**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ НАЦІОНАЛЬНОМУ УНІВЕРСИТЕТІ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**

Кафедра систем штучного інтелекту



**Лабораторна робота №4.1-4.2**

з дисципліни

«Операційні системи та системне програмування»

**Виконав:**

студент групи ШІ-24

Лещишин М. В.

**Викладач**:

Петренко Д. О.

Кривенчук Ю.П.

Львів – 2023 р.

**Варіант:** 29

**Тема.** Робота з динамічними бібліотеками в ОС Windows

**Мета.** Ознайомитися з відображенням файлів в оперативну пам’ять в ОС Windows. Навчитися реалізовувати відображення файлів в оперативну пам’ять..

**Завдання**

****

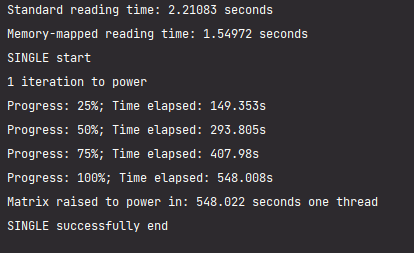
**Хід роботи:**

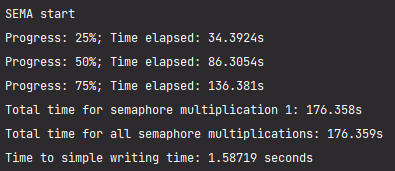
1. **Ознайомився із синхронізаційними механізмами:**
   1. Interlocked-функції
   2. Критична секція (Critical section)
   3. Спін-блокування (Spinlock)
   4. Wait-Функції
   5. М’ютекс (Mutex)
   6. Семафор (Semaphpre)
2. **Для алгоритму із лабораторної роботи №3 реалізував завдання обраної складності із нижченаведеного списку.**

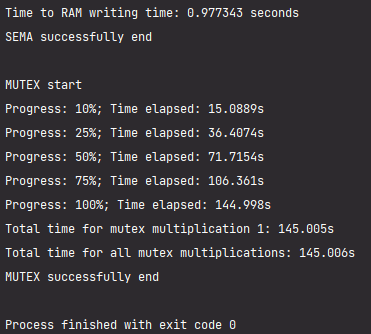
Я обрав Atomic та Mutex. На матриці 700 на 700 Mutex працює швидше бо

Це взаємовиключення, яке гарантує, що лише один потік може виконувати певний блок коду в один час. Це може бути корисним для захисту ресурсів або областей коду, які не можуть виконуватися паралельно. Mutex може вносити додаткові витрати на встановлення та зняття блокувань.

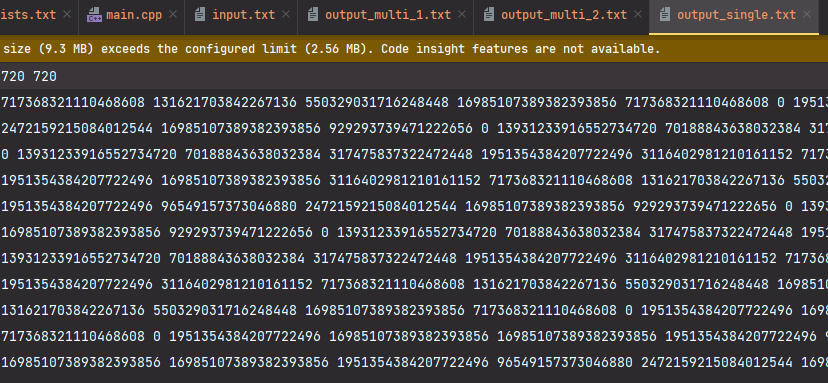
1. **Підготував файли з тестовими даними, а саме матрицю 700 на 700, тому що з меншою матрицею ніякої різниці не видно. Ось результат виконання.**







1. **Ось частина файлу у який виводиться результат.**



**Висновок**

Синхронізація потоків є важливим аспектом розробки програм для операційних систем Windows. Вона дозволяє уникнути конфліктів та забезпечити правильну роботу багатопоточних програм, де кілька потоків спільно працюють з ресурсами або даними. У ході вивчення способів синхронізації потоків та використання функцій WinAPI ми зрозуміли, що існує кілька методів досягнення цієї мети:

Додаток А:

Код програми, яка виеонує поставлене завдання:

#include <iostream>  
#include <vector>  
#include <fstream>  
#include <string>  
#include <sstream>  
#include <chrono>  
#include <thread>  
#include <mutex>  
#include <condition\_variable>  
  
*using Matrix* = std::vector<std::vector<*unsigned long long*>>;  
  
*class* Semaphore {  
*public*:  
 *explicit* Semaphore(*int count* = 1) : count\_(*count*) {}  
  
 *void* acquire()  
 {  
 std::unique\_lock<std::mutex> lock(mutex\_);  
 *while*(count\_ == 0) {  
 cond\_.wait(lock);  
 }  
 --count\_;  
 }  
  
 *void* release()  
 {  
 std::unique\_lock<std::mutex> lock(mutex\_);  
 ++count\_;  
 cond\_.notify\_one();  
 }  
  
