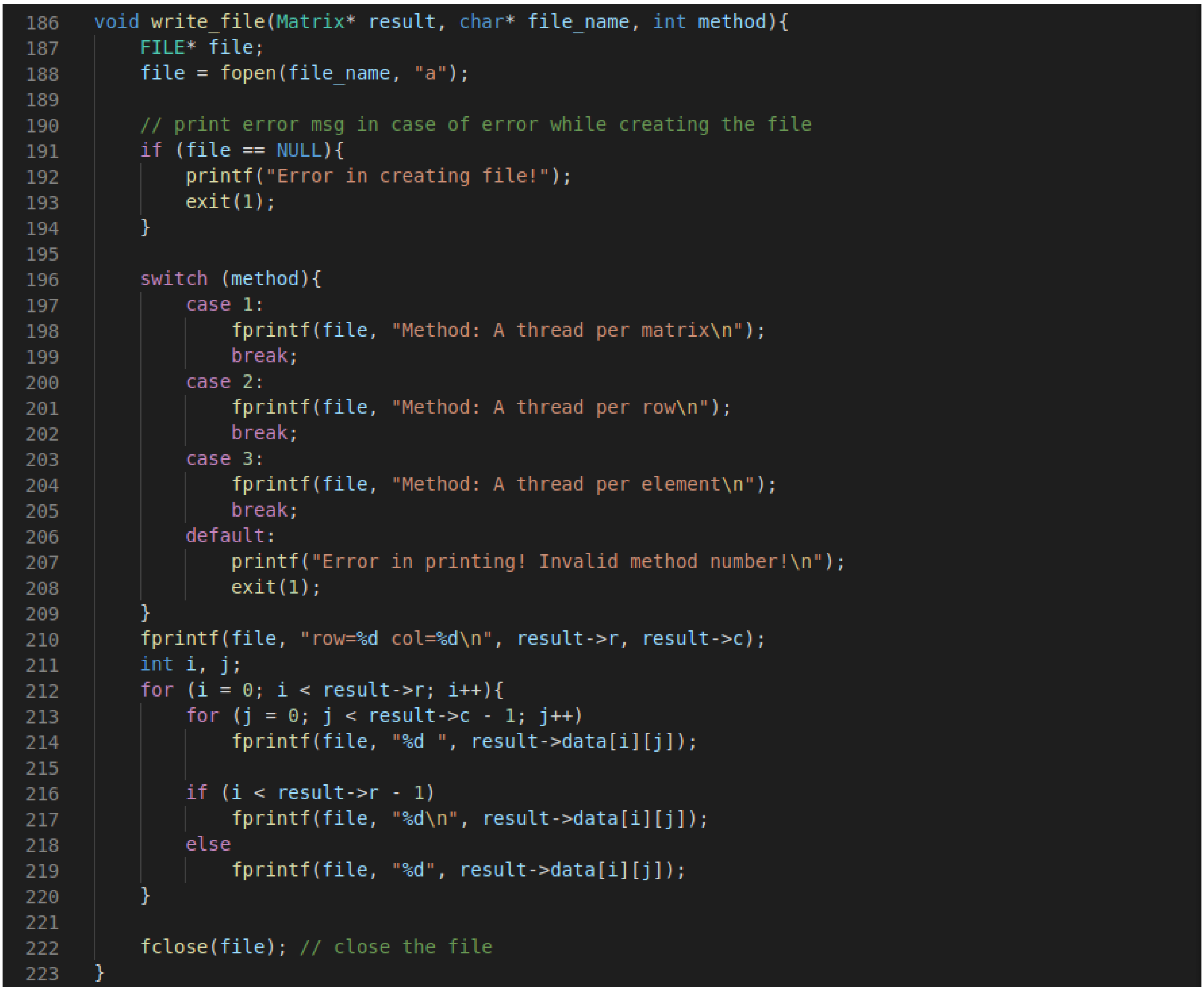
It takes the name of the file that contains the matrix to be read, reads its number of rows and columns and calls matrix_mem_alloc() to allocate memory for the read matrix. Then it reads the matrix elements storing them in their locations and returns a pointer to the Matrix created.



```c
Matrix* read_matrix(char* input){
    FILE* file;
    int r,c;
    file = fopen(input, "r");
    if (file == NULL){
        printf("Can't open file %s\n", input);
        exit(1);
    }

    if(fscanf(file, "row=%d col=%d", &r, &c) != 2){
        printf("Invalid format in file %s. Can't extract number of rows and columns!\n", input);
        exit(1);
    }

    Matrix* matrix = matrix_mem_alloc(r, c);
    for (int i = 0; i < matrix->r; i++){
        for (int j = 0; j < matrix->c; j++){
            if(fscanf(file, "%d" ,&matrix->data[i][j]) != 1){
                printf("Invalid format in file %s. Can't read the matrix!\n", input);
                free(matrix);
                exit(1);
            }
        }
    }

    fclose(file);

    return matrix;
}
```

