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 Новиков В.А.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

**СОДЕРЖАНИЕ**

[1 Формулировка задачи 3](#_Toc184318839)

[2 Краткие теоритические сведения 4](#_Toc184318840)

[3 Описание функций программы 5](#_Toc184318841)

[3.1 Ключевые функции программы 5](#_Toc184318842)

[3.2 Принцип работы программы 6](#_Toc184318843)

[Заключение 10](#_Toc184318844)

[Список использованных источников 11](#_Toc184318845)

[Приложение А (обязательное) 12](#_Toc184318846)

# **1 ФОРМУЛИРОВКА ЗАДАЧИ**

Целью выполнения лабораторной работы является изучение механизмов синхронизации и взаимного исключения при доступе потоков к разделяемым ресурсам. Для достижения цели требуется разработать программу, демонстрирующую корректное взаимодействие потоков на примере задачи «обедающие философы» с использованием различных подходов к синхронизации, таких как мьютексы и семафоры.

Задачи работы включают:

– Изучение различных механизмов синхронизации в Windows API;

– Реализацию двух стратегий разрешения конфликтов при доступе к критическим ресурсам;

– Проведение экспериментов для анализа эффективности реализованных стратегий;

– Сбор и обработку статистики работы потоков.

.

2 КРАТКИЕ ТЕОРИТИЧЕСКИЕ СВЕДЕНИЯ

Критический ресурс – такой ресурс системы, который не может использоваться одновременно более чем заданным числом пользователей. Часто речь идет о доступности его не более чем одному пользователю.

Критическая секция ‒ это участок кода программы, который запрашивает монопольный доступ к каким-либо общим ресурсам, которые не могут быть использованы в одно время, больше чем одним потоком реализации. Если в критической секции состоит более чем один процесс, тогда обнаруживается состояние так называемой «гонки».[1]

Межпроцессное взаимодействие – обмен данными между потоками одного или разных процессов. Реализуется посредством механизмов, предоставляемых ядром ОС или процессом, использующим механизмы ОС и реализующим новые возможности *IPC*. Может осуществляться как на одном компьютере, так и между несколькими компьютерами сети.[2]

Общая идея *IPC*: проверка и модификация некоторого признака (флага) перед доступом к критическому ресурсу. Значение флага отражает свободное или занятое состояние ресурса. Основная проблема: регулятор доступа к критическим ресурсам – тоже критический ресурс, обращения к нему – критическая секция. Рекурсивное замыкание требований.

Семафор – глобальная переменная-счетчик *S*, целочисленные неотрицательные значения, атомарно выполняемые примитивы для доступа *P(S)* и *V(S)*:

1 *P(S)* – условный декремент: если значение достигло 0, то ожидание ненулевого значения.

2 *V(S)* – безусловный инкремент счетчика.

Мьютекс – можно рассматривать как упрощенный двузначный семафор, состояние которого интерпретируют как «свободность» и «занятость», а примитивы доступа – «захват» и «освобождение». Попытка повторного захвата блокирует поток-инициатор до освобождения мьютекса другим потоком.

Барьер (*barrier*) – объект *ISO*, обеспечивающий синхронизацию достигших его процессов (потоков): каждый из них посредством системного вызова запрашивает синхронизацию барьером и переводится в состояние ожидания до тех пор, пока количество таких запросов не достигнет заранее заданного. После этого все получают возможность выполняться дальше, а барьер «освобождается».[3]

3 ОПИСАНИЕ ФУНКЦИЙ ПРОГРАММЫ

В данной программе реализуется функциональность модели взаимодействия потоков в задаче «обедающие философы». Основное назначение системы – демонстрация корректной синхронизации потоков при совместном доступе к критическим ресурсам и анализ эффективности различных стратегий разрешения конфликтов.

Программа реализует задачу «обедающие философы» с использованием двух стратегий синхронизации:

– Обычная блокировка: философы блокируют ресурсы в произвольном порядке.

– Приоритетная стратегия: чётные философы начинают с левой вилки, нечётные – с правой.

Для реализации программы используются потоки Windows API, мьютексы для защиты ресурсов и статистика для анализа работы. Программа собирает данные о количестве успешных попыток доступа к ресурсам и временных задержках.

