**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ЛЬВІВСЬКА ПОЛІТЕХНІКА»**

Інститут **ІКНІ**

Кафедра **ПЗ**

# ЗВІТ

До лабораторної роботи №5

На тему: «Багатопоточність в операційній сисемі WINDOWS. Створення, керування та синхронізація потоків.»

З дисципліни: «Операційні системи»

**Лектор** : ст.викл каф.ПЗ

Грицай О.Д.

**Виконала:** ст.гр.ПЗ-23

Кохман О.В.

**Прийняла:** ст.викл каф.ПЗ

Грицай О.Д.

«____»_____2022 р.

∑____ _ _ _ _ _ _ _ _.

Львів – 2022

**Тема:** Багатопоточність в операційній сисемі WINDOWS. Створення, керування та синхронізація потоків.

**Мета:** Ознайомитися 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). 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.

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.

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.

М’ютекс (mutex, від mutualexclusion – взаємне виключення) – об’єкт взаємного виключення, призначений для захисту певного об’єкта у потоці від доступу інших потоків. М’ютекси є одним із варіантів семафорних механізмів для організації взаємного виключення. Їхнє основне призначення – організація взаємного виключення для потоків з одного або різних процесів. М’ютекси – це прості двійкові семафори, які можуть перебувати в одному з двох станів – сигнальному або несигнальному (відкритий і закритий). Коли потік стає власником м’ютекса, він переводиться у несигнальний стан. Критична секція у значній мірі виконує ті ж завдання, що і м'ютекс. У операційних системах Microsoft Windows ГОЛОВНА ВІДМІННІСТЬ МІЖ М'ЮТЕКСОМ І КРИТИЧНОЮ СЕКЦІЄЮ в тому, що м'ютекс є об'єктом ядра і може бути використаний кількома процесами одночасно, критична секція ж належить процесу і служить для синхронізації тільки його потоків.

Функції очікування дозволяють потоку блокувати власне виконання. Функції очікування не повертаються, поки не будуть виконані вказані критерії. Тип функції очікування визначає набір використаних критеріїв. Коли викликається функція очікування, вона перевіряє, чи дотримані

критерії очікування. Якщо критерії не були виконані, викличний потік переходить у стан очікування, поки не будуть виконані умови критеріїв очікування або не закінчиться вказаний інтервал очікування. Існує чотири типи функцій очікування: 1. Single-object 2. multiple-object 3. Alertable 4. Registered.

Події є об'єктом синхронізації ядра. Події можуть сигналізувати іншим потокам, вказуючи на те, що зараз виконується якась умова, наприклад доступне повідомлення. Важлива додаткова можливість, яку пропонують події, полягає в тому, що кілька потоків можуть бути звільнені від очікування одночасно, коли сигналізується одна подія.

Події класифікуються як події ручного скидання та події автоматичного скидання, і ця властивість події встановлюється викликом CreateEvent. • Подія скидання вручну може сигналізувати декільком потоків, які очікують на подію одночасно, і це очікування може бути скинуто. • Подія автоматичного скидання сигналізує один потік, який чекає на подію, і скидається автоматично. Події використовують наступні функції: CreateEvent, CreateEventEx, OpenEvent, SetEvent, ResetEvent, PulseEvent

**Індивідуальне завдання**

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.

**Варіант 9:**

9. Вивід слів з файлу, що розпочинаються на задану літеру (кількість рядків у файлі > 1000, текст довільна наукова стаття). Синхронізація: 4, 9.