write_file():

It takes the required data to store the resulting matrix from multiplication of the two read matrices in the specified file.



```c
void write_file(Matrix* result, char* file_name, int method){
    FILE* file;
    file = fopen(file_name, "a");

    // print error msg in case of error while creating the file
    if (file == NULL){
        printf("Error in creating file!");
        exit(1);
    }

    switch (method){
        case 1:
            fprintf(file, "Method: A thread per matrix\n");
            break;
        case 2:
            fprintf(file, "Method: A thread per row\n");
            break;
        case 3:
            fprintf(file, "Method: A thread per element\n");
            break;
        default:
            printf("Error in printing! Invalid method number!\n");
            exit(1);
    }
    fprintf(file, "row=%d col=%d\n", result->r, result->c);
    int i, j;
    for (i = 0; i < result->r; i++){
        for (j = 0; j < result->c - 1; j++)
            fprintf(file, "%d ", result->data[i][j]);

        if (i < result->r - 1)
            fprintf(file, "%d\n", result->data[i][j]);
        else
            fprintf(file, "%d", result->data[i][j]);
    }

    fclose(file); // close the file
}
```

thread_per_matrix():

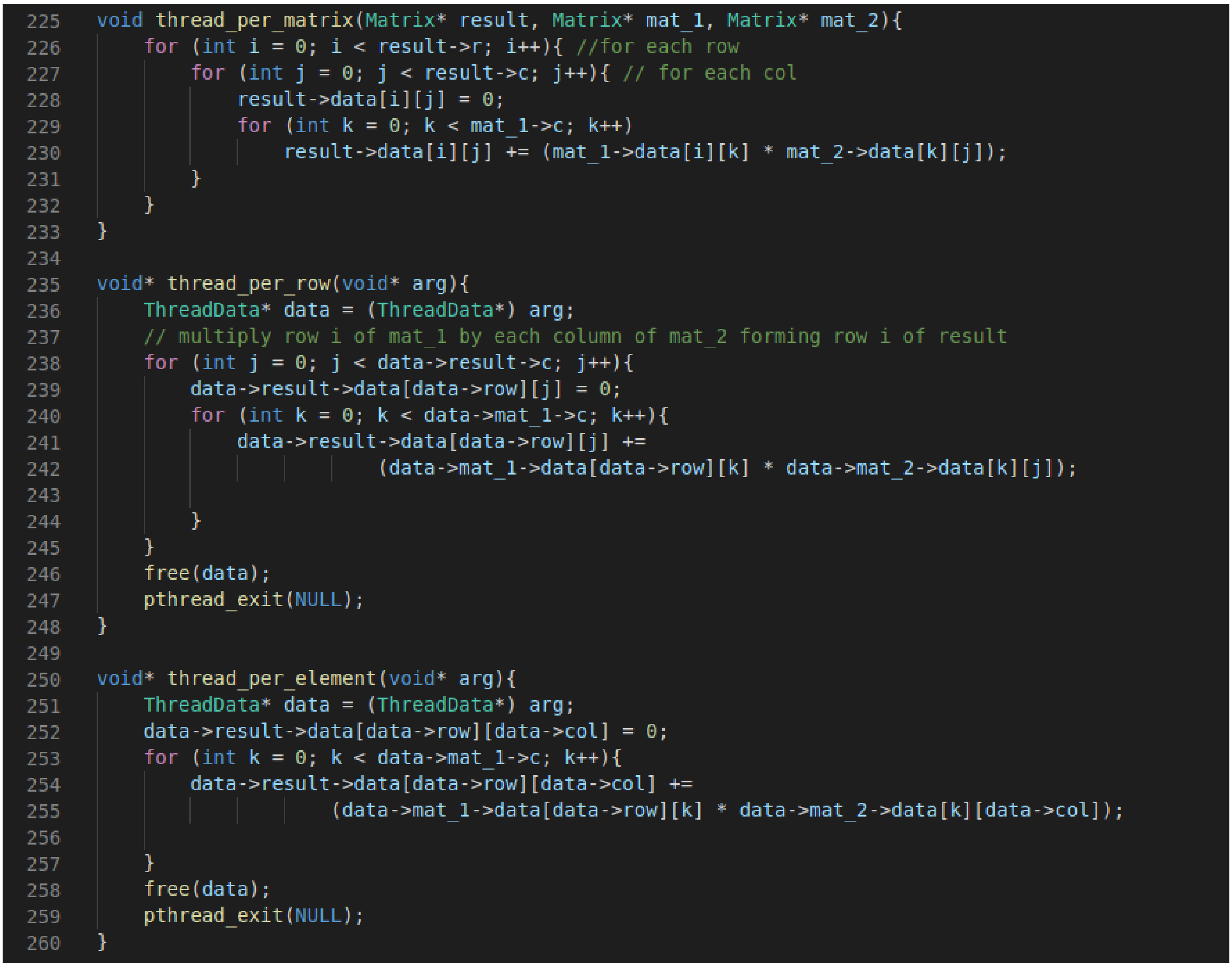
It calculates the product **matrix** of multiplying matrix 1 by matrix 2 normally by multiplying every row in matrix 1 by every column in matrix2.

