МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет "Запорізька політехніка"

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт з дисципліни
"Системне програмування"

для студентів спеціальності 123 "Комп'ютерна інженерія"
всіх форм навчання

Частина 1

2024

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмування" для студентів спеціальності 123 "Комп'ютерна інженерія" всіх форм навчання. Частина 1. / Укл. А.В. Тіменко, М.М. Хохлов – Запоріжжя: НУ "Запорізька політехніка", 2024. – 33 с.

Укладачі: А.В.Тіменко, ст.викл,
М.М. Хохлов, ст..викл.

Рецензент: С.Ю. Скрупський, к.т.н., доцент

Відповідальний за випуск: М.М. Хохлов, ст. викладач

Затверджено:
на засіданні кафедри
"Комп’ютерні системи та мережі"
Протокол № _7_ від 22 березня 2024

Рекомендовано до видання НМК ФКНТ
Протокол № _8_ від 27 березня 2024

ЗМІСТ

Вступ 4
1 Лабораторна робота № 1. Потоки вводу-виводу 5
2 Лабораторна робота № 2. Запуск процесів і потоків (Mutex/ Futex, Semaphore, Events, Timers, CritialSections) 12
Література 30
Додаток А 31

ВСТУП

Лабораторні роботи спрямовані на засвоєння базових знань з системного програмування при використанні системних викликів ОС Windows (WinAPI). Вивчаються основні концепції роботи з системними об'єктами. Вивчається створення програм з використанням мови програмування C/C++.

У лабораторних роботах розглядається створення програм для реалізації задач взаємодії з операційною системою на системному рівні, доступу до файлової системи, роботи багатопоточних програм та їх синхронізація, організація обміну даними між процесами; вивчаються питання створення об'єктів синхронізації, створення потоків, засобів передавання даних між процесами та комп'ютерними системами в мережі.

Відповідно до графіку, студенти перед виконанням лабораторної роботи повинні ознайомитися з конспектом лекцій та рекомендованою літературою.

Для одержання заліку з кожної роботи студент здає викладачу оформлений звіт, а також демонструє на екрані комп'ютера результати виконання лабораторної роботи.

Звіт має містити:

- титульний аркуш;
- тему та мету роботи; завдання до роботи;
- лаконічний опис теоретичних відомостей;
- результати виконання лабораторної роботи;
- змістовний аналіз отриманих результатів, висновки та відповіді до контрольних запитань.

Звіт виконують на білому папері формату А4 (210х297 мм) або подають в електронному вигляді. Під час співбесіди при захисті лабораторної роботи студент повинний виявити знання про зміст роботи та методи виконання кожного етапу роботи, а також вміти продемонструвати результати роботи на конкретних прикладах та правильно аналізувати отримані результати.

1 ЛАБОРАТОРНА РОБОТА № 1
ПОТОКИ ВВОДУ-ВИВОДУ

Мета роботи: навчитися користуватися потоками вводу-виведення. Операторами перенаправлення потоків. Навчитися оперувати даними в потоках вводу-виведення.

1.1 Теоретичні відомості

Консольний ввід/вивід є невід'ємною частиною будь-якої ОС. Усі стандартні потоки мають фіксовані номери. Наприклад потік введення з клавіатури має номер 0, потік виведення до консолі - номер 1, а потік виведення помилок - 2. Практично кожна ОС дозволяє виконувати перенаправлення як потоку введення даних, так і потоку виведення за допомогою допоміжних операторів командного рядку:

| (символ "або") - дозволяє спрямувати стандартній потік виводу лівого операнду в якості стандартного потоку введення для правого операнду. Наприклад,

set | **findstr** OS

> - дозволяє спрямувати стандартній потік виводу лівого операнду в файл чи інший потік, ім'я (або номер) якого вказано за допомогою правого операнду. Наприклад,

set > set.out

type some_absent_file_name 2> nul

Функції стандартної бібліотеки С та системні виклики, що використовуються для роботи з потоками та консоллю.

Отримати дані введені з клавіатури (консолі):

Бібліотека C/C++: **scanf**, **getc**, **gets**, **fread**, **cin** >>

Системний виклик:

```
BOOL WINAPI ReadConsole(
    HANDLE      hConsoleInput,
    LPVOID      lpBuffer,
```

```
    DWORD       nNumberOfCharsToRead,
    LPDWORD     lpNumberOfCharsRead,
    LPVOID      pInputControl );
```

Вивести дані до екрану (консолі):
Бібліотека C/C++: **printf**, **putc**, **puts**, **fwrite**, **cout** <<
Системний виклик:

```
BOOL WINAPI WriteConsole(
    HANDLE      hConsoleOutput,
    LPVOID      lpBuffer,
    DWORD       nNumberOfCharsToWrite,
    LPDWORD     lpNumberOfCharsWritten,
    LPVOID      lpReserved );
```