## **3.1 Ключевые функции программы**

Программа включает следующие ключевые функции:

*«DWORD WINAPI philosopher(LPVOID param)»*. Эта функция моделирует поведение философа в отдельном потоке:

1 Генерирует случайные задержки для этапов «думания» и «еды», что добавляет реалистичности в процесс;

2 Реализует захват вилок, используя мьютексы. В стандартной стратегии философы захватывают вилки в произвольном порядке. В приоритетной стратегии четные философы сначала захватывают левую вилку, затем правую, а нечетные – наоборот. При невозможности захватить вторую вилку философ освобождает первую, избегая тупиков;

3 Обновляет статистику: фиксирует успешные попытки еды и случаи блокировок.

*«int main()»*. Основная функция программы:

1 Инициализирует ресурсы: создает массив мьютексов для вилок и массив потоков для философов;

2 Запускает потоки, каждый из которых выполняет функцию philosopher с уникальным идентификатором философа;

3 Организует выполнение эксперимента, контролируя время работы программы;

4 Собирает и выводит статистику, включая успешные/неуспешные попытки использования ресурсов и общую эффективность системы;

5 Освобождает выделенные ресурсы, закрывая дескрипторы потоков и мьютексов.

## **3.2 Принцип работы программы**

Программа моделирует задачу обедающих философов, где философы представлены отдельными потоками, каждый из которых конкурирует за ограниченные ресурсы, такие как вилки. Каждый философ должен захватить две вилки, чтобы поесть, и затем освободить их. В качестве механизма синхронизации используется мьютекс, который блокирует доступ к вилке, пока философ ее использует. Это гарантирует, что только один философ может использовать вилку в каждый момент времени, что предотвращает возможные ошибки, связанные с одновременным доступом к одному ресурсу.

Каждый философ использует два мьютекса: один для левой вилки, второй – для правой. Когда философу нужно поесть, он сначала пытается захватить левую вилку, а затем правую, используя мьютексы. Если одна из вилок занята другим философом, он должен подождать, пока она не будет освобождена, чтобы продолжить процесс. Механизм синхронизации гарантирует, что доступ к вилкам будет организован корректно, без возможности одновременного использования одной вилки двумя философами.

Для решения проблемы захвата вилок реализованы две стратегии синхронизации. Первая стратегия – стандартная – заключается в том, что философы захватывают вилки в произвольном порядке. Например, философ с четным идентификатором сначала захватывает левую вилку, а затем правую, тогда как философ с нечетным идентификатором делает наоборот. Эта стратегия проста в реализации, но она может привести к взаимной блокировке. Ситуация, при которой философы, захватив по одной вилке, не могут получить вторую и начинают ожидать друг друга, может привести к тупику.

Вторая стратегия – приоритетная, также заключается в захвате вилок, но философы с четным и нечетным идентификатором действуют по разному. Философы с четным идентификатором захватывают сначала левую вилку, а философы с нечетным идентификатором – правую. Эта стратегия помогает избежать взаимных блокировок, так как нарушается симметрия, и философы не оказываются в ситуации, когда все они блокируют друг друга. В случае, если философ не может захватить вторую вилку, он освобождает первую вилку и повторяет попытку захвата, что также помогает избежать тупика, но добавляет небольшую задержку.

Выбор количества философов и стратегии решения проблемы захвата вилок производится в начале программы и изображен на рисунке 3.1.

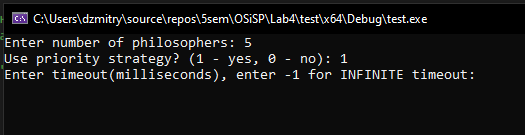


Рисунок 3.1 – Выбор количества философов и стратегии

Кроме того, программа собирает статистику, которая позволяет оценить эффективность работы каждой стратегии. В процессе выполнения фиксируется количество успешных попыток поесть, когда философу удалось захватить обе вилки, а также количество случаев блокировки, когда философ не смог получить обе вилки и был вынужден ожидать. Эта информация помогает понять, какая стратегия синхронизации наиболее эффективна в условиях конкуренции за ресурсы, а также позволяет вычислить общую пропускную способность системы. Время, которое философы проводят в ожидании, и количество блокировок также являются важными показателями, позволяющими судить о том, насколько хорошо работает выбранная стратегия.