*private*:  
 std::mutex mutex\_;  
 std::condition\_variable cond\_;  
 *int* count\_;  
};  
  
  
#ifdef **\_WIN32**#include <windows.h>  
  
*HANDLE* OpeNFileMapping(*const* std::*string*& *filename*, *HANDLE*& *hMapFile*)  
 {  
 *HANDLE* hFile = **CreateFile**( *//Унікальний дескриптор відкритого файлу разом із встановленням різних доступів до нього щоб вподальшому його використовуваи для ствоерння відображення  
 filename*.c\_str(), *// Ім'я файлу* **GENERIC\_READ**, *// Режим доступу: GENERIC\_READ для читання* **FILE\_SHARE\_READ**, *// Режим спільного використання: FILE\_SHARE\_READ дозволяє іншим процесам читати файл* **NULL**, *// Атрибути безпеки: NULL означає використання атрибутів безпеки за замовчуванням* **OPEN\_EXISTING**, *// Поведінка при створенні: OPEN\_EXISTING відкриває файл, якщо він існує* **FILE\_ATTRIBUTE\_NORMAL**, *// Атрибути файлу: FILE\_ATTRIBUTE\_NORMAL використовується для звичайного доступу до файлу* **NULL** *// Шаблон файлу: NULL означає, що шаблон не використовується* );  
 *// Якщо файл не вдається відкрити (наприклад, якщо він не існує),  
 // функція повертає INVALID\_HANDLE\_VALUE і виводить повідомлення про помилку.  
 if* (hFile == **INVALID\_HANDLE\_VALUE**)  
 {  
 std::cerr **<<** "Не вдалося відкрити файл" **<<** std::endl;  
 *return* **INVALID\_HANDLE\_VALUE**;  
 }  
  
 *//Створення дескриптора який потім використається для відображення файлу у пам'яті, на основі дескриптора створеного файлц  
 hMapFile* = **CreateFileMapping**(  
 hFile, *// Дескриптор файлу: hFile - це дескриптор файлу, отриманий з CreateFile.Який саме файл буд відображено у памяті* **NULL**, *// Атрибути безпеки: NULL означає використання атрибутів безпеки за замовчуванням* **PAGE\_READONLY**, *// Захист сторінки: PAGE\_READONLY дозволяє доступ тільки для читання* 0, *// Максимальний розмір об'єкта високого порядку: 0 означає, що система визначає розмір файлу* 0, *// Максимальний розмір об'єкта низького порядку* **NULL** *// Ім'я відображення файлу: NULL означає, що відображення не буде мати імені* );  
 *//Якщо створення відображення не вдається, функція виводить повідомлення про помилку,  
 // закриває дескриптор файлу hFile і повертає INVALID\_HANDLE\_VALUE.  
 if* (*hMapFile* == **NULL**)  
 {  
 std::cerr **<<** "Не вдалося створити відображення файлу" **<<** std::endl;  
 CloseHandle(hFile);  
 *return* **INVALID\_HANDLE\_VALUE**;  
 }  
 *return* hFile; *//Дескриптор файлу*}  
  
*LPVOID* MapFile(*HANDLE hMapFile*)  
{  
 *LPVOID* lpMapAddress = MapViewOfFile( *//Створення відображення в адресному просторі  
 hMapFile*, *// Дескриптор відображення файлу: hMapFile - це дескриптор, отриманий з CreateFileMapping* **FILE\_MAP\_READ**, *// Режим доступу до відображення: FILE\_MAP\_READ дозволяє читання з відображення* 0, *// Високий порядок зсуву файлу: 0 означає, що зсув починається з початку файлу* 0, *// Низький порядок зсуву файлу* 0 *// Кількість байтів для відображення: 0 означає відображення всього файлу* );  
 *if* (lpMapAddress == **NULL**)  
 {  
 std::cerr **<<** "Не вдалося відобразити файл у пам'яті" **<<** std::endl;  
 }  
 *return* lpMapAddress;*// вказівник (LPVOID lpMapAddress) на початок відображення файлу в пам'яті. Цей вказівник використовується для доступу до даних файлу.*}  
  
*bool* writeMatrixToFileMapping(*const Matrix*& *matrix*, *const* std::*string*& *filename*)  
{  
 *// Створюємо строковий потік для зберігання даних матриці у текстовому форматі* std::*stringstream* ss;  
 *// Записуємо розміри матриці у потік* ss **<<** *matrix*.size() **<<** " " **<<** *matrix***[**0**]**.size() **<<** "\n";  
 *// Перебираємо елементи матриці та записуємо їх у потік  
 for* (*const auto* &row: *matrix*)  
 {  
 *for* (*const auto* &elem: row)  
 {  
 ss **<<** elem **<<** " ";  
 }  
 ss **<<** "\n";  
 }  
 *// Конвертуємо потік у стрічку* std::*string* strData = ss.str();  
 *// Визначаємо розмір даних для запису в байтах  
 size\_t* dataSize = strData.size();  
  
