МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ПРИКАРПАТСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ   
ІМЕНІ ВАСИЛЯ СТЕФАНИКА

Кафедра комп’ютерних наук та інформаційних систем

**Лабораторна робота №2**

з курсу «Паралельні та розподілені обчислення»

на тему:

«Застосування шаблонів стандартної бібліотеки на основі м’ютексів для ізолювання доступу до даних»

**Виконав:**

студент групи КН-41

Сенів П.В.

«\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 р.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис)

**Перевірив:**

викладач кафедри КНІС

к.т.н., Ізмайлов А. В. «\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_ 2021 р.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(оцінка, підпис)

м. Івано-Франківськ

2021

***Мета роботи*:** Навчитись застосовувати шаблони стандартної бібліотеки на  основі м’ютексів для ізолювання доступу потоків до розділюваних даних.

**Теоретичні відомості**

У С ++ для створення м’ютекса слід сконструювати об'єкт класу std::mutex,  для захоплення м’ютекса служить функція-член lock(), а для звільнення – функція-член unlock(). Однак, викликати ці функції безпосередньо не  рекомендується. Замість цього в стандартній бібліотеці є шаблон класу  std::lock\_guard, який захоплює м’ютекс у конструкторі і звільняє в деструкторі.  Цим гарантується, що захоплений м’ютекс обов’язково буде звільнений. Обидва  класи визначені у заголовку <mutex>.

Крім самого м’ютекса, конструктору std::lock\_guard можна передати  параметр std::adopt\_lock, який повідомляє об’єктові std::lock\_guard, що м’ютекс  уже захоплений, і йому потрібно лише прийняти володіння існуючим блокуванням, а не намагатися ще раз захопити м’ютекс у конструкторі.

У більшості випадків м’ютекс і захищені ним дані поміщають в один клас,  а не у глобальні змінні. Це є стандартним застосуванням правил об’єктно орієнтованого проектування. Якщо помістити обидві сутності у клас, то чітко  вказується їх взаємопов’язаність, а крім того, забезпечується інкапсулювання  функціональності і обмеження доступу. Тому м’ютекс і захищені ним дані  оголошують закритими полями деякого класу.

Застосування більше, ніж одного м’ютекса в одній ділянці коду породжує  не лише контроль над даними, але й ряд проблем. Нехай два потоки для  виконання деякої операції повинні захопити два м’ютекса, але склалося так, що  кожен потік захопив тільки один м’ютекс і чекає іншого. Жоден потік не може  продовжити роботу, тому що кожен чекає, поки інший звільнить потрібний йому  м’ютекс. Така ситуація називається взаємним блокуванням. Це найнебезпечніша

з проблем, які виникають, коли для виконання операції потрібно захопити більше  одного м’ютекса.

Одним із засобів захисту від взаємного блокування є функція std::lock стандартної бібліотеки, яка вміє захоплювати відразу два і більше м’ютексів без  ризику отримати взаємне блокування. При цьому слід пам’ятати, що std::lock  допомагає уникнути взаємного блокування у випадках, коли потрібно захопити  відразу два або більше м’ютексів, проте, вона не в силах допомогти, якщо  м’ютекси захоплюються порізно. У такому випадку залишається покладатися  тільки на дисципліну програмування

**Хід роботи:**

1.

**Код:**

//Task1 mutex

#include <iostream>

#include <thread>

#include <mutex>

#include <Windows.h>

#include <list>

#include <algorithm>

using namespace std;

list<int> listNumber;

mutex mutex1;

void AddToList(int numberAdd){

mutex1.lock();

listNumber.push\_back(numberAdd);

cout << numberAdd << " Added\n";

mutex1.unlock();

}

void ListContains(int numberAdd){

mutex1.lock();

auto exist = find(listNumber.begin(), listNumber.end(), numberAdd);

if (exist != listNumber.end()) {

cout << numberAdd << " True \n";

