Міністерство Освіти І НАУКИ України
Національний університет "Львівська політехніка"

Інститут **ІКНІ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 5
На тему: *“Багатопоточність в операційній системі WINDOWS. Створення, керування та синхронізація потоків”*
З дисципліни: *“Операційні системи”*

Лектор:
Старший викладач ПЗ
Грицай О.Д.

Виконав:
ст. гр. ПЗ-22
Солтисюк Д.А.

Прийняв:
Старший викладач ПЗ
Грицай О.Д.

« ____ » _____ 2022 р.

∑= ____

Львів – 2022

Тема роботи: Багатопоточність в операційній системі WINDOWS. Створення, керування та синхронізація потоків.
Мета роботи: Ознайомитися з багатопоточністю в ОС Windows. Навчитись реалізовувати розпаралелювання алгоритмів за допомогою багатопоточності в ОС Windows з використанням функцій WinAPI. Навчитись використовувати різні механізми синхронізації потоків.

Індивідуальне завдання

1. Реалізувати заданий алгоритм в окремому потоці.
2. Виконати розпаралелювання 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.

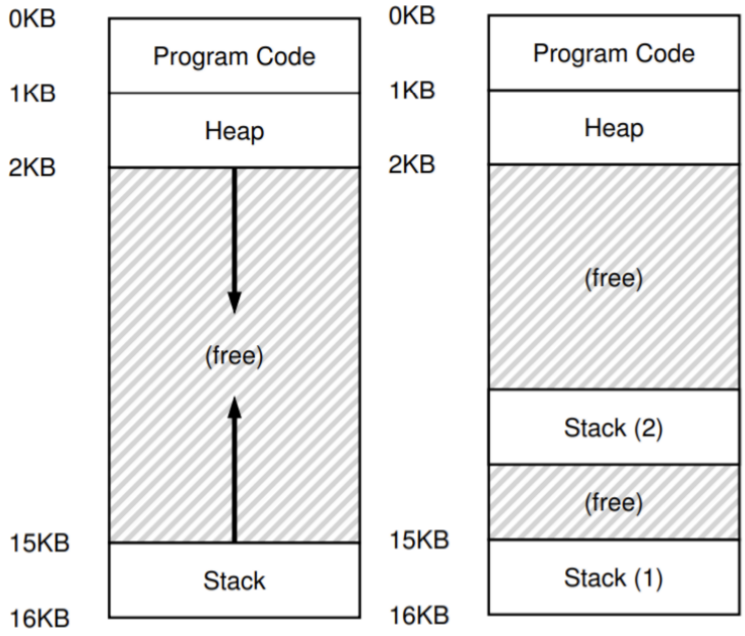
Варіант завдання №2:

2. Обчислити суму елементів заданого масиву (кількість елементів >10000, елементи рандомні). Синхронізація: 3, 2.

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). Ще одна істотна відмінність між потоками та процесами стосується стека. У простій моделі адресного простору класичного процесу (однопотокового) є єдиний стек, який зазвичай знаходиться внизу адресного простору. Однак у багатопотоковому процесі кожен потік працює окремо і, звичайно, може залучати різні підпрограми для виконання будь -якої роботи. Замість одного стека в адресному просторі буде по одному на кожен потік.



На цьому малюнку можна побачити два стеки, розповсюджені по адресному простору процесу. Таким чином, будь-які змінні, параметри, повернені значення, що виділяються стеком, та інші речі, які розміщуємо у стеку, будуть розміщені у тому, що іноді називають локальним сховищем потоків, тобто стеком відповідного потоку. Раніше стек і купа могли зростати незалежно, і проблеми виникали лише тоді, коли в адресному просторі вичерпалося місце. Тут немає такої приємної ситуації, оскільки стеки, як правило, не повинні бути дуже великими (виняток становлять програми, які дуже часто використовують рекурсію).

Хід роботи

Створю графічний інтерфейс для програми, яка запускатиме потоки. Врахую вибір синхронізацій, пріоритетів, а також можливості призупинити, відновити, вбити потоки:

Також додам відлік часу, який рахуватиме, скільки потрібно чекати, щоб усі потоки виконалися та був знайдений потрібний рядок.

Запрограмую рішення.