thread_per_row():

It calculates the specified **row** of the product matrix by multiplying the row with the same index from matrix 1 with every column of matrix 2 storing the results in the result matrix.

thread_per_element():

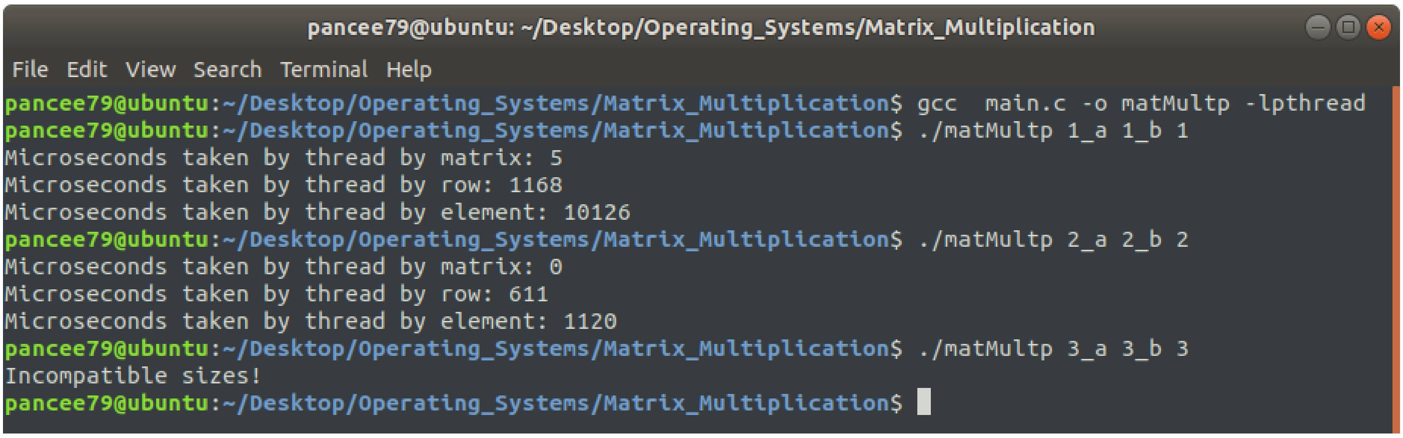
It calculates the specified **element** of the product matrix by multiplying the row with the same index from matrix 1 with the column of the same index of matrix 2 storing the results in the result matrix.



```c
void thread_per_matrix(Matrix* result, Matrix* mat_1, Matrix* mat_2){
    for (int i = 0; i < result->r; i++){ //for each row
        for (int j = 0; j < result->c; j++){ // for each col
            result->data[i][j] = 0;
            for (int k = 0; k < mat_1->c; k++)
                result->data[i][j] += (mat_1->data[i][k] * mat_2->data[k][j]);
        }
    }
}

void* thread_per_row(void* arg){
    ThreadData* data = (ThreadData*) arg;
    // multiply row i of mat_1 by each column of mat_2 forming row i of result
    for (int j = 0; j < data->result->c; j++){
        data->result->data[data->row][j] = 0;
        for (int k = 0; k < data->mat_1->c; k++){
            data->result->data[data->row][j] +=
                        (data->mat_1->data[data->row][k] * data->mat_2->data[k][j]);

        }
    }
    free(data);
    pthread_exit(NULL);
}

void* thread_per_element(void* arg){
    ThreadData* data = (ThreadData*) arg;
    data->result->data[data->row][data->col] = 0;
    for (int k = 0; k < data->mat_1->c; k++){
        data->result->data[data->row][data->col] +=
                    (data->mat_1->data[data->row][k] * data->mat_2->data[k][data->col]);

    }
    free(data);
    pthread_exit(NULL);
}
```

