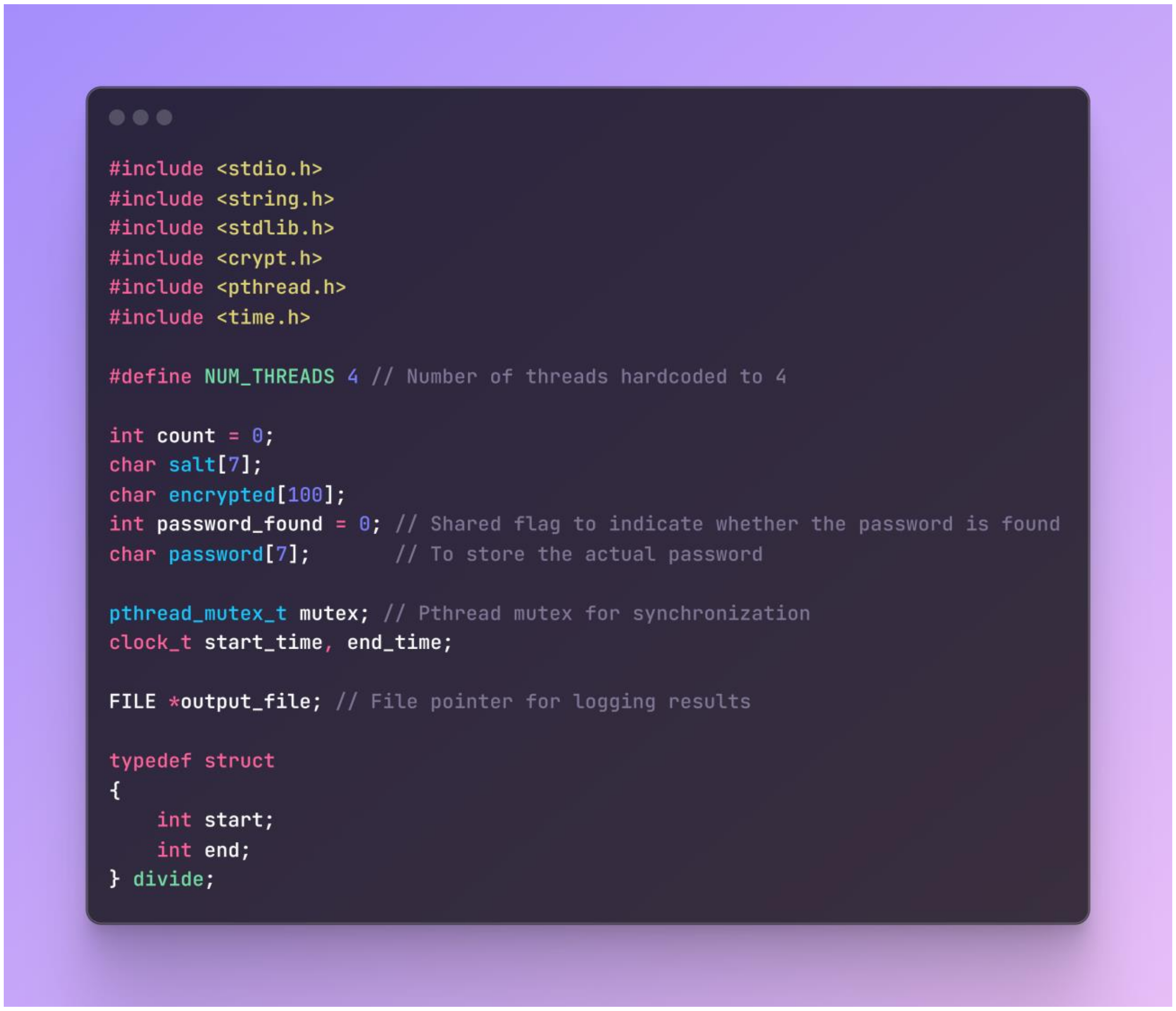
## Assessment overview

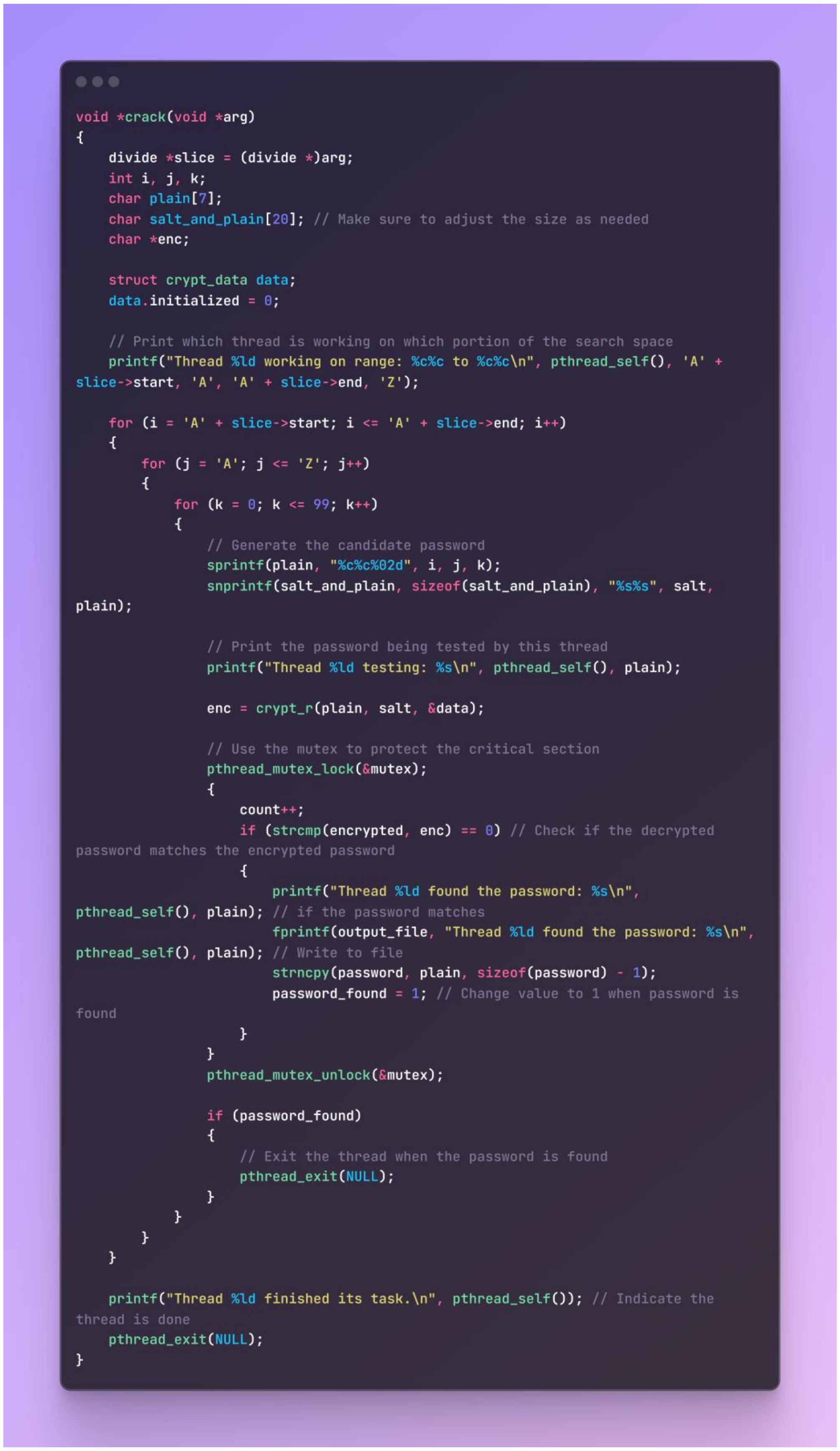
This portfolio is split up into 4 separate tasks which will test your knowledge of advanced multithreading and GPGPU programming using CUDA. Each task should be zipped up into a single zip folder containing all C/CUDA and resource files for the submission on Canvas.

