МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСТИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»

Інститут комп'ютерних наук та інформаційних технологій
Кафедра програмного забезпечення

ЗВІТ

до лабораторної роботи №5
на тему: «Багатопоточність в системі WINDOWS. Створення, керування та синхронізація потоків»
з дисципліни: «Операційні системи»

Лектор:
ст. викладач кафедри ПЗ
Грицай О. Д.

Виконав:
ст. гр. ПЗ-22
Чаус О. М.

Прийняла:
ст. викладач кафедри ПЗ
Грицай О. Д.

« ____ » _____ 2022 р.

∑= _____ .....:

Львів-2022

Тема роботи: Багатопоточність в системі WINDOWS. Створення, керування та синхронізація потоків

Мета роботи: знайомитися з багатопоточністю в ОС Windows. Навчитись реалізовувати розпаралелення алгоритмів за допомогою багатопоточності в ОС Windows з використанням функцій WinAPI. Навчитись використовувати різні механізми синхронізації потоків.

Теоретичні відомості

Розглядаючи поняття процесу, визначають ще одну абстракцію для запущеного процесу: потік. У класичному уявленні існує єдина точка виконання в рамках програми (тобто єдиний потік контролю, на якому збираються та виконуються інструкції), багатопотокова програма має більш ніж одну точку виконання (тобто кілька потоків контролю, кожен з яких який отримується та виконується).

Кожен потік дуже схожий на окремий процес, за винятком однієї відмінності: вони мають спільний адресний простір і, отже, мають доступ до одних і тих же даних. Таким чином, стан одного потоку дуже подібний до стану процесу. Він має лічильник програм (PC), який відстежує, звідки програма отримує інструкції. Кожен потік має свій власний приватний набір реєстрів, який він використовує для обчислень; таким чином, якщо на одному процесорі працюють два потоки, при переході від запуску одного (T1) до запуску іншого (T2) має відбутися перемикання контексту. Контекстний перемикач між потоками дуже подібний до перемикання контекстів між процесами, оскільки перед запуском T2 необхідно зберегти регістр стану T1 і відновити стан реєстру T2. За допомогою процесів ми зберегли стан до блоку управління процесами (PCB); тепер нам знадобиться один або кілька блоків управління потоками (TCB) для збереження стану кожного потоку процесу. Однак є одна істотна відмінність у перемиканні контексту, який ми виконуємо між потоками порівняно з процесами: адресний простір залишається незмінним (тобто немає необхідності змінювати, яку таблицю сторінок ми використовуємо).

Кожен потік починає виконання з якоїсь вхідної функції. У первинному потоці такою є main, wmain, WinMain або wWinMain. Якщо Ви хочете створити вторинний потік, в ньому теж має бути вхідна функція, яка виглядає приблизно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam)
{
      DWORD dwResult = 0;
      ...
      return(dwResult);
}
```

Функція потоку може виконувати будь-які завдання. Після того як вона закінчить свою роботу, поверне управління. У цей момент потік зупиниться, пам'ять, відведена під його стек, буде звільнена, а лічильник користувачів його об'єкта ядра «потік» зменшиться на 1. Коли лічильник обнулится, цей об'єкт ядра буде зруйнований.

Функцію потоку можна назвати як завгодно. Однак, якщо в програмі декілька функцій потоків, то потрібно присвоїти їм різні імена, інакше компілятор вирішить, що це кілька реалізацій єдиної функції. Функцій потоків передається єдиний параметр, сенс якого визначається розробником, а не операційною системою. Тому тут немає проблем з ANSI / Unicode.

Якщо треба створити додаткові потоки, потрібно викликати з первинного потоку функцію CreateThread:

```
HANDLE CreateThread(
      PSECURITY_ATTRIBUTESpsa;
      DWORD cbStack;
      PTHREAD_START_ROUTINE pfnStartAddr;
      PVOID pvParam;
      DWORD fdwCreate;
      DWORD pdwThreadID);
```

При кожному виклику цієї функції система створює об'єкт ядра «потік». Це не сам потік, а компактна структура даних, яка використовується операційною системою для управління потоком і зберігає статистичну інформацію про потік. Система виділяє пам'ять під стек потоку з адресного простору процесу. Новий потік виконується в контексті того ж процесу, що і батьківський потік. Тому він отримує доступ до всіх дескрипторів об'єктів ядра, всієї пам'яті і стека всіх потоків в процесі. За рахунок цього потоки в рамках одного процесу можуть легко взаємодіяти один з одним.

Існує три основні проблеми організації міжпоточної взаємодії:

1. Обмін даними.
2. Доступ до загальних ресурсів.
3. Узгодження дій.

КРИТИЧНІ СЕКЦІЇ. Основним способом запобігання проблем, пов'язаних з спільним доступом до загального ресурсу, є ЗАБОРОНА одночасного доступу до нього більш ніж одним потоком. Це означає, що в той момент, коли один потік використовує колективні дані, іншому потоку це робити заборонено. Проблема зі спулер виникла через те, що потік В розпочав роботу з спільно використовуваної змінної in до того, як потік А її закінчив. Дейкстра ввів поняття критичного інтервалу (критичної області, секції) - це фрагмент програми, в якому є звернення до спільно використовуваних даних. Для забезпечення правильної та ефективної спільної роботи декількох паралельних потоків, що оперують загальним ресурсом, необхідне виконання пяти умов: 1. У кожен момент часу тільки один потік може знаходитися в своїй критичній області. 2. Потік, що знаходиться поза критичною областю не може блокувати інші потоки. 3. Будь- який потік, готовий увійти в свою критичну область, має увійти в неї за кінцевий час. 4. У програмі не повинно бути припущень, щодо швидкості та кількості потоків. 5. Реалізація взаємодії повинна розглядати атомарними лише примітивні операції load, store, test.

Критичні секції

1) Необхідно створити структуру CRITICAL_SECTION. Структура повинна бути глобальною, щоб до неї могли звернутися різні потоки. Функції, керуючі критичними секціями, самі ініціалізують і модифікують елементи даної структури.

