Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

**Лабораторная работа №2 по курсу**

**«Операционные системы»**

Студент: Бурдинский Владислав Дмитриевич

Группа: М8О–203Б–23

Вариант: 12

Преподаватель: Миронов Евгений Сергеевич

Оценка: \_\_\_\_\_\_\_\_\_\_

Дата: \_\_\_\_\_\_\_\_\_\_

Подпись: \_\_\_\_\_\_\_\_\_\_

Москва, 2024.

**Постановка задачи**

**Цель работы**

Приобретение практических навыков в:

* Управление процессами в ОС
* Обеспечение обмена данных между процессами посредством каналов

**Задание**

Родительский процесс создает дочерний процесс. Предопределены процессы ввода-вывода, родительский процесс получает три числа и переводит их на ввод дочернему процессу. Дочерний процесс осуществляет деление число1 / число2 / число3 . Затем он возвращает результат родителю.

**Вариант задания**

**12 вариант) Деление число1 / число2 / число3**

**Код программы**

**my\_solution.cpp**

#include "../include/my\_solution.h"

const int KERNEL\_SIZE = 3;

pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;

std::vector<std::vector<int>> filter = {

{1, 1, 1},

{1, 1, 1},

{1, 1, 1}

};

// Эрозия матрицы

void\* matrix\_erosion(void\* arg){

ThreadData\* data = static\_cast<ThreadData\*>(arg);

const std::vector<std::vector<int>>& matrix = \*(data -> matrix);

std::vector<std::vector<int>>& result\_matrix = \*(data -> result\_matrix);

int m\_rows = data -> rows;

int m\_cols = data -> cols;

std::cout << "Поток обрабатывает строки с " << data->start\_row

<< " по " << data->end\_row << std::endl;

for (int i = data -> start\_row; i <= data -> end\_row; ++i){

for (int j = 0; j < m\_cols; ++j){

bool flag = false;

for (int ki = -KERNEL\_SIZE / 2; ki <= KERNEL\_SIZE / 2; ++ki){

for (int kj = -KERNEL\_SIZE / 2; kj <= KERNEL\_SIZE / 2; ++kj){

int ni = i + ki;

int nj = j + kj;

if (ni >= 0 && nj >= 0 && ni < m\_rows && nj < m\_cols){

if (filter[ki + KERNEL\_SIZE / 2][kj + KERNEL\_SIZE / 2] != matrix[ni][nj]){

flag = true;

}

}

}

}

pthread\_mutex\_lock(&mtx);

if (flag == true){

result\_matrix[i][j] = 0;

}else{

result\_matrix[i][j] = 1;

}

pthread\_mutex\_unlock(&mtx);

}

}

return nullptr;

}

// Наращивание матрицы

void\* matrix\_dilatation(void\* arg){

ThreadData\* data = static\_cast<ThreadData\*>(arg);

const std::vector<std::vector<int>>& matrix = \*(data -> matrix);

std::vector<std::vector<int>>& result\_matrix = \*(data -> result\_matrix);

int m\_rows = data -> rows;

int m\_cols = data -> cols;

for (int i = data -> start\_row; i <= data -> end\_row; ++i){

for (int j = 0; j < m\_cols; ++j){

bool flag = false;

for (int ki = -KERNEL\_SIZE / 2; ki <= KERNEL\_SIZE / 2; ++ki){

for (int kj = -KERNEL\_SIZE / 2; kj <= KERNEL\_SIZE / 2; ++kj){

int ni = i + ki;

int nj = j + kj;

if (ni >= 0 && nj >= 0 && ni < m\_rows && nj < m\_cols){

if ((filter[ki + KERNEL\_SIZE / 2][kj + KERNEL\_SIZE / 2] == 1) && (matrix[ni][nj] == 1)){

flag = true;

break;

}

}

}

if (flag) break;

}

// pthread\_mutex\_lock(&mtx);

if (flag == true){

result\_matrix[i][j] = 1;

}else{

result\_matrix[i][j] = 0;

}

// pthread\_mutex\_unlock(&mtx);

}

}

return nullptr;

}

my\_solution.h

#pragma once

#include <iostream>

#include <vector>

#include <pthread.h>

#include <unistd.h>

#include <getopt.h>

#include <ctime>

#include <cstdlib>

struct ThreadData {

const std::vector<std::vector<int>>\* matrix;

std::vector<std::vector<int>>\* result\_matrix;

int start\_row;

int end\_row;

int width;

int rows;

int cols;