## Код програми

Назва файлу: main.cpp

```
#include <Windows.h>
#include <iostream>
#include <fstream>
#include <chrono>
void criticalJob();
void asyncJob(bool access);
void timerJob();
using namespace std;
using namespace std::chrono;
CRITICAL_SECTION cs;
HANDLE hTimer;
DWORD dWaitForSingleObject;
int countThread = 8;
void search(string* array, int start, int end, char value) {
	for (int i = start; i < end; i++) {
		if (array[i][0] == value || array[i][0] == toupper(value) ||
array[i][0] == tolower(value)) {
			cout << array[i] << " ";
		}
	}
	cout << "\n";
}
DWORD WINAPI mythread(__in LPVOID lpParameter) {
	fstream file;
	string word, filename;
	filename = "myfile.txt";
	file.open(filename.c_str());
	int size = 0;
	while (file >> word) {
		size++;
	}
	file.close();
	string* array = new string[size];
	file.open(filename.c_str());
	int i = 0;
	while (file >> word) {
		array[i] = word;
		i++;
	}
	search(array, 0, i, 'z');
	ExitThread(0);
}
DWORD WINAPI mythread2(__in LPVOID lpParameter) {
	EnterCriticalSection(&cs);
	fstream file;
	string word, filename;
	filename = "myfile.txt";
	file.open(filename.c_str());
	int size = 0;
	while (file >> word) {
		size++;
	}
```

```
	file.close();
	string* array = new string[size];
	file.open(filename.c_str());
	int i = 0;
	while (file >> word) {
		array[i] = word;
		i++;
	}
	search(array, 0, i, 'z');
	LeaveCriticalSection(&cs);
	ExitThread(0);
}

int main(int argc, char* argv[]) {
	int choice = 0;
	while (true) {
		cout << "Enter number:\n[1] - ASYNCHRONIZATION\n[2] - CRITICAL
SECTION\n[3] - WAITABLE TIMER\n[4] - MEASURE TIME\n[5] - EXIT\n";
		cin >> choice;
		if (choice == 1) {
			cout << "\nEnter number of threads to be created:" << endl;
			cin >> countThread;
			asyncJob(false);
		}
		else if (choice == 2) {
			cout << "\nEnter number of threads to be created:" << endl;
			cin >> countThread;
			criticalJob();
		}
		else if (choice == 3) {
			cout << "\nEnter number of threads to be created:" << endl;
			cin >> countThread;
			timerJob();
		}
		else if (choice == 4) {
			int countThreadArray[] = {1,2,4,8,16};
			for (int i = 0; i < 5; i++) {
				countThread = countThreadArray[i];
				cout <<
"_____________________________________________________________________________
_____________________________________" << endl;
				cout << "\t\t\t\t\t\tNUMBER OF THREADS - " <<
countThread << endl;
				auto start = high_resolution_clock::now();
				asyncJob(false);
				auto stop = high_resolution_clock::now();
				auto duration = duration_cast<microseconds>(stop -
start);
				cout << "Time taken by ASYNCHRONIZATION: " <<
duration.count() << " microseconds" << endl;
				auto start2 = high_resolution_clock::now();
				criticalJob();
				auto stop2 = high_resolution_clock::now();
				auto duration2 = duration_cast<microseconds>(stop2 -
start2);
				cout << "Time taken by CRITICAL SECTION: " <<
duration2.count() << " microseconds" << endl;
				auto start3 = high_resolution_clock::now();
				timerJob();
				auto stop3 = high_resolution_clock::now();
				auto duration3 = duration_cast<microseconds>(stop3 -
start3);
```

```
                        cout << "Time taken by WAITABLE TIMER: " <<
duration3.count() << " microseconds" << endl;
                        cout <<
"-------------------------------------------------------------------------------
---------------------------------------" << endl;
                  }
            }
            else if (choice == 5) {
                  break;
            }
      }
      return 0;
}
void asyncJob(bool access) {
      HANDLE* myHandle;
      DWORD* threadID;
      myHandle = new HANDLE[countThread];
      threadID = new DWORD[countThread];
      for (int i = 0; i < countThread; i++) {
            myHandle[i] = CreateThread(NULL, 0, mythread, 0, CREATE_SUSPENDED,
&threadID[i]);
      }

      if (myHandle == NULL) {
            return;
      }
      else {
            for (int i = 0; i < countThread; i++) {
                  cout << i << " tid - " << threadID[i] << endl;
            }
      }
      if (access) {
            while (true) {
                  int number = 0;
                  cout << "Enter number:\n[1] - RESUME\n[2] - SUSPEND\n[3] -
CHANGE PRIORITY\n[4] - KILL\n[5] - EXIT\n";
                  cin >> number;
                  if (number == 1) {
                        int threadNumber = 0;
                        cout << "enter number of thread to resume" << endl;
                        cin >> threadNumber;
                        ResumeThread(myHandle[threadNumber]);
