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.