Код:

```
#include "mainwindow.h"

#include <tchar.h>
#include <time.h>

#include <algorithm>
#include <fstream>
#include <iostream>

#include "ui_mainwindow.h"

// semaphor and interlock practice

HANDLE sumSemaphore;

int duration;

std::vector<int> numbers;
volatile long resulting_sum = 0;

struct parameters {
  int from;
  int to;
  std::string method;
};

#define MAX_PROCS 16
#define nums_size 500000
std::array priorities = {THREAD_PRIORITY_TIME_CRITICAL,
THREAD_PRIORITY_HIGHEST,
                         THREAD_PRIORITY_ABOVE_NORMAL,
THREAD_PRIORITY_NORMAL,
                         THREAD_PRIORITY_BELOW_NORMAL,
THREAD_PRIORITY_LOWEST,
                         THREAD_PRIORITY_IDLE};
```

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), ui(new Ui::MainWindow) {
  ui->setupUi(this);
  for (int i = 0; i < MAX_PROCS; i++) {
    for (int j = 0; j < 2; j++) {
      ui->table->setItem(i, j, new QTableWidgetItem);
    }
  }

  for (int i = 1; i < nums_size; i++) {
    numbers.push_back(i);
  }

  ui->priorCb->addItem("Realtime");
  ui->priorCb->addItem("High");
  ui->priorCb->addItem("Above normal");
  ui->priorCb->addItem("Normal");
  ui->priorCb->addItem("Below normal");
  ui->priorCb->addItem("Low");
  ui->priorCb->addItem("Idle");

  ui->syncCb->addItem("Semaphore");
  ui->syncCb->addItem("Interlock");

  // Creating Semaphore
  sumSemaphore = CreateSemaphore(NULL,   // default security attributes
                                 1,      // initial count
                                 1,      // max count
                                 NULL    // unnamed
  );
}

MainWindow::~MainWindow() {
  delete ui;
  CloseHandle(sumSemaphore);
}

void MainWindow::updateThreadInfo() {
  for (unsigned long i = 0; i < MAX_PROCS; i++)
    for (unsigned long j = 0; j < 2; j++) ui->table->item(i, j)->setText("
");
  for (unsigned long i = 0; i < handles.size(); i++) {
    QString threadID = QString::number((int)threadIDs[i]);
    ui->table->item(i, 0)->setText(threadID);
    ui->table->item(i, 1)->setText(getPriority(handles[i]));
  }
}

QString MainWindow::getPriority(HANDLE handle) {
  int priority = GetThreadPriority(handle);
  switch (priority) {
    case THREAD_PRIORITY_ABOVE_NORMAL:
      return QString("Above normal");
    case THREAD_PRIORITY_BELOW_NORMAL:
```

```
      return QString("Below normal");
    case THREAD_PRIORITY_HIGHEST:
      return QString("High");
    case THREAD_PRIORITY_IDLE:
      return QString("Idle");
    case THREAD_PRIORITY_LOWEST:
      return QString("Low");
    case THREAD_PRIORITY_NORMAL:
      return QString("Normal");
    case THREAD_PRIORITY_TIME_CRITICAL:
      return QString("Realtime");
    default:
      return QString("?");
  }
}

DWORD WINAPI countSum(LPVOID lpParam) {
  parameters *params = (parameters *)lpParam;
  // perform calculations
  int partial_sum = 0;
  for (int idx = params->from; idx < params->to; idx++) {
    partial_sum += numbers[idx];
  }

  // updated shared variable
  if (params->method == "Semaphore") {
    WaitForSingleObject(sumSemaphore, INFINITE);
    resulting_sum += partial_sum;
    ReleaseSemaphore(sumSemaphore,  // handle to semaphore
                     1,             // increase count by one
                     NULL);         // not interested in previous count
  } else {
    InterlockedExchangeAdd(&resulting_sum, partial_sum); // atomic access
  }

  return 0;
}

void MainWindow::on_createBtn_clicked() {
  countThread = ui->threadCountCb->currentText().toInt();
  HANDLE pThread;
  DWORD pdwThreadID;
  parameters *params;
  std::string method = ui->syncCb->currentText().toStdString();

  int batch_size = numbers.size() / countThread;
  int bonus_size = numbers.size() % countThread;
  for (int from = 0, to = batch_size; from < numbers.size();
       from = to, to = from + batch_size) {
    if (bonus_size) {
      to++;
      bonus_size--;
    }
    params = new parameters();
```

```
    params->from = from;
    params->to = to;
    params->method = method;
    pThread = CreateThread(NULL,  // default security attributes
                           0,     // default stack size
                           countSum, // function address
                           (LPVOID)params,    // thread function arguments
                           CREATE_SUSPENDED,  // default creation flags
                           &pdwThreadID);     // receive thread identifier

    handles.push_back(pThread);
    threadIDs.push_back((int)pdwThreadID);
  }

  updateThreadInfo();
}

int MainWindow::findThread(int threadID) {
  for (int i = 0; i < threadIDs.size(); i++) {
    if (threadID == threadIDs[i]) return i;
  }
  return -1;
}

void MainWindow::on_suspendBtn_clicked() {
  int threadID = ui->txtPid->toPlainText().toInt();
  int index = findThread(threadID);