## 1. Password cracking using multithreading (15% - 100 marks)

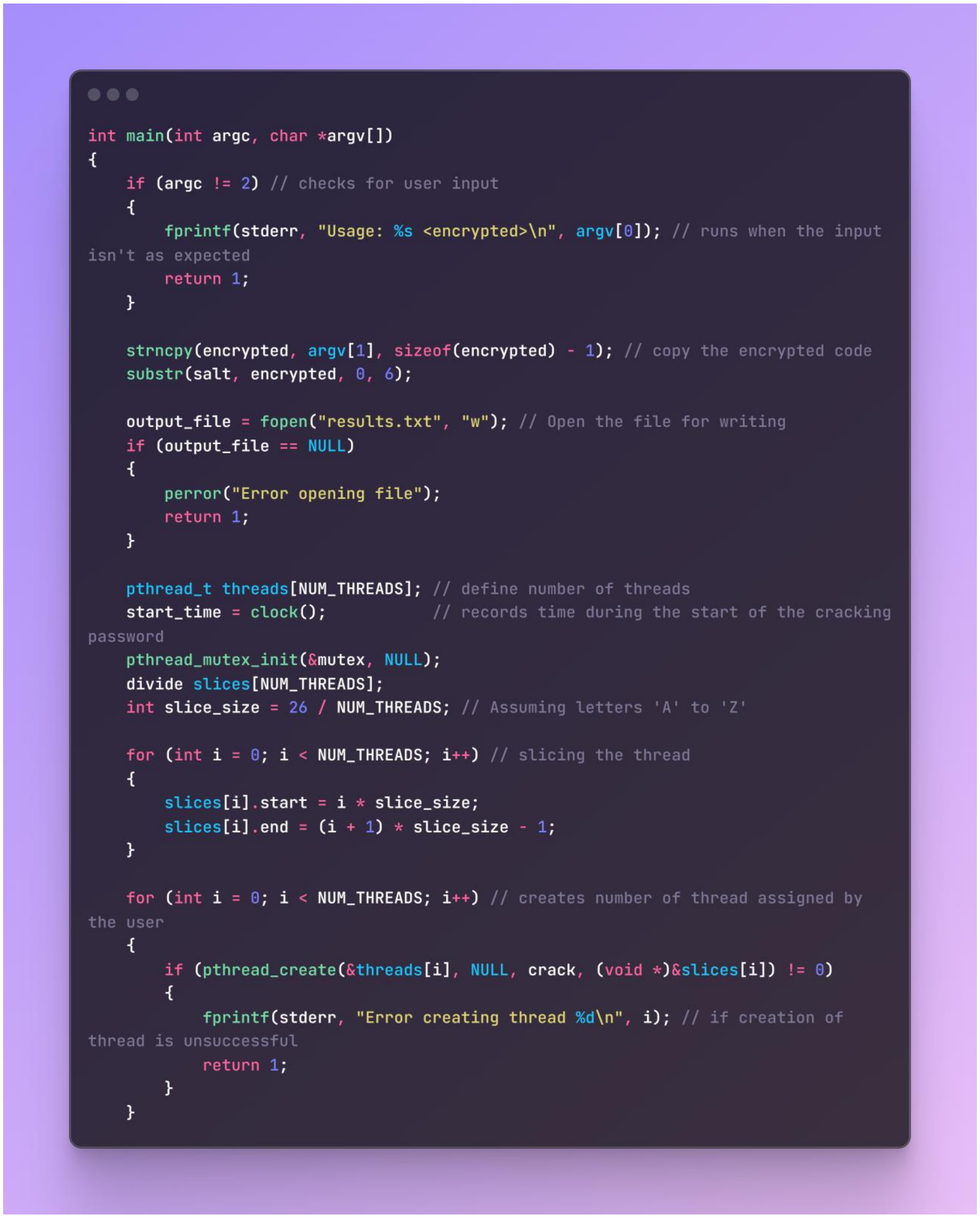
In this task, you will be asked to use the “crypt” library to decrypt a password using multithreading. You will be provided with two programs. The first program called “EncryptSHA512.c” which allows you to encrypt a password. For this assessment, you will be required to decrypt a 4-character password consisting of 2 capital letters, and 2 numbers. The format of the password should be “LetterLetterNumberNumber.” For example, “HP93.” Once you have generated your password, this should then be entered into your program to decrypt the password. The method of input for the encrypted password is up to you. The second program is a skeleton code to crack the password in regular C without any multithreading syntax. Your task is to use multithreading to split the workload over many threads and find the password. Once the password has been found, the program should finish meaning not all combinations of 2 letters and 2 numbers should be explored unless it’s ZZ99 and the last thread happens to finish last.



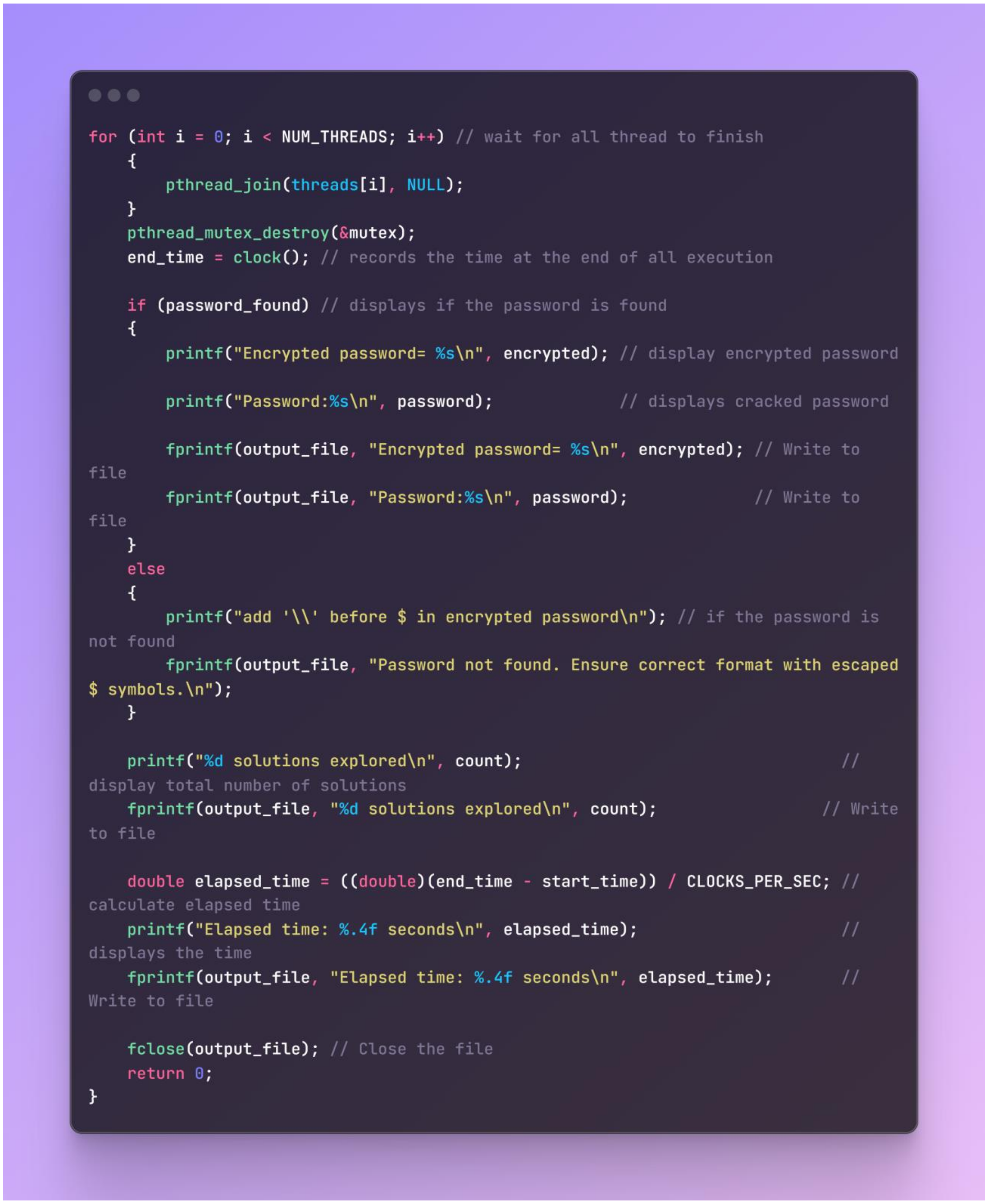
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <crypt.h>
#include <pthread.h>
#include <time.h>