}

else

{

cout << numberAdd << " False \n";

}

mutex1.unlock();

}

int main() {

auto numberMain = 16;

for (int i = 0; i <= 10; i++)

{

thread t1(AddToList, numberMain);

thread t2(ListContains, 19);

numberMain++;

t1.detach();

t2.detach();

}

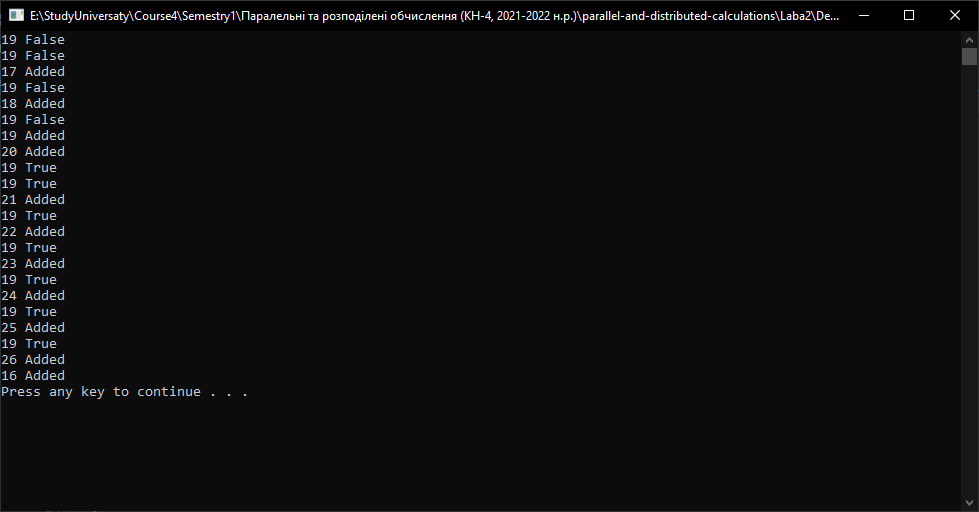
Sleep(1000);

system("pause");

return 0;

}

**Результати виконання роботи**:



**Код:**

//Task1 lock\_guard

#include <iostream>

#include <thread>

#include <mutex>

#include <Windows.h>

#include <list>

#include <algorithm>

using namespace std;

list<int> listNumber;

mutex mutex1;

void AddToList(int numberAdd){

lock\_guard<std::mutex> guard(mutex1);

listNumber.push\_back(numberAdd);

cout << numberAdd << " Added\n";

}

void ListContains(int numberAdd){

lock\_guard<std::mutex> guard(mutex1);

auto exist = find(listNumber.begin(), listNumber.end(), numberAdd);

if (exist != listNumber.end()) {

cout << numberAdd << " True \n";

}

else

{

cout << numberAdd << " False \n";

}

}

int main() {

auto numberMain = 16;

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

thread t1(AddToList, numberMain);

thread t2(ListContains, 19);

numberMain++;

t1.detach();

t2.detach();

}

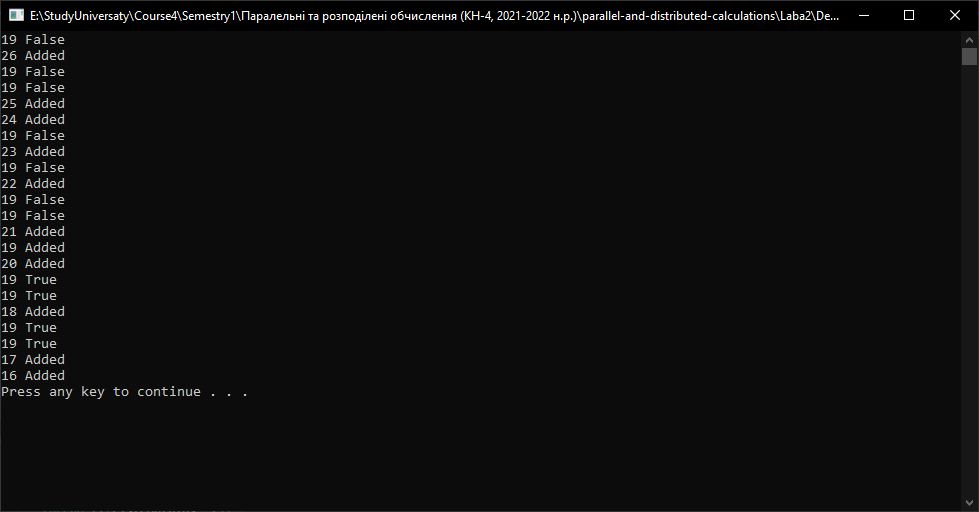
Sleep(1000);

