**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

Климок Н.І.

**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 №7**:

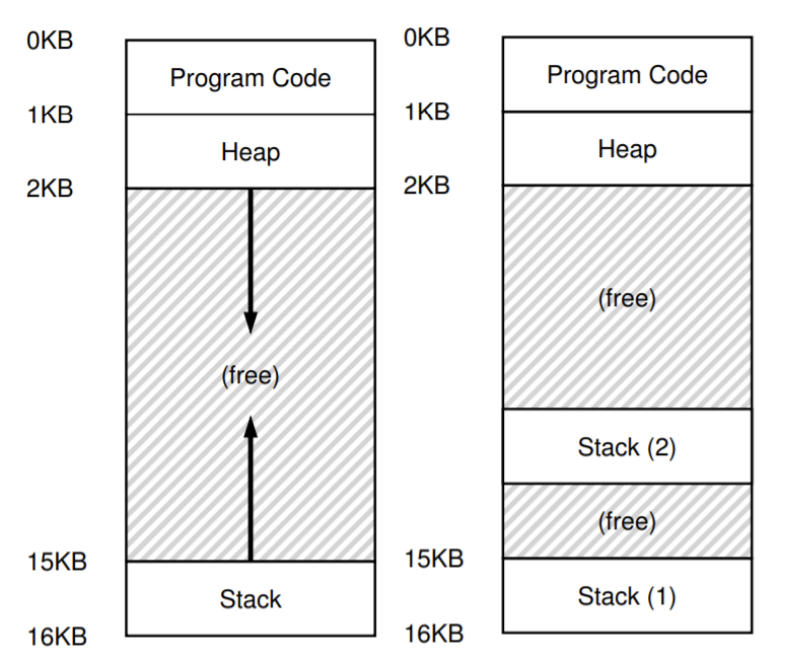
Пошук заданого слова у файлі і вивід рядка з тим словом (кількість рядків у файлі > 1000, текст довільна наукова стаття).

