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**

Выполнил: студент гр.253504

Фроленко К.Ю.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

[1 Формулировка задачи 3](#_Toc181797430)

[2 Краткие теоритические сведения 4](#_Toc181797431)

[3 Описание функций программы 5](#_Toc181797432)

[3.1 Класс Philosopher 5](#_Toc181797433)

[3.2 Класс Table 5](#_Toc181797434)

[3.3 Функция main 6](#_Toc181797435)

[4 Пример выполнения программы 7](#_Toc181797436)

[4.1 Запуск программы и процесс выполнения 7](#_Toc181797437)

[4.2 Описание работы и результатов 8](#_Toc181797438)

[Вывод 9](#_Toc181797439)

[Список использованных источников 10](#_Toc181797440)

[Приложения А (обязательное) 11](#_Toc181797441)

1 ФОРМУЛИРОВКА ЗАДАЧИ

**Цель** лабораторной работы заключается в разработке и анализе приложения, демонстрирующего механизмы взаимного исключения и синхронизации в многопоточной среде. Основной задачей является исследование и сравнение эффективности различных системных объектов и функций, предназначенных для обеспечения синхронизации и передачи управления между взаимодействующими процессами (потоками) в операционной системе. Это позволит улучшить понимание типичных проблем, возникающих при организации взаимодействия процессов, и изучить различные модели для их описания и решения.

В рамках лабораторной работы предполагается выполнение следующих задач:

1 Реализация модели взаимодействия конкурирующих параллельных процессов обедающие философы. Это включает параметризацию модели для изменения количества философов, логики разрешения конфликтов, интенсивности доступа к ресурсам и других характеристик.

2 Проверка и демонстрация различных подходов к решению задач обеспечения корректного взаимодействия, включая стратегии решения конфликтов и методы предотвращения взаимных блокировок.

3 Анализ и оценка эффективности. Измерение и сравнение времени выполнения операций в контексте различных методов синхронизации. Анализ результатов моделирования для определения соотношения времени активности и блокировки, эффективности использования критических ресурсов и общей пропускной способности системы.

Эти этапы работы позволят не только углубить знания в области системного программирования, но и получить практический опыт в оптимизации процессов взаимодействия процессов и потоков, что способствует разработке более эффективных решений для многопоточных приложений.

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. Важность данных инструментов возрастает в условиях, когда система должна поддерживать целостность данных и предотвращать коллизии и блокировки [1].

Критические секции и мьютексы служат для гарантирования, что только один поток может выполнять определенный участок кода, который взаимодействует с общим ресурсом. 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, позволяя одному потоку ожидать сигнала от другого перед продолжением выполнения [2].

Проблема обедающие философы демонстрирует типичные сложности, возникающие при организации доступа к общим ресурсам. В этой модели философы, сидящие за круглым столом, пытаются взять две вилки, расположенные по бокам от каждого из них. Конфликты и блокировки возникают, когда каждый философ пытается взять вилку одновременно с другим. Решение задачи включает разработку стратегии, которая позволяет философам избегать взаимных блокировок и гарантировать равный доступ к ресурсам [3].

Моделирование этой системы включает параметризацию количества философов, логики разрешения конфликтов, интенсивности доступа и продолжительности использования ресурсов. 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.

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В рамках разработки программы для демонстрации работы с файловой базой данных были реализованы следующие функции:

– Класс *Philosopher*;

– класс *Table*;

– функция *main*.

3.1 Класс Philosopher

Класс *Philosopher* представляет собой философа, который участвует в моделировании задачи обедающих философов:

1 **Конструктор *Philosopher*** – инициализирует философа, принимая уникальный идентификатор, ссылки на мьютексы вилок, стратегию разрешения конфликтов, флаг остановки симуляции, генератор случайных чисел и распределения для времени размышлений и еды. Конструктор обеспечивает правильную настройку параметров каждого философа для участия в симуляции.

