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»

ОТЧЁТ
к лабораторной работе №4
на тему
«ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ (ПОТОКОВ). ВЗАИМНОЕ ИСКЛЮЧЕНИЕ И СИНХРОНИЗАЦИЯ»
БГУИР 6-05-0612-02 23

Выполнил студент группы 353503
СЕБЕЛЕВ Дмитрий Юрьевич

_____
(дата, подпись студента)

Проверил ассистент каф. информатики
ГРИЦЕНКО Никита Юрьевич

_____
(дата, подпись преподавателя)

Минск 2025

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Целью работы является моделирование классической задачи синхронизации «Обедающие философы» для исследования и анализа проблем, возникающих в многопоточных системах при конкурентном доступе к общим ресурсам, в частности, проблемы взаимной блокировки (deadlock).

Исследование сфокусировано на практической реализации и сравнении различных стратегий разрешения конфликтов с использованием стандартных потоков C++ (std::thread) и объектов синхронизации Windows. В рамках работы производится оценка эффективности каждого подхода на основе собранной статистики, включая время, затраченное потоками на ожидание ресурсов, и количество успешно завершенных операций, что позволяет анализировать общую производительность и отзывчивость системы.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

2.1 Общее назначение и ключевые компоненты

Данный программный модуль реализует симуляцию классической задачи синхронизации «Обедающие философы», предназначенной для изучения конкурентного доступа к разделяемым ресурсам и анализа проблемы взаимной блокировки (deadlock) в многопоточных системах. Каждый философ моделируется отдельным потоком выполнения, создаваемым с помощью класса std::thread, что позволяет использовать стандартные средства многопоточности языка C++.

Разделяемые ресурсы — вилки — представлены в виде набора системных мьютексов Windows, хранящихся в контейнере forks. Для их создания используется функция CreateMutex, а для синхронизации доступа — системный вызов WaitForSingleObject и освобождение через ReleaseMutex. Управление жизненным циклом симуляции осуществляется через атомарную переменную std::atomic<bool> isRunning, которая используется всеми потоками в качестве флага завершения. Это обеспечивает корректное и безопасное завершение работы потоков без принудительного прерывания. Исходный код программы приведён в приложении А [1].

2.2 Жизненный цикл и поведение философа

Логика работы каждого философа реализована в функции philosopher, которая выполняется в отдельном потоке и представляет собой бесконечный цикл, управляемый флагом isRunning. В рамках этого цикла философ последовательно переходит между тремя состояниями: размышление, ожидание ресурсов и приём пищи. Длительность размышления и еды определяется случайным образом в заданных пределах с помощью генератора псевдослучайных чисел std::mt19937 и распределения std::uniform_int_distribution, что реализовано во вспомогательной функции getRandomDuration. Для имитации реальных задержек используется std::this_thread::sleep_for. Состояние ожидания ресурсов реализуется через блокирующие вызовы WaitForSingleObject, которые приостанавливают выполнение потока до момента освобождения соответствующего мьютекса. Таким образом поток эффективно уступает процессорное время другим потокам, что отражает реальное поведение потоков в операционной системе.

2.3 Реализованные стратегии предотвращения взаимных блокировок

Ключевой особенностью программы является поддержка трёх стратегий захвата ресурсов, задаваемых перечислением Strategy и обрабатываемых оператором switch внутри функции философа:

1 Простая (SIMPLE): В данной стратегии философ всегда сначала захватывает левую вилку, затем правую. Это реализовано последовательными вызовами WaitForSingleObject с бесконечным тайм-аутом (INFINITE). Такой подход наглядно демонстрирует возникновение взаимной блокировки при одновременном захвате левых вилок всеми потоками, что приводит к кольцевому ожиданию.

