**ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНТЕЛЕКТУАЛЬНИХ**

**ТЕХНОЛОГІЙ І ЗВ’ЯЗКУ**

**Звіт**

**з дисципліни Технологіх Розподілених Систем та Паралельних Обчислень**

**Лабораторна робота №1**

Виконав: студент 3 курсу, групи ІПЗ-4.04 спеціальності

121 Інженерія програмного забезпечення

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Бухта М.М.

Перевірив\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_Развалінов В.Ю.

**Одеса  2024**

**ЗАВДАННЯ**

**Опис завдання:**

1. Створіть клас, метод main() якого виводить на консоль слова речення «We have the java learning course!» із затримкою 1 секунда.
2. Створіть два потоки, один з яких виводить на консоль символ ‘-‘, а інший – символ ‘|’. Запустіть потоки в основній програмі так, щоб вони по черзі виводили свої символи в рядок. Виведіть на консоль 10 таких рядків.
3. Створіть клас Counter з методами increment() та decrement(), які збільшують та зменшують значення лічильника відповідно. Створіть два потоки, один з яких збільшує 1000 разів значення лічильника, а інший – зменшує 1000 разів значення лічильника. Використовуючи методи sleep() та/або join() добийтесь правильної роботи лічильника при одночасній роботі з ним двох потоків.
4. Реалізуйте стрічковий алгоритм або алгоритм Фокса множення матриць. Результат множення записуйте в об’єкт класу Result. Виконайте експерименти з різною кількістю потоків та різною розмірністю матриць, які перемножуються, реєструючи час виконання. Побудуйте графіки відповідних залежностей.

**Код програми:**

**main.cpp**

#include "Core/Starter.hpp"

int main(int argc, char \*argv[]) {

auto app = Core::Starter::create(argc, argv);

return app->main();

}

**Core/Starter.hpp**

#ifndef \_\_LAB1\_SOURCEFILES\_CORE\_STARTER\_HPP\_\_

#define \_\_LAB1\_SOURCEFILES\_CORE\_STARTER\_HPP\_\_

#include <QObject>

#include <QSharedPointer>

#include <QAtomicPointer>

#include <QCoreApplication>

namespace Core {

class Starter : public QObject {

Q\_OBJECT

public:

static QSharedPointer<Starter> create(int argc, char \*argv[], QObject\* parent = nullptr);

int main();

protected:

virtual int executeApp();

protected:

Starter() = delete;

Starter(int argc, char \*argv[], QObject\* parent = nullptr);

Q\_DISABLE\_COPY(Starter);

void init\_connections();

protected:

QCoreApplication \_qt\_app;

signals:

void finish();

private:

int \_argc;

char \*\* \_argv;

};

} // namespace Core

#endif // \_\_LAB1\_SOURCEFILES\_CORE\_STARTER\_HPP\_\_

**Core/Starter.cpp**

#include "Starter.hpp"

#include "Launcher/LaboratoryStarter.hpp"

namespace Core {

QSharedPointer<Starter> Starter::create(int argc, char \*argv[], QObject\* parent) {

static QSharedPointer<Starter> instance(new Launcher::Starter(argc, argv, parent));

return instance;

}

int Starter::main() {

int ret = executeApp();

return ret;

}

int Starter::executeApp() {

return \_qt\_app.exec();

}

Starter::Starter(int argc, char \*argv[], QObject\* parent)

: QObject{parent}

, \_argc{argc}

, \_argv{argv}

, \_qt\_app{argc, argv} {

init\_connections();

}

void Starter::init\_connections() {

connect(this, &Starter::finish, &\_qt\_app, &QCoreApplication::quit);

}

} // namespace Core

**Core/BaseInfoDisplayer.hpp**

#ifndef \_\_LAB1\_SOURCEFILES\_CORE\_BASEINFODISPLAYER\_HPP\_\_

#define \_\_LAB1\_SOURCEFILES\_CORE\_BASEINFODISPLAYER\_HPP\_\_

#include <QRunnable>

#include <QString>

namespace Core {

class BaseInfoDisplayer : public QRunnable {

public:

BaseInfoDisplayer(void);

void run() override;

private:

QString \_msg;

};

} // Core

#endif // \_\_LAB1\_SOURCEFILES\_CORE\_BASEINFODISPLAYER\_HPP\_\_

**Core/BaseInfoDisplayer.cpp**

#include "BaseInfoDisplayer.hpp"

#include <QDebug>

#include <QThread>

#include <iostream>

namespace Core {

BaseInfoDisplayer::BaseInfoDisplayer()

: \_msg{"Лабораторна робота №1\nЗ дисципліни Технології Розподілених Систем та Паралельних Обчислень\nСтудента групи ІПЗ-3.04\nБухти Микити"}

{

}

void BaseInfoDisplayer::run() {

for (qint64 i{0}; i < \_msg.size(); ++i) {

std::wcout << static\_cast<wchar\_t>(\_msg[i].unicode()) << std::flush;

QThread::msleep(50);

}

std::wcout << std::endl;

}

} // Core

**Launcher/LaboratoryStarter.hpp**

#ifndef \_\_LAB1\_LAUNCHER\_LABORATORYSTARTER\_HPP\_\_

#define \_\_LAB1\_LAUNCHER\_LABORATORYSTARTER\_HPP\_\_

#include "Core/Starter.hpp"

#include <QThreadPool>

namespace Launcher {

class Starter final : public Core::Starter {

public:

Starter(int argc, char \*argv[], QObject\* parent = nullptr);

Q\_DISABLE\_COPY(Starter);

private:

int executeApp() override;

private:

QThreadPool\* \_pool;

};

} // Launcher

#endif // \_\_LAB1\_LAUNCHER\_LABORATORYSTARTER\_HPP\_\_

**Launcher/LaboratoryStarter.cpp**

#include "LaboratoryStarter.hpp"

#include "Core/BaseInfoDisplayer.hpp"

#include "Tasks/Task1.hpp"

#include "Tasks/Task2.hpp"

#include "Tasks/Task3.hpp"

#include "Tasks/Task4.hpp"

#include <QCoreApplication>

#include <QDebug>

#include <QTimer>

namespace Launcher {

Starter::Starter(int argc, char \*argv[], QObject\* parent)

: Core::Starter(argc, argv, parent) {

\_pool = QThreadPool::globalInstance();

\_pool->setMaxThreadCount(1);

}

int Starter::executeApp() {

\_pool->start(new Core::BaseInfoDisplayer());

\_pool->start(new Tasks::Task1);

\_pool->start(new Tasks::Task2);

\_pool->start(new Tasks::Task3);

\_pool->start(new Tasks::Task4);

\_pool->waitForDone();

QTimer::singleShot(10, [this]() {

emit this->finish();

});

return Core::Starter::executeApp();

}

} // Launcher

**Tasks/Task1.hpp**

#ifndef \_\_LAB1\_TASKS\_TASK1\_HPP\_\_

#define \_\_LAB1\_TASKS\_TASK1\_HPP\_\_

#include <QRunnable>

namespace Tasks {

class Task1 : public QRunnable {

public:

void run() override;

};

} // Tasks

#endif // \_\_LAB1\_TASKS\_TASK1\_HPP\_\_

**Tasks/Task1.cpp**

#include "Task1.hpp"

#include <QDebug>

#include <QThread>

#include <string>

#include <iostream>

namespace Tasks {

void Task1::run() {

qDebug() << \_\_PRETTY\_FUNCTION\_\_ << "called";

std::wstringstream str{L"We have the Java learning course!"};

std::wstring word;

while(std::getline(str, word, L' ')) {

std::wcout << word << L' ' << std::flush;

QThread::msleep(1000);

}

std::wcout << std::endl;

}

} // Tasks

**Tasks/Task2.hpp**

#ifndef \_\_LAB2\_TASKS\_TASK2\_HPP\_\_

#define \_\_LAB2\_TASKS\_TASK2\_HPP\_\_

#include <QRunnable>

#include <QObject>

#include <QWaitCondition>

#include <QMutex>

namespace Tasks {

class Task2 : public QRunnable {

private:

enum class TurnEnumeration : quint8 {

Dash = 0,

Pipe = 1

};

public:

void run() override;

virtual ~Task2() = default;

private:

};

class ConsoleSyncWriter : public QObject {

public:

ConsoleSyncWriter(const std::wstring& str, const qint32 turn\_number);

public:

static qint32 get\_terminal\_width();

public slots:

void write(QAtomicInt& current\_turn, qint32 next\_turn);

void write\_repeatly(QAtomicInt& current\_turn, qint32 next\_turn, qint32 repeat\_count);

private:

std::wstring \_str;

const qint32 \_turn\_number;

static QWaitCondition \_waiter;

static QMutex \_write\_mutex;

};

} // namespace Tasks

#endif // \_\_LAB2\_TASKS\_TASK2\_HPP\_\_

**Tasks/Task2.cpp**

#include "Task2.hpp"

#include <QDebug>

#include <QThread>

#include <iostream>

#include <string>

#include <sys/ioctl.h>

namespace Tasks {

QWaitCondition ConsoleSyncWriter::\_waiter;

QMutex ConsoleSyncWriter::\_write\_mutex;

void Task2::run() {

qDebug() << \_\_PRETTY\_FUNCTION\_\_ << "called";

auto console\_writer = [](const std::wstring& str)->void {

std::wcout << str;

};

QThread dash\_th, pipe\_th;

ConsoleSyncWriter dash\_writer{L"-", static\_cast<qint32>(TurnEnumeration::Dash)};

ConsoleSyncWriter pipe\_writer{L"|", static\_cast<qint32>(TurnEnumeration::Pipe)};

qint8 rows\_count = 10;

qint8 colums\_count = ConsoleSyncWriter::get\_terminal\_width();

QAtomicInt current\_turn = static\_cast<qint32>(TurnEnumeration::Dash);

dash\_writer.moveToThread(&dash\_th);

pipe\_writer.moveToThread(&pipe\_th);

QObject::connect(&dash\_th, &QThread::started, [&](){

dash\_writer.write\_repeatly(current\_turn, static\_cast<qint32>(TurnEnumeration::Pipe), rows\_count \* colums\_count / 2);

dash\_th.quit(); // in case if quit isn't used, the main thread is locked on the wait() method called (str 46);

});

QObject::connect(&pipe\_th, &QThread::started, [&](){

pipe\_writer.write\_repeatly(current\_turn, static\_cast<qint32>(TurnEnumeration::Dash), rows\_count \* colums\_count / 2);

pipe\_th.quit();

});

dash\_th.start();

pipe\_th.start();

// There is a framework bug?

dash\_th.wait();

pipe\_th.wait();

}

ConsoleSyncWriter::ConsoleSyncWriter(const std::wstring& str, const qint32 turn\_number)

: \_str{str}

, \_turn\_number{turn\_number} {

}

qint32 ConsoleSyncWriter::get\_terminal\_width() {

struct winsize w;

ioctl(STDOUT\_FILENO, TIOCGWINSZ, &w);

return w.ws\_col;

}

void ConsoleSyncWriter::write(QAtomicInt& current\_turn, qint32 next\_turn) {

QMutexLocker locker(&\_write\_mutex);

while (current\_turn.loadAcquire() != \_turn\_number) {

\_waiter.wait(&\_write\_mutex);

}

std::wcout << \_str << std::flush;

current\_turn.storeRelease(next\_turn);

\_waiter.wakeOne();

}

void ConsoleSyncWriter::write\_repeatly(QAtomicInt& current\_turn, qint32 next\_turn, qint32 repeat\_count) {

for (qint32 i{0}; i < repeat\_count; ++i) {

write(current\_turn, next\_turn);

}

qDebug() << \_\_PRETTY\_FUNCTION\_\_ << "finished";

}

} // Tasks

**Tasks/Task3.hpp**

#ifndef \_\_LAB3\_TASKS\_TASK1\_HPP\_\_

#define \_\_LAB3\_TASKS\_TASK1\_HPP\_\_

#include <QRunnable>

#include <QAtomicInt>

namespace Tasks {

class Task3 : public QRunnable {

public:

void run() override;

