МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

В. І. Панченко, О. В. Коломійцев, С. Г. Межерицький

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ. ПРОГРАМУВАННЯ СИСТЕМНИХ МЕХАНІЗМІВ ОС

Навчальний посібник
для студентів спеціальності
123 «Комп'ютерна інженерія»
денної та заочної форм навчання

Затверджено
редакційно-видавничою
радою НТУ «ХПІ»,
протокол № 2 від 27.06.24

Харків
НТУ «ХПІ»
2024

УДК 004.45
П 16

Рецензенти:
Толстолузька О.Г., д-р техн. наук, проф. ХНУ ім. В. Н. Каразіна*;*
Фесенко Г.В., д-р техн. наук, проф. НАУ ім. М.Є. Жуковського «ХАІ»

Автори:
В. І. Панченко, ст.викл.;
О. В. Коломійцев, д-р техн. наук, проф.;
С. Г. Межерицький, ст.викл.

Панченко В. І.

П 16 Системне програмне забезпечення. Програмування системних механізмів ОС: навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання / В. І. Панченко, О. В. Коломійцев, С. Г. Межерицький. – Харків: НТУ «ХПІ», 2024. – 327 с.

ISBN

Розглянуто основні теоретичні та практичні відомості в галузі системного програмування в операційних системах Windows та Linux: питання професійної розробки програм за допомогою інтерфейсу прикладного програмування. Особлива увага приділяється керуванню процесами, файловою системою, організації взаємодії процесів та оптимізації розроблених програм. Велика кількість прикладів дозволяє легко отримати необхідні знання та практичні навички в створенні високоякісних системних програм.

Призначено для студентів спеціальності 123 «Комп'ютерна інженерія» денної та заочної форм навчання за навчальною дисципліною «Системне програмне забезпечення», а також може бути корисним при курсовому та дипломному проєктуванні.

Іл. 16. Табл. 7. Бібліогр.: 19 назв.

УДК 004.45

ISBN

ЗМІСТ

ВСТУП

Професійна розробка якісних, надійних, ефективних програм є однією з вимог до системних програмістів. Швидка розробка програм може бути виконана за допомогою безлічі інструментів, які існують на сучасному етапі розвитку інструментального програмного забезпечення. Але найбільш ефективними з точки зору швидкодії та розміру програм є використання механізмів, які дозволяють працювати з операційною системою (ОС) безпосередньо, використовуючи той функціонал, який надає сама ОС.

В операційних системах, як відомо, можна виділити два режими роботи: режим ядра (kernel mode) – привілейований режим, який використовується ядром операційної системи, та режим користувача (user mode) – режим, в якому виконується більшість програм користувача. Користувачі при повсякденній роботі зазвичай використовують утиліти командного рядка та інтерфейс користувача (UI – User Interface). Але безпосередньо взаємодія з ядром операційної системою виконується за допомогою системних викликів. Системний виклик (system call) – це механізм взаємодії програм користувача з ядром. Системні виклики дуже схожі на виклики функцій тому, що в них передаються аргументи і вони повертають значення. Єдина відмінність полягає в тому, що системні виклики реалізують звернення до процедур, які працюють лише на рівні ядра, а функції – ні. Для кожної ОС характерним є свій набір системних викликів, який являє собою сукупність системних викликів і правил їх застосування. Цей набір має назву API (Application Programming Interface), за допомогою якого організується доступ до внутрішніх програмних механізмів операційної системи.

Для прикладного програміста ОС виглядає як деяка бібліотека, що реалізує функції, які полегшують керування прикладним завданням або виконання дій, заборонених у користувальницькому режимі.

Робота через API – це найбільш близький до операційної системи спосіб взаємодії з нею із прикладних програм. Функції API, що обслуговують системні виклики, надають доступ до ресурсів системи в зручній компактній формі, без зазначення деталей їх фізичної реалізації. Програмування з використанням засобів API має ряд переваг по зрівнянню з іншими механізмами: легко розробляти багатониткові (multithreaded) програми; інтерфейс прикладного програмування є загальним для всіх апаратних платформ, на яких може функціонувати ОС.

Звісно, що розробити прикладну програму з використанням сторонніх бібліотек швидше та простіше, але не можна досягти тієї ж ефективності, яка може бути отримана при використанні тільки API. При програмуванні одних і тих же дій сучасними засобами програмування та засобами API розмір тексту програми мовою високого рівня буде в другому випадку значно більшим: всі кроки алгоритму взаємодії з операційною системою необхідно реалізувати власноруч (в тому числі перевірку на помилки). Але розмір тексту програми компенсується його високою ефективністю та великими можливостями по використанню всіх особливостей та тонкому налаштуванню програм.

У даному навчальному посібнику розглянуті базові теоретичні відомості щодо розробки системного програмного забезпечення для операційних систем Windows та Linux. Кожна тема проілюстрована текстами програм.

В якості інструментарію для операційної системи Windows було обрано мову програмування C++ та середовище розробки програм – Microsoft Visual Studio (тексти програм прикладів за темами 1–13 були розроблені та протестовані в середовищі Microsoft Visual Studio 2019 16.9 в ОС Windows 10). Розглянуті основні механізми по роботі з процесами, по створенню графічного інтерфейсу користувача, по використанню засобів синхронізації та обміну даними, по роботі з файловою системою та реєстром. Також приділена увага питанням оптимізації програм.

Для операційної системи Linux використано мову програмування С (тексти програм прикладів за темами 13–21 були розроблені та протестовані в ОС Ubuntu 22.04.4 LTS, компілятор – cc). Розглянуті основні механізми по роботі з процесами, по використанню засобів синхронізації та обміну даними.

Розглянуті в темах системні механізми дозволяють отримати практичні навички системного програмування в операційних системах Windows та Linux та виступають в ролі практичного закріплення матеріалу, що викладається в дисципліні «Системне програмне забезпечення». Після кожної теми наведені контрольні питання для закріплення теоретичного матеріалу. До розглянутих тем наведений загальний перелік рекомендованої літератури. Також для вивчення матеріалів та розробки програм може знадобитися ресурс https://learn.microsoft.com/en-us/windows/win32/api/ («Довідник з програмування для Win32 API») та https://manpages.ubuntu.com/ (підрозділи допомоги – «System calls»).

Розглянуті теми використовуються на практичних заняттях та в лабораторному практикумі за дисципліною «Системне програмне забезпечення – по кожній темі сформовано ряд індивідуальних практичних завдань, необхідних для закріплення матеріалу. Оформлення звітів виконується згідно загальноприйнятих вимог.

1. ПРОЦЕСИ, НИТКИ ТА ВОЛОКНА В ОС WINDOWS

Мета: вивчити можливості по створенню, керуванню виконанням та завершенню процесів, ниток та волокон в операційній системі Windows.

1.1. Теоретичні відомості

В ОС Windows підтримуються процеси, що складаються з кількох ниток. Процес – це об'єкт ядра, який має в своєму розпорядженні деякий набір ресурсів. Нитка (також об'єкт ядра) – незалежна частина процесу, що виконується, при цьому вона розділяє з процесом спільний адресний простір, код та змінні. Волокно – спрощена нитка, виконання якої планується в програмі вручну. Розглянемо ці об'єкти докладніше.

Процеси.

Процесом зазвичай називають екземпляр програми, що виконується. В деяких джерелах інформації не розрізнюють поняття програми і процесу. Але вони фундаментально відрізняються один від одного. Програма є статичним набором команд, а процес – це набір ресурсів і даних, що використовуються при виконанні програми. Процес (Process) в Windows складається з наступних компонентів:

- структура даних, що містить всю інформацію про процес, у тому числі список відкритих дескрипторів різних системних ресурсів, унікальний ідентифікатор процесу, різну статистичну інформацію та інше;
- адресний простір – діапазон адрес віртуальної пам'яті, яким може користуватися процес;
- виконувана програма і дані, що проєктуються на віртуальний адресний простір процесу.

Створення процесу в системі Windows здійснюється завдяки виклику однієї з таких функцій, як **CreateProcess**, **CreateProcessAsUser** (для ОС WinNT/2000) або **CreateProcessWithLogonW** (починаючи з ОС Win2000), і відбувається в декілька етапів:

- відкривається файл образу (EXE файл), який буде виконуватись у процесі. Якщо виконуваний файл не є Windows прикладною програмою, то

відбувається пошук образу підтримки (support image) для запуску цієї програми. Наприклад, якщо виконується файл з розширенням BAT, то запускається командний процесор, а саме програма **cmd.exe**. *Примітка*: починаючи з ОС WinNT/2000 реалізований наступний механізм для завантаження програм: функція **CreateProcess** після того, як знайде виконуваний Windows файл, шукає в системному реєстрі за маршрутом **HKEY_LOCAL-MACHINE \ SOFTWARE \ Microsoft \ WindowsNT \ CurrentVersion \ Image File Execution Option** розділ з ім'ям і розширенням файла, який запускається. Потім у ньому вказана функція шукає параметр **Debugger** і, якщо рядок не пустий, запускає те, що в ньому записане, замість даної програми;

- створюється об'єкт ядра «процес»;
- створюється первинна нитка (стек, контекст і об'єкт ядра «нитка»);
- підсистема Windows сповіщається про створення нових процесу і нитки;
- починається виконання первинної нитки;
- у контексті нового процесу і нитки ініціалізується адресний простір (наприклад, завантажуються необхідні бібліотеки DLL) і починається виконання програми.

Параметри виклику функції **CreateProcess** наступні:

```
BOOL CreateProcess (
 LPCTSTR lpApplicationName, // ім'я файла з програмою
 LPTSTR lpCommandLine, // командний рядок виклику
 LPSECURITY_ATTRIBUTES lpProcessAttributes,// покажчик
      // на структуру атрибутів захисту процесу
 LPSECURITY_ATTRIBUTES lpThreadAttributes, // покажчик
      // на структуру атрибутів захисту нитки
 BOOL bInheritHandles, // ознака наслідування
      // дескрипторів процесу, що викликав функцію
 DWORD dwCreationFlags, // прапорці створення процесу
      // та призначення йому пріоритету
 LPVOID lpEnvironment, //покажчик на змінні середовища
```

```
 LPCTSTR lpCurrentDirectory, //ім'я поточного каталогу
 LPSTARTUPINFO lpStartupInfo, // покажчик на структуру
      // з параметрами вікна
 LPPROCESS_INFORMATION lpProcessInformation //покажчик
      // на структуру з даними про створений процес
);
```

Функції **CreateProcessAsUser** та **CreateProcessWithLogonW** дозволяють створити процес у контексті захисту користувача, це задається одним з параметрів виклику. Функція **CreateProcessAsUser** потребує виклику функції входу користувача до системи **LogonUser**.

Процес завершується, якщо:

- вхідна функція первинної нитки повернула управління;
- одна з ниток процесу викликала функцію **ExitProcess**;
- нитка іншого процесу викликала функцію **TerminateProcess**.

Коли процес завершується, всі User- і GDI-об'єкти, що були створені процесом, знищуються, об'єкти ядра закриваються (якщо їх не використовує інший процес), адресний простір процесу також знищується.

Параметри виклику вказаних функцій:

```
VOID ExitProcess (
 UINT uExitCode   // код виходу для всіх ниток процесу
);

BOOL TerminateProcess(
 HANDLE hProcess, // дескриптор процесу
 UINT uExitCode   // код виходу для процесу
);
```

Приклад 1.1. Програма створює процес «**Notepad**».

Примітка: при створенні процесу функцією **CreateProcess** параметр `lpCommandLine` повинен містити адресу рядка символів, а не константу

(при виконанні функції командний рядок виклику процесу змінюється, а при виході з функції – відновлюється).

Слід прийняти до уваги, що всі консольні та віконні проєкти, які наводяться далі, необхідно створювати з налаштуванням «**Проєкт → Властивості → Властивості Конфігурації → Додатково → Набір Символів → Не Задано**». Також необхідно перевірити та в разі необхідності встановити налаштування «**Проєкт → Властивості → C/C++ → Попередньо Відкомпільовані Заголовки → Не Використовувати…**».

```
#include <windows.h>
int main (int argc, char* argv[])
{
      STARTUPINFO si;
      PROCESS_INFORMATION pi;
      LPTSTR lpszSystemInfo;
      TCHAR tchBuff[MAX_PATH+1];
      lpszSystemInfo = tchBuff;
      GetSystemDirectory (lpszSystemInfo,MAX_PATH+1);
      wsprintf(tchBuff, "%s\\notepad.exe",
                  lpszSystemInfo);
      ZeroMemory (&si, sizeof(si));
      si.cb = sizeof(si);
      ZeroMemory( &pi, sizeof(pi) );
      if (!CreateProcess (NULL, tchBuff, NULL, NULL,
            FALSE,0, NULL, NULL, &si, &pi))
                  return 0;
      // Закриття дескриптора процесу та основної
      // нитки
      CloseHandle( pi.hProcess );
      CloseHandle( pi.hThread );
      return 0;
}
```

Нитки.

Кожна нитка має:

▪ унікальний ідентифікатор нитки;

▪ вміст сукупності регістрів процесора, що відображають стан процесора взагалі;

▪ два стека, один із котрих використовується ниткою при її виконанні в режимі ядра, а другий – у режимі користувача;

▪ закриту область пам'яті, яка називається локальною пам'яттю нитки (TLS – Thread Local Storage) і яку використовують підсистеми ОС, run-time бібліотеки та бібліотеки DLL (Dynamic Link Library).

Щоб усі нитки, створені в системі, виконувались, ОС відводить кожній з них визначений інтервал процесорного часу. Тим самим створюється ілюзія одночасного виконання ниток (зрозуміло, що для багатопроцесорних комп'ютерів можливий дійсний паралелізм). В ОС Windows реалізована система планування з витісненням на основі пріоритетів, в якій завжди виконується нитка, що готова до виконання та має найбільший пріоритет. Обрана для виконання нитка виконується на протязі деякого проміжку часу, який називається квантом. Квант визначає, скільки часу буде виконуватись нитка, поки ОС не перерве її виконання. По закінченні кванта ОС перевіряє, чи готова до виконання інша нитка з таким же (або більшим) рівнем пріоритету. Якщо такі нитки не виявлені, поточній нитці виділяється ще один квант. З іншого боку, нитка може не повністю використати свій квант. Як тільки інша нитка з більш високим пріоритетом стане готовою до виконання, поточна нитка витісняється, навіть якщо її квант ще не вичерпаний.

Квант не вимірюється в якихось одиницях часу, він виражається цілим числом. Для кожної нитки зберігається поточне значення її кванта. Коли нитці виділяється квант процесорного часу, це означає, що її квант установлюється в початкове значення. Воно залежить від ОС. Наприклад, для Windows 2000 Professional початкове значення кванта дорівнює 6, а для Windows 2000 Server – 36.

Кожен раз, коли виникає переривання від таймера, із кванта нитки віднімається число 3, і так до тих пір, поки він не досягне нульового зна-

чення. Частота спрацьовування таймера залежить від апаратної платформи. Наприклад, для більшості однопроцесорних x86 систем вона становить 10 мс, а на більшості багатопроцесорних x86 систем – 15 мс.

У будь якому випадку ОС повинна визначити, яку нитку виконувати наступною. Обравши нову нитку, ОС переключає контекст. Ця операція полягає в тому, що зберігаються параметри нитки, яка щойно виконувалась (регістри процесора, покажчики на стеки ядра і користувача, покажчик на адресний простір, в якому виконувалась вказана нитка і інше), і завантажуються аналогічні параметри для другої нитки. Після цього починається виконання нової нитки.

Планування в ОС Windows здійснюється на рівні ниток, а не процесів. Це і зрозуміло, тому що самі процеси не виконуються, а лише виділяють ресурси і контекст для виконання ниток. Тому при плануванні ниток система не звертає увагу на те, якому процесу вони належать. Наприклад, якщо процес А має 10 готових до виконання ниток, а процес Б – дві, і всі 12 ниток мають однаковий пріоритет, то кожна з них отримає 1/12 процесорного часу.

В ОС Windows існує 32 рівня пріоритету – від 0 до 31. Вони групуються так: 31–16 – рівні реального часу; 15–1 – динамічні рівні; 0 – системний рівень, що зарезервований для процесу обнуління сторінок (zero-page thread).

Пріоритет кожної нитки (базовий пріоритет нитки) складається із пріоритету її процесу і відносного пріоритету самої нитки. Є сім відносних пріоритетів ниток:

- normal – такий же, як і в процесі;
- above normal – +1 до пріоритету процесу;
- below normal – –1;
- highest – +2;
- lowest – –2;
- time critical – встановлює числове значення базового пріоритету нитки для Real time класу в 31, для решти класів – 15;
- idle – встановлює числове значення базового пріоритету нитки для Real time класу в 16, для решти класів – 1.

Якщо ОС функціонує на машині, де розміщено більше одного процесора, то, за замовчуванням, нитка виконується на будь якому доступному процесорі. Але в деяких випадках набір процесорів, на яких нитка може виконуватись, може бути обмеженим. Цю ситуацію називають прив'язкою до процесорів (processor affinity). Можна змінити прив'язку до процесорів програмно, через WinAPI функції планування **SetProcessAffinityMask**, **SetThreadAffinityMask** (для прив'язування процесу та нитки до конкретного процесору) та **SetThreadIdealProcessor** (для встановлення бажаного процесору для нитки).

Для роботи з нитками використовуються наступні функції (перераховані найбільш часто використовувані): **CreateRemoteThread** (створення нитки в адресному просторі іншого процесу), **CreateThread** (створення нитки в адресному просторі поточного процесу), **ExitThread** (завершення роботи поточної нитки), **GetCurrentThread** (отримання псевдодескриптора поточної нитки), **GetCurrentThreadId** (отримання ідентифікатора поточної нитки), **GetExitCodeThread** (отримання коду завершення нитки), **ResumeThread** (зменшення лічильника зупинки нитки, коли він дорівнює 0 – нитка продовжує виконання), **SuspendThread** (збільшення лічильника зупинки нитки, коли він більше 0 – виконання нитки призупиняється), **SwitchToThread** (звільнення процесора поточною ниткою та переключення на іншу), **TerminateThread** (примусове завершення нитки). Для створення нитки в адресному просторі поточного процесу у функції використовуються такі параметри:

```
HANDLE CreateThread (
 LPSECURITY_ATTRIBUTES lpThreadAttributes, // покажчик
      // на структуру атрибутів захисту
 SIZE_T dwStackSize, // розмір стеку для нитки,
      // 0 – задати розмір стеку як в основної нитки
 LPTHREAD_START_ROUTINE lpStartAddress, // покажчик на
      // функцію нитки
 LPVOID lpParameter, // параметр, що передається нитці
 DWORD dwCreationFlags, // параметр створення, якщо
```

```
      // він містить CREATE_SUSPENDED – нитка
      // створюється в призупиненому стані
 LPDWORD lpThreadId // ідентифікатор нитки, для
      // Windows NT/2000 та вище – може бути NULL
);
```

Волокна.