                        WaitForSingleObject(myHandle[threadNumber], INFINITE);
                        cout << "thread was resumed - " <<
threadID[threadNumber] << endl;
                  }
                  else if (number == 2) {
                        int threadNumber = 0;
                        cout << "enter number of thread to suspend" << endl;
                        cin >> threadNumber;
                        SuspendThread(myHandle[threadNumber]);
                        cout << "thread was suspended - " <<
threadID[threadNumber] << endl;
                  }
                  else if (number == 3) {
                        int threadNumber = 0;
                        cout << "enter number of thread to change priority" <<
endl;
                        cin >> threadNumber;
                        int newPriority = 0;
                        cout << "old priority - " <<
GetThreadPriority(myHandle[threadNumber]) << " tid - " << threadID[threadNumber]
<< endl;
```

```
                        cout << "enter new priority:\n[0] -
THREAD_PRIORITY_IDLE" <<
                                "\n[1] - THREAD_PRIORITY_LOWEST" <<
                                "\n[2] - THREAD_PRIORITY_BELOW_NORMAL" <<
                                "\n[3] - THREAD_PRIORITY_NORMAL" <<
                                "\n[4] - THREAD_PRIORITY_ABOVE_NORMAL" <<
                                "\n[5] - THREAD_PRIORITY_HIGHEST" <<
                                "\n[6] - THREAD_PRIORITY_TIME_CRITICAL" << endl;
                        cin >> newPriority;
                        int priorityArray[] = { THREAD_PRIORITY_IDLE,

      THREAD_PRIORITY_LOWEST,

      THREAD_PRIORITY_BELOW_NORMAL,

      THREAD_PRIORITY_NORMAL,

      THREAD_PRIORITY_ABOVE_NORMAL,

      THREAD_PRIORITY_HIGHEST,

      THREAD_PRIORITY_TIME_CRITICAL };

                        if (SetThreadPriority(myHandle[threadNumber],
priorityArray[newPriority]) != 0) {
                                cout << "new priority - " <<
GetThreadPriority(myHandle[threadNumber]) << " tid - " << threadID[threadNumber]
<< endl;
                        }
                  }
                  else if (number == 4) {
                        int threadNumber = 0;
                        cout << "enter number of thread to kill" << endl;
                        cin >> threadNumber;
                        TerminateThread(myHandle[threadNumber], 0);
                        cout << "thread was killed - " <<
threadID[threadNumber] << endl;
                  }
                  else if (number == 5) {
                        break;
                  }
            }
      }
      else {
            for (int i = 0; i < countThread; i++) {
                  ResumeThread(myHandle[i]);
            }
            for (int i = 0; i < countThread; i++) {
                  WaitForSingleObject(myHandle[i], INFINITE);
                  CloseHandle(myHandle[i]);
            }
      }
}

void criticalJob() {
      HANDLE* myHandle;
      DWORD* threadID;
      myHandle = new HANDLE[countThread];
      threadID = new DWORD[countThread];
      InitializeCriticalSection(&cs);
      for (int i = 0; i < countThread; i++) {
            myHandle[i] = CreateThread(NULL, 0, mythread2, 0, CREATE_SUSPENDED,
&threadID[i]);
```

```
	}
	if (myHandle == NULL) {
		return;
	}
	else {
		for (int i = 0; i < countThread; i++) {
			cout << "tid - " << threadID[i] << endl;
		}
	}
	for (int i = 0; i < countThread; i++) {
		ResumeThread(myHandle[i]);
	}
	for (int i = 0; i < countThread; i++) {
		WaitForSingleObject(myHandle[i], INFINITE);
		CloseHandle(myHandle[i]);
	}
	DeleteCriticalSection(&cs);
}

void timerJob() {
	BOOL bSetWaitableTimer = FALSE;
	LARGE_INTEGER liDueTime;
	liDueTime.QuadPart = -00000000LL;
	hTimer = CreateWaitableTimer(NULL, TRUE, NULL);
	if (hTimer == NULL) {
		return;
	}
	HANDLE* thread;
	DWORD* tid;
	thread = new HANDLE[countThread];
	tid = new DWORD[countThread];
	for (int i = 0; i < countThread; i++) {
		thread[i] = CreateThread(NULL, 0, mythread, 0, CREATE_SUSPENDED,
&tid[i]);
	}
	for (int i = 0; i < countThread; i++) {
		cout << "tid - " << tid[i] << endl;
	}
	bSetWaitableTimer = SetWaitableTimer(hTimer, &liDueTime, 0, NULL, NULL,
0);
	if (bSetWaitableTimer == FALSE) {
		return;
	}
	dWaitForSingleObject = WaitForSingleObject(hTimer, INFINITE);
	if (dWaitForSingleObject == WAIT_FAILED) {
		return;
	}
	for (int i = 0; i < countThread; i++) {
		ResumeThread(thread[i]);
		WaitForSingleObject(thread[i], INFINITE);

	}

	CloseHandle(hTimer);
	for (int i = 0; i < countThread; i++) {
		CloseHandle(thread[i]);
	}
}
```

