

Міністерство освіти і науки України
Запорізький національний технічний університет

МЕТОДИЧНІ ВКАЗІВКИ
до виконання лабораторних робіт з дисципліни
"Системне програмування"
для студентів спеціальності
123 "Комп'ютерна інженерія"
всіх форм навчання
Частина 1

2019

Методичні вказівки до виконання лабораторних робіт з дисципліни "Системне програмування" Частина 1 для студентів спеціальності 123 "Комп'ютерна інженерія" всіх форм навчання. Частина 1 / Укл. А.В. Тіменко, М.М. Хохлов – Запоріжжя: ЗНТУ, 2019. – 26 с.

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

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

Відповідальний за випуск: А.В.Тіменко,асистент

Затверджено:
на засіданні кафедри
"Комп'ютерні системи та
мережі"
Протокол № 11 від 20.05.2019

Рекомендовано до видання
НМК ФКНТ
Протокол № 10 від 31.05.2019

ЗМІСТ

ЗМІСТ	3
1 ЛАБОРАТОРНА РОБОТА № 1. ПОТОКИ ВВОДУ-ВИВОДУ	4
1.1 Теоретичні відомості	4
1.2 Завдання	8
1.3 Контрольні питання:	8
2 ЛАБОРАТОРНА РОБОТА № 2. ЗАПУСК ПРОЦЕСІВ І ПОТОКІВ (MUTEX / FUTEX, SEMAPHORE, EVENTS, TIMERS, CRITICALSECTIONS)	9
2.1 Теоретичні відомості	9
2.2 Завдання	23
2.3 Контрольні питання:	24
Додаток А	25

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 введена 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 C 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(  
    _In_ HANDLE hConsoleInput,  
    _Out_ LPVOID lpBuffer,  
    _In_ DWORD nNumberOfCharsToRead,  
    _Out_ LPDWORD lpNumberOfCharsRead,  
    _In_opt_ LPVOID pInputControl );
```

Вивести дані до екрану (консолі):

Бібліотека C/C++:

printf, putc, puts, fwrite, cout<<

Системний виклик:

```
BOOL WINAPI WriteConsole(
    _In_ HANDLE hConsoleOutput,
    _In_ const VOID *lpBuffer,
    _In_ DWORD nNumberOfCharsToWrite,
    _Out_ LPDWORD lpNumberOfCharsWritten,
    _Reserved_ LPVOID lpReserved );
```

* Примітка. Системні виклики ReadConsole/WriteConsole не можуть працювати з потоками введення/виведення, якщо в якості hConsoleInput/hConsoleOutput використовуються перенаправлені дескриптор(и). В таких випадках слід використовувати системні виклики ReadFile та WriteFile.

```
HANDLE WINAPI GetStdHandle( _In_ DWORD nStdHandle);
```

```
BOOL WINAPI GetConsoleMode(
    _In_ HANDLE hConsoleHandle,
    _Out_ LPDWORD lpMode
);
```

```
BOOL WINAPI SetConsoleMode(
    _In_ HANDLE hConsoleHandle,
    _In_ DWORD dwMode
);
```

```
BOOL WINAPI SetConsoleTitle(__in LPCTSTR lpConsoleTitle);
```

```
BOOL WINAPI SetConsoleTextAttribute(
    __in HANDLE hConsoleOutput,
    __in 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(
    __in HANDLE hConsoleOutput,
    __in WORD wAttribute,
    __in DWORD nLength,
    __in COORD dwWriteCoord,
```

```
__out LPDWORD lpNumberOfAttrsWritten);
```

hConsoleOutput – дескриптор буферу консолі виводу;
wAttribute – атрибут кольору фону символу в консолі;
nLength – кількість комірок символів, фон яких встановлюється заданим кольором;
dwWriteCoord – координати першої комірки;
lpNumberOfAttrsWritten – показчик на ідентифікатор, в який буде записано кількість реально зафарбованих комірок.

```
void WINAPI OutputDebugString(_In_opt_ LPCTSTR  
lpOutputString);
```

Програма 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

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 цей приклад коду:

```
BOOL WINAPI FillConsoleOutputAttribute(
    __in HANDLE hConsoleOutput,
    __in WORD wAttribute,
    __in DWORD nLength,
    __in COORD dwWriteCoord,
    __out 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. 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 (threads). Потік, що виконується всередині процесу, може сам собі створювати нові потоки і нові незалежні процеси, а так само управляти взаємодією об'єктів між собою і їх синхронізацією.