2 Асимметричная (ASYMMETRIC): Для предотвращения deadlock используется различный порядок захвата ресурсов: чётные философы сначала захватывают левую вилку, а нечётные — правую. Условие определяется по идентификатору философа (id % 2). Это нарушает условие циклического ожидания и тем самым предотвращает взаимную блокировку без использования тайм-аутов.

3 С тайм-аутом (WITH_TIMEOUT): В данной стратегии функция WaitForSingleObject вызывается с ограниченным временем ожидания. В случае неудачи философ освобождает уже захваченный мьютекс с помощью ReleaseMutex и повторяет попытку позднее. Такой механизм позволяет обнаруживать потенциальные блокировки и выходить из них, хотя и за счёт увеличения числа неудачных попыток захвата ресурсов.

2.4 Сбор и анализ статистики производительности

Для оценки эффективности работы системы программа ведёт сбор статистики с использованием нескольких контейнеров std::vector, в которых для каждого философа хранятся количество успешных и неудачных попыток еды (successfulMeals, failedMeals), а также суммарное время размышления, ожидания и еды (thinkingTime, waitingTime, eatingTime). Измерение временных интервалов осуществляется с помощью std::chrono::high_resolution_clock, что обеспечивает высокую точность измерений. Поскольку данные структуры являются общими для всех потоков, доступ к ним синхронизируется при помощи мьютекса std::mutex statsMutex и объекта std::lock_guard, предотвращающего состояния гонки. По завершении симуляции на основе собранных данных вычисляются агрегированные показатели, такие как коэффициент активности (отношение времени еды к общему времени) и коэффициент блокировки (доля времени ожидания), которые позволяют количественно сравнить различные стратегии синхронизации.

2.5 Инициализация, управление и завершение симуляции

Функция main выполняет роль центрального управляющего компонента программы. На этапе инициализации она задаёт параметры симуляции, инициализирует векторы статистики и создаёт мьютексы-вилки с помощью CreateMutex. Затем с использованием std::vector<std::thread> запускаются потоки-философы, каждому из которых передаётся уникальный идентификатор и параметры работы. Основной поток приостанавливает выполнение на заданное время с помощью std::this_thread::sleep_for, после чего изменяет значение isRunning, инициируя корректное завершение всех

потоков. Синхронизация с завершением потоков осуществляется через вызовы join. После завершения симуляции выводится итоговая статистика, а затем вызывается функция cleanup, в которой все системные ресурсы освобождаются посредством CloseHandle. Это гарантирует отсутствие утечек ресурсов и корректное завершение программы.

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

3.1 Пример выполнения задания

На рисунке 3.1 показан фрагмент консольного вывода программы в процессе выполнения симуляции. Отображается текущее состояние каждого философа: размышление, ожидание вилок и приём пищи. Сообщения вида иллюстрируют конкурентный захват ресурсов и моменты блокировки потоков при ожидании второй вилки. Данный вывод позволяет визуально проследить динамику взаимодействия философов и выявить потенциальные ситуации взаимного ожидания.

```
Philosopher 0 is thinking...
Philosopher 4 finished eating
Philosopher 3 is eating (forks 3 and 4)
>>> Philosopher 0 took fork 0 and is WAITING for fork 1...
Philosopher 4 is thinking...
Philosopher 3 finished eating
Philosopher 2 is eating (forks 2 and 3)
>>> Philosopher 4 took fork 4 and is WAITING for fork 0...
Philosopher 3 is thinking...
Philosopher 2 finished eating
Philosopher 1 is eating (forks 1 and 2)
>>> Philosopher 3 took fork 3 and is WAITING for fork 4...
Philosopher 2 is thinking...
Philosopher 1 finished eating
Philosopher 0 is eating (forks 0 and 1)
>>> Philosopher 2 took fork 2 and is WAITING for fork 3...
Philosopher 0 finished eating
Philosopher 4 is eating (forks 4 and 0)
Philosopher 1 is thinking...
>>> Philosopher 1 took fork 1 and is WAITING for fork 2...
Philosopher 0 is thinking...
Philosopher 3 is eating (forks 3 and 4)
Philosopher 4 finished eating
>>> Philosopher 0 took fork 0 and is WAITING for fork 1...
Philosopher 4 is thinking...
```