 *// Створюємо,відкриваємо файл для запису  
 HANDLE* hFile = **CreateFile**(  
 *filename*.c\_str(), *// Шлях до файлу* **GENERIC\_READ** | **GENERIC\_WRITE**, *// Режим доступу: читання та запис* 0, *// Без спільного доступу* **NULL**, *// Без додаткових атрибутів безпеки* **CREATE\_ALWAYS**, *// Створити новий файл або перезаписати існуючий* **FILE\_ATTRIBUTE\_NORMAL**, *// Звичайні атрибути файлу* **NULL** *// Без шаблону* );  
  
 *// Перевірка на успішність створення файлу  
 if* (hFile == **INVALID\_HANDLE\_VALUE**)  
 {  
 std::cerr **<<** "Не вдалося створити файл" **<<** std::endl;  
 *return false*;  
 }  
  
 *// Створюємо відображення файлу у адресну пам'ять процесу  
 HANDLE* hMapFile = **CreateFileMapping**(  
 hFile, *// Дескриптор файлу* **NULL**, *// Без додаткових атрибутів безпеки* **PAGE\_READWRITE**, *// Доступ до відображення: читання та запис* 0, *// Розмір високого порядку (для файлів до 4 ГБ не потрібен)  
 static\_cast*<*DWORD*>(dataSize), *// Розмір низького порядку* **NULL** *// Без імені відображення* );  
  
 *// Перевірка на успішність створення відображення файлу  
 if* (hMapFile == **NULL**)  
 {  
 std::cerr **<<** "Не вдалося створити відображення файлу" **<<** std::endl;  
 CloseHandle(hFile);  
 *return false*;  
 }  
  
 *// Відображаємо вказівник на адресний просітр процесу  
 LPVOID* lpMapAddress = MapViewOfFile(  
 hMapFile, *// Дескриптор відображення файлу* **FILE\_MAP\_WRITE**, *// Режим доступу: запис* 0, *// Високий порядок зсуву файлу* 0, *// Низький порядок зсуву файлу* dataSize *// Кількість байтів для відображення* );  
  
 *// Перевірка на успішність відображення файлу у пам'ять  
 if* (lpMapAddress == **NULL**)  
 {  
 std::cerr **<<** "Не вдалося відобразити файл у пам'яті" **<<** std::endl;  
 CloseHandle(hMapFile);  
 CloseHandle(hFile);  
 *return false*;  
 }  
  
 *// Копіюємо дані у відображену область пам'яті* **CopyMemory**(lpMapAddress, strData.c\_str(), dataSize);  
  
 *// Закриваємо відображення і дескриптори* UnmapViewOfFile(lpMapAddress);  
 CloseHandle(hMapFile);  
 CloseHandle(hFile);  
  
 *return true*;  
}  
  
#else  
*// Інклюдимо необхідні заголовочні файли для роботи з файлами та пам'яттю в Linux*#include <fcntl.h> *// Для open, close*#include <unistd.h> *// Для read, write, lseek*#include <sys/mman.h> *// Для mmap, munmap*#include <sys/stat.h> *// Для fstat*#include <iostream>  
#include <sstream>  
#include <cstring> *// Для strerror  
  
// Функція для відкриття файлу  
int* openFile(*const* std::string& filename)  
{  
 *// Відкриття файлу тільки для читання  
 int* fd = open(filename.c\_str(), O\_RDONLY);  
 *if* (fd == -1)  
 {  
 std::cerr << "Не вдалося відкрити файл: " << strerror(errno) << std::endl;  
 }  
 *return* fd; *// Повертаємо дескриптор файлу*}  
  
*// Функція для відображення файлу у пам'ять  
void*\* mapFile(*int* fd, size\_t& length)  
{  
 *struct* stat sb; *//інформація про файл зберігається тут  
  
 // Отримуємо інформацію про файл за його дескриптором, включно з його розміром.  
 //Ця інформація записується у структуру struct stat, на яку вказує &sb  
 if* (fstat(fd, &sb) == -1)  
 {  
 std::cerr << "Помилка при отриманні розміру файлу: " << strerror(errno) << std::endl;  
 *return nullptr*;  
 }  
 length = sb.st\_size; *// Зберігаємо розмір файлу в байтах, щоб знати скільки памяті виділити.  
  
 // Відображаємо файл у пам'ять  
 void*\* addr = mmap(*nullptr*, length, PROT\_READ, MAP\_PRIVATE, fd, 0);  
 *//nullptr - я прошу операційну систему автоматично вибрати відповідну адресу віртуальної пам'яті для відображення файлу  
 //0 - зсув.Тобто відображення почнеться з початку файлу.  
 if* (addr == MAP\_FAILED)  
 {  
 std::cerr << "Не вдалося відобразити файл у пам'ять: " << strerror(errno) << std::endl;  
 }  
 *return* addr; *// Повертаємо вказівник на відображену область пам'яті*}  
  
*// Функція для запису матриці у файл  
bool* writeMatrixToFile(*const* Matrix& matrix, *const* std::string& filename)  
{  
 std::stringstream ss;  
 *// Конвертуємо матрицю у строку для запису у файл  
 // Код для запису матриці у ss, як у вашому вихідному прикладі* std::string strData = ss.str();  
 size\_t dataSize = strData.size();  
  