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

```

BOOL CreateProcess (
    LPCTSTR lpApplicationName, // ім'я виконуваного модуля
    LPCTSTR lpCommandLine, // Командний рядок
    // Показчик на структуру SECURITY_ATTRIBUTES
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    // Показчик на структуру SECURITY_ATTRIBUTES
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    // Прапор успадкування пов'язаних з поточною діяльністю
    BOOL bInheritHandles,
    // Прапори способів створення процесу
    DWORD dwCreationFlags,
    // Показчик на блок середовища
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory, // Поточний диск або каталог
    // Показчик на структуру STARTUPINFO
    LPSTARTUPINFO lpStartupInfo,
    // Показчик на 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, 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. Первинний потік створюється автоматично при створенні процесу. Решта потоків створюються функціями CreateThread і CreateRemoteThread (тільки в Win NT / 2000 / XP).

Синхронізація потоків. Працюючи паралельно, потоки спільно використовують адресний простір процесу. Також всі вони мають доступ до описувачів (handles) відкритих в процесі об'єктів. А що робити, якщо кілька потоків одночасно звертаються до одного ресурсу або необхідно якось упорядкувати роботу потоків? Для цього використовують об'єкти синхронізації і відповідні механізми.

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.

```
HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpSa,
    BOOL bInitialOwner, LPCTSTR lpMutexName)
    BOOL ReleaseMutex (HANDLE hMutex)
```

М'ютекс, у якого потік, що ним володів, завершився, не звільнивши його, називають покинутим (abandoned), і його дескриптор переходить в сигнальний стан. На те, що сигналізуючий дескриптор (дескриптори) являє покинутий м'ютекс (м'ютекси), 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 завершить своє виконання. Можливо також, що виконання даного потоку було перервано іншим потоком.

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 "огорожують" набори рівнозначних ресурсів (елементів), наприклад чергу, список і т.п.

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

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 отримують управління, а подія так і залишиться у встановленому стані, поки який-небудь потік не викличе ResetEvent ().

```
HANDLE CreateEvent (LPSECURITY_ATTRIBUTES lpsa, BOOL
                    bManualReset, BOOL bInitialState, LPCTSTR lpEventName)
```

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 спрацює таймер.

Критичні секції. Критична секція гарантує вам, що шматки коду програми, обгороджені нею, не виконуватимуться одночасно. Строго кажучи, критична секція не є об'єктом ядра. Вона являє собою структуру, що містить кілька прапорів і якийсь (не важливо) об'єкт ядра. При вході в критичну секцію спочатку перевіряються прапори, 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 прапори. Через це вважається, що синхронізація за допомогою критичних секцій найбільш швидка. Таку синхронізацію називають «синхронізація в призначеному для користувача режимі».

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 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 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 свідчення, природно, вище, ніж записані в елемент `g_dwTimes [1]` 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.

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_nIndex] = 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;
        LeaveCriticalSection (& 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 (мьютекса, 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 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 1 - lab2.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. Що виведеться далі однозначно стверджувати не можна: More data from first thread або Second thread. Залежить від того який потік першим встигне надрукувати повідомлення. У більшості випадків перший потік встигне швидше, другому потоку ще належить виділити ресурси і почати виконувати свою функцію.

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

Програма 2 - lab2-1.cpp

```

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

unsigned __stdcall ThreadFunc (void * arg) // Функція
потоків
{
    char ** str = (char **) arg;
    MessageBox (0, str [0], str [1], 0);
    _endthreadex (0);
    return 0;
};

int main (int argc, char * argv [])
{
    char * InitStr1 [2] = { "First thread running!",
"11111"}; // рядок для першого потоку
    char * InitStr2 [2] = { "Second thread running!",
"22222"}; // рядок для другого потоку
    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 не завершать свою роботу
    WaitForMultipleObjects (2, hThreads, TRUE,
INFINITE); // Встановлюємо без тайм-ауту
    // Закриваємо дескриптори
    CloseHandle (hThreads [0]);
    CloseHandle (hThreads [1]);
    return 0;
}

```

Програма створює два однакових потоки і очікує їх завершення. Потоки виводять текстові 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 настройках проекту набір символів з Юнікода на багатобайтове кодування.

Програма 3 - lab2-2.cpp

```

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

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

HANDLE hTerminateEvent;

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

// функція потоку
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; // число "дзвінків"
    int iFlag;
    DWORD dw;

    // трохи помучаємося згодом,
    // тому що таймер приймає його 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 наступного "години пік" - 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 в 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 крутимосся в циклі