#define NUM_THREADS 4 // Number of threads hardcoded to 4

int count = 0;
char salt[7];
char encrypted[100];
int password_found = 0; // Shared flag to indicate whether the password is found
char password[7];       // To store the actual password

pthread_mutex_t mutex; // Pthread mutex for synchronization
clock_t start_time, end_time;

FILE *output_file; // File pointer for logging results

typedef struct
{
    int start;
    int end;
} divide;
```



```
void *crack(void *arg)
{
    divide *slice = (divide *)arg;
    int i, j, k;
    char plain[7];
    char salt_and_plain[20]; // Make sure to adjust the size as needed
    char *enc;

    struct crypt_data data;
    data.initialized = 0;

    // Print which thread is working on which portion of the search space
    printf("Thread %ld working on range: %c%c to %c%c\n", pthread_self(), 'A' +
slice->start, 'A', 'A' + slice->end, 'Z');

    for (i = 'A' + slice->start; i <= 'A' + slice->end; i++)
    {
        for (j = 'A'; j <= 'Z'; j++)
        {
            for (k = 0; k <= 99; k++)
            {
                // Generate the candidate password
                sprintf(plain, "%c%c%02d", i, j, k);
                snprintf(salt_and_plain, sizeof(salt_and_plain), "%s%s", salt,
plain);

                // Print the password being tested by this thread
                printf("Thread %ld testing: %s\n", pthread_self(), plain);

                enc = crypt_r(plain, salt, &data);

                // Use the mutex to protect the critical section
                pthread_mutex_lock(&mutex);
                {
                    count++;
                    if (strcmp(encrypted, enc) == 0) // Check if the decrypted
password matches the encrypted password
                    {
                        printf("Thread %ld found the password: %s\n",
pthread_self(), plain); // if the password matches
                        fprintf(output_file, "Thread %ld found the password: %s\n",
pthread_self(), plain); // Write to file
                        strncpy(password, plain, sizeof(password) - 1);
                        password_found = 1; // Change value to 1 when password is
found
                    }
                }
                pthread_mutex_unlock(&mutex);

                if (password_found)
                {
                    // Exit the thread when the password is found
                    pthread_exit(NULL);
                }
            }
        }
    }

    printf("Thread %ld finished its task.\n", pthread_self()); // Indicate the
thread is done
    pthread_exit(NULL);
}
```



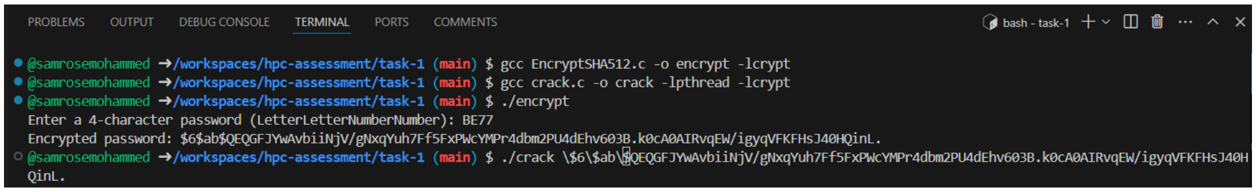
```c
int main(int argc, char *argv[])
{
    if (argc != 2) // checks for user input
    {
        fprintf(stderr, "Usage: %s <encrypted>\n", argv[0]); // runs when the input
isn't as expected
        return 1;
    }

    strncpy(encrypted, argv[1], sizeof(encrypted) - 1); // copy the encrypted code
    substr(salt, encrypted, 0, 6);

    output_file = fopen("results.txt", "w"); // Open the file for writing
    if (output_file == NULL)
    {
        perror("Error opening file");
        return 1;
    }

    pthread_t threads[NUM_THREADS]; // define number of threads
    start_time = clock();           // records time during the start of the cracking
password
    pthread_mutex_init(&mutex, NULL);
    divide slices[NUM_THREADS];
    int slice_size = 26 / NUM_THREADS; // Assuming letters 'A' to 'Z'

    for (int i = 0; i < NUM_THREADS; i++) // slicing the thread
    {
        slices[i].start = i * slice_size;
        slices[i].end = (i + 1) * slice_size - 1;
    }