Волокна підтримуються у WinAPI починаючи з Windows 2000. Як було сказано вище: волокно – це спрощена нитка, виконання якої планується самою прикладною програмою, а не планувальником процесорного часу ОС. Планування волокон може здійснюватися шляхом переключення на них тільки з інших волокон. Волокна виконуються в контексті ниток, в яких планується їх використання, і допускають повну їх ідентифікацію з нитками. У кожній нитці може бути заплановано декілька волокон. Для кожного волокна створюється власний стек, в якому зберігається інформація про стан волокна.

Волокно може бути створене за допомогою функції **CreateFiber** із основної нитки процесу або отримане шляхом перетворення у волокно поточної нитки за допомогою функції **ConvertThreadToFiber**. Переключення поміж волокнами може бути організоване за допомогою функції **SwitchToFiber**, але її виклик можна здійснювати тільки із волокна.

Параметри вказаних функцій:

```
LPVOID CreateFiber (
 SIZE_T dwStackSize, // розмір стеку для волокна,
      // 0 – задати розмір стеку як в основної нитки
 LPFIBER_START_ROUTINE lpStartAddress, // покажчик на
      // функцію волокна
 LPVOID lpParameter //параметр, що передається волокну
);

LPVOID ConvertThreadToFiber (
 LPVOID lpParameter //параметр, що передається волокну
```

```
);

VOID SwitchToFiber (
 LPVOID lpFiber  // адреса волокна, яке запускається
);
```

Для отримання параметру, що передається у волокно, використовується функція **GetFiberData**. Для зворотного перетворення волокна на нитку використовується функція **ConvertFiberToThread**.

Приклад 1.2. Програма створює 4 волокна, кожне з яких виконує переключення на наступне волокно. Якщо число переключень більше 10, робота програми завершується.

```
// задаємо мінімальну версію ОС – Windows 2000
#define _WIN32_WINNT 0x0500
#define WINVER 0x0500
#include <windows.h>
#include <stdio.h>
#include <conio.h>
int Counter;
void* fiber[5];

void WINAPI Func(void*)
{
  for (;;) {
        printf("ITERATION %d, Fiber Number %d\n",
               Counter,  Counter % 4);
        _getch();
        if ((++Counter) < 10) {
              SwitchToFiber(fiber[Counter % 4]);
        }
        else break;
```

```
    }
}

int main(int argc, char* argv[])
{
    Counter = 0;
    fiber[0] = CreateFiber(0, Func, NULL);
    fiber[1] = CreateFiber(0, Func, NULL);
    fiber[2] = CreateFiber(0, Func, NULL);
    fiber[3] = CreateFiber(0, Func, NULL);
    // для перемикання на перше волокно необхідно
    // перетворити поточну нитку у волокно
    fiber[4] = ConvertThreadToFiber(NULL);
    SwitchToFiber(fiber[0]);
    return 0;
}
```

Для примусового знищення волокон застосовується функція **DeleteFiber**. Також для знищення волокон можуть бути використані всі функції, які призначені для знищення ниток. Параметри функції знищення волокна:

```
VOID DeleteFiber (
 LPVOID lpFiber   // покажчик на волокно для знищення
);
```

Функції очікування (wait-функції).

Призупиняти виконання ниток можна різними засобами.

Функція **Sleep** призупиняє виконання нитки на задане число мілісекунд. Якщо в якості аргументу даної функції вказати 0, то нитка відмовиться від свого кванта процесорного часу, але негайно з’явиться в списку ниток, готових до виконання. Іншими словами, відбудеться навмисне перек-

лючення ниток (вірніше сказати, спроба переключення – адже наступною для виконання ниткою цілком імовірно може стати та ж сама).

Приклад 1.3. Програма створює процес «**Notepad**» і через 15 секунд його знищує.

```
#include <windows.h>
int main(int argc, char* argv[])
{
      STARTUPINFO StartUpInfo;
      PROCESS_INFORMATION ProcessInfo;
      LPTSTR lpszSystemInfo;
      TCHAR tchBuff[MAX_PATH+1];
      lpszSystemInfo = tchBuff;
      GetSystemDirectory (lpszSystemInfo, MAX_PATH+1);
      wsprintf(tchBuff,"%s\\notepad.exe",
            lpszSystemInfo);
      memset(&StartUpInfo,0,sizeof(STARTUPINFO));
      StartUpInfo.cb = sizeof(STARTUPINFO);
      if (CreateProcess (NULL, lpszSystemInfo, NULL,
            NULL, FALSE,NORMAL_PRIORITY_CLASS,NULL,
            NULL,&StartUpInfo,&ProcessInfo))
            {
            Sleep (15000);
            TerminateProcess(ProcessInfo.hProcess, 0);
            }
      ExitProcess (0);
}
```

Функція **WaitForSingleObject** призупиняє виконання нитки до тих пір, поки не відбудеться одна із двох подій:

- закінчиться час очікування;
- очікуваний об’єкт перейде в сигнальний (signaled) стан.

За значенням, яке повертає ця функція, можна зрозуміти, яка з двох подій мала місце. Очікувати події за допомогою wait-функцій можна відносно більшості об'єктів ядра, наприклад, щодо об'єктів «процес» або «нитка» – щоб визначити, коли вони завершлять свою роботу. У сигнальний стан об'єкти переходять у наступних ситуаціях:

- сповіщення (Change notification) – при змінах у файловій системі;
- введення з консолі (Console input) – при готовності введення з консолі;
- подія (Event) – при виклику функцій **SetEvent** або **PulseEvent**;
- завдання (Job) – при завершенні;
- м'ютекс (Mutex) – якщо він не належить жодній нитці;
- процес (Process) – при завершенні;
- семафор (Semaphore) – при значенні більше 0;
- нитка (Thread) – при завершенні;
- таймер (Waitable timer) – при настанні вказаного часу.

Функції **WaitForMultipleObjects** передається масив об'єктів. Можна очікувати спрацювання зразу всіх об'єктів або якогось одного з них.

Параметри вказаних функцій:

```
DWORD WaitForSingleObject (
 HANDLE hHandle,        // дескриптор об'єкту
 DWORD dwMilliseconds   // інтервал очікування
);

DWORD WaitForMultipleObjects (
 DWORD nCount, // кількість дескрипторів у масиві
 CONST HANDLE *lpHandles, // масив дескрипторів
                          // об'єктів
 BOOL bWaitAll,     // параметр чекання (всі або один)
 DWORD dwMilliseconds    // інтервал очікування
);
```

Приклад 1.4. Програма створює дві однакові нитки і очікує їх завершення. Нитки просто виводять текстове повідомлення, яке було передане їм при їх ініціалізації. Порядок виведення повідомлення від першої та другої нитки зумовлюється порядком їх виконання та не залежить від порядку створення.

```
#include <windows.h>

unsigned ThreadFunc(void* arg)
{
      char** str = (char**)arg;
      MessageBox(0, str[0], str[1], 0);
      ExitThread(0);
      return 0;
}
int main(int argc, char* argv[])
{
const char* InitStr1[2] = { "First thread","11111" };
const char* InitStr2[2] = { "Second thread","22222" };
      unsigned long uThreadIDs[2];
      HANDLE hThreads[2];
      hThreads[0] = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)ThreadFunc,
            InitStr1, 0, &uThreadIDs[0]);
      hThreads[1] = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)ThreadFunc,
            InitStr2, 0, &uThreadIDs[1]);
      // Очікування завершення роботи ниток
      WaitForMultipleObjects(2, hThreads, TRUE,
             INFINITE);
      // Закриття дескрипторів
      CloseHandle(hThreads[0]);
      CloseHandle(hThreads[1]);
```

```
        return 0;
}
```

1.2. Контрольні питання

1. Для чого використовується виклик функції **wsprintf** у прикладі 1.1?

2. Для чого використовуються при створенні процесів функції `ZeroMemory` в прикладі 1.1?

3. Навіщо закривати дескриптори перед виходом з програми в прикладі 1.1?

4. Скільки волокон створюється в прикладі 1.2?

5. Скільки разів запускається на виконання кожне волокно в прикладі 1.2?

6. Скільки разів передається управління на кожне волокно в прикладі 1.2?

7. В який стан переходить процес після виклику функції **Sleep** у прикладі 1.3?

8. Яка структура даних використовується для можливості управління створеним процесом у прикладі 1.3?

9. Як перевірити, чи було вдало створено процес у прикладі 1.3?

10. Спробуйте запустити програму в прикладі 1.4 кілька разів. Чи змінюється порядок відображення **MessageBox**? Чому?

11. Скільки ниток створюється в програмі в прикладі 1.4? Скільки з них виконуються паралельно?

12. Чим відрізняється код у створених нитках у прикладі 1.4?

1.3. Індивідуальні завдання

Розробити консольну програму в середовищі програмування Visual C++, яка виконує такі дії: виконує циклічне перемикання між волокнами за заданим алгоритмом. Як керуючу частину використовувати головне волокно, отримане з основної нитки процесу. При натисканні на клавішу 0 на клавіатурі програма повинна завершувати своє виконання (для очікування натискання необхідно використовувати окрему нитку).

Формат елемента циклічної послідовності: N(M), де N – номер волокна, M – час виконання волокна в секундах (організувати як затримку). Кожне волокно має виводити на екран свою ідентифікацію (наприклад, «ВИКОНАННЯ 1-го ВОЛОКНА 5 СЕКУНД»).

1) 1(5) → 3(2) → 1(3) → 2(5) → 3(3)
2) 1(2) → 3(2) → 5(2) → 1(3) → 2(3) → 4(3)
3) 1(1) → 2(2) → 3(3) → 3(1) → 2(2) → 1(3)
4) 1(1) → 1(2) → 1(3) → 2(1) → 2(2) → 2(3)
5) 1(1) → 2(2) → 1(1) → 2(2) → 3(1) → 3(2)
6) 1(2) → 2(3) → 2(3) → 1(3) → 1(3) → 2(4)
7) 1(2) → 2(2) → 1(2) → 3(3) → 1(3) → 4(4)
8) 1(2) → 2(2) → 1(2) → 1(2) → 3(3) → 3(3) → 3(3)
9) 1(2) → 2(2) → 1(2) → 1(2) → 3(3) → 3(3) → 3(3)
10) 1(4) → 2(3) → 3(2) → 4(1) → 1(1) → 2(2) → 3(3) → 4(4)
11) 1(4) → 2(3) → 3(2) → 4(1) → 4(1)
12) 1(2) → 2(2) → (1(2) парний виклик → 2(3) непарний виклик)
13) 1(2) → 2(2) → 1(3) → 2(2) → 1(4) →2(2)
14) 1(2) → 2(1) → 1(1) → 2(2)
15) 1(3) → 3(2) → 2(1) → 1(1) → 3(2) → 2(3)

2. ЗАВДАННЯ (JOB) В ОС WINDOWS

Мета: вивчити можливості по створенню, керуванню виконанням та завершенню процесів за допомогою механізму завдань в ОС Windows.

2.1. Теоретичні відомості

Завдання (Job) – об'єкти ядра ОС, що виконують роль контейнерів процесів. Завдання забезпечують керування одним або декількома процесами як однією групою. Процес може входити одночасно тільки до одного завдання (актуально для Windows версій до 10) і видалити процес із завдання неможливо. Всі процеси, які будуть запущені з процесу, що вже входить до завдання, також будуть входити до того ж завдання. За допомогою завдань можна встановлювати цілий ряд обмежень на виконання процесів, що входять до завдання, при чому більшість обмежень не доступна при звичайному керуванні процесами. Саме тому для розширеного керування яким-небудь процесом доцільно створювати завдання, куди цей процес необхідно підключити.

Для роботи з завданнями використовуються такі функції.

AssignProcessToJobObject – додавання процесу з дескриптором `hProcess` в існуюче завдання з дескриптором `hJob`:

```
BOOL AssignProcessToJobObject (
 HANDLE hJob,     // дескриптор завдання
 HANDLE hProcess  // дескриптор процесу
);
```

CreateJobObject – створення об'єкта-завдання з ім'ям `lpName`.

```
HANDLE CreateJobObject (
 LPSECURITY_ATTRIBUTES lpJobAttributes,   // інформація
                                          // про захист
 LPCTSTR lpName       // необов'язкове ім'я завдання
);
```

IsProcessInJob – перевірка, чи входить процес з дескриптором `ProcessHandle` у завдання з дескриптором `JobHandle`.

```
BOOL IsProcessInJob (
 HANDLE ProcessHandle, // дескриптор процесу
 HANDLE JobHandle,     // дескриптор завдання
 PBOOL Result          // покажчик на результат
);
```

Відкриття існуючого об'єкта–завдання з іменем `lpName` виконує функція **OpenJobObject**.

```
HANDLE OpenJobObject (
 DWORD dwDesiredAccess,  // права доступу
 BOOL bInheritHandles,   // дозвіл наслідування
 LPCTSTR lpName          // ім'я завдання
);
```

Функція **QueryInformationJobObject** дозволяє отримати інформацію про завдання з дескриптором `hJob`, а саме: обсяг використаного процесорного часу, лічильник кількості сторінкових відмов, перелік ідентифікаторів процесів, обмеження захисту. У полі `JobObjectInfoClass` задається тип інформації, а в полі `lpJobObjectInfo` покажчик на відповідну структуру.

```
BOOL QueryInformationJobObject (
 HANDLE hJob,     // дескриптор завдання
 JOBOBJECTINFOCLASS JobObjectInfoClass, // клас
                          // інформації про завдання
 LPVOID lpJobObjectInfo, // інформація про обмеження
                         // для вказаного класу
 DWORD cbJobObjectInfoLength, // розмір отриманої
```

```
                                // інформації
 LPDWORD lpReturnLength       // розмір даних, що
                 // записані в структурі, на яку
                 // посилається lpJobObjectInfo
);
```

Встановити обмеження на процеси, що належать завданню з дескриптором `hJob`, дозволяє функція **SetInformationJobObject**, призначення її параметрів таке саме, як і у функції **QueryInformationJobObject**.

```
BOOL SetInformationJobObject (
 HANDLE hJob,     // дескриптор завдання
 JOBOBJECTINFOCLASS JobObjectInfoClass, // клас
                          // інформації про завдання
 LPVOID lpJobObjectInfo // інформація про обмеження
                         // для вказаного класу
 DWORD cbJobObjectInfoLength // розмір отриманої
                                // інформації
);
```

За допомогою функцій **QueryInformationJobObject** та **SetInformationJobObject** можна отримувати або вносити обмеження, що перераховані нижче (формат структур можна отримати у відповідних розділах MSDN).

1) Базові обмеження (значення параметра функцій `JobObjectInfoClass = JobObjectBasicLimitInformation`, `lpJobObjectInfo` вказує на структуру JOBOBJECT_BASIC_LIMIT_INFORMATION або `JobObjectInfoClass = JobObjectExtendedLimitInformation`, `lpJobObjectInfo` вказує на структуру JOBOBJECT_EXTENDED_LIMIT_INFORMATION).

▪ Максимальна кількість активних процесів – обмеження на кількість процесів, що одночасно виконуються.

▪ Загальне обмеження на процесорний час у режимі користувача – обмеження процесорного часу на всі процеси, що входять до завдання, якщо час буде вичерпано – всі процеси будуть завершені примусово.

▪ Індивідуальне обмеження на процесорний час у режимі користувача – обмеження процесорного часу для кожного процесу окремо. У разі використання всього часу – процес буде примусово завершений.

▪ Клас планування завдання – встановлюється довжина кванту часу для ниток процесів, які входять до завдання.

▪ Прив'язування до процесорів – для кожного процесу встановлюється маска прив'язування до процесору (для багатопроцесорних комп'ютерів).

▪ Клас пріоритету для всіх процесів – обмеження максимального пріоритету для кожного процесу. Якщо нитка спробує підвищити свій пріоритет – він останеться без зміни, але помилки виконання операції не буде.

▪ Мінімальний та максимальний розміри робочого набору – для кожного процесу встановлюються обмеження на робочий набір, тобто на кількість сторінок, що постійно знаходяться в оперативній пам'яті. Розмір задається в байтах.

▪ Обмеження віртуальної пам'яті – вказується максимальний розмір віртуального адресного простору, який можна передати процесу або всьому завданню.

2) Базові обмеження інтерфейсу користувача (значення параметру функцій `JobObjectInfoClass = JobObjectBasicUIRestrictions`, `lpJobObjectInfo` вказує на структуру JOBOBJECT_BASIC_UI_RESTRICTIONS).

▪ Заборона роботи з буфером обміну – обмеження читання та очищення буфера обміну.

▪ Заборона завершення роботи системи – забороняється вихід з системи або завершення роботи всієї системи через функцію **ExitWindowsEx**.

▪ Заборона зміни системних параметрів – кожному процесу забороняється змінювати параметри системи за допомогою функції **SystemParametersInfo**. До системних параметрів відносяться: параметри робочого столу, пристрою введення, системного меню, вікон програм, живлення та програми збереження екрану та ін.

▪ Заборона зміни параметрів екрану – забороняється зміна параметрів за допомогою функції **ChangeDisplaySettings**.

▪ Заборона створення та перемикання робочих столів – забороняється використання функцій **CreateDesktop** та **SwitchDesktop**.

▪ Заборона користування USER-об'єктами, що не входять у завдання – процеси в завданні не можуть звернутися до будь-яких USER-об'єктів, що створені зовнішніми до завдання процесами. Наприклад, неможливо отримати HWND дескриптор процесу, який не входить у завдання.

3) Базові обмеження захисту (значення параметру функцій `JobObjectInfoClass= JobObjectSecurityLimitInformation`, `lpJobObjectInfo` вказує на структуру JOBOBJECT_SECURITY_LIMIT_INFORMATION). *Примітка*: якщо в завданні ввести такі обмеження, скасувати їх буде неможливо.

▪ Заборона доступу адміністратору.

▪ Заборона маркера необмеженого доступу.

▪ Блокування доступу – блокується доступ по ідентифікаторам захисту (Security ID).

Якщо задати у функції **QueryInformationJobObject** у параметрі `JobObjectInfoClass` значення `JobObjectBasicAccountingInformation` або `JobObjectBasicAndIoAccountingInformation` або `JobObjectBasicProcessIdList`, можна отримати статистичну інформацію про завдання, а саме: у першому випадку – про кількість процесів у завданні, процесорний час, кількість сторінкових відмов; у другому випадку – про кількість операцій читання, запису та про кількість переданих байтів; у третьому випадку – отримання переліку ідентифікаторів процесів, що входять до завдання. У параметрі `lpJobObjectInfo` відповідно повинен бути покажчик на структуру JOBOBJECT_BASIC_ACCOUNTING_INFORMATION або JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION або JOBOBJECT_BASIC_PROCESS_ID_LIST.