Когда программа завершает свою работу, выводится итоговая статистика (рисунок 3.2), которая демонстрирует, сколько раз философы успешно поели и сколько раз они блокировались.

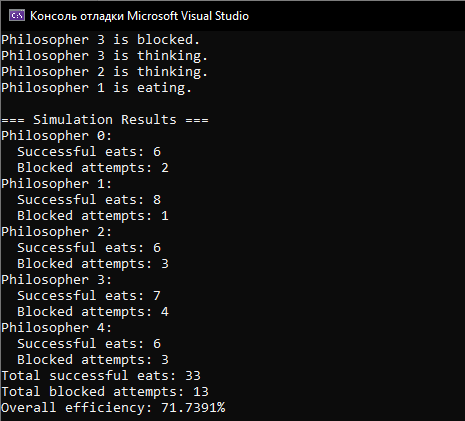


Рисунок 3.2 – Вывод итоговой статистики

Эти данные позволяют проанализировать, какая из стратегий синхронизации лучше справляется с задачей предотвращения блокировок и эффективного использования ресурсов. Выводы из этих данных помогают сделать заключение о том, какая стратегия будет более эффективна в реальных условиях, когда количество философов и уровень конкуренции могут изменяться.

Результат выполнения при стратегии обычной блокировки показан на рисунке 3.3.

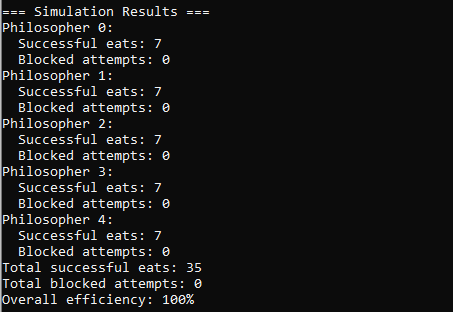


Рисунок 3.3 – Результат выполнения при стратегии обычной блокировки

Результат выполнения при приоритетной стратегии показан на рисунке 3.4.

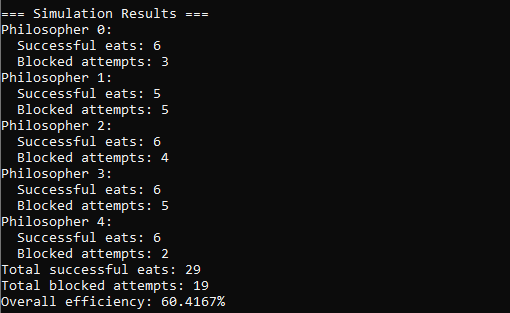


Рисунок 3.4 – Результат выполнения при приоритетной стратегии

При сравнении двух вариантов решения проблемы результат выполнения с использованием стратегии по приоритетам может показаться менее удачным, так как у него эффективность менее 100%, в отличие от выполнения без использования данной стратегии. Это связано с тем, что приоритетная стратегия требует дополнительных шагов, таких как освобождение вилки, если вторая не может быть захвачена, что добавляет небольшие задержки в процесс. Однако, несмотря на снижение общей эффективности, эта стратегия более устойчива и надежна при высоком уровне конкуренции за ресурсы.

Обычная стратегия, хотя и кажется более эффективной в идеальных условиях, может привести к возникновению взаимных блокировок, когда все философы оказываются в ситуации, при которой они все захватили по одной вилке и ожидают освобождения другой. Это создает тупиковую ситуацию (*deadlock*), в которой потоки не могут продолжить выполнение, что ведет к полному зависанию программы. В такой ситуации философы не могут поесть, и программа останавливается. Таким образом, несмотря на более высокую эффективность в теории, обычная стратегия имеет гораздо более высокие риски, связанные с возникновением взаимных блокировок.