    for (int i = 0; i < NUM_THREADS; i++) // creates number of thread assigned by
the user
    {
        if (pthread_create(&threads[i], NULL, crack, (void *)&slices[i]) != 0)
        {
            fprintf(stderr, "Error creating thread %d\n", i); // if creation of
thread is unsuccessful
            return 1;
        }
    }
```



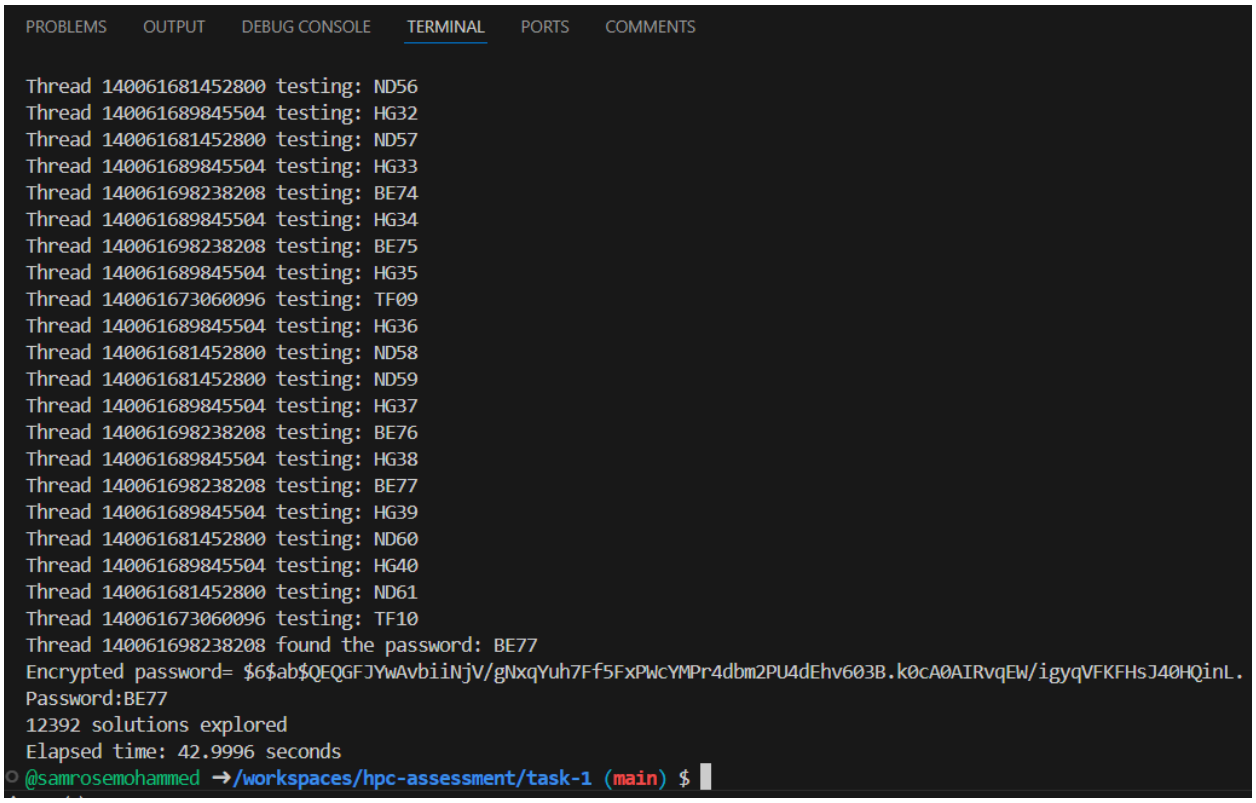
```c
for (int i = 0; i < NUM_THREADS; i++) // wait for all thread to finish
    {
        pthread_join(threads[i], NULL);
    }
    pthread_mutex_destroy(&mutex);
    end_time = clock(); // records the time at the end of all execution

    if (password_found) // displays if the password is found
    {
        printf("Encrypted password= %s\n", encrypted); // display encrypted password

        printf("Password:%s\n", password);             // displays cracked password

        fprintf(output_file, "Encrypted password= %s\n", encrypted); // Write to
file
        fprintf(output_file, "Password:%s\n", password);             // Write to
file
    }
    else
    {
        printf("add '\\' before $ in encrypted password\n"); // if the password is
not found
        fprintf(output_file, "Password not found. Ensure correct format with escaped
$ symbols.\n");
    }

    printf("%d solutions explored\n", count);                                 //
display total number of solutions
    fprintf(output_file, "%d solutions explored\n", count);                 // Write
to file

    double elapsed_time = ((double)(end_time - start_time)) / CLOCKS_PER_SEC; //
calculate elapsed time
    printf("Elapsed time: %.4f seconds\n", elapsed_time);                     //
displays the time
    fprintf(output_file, "Elapsed time: %.4f seconds\n", elapsed_time);       //
Write to file

    fclose(output_file); // Close the file
    return 0;
}
```

This C program is designed to crack an encrypted password using a brute-force method with multiple threads. The password is assumed to be in the format of two uppercase letters followed by a two-digit number (e.g., "AA00"). The program takes an encrypted password (with salt) as input and attempts to find the corresponding plaintext password by trying all possible combinations in parallel. It divides the search space across multiple threads (4 in this case) to improve efficiency.

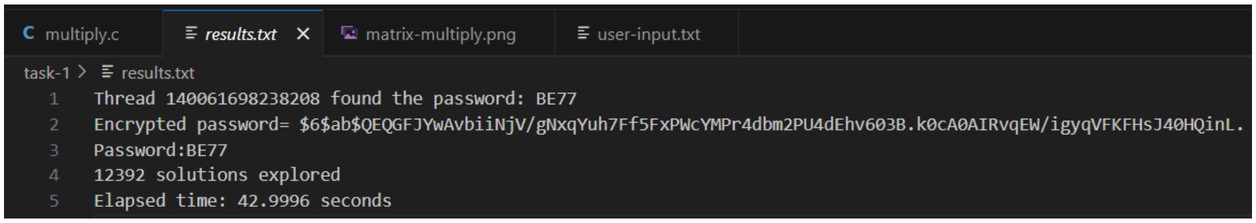
Each thread generates candidate passwords, encrypts them using the provided salt, and compares them to the encrypted password. If a match is found, the password is logged and displayed. The program also tracks and logs the number of attempted combinations and the total time taken to crack the password. Results are saved to a file called results.txt. The program uses synchronization (mutex) to handle shared resources safely and ensure accurate results across threads.