## Протокол роботи

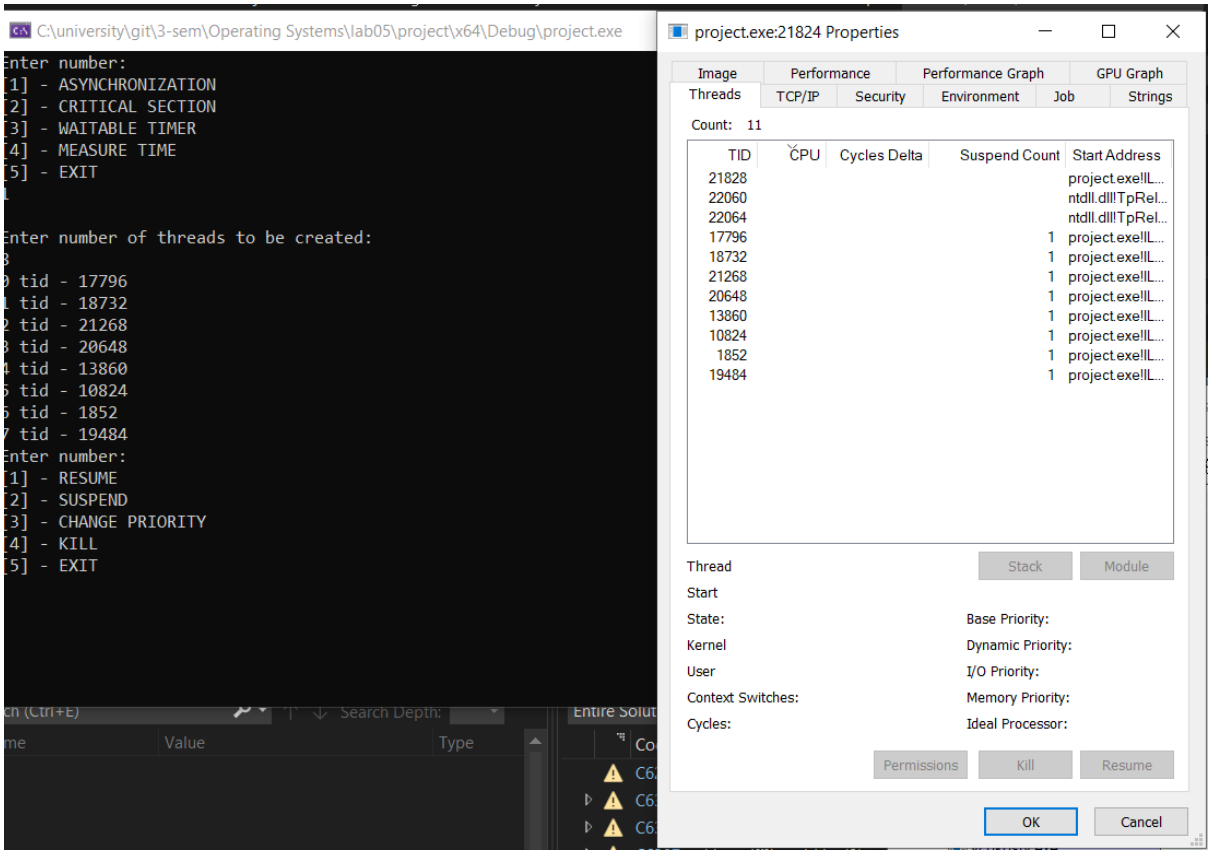


Рис. 1 Створення призупинених потоків і підтвердження у Process Explorer.

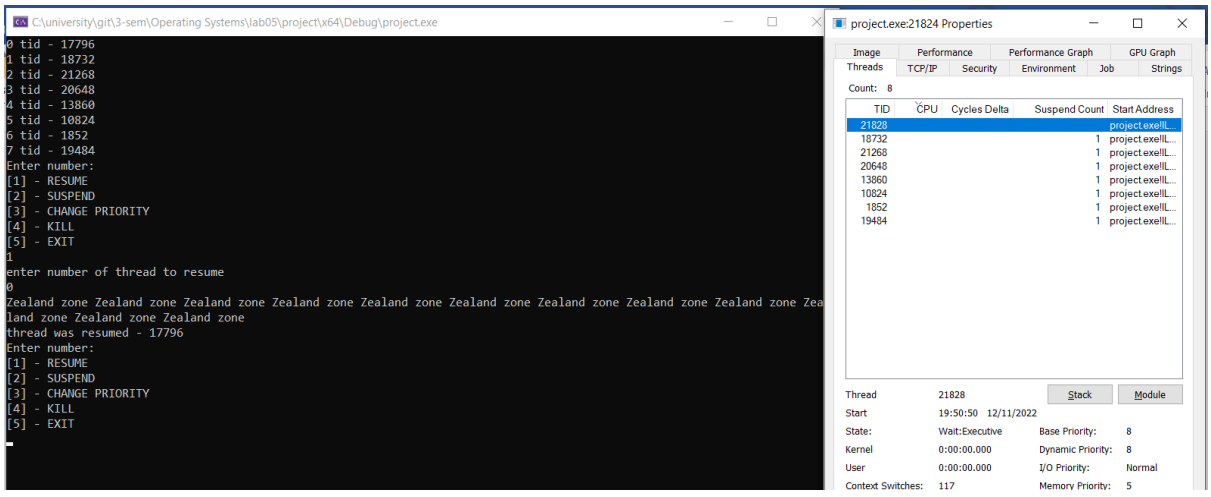


Рис. 2 Відновлення потоку і підтвердження у Process Explorer, що потік відновився і завершив свою роботу.