```
CRITICAL_SECTION cs;
```

2) Ініціалізувати критичну секцію, передавши адресу структури CRITICAL_SECTION.

```
VOID InitializeCriticalSection (LPCRITICAL_SECTION cs);
```

3) Вхід в критичну секцію:

```
VOID EnterCriticalSection (LPCRITICAL_SECTION cs);
```

Функція перевіряє значення елементів структури LPCRITICAL_SECTION і виконує наступні дії:

- Якщо ресурс вільний, модифікуються елементи структури, вказуючи, що викликаний

потік займає ресурс, після чого повертає управління, і потік, отримавши доступ до ресурсу, продовжує свою роботу.
- Якщо ресурс вже захоплений якимось потоком, функція оновлює елементи структури, відзначаючи, скільки разів поспіль цей потік захопив ресурс, і повертає управління.
- Якщо ресурс зайнятий іншим потоком - переводить і викликає потік в режим очікування. Потік, очікуючи входу в критичний інтервал, спить і не витрачає процесорного часу. Як тільки потік, який займає ресурс викличе LeaveCriticalSection, очікуваний потік буде пробуджений і отримає доступ до ресурсу.
4) Вихід із критичної секції:
```
VOID LeaveCriticalSection (LPCRITICAL_SECTION cs);
```
5) Якщо критична секція більше не потрібна, її слід видалити:
```
VOID DeleteCriticalSection (LPCRITICAL_SECTION cs);
```
Замість EnterCriticalSection можна скористатися функцією
```
VOID TryEnterCriticalSection (LPCRITICAL_SECTION cs);
```
Вона ніколи не призупиняє виконання виклику потоку. Якщо при її виклику зазначена критична секція не зайнята ні яким потоком, включаючи виклик, функція передає права на критичну секцію викликаному потоку і повертає TRUE. В іншому випадку вона лише тестує стан і негайно повертає FALSE. За допомогою критичних секцій не можна синхронізувати потоки з різних процесів. При спробі потоку увійти в критичну секцію, зайняту іншим потоком, він негайно призупиняється. Це значить, що потік переходить з режиму користувача в режим ядра, на що витрачається не менше 1000 тактів процесора. На багатопроцесорній машині потік, який володіє ресурсом, може виконуватися на іншому процесорі і може дуже швидко звільняти ресурс. Тоді існує ймовірність, що ресурс буде звільнений ще до переходу виклику потоку в режим ядра, тобто багато процесорного часу буде витрачено даремно. Для підвищення швидкодії критичних секцій в них інтегроване спінблокування: викликаючи EnterCriticalSection, функція виконує задану кількість циклів спін-блокування, намагаючись отримати доступ до ресурсу. Якщо всі спроби виявилися невдалими, функція переводить потік в режим ядра, де він перебуватиме в стані очікування. Семафор – універсальний механізм для організації взаємодії процесів та потоків. Визначення семафору зробив нідерландський вчений Едсегр Дейкстра в1965 році. Семафор – об'єкт ядра ОС, який можна розглядати як лічильник, що містить ціле число в діапазоні від 0 до заданого при його створенні максимального значення. При досягненні семафором значення 0, він переходить у несигнальний стан, а при будь яких інших значеннях – у сигнальний. Над семафором можна виконати такі операції: Ініціалізація – встановлення початкового значення. Перевірка стану семафору. Якщо семафор не рівний нулю, лічильник зменшується на 1, інакше процес блокується доти, поки лічильник не буде рівний 0.
Операція збільшення лічильника на 1.
Семафор, в залежності від значень, які він може приймати буває двійковий (0 та 1), трійковий (0, 1, 2) і т.д. Лічильник зменшується при кожному успішному завершенні функції очікування, і збільшується за допомогою виклику функції WINAPI.
З кожним семафором зв'язується список (черга) процесів, які очікують дозволи пройти семафор.
Для створення семафора служить функція CreateSemaphore:
```
HANDLE WINAPI CreateSemaphore(
      LPSECURITY_ATTRIBUTESlpSemaphoreAttributes, //атрибути доступу
      LONGlInitialCount, // початкове значення лічильника
      LONGlMaximumCount, // максимальне значення лічильника
```