```
@samrosemohammed ➜/workspaces/hpc-assessment/task-1 (main) $ gcc EncryptSHA512.c -o encrypt -lcrypt
@samrosemohammed ➜/workspaces/hpc-assessment/task-1 (main) $ gcc crack.c -o crack -lpthread -lcrypt
@samrosemohammed ➜/workspaces/hpc-assessment/task-1 (main) $ ./encrypt
Enter a 4-character password (LetterLetterNumberNumber): BE77
Encrypted password: $6$ab$QEQGFJYwAvbiiNjV/gNxqYuh7Ff5FxPWcYMPr4dbm2PU4dEhv603B.k0cA0AIRvqEW/igyqVFKFHsJ40HQinL.
@samrosemohammed ➜/workspaces/hpc-assessment/task-1 (main) $ ./crack \$6\$ab\$QEQGFJYwAvbiiNjV/gNxqYuh7Ff5FxPWcYMPr4dbm2PU4dEhv603B.k0cA0AIRvqEW/igyqVFKFHsJ40H
QinL.
```



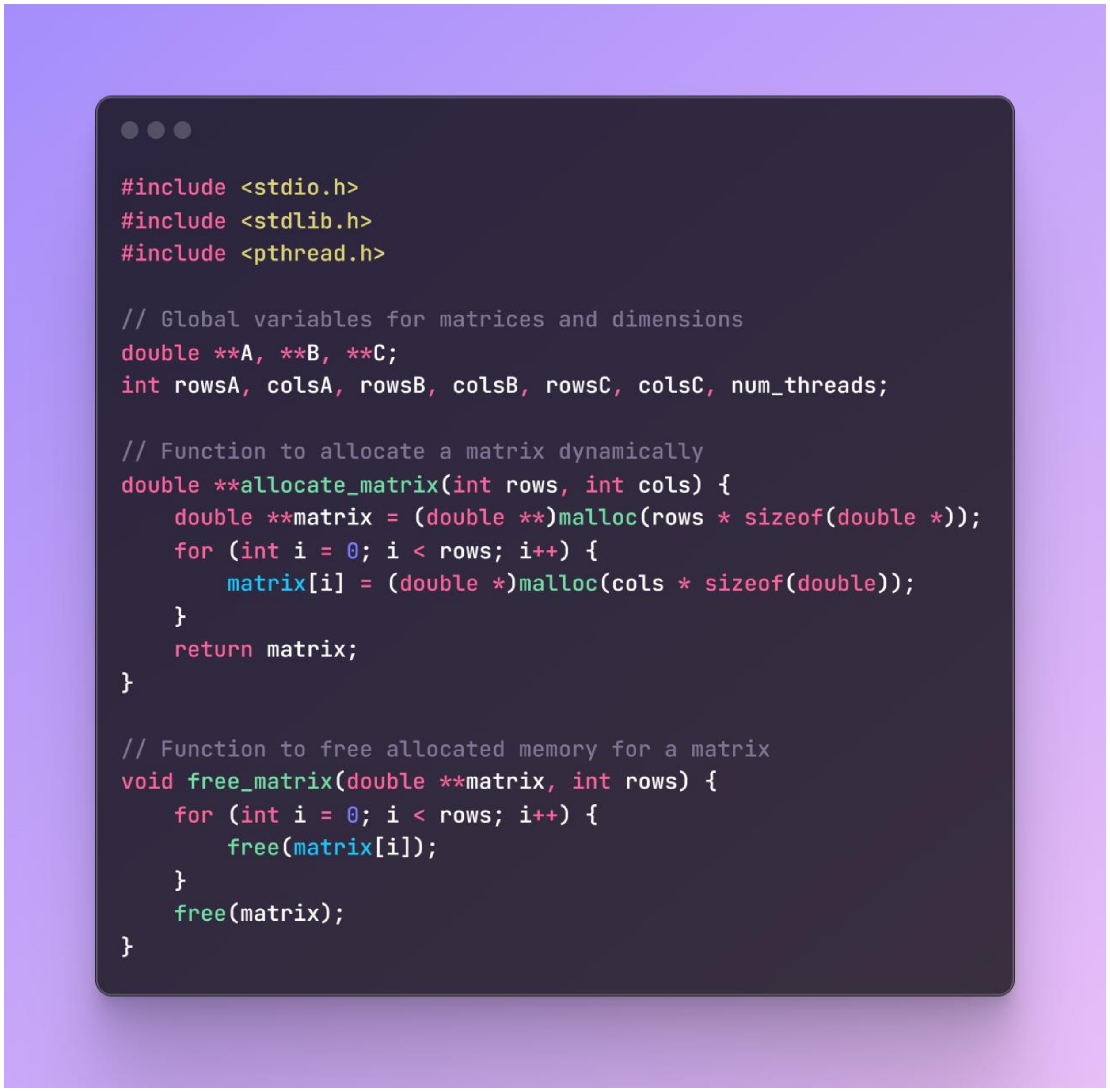
```
Thread 140061681452800 testing: ND56
Thread 140061689845504 testing: HG32
Thread 140061681452800 testing: ND57
Thread 140061689845504 testing: HG33
Thread 140061698238208 testing: BE74
Thread 140061689845504 testing: HG34
Thread 140061698238208 testing: BE75
Thread 140061689845504 testing: HG35
Thread 140061673060096 testing: TF09
Thread 140061689845504 testing: HG36
Thread 140061681452800 testing: ND58
Thread 140061681452800 testing: ND59
Thread 140061689845504 testing: HG37
Thread 140061698238208 testing: BE76
Thread 140061689845504 testing: HG38
Thread 140061698238208 testing: BE77
Thread 140061689845504 testing: HG39
Thread 140061681452800 testing: ND60
Thread 140061689845504 testing: HG40
Thread 140061681452800 testing: ND61
Thread 140061673060096 testing: TF10
Thread 140061698238208 found the password: BE77
Encrypted password= $6$ab$QEQGFJYwAvbiiNjV/gNxqYuh7Ff5FxPWcYMPr4dbm2PU4dEhv603B.k0cA0AIRvqEW/igyqVFKFHsJ40HQinL.
Password:BE77
12392 solutions explored
Elapsed time: 42.9996 seconds
@samrosemohammed ➜/workspaces/hpc-assessment/task-1 (main) $
```



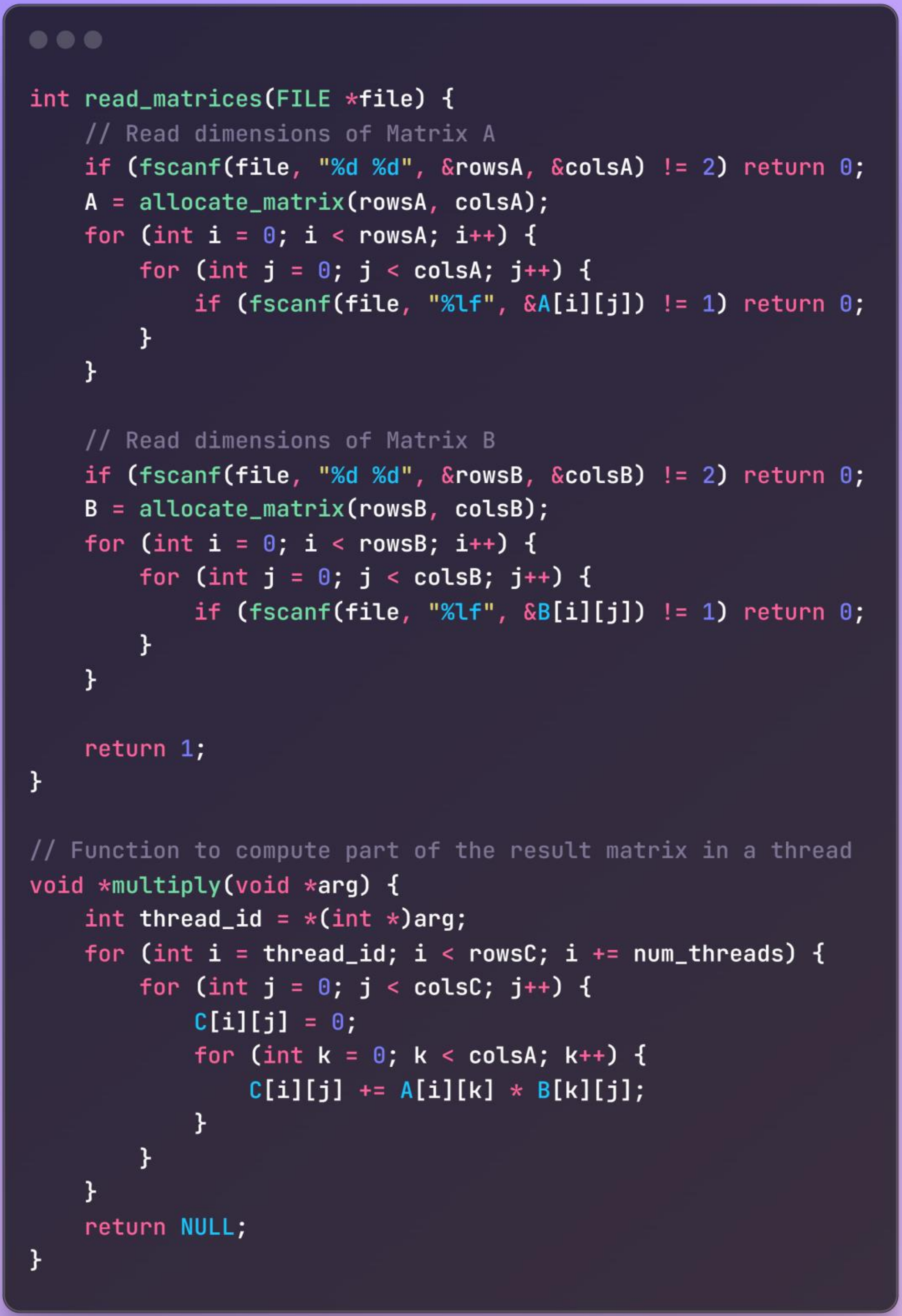
```
Thread 140061698238208 found the password: BE77
Encrypted password= $6$ab$QEQGFJYwAvbiiNjV/gNxqYuh7Ff5FxPWcYMPr4dbm2PU4dEhv603B.k0cA0AIRvqEW/igyqVFKFHsJ40HQinL.
Password:BE77
12392 solutions explored
Elapsed time: 42.9996 seconds
```