Sample Runs

The given 3 test cases give the following output:

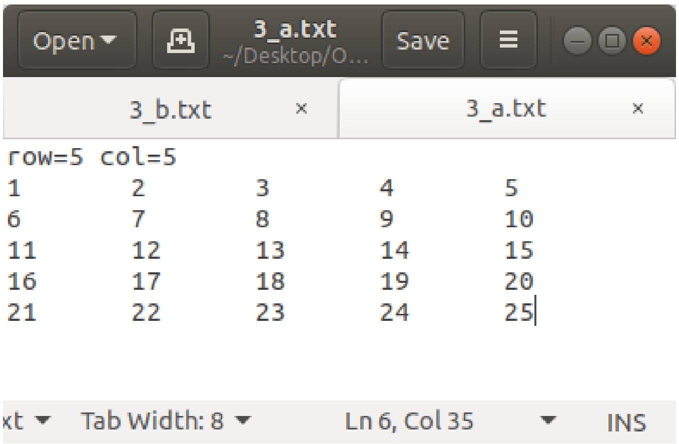


pancee79@ubuntu: ~/Desktop/Operating_Systems/Matrix_Multiplication
File Edit View Search Terminal Help

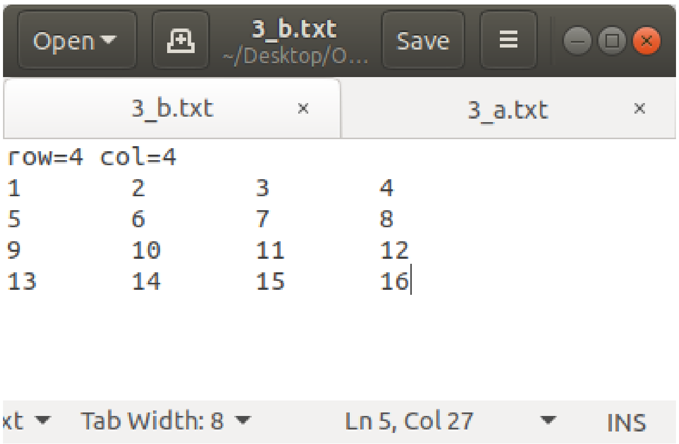
```
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ gcc  main.c -o matMultp -lpthread
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp 1_a 1_b 1
Microseconds taken by thread by matrix: 5
Microseconds taken by thread by row: 1168
Microseconds taken by thread by element: 10126
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp 2_a 2_b 2
Microseconds taken by thread by matrix: 0
Microseconds taken by thread by row: 611
Microseconds taken by thread by element: 1120
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp 3_a 3_b 3
Incompatible sizes!
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ 
```

Test case 3

Input matrices



```
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
```



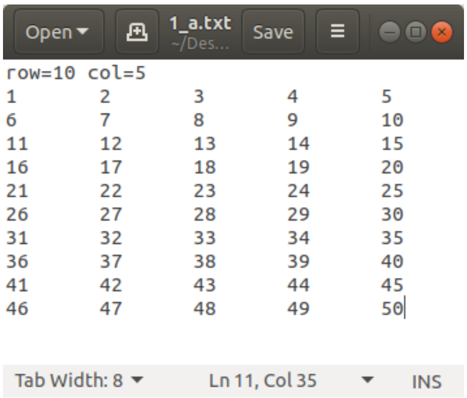
```
row=4 col=4
1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16
```