2 **Метод *dine*** – основной метод, представляющий цикл действий философа, в котором он поочерёдно размышляет и пытается поесть, следуя выбранной стратегии. При успешном захвате обеих вилок философ выполняет операцию еды и освобождает ресурсы. Метод ведёт статистику о времени размышлений, времени еды, количестве успешных приемов пищи и неудачных попыток.

3 **Метод *pickUpForks*** – вспомогательная функция для захвата вилок в определённом порядке. Метод пытается захватить первую и затем вторую вилку, и, если не удаётся захватить обе, освобождает уже захваченную вилку, предотвращая возможность взаимных блокировок.

3.2 Класс Table

Класс *Table* представляет собой стол, за которым находятся философы и который управляет процессом моделирования:

1 Конструктор *Table* – инициализирует стол с заданным количеством философов, стратегией разрешения конфликтов, диапазонами времени для размышлений и еды, а также временем симуляции. Конструктор создаёт философов, мьютексы для вилок и определяет стратегию разрешения конфликтов, обеспечивая начальную настройку для моделирования;

2 **Деструктор ~*Table*** – освобождает ресурсы, выделенные для философов, чтобы предотвратить утечки памяти и обеспечить корректное завершение работы;

3 **Метод *startDinner*** – запускает симуляцию, создавая потоки для каждого философа, которые выполняют метод *dine*. После заданного времени симуляции флаг остановки изменяется, и происходит ожидание завершения всех потоков, что обеспечивает корректное завершение процесса моделирования;

4 **Метод *printStatistics*** – выводит на экран результаты симуляции, такие как общее время размышлений и еды для каждого философа, количество успешных и неудачных попыток поесть, а также общую эффективность процесса.

3.3 Функция main

Функция *main* является входной точкой программы и демонстрирует использование классов *Philosopher* и *Table*:

1 **Инициализация** – на этапе инициализации выбирается стратегия разрешения конфликтов, и создаётся экземпляр класса *Table* с заданным количеством философов и параметрами симуляции;

2 **Запуск моделирования** – функция вызывает метод *startDinner*, который начинает процесс симуляции с участием всех философов;

3 **Вывод результатов** – после завершения симуляции вызывается метод *printStatistics*, чтобы вывести результаты моделирования, включая общее время, успешные приёмы пищи и эффективность;

4 **Завершение работы** – по завершении работы все ресурсы корректно освобождаются, что обеспечивает стабильную и надёжную работу программы.

4 ПРИМЕР ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.1 Запуск программы и процесс выполнения

Программа начинается с инициализации параметров симуляции задачи обедающих философов. На этапе инициализации создаётся объект класса *Table*, который управляет всеми философами и их взаимодействиями. Количество философов, стратегия разрешения конфликтов, время размышлений и еды, а также общее время симуляции задаются в начале программы. В качестве стратегии разрешения конфликтов может использоваться либо иерархия ресурсов (*ResourceHierarchy*), либо официант (*Waiter*).

Запуск симуляции осуществляется методом startDinner, который создаёт потоки для каждого философа и запускает метод dine у каждого из них. Философы начинают размышлять и пытаться поесть, используя мьютексы для захвата вилок. На рисунке 4.1 представлены результаты выполнения симуляции на консоли, включая время размышлений и еды для каждого философа, количество успешных приемов пищи и неудачных попыток.

