Московский Авиационный Институт

(Национальный Исследовательский Университет)

Институт №8 “Компьютерные науки и прикладная математика”

Кафедра №806 “Вычислительная математика и программирование”

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Группа: М8О-211БВ-24

Студент: Губеев Д.И.

Преподаватель: Бахарев В.Д.

Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Дата: 02.10.25

Москва, 2025

**Постановка задачи**

**Вариант 10.**

Решить систему линейных уравнений методом Гаусса.

**Общий метод и алгоритм решения**

Использованные системные вызовы:

* ssize\_t read(int fd, void \*buf, size\_t count) - читает данные из файла по дескриптору fd в буфер buf. Возвращает количество прочитанных байт
* ssize\_t write(int fd, const void \*buf, size\_t count) - записывает данные в файл по дескриптору fd. Возвращает число записанных байт
* int open(const char \*pathname, int flag, mode\_t mode) - Открывает файл, возвращает файловый дескриптор
* int close(int fd) - закрываает файловый дескриптор
* int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*work)(void\*), void \*arg) - создает новый поток с заданными атрибутами, который начинает выполнение функции work. Возвращает 0 при успехе, либо код ошибки при неудаче
* int pthread\_join(pthread\_t thread, void \*\*retval) - ожидает завершения указанного потока. Поток, вызвавший pthread\_join, блокируется до завершения thread.
* int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex) - блокирует мьютекс. Если он уже заблокирован, поток ожидает, пока мьютекс не освободится. Используется для предотвращения состояния гонки.
* int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex) - разблокирует мьютекс, позволяя другим потокам продолжить работу с разделяемыми данными.
* int pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr) - инициализирует мьютекс перед использованием.
* int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex) - уничтожает мьютекс и освобождает связанные с ним ресурсы.
* int pthread\_cond\_wait(pthread\_cond\_t \*cond, pthread\_mutex\_t \*mutex) - приостанавливает поток, ожидая сигнала от другой части программы через условную переменную cond. При этом временно отпускает мьютекс и снова захватывает его после пробуждения.
* int pthread\_cond\_broadcast(pthread\_cond\_t \*cond) - пробуждает все потоки, ожидающие на условной переменной cond.
* int pthread\_cond\_init(pthread\_cond\_t \*cond, const pthread\_condattr\_t \*attr) - инициализирует условную переменную.
* int pthread\_cond\_destroy(pthread\_cond\_t \*cond) - освобождает ресурсы условной переменной.

Читается матрица из файла и столбец свободных членов. Затем создаются потоки и решают СЛАУ методом Гаусса.

Для прямого хода первый поток находит строку, которая будет обнулять остальные в данном столбце. Далее множество потоков параллельно обнуляет данный столбец в каждой строке. Так делается до тех пор, пока матрица не будет приведена к ступенчатому виду.

В обратном ходе первый поток находит первую ненулевую ячейку в данной строке и делит строку на значение в этой ячейке(чтобы коэффициент в ячейке был равен одному). Далее потоки параллельно обнуляют ячейки во всех строках в данном столбце.

После этого выводится решение в результирующий массив. Потоки параллельно проходятся по столбцам, находят строку соответствующую корню данного столбца, и записывают решение.

**Код программы**

**main\_with\_threads.c**

#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include <pthread.h>

#include <stdbool.h>

#include <unistd.h>

#include <sys/wait.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <ctype.h>

#include <stdint.h>

#include <string.h>

#include <sys/time.h>

#define EPSILON 1e-9

#define MAX\_THREAD\_NUM 5000

static pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER;

typedef struct {

pthread\_mutex\_t mutex; // Мьютекс для защиты данных барьера

pthread\_cond\_t cond; // Условная переменная для ожидания

int count; // Текущий счётчик достигших потоков

int total; // Общее количество потоков для барьера

} Barrier;

typedef struct {

double \*\*matrix;

int rows;

int cols;

double \*b;

double \*solution;

int threadNum;

int threadID;

double \*\*mergedMatrix;

bool \*pivotIsFound;

int \*colForReverse;

Barrier \*barrier;

} ThreadArgs;

void barrier\_wait(Barrier \*barrier) {

pthread\_mutex\_lock(&barrier->mutex);

barrier->count++;

if (barrier->count < barrier->total) {

// Поток ждёт, пока все не достигнут барьера

pthread\_cond\_wait(&barrier->cond, &barrier->mutex);