};

extern const int KERNEL\_SIZE;

extern pthread\_mutex\_t mtx;

extern std::vector<std::vector<int>> filter;

void\* matrix\_erosion(void\* arg);

void\* matrix\_dilatation(void\* arg);

void perform\_erosion(const std::vector<std::vector<int>>& input\_matrix,

std::vector<std::vector<int>>& output\_matrix,

const std::vector<std::vector<int>>& filter);

void perform\_dilation(const std::vector<std::vector<int>>& input\_matrix,

std::vector<std::vector<int>>& output\_matrix,

const std::vector<std::vector<int>>& filter);

test.cpp

#include <gtest/gtest.h>

#include <pthread.h>

#include <vector>

#include <chrono>

#include <cstdlib>

#include <ctime>

#include "../include/my\_solution.h"

extern const int KERNEL\_SIZE;

extern pthread\_mutex\_t mtx;

extern std::vector<std::vector<int>> filter;

void\* matrix\_erosion(void\* arg);

void\* matrix\_dilatation(void\* arg);

void singleThreadErosion(const std::vector<std::vector<int>>& input\_matrix, std::vector<std::vector<int>>& output\_matrix) {

ThreadData data;

data.start\_row = 0;

data.end\_row = input\_matrix.size() - 1;

data.rows = input\_matrix.size();

data.cols = input\_matrix[0].size();

data.width = input\_matrix[0].size();

data.matrix = &input\_matrix;

data.result\_matrix = &output\_matrix;

matrix\_erosion(static\_cast<void\*>(&data));

}

void singleThreadDilatation(const std::vector<std::vector<int>>& input\_matrix, std::vector<std::vector<int>>& output\_matrix) {

ThreadData data;

data.start\_row = 0;

data.end\_row = input\_matrix.size() - 1;

data.rows = input\_matrix.size();

data.cols = input\_matrix[0].size();

data.width = input\_matrix[0].size();

data.matrix = &input\_matrix;

data.result\_matrix = &output\_matrix;

matrix\_dilatation(static\_cast<void\*>(&data));

}

void multiThreadErosion(const std::vector<std::vector<int>>& input\_matrix, std::vector<std::vector<int>>& output\_matrix, int num\_threads) {

int matrix\_size = input\_matrix.size();

int rows\_per\_thread = matrix\_size / num\_threads;

int remainder\_rows = matrix\_size % num\_threads;

std::vector<pthread\_t> threads(num\_threads);

std::vector<ThreadData> thread\_data(num\_threads);

int current\_row = 0;

for (int i = 0; i < num\_threads; ++i) {

thread\_data[i].start\_row = current\_row;

thread\_data[i].end\_row = current\_row + rows\_per\_thread - 1;

if (i < remainder\_rows) {

thread\_data[i].end\_row += 1;

}

thread\_data[i].rows = matrix\_size;

thread\_data[i].cols = input\_matrix[0].size();

thread\_data[i].width = input\_matrix[0].size();

thread\_data[i].matrix = &input\_matrix;

thread\_data[i].result\_matrix = &output\_matrix;

pthread\_create(&threads[i], NULL, matrix\_erosion, &thread\_data[i]);

current\_row = thread\_data[i].end\_row + 1;

}

for (int i = 0; i < num\_threads; ++i) {

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

}

}

void multiThreadDilatation(const std::vector<std::vector<int>>& input\_matrix, std::vector<std::vector<int>>& output\_matrix, int num\_threads) {

int matrix\_size = input\_matrix.size();

int rows\_per\_thread = matrix\_size / num\_threads;

int remainder\_rows = matrix\_size % num\_threads;

std::vector<pthread\_t> threads(num\_threads);

std::vector<ThreadData> thread\_data(num\_threads);

int current\_row = 0;

for (int i = 0; i < num\_threads; ++i) {

thread\_data[i].start\_row = current\_row;

thread\_data[i].end\_row = current\_row + rows\_per\_thread - 1;

if (i < remainder\_rows) {

thread\_data[i].end\_row += 1;

}

thread\_data[i].rows = matrix\_size;

thread\_data[i].cols = input\_matrix[0].size();

thread\_data[i].width = input\_matrix[0].size();

thread\_data[i].matrix = &input\_matrix;

thread\_data[i].result\_matrix = &output\_matrix;

pthread\_create(&threads[i], NULL, matrix\_dilatation, &thread\_data[i]);

current\_row = thread\_data[i].end\_row + 1;

