Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина «Операционные среды и системное программирование»

ОТЧЕТ
к лабораторной работе №4
на тему:
«ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ (ПОТОКОВ): ВЗАИМНОЕ ИСКЛЮЧЕНИЕ И СИНХРОНИЗАЦИЯ»
БГУИР 6-05-0612-02 67

Выполнил студент группы 353503
КОХАН Артём Игоревич

_____
(дата, подпись студента)

Проверил ассистент каф. информатики
Гриценко Никита Юрьевич

_____
(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 ИНДИВИДУАЛЬНОЕ ЗАДАНИЕ

Приложение, демонстрирующее параллельную согласованную работу процессов (потоков) и их взаимодействие. Анализ корректности (отсутствия коллизий). Оценка эффективности механизмов синхронизации (ISO). Специальных требований к приложениям не предъявляется; в частности, во многих случаях они могут быть не обязательно оконными, но также и консольными.

Реализация модели взаимодействия процессов (потоков) «писатели читатели» с возможностью параметризации и наглядного (не обязательно графического) представления результатов. Поскольку речь идет о модели, реальные данные не обязательны, можно ограничиться моделированием обращений к ним (обращение характеризуется параметрами запроса, моментом обращения, длительностью исполнения). Но можно реализовать и макет системы, использующей реальные (тестовые, «учебные») данные в соответствии с этой моделью. Таким образом, вместо моделирования обращений с заданными характеристиками могут быть сами обращения к «настоящим» данным (прикладные функции системы можно позаимствовать из предыдущих работ). Обеспечение корректного функционирования, т.е. избегание как «грязного» считывания данных и одновременно минимизация блокировок. Изменяемые параметры модели: количество «писателей» и «читателей»; интенсивность их обращений к ресурсам, длительность использования ресурса, размер блока данных и т.п. (характеристики случайных величин при моделировании); штрафы за простой участников; величина тайм-аутов и др. Результаты моделирования: соотношение времени активности/блокировки участников; соотношение успешных/неуспешных обращений; общая эффективность (пропускная способность) по «записи» и «чтению» и др.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Синхронизация и взаимное исключение в многопоточных приложениях являются фундаментальными для обеспечения корректной работы параллельных процессов, особенно при доступе к общим ресурсам. Основные механизмы, такие как мьютексы, семафоры, критические секции и события, предотвращают ошибки, вызванные одновременным доступом к критическим ресурсам, и обеспечивают последовательное выполнение взаимосвязанных операций. Мьютексы (mutex) представляют собой объекты ядра, которые могут использоваться для синхронизации потоков в разных процессах, обеспечивая эксклюзивный доступ к общим ресурсам [1].

Критические секции (critical sections) обеспечивают более легковесный механизм синхронизации по сравнению с мьютексами, но работают только в пределах одного процесса. Они гарантируют, что только один поток может выполнять определенный участок кода, который взаимодействует с общим ресурсом, что особенно эффективно для кратковременных блокировок [2]. Семафоры расширяют эту концепцию, позволяя ограниченному числу потоков одновременно получать доступ к ресурсу. Объекты ожидания, такие как события и таймеры, используются для координации действий между потоками, например, с помощью функции WaitForSingleObject, которая позволяет потоку ожидать сигнала от другого объекта синхронизации перед продолжением выполнения [3].

Модель "Писатели-читатели" демонстрирует типичные сложности, возникающие при организации доступа к общим ресурсам. В этой модели несколько потоков-читателей могут одновременно обращаться к данным, в то время как потоки-писатели требуют эксклюзивного доступа. Конфликты возникают при необходимости обеспечения целостности данных и предотвращения "грязного" чтения, когда читатель может получить неконсистентные данные в процессе их модификации писателем. Решение задачи включает разработку стратегии, которая позволяет избежать взаимных блокировок и гарантировать равный доступ к ресурсам.

Моделирование этой системы включает параметризацию количества читателей и писателей, логики разрешения конфликтов, интенсивности доступа и продолжительности использования ресурсов. Использование механизмов синхронизации, таких как мьютексы и критические секции, позволяет эффективно управлять доступом к разделяемым данным. Результаты такого моделирования могут быть использованы для оценки времени активности и блокировки, эффективности использования ресурсов и устранения взаимных блокировок, что позволяет значительно повысить общую производительность и надежность системы. Функции ожидания, такие как WaitForSingleObject, играют ключевую роль в организации эффективной синхронизации между потоками.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

Программа реализует классическую модель синхронизации "Писатели-читатели" с использованием механизмов взаимного исключения и условных переменных. Архитектура программы состоит из нескольких ключевых компонентов, обеспечивающих корректную работу многопоточной системы.