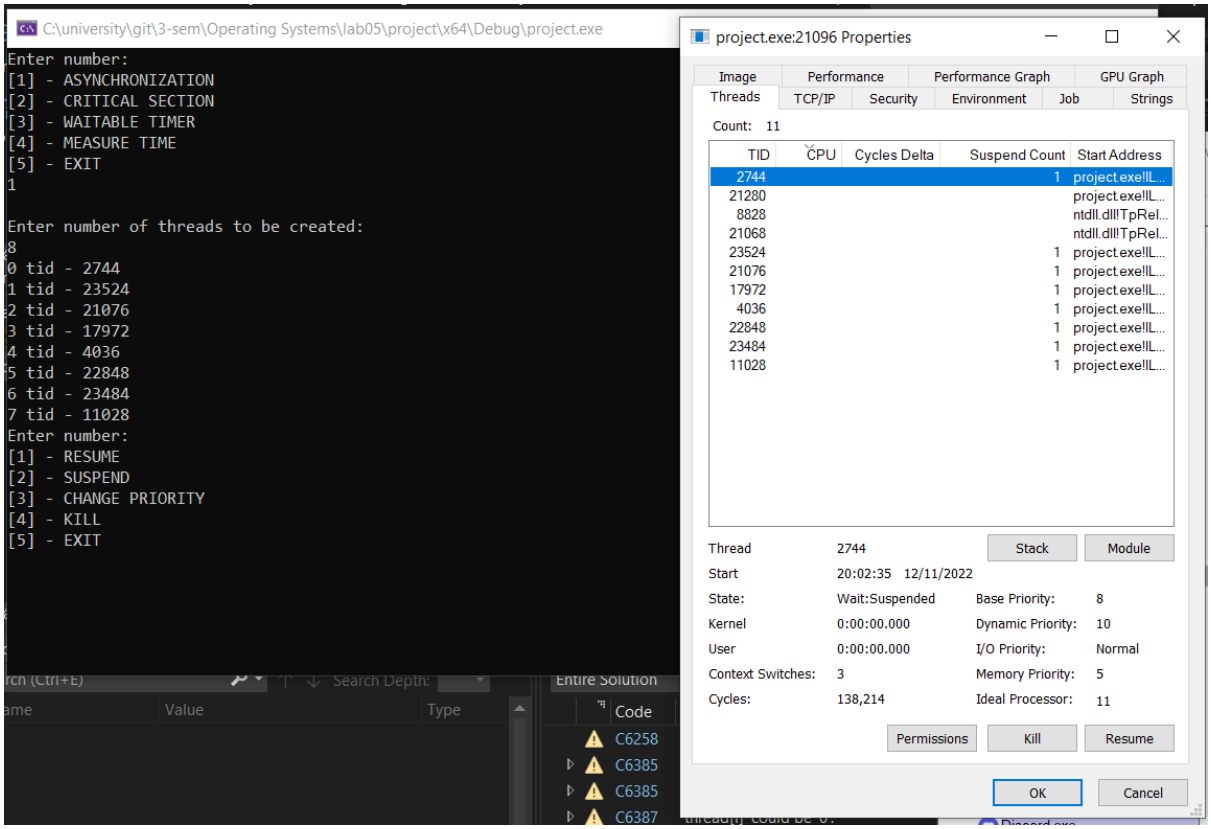


Рис. 3 Пріоритет потоку 2744 до зміни пріоритету.