}

for (int i = 0; i < num\_threads; ++i) {

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

}

}

TEST(MatrixErosionTest, test\_1) {

std::vector<std::vector<int>> input\_matrix = {

{1, 1, 1, 1, 1},

{1, 1, 1, 0, 0},

{1, 1, 1, 0, 0},

{1, 0, 0, 0, 0},

{0, 0, 0, 0, 0}

};

std::vector<std::vector<int>> expected\_output = {

{1, 1, 0, 0, 0},

{1, 1, 0, 0, 0},

{0, 0, 0, 0, 0},

{0, 0, 0, 0, 0},

{0, 0, 0, 0, 0}

};

std::vector<std::vector<int>> output\_matrix(input\_matrix.size(), std::vector<int>(input\_matrix[0].size()));

singleThreadErosion(input\_matrix, output\_matrix);

EXPECT\_EQ(output\_matrix, expected\_output);

}

TEST(MatrixDilatationTest, test\_dop) {

std::vector<std::vector<int>> input\_matrix = {

{1, 1, 1, 1, 1},

{1, 1, 1, 0, 0},

{1, 1, 1, 0, 0},

{1, 0, 0, 0, 0},

{0, 0, 0, 0, 0}

};

std::vector<std::vector<int>> expected\_output = {

{1, 1, 1, 1, 1},

{1, 1, 1, 1, 1},

{1, 1, 1, 1, 0},

{1, 1, 1, 1, 0},

{1, 1, 0, 0, 0}

};

std::vector<std::vector<int>> output\_matrix(input\_matrix.size(), std::vector<int>(input\_matrix[0].size()));

singleThreadDilatation(input\_matrix, output\_matrix);

EXPECT\_EQ(output\_matrix, expected\_output);

}

TEST(MatrixErosionTest, test\_2) {

const int size = 100;

std::vector<std::vector<int>> input\_matrix(size, std::vector<int>(size));

srand(static\_cast<unsigned int>(time(NULL)));

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

for (int j = 0; j < size; ++j) {

input\_matrix[i][j] = rand() % 2;

}

}

std::vector<std::vector<int>> single\_thread\_output(size, std::vector<int>(size));

std::vector<std::vector<int>> multi\_thread\_output(size, std::vector<int>(size));

singleThreadErosion(input\_matrix, single\_thread\_output);

for (int num\_threads = 2; num\_threads <= 8; num\_threads \*= 2) {

multiThreadErosion(input\_matrix, multi\_thread\_output, num\_threads);

EXPECT\_EQ(single\_thread\_output, multi\_thread\_output);

}

}

TEST(MatrixErosionTest, test\_3) {

const int size = 1000;

std::vector<std::vector<int>> input\_matrix(size, std::vector<int>(size));

srand(static\_cast<unsigned int>(time(NULL)));

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

for (int j = 0; j < size; ++j) {

input\_matrix[i][j] = rand() % 2;

}

}

std::vector<std::vector<int>> single\_thread\_output(size, std::vector<int>(size));

std::vector<std::vector<int>> multi\_thread\_output(size, std::vector<int>(size));

auto start\_single = std::chrono::high\_resolution\_clock::now();

singleThreadErosion(input\_matrix, single\_thread\_output);

auto end\_single = std::chrono::high\_resolution\_clock::now();

auto duration\_single = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_single - start\_single).count();

int num\_threads = 4;

auto start\_multi = std::chrono::high\_resolution\_clock::now();

multiThreadErosion(input\_matrix, multi\_thread\_output, num\_threads);

auto end\_multi = std::chrono::high\_resolution\_clock::now();

auto duration\_multi = std::chrono::duration\_cast<std::chrono::milliseconds>(end\_multi - start\_multi).count();

EXPECT\_EQ(single\_thread\_output, multi\_thread\_output);

EXPECT\_LT(duration\_multi, duration\_single);

std::cout << "Время однопоточного выполнения: " << duration\_single << " мс" << std::endl;

std::cout << "Время многопоточного выполнения: " << duration\_multi << " мс" << std::endl;

}

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

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

Main.cpp

#include "include/my\_solution.h"

#include <chrono>

std::vector<std::vector<int>> matrix;

std::vector<std::vector<int>> result\_matrix;

int max\_threads = 4;

std::string mode = "erosion";

int matrix\_size = 100;

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

int opt;

while ((opt = getopt(argc, argv, "t:m:n:")) != -1) {

switch (opt) {

case 't':

max\_threads = atoi(optarg);

break;

case 'm':

mode = optarg;

break;

case 'n':

matrix\_size = atoi(optarg);

break;

default:

std::cerr << "Использование: " << argv[0] << " -t max\_threads -m mode (erosion/dilation)\n";

exit(EXIT\_FAILURE);

}

}

matrix.resize(matrix\_size, std::vector<int>(matrix\_size));

result\_matrix.resize(matrix\_size, std::vector<int>(matrix\_size));

srand(time(NULL));

for (int i = 0; i < matrix\_size; ++i) {

for (int j = 0; j < matrix\_size; ++j) {

matrix[i][j] = rand() % 2;