 *// Відкриваємо файл для запису, створюємо його, якщо він не існує  
 int* fd = open(filename.c\_str(), O\_RDWR | O\_CREAT | O\_TRUNC, S\_IRUSR | S\_IWUSR);  
 *if* (fd == -1)  
 {  
 std::cerr << "Не вдалося створити або відкрити файл: " << strerror(errno) << std::endl;  
 *return false*;  
 }  
  
 *// Змінюємо розмір файлу під розмір наших даних  
 if* (lseek(fd, dataSize - 1, SEEK\_SET) == -1) *//використовується для забезпечення того, що файл має достатній розмір для подальшого відображення його вмісту в пам'ять (mmap). Це дозволяє mmap працювати з файлом очікуваного розміру, запобігаючи помилкам читання або запису за межами дійсного вмісту файлу.* {  
 close(fd);  
 std::cerr << "Помилка під час переміщення покажчика у файлі: " << strerror(errno) << std::endl;  
 *return false*;  
 }  
  
 *// Записуємо один байт в кінець файлу, щоб забезпечити потрібний розмір  
 if* (write(fd, "", 1) != 1)  
 {  
 close(fd);  
 std::cerr << "Помилка під час запису у файл: " << strerror(errno) << std::endl;  
 *return false*;  
 }  
  
 *// Відображаємо файл у пам'ять для запису даних  
 void*\* addr = mmap(*nullptr*, dataSize, PROT\_READ | PROT\_WRITE, MAP\_SHARED, fd, 0);  
 *if* (addr == MAP\_FAILED)  
 {  
 close(fd);  
 std::cerr << "Не вдалося відобразити файл у пам'ять: " << strerror(errno) << std::endl;  
 *return false*;  
 }  
  
 *// Копіюємо дані у відображену область пам'яті* memcpy(addr, strData.c\_str(), dataSize);  
  
 *// Закриваємо відображення та файл* munmap(addr, dataSize);  
 close(fd);  
  
 *return true*;  
}  
  
#endif  
  
  
*void* ReadMatrix(*const* std::*string*& *data*, *int*& *rows*, *int*& *cols*, std::vector<std::vector<*unsigned long long*>>& *matrix*) {  
 std::*istringstream* iss(*data*);  
 iss **>>** *rows* **>>** *cols*;  
  
 *matrix*.resize(*rows*, std::vector<*unsigned long long*>(*cols*));  
  
 *for* (*int* i = 0; i < *rows*; ++i)  
 {  
 *for* (*int* j = 0; j < *cols*; ++j)  
 {  
 iss **>>** *matrix***[**i**][**j**]**;  
 }  
 }  
}  
  
  
*/////////////////////  
  
// Функція для зчитування матриці з файлу  
bool* readMatrixFromFile(*const* std::*string*& *filename*, *Matrix*& *matrix*)  
{  
 std::*ifstream* inputFile(*filename*);  
 *if* (!inputFile.is\_open()) {  
 std::cerr **<<** "Unable to open the file: " **<<** *filename* **<<** std::endl;  
 *return false*;  
 }  
  
 *int* rows, cols;  
 inputFile **>>** rows **>>** cols;  
 *if* (inputFile.fail()) {  
 std::cerr **<<** "Failed to read the size of the matrix." **<<** std::endl;  
 *return false*;  
 }  
  
 *matrix*.resize(rows, std::vector<*unsigned long long*>(cols));  
  
 *for* (*int* i = 0; i < rows; ++i) {  
 *for* (*int* j = 0; j < cols; ++j) {  
 inputFile **>>** *matrix***[**i**][**j**]**;  
 *if* (inputFile.fail()) {  
 std::cerr **<<** "Failed to read the matrix elements." **<<** std::endl;  
 *return false*;  
 }  
 }  
 }  
  
 inputFile.close();  
 *return true*;  
}  
  
*// Функція для запису матриці у файл  
void* writeMatrixToFile(*const Matrix*& *matrix*, *const* std::*string*& *filename*)  
{  
 std::*ofstream* file(*filename*);  
 *for* (*const auto*& row : *matrix*)  
 {  
 *for* (*const auto*& elem : row)  
 {  
 file **<<** elem **<<** " ";  
 }  
 file **<<** std::endl;  
 }  
 file.close();  
}  
  
  
*// Функція для запису матриці у файл з використанням відображення файлів у пам'яті  
  
//Для відстеження прогресу однопоточного піднесення  
void* operationTrace(*long double totalOperations*, *long double size*, *auto startTime*,  
 std::vector<*double*>& *tracePoints*)  
{  
  
 *long double* progress = *totalOperations* / (*size* \* *size*);  
  
 *for* (*auto*& point : *tracePoints*) {  
 *if* (progress == point)  
 {  
 *auto* currentTime = std::chrono::system\_clock::now();  
 std::chrono::duration<*double*> elapsedSeconds = currentTime - *startTime*;  
 std::cout **<<** "Progress: " **<<** *static\_cast*<*int*>(point \* 100) **<<** "%; Time elapsed: " **<<** elapsedSeconds.count() **<<** "s\n";  
  