Рисунок 3.1 – Процесс работы симуляции «Обедающие философы»

На рисунке 3.2 представлен результат выполнения симуляции в виде сводной статистики для каждого философа. Для каждого потока указано количество успешных приёмов пищи, число неудачных попыток (при использовании стратегии с тайм-аутом), а также суммарное время, затраченное

на размышление, ожидание ресурсов и еду. Дополнительно рассчитаны коэффициенты активности и блокировки, позволяющие количественно оценить эффективность работы философов и степень конкуренции за ресурсы.

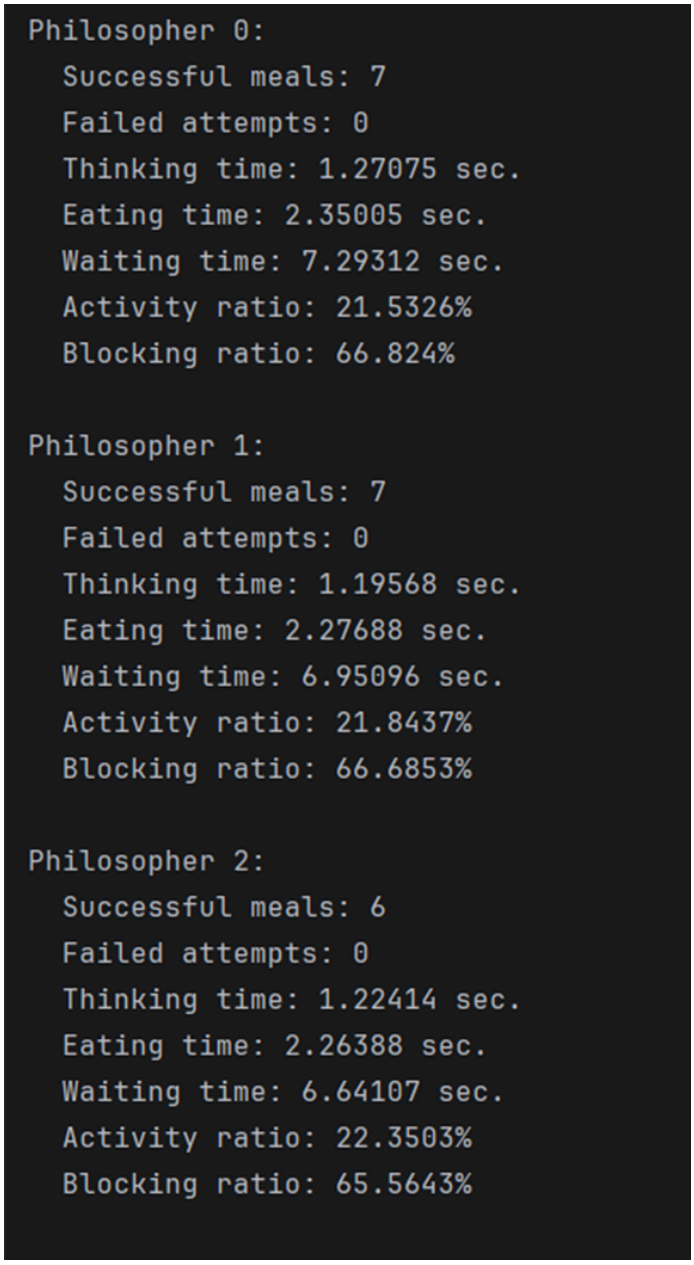

Рисунок 3.2 – Итоговая статистика работы философов

ВЫВОД

В ходе выполнения лабораторной работы были получены практические навыки разработки и анализа многопоточных программ, а также исследования механизмов синхронизации в операционной системе Windows с использованием средств языка C++ и функций WinAPI. Была реализована программная модель классической задачи синхронизации «Обедающие философы», предназначенная для изучения конкурентного доступа к разделяемым ресурсам и возникновения взаимных блокировок.