* **Примітка:** системні виклики **ReadConsole**/**WriteConsole** не можуть працювати з потоками введення/виведення, якщо в якості hConsoleInput/hConsoleOutput використовуються перенаправлення дескрипторів. В таких випадках слід використовувати системні виклики **ReadFile** та **WriteFile**.

```
HANDLE WINAPI GetStdHandle(DWORD nStdHandle);

BOOL WINAPI GetConsoleMode(
    HANDLE      hConsoleHandle,
    LPDWORD     lpMode );

BOOL WINAPI SetConsoleMode(
    HANDLE      hConsoleHandle,
    DWORD       dwMode );

BOOL WINAPI SetConsoleTitle(LPCTSTR lpConsoleTitle);

BOOL WINAPI SetConsoleTextAttribute(
```

HANDLE hConsoleOutput,
WORD wAttributes);

hConsoleOutput – дескриптор буферу консолі виводу;

wAttributes – колір літер та фону, який отримано шляхом комбінування констант. Наприклад, FOREGROUND_BLUE, FOREGROUND_RED, FOREGROUND_INTENSITY.

BOOL WINAPI **FillConsoleOutputAttribute**(
HANDLE hConsoleOutput,
WORD wAttribute,
DWORD nLength,
COORD dwWriteCoord,
LPDWORD lpNumberOfAttrsWritten);

hConsoleOutput – дескриптор буферу консолі виводу;

wAttribute – атрибут кольору фону символа в консолі;

nLength – кількість комірок символів, фон яких встановлюється заданим кольором;

dwWriteCoord – координати першої комірки;

lpNumberOfAttrsWritten – покажчик на ідентифікатор, в який буде записано кількість реально зафарбованих комірок.

void WINAPI **OutputDebugString**(
LPCTSTR lpOutputString);

Програма 1.1 – lab1.cpp

```
#include <windows.h>
#include <tchar.h>

#define BUFFER_SIZE 1024
```

```
	BOOL Read(BOOL isConsole, HANDLE hInput, LPVOID
lpBuffer, DWORD nNumberOfCharsToRead, LPDWORD
lpNumberOfCharsRead) {
		if (isConsole)
		{
		   return ReadConsole(hInput, lpBuffer,
nNumberOfCharsToRead, lpNumberOfCharsRead, NULL);
		}
		return ReadFile(hInput, lpBuffer, nNumberOfCharsToRead,
lpNumberOfCharsRead, NULL);
	}

	BOOL Write(BOOL isConsole, HANDLE hInput, LPVOID
lpBuffer, DWORD nNumberOfCharsToRead, LPDWORD
lpNumberOfCharsRead) {
		if (isConsole)
		{
		   return WriteConsole(hInput, lpBuffer,
nNumberOfCharsToRead, lpNumberOfCharsRead, NULL);
		}
		return WriteFile(hInput, lpBuffer, nNumberOfCharsToRead,
lpNumberOfCharsRead, NULL);
	}

	int _tmain(int argc, TCHAR* argv[]) {

		HANDLE hStdIn =
GetStdHandle(STD_INPUT_HANDLE);
		HANDLE hStdOut =
GetStdHandle(STD_OUTPUT_HANDLE);
		TCHAR buffer[BUFFER_SIZE];
		DWORD dwCount = 0;
		DWORD ignored;
```

```
        if ( (hStdIn == INVALID_HANDLE_VALUE) || (hStdOut
== INVALID_HANDLE_VALUE) )
        {
           ExitProcess(1);
        }

        BOOL isConsoleIn = GetConsoleMode(hStdIn, &ignored);
        BOOL isConsoleOut = GetConsoleMode(hStdOut,
&ignored);

        while(Read(isConsoleIn, hStdIn, &buffer, BUFFER_SIZE,
&dwCount) || dwCount > 0)
        {
           Write(isConsoleOut, hStdOut, &buffer, dwCount,
&dwCount);
        }

        return 0;
    }
```

Запустіть програму. Програма очікує введення з клавіатури рядків символів. Після введення Enter програма має повторити введені символи в наступному рядку й перейти в початковий стан очікування введення наступного рядку. Припинення роботи програми можна здійснити за допомогою натискання Ctrl+C.

За допомогою операторів перенаправлення потоків вводу-виведення можна спрямувати виведення програми у файл. Наприклад, виконайте програму в такий спосіб:

lab1.exe > lab1.out

Відмінністю такого запуску буде виведення до файлу усіх введених рядків символів.

Виконайте отриману програму за допомогою командного рядку в такий спосіб:

type lab1.cpp | lab1.exe > lab1.out

В результаті буде отримано файл lab1.out який має бути точною копією файлу lab1.cpp.

1.2 Завдання для виконання

1. Додайте обробку параметрів командного рядка:

Варіант 1:

-i filename – програма має використовувати в якості вхідного потоку дані з файлу filename

Варіант 2:

-o filename – програма має використовувати в якості вихідного потоку файл filename

2. За допомогою системного виклику SetConsoleTitle змінити назву вікна консолі на повний вміст командного рядку (рядок отриманий з усіх argv[] рядків)

3. Додати обробку вхідного потоку даних:

Варіант 1: вилучити з вихідного потоку усі великі літери та цифри.

Варіант 2: змінити регістр усіх літер вихідного потоку на великі літери.