Основным классом программы является ReaderWriter, который инкапсулирует логику управления доступом к разделяемым данным. Класс содержит следующие приватные поля: мьютекс для защиты критических секций, две условные переменные для координации читателей и писателей, строку с разделяемыми данными, счетчики активных и ожидающих потоков, а также атомарные переменные для сбора статистики работы системы.

Функция SafePrint обеспечивает потокобезопасный вывод сообщений в консоль с использованием мьютекса cout_mutex. Это предотвращает перемешивание вывода от разных потоков и обеспечивает читаемость логов выполнения программы.

Метод read_data реализует логику работы потоков-читателей. При входе в метод поток захватывает мьютекс и проверяет условие доступа – отсутствие активных писателей и ожидающих писателей. Если условие не выполняется, поток блокируется на условной переменной reader_cv. После получения доступа поток увеличивает счетчик активных читателей и освобождает мьютекс, позволяя другим читателям также получить доступ к данным. Далее выполняется имитация чтения данных длительностью 50 миллисекунд, после чего поток снова захватывает мьютекс для уменьшения счетчика активных читателей. Если после этого не остается активных читателей, пробуждается один из ожидающих писателей через условную переменную writer_cv.

Метод write_data обеспечивает эксклюзивный доступ потоков-писателей к разделяемым данным. Поток-писатель при входе в метод увеличивает счетчик ожидающих писателей и проверяет условие доступа – отсутствие активных читателей и писателей. Если условие не выполняется, поток блокируется на условной переменной writer_cv. После получения доступа поток уменьшает счетчик ожидающих писателей и увеличивает счетчик активных писателей. Далее выполняется имитация записи данных длительностью 100 миллисекунд, обновляется содержимое разделяемых данных с указанием идентификатора писателя. После завершения записи поток уменьшает счетчик активных писателей и в зависимости от наличия ожидающих писателей либо пробуждает одного писателя, либо всех ожидающих читателей.

Метод try_read_data предоставляет альтернативный механизм чтения с ограничением времени ожидания. Поток пытается получить доступ к данным в течение заданного таймаута (по умолчанию 100 миллисекунд). Если доступ не получен в течение указанного времени, метод возвращает false и увеличивает счетчик неудачных попыток чтения. Это позволяет избежать бесконечного блокирования потоков-читателей в условиях высокой конкуренции за ресурсы.

Метод print_statistics формирует и выводит подробную статистику работы системы, включая количество успешных и неудачных операций чтения и записи, общее количество операций и соотношение чтений к записям. Статистика собирается с использованием атомарных переменных, что обеспечивает корректность данных при параллельном доступе.

Вспомогательные функции reader_thread, writer_thread и timed_reader_thread реализуют логику работы потоков-читателей и писателей. Каждая функция выполняет заданное количество итераций, между которыми добавляются случайные задержки для имитации реальной нагрузки. Функции используют генератор случайных чисел для создания разнообразных паттернов доступа к данным.

Функция main является точкой входа в программу и организует процесс тестирования модели. Создается экземпляр класса ReaderWriter, запускаются потоки читателей и писателей в различных конфигурациях: 3 обычных читателя, 2 писателя и 2 читателя с таймаутом. Каждый поток выполняет по 5 итераций работы. После завершения работы всех потоков выводится статистика и анализ эффективности работы системы, включая расчет процента успешных операций и диагностику возможных проблем с блокировками.