**Синхронізація**: 1, 4 (Мютекс, критична секція).

**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).

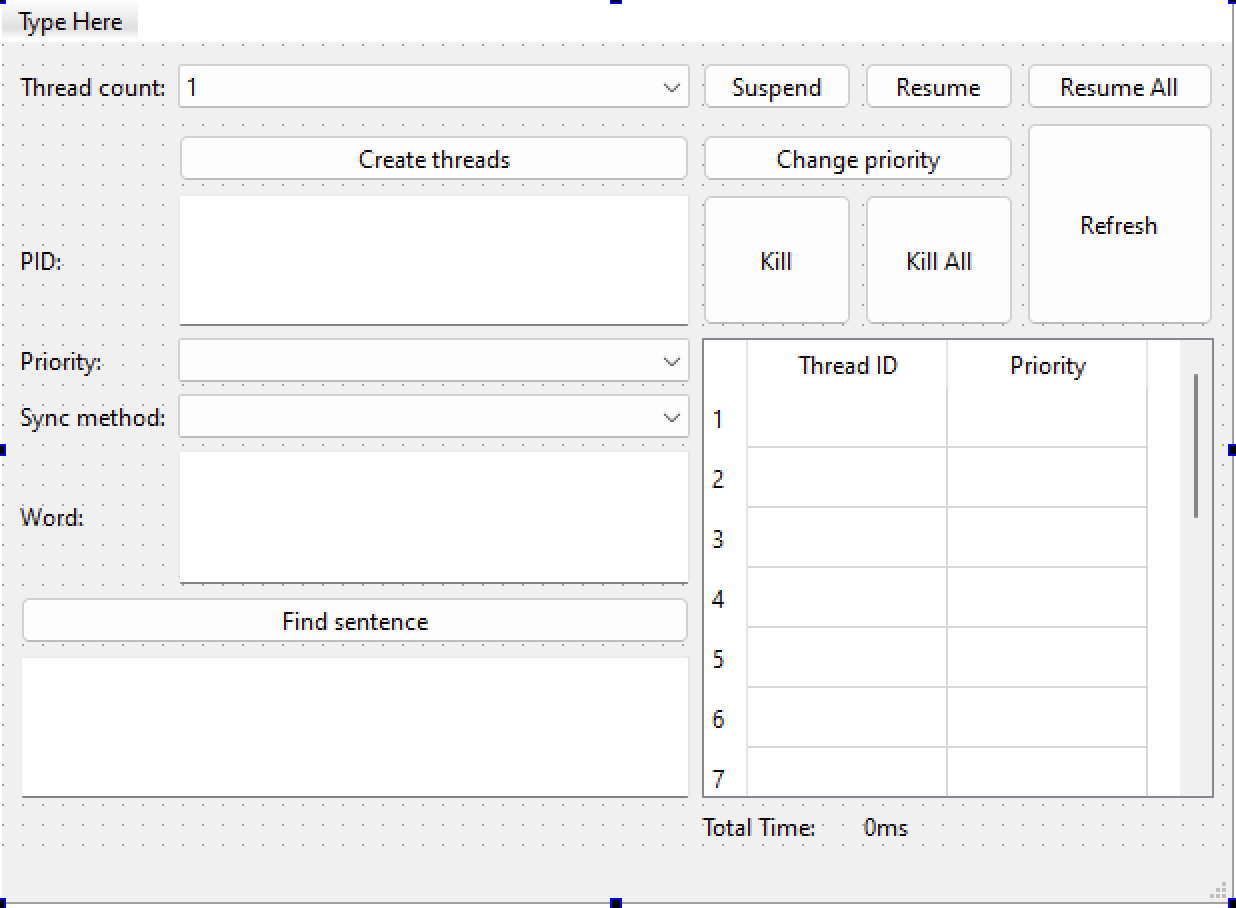
Кожен потік дуже схожий на окремий процес, за винятком однієї відмінності: вони мають спільний адресний простір і, отже, мають доступ до одних і тих же даних. Таким чином, стан одного потоку дуже подібний до стану процесу. Він має лічильник програм (PC), який відстежує, звідки програма отримує інструкції. 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 Т2 необхідно зберегти регістр стану Т1 і відновити стан реєстру Т2. 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 поотоки. 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 "ui\_mainwindow.h" #include <tchar.h> #include <time.h> #include <algorithm> #include <fstream>  HANDLE hMutex; CRITICAL\_SECTION CriticalSection;  int maxIndex, max, duration;  int totalLines; int bytesRead = 0; int whoFound = 0; std::string resultSentence = "";  std::string filename = "C:\\Users\\N\\Desktop\\txtfile.txt";  struct parameters {  int lineCount;  std::string word;  std::string method; };  #define MAX\_PROCS 16 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);  }  }  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("Mutex");  ui->syncCb->addItem("Critical Section");  std::ifstream inFile(filename);  totalLines = std::count(std::istreambuf\_iterator<char>(inFile),  std::istreambuf\_iterator<char>(), '\n');  inFile.close();  // Creating Mutex  hMutex = CreateMutex(  NULL,  FALSE,  NULL);  // Creating critical section  if (!InitializeCriticalSectionAndSpinCount(&CriticalSection,  0x00000400) )  exit(-400); // custom exit code }  MainWindow::~MainWindow() {  delete ui;  CloseHandle(hMutex);  DeleteCriticalSection(&CriticalSection); }  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("WTF");  } }   DWORD WINAPI findSentence(LPVOID lpParam) {  parameters \*params = (parameters\*) lpParam;  int maxLineCount = params->lineCount;  std::string word = params->word;  std::string method = params->method;  // TODO sync  DWORD dwWaitResult = 0;  if (method == "Mutex") {  dwWaitResult = WaitForSingleObject(  hMutex,  INFINITE);  } else {  EnterCriticalSection(&CriticalSection);  }  if (!resultSentence.empty()) {  if (method == "Mutex") {  ReleaseMutex(hMutex);  } else {  LeaveCriticalSection(&CriticalSection);  }  return 1;  }   std::ifstream inFile(filename);  inFile.seekg(bytesRead);  std::string line;  for (int i = 0; i < maxLineCount; i++) {  getline (inFile, line);  bytesRead += line.length();  if (line.find(word) != std::string::npos) {  resultSentence = line;  whoFound = GetCurrentThreadId();  break;  }  }  inFile.close();  if (method == "Mutex") {  ReleaseMutex(hMutex);  } else {  LeaveCriticalSection(&CriticalSection);  }  return 0; }  void MainWindow::on\_createBtn\_clicked() {  resultSentence = "";  countThread = ui->threadCountCb->currentText().toInt();  HANDLE pThread;  PDWORD pdwThreadID;  parameters\* params;  std::string method = ui->syncCb->currentText().toStdString();  for (int i = 0; i < countThread; i++) {  params = new parameters();  params->lineCount = std::ceil((totalLines + 0.0) / countThread);  params->method = method;  params->word = ui->txtWord->toPlainText().toStdString();   pdwThreadID = new DWORD;  pThread = CreateThread(NULL, 0, findSentence, (LPVOID) params, CREATE\_SUSPENDED, pdwThreadID);  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() {  ui->txtSentence->setText(QString::fromStdString(resultSentence));  ui->timeLbl->setText(QString::number(duration) + " | Who found: " + QString::number(whoFound)); }  void MainWindow::on\_fullResumeBtn\_clicked() {  const auto begin = std::chrono::high\_resolution\_clock::now();  for (int i = 0; i < handles.size(); i++)  ResumeThread(handles[i]);  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(); } |
| --- |