```
        LPCTSTRlpName); // ім’я об’єкта
```

Функція повертає дескриптор створеного семафора, або 0, якщо створити його не вдалося. Якщо в системі вже є семафор з таким ім'ям, то новий не створюється, а повертається дескриптор існуючого. При використанні семафора в середині одного процесу можна його створити без імені, передавши в якості
lpName значення null. Ім'я семафора не повинне збігатися з іменем уже існуючого об'єкта типу Event, Mutex, WaitableTimer, Job, або FileMapping
Дескриптор раніше створеного семафора може бути отриманий функцією
`OpenSemaphore:`

```
HANDLE WINAPI OpenSemaphore(
        DWORD dwDesiredAccess, //задає права доступу до об’єкта
        BOOLbInheritHandle, //вказує, чи може семафор успадковуватись
        LPCTSTRPlpName); //ім’я об’єкта
```

Для збільшення лічильника семафора використовується функція
`ReleaseSemaphore:`

```
BOOL WINAPI ReleaseSemaphore(
HANDLE hSemaphore,// дескриптор
LONG lReleaseCount, // величина простору семафора
LPLONGlpPreviousCount); // адреса змінної, яка прийме попереднє значення
```

Якщо значення лічильника після виконання функції перевищить заданий для нього максимум, то функція ReleaseSemaphore поверне false і значення семафора не зміниться. Параметром lpPreviousCount можна передати null, якщо це значення програмі не потрібно. На відміну від м'ютекса, семафор не можна захопити у володіння, оскільки відсутнє саме поняття прав власності над ним. Крім цього, семафор може бути звільнений будь-яким потоком

Завдання для виконання лабораторної роботи

1. Реалізувати заданий алгоритм в окремому потоці.
2. Виконати розпаралелення заданого алгоритму на 2, 4, 8, 16 потоків.
3. Реалізувати можливість зупинку роботи і відновлення, зміни пріоритету певного потоку.
4. Реалізувати можливість завершення потоку.
5. Застосувати різні механізми синхронізації потоків. (Згідно запропонованих варіантів)
6. Зобразити залежність час виконання – кількість потоків (для випадку без синхронізації і зі синхронізацією кожного виду).
7. Результати виконання роботи відобразити у звіті.
13. Вивід найменшого слова в кожному рядку. (кількість рядків у файлі > 1000, текст довільна наукова стаття) Синхронізація: критична секція, семафор.

Хід роботи

1. Реалізував заданий алгоритм в окремому потоці.

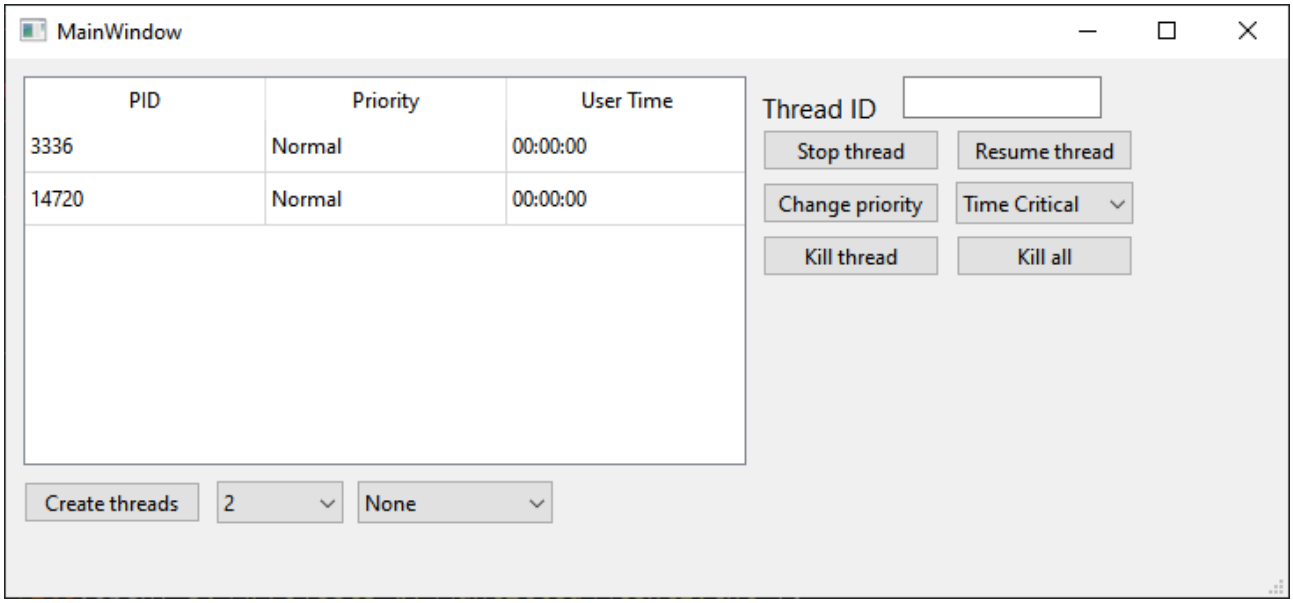
   Код розв’язку задачі:

```cpp
#include <Windows.h>
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <chrono>

struct lines {
        int start;
        int end;
};
```

```
std::size_t line_count = 0;

std::ifstream file;
CRITICAL_SECTION cs;
HANDLE my_semaphore;
std::vector<std::string> text;
std::vector<std::pair<HANDLE, DWORD>> threads;

std::vector<std::string> split(std::string str, std::string delim) {
	std::vector<std::string> res;
	std::string buffer;
	for (int i = 0; i < str.size(); i++) {
		bool is_delim = false;
		for (int j = 0; j < delim.size(); j++) {
			if (str[i] == delim[j]) {
				if (buffer != "") {
					res.push_back(buffer);
					buffer = "";
				}
				is_delim = true;
				break;
			}
		}
		if (!is_delim)
			buffer += str[i];
	}
	return res;
}

DWORD WINAPI critical_section(PVOID lpParam) {
	lines* thread_lines = (lines*)(lpParam);
	std::cout << GetCurrentThreadId() << " " << thread_lines->start << " " <<
thread_lines->end << "\n";
	std::string line;
	int start = thread_lines->start;
	int end = thread_lines->end;
	delete thread_lines;
	for (int i = start; i < end; i++) {
		line = text[i];
		std::vector<std::string> words = split(line, " ,.;:!?\"\'\n");
		std::string min_word = words[0];
		for (auto& word : words) {
			if (word.size() < min_word.size())
				min_word = word;
		}
		EnterCriticalSection(&cs);
		std::cout << i << ": " << min_word << "\n";
		LeaveCriticalSection(&cs);
	}
	return 0;
}


DWORD WINAPI semaphore(PVOID lpParam) {
	lines* thread_lines = (lines*)(lpParam);
	std::cout << GetCurrentThreadId() << " " << thread_lines->start << " " <<
thread_lines->end << "\n";
	std::string line;
	int start = thread_lines->start;
	int end = thread_lines->end;
	delete thread_lines;
	for (int i = start; i < end; i++) {
		line = text[i];
		std::vector<std::string> words = split(line, " ,.;:!?\"\'\n");
		std::string min_word = words[0];
		for (auto& word : words) {
			if (word.size() < min_word.size())
```

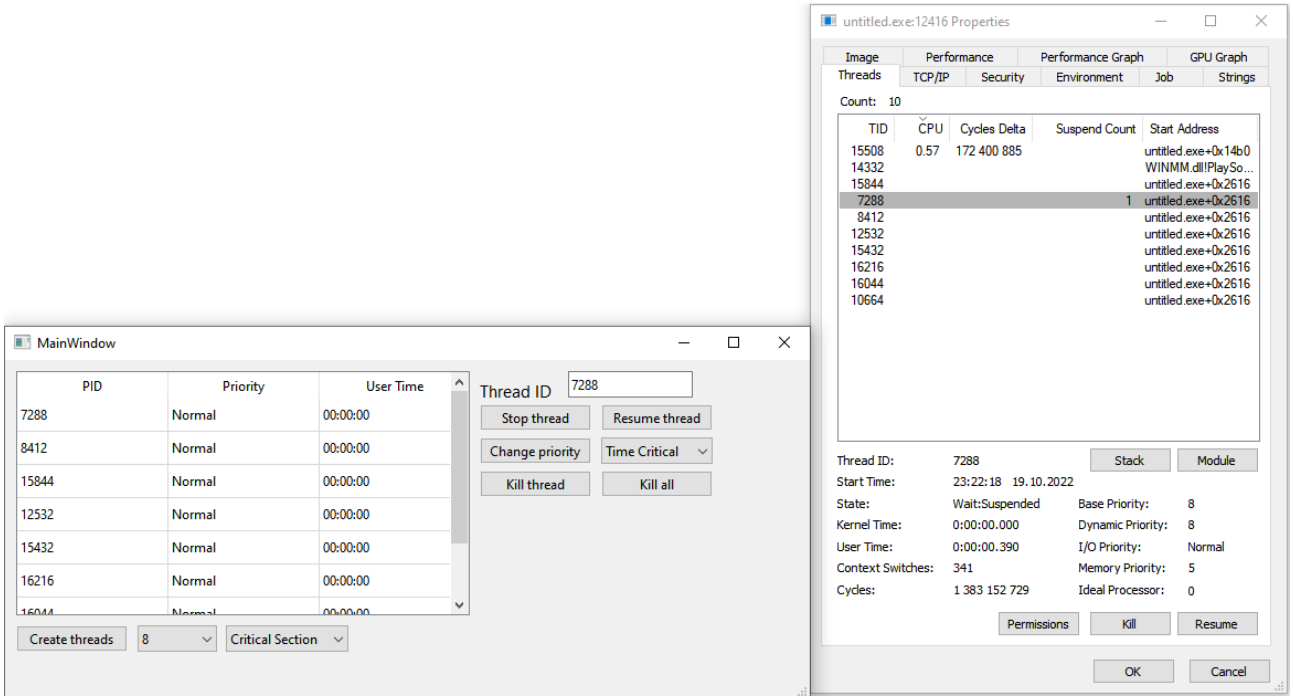
```
                        min_word = word;
                }
                WaitForSingleObject(my_semaphore, INFINITE);
                std::cout << i << ": " << min_word << "\n";
                ReleaseSemaphore(my_semaphore, 1, NULL);
        }
        return 0;
}