        // що закриваються handles, виходимо
        CancelWaitableTimer (hDayTimer);
        CancelWaitableTimer (hAlarmTimer);
        CloseHandle (hDayTimer);

```

```

        CloseHandle (hAlarmTimer);
        _endthreadex (0);
        return 0;
};

int main (int argc, char * argv [])
{
    // це подія показує потоку коли треба завершуватися
    hTerminateEvent = CreateEvent (NULL, FALSE, FALSE,
NULL);
    unsigned uThreadID;
    HANDLE hThread;
    // створюємо потік
    hThread = (HANDLE) _beginthreadex (NULL, 0, &
ThreadFunc, 0, 0, & uThreadID);
    puts ( "Press any key to exit.");
    // чекаємо any key від користувача для завершення
програми
    _getch ();
    // виставляємо подія
    SetEvent (hTerminateEvent);
    // чекаємо завершення потоку
    WaitForSingleObject (hThread, INFINITE);
    // закриваємо handle
    CloseHandle (hThread);
    return 0;
}

```

У програмі 2-2 використовуються WaitableTimer'и. Будильник буде спрацьовувати раз в день о 8 ранку і "піку" 10 разів. Використовуємо для цього два таймера, один з яких з callback-функцією.

Програма-приклад Queue. При ініціалізації Queue створює чотири клієнтських і два серверних потоки. Кожен клієнтський потік засинає на певний період часу, а потім поміщає в чергу елемент даних. Коли в чергу ставиться новий елемент, вміст списку Client Threads оновлюється. Кожен елемент даних складається з номера клієнтського потоку і порядкового номера запиту, виданого цим потоком. Наприклад, перший запис у списку повідомляє, що клієнтський потік 0 поставив в чергу свій перший запит. Наступні записи свідчать,

що далі свої перші запити видають потоки 1-3, потім потік 0 поміщає другий запит, теж саме роблять інші потоки, і все повторюється.

У цьому прикладі серверні потоки не встигають обробляти клієнтські запити і черга в кінцевому рахунку заповнюється до максимуму. Була встановлена максимальна довжина черги дорівнює 10 елементам, що призводить до швидкого заповнення цієї черги. Крім того, на чотири клієнтських потоку припадає лише два серверних. У підсумку, черга повністю заповнюється до того моменту, коли клієнтський потік 3 намагається видати свій п'ятий запит.

Чергою управляє C++ - клас CQueue:

```
class CQueue
{
    public: Struct ELEMENT
    {
        int m_nThreadNum, m_nRequestNum;
        // інші елементи даних повинні бути визначені
тут};