} else {

// Последний поток уведомляет все ждущие

barrier->count = 0; // Сбрасываем счётчик для следующего использования

pthread\_cond\_broadcast(&barrier->cond);

}

pthread\_mutex\_unlock(&barrier->mutex);

}

void \*solveWithGauss(void \* \_args) {

ThreadArgs \*args = (ThreadArgs \*)\_args;

double \*\*matrix = args->matrix;

int rows = args->rows;

int cols = args->cols;

double \*b = args->b;

double \*solution = args->solution;

double \*\*merged = args->mergedMatrix;

cols++;

size\_t rowI = 0;

size\_t iterations = rows < cols ? rows : cols;

for (size\_t i = 0; i < iterations; i++)

{

barrier\_wait(args->barrier);

pthread\_mutex\_lock(&mutex);

if (args->threadID == 0) {

\*(args->pivotIsFound) = false;

for (size\_t row = rowI; row < rows; row++)

{

if (fabs(merged[row][i]) >= EPSILON) {

double \* tmp = merged[rowI];

merged[rowI] = merged[row];

merged[row] = tmp;

\*(args->pivotIsFound) = true;

break;

}

}

}

pthread\_mutex\_unlock(&mutex);

barrier\_wait(args->barrier);

if (!(\*(args->pivotIsFound))) {

continue;

}

for (size\_t row = rowI + 1 + args->threadID; row < rows; row += args->threadNum)

{

if (fabs(merged[row][i]) < EPSILON) {

merged[row][i] = 0;

continue;

}

double toSubtract = merged[row][i] / merged[rowI][i];

merged[row][i] = 0;

for (size\_t col = i + 1; col < cols; col++)

{

merged[row][col] -= merged[rowI][col] \* toSubtract;

}

}

rowI++;

barrier\_wait(args->barrier);

}

if (rowI == 0) {

rowI = 1;

}

for (int i = rowI - 1; i >= 0; i--)

{

pthread\_mutex\_lock(&mutex);

if (args->threadID == 0) {

for (size\_t col = 0; col < cols; col++)

{

if (fabs(merged[i][col]) >= EPSILON) {

\*(args->colForReverse) = col;

break;

}

}

}

pthread\_mutex\_unlock(&mutex);

barrier\_wait(args->barrier);

int col = \*(args->colForReverse);

if (col == cols - 1) {

return NULL; // NO SOLUTION

}

pthread\_mutex\_lock(&mutex);

if (args->threadID == 0) {

merged[i][cols - 1] /= merged[i][col];

merged[i][col] = 1;

}

pthread\_mutex\_unlock(&mutex);

barrier\_wait(args->barrier);

for (size\_t row = args->threadID; row < i; row += args->threadNum)

{

merged[row][cols - 1] -= merged[row][col] \* merged[i][cols - 1];

merged[row][col] = 0;

}

barrier\_wait(args->barrier);

}

for (size\_t i = args->threadID; i < cols - 1; i += args->threadNum)

{

bool foundRow = false;

for (int row = i; row >= 0; row--)

{

if (merged[row][i] != 0) {

solution[i] = merged[row][cols - 1];

foundRow = true;

break;

}

}

if (!foundRow) {

solution[i] = 0;

}

}

}

void initMergedMatrix(ThreadArgs \*args) {

int rows = args->rows;

int cols = args->cols;

double \*\*matrix = args->matrix;

double \*b = args->b;

args->mergedMatrix = (double \*\*)malloc(sizeof(double \*) \* rows);

cols++;

for (size\_t i = 0; i < rows; i++)

{

args->mergedMatrix[i] = (double \*)malloc(sizeof(double) \* cols);

}

for (size\_t row = 0; row < rows; row++)

{

for (size\_t col = 0; col < cols - 1; col++)

{

args->mergedMatrix[row][col] = matrix[row][col];

}

}

for (size\_t row = 0; row < rows; row++)

{

args->mergedMatrix[row][cols - 1] = b[row];

}

}

double solve(ThreadArgs \*args) {

double \*\*matrix = args->matrix;

int rows = args->rows;

int cols = args->cols;

double \*b = args->b;

double \*solution = args->solution;

int threadNum = args->threadNum;

initMergedMatrix(args);

double \*\*mergedMatrix = args->mergedMatrix;

bool pivotIsFound = false;

int colForReverse = -1;

pthread\_t \*threads = (pthread\_t \*)malloc(threadNum \* sizeof(pthread\_t));

