****

**BAHRIA UNIVERSITY, (Karachi Campus)**

*Department of Software Engineering*

**REPORT**

**COURSE: csc-320**

**Operating Systems**

**CLASS: BSE – 4A (SPRING - 2024)**

**Threaded Matrix Operations**

**Group Name**

|  |  |
| --- | --- |
| **Student Name** | **Enrollment#** |
| Syed Fahad Ali Gillani | 02-131222-083 |

**Submitted to:**

Course Instructor Lab Instructor . Engr. Majid Kaleem Engr. Saniya Sarim

Submission Date**: 11/06/2024**

**Abstract**

The threaded matrix operations project aims to enhance the computational efficiency of matrix calculations within an operating system environment. By leveraging multi-threading capabilities, the project seeks to improve performance, scalability, and resource utilization for matrix operations. The project will involve the development of a robust algorithm in linux environment to efficiently manage threads and resources. Through optimization techniques, the project aims to demonstrate the advantages of threaded matrix operations over conventional approaches.

Table of Contents

[1. INTRODUCTION: 4](#_Toc169044539)

[2. ABSTRACT: 4](#_Toc169044540)

[3. PROJECT SCOPE: 4](#_Toc169044541)

[4. PROJECT FUNCTIONALITIES: 4](#_Toc169044542)

[5. MODULE DISTRIBUTION 5](#_Toc169044543)

[6. CODE 5](#_Toc169044544)

# **INTRODUCTION:**

Threaded matrix operations aim to harness the power of parallel computing by utilizing multiple threads to perform matrix calculations concurrently. This approach can significantly enhance the performance and scalability of matrix operations on contemporary hardware architectures.

# **ABSTRACT:**

The threaded matrix operations project aims to enhance the computational efficiency of matrix calculations within an operating system environment. By leveraging multi-threading capabilities, the project seeks to improve performance, scalability, and resource utilization for matrix operations. The project will involve the development of a robust algorithm in linux environment to efficiently manage threads and resources. Through optimization techniques, the project aims to demonstrate the advantages of threaded matrix operations over conventional approaches.

# **PROJECT SCOPE:**

The scope of this project encompasses the design and implementation of threaded matrix operations within an operating system environment. Key components of the project include:

* Development of a multi-threaded framework for matrix operations
* Optimization techniques to enhance performance and resource utilization
* Compatibility with standard matrix operations and data structures
* Support for various matrix sizes and configurations
* Evaluation to assess the effectiveness of threaded matrix operations compared to traditional methods

# **PROJECT FUNCTIONALITIES:**

The threaded matrix operations project will include the following functionalities:

* Creation and management of multiple threads for parallel matrix calculations
* Synchronization mechanisms to ensure data integrity and consistency across threads
* Utilization of multi-core processors to distribute matrix operations effectively
* Implementation of standard matrix operations

# **MODULE DISTRIBUTION**

The project was initiated by setting up a new C project directory. Following this, essential header and source files—namely matrix.h, matrix.c, and main.c—were created to facilitate the development process. In matrix.h, the data structures and function prototypes required for matrix operations were defined, while matrix.c housed the implementations of these functions, covering matrix creation, input/output, and various operations like addition, subtraction, multiplication, and transpose. The main user interface, crafted in main.c, provided users with options to choose different matrix operations and featured input prompts for enhanced user interaction. To ensure robustness, input validation functions (safeReadInt and safeReadChoice) were integrated to validate user inputs effectively. Extensive testing was conducted throughout the development process to guarantee the correctness and reliability of matrix operations across diverse scenarios, thus culminating in a well-structured and functional project.

# **CODE**

**Matrix.h**

#ifndef MATRIX\_H

#define MATRIX\_H

#include <pthread.h>

typedef struct {

int rows;

int cols;

int\*\* data;

} Matrix;

Matrix createMatrix(int rows, int cols);

void freeMatrix(Matrix matrix);

Matrix add(const Matrix\* A, const Matrix\* B);

Matrix subtract(const Matrix\* A, const Matrix\* B);

Matrix multiply(const Matrix\* A, const Matrix\* B);

Matrix transpose(const Matrix\* A);

void readMatrixFromInput(Matrix\* matrix);

void printMatrix(const Matrix\* matrix);

extern pthread\_mutex\_t mutex;

#endif // MATRIX\_H

**Matrix.c**

#include "matrix.h"

#include <stdlib.h>

#include <stdio.h>

#include <pthread.h>

typedef struct {

const Matrix\* A;

const Matrix\* B;