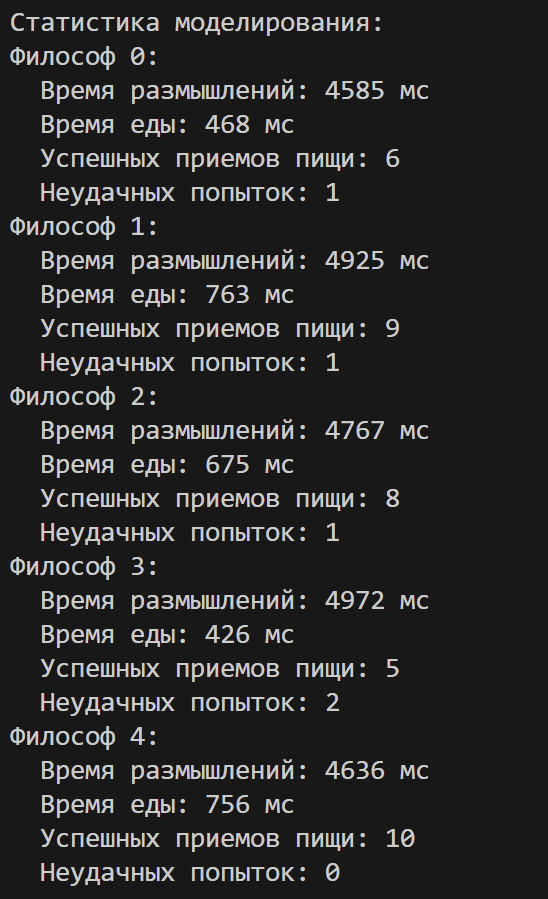
****

Рисунок 4.1 – Консоль с результатами выполнения симуляции

После завершения симуляции объект класса *Table* выводит на консоль статистику, включающую общие времена размышлений и еды, а также количество успешных и неудачных попыток поесть для всех философов. Это позволяет проанализировать эффективность используемой стратегии и её влияние на выполнение задач каждым философом.

4.2 Описание работы и результатов

Для демонстрации различий в производительности между двумя стратегиями разрешения конфликтов были проведены тесты с двумя стратегиями: ResourceHierarchy и Waiter. Время выполнения операций размышлений и еды фиксировалось для каждой стратегии и представлено в таблице 4.1.

Таблица 4.1 – Время выполнения операций для различных объемов данных

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Стратегия | Общее время размышлений (мс) | Общее время еды (мс) | Успешные приёмы пищи | Неудачные попытки |
| ResourceHierarchy | 23885 | 3088 | 38 | 5 |
| Waiter | 22778 | 3014 | 37 | 14 |

Как видно из таблицы 4.1, стратегия Waiter показала несколько лучшие результаты по количеству успешных приёмов пищи и меньшему количеству неудачных попыток по сравнению с ResourceHierarchy. Это связано с тем, что стратегия Waiter позволяет избежать ситуации взаимной блокировки, так как все философы должны запросить доступ у "официанта" перед тем, как начать попытку захвата вилок. В свою очередь, стратегия ResourceHierarchy основывается на строгом порядке захвата вилок, что может приводить к временным задержкам и большему числу неудачных попыток.

Статистика также показывает, что общие времена размышлений и еды остаются близкими в обеих стратегиях, но эффективность приёма пищи выше при использовании стратегии Waiter, что делает её предпочтительной в условиях высокой нагрузки и необходимости минимизировать время простоя философов.

Эти результаты наглядно демонстрируют, что выбор стратегии разрешения конфликтов оказывает значительное влияние на общую производительность системы, особенно при работе в многопоточном окружении. Стратегия Waiter позволяет сократить количество неудачных попыток захвата вилок и минимизировать время простоя, что делает её предпочтительной для задач, требующих высокой координации между потоками и эффективного использования ресурсов.

# ВЫВОД

В процессе выполнения данной лабораторной работы было подробно изучено, как работают механизмы взаимного исключения и синхронизации в условиях многопоточной среды, а также на примере задачи обедающих философов проведено исследование различных подходов к разрешению конфликтов между потоками. В рамках эксперимента были реализованы и проанализированы две стратегии — иерархия ресурсов и использование посредника в роли официанта, что позволило более глубоко понять, какие подходы оказывают наибольшее влияние на производительность системы.

Результаты показали, что стратегия официанта обеспечивает более плавное выполнение операций при высокой нагрузке, так как предотвращает взаимные блокировки, что способствует более стабильной работе в условиях ограниченных ресурсов и минимизации времени простоя. Стратегия иерархии ресурсов, хотя и продемонстрировала приемлемую эффективность, выявила определённые ограничения: строгий порядок захвата ресурсов иногда приводил к задержкам и увеличению количества неудачных попыток получения доступа к ресурсам (в данном случае – вилкам), что могло снижать общую производительность при высоких требованиях к скорости обработки.

