# Лабораторная работа № 3 «Конкурентное программирование»

## Цель работы

Изучение объектно-ориентированного программирования и получение навыков написания программ, содержащих конкурентное программирование на примере языка программирования C++.

## Задание

Создать программу на основе второй лабораторной работы, которая должна поэлементно (в соответствии с таблицей 1.1) разбирать все документы определённого формата, находящиеся в директории. Имя директории передаётся как аргумент командной строки. Исходный код должен быть написан с использованием механизмов конкурентного программирования (использовать стандартную библиотеку C++) и в соответствии с вариантом (см. ниже).

### Вариант 1

Извлечение элементов производить в главном треде, работу с файлами во вспомогательном / вспомогательных.

### Вариант 2

Извлечение элементов производить во вспомогательном / вспомогательных треде, работу с файлами в главном.

### Вариант 3

Извлечение элементов производить во вспомогательном / вспомогательных треде, работу с файлами во вспомогательном / вспомогательных, главный поток синхронизирует их работу.

## Теория

### \*POSIX

Не забудьте подключить библиотеку pthread, если используете POSIX:

g++ -std=c++17 techpr03.cpp -o techpr03 –lpthread

./techpr03

### Пример с std::thread

Класс thread представляет собой один тред выполнения. Треды позволяют нескольким функциям выполняться одновременно.

#include <iostream>

#include <thread>

#include <vector>

static void func(int x)

{

// Возвращает идентификатор текущего треда.

std::cout << "Hello from thread " << std::this\_thread::get\_id()

<< "." << std::endl;

std::cout << "Argument passed in: " << x

<< "." << std::endl;

}

int main()

{

std::vector<std::thread> myThreads;

for (int i { 0 }; i < 10; ++i)

{

myThreads.push\_back(std::thread{ func, i });

}

// Ждёт, пока myThreads завершат свои выполнения.

for (int i { 0 }; i < myThreads.size(); ++i) // for (auto& i : myThreads)

{

myThreads[i].join(); // i.join();

}

std::cout << "Hello from main thread." << std::endl;

return 0;

}

Output:

Hello from thread 50556.Hello from thread 32788.Hello from thread 17648.Hello from thread 29064.

Hello from thread 30472.

Argument passed in: Hello from thread 34364.

Argument passed in: 3.

...

### Пример с std::mutex

Класс mutex — это примитив синхронизации, который можно использовать для защиты общих данных от одновременного доступа нескольких тредов.

Мьютекс предлагает исключительную, нерекурсивную семантику владения:

* Вызывающий тред владеет мьютексом с момента успешного вызова lock или try\_lock до вызова unlock.
* Когда тред владеет мьютексом, все остальные треды будут блокироваться (для вызовов lock) или получат ложное возвращаемое значение (для try\_lock), если они попытаются заявить о праве собственности на мьютекс.
* Вызывающий тред не должен владеть мьютексом до вызова lock или try\_lock.

#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

#include <chrono>

#include <ctime>

std::mutex gLock;

static int sharedValue{ 0 };

void sharedValueIncrement()

{

// Блокирует мьютекс; Другой тред ждёт разблокировки.

gLock.lock();

// Критическая секция

int delay{ std::rand() % 100 };

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

sharedValue++;

// Разблокирует мьютекс.

gLock.unlock();

}

int main()

{

std::srand(std::time(nullptr));

std::vector<std::thread> myThreads;

for (int i { 0 }; i < 100; ++i)

{

myThreads.push\_back(std::thread{ sharedValueIncrement });

}

for (int i { 0 }; i < myThreads.size(); ++i)

{

myThreads[i].join();

}

std::cout << "Shared value: " << sharedValue << "." << std::endl;

return 0;

}

Output:

Shared value: 100.

### Пример с std:: lock\_guard

Класс lock\_guard — это оболочка мьютекса, которая предоставляет удобный механизм в стиле RAII для владения мьютексом на время действия блока с ограниченной областью действия.

Когда создается объект lock\_guard, он пытается стать владельцем данного ему мьютекса. Когда управление покидает область, в которой был создан объект lock\_guard, lock\_guard разрушается, а мьютекс освобождается.

#include <iostream>

#include <thread>

#include <vector>

#include <mutex>

#include <chrono>

#include <ctime>

std::mutex gLock;

static int sharedValue{ 0 };

void sharedValueIncrement()

{

// Блокирует мьютекс; Другой тред ждёт разблокировки.

// lock\_guard -- обёртка для мьютекса; "RAII-мьютекс".

std::lock\_guard<std::mutex> lockGuard{ gLock };

// Критическая секция

int delay{ std::rand() % 100 };

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

sharedValue++;

// Разблокирует мьютекс при выходе из области действия.

}

int main()

{

std::srand(std::time(nullptr));

std::vector<std::thread> myThreads;

for (int i { 0 }; i < 100; ++i)

{

myThreads.push\_back(std::thread{ sharedValueIncrement });

}

for (int i { 0 }; i < myThreads.size(); ++i)

{

myThreads[i].join();

}

std::cout << "Shared value: " << sharedValue << "." << std::endl;

return 0;

}

Output:

Shared value: 100.

### Пример с std::future и std::async

Шаблон класса std::future предоставляет механизм для доступа к результату асинхронных операций:

* Асинхронная операция (созданная с помощью std::async, std::packaged\_task или std::promise) может предоставить объект std::future создателю этой асинхронной операции.
* Затем создатель асинхронной операции может использовать различные методы для запроса, ожидания или извлечения значения из std::future. Эти методы могут блокироваться, если асинхронная операция еще не предоставила значение.
* Когда асинхронная операция готова отправить результат создателю, она может сделать это, изменив общее состояние (например, std::promise::set\_value), связанное с std::future создателя.

Шаблон функции std::async запускает функцию f асинхронно (возможно, в отдельном треде, который может быть частью пула тредов) и возвращает std::future, который в конечном итоге будет содержать результат вызова этой функции.

#include <iostream>

#include <future>

int square(int x) { return x \* x; }

int main()

{

std::future<int> asyncFunction{ std::async(&square, 13) };

// Заблокировать на операции `get()`, ...

// ... пока результат не будет вычислен.

int result{ asyncFunction.get() };

std::cout << "Result: " << result << "." << std::endl;

return 0;

}

Output:

Result: 169.

### Пример с std::future и std::async 2

#include <iostream>

#include <future>

#include <thread>

#include <chrono>

bool bufferedFileLoader()

{

size\_t bytesLoaded{ 0 };

while (bytesLoaded < 10000)

{

std::cout << "Loading file..." << std::endl;

std::this\_thread::sleep\_for(std::chrono::milliseconds(250));

bytesLoaded += 1000;

}

return true;

}

int main()

{

std::future<bool> backgroundThread{

std::async(std::launch::async, bufferedFileLoader) };

std::future\_status status;

// Основной цикл программы.

while (true)

{

std::cout << "Main thread is running." << std::endl;

// Искусственный сон для нашей программы.

std::this\_thread::sleep\_for(std::chrono::milliseconds(50));

status = backgroundThread.wait\_for(std::chrono::milliseconds(1));

// Если наши данные готовы, т. е. наш фоновый поток завершен, ...

// ... то выйти из цикла.

if (status == std::future\_status::ready)

{

std::cout << "Our data is ready..." << std::endl;

break;

}

}

std::cout << "Program is complete." << std::endl;

return 0;

}

Output:

Main thread is running.

Loading file...

Main thread is running.

Main thread is running.

Main thread is running.

Loading file...

Main thread is running.

Main thread is running.

Main thread is running.

...

Our data is ready...

Program is complete.