1.3 Контрольні питання

1. За допомогою, яких операторів можливо перенаправлення потоків введення-виведення?
2. Навести фіксовані номери потоків даних.
3. Які функції і системні виклики використовуються для роботи з потоками і консоллю?
4. У яких випадках варто використовувати ReadFile і WriteFile?
5. Методи перенаправлення виведення програми в файл.

6. Як отримати дані введені з клавіатури? (навести приклад)

7. Як вивести дані на екран? (навести приклад)

8. Привести приклад коду, за допомогою якого, можна змінити назву вікна консолі.

9. Пояснити призначення та навести приклад аргументів:

```
BOOL WINAPI FillConsoleOutputAttribute(
    HANDLE hConsoleOutput,
    WORD wAttribute,
    DWORD nLength,
    COORD dwWriteCoord,
    LPDWORD lpNumberOfAttrsWritten);
```

2 ЛАБОРАТОРНА РОБОТА № 2. ЗАПУСК ПРОЦЕСІВ І ПОТОКІВ (MUTEX/FUTEX, SEMAPHORE, EVENTS, TIMERS, CRITIALSECTIONS)

Мета роботи: Навчитися створювати і синхронізувати процеси і потоки команд. Навчитися правильному використанню потоків за допомогою об'єктів синхронізації.

2.1 Теоретичні відомості

Процес являє собою об'єкт, що володіє власним незалежним віртуальним адресним простором, в якому можуть розміщуватися код і дані, захищені від інших процесів. У свою чергу, усередині кожного процесу можуть незалежно виконуватися один або кілька потоків команд (threads). Первинний потік команд, що виконується всередині процесу, може сам собі створювати нові потоки команд і нові незалежні процеси, а також керувати взаємодією об'єктів між собою і їх синхронізацією.

Створення процесів. Створення Win32 процесу здійснюється викликом однієї з таких функцій, як **CreateProcess**, **CreateProcessAsUser** (для WinNT/2000) і **CreateProcessWithLogonW** (починаючи з Win2000).

```
BOOL CreateProcess (
    LPCTSTR lpApplicationName, // назва додатку
    LPTSTR lpCommandLine, // командний рядок
    // Покажчик на структуру SECURITY_ATTRIBUTES
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    // Покажчик на структуру SECURITY_ATTRIBUTES
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // Ознака успадкування дискрипторів
    BOOL bInheritHandles,
    // Ознаки способів створення процесу
```

```
        DWORD dwCreationFlags,
        // Покажчик на блок середовища
        LPVOID lpEnvironment,
        LPCTSTR lpCurrentDirectory, // Поточний каталог
        // Покажчик нас структуру STARTUPINFO
        LPSTARTUPINFO lpStartupInfo,
        // Покажчик нас структуру
PROCESS_INFORMATION
        LPPROCESS_INFORMATION lpProcessInformation);
```

Значення, що повертається в разі успішного створення процесу і потоку - TRUE, інакше - FALSE.

Завершення і припинення виконання процесу. Після того як процес завершив свою роботу, потік команд, що виконується в цьому процесі, може викликати функцію ExitProcess, вказавши в якості параметра кодом завершення (exit code):

VOID **ExitProcess** (UINT uExitCode)

Інший процес може визначити код завершення, викликавши функцію **GetExitCodeProcess**.

```
BOOL GetExitCodeProcess (
        HANDLE hProcess,
        LPDWORD lpExitCode)
```

Створення потоків команд. Первинний потік команд створюється автоматично при створенні процесу. Решта потоків команд створюються функціями **CreateThread** і **CreateRemoteThread** (тільки в Win NT / 2000 / XP). Більше функцій щодо керування роботою потоків команд наведено в таблиці А.1 в Додатку А.

2.1.1 Синхронізація потоків команд

Працюючи паралельно, потоки команд спільно використовують адресний простір процесу. Також всі вони мають доступ до описувачів (handles) відкритих в процесі об'єктів. У випадках коли декілька потоків команд одночасно звертаються до одного ресурсу або необхідно якось упорядкувати роботу потоків команд необхідно використовувати об'єкти синхронізації і відповідні механізми.

2.1.1.1 М'ютекси

Це об'єкти ядра, які створюються функцією CreateMutex(). М'ютекс буває в двох станах - зайнятому і вільному. М'ютексом добре захищати одиничний ресурс від одночасного звернення до нього різними потоками команд.

```
HANDLE CreateMutex (
        LPSECURITY_ATTRIBUTES lpsa,
        BOOL bInitialOwner,
        LPCTSTR lpMutexName)

BOOL ReleaseMutex (HANDLE hMutex)
```

М'ютекс, у якого потік команд, що ним володів, завершився, не звільнивши його, називають покинутим (abandoned), і його дескриптор переходить в сигнальний стан. На те, що сигналізуючий дескриптор (дескриптори) є покинутим м'ютексом (м'ютексами), вказує повернення функцією **WaitForSingleObject** значення **WAIT_ABANDONED_0** або використання значення WAIT_ABANDONED_0 як базового значення функцією **WaitForMultipleObject**.