}

}

std::cout << "Матрица стартовая:" << std::endl;

for (int i = 0; i < 10; ++i) {

for (int j = 0; j < 10; ++j) {

std::cout << matrix[i][j] << " ";

}

std::cout << std::endl;

}

int num\_threads = max\_threads < matrix\_size ? max\_threads : matrix\_size;

int rows\_per\_thread = matrix\_size / num\_threads;

std::vector<pthread\_t> threads(num\_threads);

std::vector<ThreadData> thread\_data(num\_threads);

int current\_row = 0;

auto start\_time = std::chrono::high\_resolution\_clock::now();

for (int i = 0; i < num\_threads; ++i) {

thread\_data[i].start\_row = current\_row;

if (i == num\_threads - 1) {

thread\_data[i].end\_row = matrix\_size - 1;

} else {

thread\_data[i].end\_row = current\_row + rows\_per\_thread - 1;

}

thread\_data[i].rows = matrix\_size;

thread\_data[i].cols = matrix\_size;

thread\_data[i].width = matrix\_size;

thread\_data[i].matrix = &matrix;

thread\_data[i].result\_matrix = &result\_matrix;

if (mode == "erosion") {

pthread\_create(&threads[i], NULL, matrix\_erosion, &thread\_data[i]);

} else if (mode == "dilatation") {

pthread\_create(&threads[i], NULL, matrix\_dilatation, &thread\_data[i]);

}

current\_row = thread\_data[i].end\_row + 1;

}

for (int i = 0; i < num\_threads; ++i) {

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

}

auto end\_time = std::chrono::high\_resolution\_clock::now();

// Вычисление затраченного времени

std::chrono::duration<double> elapsed = end\_time - start\_time;

std::cout << "Результирующая матрица:" << std::endl;

for (int i = 0; i < 10; ++i) {

for (int j = 0; j < 10; ++j) {

std::cout << result\_matrix[i][j] << " ";

}

std::cout << std::endl;

}

std::cout << "Время выполнения операции: " << elapsed.count() << " секунд" << std::endl;

return 0;

}

CMakeLists.txt

cmake\_minimum\_required(VERSION 3.10)

project(lab-2)

# Устанавливаем стандарт C++

set(CMAKE\_CXX\_STANDARD 11)

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

# Включаем директорию include/

include\_directories(include)

# Добавляем исходные файлы

set(SOURCES

src/my\_solution.cpp

main.cpp

)

# Основной исполняемый файл

add\_executable(main ${SOURCES})

target\_link\_libraries(main pthread)

# Подключаем Google Test через FetchContent

include(FetchContent)

FetchContent\_Declare(

googletest

URL https://github.com/google/googletest/archive/refs/tags/release-1.12.1.zip

)

# Предотвращаем переопределение настроек компилятора/линкера родительского проекта

set(gtest\_force\_shared\_crt ON CACHE BOOL "" FORCE)

FetchContent\_MakeAvailable(googletest)

# Добавляем тесты

add\_executable(run\_tests tests/test.cpp src/my\_solution.cpp)

target\_link\_libraries(run\_tests gtest pthread)

# Добавляем пути к заголовочным файлам Google Test

target\_include\_directories(run\_tests PRIVATE

${gtest\_SOURCE\_DIR}/include

${gtest\_SOURCE\_DIR}

)

**Пример работы**

(base) vladislavburdinskij@MacBook-Pro-Vladislav build % ./main

Матрица стартовая:

0 1 1 1 1 0 0 1 0 0

0 0 1 0 0 1 0 0 0 0

0 1 0 1 1 1 0 1 1 0

0 1 1 0 1 0 1 1 1 1

0 1 0 0 0 1 1 1 0 1

0 0 1 1 0 1 0 0 0 1

0 1 1 1 0 1 0 0 0 1

1 0 1 0 1 1 1 0 0 0

1 1 0 1 1 0 1 0 1 0

1 1 1 0 1 1 0 0 0 1

Поток обрабатывает строки с 0 по 24

Поток обрабатывает строки с Поток обрабатывает строки с 75 по 99

50 по 74

Поток обрабатывает строки с 25 по 49

Результирующая матрица:

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0

Время выполнения операции: 0.00310342 секунд

(base) vladislavburdinskij@MacBook-Pro-Vladislav build %

**Вывод**

В данной лабораторной удалось познакомиться с таким системным вызовом как fork() для создания новых процессов и pipe который служит для их связи этих процессов. Эти команды могут значительно ускорить многие процессы в обработке данных и упростить жизнь при проектировании системы.