DWORD WINAPI none(PVOID lpParam) {
        lines* thread_lines = (lines*)(lpParam);
        std::cout << GetCurrentThreadId() << " " << thread_lines->start << " " <<
thread_lines->end << "\n";
        std::string line;
        int start = thread_lines->start;
        int end = thread_lines->end;
        delete thread_lines;
        for (int i = start; i < end; i++) {
                line = text[i];
                std::vector<std::string> words = split(line, " ,.;:!?\"\'\n");
                std::string min_word = words[0];
                for (auto& word : words) {
                        if (word.size() < min_word.size())
                                min_word = word;
                }
                std::cout << i << ": " << min_word << "\n";
        }
        return 0;
}

int main() {
        my_semaphore = CreateSemaphore(NULL, 1, 1, NULL);
        InitializeCriticalSection(&cs);
        file.open("text1.txt");
        std::string line;
        while (std::getline(file, line)) {
                text.push_back(line);
        }
        auto start = std::chrono::high_resolution_clock::now();
        lines* thread_lines = new lines;
        thread_lines->start = 0;
        thread_lines->end = text.size();
        DWORD ThreadID;
        HANDLE handle = CreateThread(NULL, NULL, none, thread_lines, NULL, &ThreadID);
        WaitForSingleObject(handle, INFINITE);