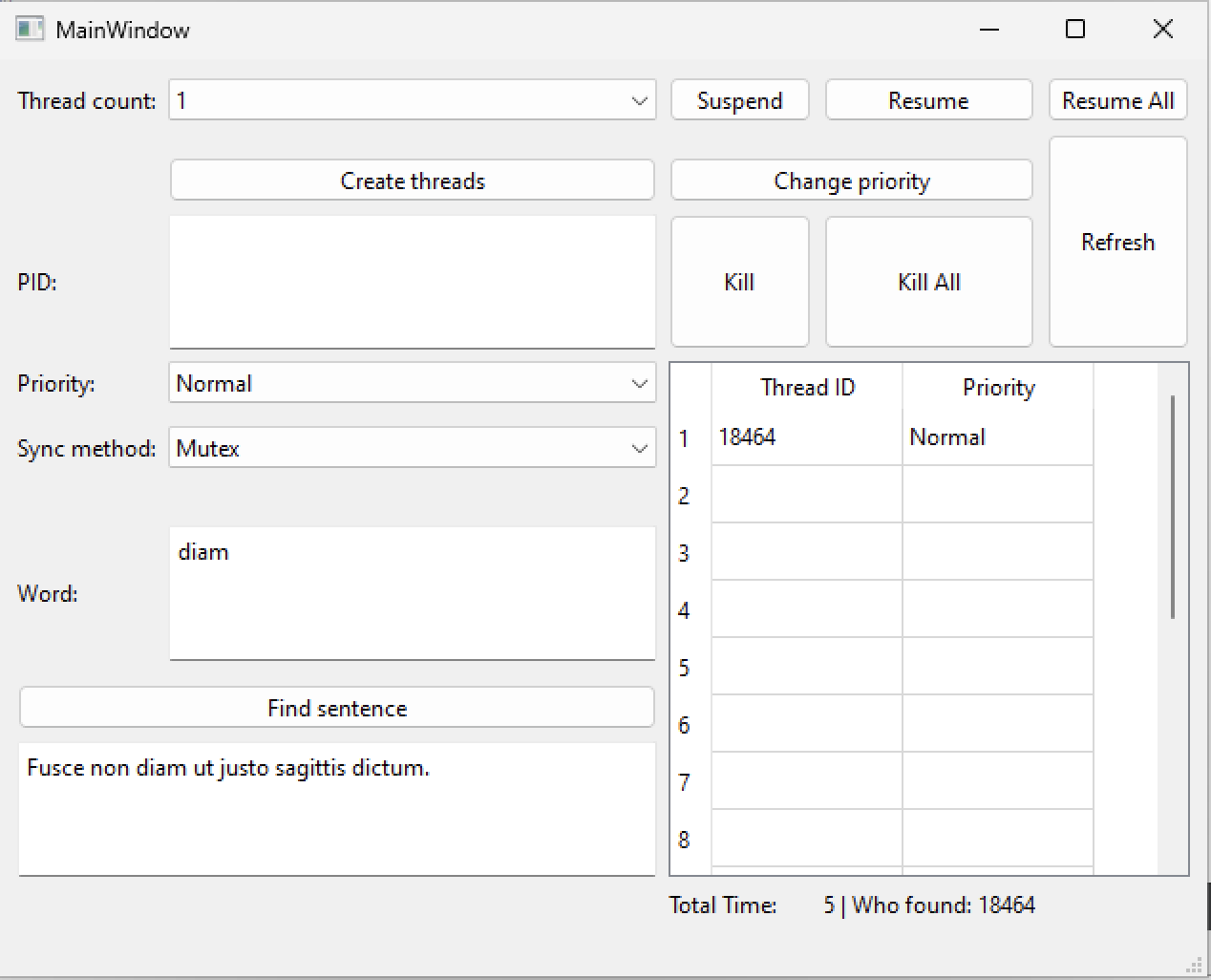
**Протокол роботи**

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

Згенерую та збережу у текстовий файл 1053 рядки псевдо-тексту “lorem ipsum”.