Также проведенные тесты подтвердили, что обе стратегии способны обеспечить приемлемые временные характеристики как для фазы размышлений, так и для фазы питания, что делает их подходящими для синхронизации потоков в системах с разной степенью нагрузки.

Благодаря этой работе удалось углубить знания в области многопоточного программирования, приобрести практические навыки в проектировании взаимодействия потоков и усвоить критически важные аспекты создания высокопроизводительных и масштабируемых программных решений.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Microsoft Documentation: Разработка приложений с помощью WinAPI [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/. – Дата доступа: 06.11.2024.   
 [2] Multithreading Tutorial - CodeProject [Электронный ресурс]. – Режим доступа: <https://www.codeproject.com/Articles/14746/Multithreading-Tutorial>. – Дата доступа: 06.11.2024.

[3] Microsoft Documentation: Build desktop Windows apps using the Win32 API [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/en-us/windows/win32/>. – Дата доступа:06.11.2024.

ПРИЛОЖЕНИЕ А

(обязательное)

Исходный код программы

#include <bits/stdc++.h>

#define ll long long

#define ld long double

#define vt vector<thread>

#define vm vector<mutex>

using namespace std;

using namespace chrono;

#define NUM\_PHILOSOPHERS 5

#define CONFLICT\_STRATEGY ConflictResolutionStrategy::Waiter // ResourceHierarchy, Waiter

ll THINK\_MIN = 50;

ll THINK\_MAX = 1000;

ll EAT\_MIN = 50;

ll EAT\_MAX = 100;

#define SIMULATION\_TIME 5

enum class ConflictResolutionStrategy

{

ResourceHierarchy,

Waiter

};

class Philosopher

{

public:

Philosopher(ll id, mutex &leftFork, mutex &rightFork, ConflictResolutionStrategy strategy,

mutex &waiter, atomic<bool> &stopFlag,

mt19937 &rng, uniform\_int\_distribution<int> &thinkDist,

uniform\_int\_distribution<int> &eatDist)

: id(id), leftFork(leftFork), rightFork(rightFork), strategy(strategy),

waiter(waiter), stopFlag(stopFlag),

rng(rng), thinkDist(thinkDist), eatDist(eatDist),

totalThinkingTime(0), totalEatingTime(0),

successfulEats(0), failedAttempts(0) {}

void dine()

{

while (!stopFlag.load())

{

auto thinkTime = thinkDist(rng);

auto startThink = steady\_clock::now();

this\_thread::sleep\_for(milliseconds(thinkTime));

auto endThink = steady\_clock::now();

totalThinkingTime += duration\_cast<milliseconds>(endThink - startThink).count();

bool eaten = false;

if (strategy == ConflictResolutionStrategy::ResourceHierarchy)

{

if (id % 2 == 0)

{

eaten = pickUpForks(leftFork, rightFork);

}

else

{

eaten = pickUpForks(rightFork, leftFork);

}

}

else if (strategy == ConflictResolutionStrategy::Waiter)

{

unique\_lock<mutex> lock(waiter);

eaten = pickUpForks(leftFork, rightFork);

}

if (eaten)

{

auto eatTime = eatDist(rng);

auto startEat = steady\_clock::now();

this\_thread::sleep\_for(milliseconds(eatTime));

auto endEat = steady\_clock::now();

totalEatingTime += duration\_cast<milliseconds>(endEat - startEat).count();

successfulEats++;

leftFork.unlock();

rightFork.unlock();

}

else

{

failedAttempts++;

}

}

}

ll totalThinkingTime;

ll totalEatingTime;

ll successfulEats;

ll failedAttempts;

ll id;

private:

mutex &leftFork;

mutex &rightFork;

ConflictResolutionStrategy strategy;

mutex &waiter;

atomic<bool> &stopFlag;

mt19937 &rng;

uniform\_int\_distribution<int> &thinkDist;

uniform\_int\_distribution<int> &eatDist;

bool pickUpForks(mutex &firstFork, mutex &secondFork)