Стратегия с приоритетами решает эту проблему, устраняя вероятность взаимных блокировок. Четные философы захватывают вилки в одном порядке, а нечетные – в другом. Такой подход нарушает симметрию и позволяет избежать ситуации, при которой все философы оказываются в блокировке. Когда философ не может захватить вторую вилку, он освобождает первую и повторяет попытку позже. Это добавляет некоторые задержки в выполнение программы, но предотвращает возникновение тупиков и позволяет системе работать стабильно.

Вместо того, чтобы допускать полное зависание системы, как в случае с обычной стратегией, приоритетная стратегия снижает вероятность возникновения блокировок и обеспечивает более предсказуемую работу программы. Конечно, в условиях высокой конкуренции и множества философов, эффективность стратегии с приоритетами может оказаться ниже на 5-10%, так как философы вынуждены ожидать освобождения вилок. Однако за счет предотвращения тупиков и повышения надежности в долгосрочной перспективе приоритетная стратегия оказывается более эффективной для масштабируемых систем, где важно обеспечить стабильную работу программы, избегая полной блокировки ресурсов.

Таким образом, приоритетная стратегия, несмотря на несколько сниженные показатели эффективности в отдельных случаях, является более удачным решением, так как она исключает потенциально разрушительные ситуации взаимных блокировок, обеспечивая более высокую надежность и стабильность работы системы.

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы было реализовано приложение, демонстрирующее синхронизированную работу потоков в задаче «обедающие философы», где философы конкурируют за ограниченные ресурсы (вилки). В программе были реализованы две стратегии синхронизации: стандартная и приоритетная. Для синхронизации потоков использовались мьютексы, которые обеспечивают безопасный доступ к критическим ресурсам, предотвращая одновременное использование вилок несколькими философами.

Каждый философ выполняет две основные задачи – «думает» и «ест». Для этих операций используется блокировка с помощью мьютексов, что исключает возможные ошибки, связанные с одновременным доступом к вилкам. В стандартной стратегии философы захватывают вилки в произвольном порядке, что может привести к взаимной блокировке, в то время как приоритетная стратегия минимизирует вероятность тупика, направляя философов с четным и нечетным идентификаторами на разные вилки в разное время.

Кроме того, программа собирает статистику, позволяющую оценить эффективность каждой из стратегий синхронизации. Проанализированы показатели успешных операций и блокировок, что позволило сделать выводы о производительности каждой стратегии и их влиянии на общую эффективность работы программы.

Использование мьютексов в данной задаче позволило обеспечить корректную синхронизацию потоков, предотвратить гонки данных и гарантировать, что доступ к критическим ресурсам осуществляется безопасно. Выводы, полученные на основе статистики, показали, что приоритетная стратегия, несмотря на небольшие потери в эффективности, обеспечивает стабильную работу программы и избегает взаимных блокировок, что делает ее более предпочтительным решением для задач с высокой конкуренцией за ресурсы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Межпроцессное взаимодействие (IPC) [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/uwp/communication/interprocess-communication

[2] Требования к Критической секции [Электронный ресурс]. – Режим доступа: https://scienceforum.ru/

[3] Межпроцессное взаимодействие [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/

ПРИЛОЖЕНИЕ А  
(обязательное)

**Исходный код программы**

#include <windows.h>

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <thread>

#include <mutex>

#include <atomic>

int NUM\_PHILOSOPHERS = 5;

int THINK\_TIME\_MIN = 1000;

int THINK\_TIME\_MAX = 3000;

int EAT\_TIME\_MIN = 1000;

int EAT\_TIME\_MAX = 2000;

bool USE\_PRIORITY = false;

int TIME\_OUT = 10;

HANDLE\* forks;

HANDLE\* philosophers;

std::mutex print\_mutex;

std::atomic<int>\* successful\_eats;

std::atomic<int>\* blocked\_attempts;

std::atomic<int> total\_eats(0);

std::atomic<int> total\_blocks(0);

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

std::uniform\_int\_distribution<int> think\_dist(THINK\_TIME\_MIN, THINK\_TIME\_MAX);

std::uniform\_int\_distribution<int> eat\_dist(EAT\_TIME\_MIN, EAT\_TIME\_MAX);

