Лабораторная работа №4. Синхронизация потоков

# Цель лабораторной работы

Изучить классические задачи синхронизации. Обеспечить правильную работу программы с использованием различных примитивов синхронизации.

# Теоретический материал

*Основная литература:*

Гергель В.П. Высокопроизводительные вычисления для многоядерных многопроцессорных систем. Глава 4.

*Более подробно можно посмотреть в*

Эндрюс Г.Р. Основы многопоточного, параллельного и распределенного программирования. Часть 1.

# 4.3. Задание на лабораторную работу

Во всех задачах выбор языка программирования ложится на автора программы. Постарайтесь обосновать, почему Вы выбрали тот или иной язык и те или иные средства языка.

1. *«Производители – потребители»*. Два потока обрабатывают заявки. Первый поток читает заявку и помещает её в очередь размера *N*, если очередь переполнена, то заявке отказывается. Второй поток берет заявки из очереди и обрабатывает их, если в очереди заявок нет, то поток засыпает. Написать параллельную программу, реализующую правильную работу этих потоков.
2. *«Фоновая обработка»*. Два потока обрабатывают файл, содержащий набор целочисленных массивов. В файле находится *M* массивов, каждый в отдельной строке. В первой строке указано число *M* – количество массивов, а в следующих *M* строках записаны элементы массивов, разделенные пробелами. Первый поток по очереди читает массивы из файла, а второй по очереди считает суммы элементов этих массивов. Результаты первый поток

должен записать в новый файл. Каждую сумму – в отдельной строке. Организовать правильную работу потоков.

1. *«Читатели – писатели»*. Имеется хранилище данных, с которым работают одновременно несколько потоков. Первые *N* потоков случайным образом изменяют данные в хранилище (при этом изменять данные в некоторый момент времени может ровно один поток), остальные *M* потоков периодически читают данные из хранилища. Несколько читателей могут находиться в хранилище одновременно, но читатель и писатель не могут одновременно использовать хранилище.

Эмулировать работу хранилища.

# 4.4. Результаты лабораторной работы

Результаты лабораторной работы представляются в виде отчета по лабораторной работе. В отчет включается титульный лист, цель работы, задание на лабораторную работу, описание и обоснование правильности алгоритма, листинг с комментариями, скриншоты, доказывающие правильность работы программы, полученные результаты и выводы по лабораторной работе.

Отчет оформляется в электронном виде и высылается на e-mail [pg\_chuprakov@vyatsu.ru](mailto:pg_chuprakov@vyatsu.ru) (в теме или тексте письма, а также в названии документа с отчетом должны фигурировать ФИ студента, его группа, номер лабораторной работы).

Лабораторная работа считается зачтенной после её устной защиты у преподавателя.

Код

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <queue>

#include <chrono>

#include <random>

using namespace std;

const int QUEUE\_SIZE = 10; // Размер очереди

queue<int> requests; // Очередь заявок

mutex m; // Мьютекс для синхронизации доступа к очереди

condition\_variable cv; // Условная переменная для уведомления о состоянии очереди

// Функция для генерации случайной задержки

int randomDelay(int min, int max) {

random\_device rd;

mt19937 gen(rd());

uniform\_int\_distribution<> distrib(min, max);

return distrib(gen);

}

// Функция производителя

void producer(int min\_delay, int max\_delay) {

int request\_id = 1;

while (true) {

{

unique\_lock<mutex> lock(m);

// Ждем, пока в очереди появится место

cv.wait(lock, [] { return requests.size() < QUEUE\_SIZE; });

// Добавляем заявку в очередь

requests.push(request\_id++);

cout << "Producer: task added " << request\_id - 1 << endl;

// Уведомляем потребителя о появлении заявки

cv.notify\_one();

}

// Засыпаем на случайное время

this\_thread::sleep\_for(chrono::milliseconds(randomDelay(min\_delay, max\_delay)));

}

}

// Функция потребителя

void consumer(int min\_delay, int max\_delay) {

while (true) {

{

unique\_lock<mutex> lock(m);

// Проверяем наличие заявок без ожидания

if (!requests.empty()) {

// Извлекаем заявку из очереди

int request\_id = requests.front();

requests.pop();

cout << "Consumer: task processed " << request\_id << endl;

// Уведомляем производителя о появлении свободного места

cv.notify\_one();

}

}

// Засыпаем на случайное время

this\_thread::sleep\_for(chrono::milliseconds(randomDelay(min\_delay, max\_delay)));

}

}

int main() {

// Параметры задержек

int producer\_min\_delay = 10; // Минимальная задержка производителя (ms)

int producer\_max\_delay = 20; // Максимальная задержка производителя (ms)

int consumer\_min\_delay = 200; // Минимальная задержка потребителя (ms)

int consumer\_max\_delay = 800; // Максимальная задержка потребителя (ms)

// Создаем потоки производителя и потребителя

thread producer\_thread(producer, producer\_min\_delay, producer\_max\_delay);

thread consumer\_thread(consumer, consumer\_min\_delay, consumer\_max\_delay);