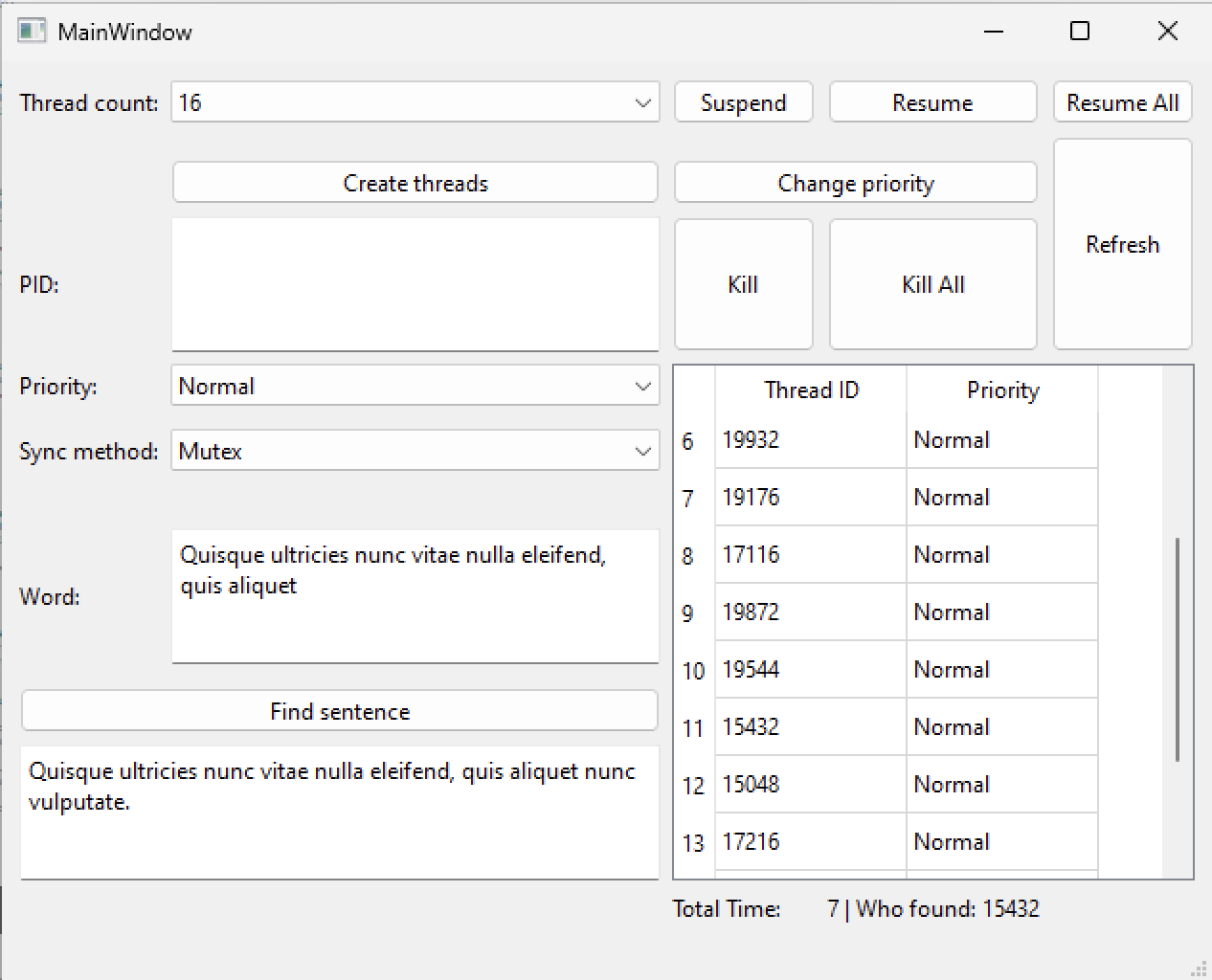
Спробую знайти слово “diam”, яке доволі часто зустрічається у тексті, із синхронізацією Mutex.