Те, що дескриптори покинутих м'ютексів переходять в сигнальний стан, є вельми корисною їх властивістю, недоступною в разі об'єктів CS. Виявлення покинутого м’ютекса може означати наявність дефекту в коді, що організує роботу

потоків, оскільки потоки повинні програмуватися таким чином, щоб ресурси завжди звільнялися, перш ніж потік команд завершить своє виконання. Можливо також, що виконання даного потоку було перервано іншим потоком команд.

2.1.1.2 Семафори

Семафор (**Semaphore**) створюється функцією CreateSemaphore (). Він дуже схожий на м'ютекс, тільки на відміну від нього у семафора є лічильник. Семафор відкритий якщо лічильник більше 0 і закритий, якщо лічильник дорівнює 0. Семафором зазвичай "огороджують" набори рівнозначних ресурсів (елементів), наприклад чергу, список і т.п.

HANDLE **CreateSemaphore** (
LPSECURITY_ATTRIBUTES lpsa,
LONG lSemInitial,
LONG lSemMax,
LPCTSTR lpSemName)

2.1.1.3 Події

Також як і м'ютекси мають два стани - встановлений і скинутий. Події бувають зі скиданням вручну і з автоскиданням. Коли потік дочекався (wait-функція повернула управління) події з автоскиданням, така подія автоматично скидається. В іншому випадку подію потрібно скидати вручну, викликавши функцію ResetEvent (). Припустимо, відразу кілька потоків очікують однієї і тієї ж події, і подія спрацювала. Якщо це була подія з автоскиданням, то вона дозволить працювати тільки одному потоку (адже відразу ж після повернення з його wait-функції подія скинеться автоматично!), а інші потоки залишаться чекати. Якщо ж це була подія зі скиданням вручну, то всі потоки отримають управління, а подія так і залишиться у встановленому стані, поки який-небудь потік не зробить виклик ResetEvent ().

HANDLE **CreateEvent** (

```
LPSECURITY_ATTRIBUTES lpsa,
BOOL bManualReset,
BOOL bInitialState,
LPTCSTR lpEventName)
```

2.1.1.4 Очікувані таймери

Найвитонченіший об'єкт ядра для синхронізації. З'явилися вони, починаючи з Windows 98. Таймери створюються функцією **CreateWaitableTimer** і бувають, також як і події, з автоскиданням і без нього. Потім таймер треба налаштувати функцією **SetWaitableTimer**. Таймер переходить в сигнальний стан, коли закінчується його таймаут. Скасувати "цокання" таймера можна функцією **CancelWaitableTimer**. Примітно, що можна вказати callback функцію при установці таймера. Вона буде виконуватися, коли спрацьовує таймер.

2.1.1.5 Критичні секції

Критична секція гарантує вам, що частини коду програми, обгороджені нею, не виконуватимуться одночасно. Строго кажучи, критична секція не є об'єктом ядра. Вона являє собою структуру, що містить кілька ознак і якийсь (не важливо) об'єкт ядра. При вході в критичну секцію спочатку перевіряються ознаки, і якщо з'ясовується, що вона вже зайнята іншим потоком, то виконується звичайна wait-функція. Критична секція примітна тим, що для перевірки, зайнята вона чи ні, програма не переходить в режим ядра (не виконується wait-функція), а лише перевіряються ознаки. Через це вважається, що синхронізація за допомогою критичних секцій найбільш швидка. Таку синхронізацію називають «синхронізація в призначеному для користувача режимі». Перелік функцій для роботи з критичними секціями наведено в таблиці А.2 в Додатку А.

Приклад коду, який демонструє, що може статися без критичної секції:

```
const int MAX_TIMES = 1000,
        int g_nIndex - 0,
        DWORD g_dwTimes [MAX_TIMES];
DWORD WINAPI FirstThread (PVOID pvParam)
{
   while (g_nIndex <MAX_TIMES)
      {
         g_dwTimes [g__nIndex] = GetTickCount ();
         g_nIndex ++;
      }
   return (0),
}
DWORD WINAPI SecondThread (PVOID pvParam)
{
   while Cg_nIndex <MAX_TIMES)

      {
         g_nIndex ++;
         g_dwTimes [g_nIndex - 1] = GetTickCount ();

      }
   return (0);
}
```

Тут передбачається, що функції обох потоків команд дають однаковий результат, хоч вони і закодовані з невеликими відмінностями. Якби виконувалася тільки функція FirstThread, вона заповнила б масив g_dwTimes набором чисел із дедалі вищими значеннями. Це вірно і по відношенню до SecondThread – якби вона теж виконувалася незалежно. В ідеалі обидві функції навіть при одночасному виконанні повинні б, як і раніше, заповнювати масив тим же набором чисел. Але в нашому коді виникає проблема: масив g_dwTimes не буде заповнений, як треба, тому що функції обох потоків одночасно

звертаються до одних і тих же глобальних змінних. Ось як це може статися.