 }  
 }  
}  
  
*//Множення матриць  
Matrix* multiplyMatrices(*const Matrix*& *m1*, *const Matrix*& *m2*, *auto startTime*,  
 std::vector<*double*>& *tracePoints*)  
{  
 *long double* totalOperations = 0;  
 *int* rows = *m1*.size();  
 *int* cols = *m2***[**0**]**.size();  
 *int* temp = *m2*.size();  
 *Matrix* result(rows, std::vector<*unsigned long long*>(cols, 0));  
  
 *for* (*int* i = 0; i < rows; ++i) {  
 *for* (*int* j = 0; j < cols; ++j) {  
 *for* (*int* k = 0; k < temp; ++k) {  
 result**[**i**][**j**]** += *m1***[**i**][**k**]** \* *m2***[**k**][**j**]**;  
 }  
 totalOperations++;  
 operationTrace(totalOperations, rows, *startTime*, *tracePoints*);  
 }  
 }  
 *return* result;  
}  
  
  
*Matrix* matrixPowerOneThread(*const Matrix*& *matrix*, *int power*, *auto startTime*,  
 std::vector<*double*>& *tracePoints*)  
{  
 *Matrix* result = *matrix*; *// Початкове значення для степеня 1  
 if* (*power* == 0)  
 {  
 *// Повертаємо одиничну матрицю, якщо степінь дорівнює нулю  
 for* (*auto* &row : result)  
 {  
 std::fill(row.begin(), row.end(), 0);  
 }  
 *for* (*size\_t* i = 0; i < *matrix*.size(); ++i)  
 {  
 result**[**i**][**i**]** = 1;  
 }  
 *return* result;  
 }  
  
 *Matrix* temp = *matrix*; *// Тимчасова матриця для зберігання проміжних результатів  
 for* (*int* p = 1; p < *power*; ++p)  
 {  
 std::cout **<<** p **<<** " iteration to power" **<<** std::endl;  
 result = multiplyMatrices(result, temp, *startTime*, *tracePoints*); *// Множимо результат на початкову матрицю* }  
 *return* result;  
}  
  
*Matrix* measureOneThread(*const Matrix*& *matrix*, *int power*)  
{  
 std::vector<*double*> tracePoints = **{**0.1, 0.25, 0.5, 0.75, 1.0**}**;  
 *auto* start = std::chrono::*high\_resolution\_clock*::now(); *// Початкова мітка часу  
  
 Matrix* result = matrixPowerOneThread(*matrix*, *power*, start, tracePoints); *// Підносимо матрицю до степеня  
  
 auto* finish = std::chrono::*high\_resolution\_clock*::now(); *// Кінцева мітка часу* std::chrono::duration<*double*> elapsed = finish **-** start;  
  
 std::cout **<<** "Matrix raised to power in: " **<<** elapsed.count() **<<** " seconds one thread" **<<** std::endl;  
  
 *return* result;  
}  
  
*///////  
//Для відстеження прогресу піднесення використовуючи семафор  
void* operationTrace(*long double totalOperations*, *long double size*, *auto startTime*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *progressMutex*)  
{  
 std::lock\_guard<std::mutex> lock(*progressMutex*);  
 *long double* progress = *totalOperations* / (*size* \* *size*);  
  
 *for* (*auto*& point : *tracePoints*)  
 {  
 *if* (progress == point)  
 {  
 *auto* currentTime = std::chrono::system\_clock::now();  
 std::chrono::duration<*double*> elapsedSeconds = currentTime - *startTime*;  
 std::cout **<<** "Progress: " **<<** *static\_cast*<*int*>(point \* 100) **<<** "%; Time elapsed: " **<<** elapsedSeconds.count() **<<** "s\n";  
 *tracePoints*.erase(std::remove(*tracePoints*.begin(), *tracePoints*.end(), point), *tracePoints*.end());  
 *break*;  
 }  
 }  
}  
  
  
*void* worker(*const Matrix*& *m1*, *const Matrix*& *m2*, *Matrix*& *result*,  
 *int startRow*, *int endRow*, Semaphore& *sem*,  
 *long double*& *totalOperations*, *long double size*,  
 std::chrono::time\_point<std::chrono::*high\_resolution\_clock*> *startTime*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *progressMutex*)  
{  
 *for* (*int* i = *startRow*; i < *endRow*; ++i) {  
 *for* (*int* j = 0; j < *m2***[**0**]**.size(); ++j) {  
 *unsigned long long* sum = 0;  
 *for* (*int* k = 0; k < *m2*.size(); ++k) {  
 sum += *m1***[**i**][**k**]** \* *m2***[**k**][**j**]**;  
 }  
 *sem*.acquire(); *// Захоплюємо семафор  
 result***[**i**][**j**]** = sum;  
 *totalOperations*++;  
 operationTrace(*totalOperations*, *size*, *startTime*, *tracePoints*, *progressMutex*);  
 *sem*.release(); *// Звільняємо семафор* }  
 }  
}  
  