В разработанной программе каждый философ моделируется отдельным потоком выполнения, создаваемым с использованием класса std::thread. Общие ресурсы — вилки — представлены в виде системных мьютексов Windows, созданных функцией CreateMutex и хранящихся в контейнере std::vector. Захват и освобождение ресурсов осуществляется с помощью функций WaitForSingleObject и ReleaseMutex, что позволяет на практике изучить поведение потоков при блокирующих операциях синхронизации.

Программа реализует несколько стратегий захвата ресурсов, включая наивный симметричный подход, асимметричную схему и стратегию с тайм-аутом. Это дало возможность наглядно проанализировать условия возникновения взаимной блокировки, а также методы её предотвращения и обнаружения. Управление выполнением потоков и корректное завершение симуляции обеспечивается с помощью атомарной переменной std::atomic, что исключает необходимость принудительного завершения потоков.

В процессе работы осуществлялся сбор детальной статистики для каждого философа, включающей количество успешных приёмов пищи, суммарное время размышления, ожидания ресурсов и еды. Измерение временных характеристик производилось с использованием средств библиотеки std::chrono, а синхронизация доступа к общим структурам данных обеспечивалась с помощью std::mutex. На основе собранных данных были рассчитаны относительные показатели активности и блокировки, позволяющие количественно оценить эффективность выбранной стратегии синхронизации.

Полученные в ходе лабораторной работы знания и навыки являются базовыми для понимания принципов параллельного программирования и проектирования многопоточных приложений. Реализованная модель демонстрирует практическое применение механизмов синхронизации, анализ конкурентного поведения потоков и влияние различных стратегий управления ресурсами на производительность системы, что имеет важное значение при разработке высоконагруженных и отказоустойчивых программных систем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] API Win32 documentation [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/. – Дата доступа: 13.11.2025.

[2] WinAPI: CreateMutexA [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/synchapi/nf-synchapi-createmutexa. – Дата доступа: 13.11.2025.

[3] WinAPI: WaitForSingleObject [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject. – Дата доступа: 13.11.2025.