Завершення всіх процесів у завданні з ідентифікатором `hJob` може виконуватися за допомогою функції **TerminateJobObject**:

```
BOOL TerminateJobObject (
 HANDLE hJob,    // дескриптор завдання
 UINT uExitCode  // код завершення
);
```

Приклад 2.1. У програмі, що наведена нижче, у змінну `lpszSystemInfo` заноситься шлях до системного каталогу Windows, який повертається функцією **GetSystemDirectory**, та додається до нього ім'я програми **notepad.exe**. Далі створюється завдання `MyJob` з дескриптором `hJob`, для нього встановлюється обмеження (з групи базових обмежень) на кількість активних процесів (4) та виконується спроба запустити та додати 10 процесів з програми **notepad.exe**. Далі програма очікує 15 секунд, та примусово завершує всі процесі в завданні. Як результат – буде створено 10 процесів, але одразу при додаванні їх у завдання, тільки перші 4 будуть активними, інші будуть примусово завершені. *Примітка*: перший рядок програми вказує, що програма призначена для роботи в ОС Windows 2000 або вище, так як підтримка завдань у системах Windows 95, 98 відсутня.

```
#define _WIN32_WINNT 0x500
#include <windows.h>
#include <stdio.h>
HANDLE hJob;
LPTSTR lpszSystemInfo;
TCHAR tchBuff[MAX_PATH + 1];
STARTUPINFO si;
PROCESS_INFORMATION pi;
int AddProcess2Job(void)
{
      int i;
      for (i = 0; i < 10; i++) {
            ZeroMemory(&si, sizeof(si));
```

```
            si.cb = sizeof(si);
            ZeroMemory(&pi, sizeof(pi));
            if (!CreateProcess(NULL, tchBuff, NULL,
                  NULL,FALSE,CREATE_BREAKAWAY_FROM_JOB
                   | CREATE_SUSPENDED, NULL, NULL,
                   &si, &pi))
            return 0;

      if (!AssignProcessToJobObject(hJob,pi.hProcess))
      {
      TerminateProcess(pi.hProcess, 0);
      printf("ERROR: AssignProcessToJobObject %i\n",
             i);
      }
      else ResumeThread(pi.hThread);
      }
}

int main(int argc, char* argv[])
{
JOBOBJECT_EXTENDED_LIMIT_INFORMATION jInfo;

lpszSystemInfo = tchBuff;
GetSystemDirectory(lpszSystemInfo, MAX_PATH + 1);
wsprintf(tchBuff, "%s\\notepad.exe", lpszSystemInfo);
hJob = CreateJobObject(NULL, "MyJob");
jInfo.BasicLimitInformation.LimitFlags =
     JOB_OBJECT_LIMIT_ACTIVE_PROCESS |
     JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK |
     JOB_OBJECT_LIMIT_KILL_ON_JOB_CLOSE;
     jInfo.BasicLimitInformation.ActiveProcessLimit=4;
     SetInformationJobObject(hJob,
         JobObjectExtendedLimitInformation, &jInfo,
```

```
        sizeof(jInfo));
    AddProcess2Job();
    Sleep(15000);
    TerminateJobObject(hJob, 0);
    return 0;
}
```

2.2. Контрольні питання

1. Скільки процесів у програмі в прикладі 2.1 додається до завдання? Чи всі процеси додаються до завдання вдало? Чому?

2. Для чого використовується створення процесів у призупиненому стані в програмі в прикладі 2.1?

3. Яким чином виконується завершення всіх створених процесів у прикладі 2.1?

4. Які основні обмеження можна налаштувати у завданні?

5. Чим відрізняється управління групою процесів без використання та з використанням завдання?

2.3. Індивідуальні завдання

Розробити консольну програму в середовищі програмування Visual C++, яка реалізує простий менеджер програм: дозволяє запускати вказані програми та завершувати їх (за вибором користувача) згідно з вимогами, що задаються варіантами, що наведені нижче. При завершенні роботи розробленого менеджера всі запущені ним програми повинні бути примусово завершені. Для реалізації обов'язково використовувати нитки та завдання.

Увага! Для кожної дії з завданнями необхідно використовувати окрему нитку. Приклад основи розробленого менеджера програм:

```
unsigned ThreadFunc1(void* arg)
{
      // підготовчі дії для запуска програми
      ...
      // запуск програми та додавання її
```

```
      // в завдання hJob1
      if (!CreateProcess(...) return 0;
      if (!AssignProcessToJobObject(hJob1,
            pi.hProcess))
      {
      TerminateProcess(pi.hProcess, 0);
      printf("ERROR: AssignProcessToJobObject %i\n",
            i);
      }
      else ResumeThread(pi.hThread);
      }
      ExitThread(0);
      return 0;
}

unsigned ThreadFunc2(void* arg)
{
      // запуск другої програми та додавання її
      // в завдання hJob1 (або інше - згідно варіанта)
      ...
}

// інші нитки для виконання дій згідно варіанта
...

int main(int argc, char* argv[])
{
// Підготовча робота – створення завдання hJob1
// (або завдань), налаштування його та інші дії
hJob1 = CreateJobObject(NULL, "MyJob1");
...
// Головний цикл
for (;;) {
```

```
      // Очікування введення з клавіатури в змінну c
      ...
      switch(c) {
            // запуск першої програми
            case 1: CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)ThreadFunc1, ...);
            // запуск другої програми
            case 2: CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)ThreadFunc2, ...);
            // інші дії по управлінню згідно завдання
            case 3: ...
            ...
            // завершення роботи та знищення об'єкта
            // завдання hJob1 з закриттям всіх
            // процесів, що належать завданню
            case 0: TerminateJobObject(hJob1, 0);
                  // закриття інших об'єктів завдань
                  ...
                  return 0;
            }
      }
}
```

1) Запуск необмеженої кількості програм блокнота (**notepad.exe**), кожна з яких виконується не більше 10 секунд.

2) Запуск програм графічного редактора Paint (**mspaint.exe**) та блокнота (**notepad.exe**) загальною кількістю не більше 5 екземплярів у будь якому співвідношенні.

3) Запуск програм блокнота (**notepad.exe**) та графічного редактора Paint (**mspaint.exe**) кількістю не більше 3 екземплярів кожна.

4) Запуск програм графічного редактора Paint (**mspaint.exe**) та блокнота (**notepad.exe**) не більше ніж по 1 екземпляру та їх вибіркове завершення.

5) Запуск необмеженої кількості програм блокнота (**notepad.exe**) та вибіркове їх завершення (передбачити також одночасне завершення всіх копій з можливістю продовження роботи менеджера).

6) Запуск не більше 10 копій одночасно редакторів WordPad (**write.exe**) з забороною використання для них буфера обміну.

7) Запуск не більше 5 редакторів WordPad (**write.exe**), при завершенні роботи видати всю доступну статистичну інформацію про роботу редакторів.

8) Запуск не більше 10 програм блокнота (**notepad.exe**), сумарний час виконання не повинен перевищувати 1 хвилини.

9) Запуск будь якої кількості програм графічного редактора Paint (**mspaint.exe**), блокнота (notepad.exe) та редактора WordPad (**write.exe**), та вибіркове їх завершення або завершення групами (групи формуються по типам програм).

10) Запуск програм графічного редактора Paint (**mspaint.exe**), блокнота (**notepad.exe**) та редактора WordPad (**write.exe**) за наступною схемою: користувач обирає, яку програму запустити. Якщо вказана програма завершується, автоматично запускається наступна із списку і так для всіх програм. Користувач на початку може обрати будь-яку програму для запуску, автоматично повинні запускатися всі інші програми із списку.

11) Запуск та завершення вказаних користувачем програм (імена задаються з клавіатури, каталог для програм за замовчанням – системний). По команді з клавіатури також виводити перелік назв файлів програм, які були успішно запущені на виконання та про невдалі спроби запуску.

3. ВІКОННІ ПРОГРАМИ В ОС WINDOWS

Мета: вивчити можливості операційної системи Windows по створенню віконних прикладних програм для забезпечення зручного графічного інтерфейсу користувача.

3.1. Теоретичні відомості

Графічний інтерфейс користувача робить можливим використання графіки на екрані дисплея. Графіка дає можливість краще сприймати інформацію, тобто мати WYSIWYG (What you see is what you get – що ви бачите, то і отримаєте) як для графіки, так і для відформатованих для друку текстових документів.

Архітектура віконної програми наведена на рис. 3.1. Процес з графічним інтерфейсом (Windows Application) (*e*) має одну головну нитку (Main Thread) (*g*), одне або кілька вікон (Window) (*a*) і одну або кілька дочірніх ниток (UI / Worker Thread) (*k*) – робочі нитки або нитки UI.

В головній функції необхідно вказати клас вікна та зареєструватися у Windows (*d*), після цього створити вікна (Window1-Window4) (*a*) і відобразити їх. Клас вікна – це структура, яка містить такі атрибути вікна, як стилі вікна, значок, курсор, колір фону, ім'я ресурсу меню, ім'я класу вікна тощо. Реєстрація класу вікна пов'язує процедуру вікна, стилі класу та інші атрибути класу з назвою класу. З кожним класом вікна пов'язана віконна процедура (**WndProc1**, **WndProc2**) (*c*), яка використовується спільно з усіма вікнами (*a*) того самого класу в програмі.

Віконна процедура (**WndProc1**, **WndProc2**) (*c*) – це функція, яка отримує та обробляє всі повідомлення, надіслані до вікна (*a*). Кожний клас вікон (Window Class1, Window Class2) (*b*) має віконну процедуру (*c*), і кожне вікно (*a*), створене за допомогою цього класу, використовує цю саму віконну процедуру для відповіді на повідомлення. Система надсилає повідомлення віконній процедурі, передаючи дані повідомлення як аргументи процедури. Потім віконна процедура виконує відповідну дію для повідомлення; перевіряє ідентифікатор повідомлення та під час обробки повідомлення використовує інформацію, визначену параметрами повідомлення.

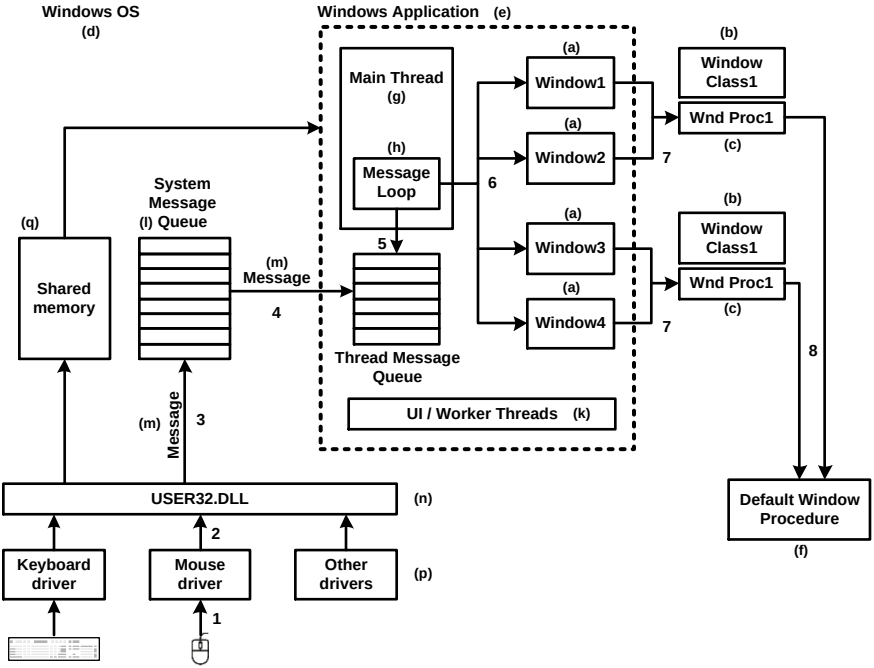


Рисунок 3.1 – Архітектура віконної програми Windows

Віконна процедура зазвичай не ігнорує повідомлення. Якщо вона не обробляє повідомлення, то повинна надіслати повідомлення назад до системи для обробки за замовчуванням. Віконна процедура робить це шляхом виклику функції **DefWindowProc** (Default Window Procedure) (*f*), яка виконує дію за замовчуванням і повертає результат повідомлення. Функція віконної процедури за замовчуванням, **DefWindowProc**, визначає поведінку, спільну для всіх вікон. Стандартна віконна процедура забезпечує мінімальну функціональність для вікна.

Повідомлення Windows (Message) (*m*) містить набір параметрів, таких як дескриптор вікна , ідентифікатор повідомлення та два значення, які називаються параметрами повідомлення (`WPARAM` і `LPARAM`). Дескриптор

вікна визначає вікно, для якого призначене повідомлення. Ідентифікатор повідомлення – це константа, яка визначає мету повідомлення. Наприклад, ідентифікатор повідомлення WM_PAINT повідомляє віконній процедурі, що клієнтська область вікна була змінена і повинна бути перемальована.

Існує два типи повідомлень Windows: визначені системою повідомлення, та повідомлення, визначені програмою. Система надсилає або публікує визначене системою повідомлення, коли вона спілкується з програмою. Програма також може надсилати або публікувати визначені системою повідомлення. Наприклад, ідентифікатор повідомлення WM_PAINT використовується у визначеному системою повідомленні, яке вимагає перемалювати вміст вікна.

Програма може створювати повідомлення для використання її власними вікнами або для зв'язку з вікнами в інших процесах. Якщо програма створює власні повідомлення, віконна процедура, яка їх отримує, повинна інтерпретувати повідомлення та забезпечувати відповідну обробку.

Операційна система Windows використовує два методи для маршрутизації повідомлень у віконну процедуру: 1) розміщення повідомлень у черзі «першим прийшов, першим вийшов» (FIFO), яка називається чергою повідомлень (System Message Queue) (*l*) або 2) через об'єкт системної пам'яті (Shared Memory) (*q*), який тимчасово зберігає повідомлення, і далі – надсилання повідомлень безпосередньо до віконної процедури.

Операційна система Windows підтримує лише одну загальносистемну чергу повідомлень (*l*) для всіх програм. Для кожної нитки графічного інтерфейсу користувача (UI) створюється окрема черга повідомлень (Thread Message Queue) (*j*). Щоразу, коли користувач створює нитку інтерфейсу користувача, для цієї нитки також створюється черга повідомлень.

Консольні процеси викликають функції C-Runtime для отримання вхідних даних від системи. Наприклад, процес викликає функцію C-Runtime **gets()** для отримання введення з клавіатури. Але віконні процеси Windows не здійснюють явних викликів функцій для отримання вхідних даних. Замість цього вони чекають, поки система передасть їм дані, тобто віконні процеси Windows керуються подіями.

Кожного разу, коли користувач рухає мишею, натискає на клавіши миші (1) або друкує на клавіатурі, драйвер пристрою (*p*) для миші (Mouse Driver) або клавіатури (Keyboard Driver) передає (2) дані до системної бібліотеки User32.dll (*n*), яка, у свою чергу, перетворює введення в повідомлення Windows і розміщує (3) їх у системній черзі повідомлень (*l*).

Операційна система Windows видаляє повідомлення по одному з системної черги повідомлень, перевіряє їх, щоб визначити вікно призначення, а потім розміщує (4) їх у черзі повідомлень нитки вікна призначення. Черга повідомлень нитки отримує всі повідомлення миші та клавіатури для вікон, створених ниткою. Нитка видаляє (5) повідомлення зі своєї черги та вказує системі надіслати (6 і 7) їх до відповідної віконної процедури для обробки.

Віконна програма Windows має функцію **WinMain()**, яка є початковою для програми, подібно до функції **main()** у консольній програмі мовою С. Функція **WinMain()** діє як функція основної нитки програми. Наступна частина коду у функції **WinMain()** називаються Message Loop або Message Pump (*h*) (наведено класичну реалізацію).

```
while (GetMessage(&Msg, NULL, 0, 0))
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
```

Виклик **GetMessage()** циклу повідомлень перевіряє чергу повідомлень на наявність повідомлень Windows. Якщо буде знайдене дійсне повідомлення, він видаляє повідомлення зі своєї черги. Після видалення повідомлення з черги застосунок може використовувати функцію **DispatchMessage()**, щоб наказати системі надіслати повідомлення до віконної процедури для обробки. Якщо в черзі повідомлень немає повідомлень, виклик **GetMessage()** буде заблоковано, доки в чергу повідомлень не надійде дійсне повідомлення.

У проміжку між цими двома викликами виклик методу **TranslateMessage()** нічого не робить, а просто перекладає віртуальні клю-

чові повідомлення в символьні повідомлення. Без цього виклику при виконанні програми будуть відбуватися помилки, пов'язані із клавіатурою.

Цикл повідомлень (цикл `while`) переривається, коли **GetMessage()** отримує повідомлення WM_QUIT із черги повідомлень, і згодом основна нитка програми завершує роботу.