  SuspendThread(handles[index]);
}

void MainWindow::on_resumeBtn_clicked() {
  int threadID = ui->txtPid->toPlainText().toInt();
  int index = findThread(threadID);

  ResumeThread(handles[index]);
}

void MainWindow::on_killBtn_clicked() {
  int threadID = ui->txtPid->toPlainText().toInt();
  int index = findThread(threadID);

  TerminateThread(handles[index], NULL);
  CloseHandle(handles[index]);
  handles.erase(handles.begin() + index);
  threadIDs.erase(threadIDs.begin() + index);
  countThread--;
  updateThreadInfo();
}

void MainWindow::on_killAllBtn_clicked() {
  for (int i = 0; i < handles.size(); i++) {
    TerminateThread(handles[i], NULL);
    CloseHandle(handles[i]);
```

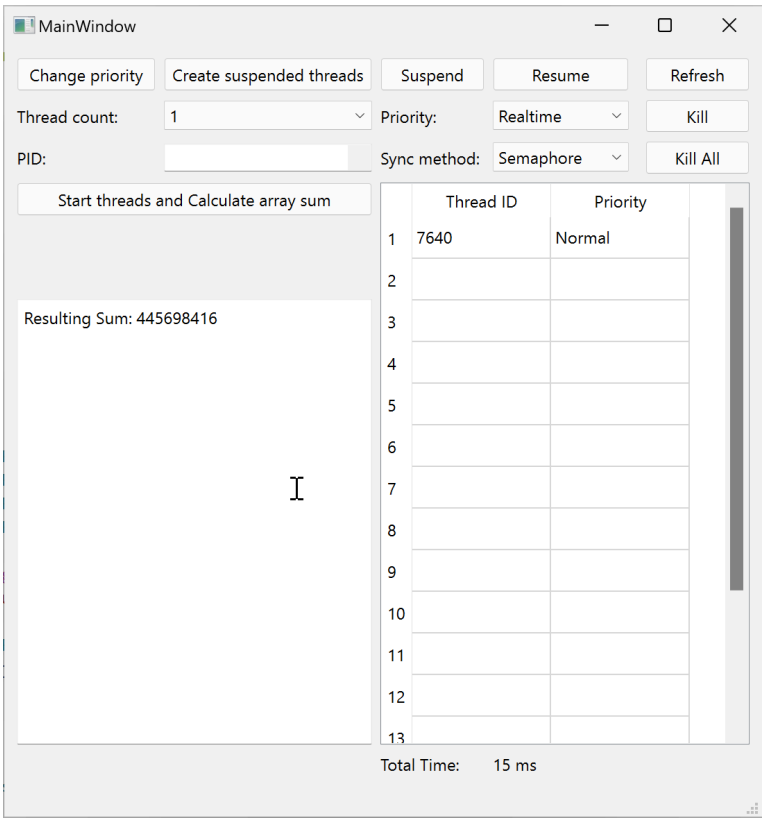
```
  }
  handles.clear();
  threadIDs.clear();
  updateThreadInfo();
}

void MainWindow::on_priorBtn_clicked() {
  int threadID = ui->txtPid->toPlainText().toInt();
  int index = findThread(threadID);

  SetThreadPriority(handles[index], priorities[ui->priorCb->currentIndex()]);
  updateThreadInfo();
}

void MainWindow::on_resultBtn_clicked() {
  const auto begin = std::chrono::high_resolution_clock::now();
  for (int i = 0; i < handles.size(); i++) ResumeThread(handles[i]);

  // wait for threads to complete
  WaitForMultipleObjects(handles.size(), &handles[0], true, INFINITE);

  for (int i = 0; i < handles.size(); i++) CloseHandle(handles[i]);

  const auto time = std::chrono::duration_cast<std::chrono::milliseconds>(
      std::chrono::high_resolution_clock::now() - begin);

  duration = time.count();
  ui->txtSentence->setText("Resulting Sum: " +
QString::number(resulting_sum));
  resulting_sum = 0;
  ui->timeLbl->setText(QString::number(duration) + " ms");
}