[4] WinAPI: ReleaseMutex [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/synchapi/nf-synchapi-releasemutex. – Дата доступа: 13.11.2025.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код программы

```
#include <iostream>
#include <windows.h>
#include <thread>
#include <vector>
#include <chrono>
#include <mutex>
#include <atomic>
#include <random>

// Вилки как мьютексы
std::vector<HANDLE> forks;

// Статистика
std::mutex statsMutex;
std::vector<int> successfulMeals;
std::vector<int> failedMeals;
std::vector<double> thinkingTime;
std::vector<double> eatingTime;
std::vector<double> waitingTime;

std::atomic<bool> isRunning(true);

thread_local std::mt19937 rng(std::random_device{}());

int getRandomDuration(int min, int max) {
    std::uniform_int_distribution<int> dist(min, max);
    return dist(rng);
}

enum class Strategy {
    SIMPLE,
    ASYMMETRIC,
    WITH_TIMEOUT
};

Strategy currentStrategy = Strategy::SIMPLE;

void philosopher(int id, int numPhilosophers, int minThinkTime, int
maxThinkTime,
                 int minEatTime, int maxEatTime, int timeout) {
    int leftFork = id;
    int rightFork = (id + 1) % numPhilosophers;

    while (isRunning) {
        // Размышление
        auto thinkStart = std::chrono::high_resolution_clock::now();
        int thinkDuration = getRandomDuration(minThinkTime,
maxThinkTime);

std::this_thread::sleep_for(std::chrono::milliseconds(thinkDuration));
        auto thinkEnd = std::chrono::high_resolution_clock::now();

        {
            std::lock_guard<std::mutex> lock(statsMutex);
            thinkingTime[id] += std::chrono::duration<double>(thinkEnd -
thinkStart).count();
            std::cout << "Philosopher " << id << " is thinking..." <<
std::endl;
        }

        auto waitStart = std::chrono::high_resolution_clock::now();
        bool gotForks = false;

        switch (currentStrategy) {
```

```
            case Strategy::SIMPLE: {
                // Простое - берем левую, потом правую вилку
                WaitForSingleObject(forks[leftFork], INFINITE);
                // deadlock may be here

std::this_thread::sleep_for(std::chrono::milliseconds(10));
                {
                    std::lock_guard<std::mutex> lock(statsMutex);
                    std::cout << ">>> Philosopher " << id << " took
fork " << leftFork
                              << " and is WAITING for fork " <<
rightFork << "..." << std::endl;
                }
                WaitForSingleObject(forks[rightFork], INFINITE);
                gotForks = true;
                break;
            }

            case Strategy::ASYMMETRIC: {
                // Асимметричное - четные философы берут сначала левую,
                // нечетные - сначала правую
                if (id % 2 == 0) {
                    WaitForSingleObject(forks[leftFork], INFINITE);
                    WaitForSingleObject(forks[rightFork], INFINITE);
                } else {
                    WaitForSingleObject(forks[rightFork], INFINITE);
                    WaitForSingleObject(forks[leftFork], INFINITE);
                }
                gotForks = true;
                break;
            }

            case Strategy::WITH_TIMEOUT: {
                // с тайм-аутом
                DWORD result = WaitForSingleObject(forks[leftFork],
timeout);
                if (result == WAIT_OBJECT_0) {
                    result = WaitForSingleObject(forks[rightFork],
timeout);
                    if (result == WAIT_OBJECT_0) {
                        gotForks = true;
                    } else {
                        // Не удалось взять правую вилку, освобождаем
левую
                        ReleaseMutex(forks[leftFork]);
                        gotForks = false;
                    }
                } else {
                    gotForks = false;
                }
                break;
            }
        }

        auto waitEnd = std::chrono::high_resolution_clock::now();

        if (gotForks) {
            {
                std::lock_guard<std::mutex> lock(statsMutex);
                waitingTime[id] += std::chrono::duration<double>(waitEnd
- waitStart).count();
                std::cout << "Philosopher " << id << " is eating (forks
"
                          << leftFork << " and " << rightFork << ")" <<
std::endl;
            }

            // Еда
            auto eatStart = std::chrono::high_resolution_clock::now();
            int eatDuration = getRandomDuration(minEatTime, maxEatTime);
```

```
std::this_thread::sleep_for(std::chrono::milliseconds(eatDuration));
            auto eatEnd = std::chrono::high_resolution_clock::now();