system("pause");

return 0;

}

**Результати виконання роботи**:



При запуску декілька разів програм я отримував правильні результати і програми виконували те, що потрібно. Проте, якщо між викликами функцій lock() та unlock() буде непередбачувана ситуація, то функція unlock() просто не виконається. Через це краще застосовувати lock\_guard, який забезпечить звільнення мютекса будь-коли.

2)

**Код:**

#include <iostream>

#include <thread>

#include <mutex>

#include <Windows.h>

#include <list>

#include <algorithm>

using namespace std;

class someData {

public:

string firstNamePeople;

string lastNamePeople;

string addressPeople;

int agePeople;

};

class exchangePerson {

public:

someData someData;

mutex Mutex1;

static void JohnDoe(exchangePerson& people1) {

lock\_guard<mutex> guard(people1.Mutex1);

people1.someData.firstNamePeople = "John";

people1.someData.lastNamePeople = "Doe";

people1.someData.addressPeople = "Unknown";

people1.someData.agePeople = 120;

}

static void JacobSmith(exchangePerson& people2) {

lock\_guard<mutex> guard(people2.Mutex1);

people2.someData.firstNamePeople = "Jacob";

people2.someData.lastNamePeople = "Smith";

people2.someData.addressPeople = "Known";

people2.someData.agePeople = 1;

}

static void Swap(exchangePerson& people1, exchangePerson& people2) {

Show(people1, "Person 1 before swap");

Show(people2, "Person 2 before swap");

if (addressof(people1) != addressof(people2))

{

lock(people1.Mutex1, people2.Mutex1);

lock\_guard<mutex> lock1(people1.Mutex1, adopt\_lock);

lock\_guard<mutex> lock2(people2.Mutex1, adopt\_lock);

exchangePerson newPerson;

newPerson.someData.firstNamePeople = people1.someData.firstNamePeople;

newPerson.someData.lastNamePeople = people1.someData.lastNamePeople;

newPerson.someData.addressPeople = people1.someData.addressPeople;

newPerson.someData.agePeople = people1.someData.agePeople;

people1.someData.firstNamePeople = people2.someData.firstNamePeople;

people1.someData.lastNamePeople = people2.someData.lastNamePeople;

people1.someData.addressPeople = people2.someData.addressPeople;

people1.someData.agePeople = people2.someData.agePeople;

people2.someData.firstNamePeople = newPerson.someData.firstNamePeople;

people2.someData.lastNamePeople = newPerson.someData.lastNamePeople;

people2.someData.addressPeople = newPerson.someData.addressPeople;

people2.someData.agePeople = newPerson.someData.agePeople;

Show(people1, "Person 1 after swap");

Show(people2, "Person 2 after swap");

}

}

static void Show(exchangePerson& person, string text) {

cout << text << " -> firstName - " << person.someData.firstNamePeople << ", lastName - " << person.someData.lastNamePeople << ", age - " << person.someData.agePeople << ", address - " << person.someData.addressPeople << "\n";