        auto end = std::chrono::high_resolution_clock::now();
        auto time = std::chrono::duration_cast<std::chrono::milliseconds>(end - start);
        std::cout << "time elapsed: " << time.count() << " ms.\n";
        DeleteCriticalSection(&cs);
        delete[] handles;
        return 0;
}
```

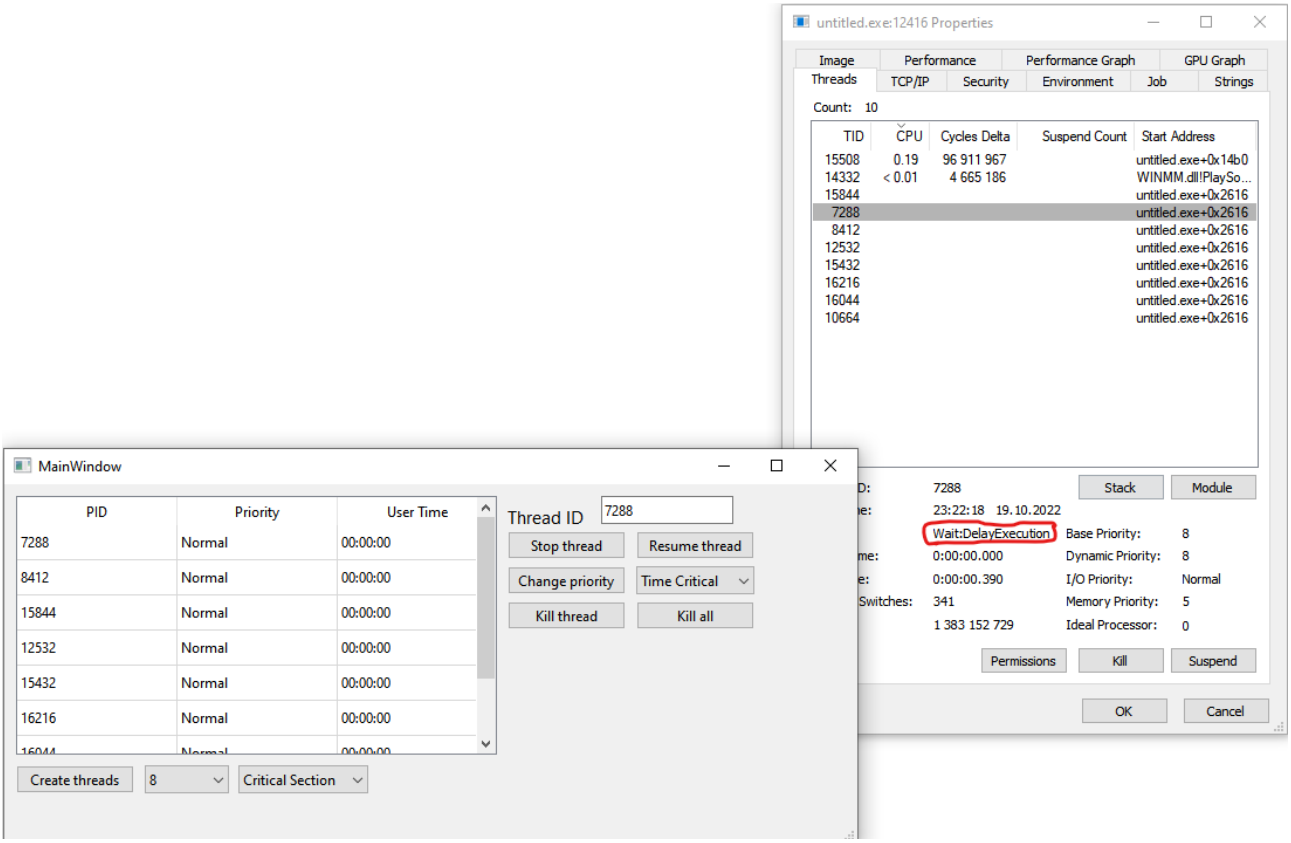
1. Реалізував розв’язок задачі у 2-ох, 4-ох, 8-ох та 16-ох потоках. Виміряв час роботи потоків.



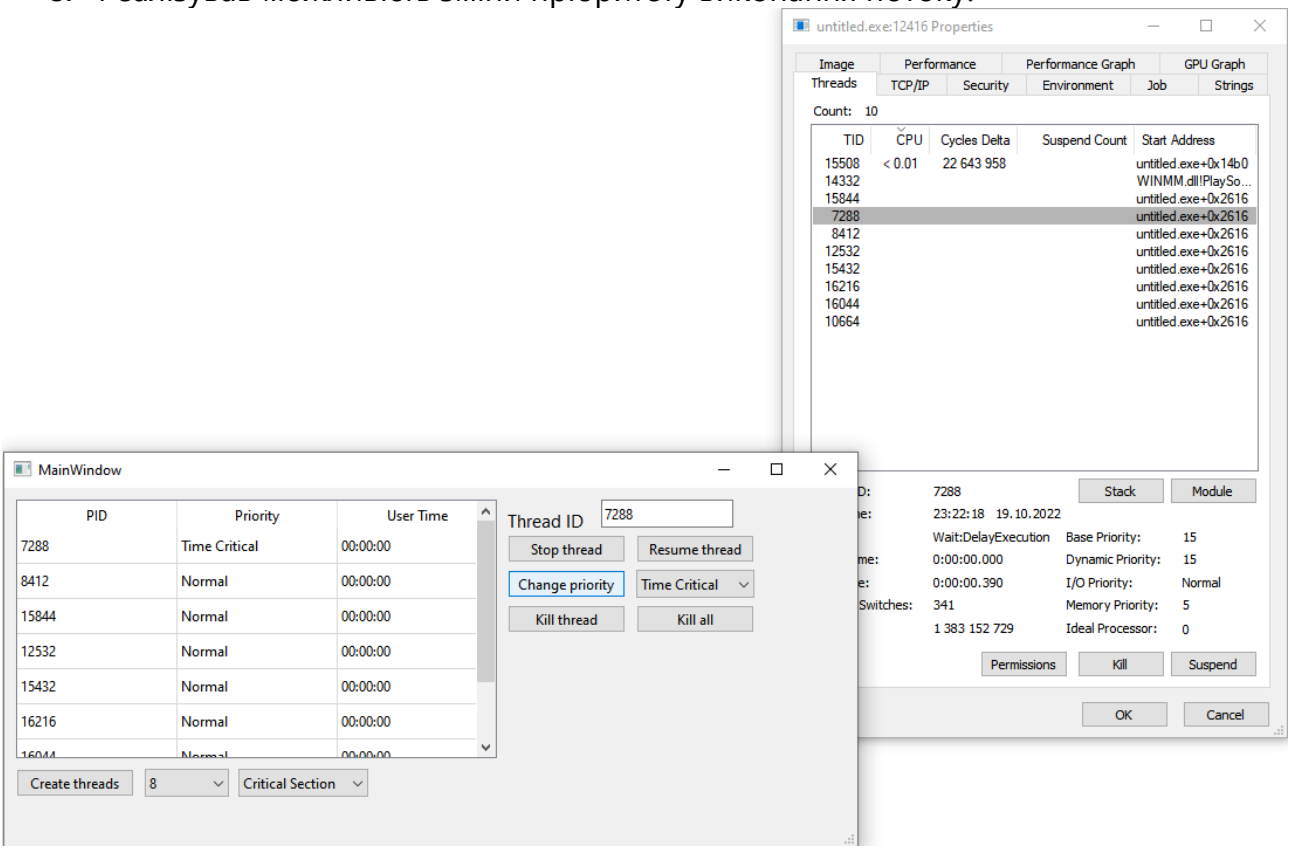
Інтерфейс програми

2. Для кожного потоку реалізував можливість його запуску та зупинення. На зображенні показано зазначений функціонал.

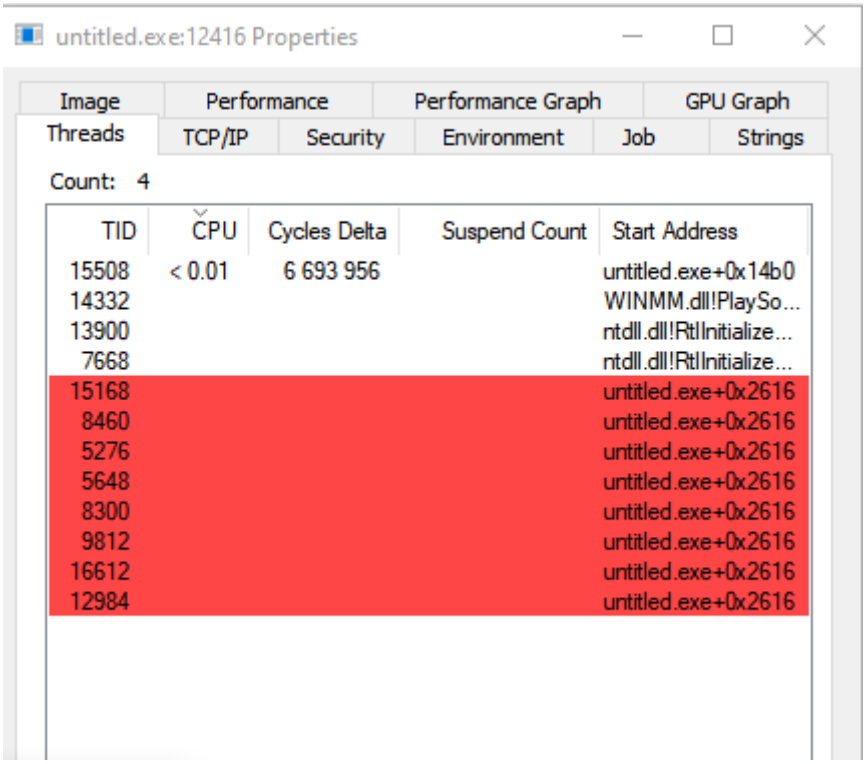




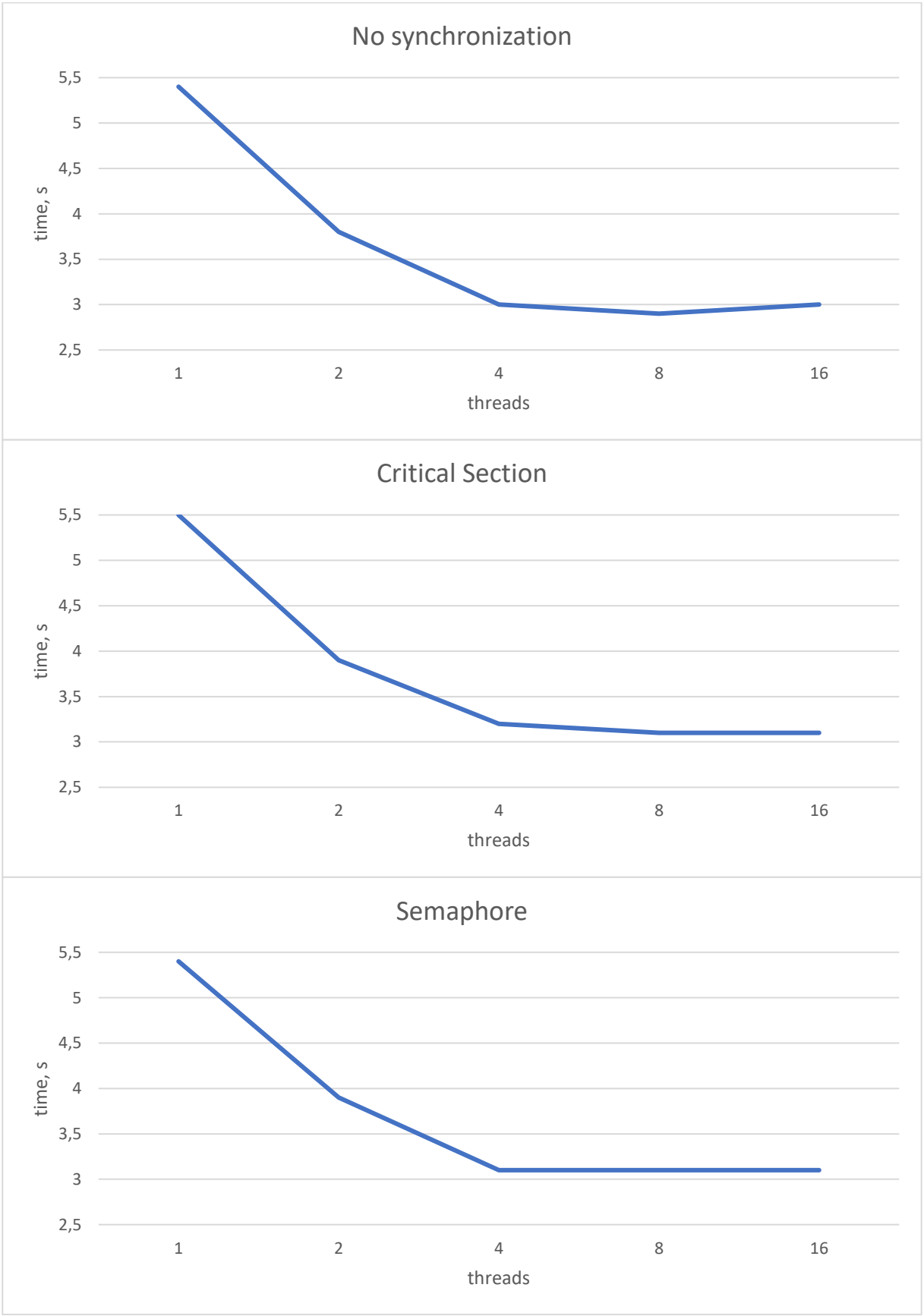