Since these matrices can't be multiplied → as they don't follow the rule that if size of A is nxm, then size of B should be mxk (number of columns of first matrix ≠ number of rows of second matrix)

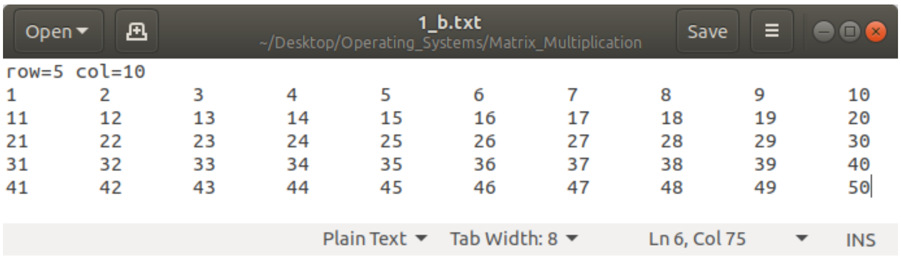
Therefore, the program printed an error message that size is incompatible and terminated.

Test case 1

Input matrices

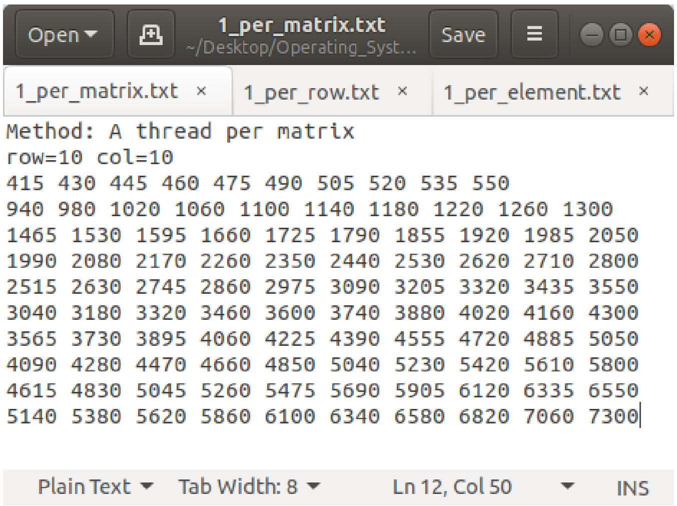


```
row=10 col=5
1       2       3       4       5
6       7       8       9       10
11      12      13      14      15
16      17      18      19      20
21      22      23      24      25
26      27      28      29      30
31      32      33      34      35
36      37      38      39      40
41  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

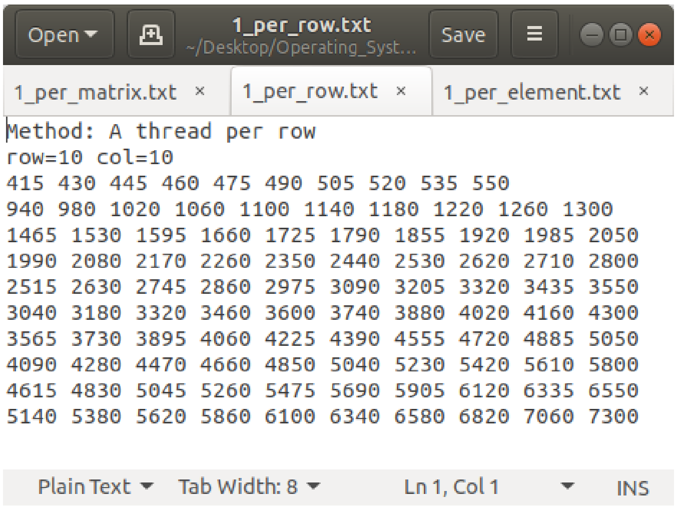


```
row=5 col=10
1       2       3       4       5       6       7       8       9       10
11      12      13      14      15      16      17      18      19      20
21      22      23      24      25      26      27      28      29      30
31      32      33      34      35      36      37      38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