  
*Matrix* multiplyMatricesSemaphore(*const Matrix*& *m1*, *const Matrix*& *m2*, Semaphore& *sem*,  
 *long double*& *totalOperations*, *long double size*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *progressMutex*)  
{  
 *int* rows = *m1*.size();  
 *int* cols = *m2***[**0**]**.size();  
 *Matrix* result(rows, std::vector<*unsigned long long*>(cols, 0));  
 std::vector<std::thread> threads;  
 *int* rowsPerThread = rows / 4;  
  
 *auto* startTime = std::chrono::*high\_resolution\_clock*::now();  
  
 *for* (*int* i = 0; i < 4; ++i) {  
 *int* startRow = i \* rowsPerThread;  
 *int* endRow = (i == 3) ? rows : (i + 1) \* rowsPerThread;  
 threads.emplace\_back(worker, std::cref(*m1*), std::cref(*m2*),  
 std::ref(result), startRow, endRow,  
 std::ref(*sem*), std::ref(*totalOperations*), *size*,  
 startTime, std::ref(*tracePoints*), std::ref(*progressMutex*));  
 }  
  
 *for* (*auto*& thread : threads) {  
 *if* (thread.joinable()) {  
 thread.join();  
 }  
 }  
  
 *return* result;  
}  
  
  
*Matrix* matrixPowerSemaphore(*const Matrix*& *matrix*, *int power*, Semaphore& *sem*,  
 *long double size*, *const* std::vector<*double*>& *originalTracePoints*,  
 std::mutex& *progressMutex*)  
{  
 *Matrix* result = *matrix*;  
 *auto* totalStartTime = std::chrono::*high\_resolution\_clock*::now(); *// Загальний час старту  
  
 for* (*int* p = 1; p < *power*; ++p) {  
 *auto* startTime = std::chrono::*high\_resolution\_clock*::now(); *// Час старту для цього множення  
 long double* totalOperations = 0;  
 std::vector<*double*> tracePoints = *originalTracePoints*; *// Копія для кожного множення* result **=** multiplyMatricesSemaphore(result, *matrix*, *sem*,  
 totalOperations, *size*, tracePoints, *progressMutex*);  
  
 *auto* endTime = std::chrono::*high\_resolution\_clock*::now(); *// Час завершення для цього множення* std::chrono::duration<*double*> elapsed = endTime **-** startTime;  
 std::cout **<<** "Total time for semaphore multiplication " **<<** p **<<** ": " **<<** elapsed.count() **<<** "s\n";  
 }  
  
 *auto* totalEndTime = std::chrono::*high\_resolution\_clock*::now(); *// Загальний час завершення* std::chrono::duration<*double*> totalElapsed = totalEndTime **-** totalStartTime;  
 std::cout **<<** "Total time for all semaphore multiplications: " **<<** totalElapsed.count() **<<** "s\n";  
  
 *return* result;  
}  
  
  
*Matrix* measureSemaphore(*const Matrix*& *matrix*, *int power*)  
{  
 Semaphore sem(4);  
 *Matrix* result;  
  
 *long double* size = *matrix*.size();  
 std::vector<*double*> tracePoints = **{**0.1, 0.25, 0.5, 0.75, 1.0**}**;  
 std::mutex progressMutex;  
  
 *if* (*power* == 0) {  
 *// Create an identity matrix* result **=** Matrix(*matrix*.size(), std::vector<*unsigned long long*>(*matrix*.size(), 0));  
 *for* (*size\_t* i = 0; i < *matrix*.size(); ++i) {  
 result**[**i**][**i**]** = 1;  
 }  
 } *else* {  
 result **=** matrixPowerSemaphore(*matrix*, *power*, sem, size, tracePoints, progressMutex);  
 }  
  
 *return* result;  
}  
  
*/////////////////////  
  
void* operationTraceMutex(*long double*& *totalOperations*, *long double size*, *const* std::chrono::time\_point<std::chrono::*high\_resolution\_clock*>& *startTime*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *progressMutex*)  
{  
 std::lock\_guard<std::mutex> lock(*progressMutex*);  
  
 *long double* progress = *totalOperations* / (*size* \* *size*);  
 *for* (*auto*& point : *tracePoints*)  
 {  
 *if* (progress >= point)  
 {  
 *auto* currentTime = std::chrono::system\_clock::now();  
 std::chrono::duration<*double*> elapsedSeconds = currentTime **-** *startTime*;  
 std::cout **<<** "Progress: " **<<** *static\_cast*<*int*>(point \* 100) **<<** "%; Time elapsed: " **<<** elapsedSeconds.count() **<<** "s\n";  
 *tracePoints*.erase(std::remove(*tracePoints*.begin(), *tracePoints*.end(), point), *tracePoints*.end());  
 *break*;  
 }  
 }  
}  
  