Нижче наведений базовий каркас віконної програми. Цей каркас використовується далі для створення прикладів віконних програм.

```
#include <windows.h>

LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance,
      HINSTANCE hPrevInstance, PSTR szCmdLine,
      int iCmdShow)
{
static char szAppName[] = "MyWindowApp";
HWND        hwnd;
MSG         msg;
WNDCLASSEX  wndclass;

wndclass.cbSize = sizeof(wndclass); //розмір структури
wndclass.style = CS_HREDRAW|CS_VREDRAW; // стиль вікна
      // - перемальовування при зміні розмірів
wndclass.lpfnWndProc = WndProc; // покажчик на функцію
      // зворотного виклику для обробки повідомлень
wndclass.cbClsExtra = 0; // число додаткових байтів в
      // кінці цієї структури
wndclass.cbWndExtra = 0; // число додаткових байтів за
      // екземпляром вікна
wndclass.hInstance = hInstance; // дескриптор
      // екземпляру, в якому знаходиться
      // віконна процедура
```

```
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
      // дескриптор ярлика для класу вікна
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
      // дескриптор курсора для класу вікна
wndclass.hbrBackground =
      (HBRUSH) GetStockObject(COLOR_BACKGROUND);
      // дескриптор пензля фона вікна
wndclass.lpszMenuName = NULL; // ім'я меню
wndclass.lpszClassName = szAppName; // ім'я класу
wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
      //дескриптор ярлика для заголовка вікон програми
RegisterClassEx (&wndclass); // реєстрація класу вікна
hwnd = CreateWindow // створення вікна на базі класу
      (szAppName, // ім'я класу вікна
      "My Window App", // заголовок вікна
      WS_OVERLAPPEDWINDOW, // стиль вікна
      CW_USEDEFAULT, // початкове розміщення по x
      CW_USEDEFAULT, // початкове розміщення по y
      CW_USEDEFAULT, // початковий розмір по x
      CW_USEDEFAULT, // початковий розмір по y
      NULL, // дескриптор батьківського вікна
      NULL, // дескриптор меню вікна
      hInstance, // дескриптор екземпляра програми
      NULL); // параметри створення
ShowWindow(hwnd,iCmdShow);//зображення вікна на екрані
UpdateWindow (hwnd); // перемальовування змісту вікна
while (GetMessage (&msg, NULL, 0, 0))
      // цикл отримання повідомлень з черги
      {
      TranslateMessage (&msg); // перетворення
            //повідомлення від клавіатури
      DispatchMessage (&msg); //відправка повідомлення
            // віконній функції
```

```
      }
return msg.wParam;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
            WPARAM wParam, LPARAM lParam)
{
HDC hdc;
PAINTSTRUCT ps;
RECT rect;
// обробка повідомлень
switch (iMsg)
      {
      case WM_CREATE : // створення вікна
            . . .
            return 0;
            break;
      case WM_PAINT : // малювання в вікні
            hdc = BeginPaint (hwnd, &ps);
            DrawText (hdc, MyStr, -1, &rect,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
                  // виведення тексту
            EndPaint (hwnd, &ps);
            return 0;
            break;
      . . .
      case WM_DESTROY : // знищення вікна
            . . .
            PostQuitMessage (0); // запит операційній
                  // системі на завершення програми
      return 0;
      }
// обробка за замовчанням всіх повідомлень, що не були
```

```
// оброблені в програмі
return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

Розглянемо основні дії, що виконуються при створенні віконної програми на прикладі наведеного каркаса.

Функція **WinMain** використовує послідовність викликів функцій WinAPI та, після свого завершення, повертає операційній системі Windows ціле число. У цієї функції є чотири параметра. Параметр `hInstance` називається описувачем екземпляра (instance handle) – це унікальне число, яке ідентифікує програму, коли вона виконується під управлінням ОС Windows. Може так статись, що користувач запустить під ОС Windows декілька копій однієї й тієї ж програми. Кожна копія в цьому випадку називається «екземпляром», і кожна буде мати своє значення `hInstance`. Параметр `hPrevInstance` – не використовується. Параметр `szCmdLine` – покажчик на рядок, який закінчується нульовим символом. У ньому знаходяться будь-які параметри, що передані в програму із командного рядка. Параметр `iCmdShow` – число, яке показує, яким має бути виведене на екран вікно прикладної програми в початковий момент.

Вікно завжди створюється на основі класу вікна. Клас вікна ідентифікує віконну процедуру, яка виконує процес обробки повідомлень, що надходять вікну. На основі одного класу вікна можна створити декілька вікон. Наприклад, усі вікна-кнопки в ОС Windows створюються на основі одного й того ж класу вікна. Клас вікна визначає віконну процедуру та деякі інші характеристики вікон, які були створені на основі цього класу.

Перед створенням вікна необхідно зареєструвати клас вікна шляхом виклику функції **RegisterClassEx**. У функції **RegisterClassEx** є тільки один параметр – покажчик на структуру даних WNDCLASSEX. Структура WNDCLASSEX визначається в заголовних файлах Windows.

Значення **WndProc** поля `wndclass.lpfnWndProc` у структурі WNDCLASSEX встановлює ім'я **WndProc** в якості імені віконної процедури даного вікна.

Створення вікна відбувається завдяки виклику функції **CreateWindow**. Необхідно, щоб вся інформація через параметри передавалась функції **CreateWindow** під час її виклику. У той час, коли функція **CreateWindow** повертає управління прикладній програмі, вікно вже створене усередині ОС Windows. Але на екрані монітора воно ще не з'являється. Необхідні виклики ще двох функцій.

Перша з них – це функція **ShowWindow**.

Першим її параметром є дескриптор `hwnd` щойно створеного функцією **CreateWindow** об'єкта «вікно». Другим параметром є значення `iCmdShow`, що передається як параметр функції **WinMain**. Цей параметр задає початковий вигляд вікна на екрані. Якщо параметр `iCmdShow` має значення SW_SHOWNORMAL, на екран виводиться звичайне вікно. Якщо параметр `iCmdShow` має значення SW_SHOWMINNOACTIVE, то вікно не виводиться, а на панелі задач з'являються його ім'я та іконка. У параметра `iCmdShow` можуть бути ще і інші значення. Виклик функції **ShowWindow** супроводжується виведенням вікна на екран.

Другою необхідно викликати функцію **UpdateWindow**.

Ця функція перемальовує робочу область. Для цього у віконну процедуру (функція **WndProc**) відправляється повідомлення WM_PAINT.

Далі виконується цикл обробки повідомлень (Message Loop). Ім'я `msg` – це ім'я структури типу MSG, яка визначається в заголовних файлах Windows. Якщо поле `message` повідомлення, добутого із черги повідомлень, дорівнює любому значенню, окрім WM_QUIT, функція **GetMessage** повертає ненульове значення. Повідомлення WM_QUIT вимушує програму перервати цикл обробки повідомлень.

Функція **TranslateMessage** передає структуру `msg` назад у Windows для перетворення повідомлення з клавіатури. Функція **DispatchMessage** також передає структуру `msg` назад у Windows. У свою чергу Windows відправляє повідомлення для його обробки відповідній віконній процедурі. Таким чином, віконну процедуру викликає Windows.

Для створення можливості обробки помилок при отриманні повідомлень за допомогою функції **GetMessage** класичний цикл обробки повідомлень рекомендовано модифікувати наступним чином:

```
BOOL bRet;
while ((bRet = GetMessage(&msg, NULL, 0, 0)) != 0)
{
	if (bRet == -1)
	{
		//обробка помилки та найймовірніше - вихід
	}
	else
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
}
```

Кожне отримуване вікном повідомлення ідентифікується номером, який міститься в параметрі `iMsg` віконної процедури. У заголовних файлах Windows визначені ідентифікатори, в яких є префікс WM (window message) для кожного типу повідомлень.

Звичайно використовується конструкція `switch` для визначення того, яке саме повідомлення отримала віконна процедура, та того, як його обробляти. Якщо віконна процедура обробляє повідомлення без помилки, то значенням, яке вона повертає, має бути нуль. Усі повідомлення, які не обробляються віконною процедурою, повинні передаватись функції у Windows через функцію **DefWindowProc**. Значення, що повертає функція **DefWindowProc**, має бути значенням, яке повертає віконна процедура.

Додаткову інформацію про роботу віконної прикладної програми можна отримати із джерел інформації по даній темі та із системи допомоги MS SDK.

Базові елементи графічного інтерфейсу та їх використання.

За допомогою функції **CreateWindow** можна створювати вікна зумовленого типу: кнопки, статичні елементи керування, списки, поля зі списком, елементи керування редагуванням та лінійки прокрутки. Клас вікна задається параметром `lpClassName` у функції **CreateWindow**. Розглянемо деякі з них.

Примітка: фрагменти коду, що демонструють створення вікон необхідно вставити в блок обробки повідомлення WM_CREATE в наведеному вище тексті програми, для кожного вікна необхідно об'явити змінну – дескриптор вікна (для меню – дескриптор меню), та задати ідентифікатори вікон (наприклад: `#define GROUPBOX_1 101`).

Створення кнопки та робота з нею. Для створення кнопки необхідно задати клас вікна BUTTON. Задаючи різні стилі в параметрі `dwStyle` можна створити вікно кнопки (стилі BS_PUSHBUTTON, BS_DEFPUSHBUTTON), групи (BS_GROUPBOX), прапорця (BS_CHECKBOX, BS_AUTOCHECKBOX, BS_3STATE, BS_AUTO3STATE), перемикача (BS_RADIOBUTTON, BS_AUTORADIOBUTTON) або піктограми (BS_ICON, BS_BITMAP). Інші стилі дозволяють керувати розміщенням тексту у вікні даного типу. Стилі BS_AUTOCHECKBOX, BS_AUTO3STATE, BS_AUTORADIOBUTTON задають автоматичну зміну стану вікна при його виборі. Стиль BS_NOTIFY дозволяє посилати повідомлення BN_CLICKED, BN_DBLCLK та BN_DOUBLECLICKED батьківському вікну при роботі з кнопкою.

За допомогою стилю BS_PUSHLIKE можна задати для прапорця або перемикача зображення у вигляді звичайної кнопки (наприклад – при виборі перемикача кнопка буде в натиснутому стані).

Приклад 3.1. У наведеному фрагменті коду демонструється створення групи з двох перемикачів, одного прапорця у вигляді кнопки з трьома станами та звичайної кнопки. При натисканні на звичайну кнопку перемикач 1 стає натиснутим, перемикач 2 – ні. Результат роботи наведено на рис. 3.2.

```
// фрагмент опису змінних
HWND  hwnd1, hwnd2, hwnd3, hwnd4, hwnd5;
```

```
#define     GROUPBOX_1  101
#define     RADIOBUTTON_1     102
#define     RADIOBUTTON_2     103
#define     AUTO3STATE_1      104
#define     BUTTON_1    105

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
hwnd1 = CreateWindow ("BUTTON", "RADIO GROUP",
      BS_GROUPBOX|WS_CHILD, 10,10,160,85,
      hwnd,(HMENU)GROUPBOX_1,NULL,NULL) ;
ShowWindow (hwnd1, SW_SHOW) ;
hwnd2 = CreateWindow  ("BUTTON", "First",
      BS_AUTORADIOBUTTON|WS_CHILD,
      10,30,100,20,
      hwnd1,(HMENU)RADIOBUTTON_1,NULL,NULL) ;
ShowWindow (hwnd2, SW_SHOW) ;
hwnd3 = CreateWindow ("BUTTON", "Second",
      BS_AUTORADIOBUTTON|WS_CHILD,  10,55,100,20,
      hwnd1,(HMENU)RADIOBUTTON_2,NULL,NULL) ;
ShowWindow (hwnd3, SW_SHOW) ;
hwnd4 = CreateWindow ("BUTTON","3 State CheckBox",
      BS_PUSHBUTTON|WS_CHILD | BS_AUTO3STATE,
      10,110,160,20,
      hwnd,(HMENU)AUTO3STATE_1,NULL, NULL) ;
ShowWindow (hwnd4, SW_SHOW) ;
hwnd5 = CreateWindow ("BUTTON", "Simple Button",
      BS_PUSHBUTTON | WS_CHILD, 10, 140, 160, 20,
      hwnd, (HMENU)BUTTON_1, NULL, NULL);
ShowWindow(hwnd5, SW_SHOW);
break;

// обробка натискання на кнопку
```

```
case WM_COMMAND:
      switch (wParam)
      {
      case BUTTON_1:
      SendMessage(hwnd2,BM_SETCHECK,BST_CHECKED, 0);
      SendMessage(hwnd3,BM_SETCHECK,BST_UNCHECKED, 0);
      break;
      }
break;
```

Рисунок 3.2 – Приклад створення кнопок

Для роботи з кнопками можна скористуватися функціями **CheckDlgButton** (зміна стану кнопки), **CheckRadioButton** (вибір перемикача та очищення стану в інших перемикачів у групі) та **IsDlgButtonChecked** (перевірка стану кнопки), а також використовувати передачу повідомлень для керування кнопками. Найчастіше використовуються повідомлення: BM_CLICK (імітація натискання на кнопку), BM_GETCHECK (отримання стану перемикача або прапорця), BM_GETSTATE (отримання стану кнопки або прапорця), BM_SETCHECK (встановлення стану перемикача або прапорця), BM_SETSTATE (встановлення стану натискання кнопки), BN_CLICKED (повідомлення про подію

натискання на кнопку), BN_DBLCLK та BN_DOUBLECLICKED (два однакових повідомлення про подвійні натискання на кнопку).

Комбіновані списки. Для створення списку необхідно задати клас вікна COMBOBOX. Можна створювати списки з полем редагування у верхній частині (стиль CBS_SIMPLE) або списки вибору, що випадають (стилі CBS_DROPDOWN та CBS_DROPDOWNLIST). Інші варіанти стилів задають форматування виведення рядків символів.

Для роботи зі списками можна використовувати функцію **GetComboBoxInfo** (отримання інформації про список – дескриптори вікон, розміри). Також для роботи зі списками використовуються наступні повідомлення: CB_ADDSTRING (додавання рядка символів у список, якщо не встановлено стиль CBS_SORT, тоді рядок додається в кінець списку, інакше – згідно з сортуванням), CB_DELETESTRING (видалення рядка зі списку за його номером), CB_DIR (додавання в список рядків з переліком файлів та каталогів), CB_GETCOMBOBOXINFO (отримання інформації про список – аналог функції **GetComboBoxInfo**), CB_GETCOUNT (отримання кількості рядків у списку), CB_GETCURSEL (отримання індексу вибраного елементу списку), CB_GETLBTEXT (отримання рядка символів із списку за індексом), CB_GETLBTEXTLEN (отримання довжини рядка символів зі списку за індексом), CB_INSERTSTRING (додавання рядка символів у вказану позицію, або при індексу -1 – у кінець списку), CB_RESETCONTENT (видалення всіх рядків зі списку), CBN_CLOSEUP (повідомлення про закриття списку), CBN_DBLCLK (повідомлення про подвійні натискання на рядок у списку), CBN_DROPDOWN (повідомлення про розкриття списку, що випадає), CBN_SELCHANGE, CBN_SELENDCANCEL, CBN_SELENDOK (три повідомлення про вибір рядка в списку).

Приклад 3.2. У наведеному фрагменті коду створюються два списки – один, що випадає, та другий з полем редагування у верхній частині. У першому списку задано сортування та перетворення до нижнього регістру, у другому – перетворення до верхнього регістру та додавання списку файлів та каталогів з кореню диска C. При подвійному натисканні на ліву клавішу миші на будь-який елемент другого списку, цей елемент додається в перший список. Результат створення списків наведено на рис. 3.3.

```
// фрагмент опису змінних
HWND  hwnd1, hwnd2;
#define COMBOBOX_1      106
#define COMBOBOX_2      107

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
hwnd1 = CreateWindow
      ("COMBOBOX", "Combo1",
      CBS_DROPDOWN|WS_CHILD|CBS_SORT|CBS_LOWERCASE,
      10,10,160,80,
      hwnd,(HMENU)COMBOBOX_1, NULL, NULL);
ShowWindow (hwnd1, SW_SHOW);
SendMessage (hwnd1,CB_ADDSTRING,0, (LPARAM)"1 First");
SendMessage(hwnd1,CB_ADDSTRING,0, (LPARAM)"1 Second");
SendMessage (hwnd1,CB_ADDSTRING,0, (LPARAM)"1 Third");
SendMessage(hwnd1,CB_ADDSTRING,0, (LPARAM)"1 Fourth");
hwnd2 = CreateWindow
      ("COMBOBOX","Combo2",
      CBS_SIMPLE|WS_CHILD|CBS_UPPERCASE,
      10,50,160,120,
      hwnd,(HMENU)COMBOBOX_2, NULL, NULL);
ShowWindow (hwnd2, SW_SHOW);
SendMessage (hwnd2,CB_ADDSTRING,0, (LPARAM)"2 First");
SendMessage(hwnd2,CB_ADDSTRING,0, (LPARAM)"2 Second");
SendMessage (hwnd2, CB_DIR, DDL_DIRECTORY,
      (LPARAM)"c:\\*.*");
break;

// обробка подвійного натискання на елементі 2 списку
case WM_COMMAND:
switch LOWORD(wParam)
      {
```

```
      case COMBOBOX_2:
      {
      switch (HIWORD(wParam))
      {
      case CBN_DBLCLK:
      {
      char buf[100]; //обмежений буфер для елемента
      // отримання індексу обраного елемента
      int ind = SendMessage(hwnd2,CB_GETCURSEL,0,0);
      // зчитування елементу за індексом
      SendMessage(hwnd2, CB_GETLBTEXT, ind,
            (LPARAM)buf);
      // додавання зчитаного елементу в 1 список
      SendMessage(hwnd1, CB_ADDSTRING, 0,
            (LPARAM)buf);
            }
      break;
      }}
}
break;
```

Рисунок 3.3 – Приклад створення списків

Елемент редагування. Для створення прямокутного елементу редагування тексту, що має один або декілька рядків використовується клас вікна EDIT. Усі стилі задають форматування виведення рядка символів в елементі редагування. Деякі стилі наведені в прикладі 3.3.

Для роботи з елементом редагування можна використовувати такі повідомлення: EM_CANUNDO (перевірка можливості відміни останніх дій), EM_EMPTYUNDOBUFFER (очищення журналу останніх дій), EM_GETFIRSTVISIBLELINE (для багаторядкових елементів керування отримує перший рядок, який видно в елементі), EM_GETLINE (копіює рядок з елементу редагування у виділений буфер), EM_GETLINECOUNT (отримує кількість рядків тексту в багаторядковому елементі редагування), EM_GETPASSWORDCHAR (отримує символ, що виводиться на екран, у разі коли користувач вводить пароль. Є сенс використовувати, якщо стиль елемента редагування – ES_PASSWORD), EM_GETSEL (отримує першу та останню позицію виділеного тексту), EM_REPLACESEL (виконується заміна виділеного тексту), EM_SETMODIFY (встановлення ознаки того, що текст було змінено), EM_SETPASSWORDCHAR (встановлення символу, що виводиться на екран, коли користувач вводить пароль. Якщо задати параметр – 0, символи виводяться без заміни), EM_SETREADONLY (встановлює або видаляє стиль тільки для читання. Також можна встановити режим тільки для читання при створенні елемента за допомогою стилю ES_READONLY), EM_SETSEL (виділяє текст у межах заданих позицій), EM_UNDO (відміняє останню операцію над текстом в елементі редагування), EN_CHANGE (ознака зміни тексту), WM_CLEAR (очищення елементу редагування), WM_COPY (копіювання тексту в буфер обміну), WM_CUT (видалення тексту з елементу редагування та розміщення його в буфері обміну), WM_PASTE (вставлення тексту з буфера обміну в елемент редагування), WM_UNDO (відміна останньої операції).

Приклад 3.3. У наведеному фрагменті коду програми створюються два елементи редагування з різними стилями, для другого елементу при введенні паролю символ «*» замінюється на «=», для першого – виділяється частина слова «Edit1» та замінюється на слово «TEXT». При натисканні на

праву клавішу миші виконується зчитування тексту з першого елемента редагування з видаленням та його виведення у вікні **MessageBox**. При повторних натисканнях на праву клавішу миші буде виводитись оновлений вміст елемента редагування: порожній рядок або текст, який буде введено з клавіатури. На рис. 3.4 наведений результат виконання програми.