};

class Counter {

public:

Counter();

void increment();

void decrement();

qint32 get\_counter() const noexcept;

private:

QAtomicInt \_counter;

};

} // Tasks

#endif // \_\_LAB3\_TASKS\_TASK1\_HPP\_\_

**Tasks/Task3.cpp**

#include "Task3.hpp"

#include <QDebug>

#include <QMetaObject>

#include <QtConcurrent/QtConcurrent>

#include <QThread>

#include <thread>

namespace Tasks {

void Task3::run() {

qDebug() << \_\_PRETTY\_FUNCTION\_\_ << "called";

QThread increment\_th, decrement\_th;

Counter counter;

qint32 loop\_count{1000};

QObject::connect(&increment\_th, &QThread::started, [&counter, loop\_count, &increment\_th](){

for (qint32 i{0}; i < loop\_count; ++i) {

counter.increment();

increment\_th.quit();

}

});

QObject::connect(&decrement\_th, &QThread::started, [&counter, loop\_count, &decrement\_th](){

for (qint32 i{0}; i < loop\_count; ++i) {

counter.decrement();

decrement\_th.quit();

}

});

increment\_th.start();

decrement\_th.start();

increment\_th.wait();

decrement\_th.wait();

qInfo() << "Counter value = " << counter.get\_counter();

}

Counter::Counter()

: \_counter{0} {

}

void Counter::increment() {

++\_counter;

}

void Counter::decrement() {

--\_counter;

}

qint32 Counter::get\_counter() const noexcept {

return \_counter.loadAcquire();

}

} // Tasks

**Tasks/Task4.hpp**

#ifndef \_\_LAB4\_TASKS\_TASK1\_HPP\_\_

#define \_\_LAB4\_TASKS\_TASK1\_HPP\_\_

#include <QRunnable>

#include <QVector>

namespace Tasks {

class Task4 : public QRunnable {

public:

void run() override;

};

class Result {

public:

QVector<QVector<int>> \_matrix;

Result(int rows, int cols);

void set\_value(int row, int col, int value);

int get\_value(int row, int col) const;

};

} // Tasks

#endif // \_\_LAB4\_TASKS\_TASK1\_HPP\_\_

**Tasks/Task4.cpp**

#include "Task4.hpp"

#include <QDebug>

#include <QElapsedTimer>

#include <QVector>

#include <QThreadPool>

#include <QMutex>

#include <QMutexLocker>

namespace Tasks {

QMutex mutex; // Глобальный мьютекс для синхронизации доступа к матрице C

void Task4::run() {

qDebug() << \_\_PRETTY\_FUNCTION\_\_ << "called";

int size = 1000;

int num\_threads = 4;

QVector<QVector<int>> A(size, QVector<int>(size, 1)); // Матрица A

QVector<QVector<int>> B(size, QVector<int>(size, 2)); // Матрица B

Result C(size, size);

QElapsedTimer timer;

timer.start();

auto multiply\_block = [&A, &B, &C, size](int start\_row, int end\_row) {

qDebug() << "Thread started for rows" << start\_row << "to" << end\_row;

for (int i = start\_row; i < end\_row; ++i) {

for (int j = 0; j < size; ++j) {

int sum = 0;

for (int k = 0; k < size; ++k) {

sum += A[i][k] \* B[k][j];

}

// Использование мьютекса для защиты доступа к матрице C

QMutexLocker locker(&mutex);

C.set\_value(i, j, sum);

}

}

qDebug() << "Thread finished for rows" << start\_row << "to" << end\_row;

};

int rows\_per\_thread = size / num\_threads;

QThreadPool pool;

pool.setMaxThreadCount(num\_threads);

for (int t = 0; t < num\_threads; ++t) {

int start\_row = t \* rows\_per\_thread;

int end\_row = (t == num\_threads - 1) ? size : start\_row + rows\_per\_thread;

pool.start([&multiply\_block, start\_row, end\_row](){ multiply\_block(start\_row, end\_row); });

}

pool.waitForDone();

qint64 elapsed = timer.elapsed();

qDebug() << "Time elapsed with" << num\_threads << "threads:" << elapsed << "ms";