            // Освобождаем вилки
            ReleaseMutex(forks[leftFork]);
            ReleaseMutex(forks[rightFork]);

            {
                std::lock_guard<std::mutex> lock(statsMutex);
                successfulMeals[id]++;
                eatingTime[id] += std::chrono::duration<double>(eatEnd -
eatStart).count();
                std::cout << "Philosopher " << id << " finished eating"
<< std::endl;
            }
        } else {
            std::lock_guard<std::mutex> lock(statsMutex);
            failedMeals[id]++;
            waitingTime[id] += std::chrono::duration<double>(waitEnd -
waitStart).count();
            std::cout << "Philosopher " << id << " failed to get forks
(timeout)" << std::endl;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
}

void cleanup(int numPhilosophers) {
    for (int i = 0; i < numPhilosophers; ++i) {
        CloseHandle(forks[i]);
    }
}

int main() {
    int numPhilosophers = 5;
    int minThinkTime = 100;
    int maxThinkTime = 300;
    int minEatTime = 200;
    int maxEatTime = 500;
    int timeout = 100; // мс
    int simulationTime = 10; // секунды

    std::cout << "Dining Philosophers\n";
    std::cout << "Number of philosophers: " << numPhilosophers <<
std::endl;
    std::cout << "Strategy: ";
    switch (currentStrategy) {
        case Strategy::SIMPLE:
            std::cout << "Simple (may deadlock)\n";
            break;
        case Strategy::ASYMMETRIC:
            std::cout << "Asymmetric\n";
            break;
        case Strategy::WITH_TIMEOUT:
            std::cout << "With timeout (" << timeout << "ms)\n";
            break;
    }
    std::cout << "Simulation time: " << simulationTime << " seconds\n\n";

    forks.resize(numPhilosophers);
    successfulMeals.resize(numPhilosophers, 0);
    failedMeals.resize(numPhilosophers, 0);
    thinkingTime.resize(numPhilosophers, 0.0);
    eatingTime.resize(numPhilosophers, 0.0);
    waitingTime.resize(numPhilosophers, 0.0);

    for (int i = 0; i < numPhilosophers; ++i) {
        forks[i] = CreateMutex(NULL, FALSE, NULL);
    }
```

```
    // Создание потоков-философов
    std::vector<std::thread> threads;
    for (int i = 0; i < numPhilosophers; ++i) {
        threads.emplace_back(philosopher, i, numPhilosophers,
                             minThinkTime, maxThinkTime,
                             minEatTime, maxEatTime, timeout);
    }

    // Симуляция
    std::this_thread::sleep_for(std::chrono::seconds(simulationTime));
    isRunning = false;

    for (auto &th: threads) {
        th.join();
    }

    // Вывод результатов
    std::cout << "\nSimulation Results\n\n";

    int totalSuccessful = 0;
    int totalFailed = 0;
    double totalThinking = 0.0;
    double totalEating = 0.0;
    double totalWaiting = 0.0;

    for (int i = 0; i < numPhilosophers; ++i) {
        std::cout << "Philosopher " << i << ":\n";
        std::cout << "  Successful meals: " << successfulMeals[i] <<
std::endl;
        std::cout << "  Failed attempts: " << failedMeals[i] <<
std::endl;
        std::cout << "  Thinking time: " << thinkingTime[i] << " sec."
<< std::endl;
        std::cout << "  Eating time: " << eatingTime[i] << " sec." <<
std::endl;
        std::cout << "  Waiting time: " << waitingTime[i] << " sec." <<
std::endl;

        double totalTime = thinkingTime[i] + eatingTime[i] +
waitingTime[i];
        if (totalTime > 0) {
            std::cout << "  Activity ratio: " << (eatingTime[i] /
totalTime * 100) << "%" << std::endl;
            std::cout << "  Blocking ratio: " << (waitingTime[i] /
totalTime * 100) << "%" << std::endl;
        }
        std::cout << std::endl;

        totalSuccessful += successfulMeals[i];
        totalFailed += failedMeals[i];
        totalThinking += thinkingTime[i];
        totalEating += eatingTime[i];
        totalWaiting += waitingTime[i];
    }

    std::cout << "Statistics\n";
    std::cout << "Total successful meals: " << totalSuccessful <<
std::endl;
    std::cout << "Average meals per philosopher: " << (totalSuccessful /
(double) numPhilosophers) << std::endl;
    std::cout << "Total thinking time: " << totalThinking << " sec." <<
std::endl;
    std::cout << "Total eating time: " << totalEating << " sec." <<
std::endl;
    std::cout << "Total waiting time: " << totalWaiting << " sec." <<
std::endl;

    double totalTime = totalThinking + totalEating + totalWaiting;
    if (totalTime > 0) {
        std::cout << "Overall efficiency: " << (totalEating / totalTime
* 100) << "%" << std::endl;
```

```
    }

    cleanup(numPhilosophers);
    return 0;
}
```