Matrix\* result;

int row;

int col;

} ThreadData;

pthread\_mutex\_t mutex;

Matrix createMatrix(int rows, int cols) {

Matrix matrix;

matrix.rows = rows;

matrix.cols = cols;

matrix.data = (int\*\*)malloc(rows \* sizeof(int\*));

for (int i = 0; i < rows; ++i) {

matrix.data[i] = (int\*)malloc(cols \* sizeof(int));

}

return matrix;

}

void freeMatrix(Matrix matrix) {

for (int i = 0; i < matrix.rows; ++i) {

free(matrix.data[i]);

}

free(matrix.data);

}

void readMatrixFromInput(Matrix\* matrix) {

for (int i = 0; i < matrix->rows; ++i) {

for (int j = 0; j < matrix->cols; ++j) {

printf("Enter value for element [%d][%d]: ", i, j);

scanf("%d", &matrix->data[i][j]);

}

}

}

void printMatrix(const Matrix\* matrix) {

for (int i = 0; i < matrix->rows; ++i) {

for (int j = 0; j < matrix->cols; ++j) {

printf("%d ", matrix->data[i][j]);

}

printf("\n");

}

}

void\* add\_thread(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

pthread\_mutex\_lock(&mutex);

data->result->data[data->row][data->col] = data->A->data[data->row][data->col] + data->B->data[data->row][data->col];

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* subtract\_thread(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

pthread\_mutex\_lock(&mutex);

data->result->data[data->row][data->col] = data->A->data[data->row][data->col] - data->B->data[data->row][data->col];

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* multiply\_thread(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

int sum = 0;

for (int k = 0; k < data->A->cols; ++k) {

sum += data->A->data[data->row][k] \* data->B->data[k][data->col];

}

pthread\_mutex\_lock(&mutex);

data->result->data[data->row][data->col] = sum;

pthread\_mutex\_unlock(&mutex);

return NULL;

}

void\* transpose\_thread(void\* arg) {

ThreadData\* data = (ThreadData\*)arg;

pthread\_mutex\_lock(&mutex);

data->result->data[data->col][data->row] = data->A->data[data->row][data->col];

pthread\_mutex\_unlock(&mutex);

return NULL;

}

Matrix add(const Matrix\* A, const Matrix\* B) {

if (A->rows != B->rows || A->cols != B->cols) {

fprintf(stderr, "Error: Matrices must have the same dimensions for addition.\n");

exit(EXIT\_FAILURE);

}

Matrix result = createMatrix(A->rows, A->cols);

pthread\_t threads[A->rows \* A->cols];

ThreadData data[A->rows \* A->cols];

int threadIndex = 0;

for (int i = 0; i < A->rows; ++i) {

for (int j = 0; j < A->cols; ++j) {

data[threadIndex] = (ThreadData){A, B, &result, i, j};

pthread\_create(&threads[threadIndex], NULL, add\_thread, &data[threadIndex]);

++threadIndex;

}

}

for (int i = 0; i < threadIndex; ++i) {

pthread\_join(threads[i], NULL);

}

return result;

}

Matrix subtract(const Matrix\* A, const Matrix\* B) {

if (A->rows != B->rows || A->cols != B->cols) {

fprintf(stderr, "Error: Matrices must have the same dimensions for subtraction.\n");

exit(EXIT\_FAILURE);

}

Matrix result = createMatrix(A->rows, A->cols);

pthread\_t threads[A->rows \* A->cols];

ThreadData data[A->rows \* A->cols];

int threadIndex = 0;

for (int i = 0; i < A->rows; ++i) {

for (int j = 0; j < A->cols; ++j) {

data[threadIndex] = (ThreadData){A, B, &result, i, j};

pthread\_create(&threads[threadIndex], NULL, subtract\_thread, &data[threadIndex]);

++threadIndex;

}

}

for (int i = 0; i < threadIndex; ++i) {

pthread\_join(threads[i], NULL);

}

return result;

}

Matrix multiply(const Matrix\* A, const Matrix\* B) {

if (A->cols != B->rows) {

fprintf(stderr, "Error: Number of columns in A must be equal to number of rows in B for multiplication.\n");

exit(EXIT\_FAILURE);

}

Matrix result = createMatrix(A->rows, B->cols);

pthread\_t threads[A->rows \* B->cols];

ThreadData data[A->rows \* B->cols];

int threadIndex = 0;

for (int i = 0; i < A->rows; ++i) {

for (int j = 0; j < B->cols; ++j) {

data[threadIndex] = (ThreadData){A, B, &result, i, j};

pthread\_create(&threads[threadIndex], NULL, multiply\_thread, &data[threadIndex]);