DWORD WINAPI philosopher(LPVOID param) {

int id = (int)param;

int leftFork = id;

int rightFork = (id + 1) % NUM\_PHILOSOPHERS;

while (true) {

std::lock\_guard<std::mutex> lock(print\_mutex);

std::cout << "Philosopher " << id << " is thinking.\n";

std::this\_thread::sleep\_for(std::chrono::milliseconds(think\_dist(rng)));

bool acquired = false;

if (USE\_PRIORITY) {

HANDLE index1, index2;

if (id % 2 == 0) {

index1 = forks[leftFork];

index2 = forks[rightFork];

}

else {

index1 = forks[rightFork];

index2 = forks[leftFork];

}

WaitForSingleObject(index1, INFINITE);

if (WaitForSingleObject(index2, TIME\_OUT < 0 ? INFINITE : TIME\_OUT) == WAIT\_OBJECT\_0) { //10

acquired = true;

}

else {

ReleaseMutex(index1);

}

}

else {

WaitForSingleObject(forks[leftFork], INFINITE);

WaitForSingleObject(forks[rightFork], INFINITE);

acquired = true;

}

if (acquired) {

std::lock\_guard<std::mutex> lock(print\_mutex);

std::cout << "Philosopher " << id << " is eating.\n";

successful\_eats[id]++;

total\_eats++;

std::this\_thread::sleep\_for(std::chrono::milliseconds(eat\_dist(rng)));

ReleaseMutex(forks[leftFork]);

ReleaseMutex(forks[rightFork]);

}

else {

blocked\_attempts[id]++;

total\_blocks++;

{

std::lock\_guard<std::mutex> lock(print\_mutex);

std::cout << "Philosopher " << id << " is blocked.\n";

}

}

}

return 0;

}

void print\_statistics() {

std::lock\_guard<std::mutex> lock(print\_mutex);

std::cout << "\n=== Simulation Results ===\n";

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i) {

std::cout << "Philosopher " << i << ":\n";

std::cout << " Successful eats: " << successful\_eats[i] << "\n";

std::cout << " Blocked attempts: " << blocked\_attempts[i] << "\n";

}

std::cout << "Total successful eats: " << total\_eats << "\n";

std::cout << "Total blocked attempts: " << total\_blocks << "\n";

double efficiency = (double)total\_eats / (total\_eats + total\_blocks) \* 100.0;

std::cout << "Overall efficiency: " << efficiency << "%\n";

}

int main() {

std::cout << "Enter number of philosophers: ";

std::cin >> NUM\_PHILOSOPHERS;

std::cout << "Use priority strategy? (1 - yes, 0 - no): ";

std::cin >> USE\_PRIORITY;

if (USE\_PRIORITY) {

std::cout << "Enter timeout(milliseconds), enter -1 for INFINITE timeout: ";

std::cin >> TIME\_OUT;

}

forks = new HANDLE[NUM\_PHILOSOPHERS];

philosophers = new HANDLE[NUM\_PHILOSOPHERS];

successful\_eats = new std::atomic<int>[NUM\_PHILOSOPHERS]();

blocked\_attempts = new std::atomic<int>[NUM\_PHILOSOPHERS]();

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i) {

forks[i] = CreateMutex(NULL, FALSE, NULL);

if (forks[i] == NULL) {

std::cerr << "Failed to create mutex for fork " << i << ".\n";

return 1;

}

successful\_eats[i] = 0;

blocked\_attempts[i] = 0;

}

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i) {

philosophers[i] = CreateThread(

NULL, 0, philosopher, (LPVOID)i, 0, NULL);

if (philosophers[i] == NULL) {

std::cerr << "Failed to create thread for philosopher " << i << ".\n";

return 1;

}

}

std::this\_thread::sleep\_for(std::chrono::seconds(30));

for (int i = 0; i < NUM\_PHILOSOPHERS; ++i) {

TerminateThread(philosophers[i], 0);

CloseHandle(philosophers[i]);

CloseHandle(forks[i]);

}

print\_statistics();

delete[] forks;

delete[] philosophers;

delete[] successful\_eats;

delete[] blocked\_attempts;

return 0;

}