void MainWindow::on_refreshBtn_clicked() { updateThreadInfo(); }
```

Протокол роботи

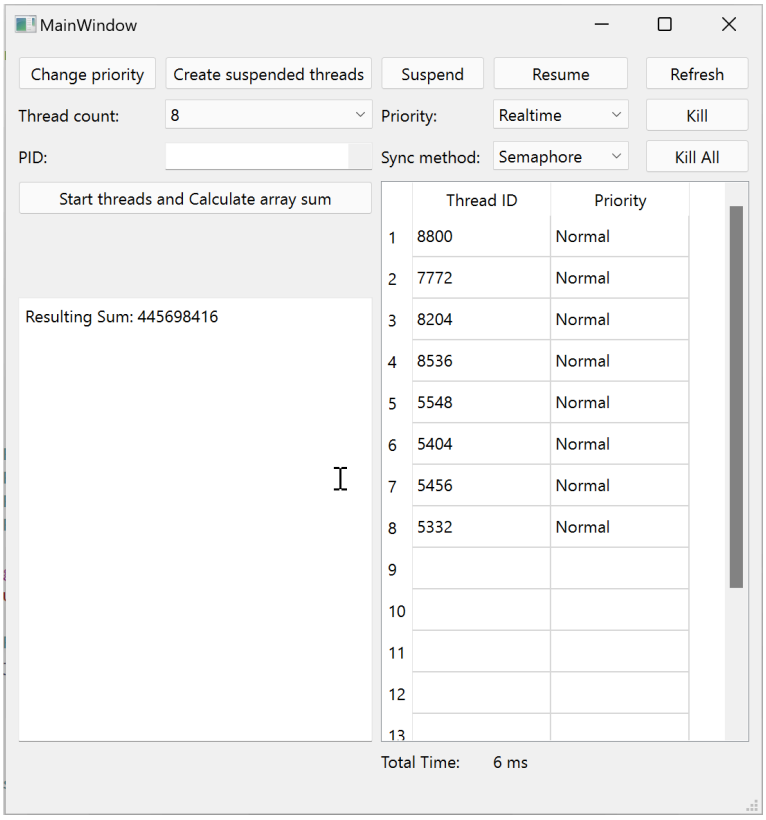
Запущу 1 потік, щоб порівняти роботу без та із синхронізаціями.

Порахую суму елементів масивів використовуючи Semaphore

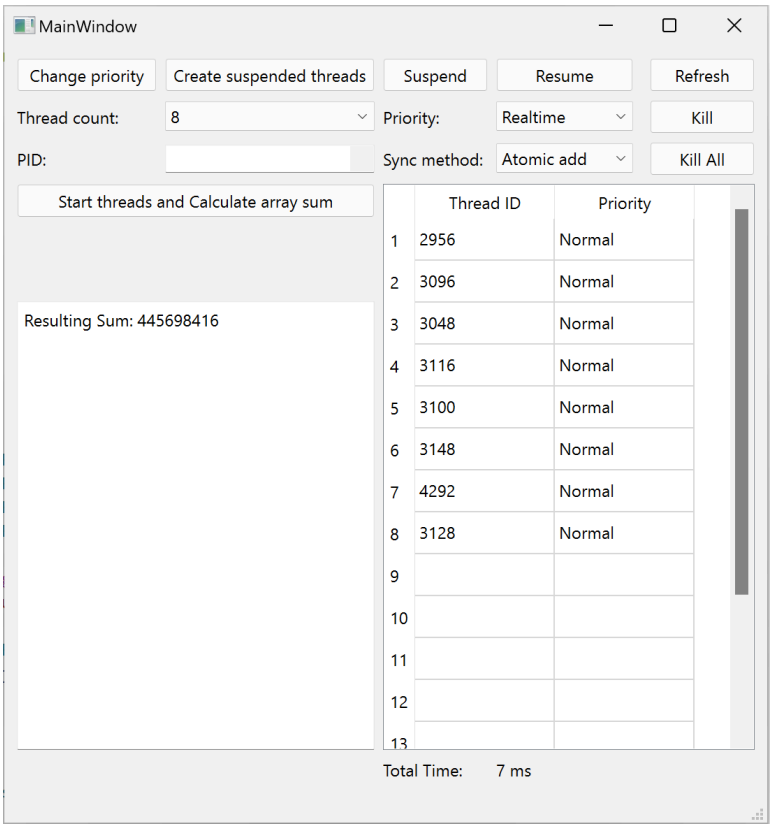
Запущу потік та виведу результат:



Результат з 8ми потоками, бачимо покращення в часі (майже в два рази)



Використовуючи атомарні функції:



Висновок

У цій лабораторній роботі я навчився працювати із потоками та методами їхньої синхронізації, використовуючи WinAPI, розбив завдання на окремі частинки та використав синхронізацію, щоб отримати потрібний результат.