++threadIndex;

}

}

for (int i = 0; i < threadIndex; ++i) {

pthread\_join(threads[i], NULL);

}

return result;

}

Matrix transpose(const Matrix\* A) {

Matrix result = createMatrix(A->cols, A->rows);

pthread\_t threads[A->rows \* A->cols];

ThreadData data[A->rows \* A->cols];

int threadIndex = 0;

for (int i = 0; i < A->rows; ++i) {

for (int j = 0; j < A->cols; ++j) {

data[threadIndex] = (ThreadData){A, NULL, &result, i, j};

pthread\_create(&threads[threadIndex], NULL, transpose\_thread, &data[threadIndex]);

++threadIndex;

}

}

for (int i = 0; i < threadIndex; ++i) {

pthread\_join(threads[i], NULL);

}

return result;

}

**Main.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <errno.h>

#include "matrix.h"

// Function to safely read an integer from standard input

int safeReadInt(const char\* prompt) {

char input[100];

int value;

char\* endptr;

while (1) {

printf("%s", prompt);

if (!fgets(input, sizeof(input), stdin)) {

printf("Error reading input. Please try again.\n");

continue;

}

// Remove newline character if present

input[strcspn(input, "\n")] = '\0';

// Convert string to integer

errno = 0;

value = strtol(input, &endptr, 10);

if (errno != 0 || \*endptr != '\0') {

printf("Invalid input. Please enter a valid integer.\n");

} else {

return value;

}

}

}

// Function to safely read a single character choice from standard input

char safeReadChoice(const char\* prompt) {

char input[100];

while (1) {

printf("%s", prompt);

if (!fgets(input, sizeof(input), stdin)) {

printf("Error reading input. Please try again.\n");

continue;

}

// Remove newline character if present

input[strcspn(input, "\n")] = '\0';

// Check if input is a single character

if (strlen(input) == 1) {

return input[0];

} else {

printf("Invalid input. Please enter a single character.\n");

}

}

}

int main() {

int rows\_A, cols\_A, rows\_B, cols\_B;

char choice;

pthread\_mutex\_init(&mutex, NULL); // Initialize the mutex

do {

printf("\nChoose an operation:\n");

printf("1. Addition\n");

printf("2. Subtraction\n");

printf("3. Multiplication\n");

printf("4. Transpose\n");

printf("5. Exit\n");

choice = safeReadChoice("Enter your choice: ");

switch (choice) {

case '1':

rows\_A = safeReadInt("Enter the number of rows for Matrix A: ");

cols\_A = safeReadInt("Enter the number of columns for Matrix A: ");

Matrix A = createMatrix(rows\_A, cols\_A);

printf("Enter the values for Matrix A:\n");

readMatrixFromInput(&A);

// Clear input buffer

while ((getchar()) != '\n');

rows\_B = safeReadInt("Enter the number of rows for Matrix B: ");

cols\_B = safeReadInt("Enter the number of columns for Matrix B: ");

Matrix B = createMatrix(rows\_B, cols\_B);

printf("Enter the values for Matrix B:\n");

readMatrixFromInput(&B);

if (rows\_A == rows\_B && cols\_A == cols\_B) {

Matrix result = add(&A, &B);

printf("Addition Result:\n");

printMatrix(&result);

freeMatrix(result);

} else {

printf("Matrices must have the same dimensions for addition.\n");

}

freeMatrix(A);

freeMatrix(B);

// Clear input buffer

while ((getchar()) != '\n');

break;

case '2':

rows\_A = safeReadInt("Enter the number of rows for Matrix A: ");

cols\_A = safeReadInt("Enter the number of columns for Matrix A: ");

Matrix A\_sub = createMatrix(rows\_A, cols\_A);

printf("Enter the values for Matrix A:\n");

readMatrixFromInput(&A\_sub);

while ((getchar()) != '\n');

rows\_B = safeReadInt("Enter the number of rows for Matrix B: ");

cols\_B = safeReadInt("Enter the number of columns for Matrix B: ");

Matrix B\_sub = createMatrix(rows\_B, cols\_B);

printf("Enter the values for Matrix B:\n");

readMatrixFromInput(&B\_sub);

if (rows\_A == rows\_B && cols\_A == cols\_B) {

Matrix result = subtract(&A\_sub, &B\_sub);

printf("Subtraction Result:\n");

printMatrix(&result);

freeMatrix(result);