Припустимо, ми тільки що почали виконання обох потоків в системі з одним процесором. Першим включився в роботу другий потік, тобто функція SecondThread (що цілком ймовірно), і тільки вона встигла збільшити лічильник g_nIndex 1, як система витіснила її потік команд і перейшла до виконання FirstThread. Та заносить в g_dwTimes [1] показання системного часу, і процесор знову перемикається на виконання другого потоку. SecondThread тепер привласнює елементу g_dwTimes [1-1] нові свідчення системного часу. Оскільки ця операція виконується пізніше, нові значення, природно, вище, ніж записані в елемент g_dwTimes функцією FirstThread. Відзначте також, що спочатку заповнюється перший елемент масиву і тільки потім нульовий. Таким чином, дані в масиві виявляються помилковими. Використовуючи критичні секції, можна і потрібно координувати доступ потоків до структур даних. Для потоко-безпечної зміни значень в змінних деяких типів можна скористатися функціями, які наведені в таблиці А.3 додатку А.

Виправлений фрагмент коду за допомогою критичної секції:

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes [MAX_TIMES];
CRITICAL_SECTION g_cs;
DWORD WINAPI FirstThread (PVOID pvParam)
{
   for (BOOL fContinue = TRUE; fContinue;) {
      EnterCriticalSection (& g_cs);
      if (g_nIndex <MAX_TIMES) {
         g_dwTimes [g_nlndex] = GetTickCount ();
         g_nIndex ++;
      }
      else
```

```
            fContinue = FALSE;
         LeaveCriticalSection (& g_cs);
      }
      return (0);
   }
   DWORD WINAPI SecondThread (PVOID pvParam)
   {
      for (BOOL fContinue = TRUE; fContinue;) {
         EnterCriticalSection (& g_cs);
         if (g__nIndex <MAX_TIMES) {
            g_nIndex ++;
            g_dwTimes [g_nIndex - 1] = GetTickCount ();
         }
         else
            fContinue = FALSE;
         LeaveCriticalSecLion (& g_cs);
      }
      return (0);
   }
```

Синхронізація процесів. Описувачі об'єктів ядра залежні від конкретного процесу (process specific). Простіше кажучи, handle об'єкта, отриманий в одному процесі, не має сенсу в іншому. Однак існують способи роботи з одними і тими ж об'єктами ядра з різних процесів.

По-перше, це успадкування опису. При створенні об'єкта можна вказати чи буде його описувач успадковуватися дочірніми (породженими цим процесом) процесами.

По-друге, дублювання опису. Функція **DuplicateHandle** дублює описувач об'єкта одного процесу в інший, тобто по суті, бере запис в таблиці описувачів одного процесу і створює її копію в таблиці іншого.

І, нарешті, іменування об'єкта ядра. При створенні об'єкта ядра для синхронізації (м’ютекса, семафора, очікуваного

таймера або події) можна задати його ім'я. Воно повинно бути унікальним в системі. Тоді інший процес може відкрити цей об'єкт ядра, вказавши в функції Open... (OpenMutex, OpenSemaphore, OpenWaitableTimer, OpenEvent) це ім'я.

Насправді, при виконанні функції Create... () система спочатку перевіряє, чи не існує вже об'єкт ядра з таким ім'ям. Якщо немає, то створюється новий об'єкт. Якщо так, ядро перевіряє тип цього об'єкта і права доступу. Якщо типи не збігаються або ж процес, що викликає, не має повних прав на доступ до об'єкта, виклик Create... функції закінчується невдало і повертається NULL. Якщо все нормально, то просто створюється новий описувач (handle) існуючого вже об'єкта ядра. За кодом повернення функції GetLastError () можна зрозуміти що сталося: створився новий об'єкт або Create () повернула вже існуючий.

Тому, синхронізувати потоки команд всередині різних процесів можна точно так само як і в межах одного. Потрібно тільки правильно передати описувач синхронізуючого об'єкта від одного процесу до іншого будь-яким з перерахованих вище способів.

Програма 2.1 - lab2-1.cpp

```
#include <iostream> // cout
#include <windows.h> // CreateThread ()
#include <conio.h> // _getch ()

using namespace std;

DWORD WINAPI thread2 (LPVOID);

int main ()
{
  cout << "First thread \ n";
```

```
    HANDLE thread = CreateThread (NULL, 0, thread2, NULL,
0, NULL);
    cout << "More data from first thread \ n";
    for (int i = 0; i <1000000; i ++) {}
    cout << "Even more data from first thread \ n";
    _getch ();
    return 0;
}

DWORD WINAPI thread2 (LPVOID t)
{
    cout << "Second thread \ n";
    return 0;
}
```

Запустіть програму. Спочатку виводиться текст First thread. Далі створюється другий потік команд. Що виведеться далі однозначно стверджувати не можна: More data from first thread або Second thread. Залежить від того який потік команд першим встигне надрукувати повідомлення. У більшості випадків перший потік команд встигне швидше, другому потоку команд ще належить виділити ресурси і почати виконувати свою функцію.

Далі в першому потоці команд запускається порожній цикл, що виконується мільйон разів. За цей час другий потік команд встигає надрукувати повідомлення Second thread. І після цього виводиться останні повідомлення першого потоку команд.

Програма 2.2 - lab2-2.cpp