```
// фрагмент опису змінних
HWND  hwnd1, hwnd2;
#define EDIT_1    108
#define EDIT_2    109
int ind;

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
hwnd1 = CreateWindow ("EDIT", "Edit1",
      ES_CENTER|WS_CHILD|WS_BORDER, 10,10,160,20,
      hwnd,(HMENU)EDIT_1,NULL,NULL);
ShowWindow (hwnd1, SW_SHOW);
hwnd2 = CreateWindow ("EDIT", "Edit2",
      ES_PASSWORD|WS_CHILD, 10,40,160,20,
      hwnd,(HMENU)EDIT_2,NULL,NULL);
ShowWindow (hwnd2, SW_SHOW);
SendMessage(hwnd1,EM_SETSEL,2,4);
SendMessage(hwnd1,EM_REPLACESEL,TRUE,(LPARAM)"TEXT");
SendMessage(hwnd2,EM_SETPASSWORDCHAR,'=',0);
break;

// обробка натискання на праву клавішу миші
case WM_RBUTTONDOWN:
char buf[100];
// отримуємо довжину текста в елементі редагування
ind = (WORD)SendMessage(hwnd1, EM_LINELENGTH, 0, 0);
// зчитування тексту з поля редагування
```

```
SendMessage(hwnd1, EM_GETLINE, 0, (LPARAM)buf);
buf[ind] = 0; // завершення рядка символом з кодом 0
// вибір всього тексту в елементі редагування
SendMessage(hwnd1, EM_SETSEL, 0, -1);
// очищення елемента редагування
SendMessage(hwnd1, WM_CLEAR, 0, 0);
MessageBox(hwnd, buf, "TEXT FROM EDIT FIELD", MB_OK);
break;
```

Рисунок 3.4 – Приклад створення елементів редагування

Елемент керування списком. Для створення елемента керування списком необхідно задати клас вікна LISTBOX. Стилі задають форму елемента та можливості по керуванню списком рядків символів. Деякі стилі наведені в переліку повідомлень та прикладі 3.4.

Для роботи з елементом керування списком використовуються такі повідомлення: LB_ADDFILE (додає ім'я файла, якщо список містить каталог файлів), LB_ADDSTRING (додає рядок символів у кінець списку або згідно з сортуванням, якщо список має стиль LBS_SORT), LB_DELETESTRING (видаляє рядок символів зі списку за його індексом), LB_DIR (додавання в список рядків з переліком файлів та каталогів), LB_FINDSTRING та LB_FINDSTRINGEXACT (пошук рядка в списку за входженням або за повним збігом), LB_GETCOUNT (отримання кількості рядків у списку), LB_GETCURSEL (отримання індексу обраного елемента

списку), LB_GETSEL (визначення, чи обраний елемент списку за вказаним індексом), LB_GETSELCOUNT (отримання кількості обраних елементів списку. використовується для списків з можливістю вибору кількох елементів. Для створення такого списку необхідно задати стиль LBS_MULTIPLESEL або LBS_EXTENDEDSEL), LB_GETSELITEMS (заповнює вказаний буфер масивом індексів обраних елементів списку), LB_GETTEXT (отримує рядок за вказаним індексом зі списку), LB_GETTEXTLEN (отримує довжину рядка за вказаним індексом зі списку), LB_INITSTORAGE (виділення пам'яті для списку в разі наступного додавання великої кількості рядків символів. Ця операція прискорює роботу зі списком, якщо кількість елементів у ньому буде перевищувати 100), LB_INSERTSTRING (додавання рядка символів в указану позицію, або при індексу -1 – у кінець списку), LB_RESETCONTENT (видалення всіх рядків зі списку), LB_SELECTSTRING (пошук у списку рядка за входженням та в разі успіху – виділення його. Повертається індекс знайденого елемента), LB_SELITEMRANGE та LB_SELITEMRANGEEX (виділення одного або більше підряд розташованих елементів), LB_SETCURSEL (виділення елемента списку та в разі необхідності прокрутка списку так, щоб виділений елемент було видно), LB_SETSEL (в списку з можливістю вибору кількох елементів встановлює або прибирає вибір одного елементу, що задається його індексом), LBN_DBLCLK (повідомлення про подвійні натискання на рядок у списку. Передається батьківському вікну, якщо встановлено стиль LBS_NOTIFY).

Приклад 3.4. У наведеному фрагменті коду виконується створення елементу керування списком, заповнення його чотирма рядками та виділення другого та четвертого рядка (нумерація індексів елементів списку починається з 0). У разі подвійного натискання на ліву клавішу миші на будь-якому рядку в списку – цей рядок додається в кінець списку. Результати виконання коду після створення елементу керування списком та після подвійних натискань на ліву клавішу миші наведені на рис. 3.5.

```
// фрагмент опису змінних
HWND  hwnd1;
```

```
#define LISTBOX_1 110

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
hwnd1 = CreateWindow
      ("LISTBOX", "ListBox1",
      WS_CHILD | LBS_EXTENDEDSEL | LBS_NOTIFY,
      10,40,160,120,hwnd,(HMENU)LISTBOX_1,NULL,NULL);
ShowWindow(hwnd1, SW_SHOW);
SendMessage(hwnd1, LB_ADDSTRING, 0, (LPARAM)"First");
SendMessage(hwnd1, LB_ADDSTRING, 0, (LPARAM)"Second");
SendMessage(hwnd1, LB_ADDSTRING, 0, (LPARAM)"Third");
SendMessage(hwnd1, LB_ADDSTRING, 0, (LPARAM)"Fourth");
SendMessage(hwnd1, LB_SETSEL,TRUE,1);
SendMessage(hwnd1, LB_SETSEL, TRUE, 3);
break;

// обробка подвійного натискання на елементі списку
case WM_COMMAND:
switch LOWORD(wParam)
{
      case LISTBOX_1:
      {
      switch (HIWORD(wParam))
      {
      case LBN_DBLCLK:
      char buf[100];
      // отримання індексу обраного рядка
      int ind = SendMessage(hwnd1,LB_GETCURSEL,0,0);
      // отримання довжини рядка
      int lngth = SendMessage(hwnd1, LB_GETTEXTLEN,
            ind, 0);
      // зчитування рядка
```

```
        SendMessage(hwnd1,LB_GETTEXT, ind, (LPARAM)buf);
        // у кінець рядка додавання символа з кодом 0
        buf[lngth] = 0;
        // додавання рядка в список
        SendMessage(hwnd1,LB_ADDSTRING, 0, (LPARAM)buf);
        break;
        }
        }
}
```

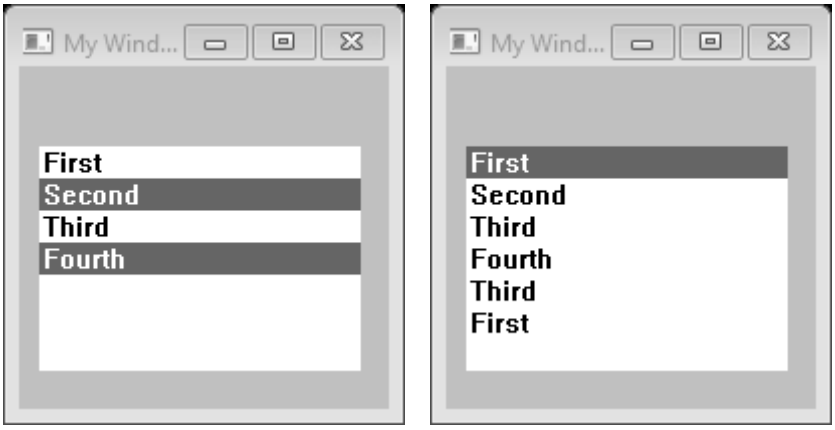


Рисунок 3.5 – Приклад створення елемента керування списком

Статичний елемент керування. Для створення елемента керування статичним текстом необхідно задати клас вікна STATIC. Можна створювати текстові та графічні елементи.

Можуть бути використані такі стилі для статичного елемента: SS_BITMAP та SS_ICON (задається, що в елементі керування буде зображення. Задається ім'я зображення з ресурсів програми, а не файла із зображенням), SS_BLACKFRAME, SS_GRAYFRAME та SS_WHITEFRAME (встановлення рамки, за стандартною схемою кольорів – чорна, колір фону та біла), SS_BLACKRECT, SS_GRAYRECT та SS_WHITERECT (заповнен-

ня елементу кольором, за стандартною схемою кольорів – чорним, кольором фону та білим), SS_CENTER (розміщення тексту посередині прямокутника елемента керування. Якщо текст не вміщається в один рядок – він автоматично переноситься на інший рядок, при цьому слова, довші ніж ширина елемента – обрізаються з обох країв), SS_CENTERIMAGE (розміщення зображення по центру елемента керування. Якщо зображення не вміщується – воно обрізується. Якщо в елементі знаходиться текст – він розміщується симетрично до верхнього та нижнього країв елемента), SS_ENDELLIPSIS та SS_WORDELLIPSIS (якщо текст не вміщується в елемент, він обрізається, а в кінці зображуються три крапки), SS_ETCHEDFRAME (зображення трохи утопленої рамки для елемента керування), SS_ETCHEDHORZ та SS_ETCHEDVERT (зображення горизонтальних та вертикальних рамок), SS_LEFT, SS_RIGHT та SS_LEFTNOWORDWRAP (вирівнювання тексту по лівому (правому) краю. У перших двох випадках у разі необхідності текст розбивається на декілька рядків, у третьому – текст, що не вміщується в елемент – обрізається), SS_NOTIFY (дозвіл посилати батьківському вікну повідомлення STN_CLICKED, STN_DBLCLK, STN_DISABLE та STN_ENABLE), SS_PATHELLIPSIS (заміна символів усередині тексту трьома крапками в разі, якщо текст не вміщується в елемент керування. При використанні імен файлів текст замінюється по можливості в частині після останнього символу «\»), SS_REALSIZECONTROL (зменшення або збільшення зображення під розміри елемента керування. Використовується, починаючи з операційної системи Windows XP), SS_SIMPLE (створення простого елементу з текстом, вирівняним по лівому краю), SS_SUNKEN (зображення трохи утопленого елемента).

Для роботи зі статичним елементом керування використовуються такі повідомлення: STM_GETICON та STM_SETICON (отримання та установка дескриптора ярлика, що асоційовано з елементом керування), STM_GETIMAGE та STM_SETIMAGE (отримання та установка дескриптора зображення, що асоційовано з елементом керування). Якщо встановлено стиль SS_NOTIFY, можна використовувати такі повідомлення: STN_CLICKED (натискання в елементі), STN_DBLCLK (подвійне натис-

кання в елементі), STN_DISABLE та STN_ENABLE (повідомлення про заборону та дозвіл елемента).

Приклад 3.5. У наведеному фрагменті коду виконується створення трьох статичних елементів керування: рядка символів по центру прямокутника елемента керування з заміною символів, ярлика, розмір якого розтягнуто до розмірів статичного елемента, та тексту в трохи утопленому прямокутнику. При натисканні на ліву клавішу миші на другому статичному елементі, у третьому статичному елементі змінюється текст. Результат виконання наведений на рис. 3.6.

```
// фрагмент опису змінних
HWND  hwnd1, hwnd2, hwnd3;
#define STATIC_1  111
#define STATIC_2  112
#define STATIC_3  113

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
static HICON hIcon;
hIcon = LoadIcon (NULL, IDI_APPLICATION);
hwnd1 = CreateWindow
      ("STATIC","c:\\Program\\abcde.exe",WS_CHILD|
      SS_CENTER| SS_CENTERIMAGE |SS_PATHELLIPSIS ,
      10,10,160,40,hwnd,(HMENU)STATIC_1,NULL,NULL);
ShowWindow (hwnd1, SW_SHOW);
hwnd2 = CreateWindow
      ("STATIC","Static2",WS_CHILD|SS_ICON|SS_NOTIFY|
      SS_REALSIZECONTROL,10,65,160,40,
      hwnd,(HMENU)STATIC_2,NULL,NULL);
SendMessage(hwnd2,STM_SETICON,(LPARAM)hIcon,0);
ShowWindow (hwnd2, SW_SHOW);
hwnd3 = CreateWindow
      ("STATIC","Static3",WS_CHILD|SS_SUNKEN ,
```

```
      10,120,160,20,hwnd,(HMENU)STATIC_3,NULL,NULL);
ShowWindow (hwnd3, SW_SHOW);
break;

// обробка натискання на другому статичному елементі
case WM_COMMAND:
switch LOWORD(wParam)
{
case STATIC_2:
{
switch (HIWORD(wParam))
      {
      case STN_CLICKED:
      SendMessage(hwnd3, WM_SETTEXT, 0,
            (LPARAM)"NewText");
      break;
}
}
}
break;
```

Рисунок 3.6 – Приклад створення статичного елемента керування

Меню. Меню є вікном з системним класом 32768, створити таке вікно за допомогою функції **CreateWindow** неможливо. Для роботи з цим вікном зарезервовано цілий ряд функцій, серед яких розглянемо тільки ті, що найчастіше використовуються для роботи з меню.

AppendMenu (додавання елемента в меню. Елемент додається в кінець існуючого меню, за допомогою прапорців задаються властивості елемента. Після додавання елементів у меню обов'язково необхідно викликати функцію **DrawMenuBar**), **CheckMenuItem** (виділення або спростовування виділення елемента меню), **CheckMenuRadioItem** (виділяє елемент меню-прапорець, та знімає виділення для інших елементів-прапорців в указаній групі), **CreateMenu** (створення порожнього меню), **CreatePopupMenu** (створення порожнього контекстного меню або підменю), **DeleteMenu** (видалення елемента з меню за вказаною позицією), **DestroyMenu** (знищення меню і всіх його підменю та звільнення пам'яті, яку воно займало), **DrawMenuBar** (перемальовування меню), **EnableMenuItem** (заборона або дозвіл роботи з елементом меню), **GetMenu** (отримання для вікна дескриптора його меню), **GetMenuItemCount** (отримання кількості елементів у меню), **GetMenuItemInfo** (отримання інформації про елемент меню, у тому числі – назви елемента), **GetSubMenu** (отримання дескриптора на підменю), **GetSystemMenu** (отримання дескриптора системного меню для його модифікації), **HiliteMenuItem** (виділення елемента меню), **InsertMenu** (додавання елемента в указану позицію меню. Якщо позиція дорівнює -1 – елемент додається в кінець меню), **InsertMenuItem** (додавання елемента в указану позицію меню з розширеним налаштуванням параметрів), **ModifyMenu** (зміна існуючого елемента меню), **RemoveMenu** (видалення меню або підменю), **SetMenu** (встановлення меню для вказаного вікна. Для вікна може бути тільки одне меню, ця функція міняє меню для вікна, але не знищує попереднє меню в пам'яті. Вікно не може бути дочірнім), **SetMenuItemInfo** (встановлює інформацію для елемента меню), **TrackPopupMenu** та **TrackPopupMenuEx** (відображують контекстне меню в будь якій позиції вікна).

Приклад 3.6. У наведеному фрагменті коду демонструється створення меню з трьох елементів. Третій елемент є меню, що випадає, з чотирьох пунктів, розділених лінією, при цьому третій та четвертий пункти обрані різними варіантами. Елементи в меню додаються різними методами. Створене контекстне меню з меню, що випадає, яке може викликатися натисканням на праву клавішу миші. При виборі елемента `MenuItem2` виконується відображення діалогового вікна **MessageBox**. Також додано 1 елемент у системне меню програми.

```
// фрагмент опису змінних
HMENU       hMenu, hMenu1, hSMenu;
#define     MENUITEM_1  118
#define     MENUITEM_2  119
#define     MENUITEM_3  120
#define     MENUITEM_4  121
#define     MENUITEM_0  122

// фрагмент віконної функції вікна hwnd
case WM_CREATE:
// створення вікна меню з перевіркою на помилки
hMenu = CreateMenu () ;
if (!hMenu)
{
      MessageBox (hwnd, "Cannot Create Menu",
             "ERROR", MB_OK) ;
      DestroyWindow (hwnd) ;
}
hMenu1 = CreatePopupMenu () ;
if (!hMenu1)
{
      MessageBox (hwnd, "Cannot Create Menu",
            "ERROR", MB_OK) ;
      DestroyWindow (hwnd) ;
```

```
}
// перший варіант створення
InsertMenu(hMenu, 1, MFT_STRING, (UINT)MENUITEM_0,
       "Menu0") ;
// другий варіант створення
MENUITEMINFO mii;
mii.cbSize = sizeof(mii);
mii.fMask = MIIM_STRING|MIIM_ID|MIIM_FTYPE;
mii.wID = MENUITEM_0;
mii.fType = MFT_STRING;
mii.dwTypeData = (LPSTR)"Menu";
InsertMenuItem(hMenu, 0, TRUE, &mii);

// створення меню, що випадає
InsertMenu(hMenu, 0xFFFFFFFF, MF_BYPOSITION |
      MFT_STRING | MF_POPUP, (UINT)hMenu1, "Menu1") ;
// додавання елементів у меню
InsertMenu(hMenu1, 0xFFFFFFFF, MF_BYPOSITION |
      MFT_STRING, MENUITEM_1, "MenuItem1") ;
InsertMenu(hMenu1, 0xFFFFFFFF, MF_BYPOSITION |
      MFT_STRING, MENUITEM_2, "MenuItem2");
DrawMenuBar (hwnd);
// додавання лінії розділювача
AppendMenu (hMenu1, MFT_SEPARATOR, 0, NULL );
AppendMenu (hMenu1, MFT_STRING | MFT_RADIOCHECK |
      MFS_CHECKED, MENUITEM_3, "MenuItem3");
AppendMenu (hMenu1, MFT_STRING | MFS_CHECKED,
      MENUITEM_4, "MenuItem4");
SetMenu (hwnd, hMenu) ;

// отримання дескриптора системного меню
hSMenu=GetSystemMenu (hwnd,FALSE);
// додавання елемента в системне меню
```

```
AppendMenu (hSMenu, MFT_STRING, MENUITEM_4,
      "MenuItem4");

// встановлення перемикачів у меню
CheckMenuRadioItem(hMenu1, 0,1,1, MF_BYPOSITION);

// обробка натискання на праву клавішу миші
case WM_CONTEXTMENU:
      SetForegroundWindow(hwnd);
      // відображення контекстного меню
      TrackPopupMenu(hMenu1,  TPM_RIGHTBUTTON |
            TPM_TOPALIGN | TPM_LEFTALIGN,
            LOWORD(lParam), HIWORD(lParam), 0,
            hwnd, NULL);
break;