}

Result::Result(int rows, int cols) {

\_matrix.resize(rows);

for (auto& row : \_matrix) {

row.resize(cols, 0);

}

}

void Result::set\_value(int row, int col, int value) {

\_matrix[row][col] = value;

}

int Result::get\_value(int row, int col) const {

return \_matrix[row][col];

}

} // namespace Tasks

**Результат виконання:**

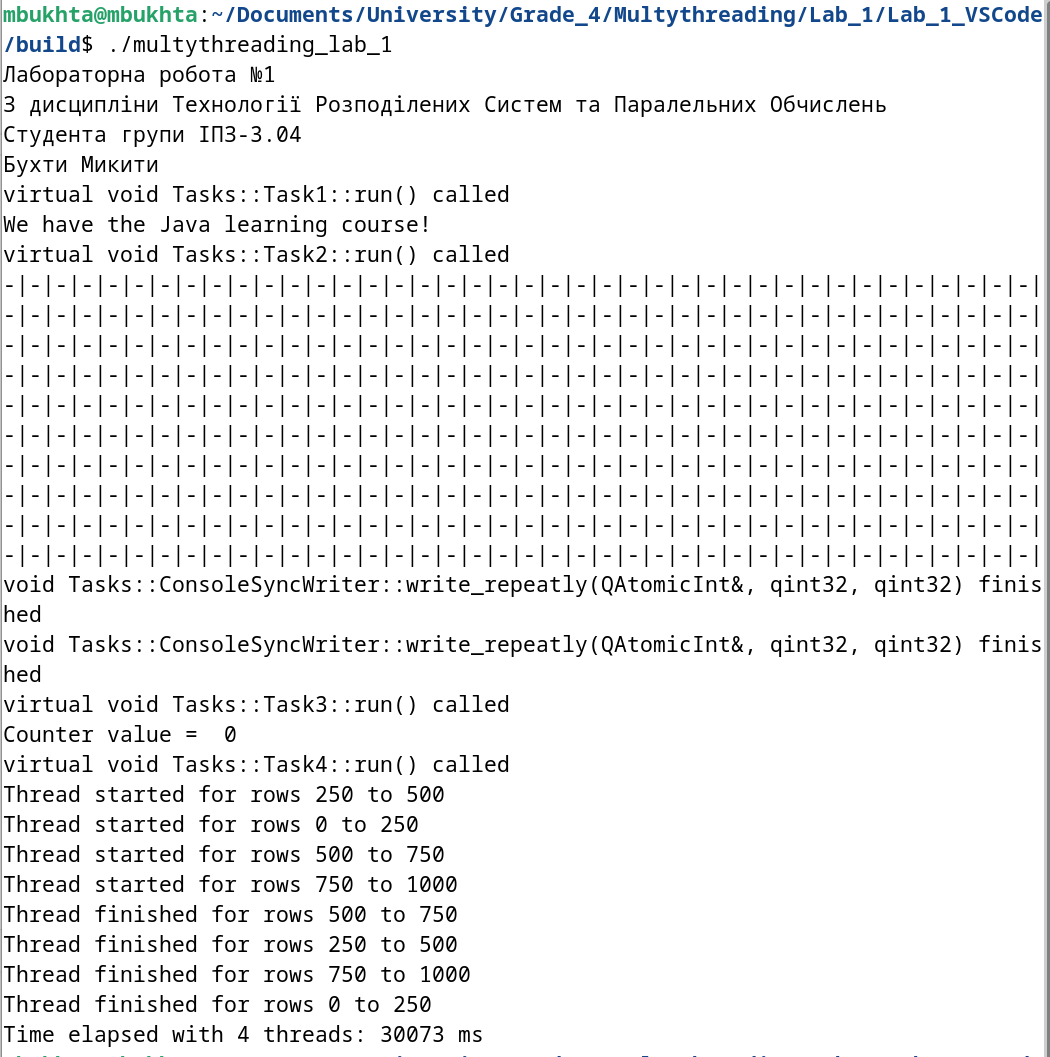


Рисунок 1 – результат виконання лаборатнорної роботи №1