Программа демонстрирует эффективное использование механизмов синхронизации стандартной библиотеки C++ для решения классической проблемы параллельного программирования и обеспечивает наглядное представление процессов взаимодействия потоков при доступе к общим ресурсам.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Программа была запущена в среде разработки для тестирования алгоритма читателей и писателей. После запуска произошла инициализация необходимых структур данных – мьютексов и семафоров для синхронизации доступа к разделяемому ресурсу. Затем были созданы и запущены потоки двух типов: потоки-читатели и потоки-писатели. Количество потоков каждого типа задавалось параметрами запуска. Все потоки начали параллельное выполнение, конкурируя за доступ к общему ресурсу. В процессе работы каждый поток пытался выполнить свои операции чтения или записи в соответствии с логикой алгоритма. Программа отслеживала время выполнения операций и фиксировала случаи, когда потоки не могли получить доступ к ресурсу в течение заданного времени ожидания. Работа программы продолжалась до тех пор, пока все потоки не завершили выполнение запланированных операций.

```
=== STATISTICS ===
Successful reads: 22
Successful writes: 10
Failed reads: 3
Failed writes: 0
Total operations: 32
Readers/Writers ratio: 2.2

=== PERFORMANCE ANALYSIS ===
Success rate: 91.4286%
Note: Some reads failed due to timeouts (reader starvation possible)
```

Рисунок 4.1 – Результат выполнения программы

4.2 Описание работы и результатов

В ходе выполнения программы было зафиксировано 32 операции с разделяемым ресурсом. Статистика показывает, что 22 операции чтения и 10 операций записи завершились успешно. Однако 3 операции чтения завершились неудачно из-за превышения времени ожидания. Операции записи всегда выполнялись успешно. Общий показатель успешности операций составил 91.43%. Соотношение количества операций чтения к операциям записи равнялось 2.2. Анализ результатов указывает на проблему голодания читателей – ситуацию, когда потоки-читатели не могут получить доступ к ресурсу из-за приоритетного доступа писателей. Это проявляется в том, что при активной работе писателей некоторые читатели не успевают получить доступ к ресурсу в установленный таймаут. Результаты демонстрируют характерное поведение алгоритма с приоритетом для писателей, где обеспечивается целостность данных при записи, но может страдать доступность ресурса для чтения при высокой конкурентной нагрузке.

ВЫВОД

В ходе выполнения лабораторной работы была успешно разработана и протестирована многопоточная система, реализующая классическую задачу синхронизации "Читатели и писатели". Программа продемонстрировала работу с общим разделяемым ресурсом в условиях конкурентного доступа со стороны потоков-читателей и потоков-писателей.

Результаты тестирования показали, что разработанный алгоритм синхронизации обеспечивает корректный доступ к разделяемым данным – все операции записи завершились успешно, что подтверждает соблюдение условия взаимоисключения для писателей. Однако анализ статистики выполнения выявил характерную проблему данной модели: наличие 3 неудачных операций чтения из-за превышения времени ожидания свидетельствует о ситуации "голодания читателей". Это объясняется реализованным в алгоритме приоритетом для потоков-писателей.

Практическая ценность работы заключается в приобретении опыта работы с механизмами синхронизации в многопоточной среде, включая мьютексы и семафоры. Полученные результаты наглядно демонстрируют важность балансировки приоритетов доступа при проектировании многопоточных приложений и необходимость учета компромисса между эффективностью использования ресурса и справедливостью распределения доступа между потоками различного типа. Разработанная программа может служить основой для исследования более сложных алгоритмов синхронизации, обеспечивающих предотвращение голодания потоков.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] WinAPI: CreateMutex [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi createmutexa. – Дата доступа: 31.10.2025.

[2] API Win32 Critical section [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/sync/critical-section-objects. Дата доступа: 31.10.2025.