ThreadArgs \*threadArgs = (ThreadArgs \*)malloc(threadNum \* sizeof(ThreadArgs));

Barrier barrier;

pthread\_mutex\_init(&barrier.mutex, NULL);

pthread\_cond\_init(&barrier.cond, NULL);

barrier.count = 0;

barrier.total = threadNum;

struct timeval start, end;

gettimeofday(&start, NULL);

for (size\_t i = 0; i < threadNum; i++)

{

threadArgs[i] = (ThreadArgs) {

.matrix = matrix,

.rows = rows,

.cols = cols,

.b = b,

.solution = solution,

.threadNum = threadNum,

.threadID = i,

.mergedMatrix = mergedMatrix,

.pivotIsFound = &pivotIsFound,

.colForReverse = &colForReverse,

.barrier = &barrier,

};

pthread\_create(&threads[i], NULL, solveWithGauss, &threadArgs[i]);

}

for (size\_t i = 0; i < threadNum; i++)

{

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

}

gettimeofday(&end, NULL);

pthread\_mutex\_destroy(&barrier.mutex);

pthread\_cond\_destroy(&barrier.cond);

free(threadArgs);

free(threads);

for (size\_t k = 0; k < rows; k++)

{

free(mergedMatrix[k]);

}

free(mergedMatrix);

double elapsed = (end.tv\_sec - start.tv\_sec) + (end.tv\_usec - start.tv\_usec) / 1e6;

return elapsed;

}

void printUsage() {

const char msg[] = "Использование: ./main.out <количество потоков> <входной файл> <файл для вывода решения>\n";

write(STDERR\_FILENO, msg, strlen(msg));

}

void writeToConsole(const char \*msg) {

write(STDOUT\_FILENO, msg, strlen(msg));

}

void quit(const char \*msg) {

write(STDERR\_FILENO, msg, strlen(msg));

exit(EXIT\_FAILURE);

}

int main(int argc, char \*argv[]) {

if (argc != 4) {

printUsage();

exit(EXIT\_FAILURE);

}

char \*endptr;

long \_threadNum = strtol(argv[1], &endptr, 10);

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

printUsage();

exit(EXIT\_FAILURE);

}

if (\_threadNum < 1 || \_threadNum > MAX\_THREAD\_NUM) {

quit("Количество потоков должно быть в диапазоне от 1 до 5000 включительно\n");

}

int threadNum = (int)\_threadNum;

int32\_t file = open(argv[2], O\_RDONLY);

if (file == -1) {

quit("Ошибка: не удалось открыть входной файл\n");

}

int32\_t fileSol = open(argv[3], O\_WRONLY | O\_CREAT | O\_TRUNC | O\_APPEND, 0600);

if (fileSol == -1) {

quit("Ошибка: не удалось открыть файл для вывода решения\n");

}

struct stat st;

fstat(file, &st);

off\_t fsize = st.st\_size;

char \*buffer = malloc((size\_t)fsize + 1);

size\_t offset = 0;

while (offset < fsize) {

ssize\_t bytes\_read = read(file, buffer + offset, fsize - offset);

if (bytes\_read < 0) {

close(file);

free(buffer);

quit("Ошибка: не удалось прочитать входной файл\n");

}

if (bytes\_read == 0) break;

offset += (size\_t)bytes\_read;

}

close(file);

buffer[offset] = '\0';

long \_rows = strtol(buffer, &endptr, 10);

long \_cols = strtol(endptr, &endptr, 10);

int rows = (int)\_rows;

int cols = (int)\_cols;

double \*\* matrix = (double \*\*)malloc(sizeof(double \*) \* rows);

for (size\_t i = 0; i < rows; i++)

{

matrix[i] = (double \*)malloc(sizeof(double) \* cols);

}

double \*b = (double \*)malloc(sizeof(double) \* rows);

double \*solution = (double \*)malloc(sizeof(double) \* cols);

char \* p = endptr;

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

{

for (int j = 0; j < cols + 1; j++)

{

double val = strtod(p, &endptr);

if (p == endptr) {

quit("Ошибка: недостаточно чисел в файле");

}

if (j == cols)

b[i] = val;

else

matrix[i][j] = val;

p = endptr;

}

}

ThreadArgs args;

args.matrix = matrix;

args.b = b;

args.rows = rows;

args.cols = cols;

args.solution = solution;

args.threadNum = threadNum;

double time = solve(&args);

const char msg[] = "solution:\n";

write(fileSol, msg, strlen(msg));

if (solution == NULL) {

writeToConsole("No solution\n");