3. Реалізував можливість зміни пріоритету виконання потоку.

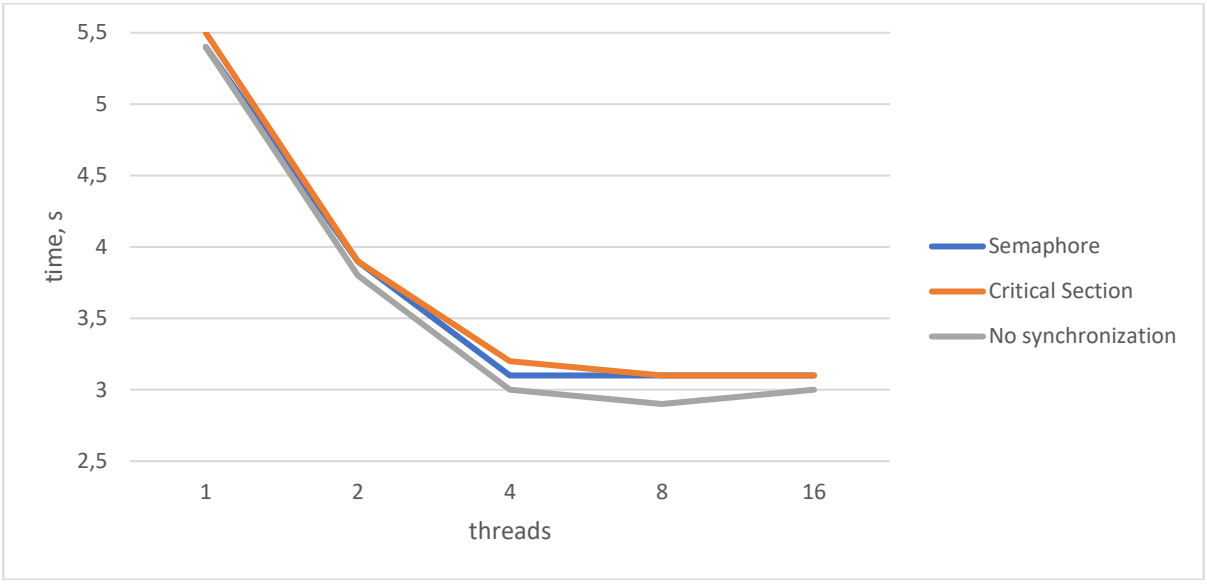


4. Реалізував можливість завершення потоку.



5. Виміряв час виконання для 2, 4, 8 та 16 потоків та зобразив залежність кількість потоків-час для кожного виду синхронізації.


No synchronization
5,5
5
4,5
4
3,5
3
2,5
time, s
1
2
4
8
16
threads
Critical Section
5,5
5
4,5
4
3,5
3
2,5
time, s
1
2
4
8
16
threads
Semaphore
5,5
5
4,5
4
3,5
3
2,5
time, s
1
2
4
8
16
threads



Висновок: під час виконання лабораторної роботи ознайомився з потоками в системі Windows, навчився їх створювати, змінювати пріоритет і стан, завершувати. Навчився синхронізувати потоки використовуючи критичні секції та семафори.