        typedef ELEMENT * PELEMENT;
    private:
        PELEMENT m_pElements; // масив елементів, що
підлягають обробці
        int m_nMaxElements; // кількість елементів в
масиві
        HANDLE m_h [2]; // описатели мьютекса і семафора
        HANDLE & m_hmtxQ; // посилання на m_h [0]
        HANDLE & rn_hsemNumElements; // посилання на rc_h [1]
    public:
        CQueue (int nMaxElements);
        ~ CQueue ();
        BOOL Append (PELEMENT pElement, DWORD
dwMilHseconds);
        BOOL Remove (PELEMENT pElement, DWORD
dwMilliseconds);
};
```

Відкрита структура ELEMENT всередині цього класу визначає, що є елементом даних, що поміщається в чергу. Його реальний вміст в даному випадку не має значення. У цій програмі-прикладі клієнтські потоки записують в елемент даних

власний номер і порядковий номер свого чергового запиту, а серверні потоки, обробляючи запити, показують цю інформацію в списку. У реальному додатку така інформація навряд чи б знадобилася.

Як Ви незабаром побачите, програма періодично викликає `WaitForMultipleObjectS`, передаючи цій функції адресу масиву описувачів. Ви також переконаєтесь, що програмі час від часу доводиться посилатися тільки на один з цих описувачів. Що б полегшити читання коду і його модифікацію, було об'явлено два елементи, кожен з яких містить посилання на один з описувачів, `m_bmtxQ` і `m_hsemNumElements`. Конструктор `CQueue` ініціалізує ці елементи вмістом `m_h [0]` і `m_h [1]` відповідно. Метод `Append` намагається додати `ELEMENT` в чергу. Але спочатку він повинен переконатися, що потоку, який викликається, дозволений монопольний доступ до черги. Для цього метод `Append` викликає `WaitForSingleObject`, передаючи їй описувач об'єкта-мьютекса, `m_hmtxQ`. Якщо функція повертає `WAIT_OBJECT_0`, значить, потік отримав монопольний доступ до черги.

Далі метод `Append` повинен спробувати збільшити число елементів в черзі, викликавши функцію `ReleaseSemaphore` і передавши їй лічильник числа звільнень (`release count`), що дорівнює 1. Якщо виклик `ReleaseSemaphore` проходить успішно, в черзі ще є місце, і в неї можна помістити новий елемент.

`ReleaseSemaphore` повертає у змінній `PreviousCount` попередню кількість елементів в черзі. Завдяки цьому Ви точно знаєте, в який елемент масиву слід записати новий елемент даних. Скопіювавши елемент в масив черги, функція повертає управління.

Після закінчення цієї операції `Append` викликає `ReleaseMutex`, щоб і інші потоки могли отримати доступ до черги. Решта коду в методі `Append` відповідає за обробку помилок і невдалих викликів.

Тепер подивимося, як серверний потік викликає метод `Remove` для вибірки елемента з черги. Спочатку цей метод повинен переконатися, що потік, який викликається, отримав монопольний доступ до черги і що в ній є хоча б один елемент. Зрозуміло, серверному потоку немає сенсу прокидатися, якщо черга порожня. Тому метод `Remove` попередньо звертається до `WaitForMultipleObjects`, передаючи їй описувачі мьютекса і семафора. І тільки після звільнення обох об'єктів серверний потік може прокинутися.

Якщо повертається `WAIT_OBJECT_0`, значить, потік отримав монопольний доступ до черги і в ній є хоча б один елемент. У цей момент програма витягує з масиву елемент з індексом 0, а інші елементи зрушує вниз на одну позицію. Це, звичайно, не найефективніший спосіб реалізації черги, так як вимагає занадто великої кількості операцій копіювання в пам'яті, але наша мета полягає лише в тому, щоб продемонструвати синхронізацію потоків. Після закінчення цих операцій викликається `ReleaseMutex`, і черга стає доступною іншим потокам.

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, головний потік якої запускає кілька додаткових потоків. Реалізувати обмін повідомленнями між головним і побічними потоками за допомогою передачі повідомлень. Використовувати функції `CreateThread`, `PostThreadMessage`, `GetMessage`. Як мінімум, вторинні потоки повинні за допомогою повідомлень повідомляти головний потік про свою готовність до роботи і про завершення свого виконання.

3 *. Написати програму, головний потік якої запускає кілька додаткових потоків. Реалізувати критичну секцію коду,

яку всі потоки проходять строго послідовно і по чергово. Використовувати функції `CreateThread`, `InitializeCriticalSection`, `DeleteCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`. Як мінімум, в критичну секцію коду потрібно помістити друк повідомлення - який потік зайшов в критичну секцію і підрахунок числа відвідувань цієї секції коду.

2.3 Контрольні питання:

1. Що таке процес і потік? У чому їх різниця?
2. Як створювати і завершувати процеси і потоки?
3. В яких випадках потік завершується?
4. Як можна призупинити роботу потоку?
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?
7. Які об'єкти синхронізації потоків 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 розділяється відображається області пам'яті?

Додаток А

```

#include "stdafx.h"
#include <stdio.h>
#include <windows.h>
#include <process.h>
#include <locale.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)
{
    setlocale(LC_ALL, "Russian");
    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);
    system("pause");
    return 0;
}

```

ЛІТЕРАТУРА

1. Харт Д. Системное программирование в среде Windows, 3-е издание.: Пер. с англ. - М.: Издательский дом "Вильямс", 2015. - 592 с.
2. Басе Л., Клементе П., Кацман Р. Архитектура программного обеспечения на практике. 2-е издание. — СПб.: Питер, 2016. — 575 с
3. Уотсон К., Нейгел К., Педерсен Я.Х., Рид Д.Д., Скиннер С. Visual C 2018: полный курс. М: Диалектика, 2018. - 1340 с.
4. Абель «Системное программирование», М. Высшая школа, 1990. - 456 с.
5. Вильямс И.Д. Системное программирование в среде Windows, М: Вильямс, 2015. - 592 с.
6. Таненбаум Э. Современные операционные системы, 3-е издание, СПб: Питер, 2017. - 1038 с.
7. Таненбаум Э. Архитектура компьютера, 5-е издание, СПб: Питер, 2007. - 698 с.
8. Орлов С. Технологии разработки программного обеспечения. Издание 3. СПб: «Питер», 2004., 528 с.
9. Несвижский В. Программирование аппаратных средств в Windows (2-е издание), М: Диалектика-Вильямс, 2008. 528 с.