## 2. Matrix Multiplication using multithreading (25% - 100 marks)

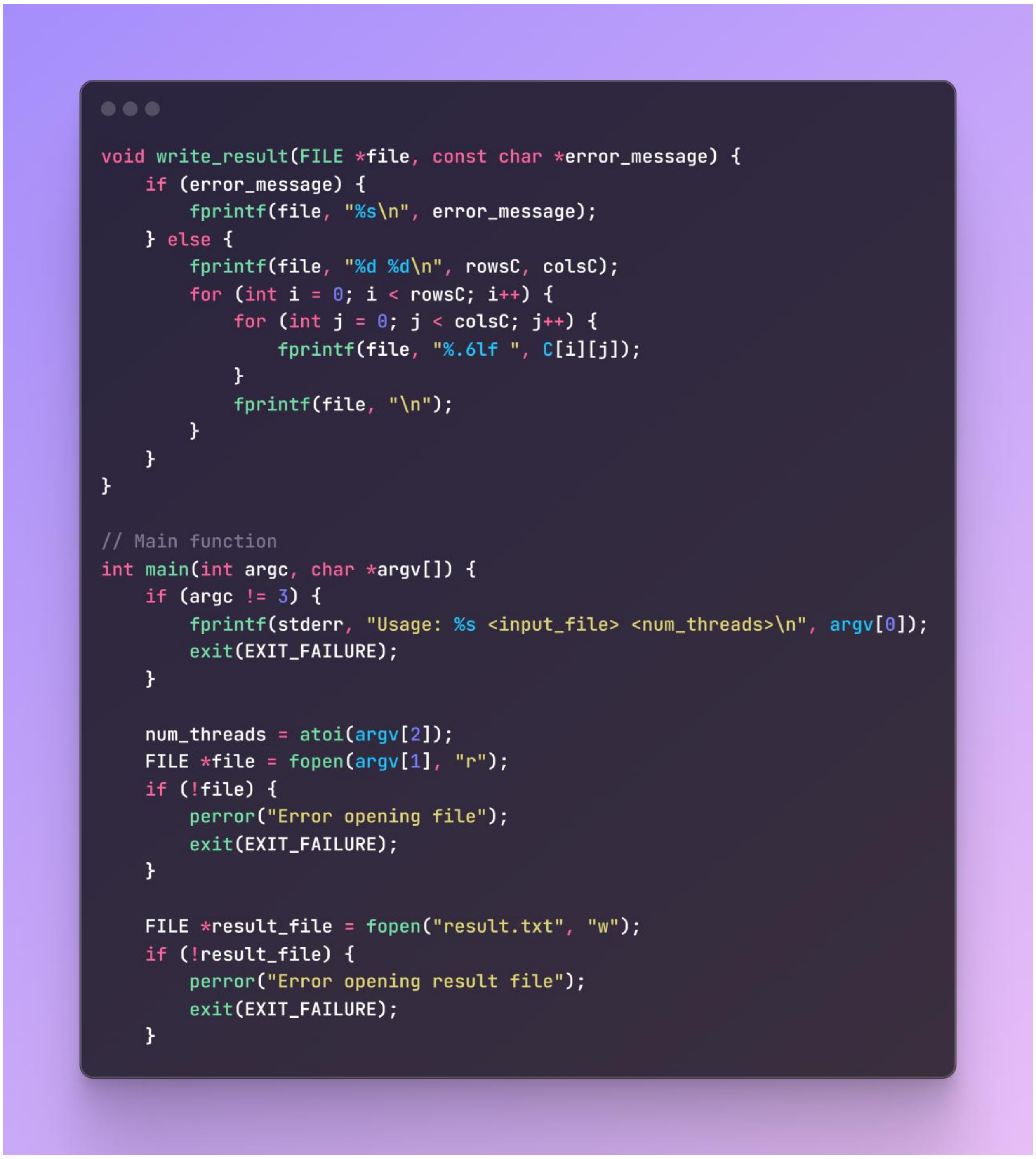
You will create a matrix multiplication program which uses multithreading. Matrices are often two-dimensional arrays varying in sizes, for your application, you will only need to multiply two-dimensional ones. Your program will read in the matrices from a supplied file (txt), store them appropriately using dynamic memory allocation features and multiply them by splitting the tasks across “n” threads (any amount of threads). You should use command line arguments to specify which file to read from (argv[1]) and the number of threads to use (argv[2]). If the number of threads requested by the user is greater than the biggest dimension of the matrices to be multiplied, the actual number of threads used in the calculation should be limited to the maximum dimension of the matrices. Your program should be able to take “any” size matrices and multiply them depending on the data found within the file. Some sizes of matrices cannot be multiplied together, for example, if Matrix A is 3x3 and Matrix B is 2x2, you cannot multiply them. If Matrix A is 2x3 and Matrix B is 3x2, then this can be multiplied. You will need to research how to multiply matrices, this will also be covered in the lectures (briefly). If the matrices cannot be multiplied, your program should output an error message notifying the user and move on to the next pair of matrices. Your program should store the results of your successful matrix multiplications in a text file – for every 2 matrices, there should be one resulting matrix. As a minimum, you are expected to use the standard C file handling functions: fopen(), fclose(), fscanf(), and fprintf(), to read and to write your files. stdin and stdout redirection will not be acceptable.



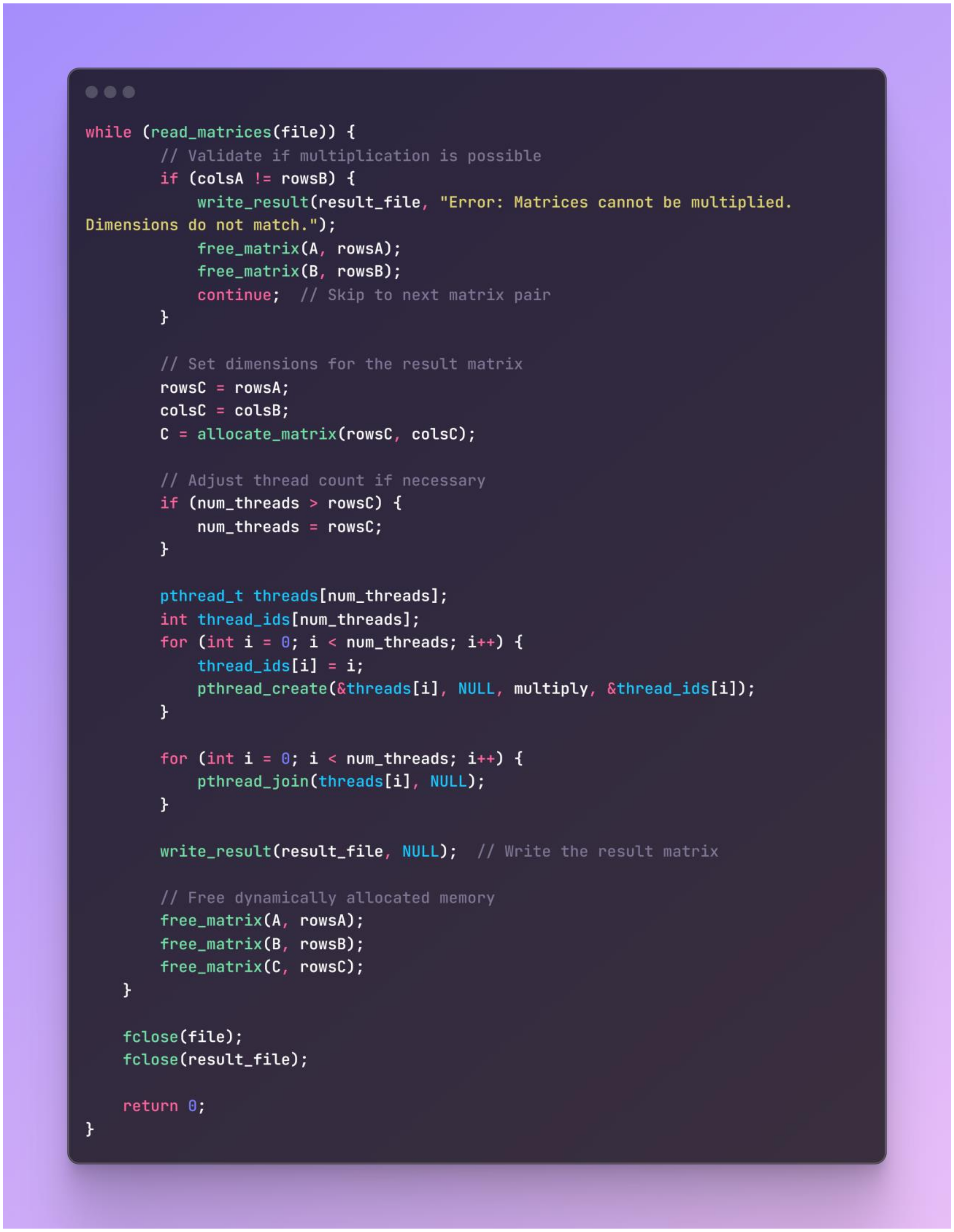
```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Global variables for matrices and dimensions
double **A, **B, **C;
int rowsA, colsA, rowsB, colsB, rowsC, colsC, num_threads;