Запущу потік та виведу результат (час, речення):



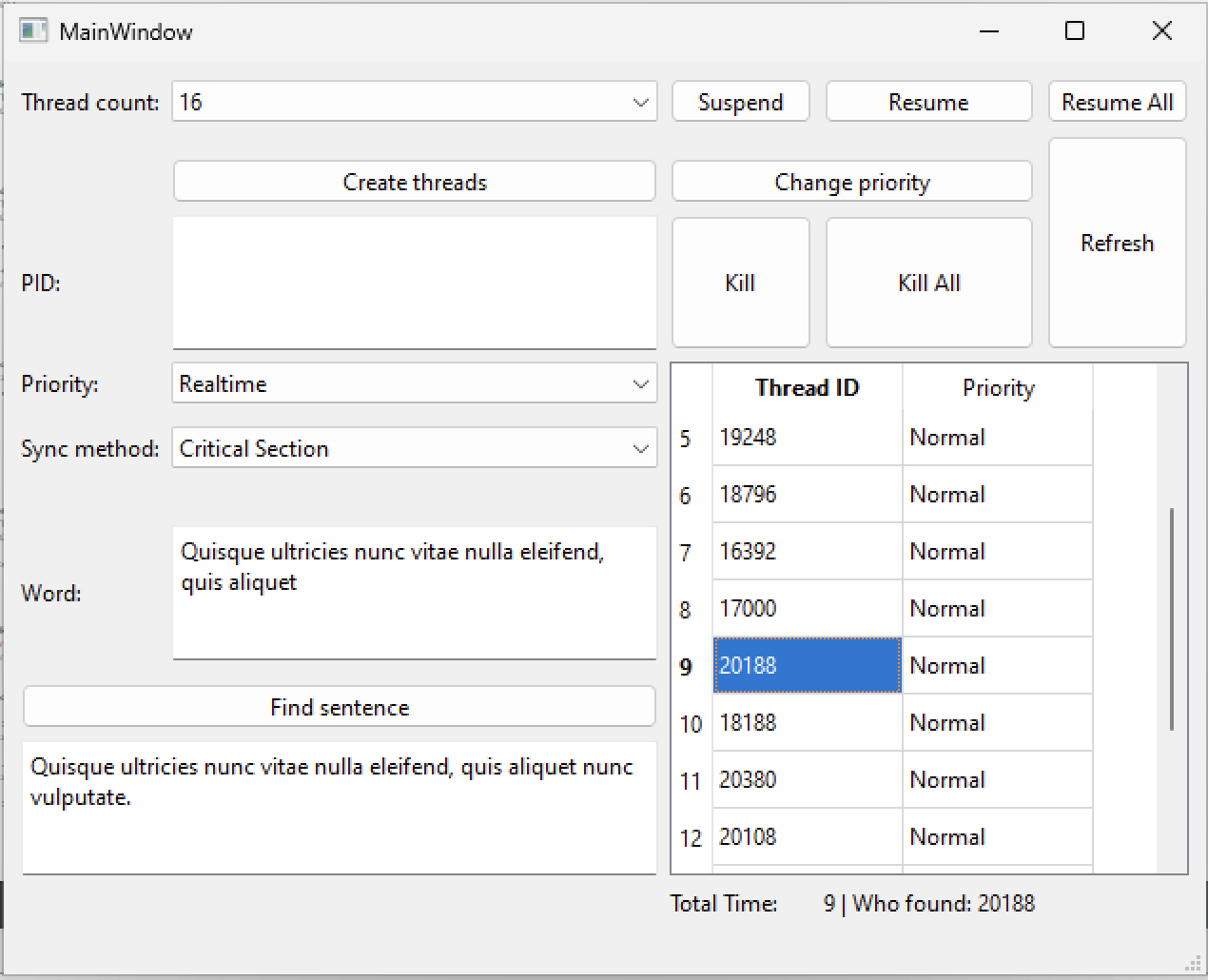
Отримуємо шукане речення, а також час (5мс) та ID потоку, що знайшов речення.