} else {

char buf[64];

for (size\_t i = 0; i < cols; i++)

{

int len = snprintf(buf, sizeof(buf), "%lf ", solution[i]);

write(fileSol, buf, len);

if ((i + 1) % 5 == 0) {

write(fileSol, "\n", 1);

}

}

}

char buf[1024];

int len = snprintf(buf, sizeof(buf), "Количество потоков: %d\n", threadNum);

write(STDOUT\_FILENO, buf, len);

len = snprintf(buf, sizeof(buf), "Размер массива: %dx%d\n", rows, cols);

write(STDOUT\_FILENO, buf, len);

len = snprintf(buf, sizeof(buf), "Затраченное время: %lfс\n", time);

write(STDOUT\_FILENO, buf, len);

close(fileSol);

}

**Протокол работы программы**

**Входные данные:** Большой массив и вектор свободных членов

**Выходные данные:**

Количество потоков: 1

Размер массива: 3000x3000

Затраченное время: 25.478508с

**—--------------------------------------**

Количество потоков: 2

Размер массива: 3000x3000

Затраченное время: 14.226613с

**—--------------------------------------**

Количество потоков: 4

Размер массива: 3000x3000

Затраченное время: 8.039504с

**—--------------------------------------**

Количество потоков: 4

Размер массива: 5000x5000

Затраченное время: 33.592541с

**—--------------------------------------**

Количество потоков: 8

Размер массива: 5000x5000

Затраченное время: 29.023236с

**Вывод**

При выполнении данной лабораторной работы я научился работать с потоками. Увидел на практике, как контролировать поведение потоков мьютексами и барьерами. Получены следующие таблица и графики измерений:

|  |  |  |  |
| --- | --- | --- | --- |
| **Число потоков** | **Время исполнения (мс)** | **Ускорение** | **Эффективность** |
| 1 | 25478.508 | 1.00 | 1.00 |
| 2 | 14226.613 | 1.79 | 0.90 |
| 4 | 8039.504 | 3.17 | 0.79 |
| 8 | 9678.120 | 2.63 | 0.33 |
| 16 | 11225.562 | 2.27 | 0.14 |
| 32 | 17155.895 | 1.49 | 0.047 |
| 64 | 30009.268 | 0.85 | 0.013 |
| 128 | 52758.620 | 0.48 | 0.004 |
| 256 | 97038.930 | 0.26 | 0.001 |

Ускорение: T1 / Tn, где T1 - время выполнения одним потоком, Tn - время выполнения с n потоками

Эффективность: Ускорение / n, где n - количество потоков

**Анализ результатов:**

1. **Количество потоков меньше логических ядер процессора (1-8)**

**При увеличении числа потоков от 1 до 4 наблюдается ускорение:**

* Потоки получают отдельные физические ядра и выполняются параллельно.
* Потери времени минимальны, влияние синхронизации и мьютексов не так значительно
* Накладные расходы минимальны

**При чисел потоков от 4 до 16 наблюдается замедление:**

* Синхронизация с мьютексами и барьерами замедляет работу
* Накладные расходы превышают пользу от увеличенного количества потоков.

**Вывод:**

При небольшом числе потоков, меньшем количества логических и физических ядер, наблюдается максимальная эффективность от их использования. Соблюдается баланс между накладными расходами и пользой от параллельной работы с данными

1. **Количество потоков равно числу логических ядер (16)**

**При переходе от 4 к 16 потокам ускорение падает:**

* Логические ядра заняты, но прироста нет из-за сильной синхронизации
* Большое количество накладных расходов.
* Тем не менее время исполнения меньше, чем при последовательной работе

**Вывод:**

При числе потоков, равном логическим ядрам, наблюдается снижение эффективности по сравнению с меньшим количеством потоков. Накладные расходы от дополнительных потоков превышают их пользу

1. **Количество потоков больше логических ядер(32-256)**

* **При 32 потоках ускорение снижается, но скорость все еще выше, чем при однопоточной работе**
* **При 64 потоках и выше программа начинает работать еще медленнее, чем при одном потоке:**
  + ОС вынуждена постоянно переключать контекст между большим количеством потоков
  + Синхронизация большого количества потоков сильно замедляет код
  + Накладные расходы на работу с потоками превышают пользу от них.

**Вывод:**

При числе потоков больше количества логических ядер производительность снижается. Система перегружена планированием и синхронизацией потоков, и их слишком много для эффективной параллельной работы.