```
#include <windows.h>
#include <process.h>

// Функція потоку команд
```

```
unsigned __stdcall ThreadFunc (void * arg)
{
   char ** str = (char **) arg;
   MessageBox (0, str [0], str [1], 0);
   _endthreadex (0);
   return 0;
};
int main (int argc, char * argv [])
{
   // рядок для першого потоку команд
   char * InitStr1 [2] = { "First thread running!", "11111"};
   // рядок для другого потоку команд
   char * InitStr2 [2] = { "Second thread running!", "22222"};
   unsigned uThreadIDs [2];

   HANDLE hThreads [2];
   hThreads [0] = (HANDLE) _beginthreadex (NULL, 0, &
ThreadFunc, InitStr1, 0, & uThreadIDs [0]);
   hThreads [1] = (HANDLE) _beginthreadex (NULL, 0, &
ThreadFunc, InitStr2, 0, & uThreadIDs [1]);

   // Чекаємо, поки потоки команд не закінчать свою роботу
   // чекаемо без тайм-ауту
   WaitForMultipleObjects (2, hThreads, TRUE, INFINITE);
   // Закриваємо дескриптори
   CloseHandle (hThreads [0]);
   CloseHandle (hThreads [1]);
   return 0;
}
```

Програма створює два однакових потоки команд і очікує їх завершення. Потоки команд виводять текстове повідомлення, яке передано їм при ініціалізації (InitStr1, InitStr2).

Примітка: якщо у вас виникла помилка "Аргумент типу" char * "несумісний з параметром типу" LPCWSTR ", то необхідно переключити в налаштуваннях проекту набір символів з Юнікоду на багатобайтове кодування.

Програма 2.3 - lab2-3.cpp

```
#include <process.h>
#include <windows.h>
#include <stdio.h>
#include <conio.h>

#define HOUR (8) // час, коли спрацьовує будильник (тільки години)
#define RINGS (10) // скільки разів дзвонити

HANDLE hTerminateEvent;

// callback функція таймера
VOID CALLBACK TimerAPCProc (LPVOID, DWORD, DWORD)
{
  Beep (1000, 500); // дзвонимо!
};

// функція потоку команд
unsigned __stdcall ThreadFunc (void *)
{
  HANDLE hDayTimer = CreateWaitableTimer (NULL, FALSE, NULL);
  HANDLE hAlarmTimer = CreateWaitableTimer (NULL, FALSE, NULL);
  HANDLE h [2]; // ми будемо чекати ці об'єкти
  h [0] = hTerminateEvent;
```

```
        h [1] = hDayTimer;
        int iRingCount = 0; // кількість "дзвінків"
        int iFlag;
        DWORD dw;

        // таймер приймає значення тільки в форматі FILETIME
        LARGE_INTEGER liDueTime, liAllDay;
        liDueTime.QuadPart = 0;
        // добу в 100-наносекундних інтервалах = 10000000 * 60
* 60 * 24 = 0xC92A69C000
        liAllDay.QuadPart = 0xC9;
        liAllDay.QuadPart = liAllDay.QuadPart << 32;
        liAllDay.QuadPart | = 0x2A69C000;
        SYSTEMTIME st;
        GetLocalTime (& st); // дізнаємося поточну дату / час
        iFlag = st.wHour> HOUR; // якщо призначений час ще не
настав,
        // то ставимо будильник на сьогодні, інакше - на завтра
        st.wHour = HOUR;
        st.wMinute = 0;
        st.wSecond = 0;
        FILETIME ft;
        SystemTimeToFileTime (& st, & ft);
        if (iFlag)
          ((LARGE_INTEGER *) & ft) -> QuadPart =
((LARGE_INTEGER *) & ft) -> QuadPart + liAllDay.QuadPart;

        LocalFileTimeToFileTime (& ft, & ft);
        // Встановлюємо таймер,
        // він буде спрацьовувати раз на добу (24 * 60 * 60 *
1000ms),
        // починаючи з наступної "години дзвоника" - HOUR
        SetWaitableTimer (hDayTimer, (LARGE_INTEGER *) & ft,
24 * 60 * 60000, 0, 0, 0);
```

```
        do {
          dw = WaitForMultipleObjectsEx (2, h, FALSE,
INFINITE, TRUE);
          if (dw == WAIT_OBJECT_0 + 1) // спрацював
hDayTimer
          {
            // Встановлюємо таймер, він буде викликати callback
ф-ію раз на секунду,
            // почне з поточного моменту
            SetWaitableTimer (hAlarmTimer, & liDueTime, 1000,
TimerAPCProc, NULL, 0);
            iRingCount = 0;
          }
          if (dw == WAIT_IO_COMPLETION) // закінчила
працювати callback ф-ія
          {
            iRingCount ++;
            if (iRingCount == RINGS)
              CancelWaitableTimer (hAlarmTimer);
          }
        } While (dw! = WAIT_OBJECT_0); // поки не спрацювало
hTerminateEvent залишаємося в циклі