// Function to allocate a matrix dynamically
double **allocate_matrix(int rows, int cols) {
    double **matrix = (double **)malloc(rows * sizeof(double *));
    for (int i = 0; i < rows; i++) {
        matrix[i] = (double *)malloc(cols * sizeof(double));
    }
    return matrix;
}

// Function to free allocated memory for a matrix
void free_matrix(double **matrix, int rows) {
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
}
```



```c
int read_matrices(FILE *file) {
    // Read dimensions of Matrix A
    if (fscanf(file, "%d %d", &rowsA, &colsA) != 2) return 0;
    A = allocate_matrix(rowsA, colsA);
    for (int i = 0; i < rowsA; i++) {
        for (int j = 0; j < colsA; j++) {
            if (fscanf(file, "%lf", &A[i][j]) != 1) return 0;
        }
    }

    // Read dimensions of Matrix B
    if (fscanf(file, "%d %d", &rowsB, &colsB) != 2) return 0;
    B = allocate_matrix(rowsB, colsB);
    for (int i = 0; i < rowsB; i++) {
        for (int j = 0; j < colsB; j++) {
            if (fscanf(file, "%lf", &B[i][j]) != 1) return 0;
        }
    }

    return 1;
}

// Function to compute part of the result matrix in a thread
void *multiply(void *arg) {
    int thread_id = *(int *)arg;
    for (int i = thread_id; i < rowsC; i += num_threads) {
        for (int j = 0; j < colsC; j++) {
            C[i][j] = 0;
            for (int k = 0; k < colsA; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    return NULL;
}
```



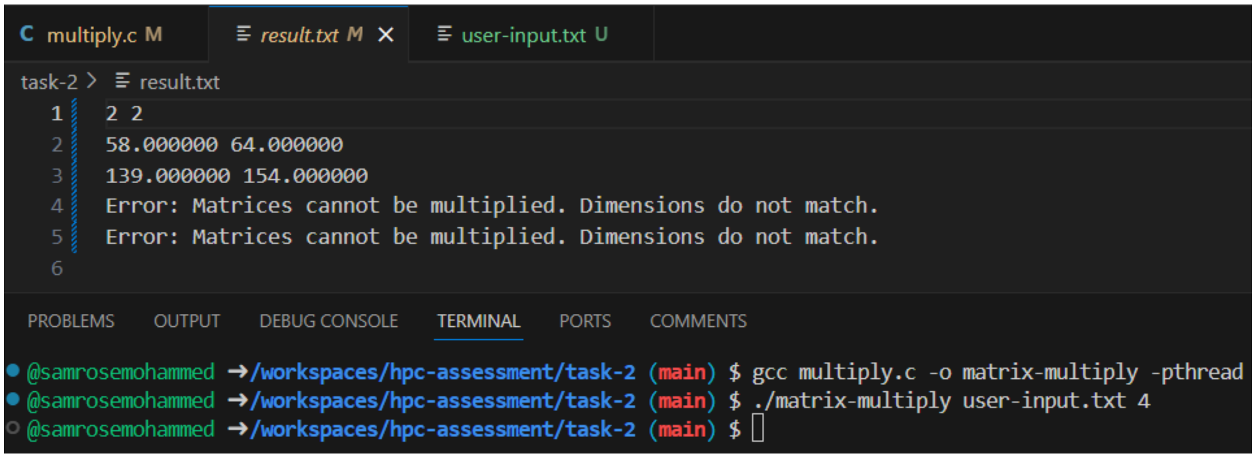
```c
void write_result(FILE *file, const char *error_message) {
    if (error_message) {
        fprintf(file, "%s\n", error_message);
    } else {
        fprintf(file, "%d %d\n", rowsC, colsC);
        for (int i = 0; i < rowsC; i++) {
            for (int j = 0; j < colsC; j++) {
                fprintf(file, "%.6lf ", C[i][j]);
            }
            fprintf(file, "\n");
        }
    }
}

// Main function
int main(int argc, char *argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: %s <input_file> <num_threads>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    num_threads = atoi(argv[2]);
    FILE *file = fopen(argv[1], "r");
    if (!file) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    FILE *result_file = fopen("result.txt", "w");
    if (!result_file) {
        perror("Error opening result file");
        exit(EXIT_FAILURE);
    }
```



```c
while (read_matrices(file)) {
        // Validate if multiplication is possible
        if (colsA != rowsB) {
            write_result(result_file, "Error: Matrices cannot be multiplied.
Dimensions do not match.");
            free_matrix(A, rowsA);
            free_matrix(B, rowsB);
            continue;  // Skip to next matrix pair
        }

        // Set dimensions for the result matrix
        rowsC = rowsA;
        colsC = colsB;
        C = allocate_matrix(rowsC, colsC);