} else {

printf("Matrices must have the same dimensions for subtraction.\n");

}

freeMatrix(A\_sub);

freeMatrix(B\_sub);

while ((getchar()) != '\n');

break;

case '3':

rows\_A = safeReadInt("Enter the number of rows for Matrix A: ");

cols\_A = safeReadInt("Enter the number of columns for Matrix A: ");

Matrix A\_mul = createMatrix(rows\_A, cols\_A);

printf("Enter the values for Matrix A:\n");

readMatrixFromInput(&A\_mul);

while ((getchar()) != '\n');

rows\_B = safeReadInt("Enter the number of rows for Matrix B: ");

cols\_B = safeReadInt("Enter the number of columns for Matrix B: ");

Matrix B\_mul = createMatrix(rows\_B, cols\_B);

printf("Enter the values for Matrix B:\n");

readMatrixFromInput(&B\_mul);

if (cols\_A == rows\_B) {

Matrix result = multiply(&A\_mul, &B\_mul);

printf("Multiplication Result:\n");

printMatrix(&result);

freeMatrix(result);

} else {

printf("Number of columns in Matrix A must be equal to the number of rows in Matrix B for multiplication.\n");

}

freeMatrix(A\_mul);

freeMatrix(B\_mul);

while ((getchar()) != '\n');

break;

case '4':

rows\_A = safeReadInt("Enter the number of rows for Matrix A: ");

cols\_A = safeReadInt("Enter the number of columns for Matrix A: ");

Matrix A\_transpose = createMatrix(rows\_A, cols\_A);

printf("Enter the values for Matrix A:\n");

readMatrixFromInput(&A\_transpose);

while ((getchar()) != '\n');

Matrix result\_transpose = transpose(&A\_transpose);

printf("Transpose of Matrix A Result:\n");

printMatrix(&result\_transpose);

freeMatrix(result\_transpose);

freeMatrix(A\_transpose);

while ((getchar()) != '\n');

break;

case '5':

printf("Exiting program.\n");

break;

default:

printf("Invalid choice. Please try again.\n");

break;

}

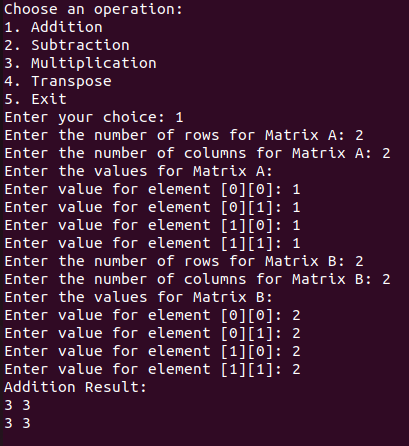
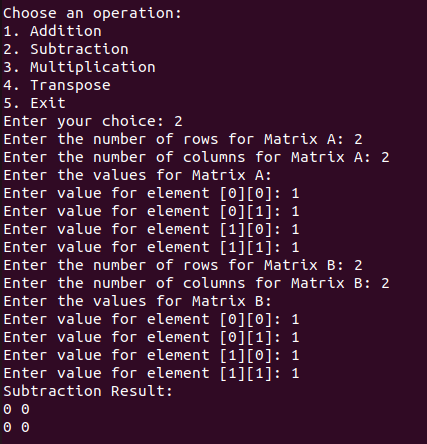
} while (choice != '5');

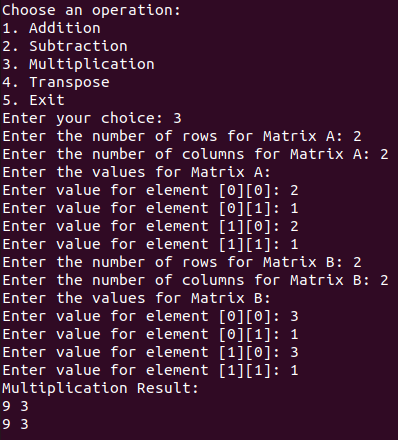
pthread\_mutex\_destroy(&mutex); // Destroy the mutex

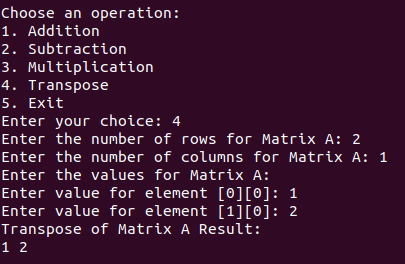
return 0;

}

1. **OUTPUTS**

** **

****

****

**Teacher Signature**: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Remarks**: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Submission Date**: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_