// обробка вибору елемента меню MenuItem2
case WM_COMMAND:
switch LOWORD(wParam)
{
      case MENUITEM_2:
      {
      MessageBox(hwnd, "Selected 2", "Menu", MB_OK);
      break;
      }
}
break;
```

На рис. 3.7 зображено зовнішній вид програми з меню: при виборі елемента меню та при відображенні контекстного меню.

На рис. 3.8 зображено змінене системне меню для програми.

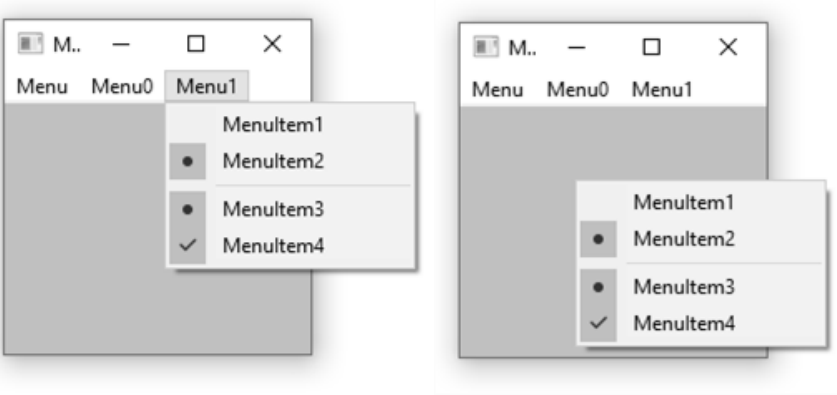


Рисунок 3.7 – Приклад створення меню

Рисунок 3.8 – Приклад додавання елемента в системне меню

Обробка повідомлень від вікон зі стандартними класами та пристроїв введення.

Для того, щоб оперувати вікнами в програмі, необхідно оброблювати повідомлення, які можуть надходити від них або від пристроїв введення. Обробка повідомлень виконується в віконній функції зворотного виклику, яка встановлена для батьківського вікна. У прикладах програм, що наведені вище, для кожного типу стандартного вікна наведені певні (але не всі) мо-

жливі дії по обробці повідомлень. У таблиці 3.1 наведені деякі найбільш часто вживані повідомлення (`uMsg`), які передаються у віконну функцію від різноманітних вікон та пристроїв введення, їх параметри (`wParam` та `lParam`) і призначення.

Таблиця 3.1 – Повідомлення та їх параметри

Призначення повідомлення	`uMsg`	`wParam`	`lParam`
1	2	3	4
Натиснута символьна клавіша (результат транслювання повідомлення WM_KEYDOWN функцією **TranslateMessage**)	WM_CHAR	Код символу TCHAR	Дані про клавішу (скан-код та інше)
Натиснута несистемна клавіша (не натиснута клавіша ALT)	WM_KEYDOWN	Віртуальний код (деякі коди наведені в табл. 3.2)	Дані про клавішу (скан-код та інше)
Натиснута гаряча комбінація клавіш	WM_HOTKEY	Ідентифікатор комбінації клавіш	Індикатор натиснутих віртуальних клавіш
Зображення контекстного меню	WM_CONTEXT MENU	Дескриптор вікна, в якому натиснута права клавіша миші	Координати миші
Повідомлення від системного меню	WM_SYS COMMAND	Команда системного меню	Координати миші

Продовження табл. 3.1

1	2	3	4
Натиснута ліва (права) клавіша миші	WM_LBUTTON DOWN WM_RBUTTON DOWN	Індикатор натиснутих віртуальних клавіш	Координати миші xPos = GET_X_ LPARAM (lParam); yPos = GET_Y_ LPARAM (lParam);
Натискання на кнопку, вибір меню	WM_COMMAND	Ідентифікатор вікна або пункту меню	Дескриптор вікна
Сповіщення про події	WM_COMMAND	Старші 2 байти – тип сповіщення, молодші 2 байти - ідентифікатор вікна	Дескриптор вікна
Зображення контекстного меню	WM_CONTEXT MENU	Дескриптор вікна, в якому натиснута права клавіша миші	Координати миші
Повідомлення від системного меню	WM_SYS COMMAND	Команда системного меню	Координати миші

Таблиця 3.2 – Деякі віртуальні коди клавіш

Символьна константа	Код	Еквівалент
1	2	3
VK_LBUTTON	01	Ліва клавіша миші
VK_RBUTTON	02	Права клавіша миші

Продовження табл. 3.2

1	2	3
VK_CANCEL	03	Комбінація Control-break
VK_BACK	08	Клавіша BACKSPACE
VK_TAB	10	Клавіша TAB
VK_RETURN	0D	Клавіша ENTER
VK_SHIFT	10	Клавіша SHIFT
VK_CONTROL	11	Клавіша CTRL
VK_MENU	12	Клавіша ALT
VK_CAPITAL	14	Клавіша CAPS LOCK
VK_ESCAPE	1B	Клавіша ESC
VK_SPACE	20	Клавіша – пробіл
VK_PRIOR	21	Клавіша PAGE UP
VK_NEXT	22	Клавіша PAGE DOWN
VK_END	23	Клавіша END
VK_HOME	24	Клавіша HOME
VK_LEFT	25	Клавіша – стрілка вліво
VK_UP	26	Клавіша – стрілка вверх
VK_RIGHT	27	Клавіша – стрілка вправо
VK_DOWN	28	Клавіша – стрілка вниз
VK_INSERT	2D	Клавіша INS
VK_DELETE	2E	Клавіша DEL
VK_F1 … VK_F12	70 … 7B	Клавіші F1 … F12
VK_NUMLOCK	90	Клавіша NUM LOCK
VK_SCROLL	91	Клавіша SCROLL LOCK
VK_LSHIFT	A0	Клавіша – лівий SHIFT
VK_RSHIFT	A1	Клавіша – правий SHIFT
VK_LCONTROL	A2	Клавіша – лівий CONTROL
VK_RCONTROL	A3	Клавіша – правий CONTROL
VK_LMENU	A4	Клавіша – лівий ALT
VK_RMENU	A5	Клавіша – правий ALT

Нижче наведений *приклад* 3.7 – обробка повідомлення від системного меню, що забороняє виконання операції по згортанню та розкриванню на весь екран вікна програми (точніше – ігнорування цих операцій шляхом встановлення замість стандартних дій за замовчанням порожнього оператора), а також обробка повідомлення при натисканні клавіші на клавіатурі F2 – виведення повідомлення у вікні **MessageBox**.

```
case WM_SYSCOMMAND :
      switch (wParam) {
            case SC_MAXIMIZE : case SC_MINIMIZE :
                  return 0;
      }
break;
case WM_KEYDOWN :
      switch (wParam) {
            case VK_F2 :
                  MessageBox (hwnd, "F2 Pressed",
                        "Pressed Virtual Key", MB_OK);
                  break;
      }
      return 0;
break;
```

3.2. Контрольні питання

1. Що відбудеться, якщо видалити цикл обробки повідомлень з віконної програми?

2. Яка частина програми або операційної системи є ініціатором виклику віконної процедури?

3. Чи може віконна програма не мати віконної процедури? Чому?

4. Яким чином пов’язується віконна процедура з конкретним вікном?

5. Чи може бути для різних вікон одна віконна процедура?

6. Чи обов’язково всі дочірні вікна створювати при обробці повідомлення `WM_CREATE` в батьківському вікні?

3.3. Індивідуальні завдання

Розробити віконну програму, що виконує поставлене завдання. Для роботи з елементами графічного інтерфейсу використовувати тільки функції WinAPI (використання засобів MFC або робота з ресурсами не допускається). Для оформлення програми обов'язково використовувати статичні елементи керування (клас вікна – STATIC).

1) Розробити програму, в якій при виборі елемента з LISTBOX (п'ять елементів) і при натисканні на кнопку виводиться результат з повідомленням про вибраний елемент у поле EDIT, захищеному від редагування.

2) Розробити програму, в якій при виборі елемента з COMBOBOX (п'ять елементів) і натисканні лівої клавіші миші в певному регіоні виводиться результат з повідомленням про вибраний елемент у **MessageBox**.

3) Розробити програму, в якій при натисканні на одну з кнопок (п'ять елементів) і натисканні на клавіатурі клавіші F5 виводиться результат з повідомленням про вибраний елемент у **MessageBox**.

4) Розробити програму, в якій при натисканні певної комбінації «гарячих» клавіш (не менше п'яти комбінацій) і при натисканні на кнопку виводиться результат з повідомленням про останню натиснуту комбінацію в **MessageBox** (задати «гарячу» комбінацію клавіш можна за допомогою функції **RegisterHotKey**).

5) Розробити програму, в якій можна вводити текст у поле EDIT, при виборі пункту меню CLEAR – очищати поле введення, при виборі пункту меню ACCEPT – додавати текст з поля EDIT у LISTBOX.

6) Розробити програму, в якій при натисканні на ліву клавішу миші створюється меню з трьома пунктами і двома підпунктами в кожному з пунктів, а при натисканні на праву клавішу миші – створюється кнопка з реакцією – завершення процесу. У разі двох і більше підряд натискань на одну і ту ж клавішу миші виводити повідомлення в **MessageBox**.

7) Розробити програму, в якій при виборі елементу з LISTBOX (п'ять елементів) вибирається відповідний перемикач RADIOBUTTON (п'ять елементів) і навпаки – при виборі перемикача, вибирається елемент з LISTBOX.

8) Розробити програму, в якій при виборі елементу з COMBOBOX (п'ять елементів) результат з повідомленням про вибраний елемент додається і відображується в LISTBOX.

9) Організувати при натисканні на кнопку запуск програми – блокнота (**notepad.exe**), яка повинна завершуватися через 15 секунд. Кількість запущених копій програми – блокнота не обмежується. Використовувати нитки. Для взаємодії з користувачем використовувати кнопки. В елемент LISTBOX накопичувати статистику про дії програми.

10) Написати програму, яка накопичує статистику натискань на клавіші клавіатури і миші. Результат відображувати по центру вікна і динамічно його корегувати (використовувати функцію **DrawText**). Вихід з програми виконати за допомогою меню.

11) Організувати при натисканні на одну кнопку запуск програми – блокнота (**notepad.exe**), яка повинна завершуватися при натисканні на іншу кнопку. Для взаємодії з користувачем використовувати кнопки. Передбачити запуск та завершення програми – блокнота через системне меню програми.

12) Розробити програму, в якій введений текст переводиться до верхнього регістру. Введення тексту виконувати в полі EDIT, початок роботи по натисканню на кнопку, накопичення результатів – у LISTBOX.

13) Розробити програму, в якій введений текст переводиться до нижнього регістру. Введення тексту виконувати в поле EDIT, початок роботи по натисканню на клавіатурі клавіші F2, результат виводити по центру вікна (використовувати функцію **DrawText**).

14) Розробити програму, в якій створюється меню з елементів, текст яких задається послідовно в полі EDIT і підтверджується при натисканні на праву клавішу миші в заданому регіоні.

15) Розробити програму, в якій введений текст переводиться до верхнього або нижнього регістру в залежності від стану перемикачів RADIOBUTTON. Роботу виконувати при натисканні «гарячої» комбінації клавіш (задати «гарячу» комбінацію клавіш можна за допомогою функції **RegisterHotKey**).

4. СИНХРОНІЗАЦІЯ НИТОК УСЕРЕДИНІ ПРОЦЕСУ В ОС WINDOWS

Мета: вивчити можливості застосування механізмів синхронізації ниток одного процесу в ОС Windows.

4.1. Теоретичні відомості

Як було вказано в темі «Процеси, нитки та волокна в ОС Windows» – процесом (process) називається екземпляр програми, завантажений для виконання в пам'ять. Цей екземпляр може створювати нитки (thread), які являють собою послідовність інструкцій на виконання. Важливо розуміти, що виконуються не процеси, а саме нитки. Причому кожний процес має хоча б одну нитку. Ця нитка називається головною (основною) ниткою прикладної програми.

Практично завжди ниток значно більше, ніж фізичних процесорів для їх виконання, тому нитки насправді виконуються не одночасно, а по черзі (розподіл процесорного часу відбувається саме між нитками). Але перемикання між ними відбувається так часто, що здається ніби вони виконуються паралельно.

Залежно від ситуації нитки можуть перебувати в трьох базових станах (це спрощена модель, у конкретних операційних системах станів процесів значно більше. В ОС Windows для робочих станцій – 7 основних станів, Windows Server – 8 основних станів). По-перше, нитка може виконуватись, коли їй виділений процесорний час, тобто вона знаходиться в стані активності. По-друге, вона може бути неактивною і очікувати виділення процесора, тобто бути в стані готовності. І є ще третій, також дуже важливий стан – стан блокування. Коли нитка заблокована, їй взагалі не виділяється процесорний час. За звичаєм блокування використовується на час очікування процесом якої-небудь події. Під час виникнення цієї події нитка автоматично переводиться зі стану блокування в стан готовності. Наприклад, якщо одна нитка виконує обчислення, то друга повинна чекати результатів, щоб зберегти їх на диску. Друга нитка могла б використати цикл типу «`while (!isCalcFinished) continue;`», але легко переконатись

практично, що під час виконання цього циклу процесор зайнятий на 100 % (це називають активним очікуванням). Саме таких циклів треба по можливості уникати, і в цьому дуже допомагає механізм блокування. Друга нитка може заблокувати себе до тих пір, поки перша не зафіксує ситуацію, яка сигналізуватиме про те, що обчислення завершене.

В ОС Windows реалізована багатозадачність на основі витіснення. Це означає, що в будь який момент часу система може перервати виконання поточної нитки і передати управління іншій. Раніше, в ОС Windows 3.1, використовувався спосіб, який називається кооперативною багатозадачністю: система чекала, поки нитка сама не передасть їй управління. І саме тому при зависанні однієї прикладної програми доводилось перезавантажувати комп'ютер.

Усі нитки, які належать одному і тому ж процесу, розділяють деякі спільні ресурси, а саме адресний простір оперативної пам'яті або відкриті файли. Ці ресурси належать процесу, а отже і кожній його нитці. Тобто, кожна нитка може використовувати ці ресурси без будь-яких обмежень. Але, якщо одна нитка ще не закінчила використання якогось спільного ресурсу, а система переключилась на іншу нитку, якій потрібен той самий ресурс, то результат виконання цих ниток може занадто сильно відрізнятись від очікуваного. Такі конфлікти можуть виникати і поміж нитками, які належать різним процесам. Завжди, коли дві або більше ниток використовують який-небудь спільний ресурс, виникає ця проблема.

Приклад 4.1. Несинхронізована робота ниток: якщо тимчасово призупинити виконання нитки, яка виводить дані на екран (використовуємо клавішу на клавіатурі «Pause»), фонова нитка заповнення масиву даними буде продовжувати виконуватись. На рис. 4.1 наведено виведення одного з можливих варіантів результатів виконання програми.

```
#include <windows.h>
#include <stdio.h>
int a[5];
HANDLE hThr;
```

```
unsigned long uThrID;

void Thread (void* pParams)
{
int i, num = 0;
while (1)
      {
      for (i=0; i<5; i++) a[i] = num;
      num++;
      }
}

int main( void )
{
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1) printf("%d %d %d %d %d\n",
           a[0],a[1],a[2],a[3],a[4]);
return 0;
}
```

```
0 0 0 0 0
0 0 0 0 0
26154 26154 26154 26153 26153
30559 30558 30558 30558 30558
34479 34478 34478 34478 34478
38120 38120 38119 38119 38119
41676 41676 41675 41675 41675
45190 45190 45190 45190 45190
48689 48689 48689 48689 48688
52251 52251 52251 52251 52251
```

Рисунок 4.1. Екран результатів виконання програми

На рис. 4.2 наведено властивості процесу при виконанні програми з прикладу в момент призупинки головної нитки (використовується безкоштовна утиліта Process Explorer https://learn.microsoft.com/uk-ua/sysinternals/). Як видно – фонова нитка (в якій виконується нарощування значень елементів масиву) продовжує виконуватися. При продовженні виконання головної нитки можна побачити різке збільшення значень елементів масиву залежно від часу призупинки.

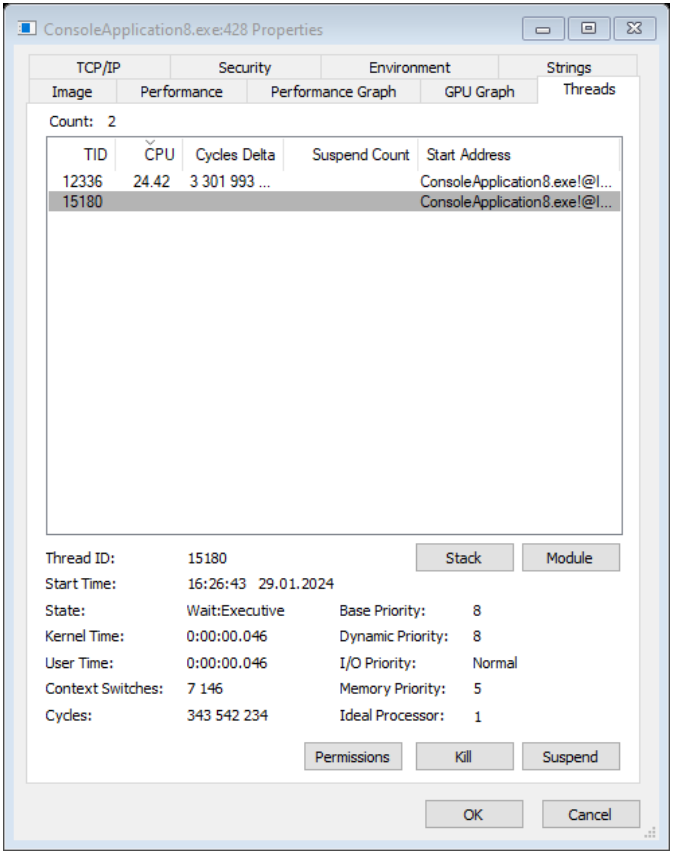


Рисунок 4.2. Властивості процесу для запущеної програми

Так як нитки виконуються повністю незалежно одна від одної, на рис. 4.1 можна побачити ситуацію, коли фонова нитка заповнення масиву змінила не всі елементи масиву, а основна нитка вже виконала виведення значень елементів на екран (наприклад рядок на рис. 4.1: 26154 26154 26154 26153 26153), тобто виконання фонової нитки було перерване під час заповнення масиву.

Для виконання контрольованої сумісної роботи над спільними ресурсами потрібен механізм, який би дозволив ниткам погоджувати використання ними цих спільних ресурсів. Цей механізм називається «синхронізація ниток» (thread synchronization).

Реалізацію цього механізму забезпечує деяка сукупність об'єктів ОС. Вони створюються і управляються програмно, являються спільними для всіх ниток у системі (декотрі – для ниток, які належать одному і тому ж процесу) і використовуються для координації доступу до ресурсів. В якості ресурсу може виступати все, що може бути спільним для двох і більшої кількості ниток – файл на диску, порт, запис у базі даних, об'єкт GDI (Graphics Device Interface) і навіть глобальна змінна програми (вона може бути доступною із ниток, які належать одному і тому ж процесу).

Синхронізація в ОС Windows.

Існує декілька об'єктів ядра, а саме об'єктів синхронізації, які спеціально використовуються щоб забезпечити синхронізацію. Найбільш важливі з них – це взаємовиключення (mutex), критична секція (critical section), подія (event) і семафор (semaphore). Кожний з цих об'єктів реалізує свій спосіб синхронізації. Потреба в синхронізації може виникати для самих процесів і ниток (коли одна нитка повинна чекати завершення іншої нитки або процесу), а також при роботі з файлами, комунікаційними пристроями, консольним введенням і сповіщеннями про зміни.

Будь-який об'єкт синхронізації може знаходитись у так званому сигнальному стані. Для кожного типу об'єкта синхронізації цей стан має відмінну суть. Нитки можуть перевіряти поточний стан об'єкта синхронізації та/або чекати, коли цей стан зміниться, і, таким чином, узгоджувати свої

дії. При цьому гарантується, що, коли нитка працює з об'єктом синхронізації (створює його, змінює його стан тощо), ОС не переривається її виконання, поки нитка не завершить цю дію. Таким чином, усі скінченні операції з об'єктами синхронізації являються атомарними, тобто неподільними.

Робота з об'єктами синхронізації.

Щоб створити той чи інший об'єкт синхронізації, виконується виклик спеціальної функції WinAPI типу **Create**... (наприклад, **CreateMutex**). Цей виклик повертає дескриптор об'єкта синхронізації (тобто HANDLE), який можуть використовувати всі нитки, що належать даному процесу. Є можливість отримати доступ до об'єкта синхронізації із іншого процесу – або отримавши в спадок дескриптор цього об'єкта, або, що краще, скориставшись викликом функції відкриття об'єкта **Open**.... Після цього виклику процес отримає дескриптор, який можна використати в подальшій роботі з цим об'єктом. Об'єкту синхронізації, якщо тільки він не призначений для використання усередині тільки одного процесу, обов'язково присвоюється ім'я. Імена всіх цих об'єктів повинні бути різними (навіть якщо ці об'єкти різного типу). Не можна, наприклад, створити об'єкти подія і семафор з одним і тим же ім'ям.

Якщо дескриптор об'єкту синхронізації вже відомий, його можна використати, щоб, зокрема, визначити поточний стан об'єкта. Існують так звані функції очікування. Частіше за все використовується функція **WaitForSingleObject**. Ця функція приймає два параметри, перший з яких – дескриптор об'єкта синхронізації, а другий – час очікування в мілісекундах. Функція повертає: значення WAIT_OBJECT_0, якщо об'єкт синхронізації знаходиться в сигнальному стані; значення WAIT_TIMEOUT, якщо закінчився час очікування; значення WAIT_ABANDONED, якщо об'єкт синхронізації не був звільнений до того, як нитка, що ним володіє, завершилась. Якщо вказаний час очікування дорівнює нулю, функція повертає результат негайно, інакше вона чекає на протязі вказаного проміжку часу. Якщо стан об'єкта синхронізації стане сигнальним до закінчення вказаного проміжку часу, функція поверне значення WAIT_OBJECT_0, а інакше функція поверне значення WAIT_TIMEOUT. Якщо в якості значення проміжку часу вка-

зана символічна константа INFINITE, то функція буде чекати необмежено довго, аж поки стан об'єкта синхронізації не стане сигнальним.

Дуже важливим є той факт, що звернення до функції очікування блокує поточну нитку, тобто поки нитка знаходиться в стані очікування, їй не виділяється процесорний час.

Критичні секції. Об'єкт «критична секція» допомагає програмісту виділити частину коду, де нитка отримує доступ до розподілюваного ресурсу, і запобігти одночасному використанню ресурсу. Перед використанням ресурсу нитка входить у «критичну секцію», а саме – викликає функцію **EnterCriticalSection**. Якщо після цього яка-небудь інша нитка спробує війти в ту ж саму «критичну секцію», її виконання призупиниться до тих пір, поки перша нитка не покине «критичну секцію» за допомогою виклику функції **LeaveCriticalSection**. Цей об'єкт синхронізації використовується тільки для ниток одного і того ж процесу. Створення критичної секції виконується за допомогою функції **InitializeCriticalSection**, знищення – **DeleteCriticalSection**. Сама «критична секція» містить недокументовану структуру ОС, до якої прямий доступ заборонений. Порядок входження в «критичну секцію» не визначений. У разі постійно зайнятої «критичної секції» функція **EnterCriticalSection** насправді блокує нитку тільки на деякий час, через який генерується виключення. Довжина часу задається в реєстрі за маршрутом **HKEY_LOCAL_MACHINE \ System \ CurrentControlSet \ Control \ Session Manager** у параметрі **CriticalSectionTimeout**, за замовчанням вона дорівнює 2 592 000 секунд (30 діб). Для багатопроцесорних обчислювальних систем виклик функції **EnterCriticalSection** у разі зайнятої «критичної секції» не одразу викликає блокування нитки, а виконується деяка кількість спроб увійти в «критичну секцію» (активне очікування) і лише при невдачі нитка блокується. Це пов'язано з тим, що при блокуванні (перехід нитки із стану активності в стан блокування) витрачається близько 1000 тактів процесора. І в разі коли в цей час (час переходу в стан блокування) «критична секція» звільниться, цей перехід тільки знизить швидкість роботи програми. Для використання активного очікування використовується функція **InitializeCriticalSectionAndSpinCount**, в якій в

параметрі dwSpinCount задається кількість спроб отримання доступу до «критичної секції» до переведення нитки в стан блокування. Для зміни кількості спроб використовується функція **SetCriticalSectionSpinCount**. Існує також функція **TryEnterCriticalSection**, яка перевіряє, чи зайнята «критична секція» на даний час. За її допомогою нитка в процесі очікування доступу до ресурсу може не блокуватись, а виконувати якісь корисні дії.

Параметри виклику вказаних функцій:

```
VOID InitializeCriticalSection (
 LPCRITICAL_SECTION lpCriticalSection // критична
                                      // секція
);

BOOL InitializeCriticalSectionAndSpinCount (
 LPCRITICAL_SECTION lpCriticalSection,// критична
                                      // секція
 DWORD dwSpinCount  // кількість спроб входу в
                  // критичну секцію до блокування
);

DWORD SetCriticalSectionSpinCount (
 LPCRITICAL_SECTION lpCriticalSection,// критична
                                      // секція
 DWORD dwSpinCount // кількість спроб входу
                  // в критичну секцію до блокування
);

VOID EnterCriticalSection (
 LPCRITICAL_SECTION lpCriticalSection // критична
                                      // секція
);
```

```
VOID LeaveCriticalSection (
 LPCRITICAL_SECTION lpCriticalSection // критична
                                      // секція
);

BOOL TryEnterCriticalSection (
 LPCRITICAL_SECTION lpCriticalSection // критична
                                      // секція
);

VOID DeleteCriticalSection (
 LPCRITICAL_SECTION lpCriticalSection // критична
                                      // секція
);
```

Приклад 4.2. Синхронізація ниток за допомогою «критичних секцій».

```
#include <windows.h>
#include <stdio.h>
CRITICAL_SECTION cs;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread (void* pParams)
{
int i, num = 0;
while (1)
      {
      EnterCriticalSection (&cs);
      for (i=0; i<5; i++) a[i] = num;
      num++;
      LeaveCriticalSection (&cs);
```

```
      }
}

int main( void )
{
InitializeCriticalSection (&cs);
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1)
      {
      EnterCriticalSection (&cs);
      printf("%d %d %d %d %d\n",
            a[0],a[1],a[2],a[3],a[4]);
      LeaveCriticalSection (&cs);
      }
return 0;
}
```

При роботі програми можна побачити, що в межах одного рядка виводяться однакові значення (на відміну від програми з несинхронізованою роботою ниток, де значення могли змінюватися). Тобто код всередині «критичної секції» виконується повністю або в головній нитці або у фоновій.

Для демонстрації впливу активного очікування замість виклику функції **InitializeCriticalSection** використаємо такий фрагмент коду:

```
InitializeCriticalSectionAndSpinCount (&cs,
      0x7FFFFFFF); // задаємо максимальну кількість
                   // спроб входу до критичної секції
// SetCriticalSectionSpinCount (&cs, 0x0000);
      // Якщо убрати коментар – кількість спроб
      // активного очікування дорівнюватиме 0
```

У такому разі в першому випадку навантаження процесору процесом буде приблизно 50 %, а при використанні функції **SetCriticalSectionSpinCount** (виклик закоментовано у фрагменті коду) – менше 10 %. *Примітка*: для роботи програми необхідно задати мінімальну версію ОС – Windows 2000 (див. текст програми з волокнами з теми «Процеси, нитки та волокна в ОС Windows»).

Інший варіант – демонстрація використання функції **TryEnterCriticalSection**. Для цього необхідно в тексті програми замінити виклик функції **EnterCriticalSection** в основній нитці на фрагмент коду:

```
if (!TryEnterCriticalSection (&cs))
      {
      printf("CS Blocked!\n");
      continue;
      }
```

Таким чином можна відстежувати ситуації, коли «критична секція» зайнята, але використання активного очікування в цьому випадку значно навантажує процесор.

Взаємовиключення. Об’єкти взаємовиключення, а саме – м’ютекси (mutex – від MUTual EXclusion), дозволяють координувати взаємне виключення доступу до ресурсу, що розподіляється. Сигнальний стан об’єкта (тобто стан «установлений») відповідає моменту часу, коли об’єкт не належить жодній нитці, і його можна «захопити». І навпаки, стан «скинутий» (не сигнальний) відповідає моменту, коли яка-небудь нитка вже володіє цим об’єктом. Доступ до об’єкта дозволяється, коли нитка, яка володіє об’єктом, звільнить його.

Дві (або більше) нитки можуть створити м’ютекс з одним і тим же ім’ям, викликавши функцію **CreateMutex**. Перша нитка дійсно створює м’ютекс, а наступні лише отримують дескриптор уже існуючого об’єкта. Це дає змогу декільком ниткам отримати дескриптор одного і того ж м’ютекса, що звільняє програміста від необхідності встановлювати, яка

саме нитка в дійсності створила м'ютекс. Якщо використовується такий підхід, бажано ознаці `bInitialOwner` присвоїти значення FALSE, інакше виникнуть певні труднощі при необхідності встановлення дійсного створювача м'ютекса: на відміну від інших об'єктів ядра ОС права на користування м'ютексом повністю належать нитці, яка володіє ним. І, відповідно, звільнити м'ютекс може тільки його володар.

Декілька ниток можуть отримати дескриптор одного й того ж м'ютекса, що робить можливою взаємодію між процесами. Можна використати такі механізми такого підходу:

- дочірній процес, створений при виклику функції **CreateProcess**, може наслідувати дескриптор м'ютекса у випадку, якщо при створенні м'ютекса функцією **CreateMutex** був указаний параметр `lpMutexAttributes`;
- нитка може отримати дублікат існуючого м'ютекса за допомогою функції **DuplicateHandle**;
- нитка може вказати ім'я існуючого м'ютекса під час виклику функції **OpenMutex** або функції **CreateMutex**.

Для того щоб сповістити, що взаємовиключення належить поточній нитці, треба викликати одну з функцій очікування. Нитка, якій належить об'єкт, може його «захоплювати» повторно скільки завгодно разів (це не призведе до самоблокування), але стільки ж разів вона повинна буде його звільнювати за допомогою функції **ReleaseMutex**. Для закриття об'єкту необхідно скористатися функцією **CloseHandle**.

Для синхронізації ниток одного й того ж процесу більш ефективне використання «критичних секцій», ніж м'ютексів.

Основні функції та їх параметри для роботи з м'ютексами:

```
HANDLE CreateMutex (
 LPSECURITY_ATTRIBUTES lpMutexAttributes,  // покажчик
            // на структуру атрибутів захисту
 BOOL bInitialOwner,  // початкове володіння м'ютексом
 LPCTSTR lpName       // ім'я м'ютексу
);
```

```
HANDLE OpenMutex (
 DWORD dwDesiredAccess,  // тип доступу
 BOOL bInheritHandle,    // ознака наслідування
 LPCTSTR lpName          // ім'я м'ютексу
);

BOOL ReleaseMutex (
 HANDLE hMutex   // дескриптор м'ютексу
);
```

Приклад 4.3. Синхронізація ниток за допомогою м'ютексів.

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

HANDLE hMutex;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread (void* pParams)
{
int i, num = 0;
while (1)
      {
      WaitForSingleObject (hMutex, INFINITE);
      for (i=0; i<5; i++) a[i] = num;
      num++;
      ReleaseMutex (hMutex);
      }
}
```

```
int main (void)
{
hMutex = CreateMutex (NULL, FALSE, NULL);
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1)
      {
      WaitForSingleObject (hMutex, INFINITE);
      printf("%d %d %d %d %d\n",
            a[0],a[1],a[2],a[3],a[4]);
      ReleaseMutex (hMutex);
      }
return 0;
}
```

Також розглянемо *приклад* 4.4, що демонструє спробу звільнення м'ютекса не його володарем. У результаті виконання програми м'ютекс залишиться зайнятим (буде видано повідомлення «MUTEX RELEASE ERROR»), але виконання продовжиться, тому не допускається займати м'ютекс однією ниткою, а звільняти його в іншій, також завжди необхідно перевіряти результат виконання функції **ReleaseMutex**.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
HANDLE hMutex;
HANDLE hThr;

void Thread(void* pParams)
{
      if (ReleaseMutex(hMutex))
            printf("MUTEX FREE\n"); else
            printf("MUTEX RELEASE ERROR\n");
```

```
}

int main(void)
{
      hMutex = CreateMutex(NULL, TRUE, NULL);
      printf("MUTEX OWNED\n");
      hThr = CreateThread(NULL, 0,
            (LPTHREAD_START_ROUTINE)Thread,
            NULL, 0, NULL);
      WaitForSingleObject(hThr, INFINITE);
      printf("THE END\n");
      _getch();
      return 0;
}
```

Події. Об'єкти «події» використовуються для сповіщення ниток, що очікують, про настання якої-небудь події. Розрізняють два види «подій» – з ручним і автоматичним скиданням. Ручне скидання здійснюється функцією **ResetEvent**. «Події» з ручним скиданням використовуються для сповіщення одразу декількох ниток.

При використанні «подій» з автоматичним скиданням сповіщення отримає і продовжить своє виконання тільки одна нитка, що очікує, всі інші будуть продовжувати чекати далі.

Для роботи з «подіями» використовуються функції: **CreateEvent** – створює об'єкт «подію», **SetEvent** – встановлює об'єкт «подію» в сигнальний стан, а **ResetEvent** – скидає об'єкт «подію». Функція **PulseEvent** встановлює об'єкт «подію», а після відновлення ниток, що очікують цей об'єкт (всіх при ручному скиданні і тільки однієї – при автоматичному), скидає його. Якщо немає ниток, що очікують, функція **PulseEvent** просто скидає об'єкт «подію». Для відкриття існуючого об'єкту необхідно скористуватись функцією **CreateEvent** або **OpenEvent**, у параметрі `lpName` яких необхідно задати ім'я існуючого об'єкту «подія». Для закриття об'єкту необхідно скористатися функцією **CloseHandle**.

Параметри вказаних функцій:

```
HANDLE CreateEvent (
 LPSECURITY_ATTRIBUTES lpEventAttributes, //покажчик
           //на структуру атрибутів захисту
 BOOL bManualReset, //тип скидання (ручне/автоматичне)
 BOOL bInitialState, //початковий стан
 LPCTSTR lpName     //ім'я об'єкта «подія»
);

HANDLE OpenEvent (
 DWORD dwDesiredAccess,  // тип доступу
 BOOL bInheritHandle,    // ознака наслідування
 LPCTSTR lpName          // ім'я об'єкта «подія»
);

BOOL PulseEvent (
 HANDLE hEvent   // дескриптор об'єкта «подія»
);

BOOL ResetEvent (
 HANDLE hEvent   // дескриптор об'єкта «подія»
);

BOOL SetEvent (
 HANDLE hEvent   // дескриптор об'єкта «подія»
);
```

Приклад 4.5. Синхронізація ниток за допомогою об'єкта «подія».

```
#include <windows.h>
#include <stdio.h>
HANDLE hEvent1, hEvent2;
```

```
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread (void* pParams)
{
int i, num = 0;
while (1)
      {
      WaitForSingleObject (hEvent2, INFINITE);
      for (i=0; i<5; i++) a[i] = num;
      num++;
      SetEvent (hEvent1);
      }
}

int main (void)
{
hEvent1 = CreateEvent (NULL, FALSE, TRUE, NULL);
hEvent2 = CreateEvent (NULL, FALSE, FALSE, NULL);
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1)
      {
      WaitForSingleObject (hEvent1, INFINITE);
      printf("%d %d %d %d %d\n",
            a[0],a[1],a[2],a[3],a[4]);
      SetEvent (hEvent2);
      }
return 0;
}
```

Семафори. Об'єкт «семафор» – це фактично об'єкт взаємовиключення, але з лічильником. Даний об'єкт дозволяє «захопити» себе визначеній кількості ниток. Після цього чергове «захоплення» буде неможливим, поки одна з ниток, які раніше «захопили» семафор, не «звільнить» його. Семафори використовуються для обмеження кількості ниток, які одночасно використовують ресурс. Об'єкту під час ініціалізації передається максимально необхідне число, яке відповідає кількості ниток, що мають намір його використати. Після кожного «захоплення» лічильник семафора зменшується. Сигнальному стану об'єкта відповідає значення лічильника, яке більше нуля. Коли значення лічильника дорівнює нулю, семафор вважається не встановленим (скинутим).

Функція **CreateSemaphore** створює об'єкт «семафор», в якій вказується максимально можливе початкове значення його лічильника. Функція **OpenSemaphore** – повертає дескриптор існуючого семафора. Захоплення семафора відбувається за допомогою функцій очікування, при цьому значення семафора зменшується на одиницю. Функція **ReleaseSemaphore** звільняє семафор – вона збільшує значення лічильника семафора на вказане в одному з її параметрів число. Цю функцію можна також використовувати для отримання попереднього значення об'єкта «семафор», при цьому отримати поточне значення об'єкта без його зміни – неможливо. Для закриття об'єкта необхідно скористатися функцією **CloseHandle**.

Параметри вказаних функцій:

```
HANDLE CreateSemaphore (
LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,//покажчик
            // на структуру атрибутів захисту
LONG lInitialCount, // початкове значення семафора
LONG lMaximumCount, // максимальне значення семафора
LPCTSTR lpName      // ім'я об'єкта «семафор»
);

HANDLE OpenSemaphore (
 DWORD dwDesiredAccess,  // тип доступу
```

```
 BOOL bInheritHandle,    // ознака наслідування
 LPCTSTR lpName          // ім'я об'єкта «семафор»
);

BOOL ReleaseSemaphore (
 HANDLE hSemaphore,     //дескриптор об'єкта «семафор»
 LONG lReleaseCount,    //число, на яке збільшується
      // значення семафора
 LPLONG lpPreviousCount //покажчик на місце збереження
      // попереднього значення лічильника семафора
);
```

Приклад 4.6. Синхронізація ниток за допомогою семафорів.

```
#include <windows.h>
#include <stdio.h>
HANDLE hSem;
int a[5];
HANDLE hThr;
unsigned long uThrID;

void Thread (void* pParams)
{
int i, num = 0;
while (1) {
      WaitForSingleObject (hSem, INFINITE);
      for (i=0; i<5; i++) a[i] = num;
      num++;
      ReleaseSemaphore (hSem, 1, NULL);
      }
}

int main (void)
```

```
{
hSem = CreateSemaphore (NULL, 1, 1, "MySemaphore1");
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1)
      {
      WaitForSingleObject (hSem, INFINITE);
      printf("%d %d %d %d %d\n",
            a[0],a[1],a[2],a[3],a[4]);
      ReleaseSemaphore (hSem, 1, NULL);
      }
return 0;
}
```

Захищений доступ до змінних. Існує ряд функцій, які дозволяють використовувати глобальні змінні із всіх ниток, не дбаючи про синхронізацію, тому що ці функції самі її відстежують – їх виконання атомарне. Крім цього, їх виконання не потребує переходу в режим ядра ОС, і, відповідно, забезпечується велика швидкість виконання (приблизно 50 тактів процесора). До таких функцій відносяться так звані Interlocked-функції: **InterlockedIncrement** (збільшує значення змінної на одиницю), **InterlockedDecrement** (зменшує значення змінної на одиницю), **InterlockedExchange** (обмінює значення захищеної змінної на інше значення), **InterlockedExchangeAdd** (збільшує значення захищеної змінної на значення свого параметру. Параметр може бути і негативним) і **InterlockedCompareExchange** (виконується порівняння параметрів `Destination` та `Comperand`. Якщо вони рівні між собою, тоді в `Destination` заноситься значення `Exchange`, інакше – нічого не робиться). Всі функції (крім **InterlockedIncrement** та **InterlockedDecrement**) повертають початкове значення захищеної змінної. Функції **InterlockedIncrement** та **InterlockedDecrement** повертають 0, якщо результат дорівнює 0, негативне значення, якщо результат менше 0, і позитивне значення, якщо результат

більше нуля, але усі ці значення не дорівнюють значенню захищеної змінної.

Параметри вказаних функцій:

```
LONG InterlockedCompareExchange (
LPLONG volatile Destination,//адреса захищеної змінної
LONG Exchange,              // значення для зміни
LONG Comperand              //значення для порівняння
);

LONG InterlockedDecrement (
 LPLONG volatile lpAddend  //адреса захищеної змінної
);

LONG InterlockedExchange (
 LPLONG volatile Target, // адреса захищеної змінної
 LONG Value              // нове значення
);

LONG InterlockedExchangeAdd (
 LPLONG volatile Addend,  // адреса захищеної змінної
 LONG Value               // значення для додавання
);

LONG InterlockedIncrement (
 LPLONG volatile lpAddend   //адреса захищеної змінної
);
```

Для роботи з 64-розрядними даними можна скористуватись функціями **InterlockedCompareExchangePointer** та **InterlockedExchangePointer**.

Основний принцип роботи вказаних функцій полягає в наступному:

▪ встановлюється прапорець процесора, який сигналізує про те, що вказана адреса пам’яті зайнята;

▪ зчитується значення з пам'яті в регістр;

▪ змінюється значення в регістрі;

▪ якщо прапорець скинуто, операція повторюється, інакше значення з регістру розміщується в пам'яті.

Приклад 4.7. Синхронізація ниток за допомогою функцій доступу до захищених змінних.