        // Adjust thread count if necessary
        if (num_threads > rowsC) {
            num_threads = rowsC;
        }

        pthread_t threads[num_threads];
        int thread_ids[num_threads];
        for (int i = 0; i < num_threads; i++) {
            thread_ids[i] = i;
            pthread_create(&threads[i], NULL, multiply, &thread_ids[i]);
        }

        for (int i = 0; i < num_threads; i++) {
            pthread_join(threads[i], NULL);
        }

        write_result(result_file, NULL);  // Write the result matrix

        // Free dynamically allocated memory
        free_matrix(A, rowsA);
        free_matrix(B, rowsB);
        free_matrix(C, rowsC);
    }

    fclose(file);
    fclose(result_file);

    return 0;
}
```

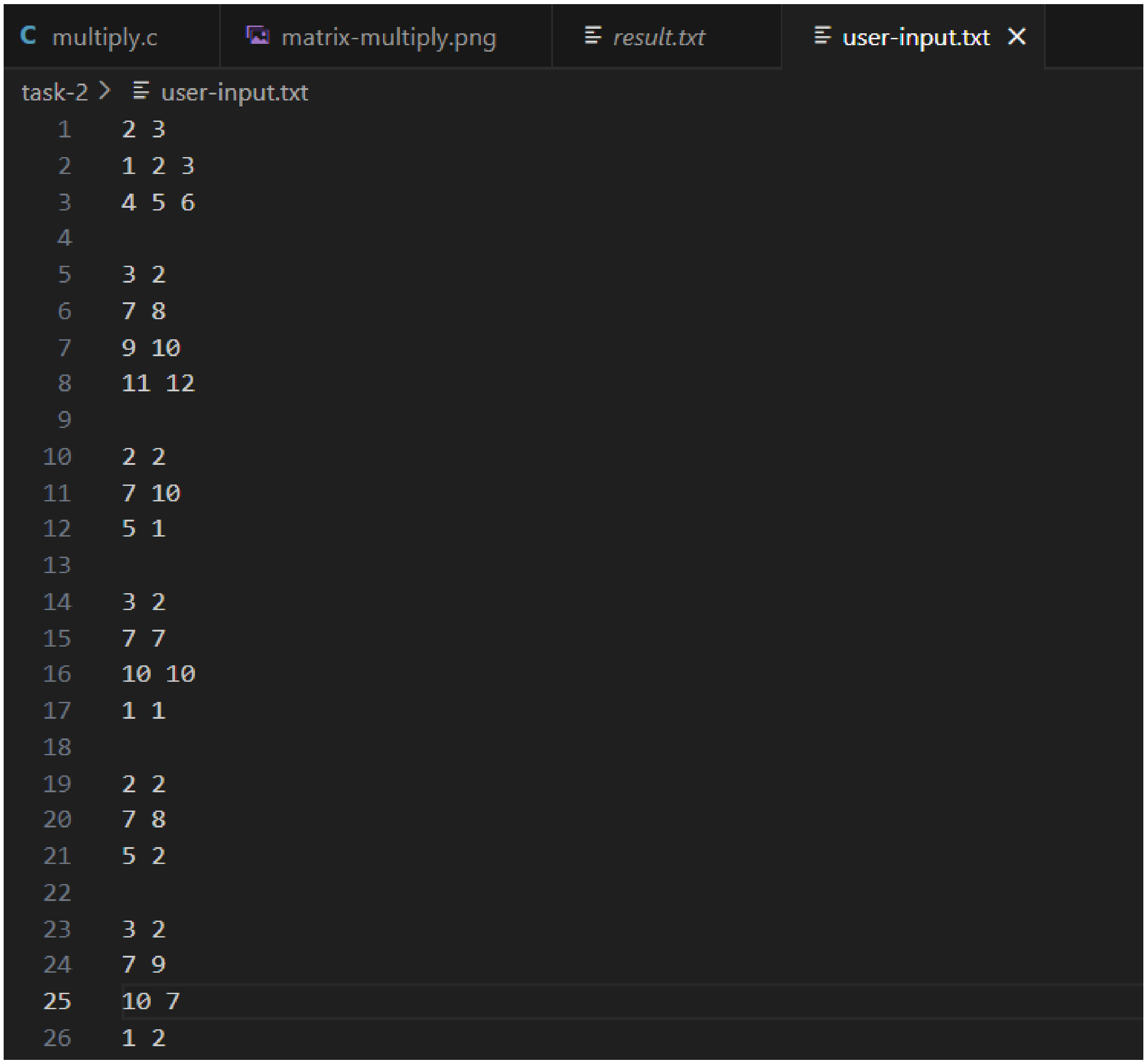
This C program performs multi-threaded matrix multiplication, reading matrices from an input file and writing the results to an output file (result.txt). It dynamically allocates memory for matrices and distributes the computation across multiple threads, where each thread processes a subset of rows in the resulting matrix. The number of threads is specified as a command-line argument, ensuring parallel execution for improved performance.

The program validates whether matrix multiplication is feasible by checking dimension compatibility. If the matrices cannot be multiplied, it logs an error message. After computing the product, it writes the resulting matrix to the output file. It also handles file I/O errors and ensures memory is properly allocated and freed, making it robust and efficient for large-scale matrix operations.

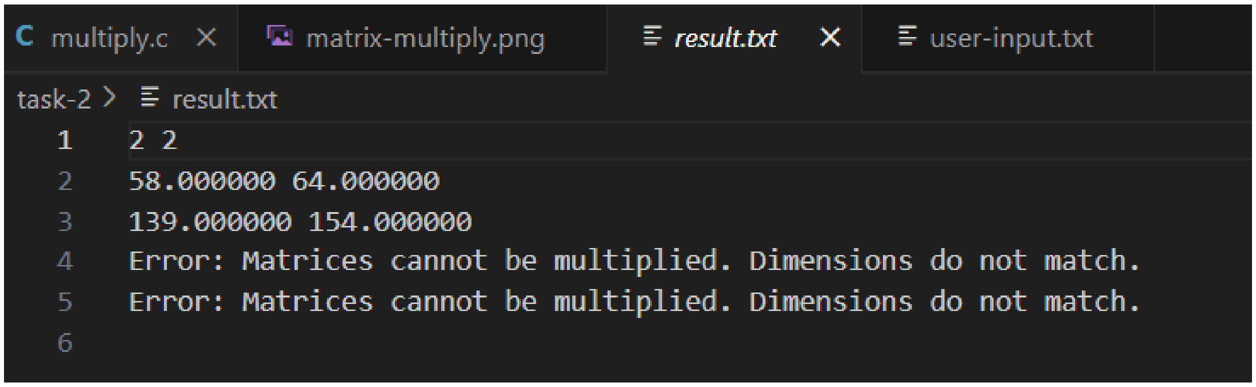


```
2 2
58.000000 64.000000
139.000000 154.000000
Error: Matrices cannot be multiplied. Dimensions do not match.
Error: Matrices cannot be multiplied. Dimensions do not match.
```

```
@samrosemohammed ➜ /workspaces/hpc-assessment/task-2 (main) $ gcc multiply.c -o matrix-multiply -pthread
@samrosemohammed ➜ /workspaces/hpc-assessment/task-2 (main) $ ./matrix-multiply user-input.txt 4
@samrosemohammed ➜ /workspaces/hpc-assessment/task-2 (main) $ 
```



```
2 3
1 2 3
4 5 6

3 2
7 8
9 10
11 12

2 2
7 10
5 1

3 2
7 7
10 10
1 1

2 2
7 8
5 2

3 2
7 9
10 7
1 2
```



```
2 2
58.000000 64.000000
139.000000 154.000000
Error: Matrices cannot be multiplied. Dimensions do not match.
Error: Matrices cannot be multiplied. Dimensions do not match.

```