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. 253503
Тимошевич К.С.

Проверил: ассистент кафедры информатики Гриценко Н.Ю.

Минск 2024

СОДЕРЖАНИЕ

1 ПОСТАНОВКА ЗАДАЧИ

Задачей данной лабораторной работы является разработка и реализация модели взаимодействия процессов (или потоков) «Писатели-Читатели», которая позволит имитировать и оценивать принципы синхронного доступа к разделяемым данным. Цель модели: обеспечить корректный доступ к общему ресурсу, избежать возникновения коллизий (таких как «грязное» считывание) и минимизировать блокировки между взаимодействующими потоками.

Моделирование предполагает выполнение обращений процессов к данным с параметрами, характеризующими момент обращения, длительность его выполнения и задержки между запросами. Поскольку модель служит для демонстрации и оценки алгоритмов взаимодействия, данные могут быть как реальными (тестовыми), так и симулированными, не представляя реальных значений. Функциональность модели также предусматривает расчет и представление ключевых результатов: соотношение времени активности и блокировки потоков, успешные и неуспешные обращения.

2 ОПИСАНИЕ РАБОТЫ ПРОГРАММЫ

В этом разделе будут рассмотрены основные функции программы, которые были задействованы для выполнения работы.

2.1 Чтение данных

Функция *reader* отвечает за чтение данных из общего ресурса. Каждый поток читателя сначала блокирует доступ к счетчику читателей с помощью критической секции (*countLock*). Если присутствуют ожидающие писатели и это первый читатель, функция откладывает чтение, увеличивая счетчик неудачных операций и освобождая *countLock*. В противном случае поток увеличивает счетчик читателей и выполняет чтение данных, после чего снижает значение счетчика и, если он последний читатель, освобождает семафор *readSemaphore*, разрешая запись. Время блокировки и время операции чтения учитываются для анализа эффективности программы.

2.2 Запись данных

Функция *writer* обеспечивает запись данных в разделяемую память. Поток писателя пытается захватить мьютекс *writeMutex* с тайм-аутом 100 миллисекунд. Если тайм-аут истекает, поток фиксирует неудачное обращение, увеличивая счетчик неудачных записей, и продолжает ожидание. При успешной блокировке *writeMutex* поток вносит изменения в разделяемую память и освобождает мьютекс [1], что позволяет другому писателю или читателю захватить доступ к ресурсу. Время блокировки и время записи также фиксируются в статистике для дальнейшего анализа.

2.3 Синхронизация через мьютексы и семафоры

Взаимодействие читателей и писателей организовано через мьютекс *writeMutex*, семафор *readSemaphore* и критическую секцию *countLock*. Критическая секция защищает общий счетчик читателей, исключая ситуации гонок, и обеспечивает, что писатели могут захватить доступ к ресурсу только в отсутствие активных читателей [2].

3 ХОД ВЫПОЛНЕНИЯ ПРОГРАММЫ

3.1 Примеры выполнения задания

На рисунке 3.1 представлена часть результата работы программы с несколькими потоками reader и writer при попытке записать и прочитать данные из разделяемой памяти. Модуль *writer* записывает данные, увеличивая общий счетчик *shared_data* с защитой доступа через мьютекс, а модуль *reader* выводит текущее значение *shared_data*, блокируя доступ к разделяемой памяти. На первом этапе *writer* успешно записывает значение в память, после чего *reader* получает доступ к разделяемой памяти и считывает данные. В случае, если доступ к памяти заблокирован, *reader* или *writer* фиксирует неудачную попытку в статистике.

Рисунок 3.1 - Вывод результатов в момент выполнения программы

На рисунке 3.2 представлен вывод сообщения из *reader* о том, что в данный момент доступ к данным заблокирован модулем *writer*.

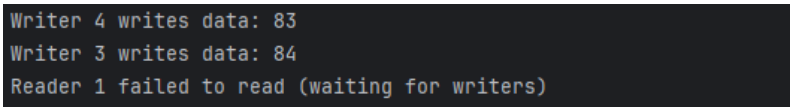

Рисунок 3.2 - Вывод сообщения о невозможности чтения

На рисунке 3.3 приведен пример вывода итоговой статистики, включающий общее количество успешных и неуспешных попыток обращения, среднее время выполнения операций, а также общее время блокировки потоков.

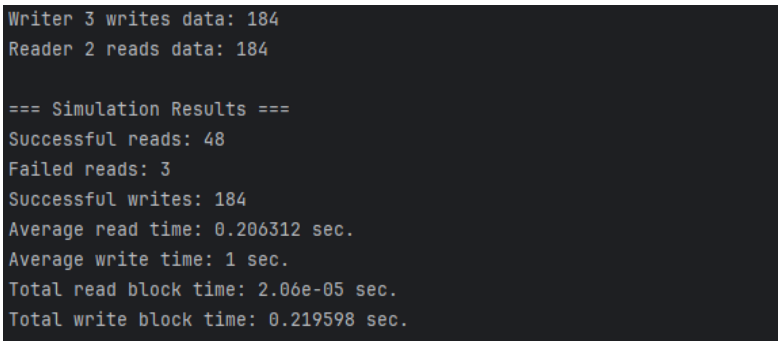

Рисунок 3.3 - Вывод статистики

ВЫВОД

В ходе выполнения лабораторной работы была реализована модель взаимодействия потоков по принципу «Писатели-Читатели» с использованием механизмов синхронизации и разделяемых данных. Разработанные модули *reader* и *writer* обеспечивают безопасный доступ к общим данным через разделяемую переменную *shared_data*. Для предотвращения конкурентного доступа используются мьютексы и критические секции, что исключает ситуации гонок и «грязного» чтения данных [3].