}

};

int main() {

exchangePerson people1;

exchangePerson people2;

thread t1(exchangePerson::JohnDoe, ref(people1));

t1.detach();

thread t2(exchangePerson::JacobSmith, ref(people1));

t2.detach();

thread t3(exchangePerson::JacobSmith, ref(people2));

t3.detach();

thread t4(exchangePerson::JohnDoe, ref(people2));

t4.detach();

thread t5(exchangePerson::Swap, ref(people1), ref(people2));

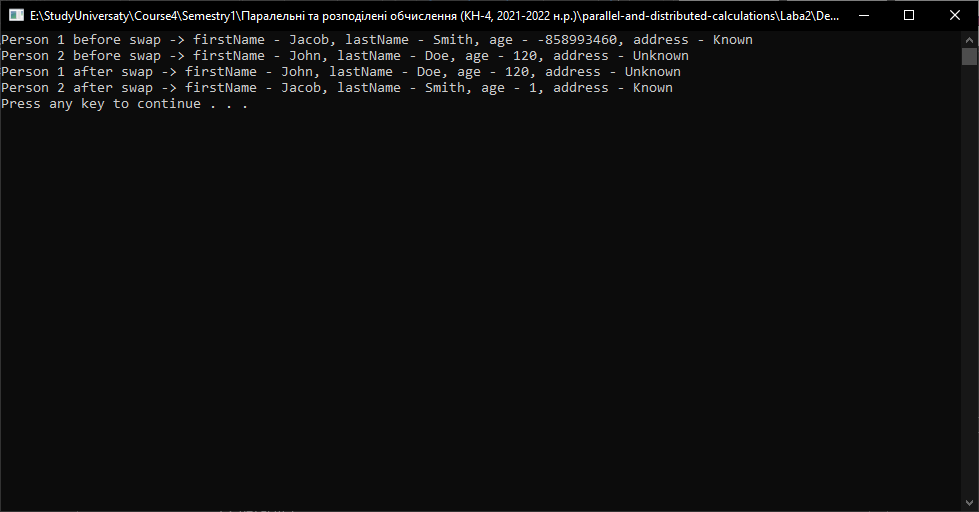
t5.join();

system("pause");

return 0;

}}

**Результати виконання роботи**:



3)

**Код:**

#include <iostream>

#include <thread>

#include <mutex>

#include <Windows.h>

#include <list>

#include <algorithm>

using namespace std;

class someData {

public:

string firstNamePeople;

string lastNamePeople;

string addressPeople;

int agePeople;

};

class exchangePerson {

public:

someData someData;

mutex Mutex1;

static void JohnDoe(exchangePerson& people1) {

lock\_guard<mutex> guard(people1.Mutex1);

people1.someData.firstNamePeople = "John";

people1.someData.lastNamePeople = "Doe";

people1.someData.addressPeople = "Unknown";

people1.someData.agePeople = 120;

}

static void JacobSmith(exchangePerson& people2) {

lock\_guard<mutex> guard(people2.Mutex1);

people2.someData.firstNamePeople = "Jacob";

people2.someData.lastNamePeople = "Smith";

people2.someData.addressPeople = "Known";

people2.someData.agePeople = 1;

}

static void Swap(exchangePerson& people1, exchangePerson& people2) {

Show(people1, "Person 1 before swap");

Show(people2, "Person 2 before swap");

if (addressof(people1) != addressof(people2))

{

unique\_lock<mutex> lock1(people1.Mutex1, defer\_lock);

unique\_lock<mutex> lock2(people2.Mutex1, defer\_lock);

lock(lock1, lock2);

exchangePerson newPerson;

newPerson.someData.firstNamePeople = people1.someData.firstNamePeople;

newPerson.someData.lastNamePeople = people1.someData.lastNamePeople;

newPerson.someData.addressPeople = people1.someData.addressPeople;

newPerson.someData.agePeople = people1.someData.agePeople;

people1.someData.firstNamePeople = people2.someData.firstNamePeople;

people1.someData.lastNamePeople = people2.someData.lastNamePeople;

people1.someData.addressPeople = people2.someData.addressPeople;

people1.someData.agePeople = people2.someData.agePeople;

people2.someData.firstNamePeople = newPerson.someData.firstNamePeople;

people2.someData.lastNamePeople = newPerson.someData.lastNamePeople;

people2.someData.addressPeople = newPerson.someData.addressPeople;

people2.someData.agePeople = newPerson.someData.agePeople;