        // закриваються handles, завершення роботи
        CancelWaitableTimer (hDayTimer);
        CancelWaitableTimer (hAlarmTimer);
        CloseHandle (hDayTimer);
        CloseHandle (hAlarmTimer);
        _endthreadex (0);
        return 0;
      };

      int main (int argc, char * argv [])
      {
```

```
        // це подія сповіщає потік команд коли треба завершуватися
        hTerminateEvent = CreateEvent (NULL, FALSE, FALSE, NULL);
        unsigned uThreadID;
        HANDLE hThread;
        // створюємо потік команд
        hThread = (HANDLE) _beginthreadex (NULL, 0, & ThreadFunc, 0, 0, & uThreadID);
        puts ( "Press any key to exit.");
        // чекаємо any key від користувача для завершення програми
        _getch ();
        // встановлюємо подію
        SetEvent (hTerminateEvent);
        // чекаємо завершення потоку команд
        WaitForSingleObject (hThread, INFINITE);
        // закриваємо handle
        CloseHandle (hThread);
        return 0;
    }
```

У програмі 2.3 використовуються WaitableTimer'и. Будильник буде спрацьовувати раз на день о 8 ранку і "дзвонити" 10 разів. Використовуємо для цього два таймера, один з яких з callback-функцією.

2.2 Завдання

1. Написати дві програми: головну і дочірню. Головна програма повинна запускати дочірню і чекати її завершення. Використовувати функції CreateProcess і WaitForSingleObject.

2. Написати програму, первинний потік команд якої запускає кілька додаткових потоків команд. Реалізувати обмін

повідомленнями між головним і побічними потоками команд за допомогою передачі повідомлень. Використовувати функції CreateThread, PostThreadMessage, GetMessage. Як мінімум, вторинні потоки команд повинні за допомогою повідомлень повідомляти головний потік про свою готовність до роботи і про завершення свого виконання.

3. *Написати програму, первинний потік команд якої запускає кілька додаткових потоків команд. Реалізувати критичну секцію коду, яку всі потоки команд проходять строго послідовно і почергово. Використовувати функції CreateThread, InitializeCriticalSection, DeleteCriticalSection, EnterCriticalSection, LeaveCriticalSection. Як мінімум, в критичну секцію коду потрібно помістити друк повідомлення - який потік команд зайшов в критичну секцію і підрахунок числа відвідувань цієї секції коду. За основу можна використати програму 2.4.

Програма 2.4 - lab2-4.cpp

```
#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include <process.h>

#define MAX_THREADS 10

CRITICAL_SECTION critsect;
int count = 0;

DWORD WINAPI ThreadFunc(void *param)
{
   DWORD main_thread = *((DWORD*)param);
   DWORD id = GetCurrentThreadId();

   EnterCriticalSection(&critsect);
   count++;
```

```
    printf("%u : Thread %u enter\n", count, id);

    // for (int i = 0; i<10; i++)
    // {
        Sleep(150);
     // }
    printf("    Thread %u complete\n", id);

    LeaveCriticalSection(&critsect);
    return 1;
}
int main(int argc, char **argv)
{
    DWORD main_id = GetCurrentThreadId();
    HANDLE ThreadHandleArray[MAX_THREADS];

    InitializeCriticalSection(&critsect);
    for (int i = 0; i<MAX_THREADS; i++)
    {
      ThreadHandleArray[i] = CreateThread(NULL, 0,
ThreadFunc, (void*)& main_id, 0, NULL);
    }
    WaitForMultipleObjects(MAX_THREADS,
ThreadHandleArray, TRUE, INFINITE);
    DeleteCriticalSection(&critsect);
    return 0;
}
```

2.3 Контрольні питання

1. Що таке процес і потік команд? У чому їх різниця?
2. Як створювати і завершувати процеси і потоки команд?
3. В яких випадках потік команд завершується?

4. Як можна призупинити роботу потоку команд ?

5. Пояснити, що буде виконано після запуску даної програми.