[3] WaitForSingleObject [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject. – Дата доступа: 31.10.2025.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

```
#include <iostream>
#include <thread>
#include <mutex>
#include <vector>
#include <chrono>
#include <random>
#include <atomic>
#include <condition_variable>
#include <string>
#include <sstream>

// Мьютекс для синхронизации вывода
std::mutex cout_mutex;

void SafePrint(const std::string& message) {
    std::lock_guard<std::mutex> lock(cout_mutex);
    std::cout << message << std::endl;
}

class ReaderWriter {
private:
    std::mutex mutex_;                  // Основной мьютекс для данных
    std::condition_variable reader_cv_; // Условная переменная для читателей
    std::condition_variable writer_cv_; // Условная переменная для писателей
    std::string shared_data;            // Разделяемые данные

    // Счетчики для управления доступом
    int active_readers;
    int active_writers;
    int waiting_writers;

    // Статистика
    std::atomic<int> successful_reads;
    std::atomic<int> successful_writes;
    std::atomic<int> failed_reads;
    std::atomic<int> failed_writes;
    std::atomic<int> total_reads;
    std::atomic<int> total_writes;

public:
    ReaderWriter() : shared_data("Initial Data"),
        active_readers(0),
        active_writers(0),
        waiting_writers(0),
        successful_reads(0),
        successful_writes(0),
        failed_reads(0),
        failed_writes(0),
        total_reads(0),
        total_writes(0) {
    }

    // Чтение данных (могут работать несколько читателей одновременно)
    void read_data(int reader_id) {
        auto start_time = std::chrono::steady_clock::now();

        {
            std::unique_lock<std::mutex> lock(mutex_);
```

```
            // Ждем, пока нет активных писателей
            while (active_writers > 0 || waiting_writers > 0) {
                reader_cv_.wait(lock);
            }

            active_readers++;
        }

        // Критическая секция для чтения
        // Имитация времени чтения
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        auto end_time = std::chrono::steady_clock::now();
        auto                              duration                              =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

        // Безопасный вывод
        std::stringstream ss;
        ss << "Reader " << reader_id << " read: '" << shared_data
            << "' | Active readers: " << active_readers
            << " | Wait time: " << duration.count() << "ms";
        SafePrint(ss.str());

        {
            std::unique_lock<std::mutex> lock(mutex_);
            active_readers--;

            // Если это был последний читатель, будим писателей
            if (active_readers == 0) {
                writer_cv_.notify_one();
            }
        }

        successful_reads++;
        total_reads++;
    }

    // Запись данных (только один писатель одновременно)
    void write_data(int writer_id, const std::string& new_data) {
        auto start_time = std::chrono::steady_clock::now();

        {
            std::unique_lock<std::mutex> lock(mutex_);
            waiting_writers++;

            // Ждем, пока нет активных читателей и писателей
            while (active_readers > 0 || active_writers > 0) {
                writer_cv_.wait(lock);
            }

            waiting_writers--;
            active_writers++;
        }

        // Критическая секция для записи
        // Имитация времени записи
        std::this_thread::sleep_for(std::chrono::milliseconds(100));

        // Безопасное преобразование writer_id в строку
        std::stringstream ss_id;
        ss_id << writer_id;
        shared_data = new_data + " (written by " + ss_id.str() + ")";
```

```
        auto end_time = std::chrono::steady_clock::now();
        auto                              duration                              =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

        // Безопасный вывод
        std::stringstream ss;
        ss << "Writer " << writer_id << " wrote: '" << shared_data
            << "' | Wait time: " << duration.count() << "ms";
        SafePrint(ss.str());

        {
            std::unique_lock<std::mutex> lock(mutex_);
            active_writers--;

            // Будим всех ожидающих
            if (waiting_writers > 0) {
                writer_cv_.notify_one();
            }
            else {
                reader_cv_.notify_all();
            }
        }

        successful_writes++;
        total_writes++;
    }

    // Попытка чтения с таймаутом
    bool try_read_data(int reader_id, int timeout_ms = 100) {
        auto start_time = std::chrono::steady_clock::now();
        bool success = false;

        {
            std::unique_lock<std::mutex> lock(mutex_);

            // Пытаемся получить доступ в течение timeout_ms
            auto timeout_time = std::chrono::steady_clock::now() +
                std::chrono::milliseconds(timeout_ms);

            while ((active_writers > 0 || waiting_writers > 0) &&
                std::chrono::steady_clock::now() < timeout_time) {
                reader_cv_.wait_until(lock, timeout_time);
            }

            if (active_writers == 0 && waiting_writers == 0) {
                active_readers++;
                success = true;
            }
        }

        if (success) {