Спробуємо ускладнити завдання та знайти частинку речення, що знаходиться в середині та в кінці файлу: “Quisque ultricies nunc vitae nulla eleifend, quis aliquet” Використаємо також Mutex, але цього разу використаємо 16 потоків:



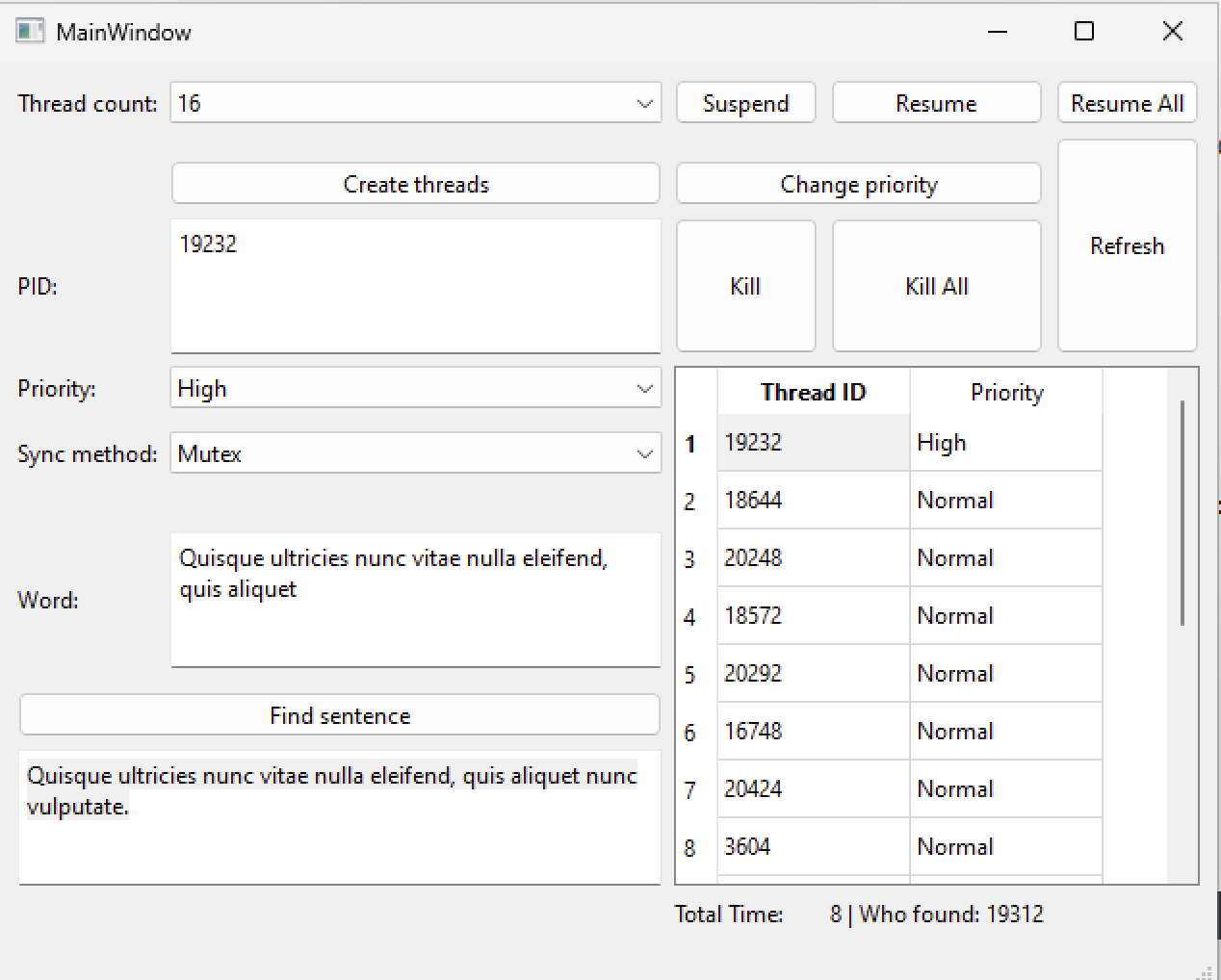
Бачимо, що речення знайшов потік №11 (15432). Оскільки ми запускаємо потоки в циклі послідовно, то це демонструє, що метод синхронізації Mutex спрацював.