```
#include <windows.h>
int main (int argc, char * argv [])
{
   STARTUPINFO si;
   PROCESS_INFORMATION pi;
   ZeroMemory (& si, sizeof (si));
   si.cb = sizeof (si);
   ZeroMemory (& pi, sizeof (pi));
   if ( !CreateProcess (NULL, "c: /windows/calc.exe", NULL,
NULL, FALSE, 0, NULL, NULL, & si, & pi)) {
      return 0;
   }
   CloseHandle (pi.hProcess);
   CloseHandle (pi.hThread);
   return 0;
}
```

6. У чому необхідність синхронізації потоків команд?

7. Які об'єкти синхронізації потоків команд Ви знаєте?

8. Що таке «покинуті м'ютекси»?

9. Проведіть порівняльний огляд м'ютексів і об'єктів CRITICAL_SECTION.

10. *Об'єкти CRITICAL_SECTION призначені для використання потоками в рамках одного і того ж процесу. Що станеться, якщо об'єкт CS буде створений в загальній області пам'яті (shared) як відображається у декілька процесів?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

1. Геннадій Галісєєв. Системне програмування. – Університет "Україна". 2019 – 113 с.

2. Robert Love. Linux System Programming, Second Edition. – O'Reilly Media, Inc. 2013 – 456 с.

3. Johnson M. Hart. Windows System Programming. Fourth Edition. – Addison-Wesley Microsoft Technology Series. 2010 – 609 с.

4. Костенко А.В., Костирко В.С., Плеша М.І. Кросплатформне програмування : навч. посіб. – Центр. спілка спожив. т-в України, Львів. 2019. - 247 с.

5. Рисований О.М. Системне програмування. Графічний інтерфейс користувача (GUI). Навч. посібник. – Х.: НТУ "ХПІ", 2018. – 160 с.

6. Рисований О.М. Системне програмування: підручник для студентів напрямку "Комп'ютерна інженерія" вищих навчальних закладів в 2-х томах. Том 1. – Видання четверте: виправлено та доповнено – Х.: "Слово", 2015. – 576 с.

7. Рисований О.М. Системне програмування: підручник для студентів напрямку "Комп'ютерна інженерія" вищих навчальних закладів в 2-х томах. Том 2. – Видання четверте: виправлено та доповнено – Х.: "Слово", 2015. – 378 с.

8. Шевцов В.А. Операційні системи. - К.: Видавнича група BHV, 2005 – 576 с.

9. Andrew S. Tanenbaum, Herbert Bos. Modern Operating Systems. Fifth Edition – Hoboken, New Jersey 07458 – 2024. – 1185 с. ISBN: 978-1-292-45966-0

10. Windows Internals, Seventh Edition, Part 1: System architecture, processes, threads, memory management, and more by Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. – Microsoft Press, 2017. – ISBN: 978-0-7356-8418-8.

11. Windows Internals, Seventh Edition, Part 2 by Andrea Allievi, Alex Ionescu, Mark E. Russinovich, and David A. Solomon. – Microsoft Press, 2021. – ISBN: 978-0-13-546240-9.

Додаток А
Функції роботи з потоками команд

Таблиця А.1 - функції роботи з потоками команд

Функція	Опис
_BeginThread	Створення й запуск потоку команд
_beginthredex	Запускає ініціалізований потік.
AttachThreadInput	Зв'язує обробку потоку вводу з іншим потоком команд, дозволяє передавати фокус вводу вікнам іншого потоку команд, а також використовувати загальний стан вводу.
CreateRemoteThread	Створює потік в іншому процесі
CreateThread	Створює порожній потік у поточному процесі
ExitThread	Завершує роботу поточного потоку.
TerminateThread	Припиняє роботу потоку без виконання необхідних підготовчих процедур
GetCurrentThread	Повертає дескриптор поточного потоку.
GetCurrentThreadID	Повертає ідентифікатор поточного потоку
GetExitCodeThread	Повертає код завершення потоку.
SetThreadPriority	Установлює рівень пріоритету потоку
GetThreadPriority	Повертає рівень пріоритету потоку
SetTheadPriorityBoost	Дозволяє або забороняє динамічні зміни рівня пріоритету даного потоку
GetThreadPriorityBoost	Повертає статус динамічних змін пріоритету для даного потоку
GetThreadTimes	потоку. GetThreadTimes Повертає час, коли був створений або знищений потік, та яка кількість процесорного часу була їм використана

Продовження таблиці А.1

SuspendThread	Тимчасово припиняє виконання потоку
ResumeThread	Продовжує роботу потоку, роботу якого було припинено в результаті звертання до SuspendThread
SetThreadAffinityMask	Установлює, які процесори можуть використовуватися для виконання даного потоку
SetThreadIdealProcessor	Установлює який процесор переважно використовується для виконання даного потоку
SwichToThread	Переключає процесор на виконання іншого потоку (невідомо, якого саме).

Таблиця А.2 – функції для роботи із критичною секцією

InitializeCriticalSection	Ініціалізує змінну типу CRITICAL_SECTION
InitializeCriticalSectionAndSpinCount	Ініціалізує змінні типу CRITICAL_SECTION і лічильник її очікування
EnterCriticalSection	“Входить” у критичну секцію й блокує інші потоки команд, що намагаються також увійти в цю критичну секцію
LeaveCriticalSection	“Полишає” критичну секцію, дозволяючи іншим потокам команд увійти до неї.

Продовження таблиці А.2

TryEnterCriticalSection	Намагається ввійти в критичну секцію, повертаючи помилку у випадку якщо критична секція вже виконується будь-яким потоком.
SetCriticalSectionSpinCount	Установлює значення лічильника очікування критичної секції.
DeleteCriticalSection	Знищує змінні CRITICAL_SECTION.

Таблиця А.3 – Windows API виклики зміни даних з блокуванням

InterlockedDecrement	Віднімає одиницю із блокованої змінної.
InterlockedIncrement	Додає одиницю до блокованої змінної.
InterlockedExchange	Міняє місцями значення блокованої змінної й значення іншої змінної
InterlockedExchangeAdd	Додає значення до блокованої змінної
InterlockedCompareExchange	Порівнює значення блокованої змінної з деяким значенням, і у випадку, якщо значення рівні, міняє місцями значення блокованої змінної й значення іншої змінної