// Ожидаем завершения потоков

producer\_thread.join();

consumer\_thread.join();

return 0;

}

#include <iostream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <semaphore>

#include <chrono>

#include <functional>

// Класс для представления общего ресурса

class SharedResource {

public:

static constexpr int MAX\_READERS = 10;

SharedResource() : rsemaphore(std::make\_shared<std::counting\_semaphore<MAX\_READERS>>(MAX\_READERS)) {}

std::shared\_ptr<std::counting\_semaphore<MAX\_READERS>> rsemaphore;

std::mutex wlock;

};

// Класс для представления читателя

class Reader {

public:

Reader(const std::string& name, std::shared\_ptr<SharedResource> resource) :

name(name), resource(resource) {}

void read() {

std::unique\_lock<std::mutex> lock(resource->wlock);

// Ждем, пока мьютекс записи не будет разблокирован

while (resource->wlock.try\_lock()) {

lock.unlock();

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

lock.lock();

}

resource->rsemaphore->acquire(); // Захватываем разрешение на чтение

std::cout << "Reader " << name << " reading data" << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(1));

std::cout << "Reader " << name << " reading done" << std::endl;

resource->rsemaphore->release(); // Освобождаем разрешение на чтение

}

private:

std::string name;

std::shared\_ptr<SharedResource> resource;

};

// Класс для представления писателя

class Writer {

public:

Writer(const std::string& name, std::shared\_ptr<SharedResource> resource) :

name(name), resource(resource) {}

void write() {

// Ждем, пока все разрешения на чтение не будут освобождены

while (resource->rsemaphore->try\_acquire() == false) {

std::this\_thread::sleep\_for(std::chrono::milliseconds(100));

}

std::lock\_guard<std::mutex> lock(resource->wlock);

std::cout << "Writer " << name << " writing data" << std::endl;

std::this\_thread::sleep\_for(std::chrono::seconds(2));

std::cout << "Writer " << name << " writing done" << std::endl;

}

private:

std::string name;

std::shared\_ptr<SharedResource> resource;

};

int main() {

// Создаем общий ресурс

auto sr = std::make\_shared<SharedResource>();

// Создаем объекты читателей и писателей

Reader reader1("reader1", sr);

Reader reader2("reader2", sr);

Reader reader3("reader3", sr);

Writer writer1("writer1", sr);

Writer writer2("writer2", sr);

// Запускаем потоки

std::thread t1(std::bind(&Reader::read, &reader1));

std::thread t2(std::bind(&Writer::write, &writer1));

std::this\_thread::sleep\_for(std::chrono::seconds(2));

std::thread t3(std::bind(&Reader::read, &reader2));

std::thread t4(std::bind(&Writer::write, &writer2));

std::thread t5(std::bind(&Reader::read, &reader3));

// Ожидаем завершения потоков

t1.join();

t2.join();

t3.join();

t4.join();

t5.join();

std::cout << "All threads finished!" << std::endl;

return 0;

}

#include <iostream>

#include <fstream>

#include <thread>

#include <mutex>

#include <condition\_variable>

#include <queue>

#include <sstream>

#include <vector>

using namespace std;

queue<vector<int>> dataQueue; // Queue for storing arrays

mutex m; // Mutex for synchronizing access to the queue

condition\_variable cv; // Condition variable to notify about queue status

// Function for reading arrays from a file

void readerThread(const string& inputFilename) {

ifstream inputFile(inputFilename);

if (!inputFile.is\_open()) {

cerr << "Error opening input file: " << inputFilename << endl;

return;

}

string line;

getline(inputFile, line);// Read the number of arrays

int M = std::stoi(line);

for (int i = 0; i < M; ++i) {

vector<int> data;

string line;

getline(inputFile, line); // Read a line with an array

stringstream ss(line);

int num;

while (ss >> num) {

data.push\_back(num);

}

{

unique\_lock<mutex> lock(m);

dataQueue.push(data); // Add the array to the queue

cv.notify\_one(); // Notify the calculator thread

}

}

inputFile.close();

}

// Function for calculating the sum of array elements

void calculatorThread(const string& outputFilename) {

ofstream outputFile(outputFilename);

if (!outputFile.is\_open()) {

cerr << "Error opening output file: " << outputFilename << endl;

return;

}

while (true) {

unique\_lock<mutex> lock(m);

cv.wait(lock, [] { return !dataQueue.empty(); }); // Wait for data to appear in the queue

vector<int> data = dataQueue.front();

dataQueue.pop();

// Calculate the sum of array elements

int sum = 0;

for (int num : data) {

sum += num;

}

outputFile << sum << endl; // Write the sum to the file

}

}

int main() {

string inputFilename = "C:\\MyStuff\\Programming\\ParallelProgramming\\Lab4\\input.txt";

string outputFilename = "C:\\MyStuff\\Programming\\ParallelProgramming\\Lab4\\output.txt";

// Create threads

thread reader(readerThread, inputFilename);

thread calculator(calculatorThread, outputFilename);

// Wait for threads to complete

reader.join();

calculator.join();

return 0;

}