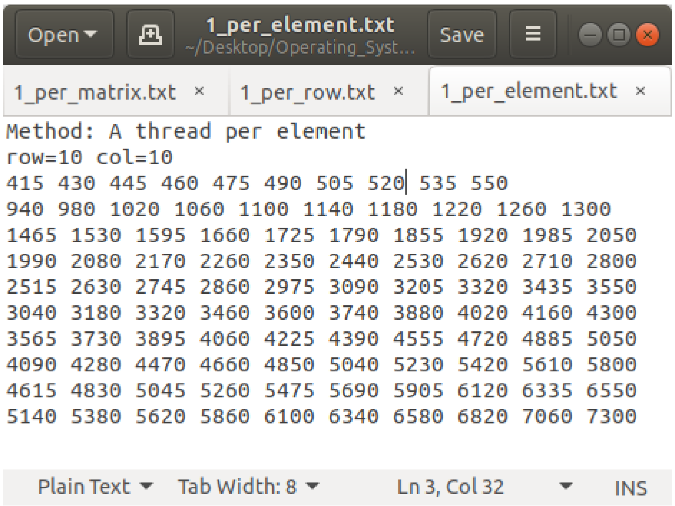
Output matrix



```
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
```



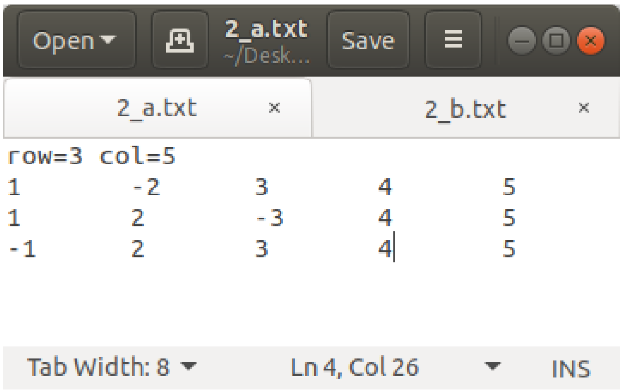
```
Method: A thread per row
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460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



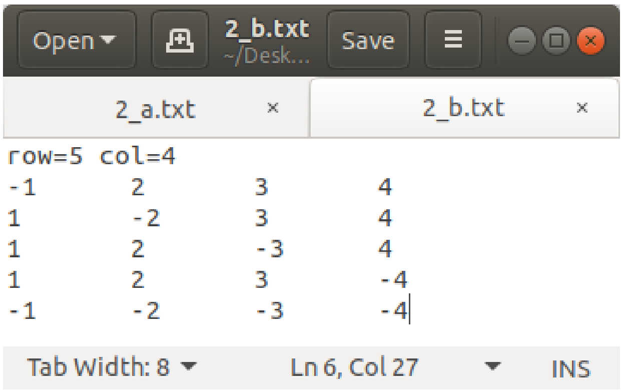
```
Method: A thread per element
row=10 col=10
415 430 445 460 475 490 505 520 535 550
940 980 1020 1060 1100 1140 1180 1220 1260 1300
1465 1530 1595 1660 1725 1790 1855 1920 1985 2050
1990 2080 2170 2260 2350 2440 2530 2620 2710 2800
2515 2630 2745 2860 2975 3090 3205 3320 3435 3550
3040 3180 3320 3460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

Test case 2

Input matrices

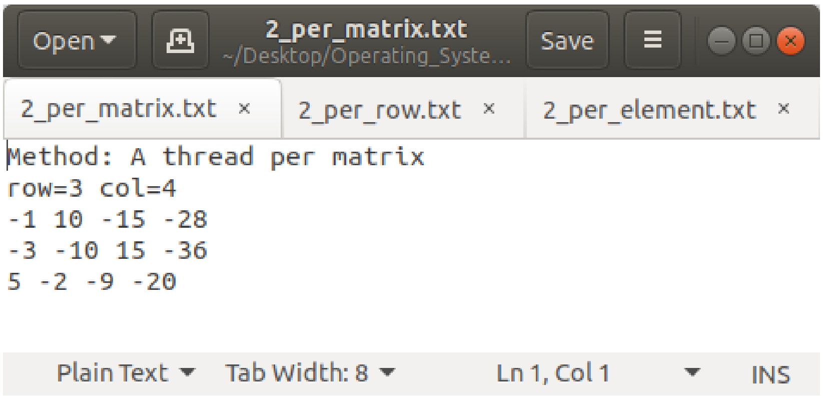


```
row=3 col=5
1       -2      3       4       5
1       2       -3      4       5
-1      2       3       4       5
```