В результате тестирования стало ясно, что система корректно управляет доступом потоков к общим данным и отслеживает количество успешных и неудачных операций чтения и записи. В итоге была собрана информация о количестве операций, времени их выполнения и ожидания.

Таким образом, лабораторная работа показала на практике, как можно организовать доступ к общим ресурсам в многопоточных приложениях, используя синхронизацию и работу с разделяемой памятью в *Windows*.

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: 12.10.2024.

[2] API Win32 Critical section [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/sync/critical-section-objects. – Дата доступа: 12.10.2024.

[3] WaitForSingleObject [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/windows/win32/api/synchapi/nf-synchapi-waitforsingleobject. – Дата доступа: 12.10.2024.

ПРИЛОЖЕНИЕ А
(справочное)
Исходный код

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

HANDLE readSemaphore;
HANDLE writeMutex;
int readersCount = 0;
int writersWaiting = 0;
int shared_data = 0;
CRITICAL_SECTION countLock;

std::mutex statsMutex;
int successfulReads = 0;
int successfulWrites = 0;
int failedReads = 0;
int failedWrites = 0;
double totalReadTime = 0.0;
double totalWriteTime = 0.0;
double totalReadBlockTime = 0.0;
double totalWriteBlockTime = 0.0;

// Flag for stopping threads
std::atomic<bool> isRunning(true);

void reader(int id, int duration) {
    while (isRunning) {
        auto blockStartTime = std::chrono::high_resolution_clock::now();
        EnterCriticalSection(&countLock);
        if (readersCount == 0 && writersWaiting > 0) {
            LeaveCriticalSection(&countLock);
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
            {
                std::lock_guard<std::mutex> lock(statsMutex);
                failedReads++;
                std::cout << "Reader " << id << " failed to read (waiting for
writers)" << std::endl;
            }
            continue;
        }

        readersCount++;
        LeaveCriticalSection(&countLock);
        auto blockEndTime = std::chrono::high_resolution_clock::now();
        totalReadBlockTime += std::chrono::duration<double>(blockEndTime -
blockStartTime).count();

        auto startTime = std::chrono::high_resolution_clock::now();
        std::this_thread::sleep_for(std::chrono::milliseconds(duration));
        std::cout << "Reader " << id << " reads data: " << shared_data <<
std::endl;
        auto endTime = std::chrono::high_resolution_clock::now();
        EnterCriticalSection(&countLock);
        readersCount--;
```

```
        if (readersCount == 0) {
            ReleaseSemaphore(readSemaphore, 1, NULL);
        }
        LeaveCriticalSection(&countLock);
        {
            std::lock_guard<std::mutex> lock(statsMutex);
            successfulReads++;
            totalReadTime += std::chrono::duration<double>(endTime -
startTime).count();
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void writer(int id, int duration) {
    while (isRunning) {
        auto blockStartTime = std::chrono::high_resolution_clock::now();
        ++writersWaiting;
        DWORD waitResult = WaitForSingleObject(writeMutex, 10); // Тайм-аут 10
миллисекунд
        if (waitResult == WAIT_TIMEOUT) {
            std::lock_guard<std::mutex> lock(statsMutex);
            failedWrites++;
           // totalWriterWaitTime +=
std::chrono::duration<double>(std::chrono::high_resolution_clock::now() -
waitStartTime).count();
            --writersWaiting;
            // Логирование времени ожидания
            std::cout << "Writer " << id << " timed out waiting to write." <<
std::endl;
            continue;
        }
        auto blockEndTime = std::chrono::high_resolution_clock::now();
        totalWriteBlockTime += std::chrono::duration<double>(blockEndTime -
blockStartTime).count();

        shared_data++;
        std::cout << "Writer " << id << " writes data: " << shared_data <<
std::endl;
        ReleaseMutex(writeMutex);
        --writersWaiting;

        {
            std::lock_guard<std::mutex> lock(statsMutex);
            successfulWrites++;
            totalWriteTime += duration / 1000.0;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(100));
    }
}

void cleanup() {
    CloseHandle(readSemaphore);
    CloseHandle(writeMutex);
    DeleteCriticalSection(&countLock);
}

int main() {
    int numReaders = 3;
    int numWriters = 4;
    int readDuration = 200;
```

```
    int writeDuration = 1000;

    readSemaphore = CreateSemaphore(NULL, numReaders, numReaders, NULL);
    writeMutex = CreateMutex(NULL, FALSE, NULL);
    InitializeCriticalSection(&countLock);
    std::vector<std::thread> threads;

    for (int i = 0; i < numReaders; ++i) {
        threads.emplace_back(reader, i + 1, readDuration);
    }

    for (int i = 0; i < numWriters; ++i) {
        threads.emplace_back(writer, i + 1, writeDuration);
    }
    std::this_thread::sleep_for(std::chrono::seconds(5));
    isRunning = false;
    for (auto& th : threads) {
        th.join();
    }

    std::cout << "\n=== Simulation Results ===\n";
    std::cout << "Successful reads: " << successfulReads << std::endl;
    std::cout << "Failed reads: " << failedReads << std::endl;
    std::cout << "Successful writes: " << successfulWrites << std::endl;
   // std::cout << "Failed writes: " << failedWrites << std::endl;
    std::cout << "Average read time: " << (successfulReads ? totalReadTime /
successfulReads : 0) << " sec." << std::endl;
    std::cout << "Average write time: " << (successfulWrites ? totalWriteTime /
successfulWrites : 0) << " sec." << std::endl;
    std::cout << "Total read block time: " << totalReadBlockTime << " sec." <<
std::endl;
    std::cout << "Total write block time: " << totalWriteBlockTime << " sec."
<< std::endl;
    cleanup();
    return 0;
}
```