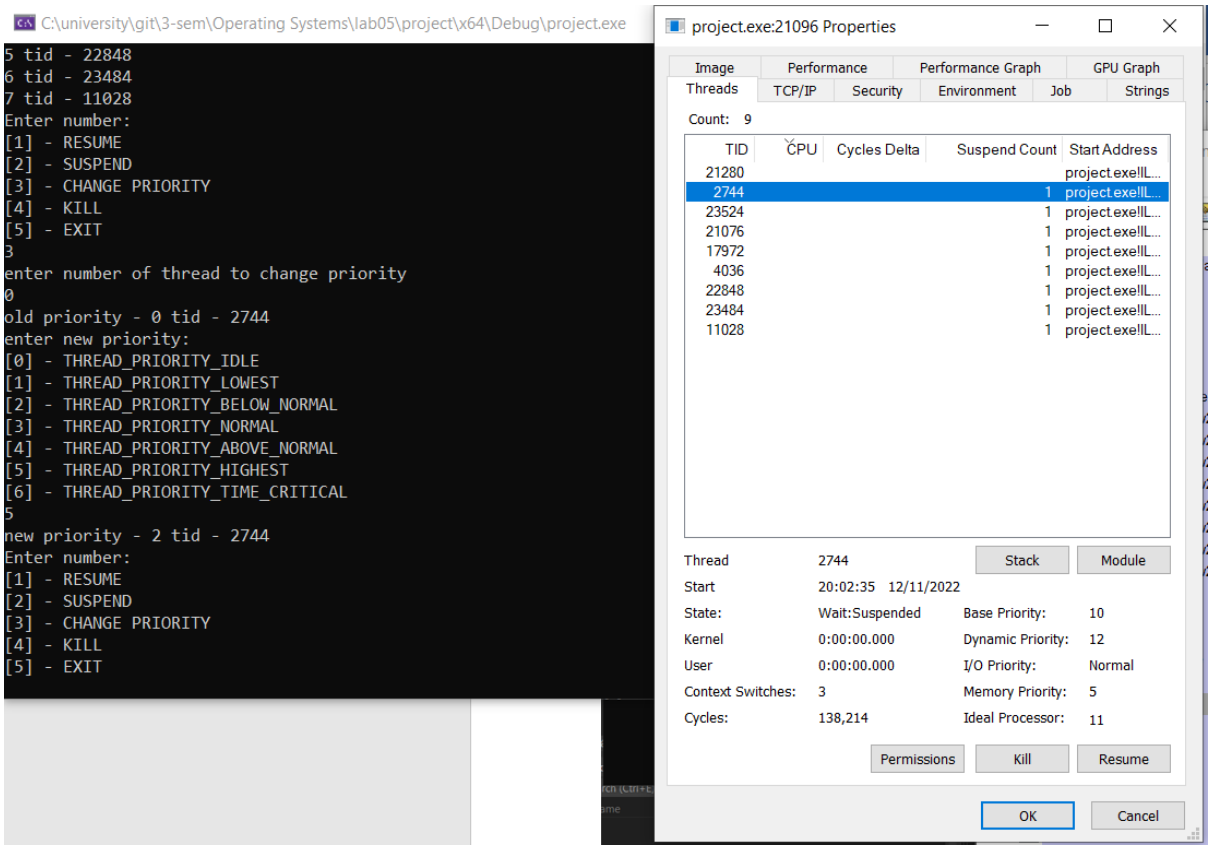


Рис. 4 Пріоритет потоку 2744 після зміни пріоритету.

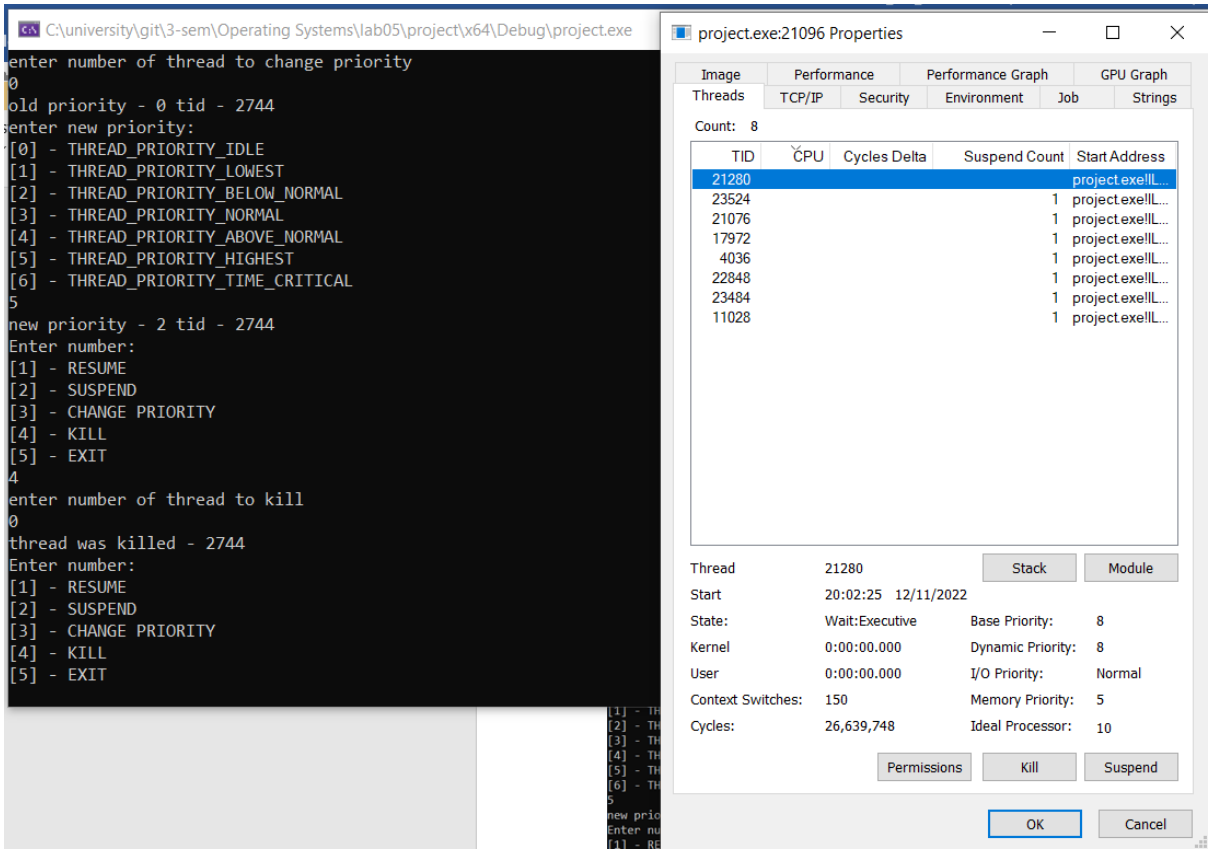


Рис. 5 Вбиття потоку.