Спробуємо повторити це із синхронізацією “Критична секція”:

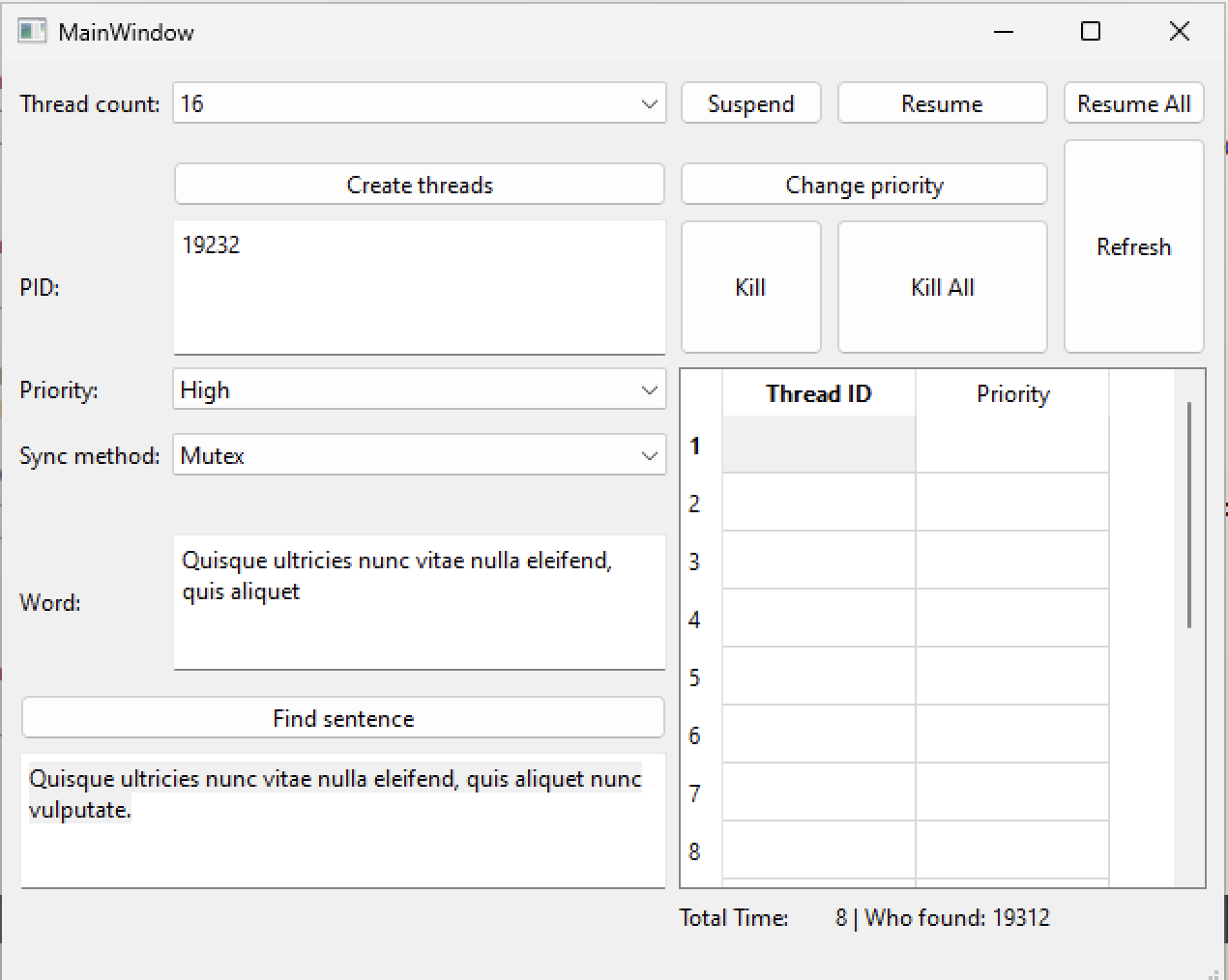


Бачимо приблизно однаковий результат, проте Мютекс виконався швидше. Для великих обʼємів роботи результат є, зазвичай, протилежним, оскільки Мютекс є системним обʼєктом, що призупиняє вхідні потоки, якщо він заблокований. Це вимагає відправку сигналу до ОС, що є ресурсозатратним завданням.

Спробуємо змінити пріоритет довільного потоки на високий:



Вбʼємо усі потоки



**Висновок**

У цій лабораторній роботі я навчився працювати із потоками та методами їхньої синхронізації, використовуючи WinAPI. Я ознайомився із мютексом та критичною секцією, зрозумів, який метод швидший та чому. Я розбив завдання на окремі частинки та використав синхронізацію, щоб отримати потрібний результат.