TASK 2:

This C code implements a matrix multiplication program using pthreads (POSIX threads). The program reads matrices from an input file, performs matrix multiplication using a specified number of threads, and writes the result to an output file. It defines a structure ThreadInfo to hold information about each thread, including the matrices' dimensions and the range of rows it is responsible for. Functions for matrix allocation, freeing, reading from a file, and the actual matrix multiplication are implemented. The main function reads matrices from the input file, checks for compatibility, initializes threads, assigns work to each thread, waits for thread completion, and writes the result to the output file. The program takes command-line arguments for the input file and the number of threads to use. Additionally, it handles errors, such as insufficient command-line arguments, file opening failures, and matrix compatibility issues.

1. Read data from file appropriately:

The readMatrix function is responsible for reading matrices from a file. It uses dynamic memory allocation with malloc to create the matrices and then reads the matrix elements from the file using fscanf.

```c
// Read a matrix from a file and return the matrix
double **readMatrix(FILE *file, 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));
        for (int j = 0; j < cols; j++)
        {
            fscanf(file, "%lf,", &matrix[i][j]);
        }
    }
    return matrix;
}
```

2. Using dynamic memory (malloc) for matrix A and matrix B:

The program allocates memory for matrices using the allocateMatrix function, which also utilizes dynamic memory allocation with malloc. This ensures that memory is allocated based on the specified number of rows and columns for each matrix.

```c
// Allocate memory for a matrix and return the matrix
double **allocateMatrix(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;
}
```

3. Creating an algorithm to multiply matrices correctly:

The core matrix multiplication logic is implemented in the multiplyMatrices function. Each thread is assigned a specific range of rows to compute, and within this range, it performs the matrix multiplication using nested loops. The ThreadInfo structure holds information necessary for each thread to perform its computation.

```c
void *multiplyMatrices(void *arg)
{
    ThreadInfo *info = (ThreadInfo *)arg;
    printf("Thread %d is multiplying matrices of shape (%d, %d) and (%d, %d)\n",
           info->threadID, info->rowsMatrixA, info->colsMatrixA, info-
>rowsMatrixB, info->colsMatrixB);

    sleep(0.1);

    // Perform matrix multiplication for the assigned rows
    for (int i = info->startRow; i < info->endRow; i++)
    {
        for (int j = 0; j < info->colsMatrixB; j++)
        {
            double sum = 0.0;
            for (int k = 0; k < info->colsMatrixA; k++)
            {
                sum += info->matrixA[i][k] * info->matrixB[k][j];
            }
            info->resultMatrix[i][j] = sum;
        }
    }

    printf("Thread %d completed multiplication.\n", info->threadID);

    pthread_exit(NULL);
}
```

4. Using multithreading with equal computations:

The program creates a specified number of threads (determined by the command-line argument) in the main function. Each thread is assigned a portion of the matrix to compute, ensuring equal distribution of computation. The threads run concurrently, improving the overall performance of matrix multiplication.

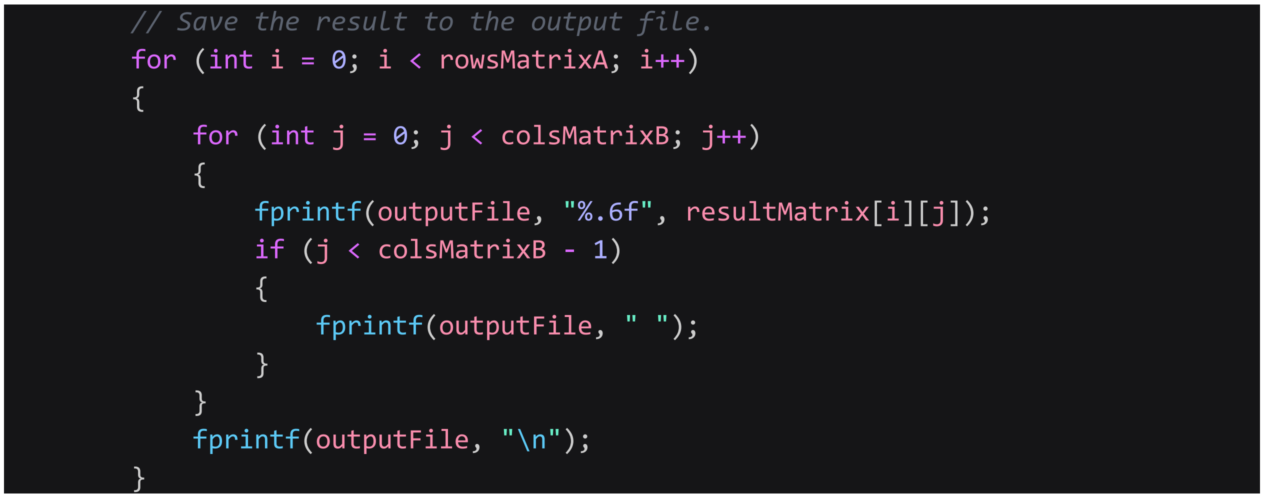
```
        // Calculate the number of rows each thread will process.
        int rowsPerThread = rowsMatrixA / numThreads;

        // Create threads for matrix multiplication.
        for (int i = 0; i < numThreads; i++)
        {
            // Create ThreadInfo structure for each thread and assign values.
            threadInfo[i] = (ThreadInfo){
                .matrixA = matrixA,
                .matrixB = matrixB,
                .resultMatrix = resultMatrix,
                .rowsMatrixA = rowsMatrixA,
                .colsMatrixA = colsMatrixA,
                .rowsMatrixB = rowsMatrixB,
                .colsMatrixB = colsMatrixB,
                .startRow = i * rowsPerThread,
                .endRow = (i == numThreads - 1) ? rowsMatrixA : (i + 1) *
rowsPerThread,
                .threadID = i,
            };

            // Create threads for matrix multiplication.
            pthread_create(&threads[i], NULL, multiplyMatrices, (void
*)&threadInfo[i]);
        }
```

5. Storing the correct output matrices in the correct format to a file:

The program writes the shape of the resultant matrix to the output file and then writes the calculated matrix. The resulting matrices are formatted and written to the file, with spaces between elements and newline characters to separate rows. Empty lines are added between matrices in the output file.



```
        // Save the result to the output file.
        for (int i = 0; i < rowsMatrixA; i++)
        {
            for (int j = 0; j < colsMatrixB; j++)
            {
                fprintf(outputFile, "%.6f", resultMatrix[i][j]);
                if (j < colsMatrixB - 1)
                {
                    fprintf(outputFile, " ");
                }
            }
            fprintf(outputFile, "\n");
        }
```