{

if (!firstFork.try\_lock())

{

return false;

}

if (!secondFork.try\_lock())

{

firstFork.unlock();

return false;

}

return true;

}

};

class Table

{

public:

Table(int numPhilosophers, ConflictResolutionStrategy strategy,

int thinkMin, int thinkMax, int eatMin, int eatMax, int simulationTime)

: numPhilosophers(numPhilosophers), strategy(strategy),

thinkDist(thinkMin, thinkMax), eatDist(eatMin, eatMax),

simulationTime(simulationTime), stopFlag(false),

waiter()

{

forks = vm(numPhilosophers);

for (ll i = 0; i < numPhilosophers; ++i)

{

philosophers.emplace\_back(new Philosopher(

i,

forks[i],

forks[(i + 1) % numPhilosophers],

strategy,

waiter,

stopFlag,

rng,

thinkDist,

eatDist));

}

}

~Table()

{

for (auto p : philosophers)

{

delete p;

}

}

void startDinner()

{

for (auto p : philosophers)

{

threads.emplace\_back(&Philosopher::dine, p);

}

this\_thread::sleep\_for(seconds(simulationTime));

stopFlag.store(true);

for (auto &t : threads)

{

if (t.joinable())

t.join();

}

}

void printStatistics() const

{

cout << "Статистика моделирования:\n";

ll totalThinking = 0;

ll totalEating = 0;

ll totalSuccess = 0;

ll totalFailures = 0;

for (const auto p : philosophers)

{

cout << "Философ " << p->id << ":" << endl;

cout << " Время размышлений: " << p->totalThinkingTime << " мс" << endl;

cout << " Время еды: " << p->totalEatingTime << " мс" << endl;

cout << " Успешных приемов пищи: " << p->successfulEats << endl;

cout << " Неудачных попыток: " << p->failedAttempts << endl;

totalThinking += p->totalThinkingTime;

totalEating += p->totalEatingTime;

totalSuccess += p->successfulEats;

totalFailures += p->failedAttempts;

}

cout << endl

<< "Общая статистика:" << endl;

cout << " Общее время размышлений: " << totalThinking << " мс" << endl;

cout << " Общее время еды: " << totalEating << " мс" << endl;

cout << " Общие успешные приемы пищи: " << totalSuccess << endl;

cout << " Общие неудачные попытки: " << totalFailures << endl;

cout << " Общая эффективность (пропускная способность): "

<< static\_cast<ld>(totalSuccess) / simulationTime << " приемов пищи в секунду\n";

}

private:

ll numPhilosophers;

ConflictResolutionStrategy strategy;

vm forks;

vector<Philosopher \*> philosophers;

vt threads;

uniform\_int\_distribution<int> thinkDist;

uniform\_int\_distribution<int> eatDist;

mt19937 rng{random\_device{}()};

ll simulationTime;

atomic<bool> stopFlag;

mutex waiter;

};

int main()

{

ConflictResolutionStrategy strategy;

try

{

strategy = CONFLICT\_STRATEGY;

}

catch (exception &e)

{

cerr << e.what() << endl;

return 1;

}

cout << "Моделирование проблемы обедающих философов" << endl;

cout << "Количество философов: " << NUM\_PHILOSOPHERS << endl;

cout << "Стратегия разрешения конфликтов: "

<< (strategy == ConflictResolutionStrategy::ResourceHierarchy ? "ResourceHierarchy" : "Waiter")

<< endl;

cout << "Время размышлений: " << THINK\_MIN << "-" << THINK\_MAX << " мс" << endl;

cout << "Время еды: " << EAT\_MIN << "-" << EAT\_MAX << " мс" << endl;

cout << "Время симуляции: " << SIMULATION\_TIME << " секунд" << endl

<< endl;

Table table(NUM\_PHILOSOPHERS, strategy, THINK\_MIN, THINK\_MAX, EAT\_MIN, EAT\_MAX, SIMULATION\_TIME);

table.startDinner();

table.printStatistics();

return 0;

}