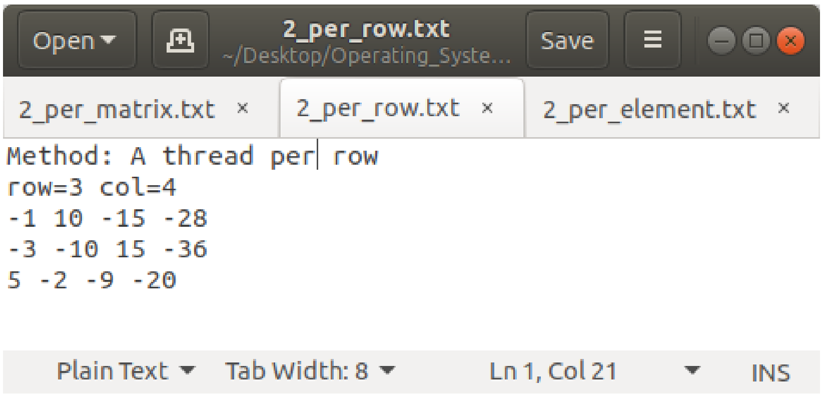


```
row=5 col=4
-1      2       3       4
1       -2      3       4
1       2       -3      4
1       2       3       -4
-1      -2      -3      -4
```