            // Критическая секция для чтения
            std::this_thread::sleep_for(std::chrono::milliseconds(50));

            auto end_time = std::chrono::steady_clock::now();
            auto                              duration                              =
std::chrono::duration_cast<std::chrono::milliseconds>(end_time - start_time);

            // Безопасный вывод
            std::stringstream ss;
            ss << "Reader " << reader_id << " read: '" << shared_data
                << "' | Wait time: " << duration.count() << "ms";
            SafePrint(ss.str());
```

```
            {
                std::unique_lock<std::mutex> lock(mutex_);
                active_readers--;

                if (active_readers == 0) {
                    writer_cv_.notify_one();
                }
            }

            successful_reads++;
            total_reads++;
        }
        else {
            failed_reads++;
            // Безопасный вывод ошибки
            std::stringstream ss;
            ss << "Reader " << reader_id << " failed to acquire lock (timeout)";
            SafePrint(ss.str());
        }

        return success;
    }

    // Статистика
    void print_statistics() {
        SafePrint("\n=== STATISTICS ===");

        std::stringstream ss1;
        ss1 << "Successful reads: " << successful_reads.load();
        SafePrint(ss1.str());

        std::stringstream ss2;
        ss2 << "Successful writes: " << successful_writes.load();
        SafePrint(ss2.str());

        std::stringstream ss3;
        ss3 << "Failed reads: " << failed_reads.load();
        SafePrint(ss3.str());

        std::stringstream ss4;
        ss4 << "Failed writes: " << failed_writes.load();
        SafePrint(ss4.str());

        std::stringstream ss5;
        ss5   <<   "Total   operations:   "   <<   (total_reads.load()   +
total_writes.load());
        SafePrint(ss5.str());

        int reads = total_reads.load();
        int writes = total_writes.load();
        double ratio = (writes > 0) ? (double)reads / writes : reads;

        std::stringstream ss6;
        ss6 << "Readers/Writers ratio: " << ratio;
        SafePrint(ss6.str());
    }

    // Геттеры для статистики
    int get_successful_reads() const { return successful_reads.load(); }
    int get_successful_writes() const { return successful_writes.load(); }
    int get_failed_reads() const { return failed_reads.load(); }
    int get_failed_writes() const { return failed_writes.load(); }
};
```

```
// Вспомогательная функция для преобразования int в string
std::string int_to_string(int value) {
    std::stringstream ss;
    ss << value;
    return ss.str();
}

// Функция для читателя
void reader_thread(ReaderWriter& rw, int reader_id, int iterations) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(100, 500);

    for (int i = 0; i < iterations; ++i) {
        rw.read_data(reader_id);
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
    }
}

// Функция для писателя
void writer_thread(ReaderWriter& rw, int writer_id, int iterations) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(200, 800);

    for (int i = 0; i < iterations; ++i) {
        rw.write_data(writer_id, "Update_" + int_to_string(i));
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
    }
}

// Функция для читателя с таймаутом
void timed_reader_thread(ReaderWriter& rw, int reader_id, int iterations) {
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_int_distribution<int> dis(50, 300);

    for (int i = 0; i < iterations; ++i) {
        rw.try_read_data(reader_id, 50);
        std::this_thread::sleep_for(std::chrono::milliseconds(dis(gen)));
    }
}

int main() {
    ReaderWriter rw;

    SafePrint("=== READER-WRITER MODEL ===");
    SafePrint("Starting simulation with 3 readers and 2 writers...");

    // Параметры симуляции
    const int num_readers = 3;
    const int num_writers = 2;
    const int num_timed_readers = 2;
    const int iterations = 5;

    std::vector<std::thread> threads;

    // Запуск обычных читателей
    for (int i = 0; i < num_readers; ++i) {
        threads.push_back(std::thread(reader_thread,  std::ref(rw),  i  +  1,
iterations));
    }

    // Запуск писателей
```

```
    for (int i = 0; i < num_writers; ++i) {
        threads.push_back(std::thread(writer_thread,  std::ref(rw),  i  +  1,
iterations));
    }

    // Запуск читателей с таймаутом
    for (int i = 0; i < num_timed_readers; ++i) {
        threads.push_back(std::thread(timed_reader_thread, std::ref(rw),
            num_readers + i + 1, iterations));
    }

    // Ожидание завершения всех потоков
    for (size_t i = 0; i < threads.size(); ++i) {
        threads[i].join();
    }

    // Вывод статистики
    rw.print_statistics();

    // Анализ эффективности
    SafePrint("\n=== PERFORMANCE ANALYSIS ===");

    int total_success = rw.get_successful_reads() + rw.get_successful_writes();
    int   total_operations   =   total_success   +   rw.get_failed_reads()   +
rw.get_failed_writes();
    double success_rate = (total_operations > 0) ? (double)total_success /
total_operations : 0.0;

    std::stringstream ss;
    ss << "Success rate: " << (success_rate * 100) << "%";
    SafePrint(ss.str());

    if (rw.get_failed_reads() > 0) {
        SafePrint("Note: Some reads failed due to timeouts (reader starvation
possible)");
    }

    return 0;
}
```