  
*void* workerMutex(*const Matrix*& *m1*, *const Matrix*& *m2*, *Matrix*& *result*,  
 *int startRow*, *int endRow*, *long double*& *totalOperations*,  
 *long double size*, *const* std::chrono::time\_point<std::chrono::*high\_resolution\_clock*>& *startTime*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *mtx*, std::mutex& *progressMutex*)  
{  
 *for* (*int* i = *startRow*; i < *endRow*; ++i)  
 {  
 *for* (*int* j = 0; j < *m2***[**0**]**.size(); ++j)  
 {  
 *unsigned long long* sum = 0;  
 *for* (*int* k = 0; k < *m2*.size(); ++k)  
 {  
 sum += *m1***[**i**][**k**]** \* *m2***[**k**][**j**]**;  
 }  
 {  
 std::lock\_guard<std::mutex> guard(*mtx*);  
 *result***[**i**][**j**]** = sum;  
 *totalOperations*++;  
 operationTraceMutex(*totalOperations*, *size*, *startTime*, *tracePoints*, *progressMutex*);  
 }  
 }  
 }  
}  
  
  
*Matrix* multiplyMatricesMutex(*const Matrix*& *m1*, *const Matrix*& *m2*, *long double*& *totalOperations*,  
 *long double size*, *const* std::chrono::time\_point<std::chrono::*high\_resolution\_clock*>& *startTime*,  
 std::vector<*double*>& *tracePoints*, std::mutex& *mtx*, std::mutex& *progressMutex*)  
{  
 *int* rows = *m1*.size();  
 *int* cols = *m2***[**0**]**.size();  
 *Matrix* result(rows, std::vector<*unsigned long long*>(cols, 0));  
  
 std::vector<std::thread> threads;  
 *int* rowsPerThread = rows / 4;  
 *int* extraRows = rows % 4;  
  
 *for* (*int* i = 0; i < 4; ++i)  
 {  
 *int* startRow = i \* rowsPerThread;  
 *int* endRow = (i + 1) \* rowsPerThread + (extraRows > i ? 1 : 0);  
 threads.emplace\_back(workerMutex, std::cref(*m1*), std::cref(*m2*),  
 std::ref(result), startRow, endRow,  
 std::ref(*totalOperations*), *size*,  
 std::cref(*startTime*), std::ref(*tracePoints*),  
 std::ref(*mtx*), std::ref(*progressMutex*));  
 }  
  
 *for* (*auto*& thread : threads)  
 {  
 *if* (thread.joinable())  
 {  
 thread.join();  
 }  
 }  
  
 *return* result;  
}  
  
  
*Matrix* matrixPowerMutex(*const Matrix*& *matrix*, *int power*,  
 *long double size*, std::vector<*double*>& *tracePoints*,  
 std::mutex& *mtx*, std::mutex& *progressMutex*)  
{  
 *Matrix* result = *matrix*;  
 *Matrix* temp = *matrix*;  
  
 *auto* totalStartTime = std::chrono::*high\_resolution\_clock*::now();  
 *for* (*int* p = 1; p < *power*; ++p)  
 {  
 *long double* totalOperations = 0;  
 *auto* startTime = std::chrono::*high\_resolution\_clock*::now(); *// Час старту для цього множення* std::vector<*double*> tracePoints1 = *tracePoints*; *// Копія для кожного множення* result **=** multiplyMatricesMutex(result, temp, totalOperations,  
 *size*, startTime, tracePoints1, *mtx*, *progressMutex*);  
  
 *auto* endTime = std::chrono::*high\_resolution\_clock*::now(); *// Час завершення для цього множення* std::chrono::duration<*double*> elapsed = endTime **-** startTime;  
 std::cout **<<** "Total time for mutex multiplication " **<<** p **<<** ": " **<<** elapsed.count() **<<** "s\n";  
 }  
  
 *auto* totalEndTime = std::chrono::*high\_resolution\_clock*::now(); *// Загальний час завершення* std::chrono::duration<*double*> totalElapsed = totalEndTime **-** totalStartTime;  
 std::cout **<<** "Total time for all mutex multiplications: " **<<** totalElapsed.count() **<<** "s\n";  
  
 *return* result;  
}  
  
  
*Matrix* measureMutex(*const Matrix*& *matrix*, *int power*)  
{  
 std::mutex mtx;  
 std::mutex progressMutex;  
 *Matrix* result;  
  
 *long double* size = *matrix*.size();  
 std::vector<*double*> tracePoints = **{**0.1, 0.25, 0.5, 0.75, 1.0**}**;  
  
 result **=** matrixPowerMutex(*matrix*, *power*, size, tracePoints, mtx, progressMutex);  
  
 *return* result;  
}  
  
  
*int* main()  
{  
*///////////////////////////////////////////////////////////////////////////////////////////////* std::*string* filename = *R*"(D:\UNIK2\OS\LAB5.1\matrix.txt)";  
 *int* power = 2;  
  
 *Matrix* matrix;  
 *//Звичайне зчитування  
 auto* start = std::chrono::*high\_resolution\_clock*::now();  
 *if* (!readMatrixFromFile(filename, matrix))  
 {  
 *return* 1;  
 }  
 *auto* end = std::chrono::*high\_resolution\_clock*::now();  
  
 std::chrono::duration<*double*> elapsed = end **-** start;  
 std::cout **<<** "Standard reading time: " **<<** elapsed.count() **<<** " seconds" **<<** std::endl;  
  