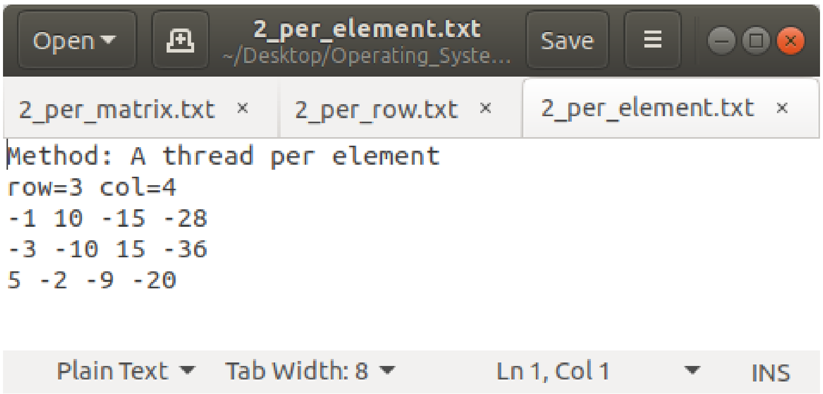
Output matrix



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

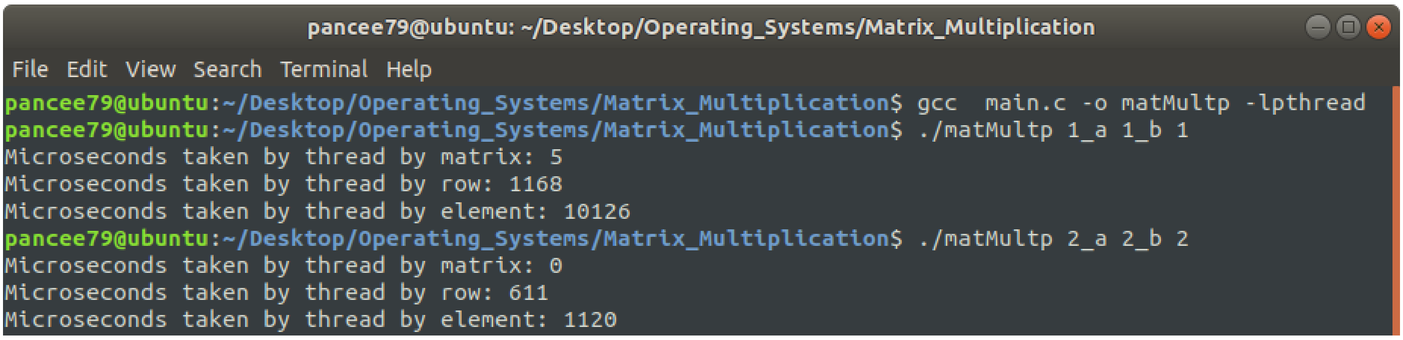


```
Method: A thread per row
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```



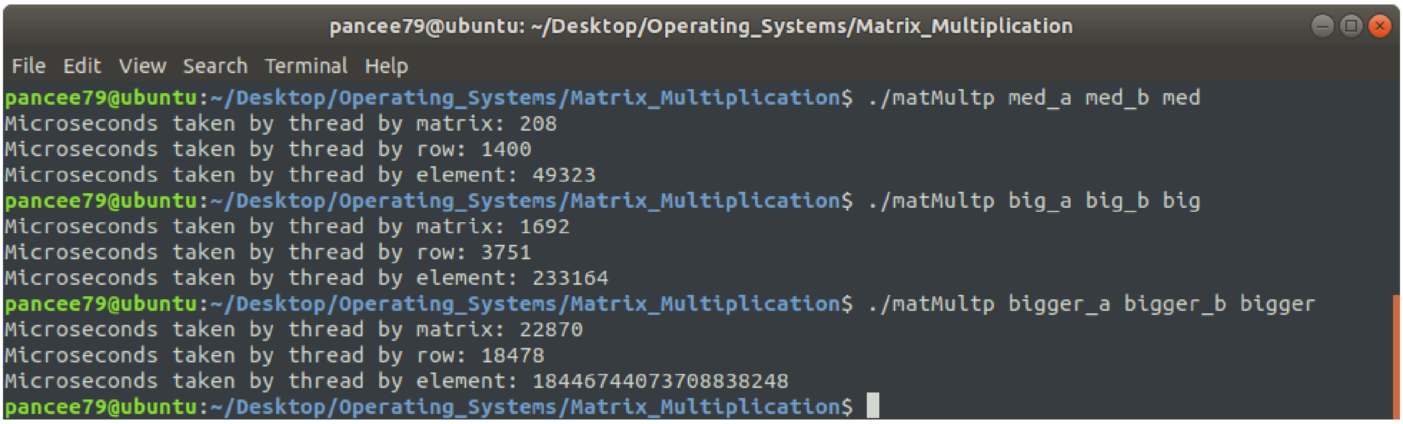
```
Method: A thread per element
row=3 col=4
-1 10 -15 -28
-3 -10 15 -36
5 -2 -9 -20
```

Comparison



pancee79@ubuntu: ~/Desktop/Operating_Systems/Matrix_Multiplication
File Edit View Search Terminal Help

```
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ gcc  main.c -o matMultp -lpthread
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp 1_a 1_b 1
Microseconds taken by thread by matrix: 5
Microseconds taken by thread by row: 1168
Microseconds taken by thread by element: 10126
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp 2_a 2_b 2
Microseconds taken by thread by matrix: 0
Microseconds taken by thread by row: 611
Microseconds taken by thread by element: 1120
```



pancee79@ubuntu: ~/Desktop/Operating_Systems/Matrix_Multiplication
File Edit View Search Terminal Help

```
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp med_a med_b med
Microseconds taken by thread by matrix: 208
Microseconds taken by thread by row: 1400
Microseconds taken by thread by element: 49323
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp big_a big_b big
Microseconds taken by thread by matrix: 1692
Microseconds taken by thread by row: 3751
Microseconds taken by thread by element: 233164
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ ./matMultp bigger_a bigger_b bigger
Microseconds taken by thread by matrix: 22870
Microseconds taken by thread by row: 18478
Microseconds taken by thread by element: 18446744073708838248
pancee79@ubuntu:~/Desktop/Operating_Systems/Matrix_Multiplication$ 
```

Note:

- The used matrices for med_a and med_b were of size (20x30) and (30x50).
- The used matrices for big_a and big_b were of size (40x60) and (60x100).
- The used matrices for bigger_a and bigger_b were of size (150x150) and (150x150).

Can be found here.

Observation:

- For the test cases shown before, med and big matrices, performing matrix multiplication with one thread for the whole matrix was faster than the two other methods.
- For bigger matrices, performing matrix multiplication with one thread per row was better than the two other methods.
- The method of creating thread to calculate each element was bad in small matrices and worst in bigger ones.

Conclusion:

After running the three methods: thread per matrix, thread per row and thread per element on matrices of different sizes, it was deduced that:

- For small matrices (generally not too many computations), it's more efficient to use a single thread to avoid the overhead of creating threads which will waste much time than performing the computations using a single thread.
- For large matrices (generally heavy computations), it's more efficient to distribute the work on a reasonable number of threads which will improve the execution time of the program. However, creating so many threads may be more waste of time and resources than not using threads at all.