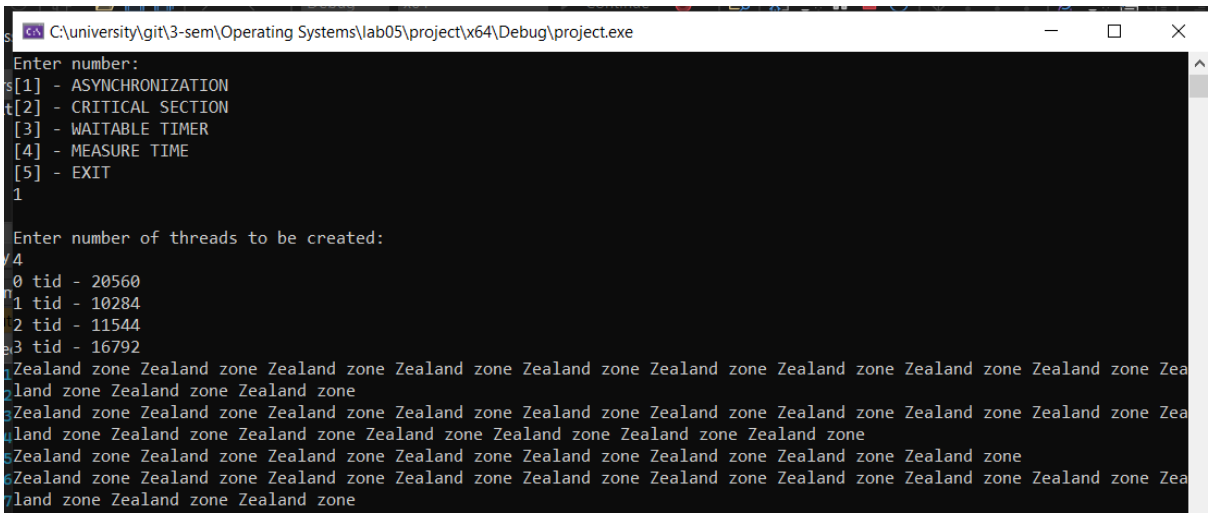


Рис. 6 Результат виконання  програми при асинхронізації.

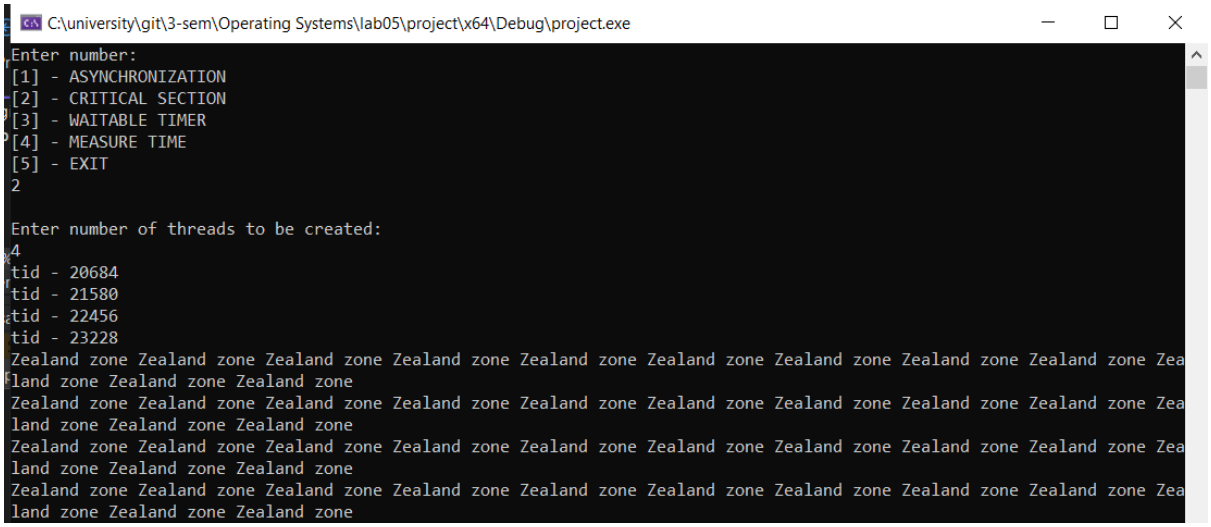


Рис. 7 Результат виконання програми при синхронізації за допомогою критичної секції.


Рис. 8 Результат виконання програми при синхронізації за допомогою таймера очікування.