*///////////////////////////////////////////////////////////////////////////////////////////////*#ifdef **\_WIN32** *// Спроба відкрити файл і створити відображення файлу в пам'яті  
 HANDLE* hMapFile; *//Дескриптор для зберігання файлу, який буде зберігатися у пам'яті  
 HANDLE* hFile = OpeNFileMapping(filename, hMapFile); *//Відкриття дескриптора  
  
 // Спроба відобразити файл у пам'яті  
 LPVOID* lpMapAddress = MapFile(hMapFile);  
 *if* (lpMapAddress == **NULL**)  
 {  
 *// Якщо відображення не вдалося, закриваємо відкриті дескриптори і виходимо з програми* CloseHandle(hMapFile);  
 CloseHandle(hFile);  
 *return* 1;  
 }  
  
 *// Початок вимірювання часу для процесу зчитування* start = std::chrono::*high\_resolution\_clock*::now();  
  
 *// Ініціалізація змінних для розмірів матриці  
 int* rows, cols;  
 std::vector<std::vector<*unsigned long long*>> matrix\_ram;  
  
 *//LPVOID вказує на початок цієї послідовності, але не містить інформації про тип даних або їхню структуру.  
 // Приведення до char\* дозволяє інтерпретувати ці байти як послідовність символів.  
  
 char*\* charPtr = *static\_cast*<*char*\*>(lpMapAddress); *// Приведення до char\*  
  
 //std::string має конструктор, який приймає вказівник на char і створює рядок, копіюючи символи до кінця рядка (тобто до першого входження нульового символу \0)* std::*string* strData(charPtr);  
 *// Читання матриці з відображення файлу в пам'яті* ReadMatrix(strData, rows, cols, matrix\_ram);  
  
 *// Закінчення вимірювання часу і розрахунок тривалості* end = std::chrono::*high\_resolution\_clock*::now();  
 elapsed **=** end **-** start;  
 std::cout **<<** "Memory-mapped reading time: " **<<** elapsed.count() **<<** " seconds" **<<** std::endl;  
  
*///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////*#else  
 *// Код для Linux...  
 int* fd = open(filename.c\_str(), O\_RDONLY);  
 *if* (fd == -1) {  
 std::cerr << "Не вдалося відкрити файл: " << strerror(errno) << std::endl;  
 *return* 1;  
 }  
  
 *struct* stat sb;  
 *if* (fstat(fd, &sb) == -1) {  
 close(fd);  
 std::cerr << "Помилка при отриманні розміру файлу: " << strerror(errno) << std::endl;  
 *return* 1;  
 }  
  
 size\_t length = sb.st\_size;  
 *void*\* addr = mmap(*nullptr*, length, PROT\_READ, MAP\_PRIVATE, fd, 0);  
 *if* (addr == MAP\_FAILED) {  
 close(fd);  
 std::cerr << "Не вдалося відобразити файл у пам'ять: " << strerror(errno) << std::endl;  
 *return* 1;  
 }  
  
 *// Читання даних із відображення в пам'яті...  
 // ...  
  
 // Прибирання* munmap(addr, length);  
 close(fd);  
  
#endif  
  
*// Початок виконання програми* std::cout **<<** "SINGLE start" **<<** std::endl;  
 *Matrix* matrixOneThread = measureOneThread(matrix\_ram, power);  
 writeMatrixToFile(matrixOneThread, *R*"(D:\UNIK2\OS\LAB5.1\output\_matrix.txt)");  
 std::cout **<<** "SINGLE successfully end" **<<** std::endl;  
  
 std::cout **<<** std::endl;  
  
 std::cout **<<** "SEMA start" **<<** std::endl;  
 *Matrix* matrixSemaphore = measureSemaphore(matrix\_ram, power);  
  
 start = std::chrono::*high\_resolution\_clock*::now();  
 writeMatrixToFile(matrixSemaphore, *R*"(D:\UNIK2\OS\LAB5.1\output\_matrix\_semaphore.txt)");  
 end = std::chrono::*high\_resolution\_clock*::now();  
 elapsed **=** end **-** start;  
 std::cout **<<** "Time to simple writing time: " **<<** elapsed.count() **<<** " seconds" **<<** std::endl;  
  
 start = std::chrono::*high\_resolution\_clock*::now();  
 writeMatrixToFileMapping(matrixSemaphore, *R*"(D:\UNIK2\OS\LAB5.1\output\_matrix.txt)");  
 end = std::chrono::*high\_resolution\_clock*::now();  
 elapsed **=** end **-** start;  
 std::cout **<<** "Time to RAM writing time: " **<<** elapsed.count() **<<** " seconds" **<<** std::endl;  
  
 std::cout **<<** "SEMA successfully end" **<<** std::endl;  
  
 std::cout **<<** std::endl;  
  
 std::cout **<<** "MUTEX start" **<<** std::endl;  
 *Matrix* matrixMutex = measureMutex(matrix\_ram, power);  
 writeMatrixToFile(matrixMutex, *R*"(D:\\UNIK2\\OS\\LAB5.1\\output\_matrix\_mutex.txt)");  
 std::cout **<<** "MUTEX successfully end" **<<** std::endl;  
  
 *return* 0;  
}