Show(people1, "Person 1 after swap");

Show(people2, "Person 2 after swap");

}

}

static void Show(exchangePerson& person, string text) {

cout << text << " -> firstName - " << person.someData.firstNamePeople << ", lastName - " << person.someData.lastNamePeople << ", age - " << person.someData.agePeople << ", address - " << person.someData.addressPeople << "\n";

}

};

int main() {

exchangePerson people1;

exchangePerson people2;

thread t1(exchangePerson::JohnDoe, ref(people1));

t1.detach();

thread t2(exchangePerson::JacobSmith, ref(people1));

t2.detach();

thread t3(exchangePerson::JacobSmith, ref(people2));

t3.detach();

thread t4(exchangePerson::JohnDoe, ref(people2));

t4.detach();

thread t5(exchangePerson::Swap, ref(people1), ref(people2));

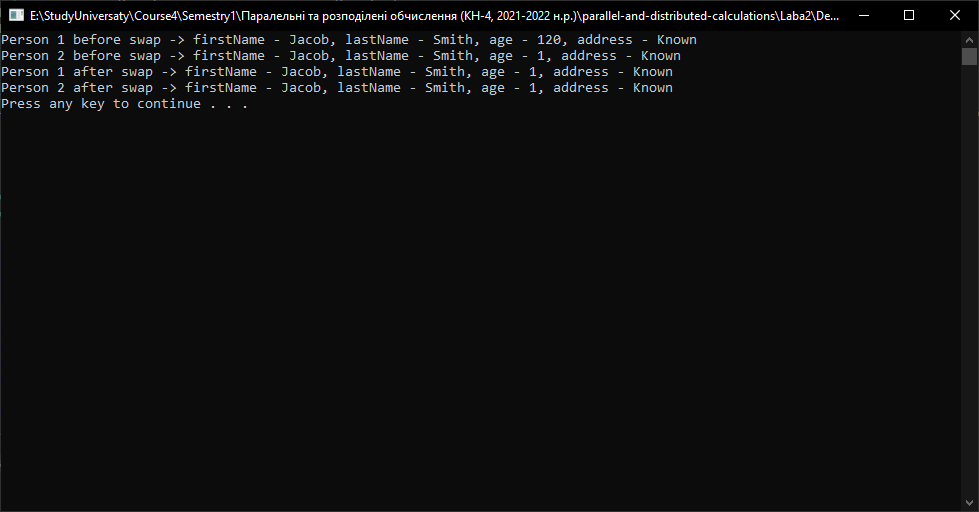
t5.join();

system("pause");

return 0;

}

**Результати виконання роботи:**



Різниця між lock\_guard і unique\_lock в тому, що можна як заблокувати так і розблокувати std :: unique\_lock. Тоді, як std :: lock\_guard буде заблокований тільки один раз при білді і розблоковано при знищенні.

Шаблон lock\_guard використовують, коли об'єкт створений, тоді він намагається отримати м'ютекс. Шаблон unique\_lock підтримує відкладене блокування, тимчасове заблокування, рекурсивне блокування і використання умовних змінних.

**Висновок:** на лабораторній роботі я навчився застосовувати шаблони стандартної бібліотеки на  основі м’ютексів для ізолювання доступу потоків до розділюваних даних. Мої вихідні файли з проектом знаходяться за адресою: <https://github.com/PavloSeniv/parallel-and-distributed-calculations/tree/main/Laba2>