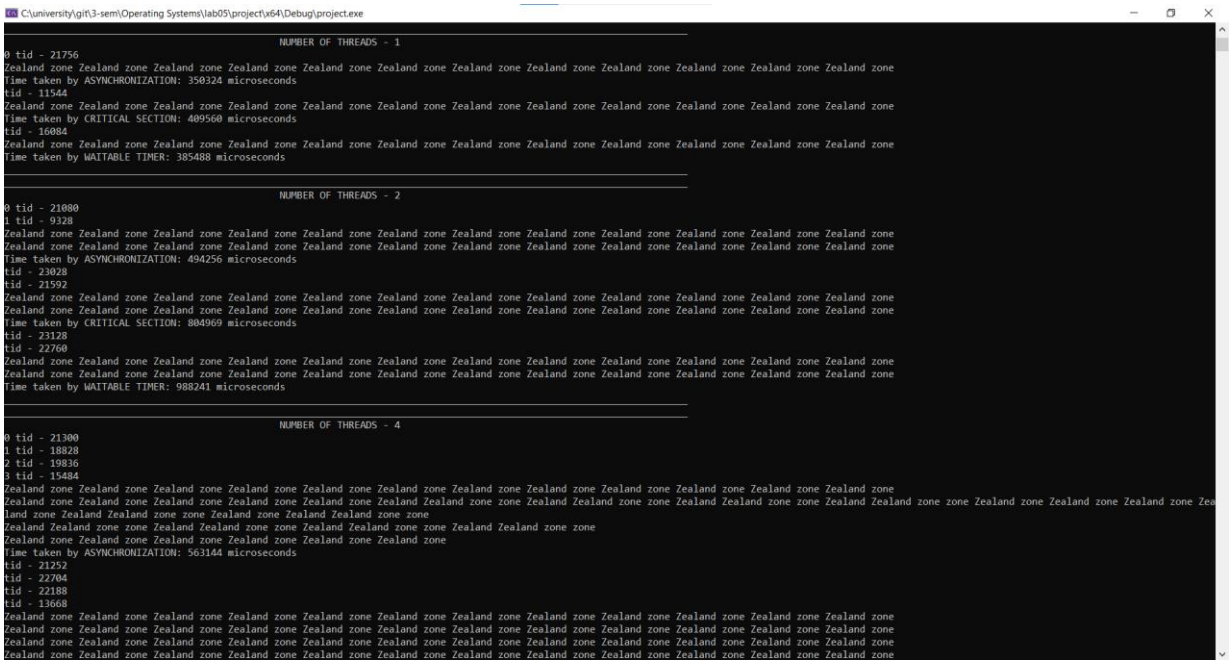


Рис. 9 Вимірювання часу виконання при різній кількості потоків та синхронізації, асинхронізації за допомогою критичної секції та асинхронізації за допомогою таймера очікування.

| Thread number | Asynchronization | Critical Section | Waitable Timer |
|---|---|---|---|
| 1 | 350324 | 409560 | 385488 |
| 2 | 494256 | 804969 | 988241 |
| 4 | 563144 | 1889351 | 1742548 |
| 8 | 1295051 | 2734277 | 2887695 |
| 16 | 2779272 | 5095922 | 5475117 |

Таблиця 1 Кількість потоків та час виконання при асинхронізації, синхронізації за допомогою критичної секції та синхронізації за допомогою таймера очікування. З таблиці можна зробити висновок, що при синхронізації за допомогою критичної секції та синхронізації таймером очікування час виконання програми збільшується майже вдвічі, якщо порівнювати із асинхронізацією.

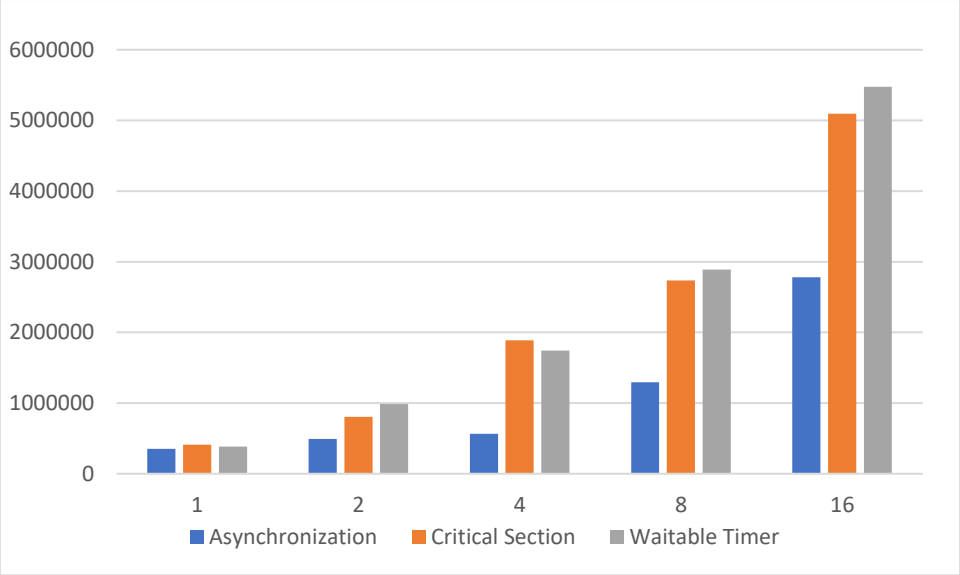


Рис. 10 Порівняння часу виконання різної кількості потоків при при асинхронізації, синхронізації за допомогою критичної секції та синхронізації за допомогою таймера очікування.

## Висновок

На цій лабораторній роботі я дізналась, що таке потік та багатопоточність, також реалізувала програму, яка створює потоки, може призупиняти, відновлювати, вбивати потік , змінювати пріоритет потоку, а також синхронізувала програму за допомогою критичної секції та за допомогою таймера очікування. Виміряла час виконання програми при різній кількості потоків і різній синхронізації та порівняла результати.