```
#include <windows.h>
#include <stdio.h>
int a[5];
HANDLE hThr;
unsigned long uThrID;
BYTE AccessFlag=0;

void Thread (void* pParams)
{
int i, num = 0;
while (1) {
      while (InterlockedExchange (
            (LONG volatile*)&AccessFlag, 1) == 1)
                  Sleep(0);
      for (i=0; i<5; i++) a[i] = num;
      num++;
      InterlockedExchange((LONG volatile*)&AccessFlag,
            0);
      }
}

int main( void )
{
hThr = CreateThread (NULL,0,
      (LPTHREAD_START_ROUTINE)Thread,NULL,0,&uThrID);
while(1) {
```

```
      while (InterlockedExchange (
            (LONG volatile*)&AccessFlag, 1) == 1)
                  Sleep(0);
      printf("%d %d %d %d %d\n",
            a[0],a[1],a[2],a[3],a[4]);
      InterlockedExchange((LONG volatile*)&AccessFlag,
            0);
      }
return 0;
}
```

4.2. Контрольні питання

1. Опишіть результати виконання прикладу 4.2 на критичних секціях. Чому отримані саме такі результати?

2. Опишіть результати виконання прикладу 4.3 на взаємовиключеннях. Чому отримані саме такі результати?

3. Опишіть результати виконання прикладу 4.5 на подіях. Чому отримані саме такі результати?

4. Опишіть результати виконання прикладу 4.6 на семафорах. Чому отримані саме такі результати?

5. Опишіть результати виконання прикладу 4.7 на захищеному доступу до змінних. Чому отримані саме такі результати?

4.3. Індивідуальні завдання

Необхідно вирішити одну з двох класичних задач на основі вказаних засобів синхронізації.

І. Задача «виробники-споживачі». Має бути вирішена проблема роботи декількох ниток з одним буфером. Частина ниток – це «виробники»: у випадкові моменти часу вони виконують записування інформації в буфер. Друга частина ниток – це «споживачі»: у випадкові моменти часу вони зчитують інформацію з буфера (після зчитування інформація в буфері губиться). Необхідно так організувати виконання ниток, щоб не було колізій при сумісній роботі «виробників» і «споживачів».

ІІ. Задача «читачі-письменники». Є дані, які сумісно використовуються декількома нитками. Є декілька ниток, які тільки читають ці дані («читачі») і декілька ниток, які тільки записують дані або змінюють їх («письменники»). При цьому повинні враховуватись наступні вимоги:

▪ будь-яке число «читачів» може одночасно читати дані;

▪ записувати дані в означений момент часу може тільки один «письменник».

▪ коли «письменник» записує дані, жоден «читач» не може їх у цей час читати.

Індивідуальні завдання наведені в таблиці 4.1.

Таблиця 4.1 – Індивідуальні завдання

Номер варіанту	Номер задачі	Механізм синхронізації	Додаткові умови
1	задача I	семафори	циклічний буфер
2	задача II	семафори	перевага в «письменників»
3	задача I	події	лінійний буфер
4	задача II	події	перевага в «письменників»
5	задача I	Interlocked-функції	буфер – стек
6	задача II	Interlocked-функції	перевага в «письменників»
7	задача I	семафори	лінійний буфер
8	задача II	семафори	перевага в «читачів»
9	задача I	семафори	буфер – стек
10	задача II	події	перевага в «читачів»
11	задача I	події	циклічний буфер
12	задача II	критичні секції	перевага в «письменників»
13	задача I	події	буфер – стек
14	задача II	семафори	перевага в «читачів»
15	задача I	Interlocked-функції	лінійний буфер
16	задача II	критичні секції	перевага в «читачів»
17	задача I	Interlocked-функції	циклічний буфер
18	задача II	м’ютекси	перевага в «читачів»

Перевага «читачів» або «письменників» визначається таким чином:

▪ перевага «читачів» – якщо є дозвіл на читання, «письменники» очікують, коли не буде жодного «читача» (нові «читачі» одразу ж починають читати);

▪ перевага «письменників» – як тільки «письменник» встановить ознаку бажання записувати, жоден новий «читач» не повинен починати читання (але необхідно дочекатися, поки всі «читачі» закінчать операцію читання).

Примітка: нижче наводиться реалізація функцій створення та знищення циклічного буфера з M_BUF елементів. Створити лінійний буфер та буфер у вигляді стеку можна подібним чином. У завданнях розмір буфера повинен бути не менше п’яти елементів.

```
#define M_BUF //максимальний розмір буфера в елементах
HANDLE hHeap;
struct BUF {
      // тут дані елемента буфера
      // ...
      struct BUF *next;
} *Buf, *StartBuf, *CurBuf;

void CreateBuf (void)
{
// створення об’єкта «купа»
hHeap = HeapCreate (HEAP_GENERATE_EXCEPTIONS,
                  0x4000,0);
StartBuf = NULL;
int i = 0;
while (i<M_BUF)
      {
      if (i==0)
            {
            // виділення блоку пам’яті в «купі»
            StartBuf = (struct BUF *)HeapAlloc (hHeap,
```

```
                  HEAP_ZERO_MEMORY, sizeof (BUF));
            CurBuf = StartBuf;
            }
            else
            {
            CurBuf->next=(struct BUF *)HeapAlloc(
                  hHeap, HEAP_ZERO_MEMORY,
                  sizeof(BUF));
            CurBuf = CurBuf->next;
            }
      CurBuf->next = NULL;
      i++ ;
      }
CurBuf->next = StartBuf;
}

void DestroyBuf (void)
{
Buf = StartBuf;
if (StartBuf != NULL)
      {
      while (StartBuf->next != Buf)
            {
            CurBuf = StartBuf->next;
            // звільнення блоку пам'яті в «купі»
            HeapFree (hHeap, 0, (LPVOID)StartBuf);
            StartBuf = CurBuf;
            }
      // знищення об'єкта «купа»
      HeapDestroy (hHeap);
      }
}
```

5. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ ФАЙЛІВ, ЩО ВІДОБРАЖУЮТЬСЯ В ПАМ’ЯТЬ, В ОС WINDOWS

Мета: вивчити можливості застосування механізму обміну даними між процесами в ОС Windows на основі файлів, що відображуються в пам’ять.

5.1. Теоретичні відомості

ОС Windows дозволяє виконувати роботу з файлами без застосування в програмі операцій читання та запису. Механізм, який має назву «проекція файла в пам’ять» (file mapping), полягає в тому, що для файла виділяється адресний простір у пам’яті, куди він копіюється. Всі операції з файлом після відображення замінюються далі на операції з пам’яттю.

Відображення файла виконується таким чином:

▪ створення або відкриття файла за допомогою функції **CreateFile**;

▪ створення об’єкта ядра системи «проекція файла» за допомогою функції **CreateFileMapping**. При цьому також задається розмір файла та режим доступу до нього;

▪ створення в лінійному адресному просторі процесу області для відображення і виконання самого відображення за допомогою функції **MapViewOfFile**. Також на даному етапі задається режим повного або часткового відображення.

Завершення роботи з відображенням файла полягає у виконанні таких операцій:

▪ відмінити відображення за допомогою функції **UnmapViewOfFile**;

▪ закрити об’єкт «проекція файла» за допомогою функції **CloseHandle**;

▪ закрити файл за допомогою функції **CloseHandle**.

Оскільки при відкритті файла та створенні об’єкта «проекція файла» в системі збільшуються значення лічильників кількості відкриттів, то у випадку виконання відображення тільки один раз слід змінити порядок роботи з відображенням на наступний:

▪ створення або відкриття файла за допомогою функції **CreateFile**;

▪ створення об’єкта ядра системи «проекція файла» за допомогою функції **CreateFileMapping**;

▪ закриття файла за допомогою функції **CloseHandle**.

▪ створення в лінійному адресному просторі процесу області для відображення і виконання самого відображення за допомогою функції **MapViewOfFile**;

▪ закриття об’єкту «проекція файла» за допомогою функції **CloseHandle**;

▪ робота з відображенням файла;

▪ відміна відображення за допомогою функції **UnmapViewOfFile**.

Закриття об’єктів одразу після того, як вони стають непотрібні, дозволяє значно зменшити можливі проблеми з обліком ресурсів у системі у випадку аварійного закриття програми.

Основним недоліком при використанні відображення файлів у пам’ять є неможливість збільшення розміру файла.

Відображення файлів у пам’ять використовується не тільки для прискорення роботи з файлами. Спосіб обміну даними між процесами за допомогою файлів, що відображуються в пам’ять, має високу швидкодію, тому що дані передаються між процесами безпосередньо через віртуальну пам’ять. Відображення файлів у пам’ять лежить в основі більшості механізмів передачі даних між процесами – IPC (Interprocess Communication). Тому, якщо необхідно забезпечити максимальну швидкодію, краще користуватися саме цим механізмом.

Методика роботи з відображенням файла для передачі даних між процесами така: об’єкт «проекція файла» створюється функцією **CreateFileMapping**, В якості першого параметра функції треба передати ідентифікатор відкритого файла. Якщо об’єкт «проекція файла» буде використовуватись для передачі даних між процесами, зручно, щоб він був з ім’ям. Використовуючи це ім’я, інші процеси зможуть відкрити об’єкт «проекція файла» функцією **OpenFileMapping**.

Якщо створюється об’єкт «проекція файла» тільки для того, щоб забезпечити передачу даних між процесами, не треба створювати файл у зовнішній пам’яті комп’ютера. Треба вказати замість ідентифікатора файла

значення (HANDLE)0xFFFFFFFF, і відображення буде створене безпосередньо у віртуальній пам'яті без використання додаткового файла.

Після створення об'єкта «проекція файла», необхідно виконати відображення файла в пам'ять за допомогою функції **MapViewOfFile**. При успішному виконанні ця функція поверне покажчик на відображену область пам'яті.

Перед завершенням своєї роботи процеси повинні відмінити відображення файла за допомогою функції **UnmapViewOfFile** і звільнити ідентифікатор створеного об'єкта «проекція файла» за допомогою функції **CloseHandle**.

Параметри вказаних функцій:

```
HANDLE CreateFileMapping(
 HANDLE hFile,        // дескриптор файла
 LPSECURITY_ATTRIBUTES lpAttributes, ,//покажчик
      // на структуру атрибутів захисту
 DWORD flProtect,   // захист сторінки (читання/запис)
 DWORD dwMaximumSizeHigh, // старші 32 біти розміру
      // зображення (для файлів менше 4 Гбайтів = 0)
 DWORD dwMaximumSizeLow,  // молодші 32 біти розміру
 LPCTSTR lpName  // ім'я об'єкта «проекція файла»
);

LPVOID MapViewOfFile(
 HANDLE hFileMappingObject, // дескриптор об'єкта
 DWORD dwDesiredAccess,     // режим доступу
 DWORD dwFileOffsetHigh,   // старші 32 біти, з яких
      // починати відображення
 DWORD dwFileOffsetLow,    // молодші 32 біти, з яких
      // починати відображення
 SIZE_T dwNumberOfBytesToMap  // число байтів для
      // відображення (0 – весь файл)
);
```

```
HANDLE OpenFileMapping(
 DWORD dwDesiredAccess, // режим доступу
 BOOL bInheritHandle, // ознака наслідування
 LPCTSTR lpName       // ім'я об'єкта «проекція файла»
);

BOOL UnmapViewOfFile(
LPCVOID lpBaseAddress   // адреса початку відображення
);
```

Приклад 5.1. Програма–сервер створює відображення файла в пам'ять і записує в нього символьні рядки, що вводяться з клавіатури. Програма–клієнт – зчитує ці рядки із відображуваного файла. Для синхронізації використовуються об'єкти «події». Кількість серверних процесів обмежена одним екземпляром, кількість клієнтських процесів необмежена. Обмеження реалізовано за допомогою перевірки наявності створених у системі подій `MyEvent1` та `MyEvent2`: якщо події вже створено, то вважається, що сервер вже працює. Для завершення роботи сервера та всіх клієнтів необхідно ввести рядок «*=exit*» (для завершення роботи клієнтів цей рядок сервером записується також у спільну пам'ять і перевіряється клієнтами).

Текст програми–сервера.

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

HANDLE hEvent1, hEvent2;
HANDLE hMFile;
char Ev1[] = "MyEvent1";
char Ev2[] = "MyEvent2";
char Fn1[] = "MyMemoryMappedFile";
LPVOID StartMFile;
char Buf[100];
```

```
int main(int argc, char* argv[])
{
	// спроба відкрити подію – ознака сервера
	hEvent1 = OpenEvent(EVENT_ALL_ACCESS,TRUE,Ev1);
	if (!hEvent1) {
		hEvent1 =CreateEvent(NULL,TRUE,FALSE,Ev1);
		hEvent2 =CreateEvent(NULL,TRUE,TRUE,Ev2);
		if ((!hEvent1) || (!hEvent2)) {
			// не вдалося створити події
			printf("ERROR Events Creation\n");
			return 0;
		}
	}
	else {		// сервер вже запущено
		CloseHandle(hEvent1);
		return 0;
	}
	hMFile =CreateFileMapping((HANDLE)0xFFFFFFFF,
		NULL, PAGE_READWRITE, 0, 100, Fn1);
	if (!hMFile) {
		printf("ERROR File Mapping Creation\n");
		return 0;
	}
	StartMFile = MapViewOfFile(hMFile,
		FILE_MAP_WRITE, 0, 0, 100);
	if (!StartMFile) {
		printf("ERROR MAp View of File\n");
		return 0;
	}
	printf("SERVER Started\n");
	do {
		WaitForSingleObject(hEvent2, INFINITE);
```

```
            scanf_s("%99s", Buf,
                  (unsigned)_countof(Buf));
            CopyMemory(StartMFile, Buf, sizeof(Buf));
            PulseEvent(hEvent1);
      } while (CompareString(LOCALE_SYSTEM_DEFAULT,
            NORM_IGNORECASE, Buf, -1, "=exit", -1) !=
            CSTR_EQUAL);
      UnmapViewOfFile(StartMFile);
      CloseHandle(hMFile);
      return 0;
}
```

Текст програми–клієнта.

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

HANDLE hEvent1, hEvent2;
HANDLE hMFile;
char Ev1[] = "MyEvent1";
char Ev2[] = "MyEvent2";
char Fn1[] = "MyMemoryMappedFile";
LPVOID StartMFile;
char Buf[100];

int main(int argc, char* argv[])
{
hEvent1 = OpenEvent (EVENT_ALL_ACCESS, TRUE, Ev1);
hEvent2 = OpenEvent (EVENT_ALL_ACCESS, TRUE, Ev2);
if((!hEvent1)||(!hEvent2)) {
      printf("ERROR Events Opening\n");
      return 0;}
hMFile = OpenFileMapping (FILE_MAP_READ, TRUE, Fn1);
if(!hMFile) { printf("ERROR File Mapping Opening\n");
```

```
      return 0;}
StartMFile = MapViewOfFile (hMFile, FILE_MAP_READ,
      0, 0, 100);
// закриваємо дескриптор об'єкта – більше не потрібен
CloseHandle (hMFile);
if(!StartMFile) { printf("ERROR Map View of File\n");
      return 0;}
printf("CLIENT Started\n");
do {
      WaitForSingleObject (hEvent1, INFINITE);
      CopyMemory (Buf, StartMFile, 100);
      printf("%s\n",Buf);
      PulseEvent (hEvent2);
      } while (CompareString(LOCALE_SYSTEM_DEFAULT,
      NORM_IGNORECASE,Buf,-1,"=exit",-1)!=CSTR_EQUAL);
UnmapViewOfFile (StartMFile);
return 0;
}
```

5.2. Контрольні питання

1. Що відбудеться, якщо першою запустити програму-«клієнт» у прикладі 5.1? Чому?

2. Які операції неможливо зробити з «проекцією файла» в пам'ять?

3. Назвіть переваги використання «проекції файла» в пам'ять по зрівнянню зі звичайними файловими операціями.

4. Яким чином зробити проекцію в пам'ять для неіснуючого файла?

5. Яким чином забезпечити сумісний доступ до «проекції файла» в пам'ять?

5.3. Індивідуальні завдання

Необхідно вирішити одну з двох класичних задач на основі вказаних засобів синхронізації.

І. Задача «виробники-споживачі». Має бути вирішена проблема роботи декількох процесів з одним буфером. Частина процесів – це «виробники»: у випадкові моменти часу вони виконують записування інформації в буфер. Друга частина процесів – це «споживачі»: у випадкові моменти часу вони зчитують інформацію із буфера (після зчитування інформація в буфері губиться). Необхідно так організувати виконання процесів, щоб не було колізій при сумісній роботі «виробників» і «споживачів».

ІІ. Задача «читачі-письменники». Є дані, які сумісно використовуються декількома процесами. Є декілька процесів, які тільки читають ці дані («читачі») і декілька інших процесів, які тільки записують дані або змінюють їх («письменники»). При цьому повинні враховуватись наступні вимоги:

▪ будь-яке число «читачів» може одночасно читати дані;

▪ записувати дані в означений момент часу може тільки один «письменник».

▪ коли «письменник» записує дані, жодний читач не може їх у цей час читати.

Індивідуальні завдання наведені в таблиці 5.1. Буфер для передачі даних між процесами повинен бути організований у вигляді відображення файла в пам’ять, поділений на кілька порцій. Кількість порцій інформації, що можна помістити в буфері, повинна бути в межах від 5 до 10. Рекомендується для допоміжних даних використовувати одну з порцій у відображенні файла в пам’ять. Одна операція запису або читання виконується тільки над однією порцією інформації. Перевага «читачів» або «письменників» визначається таким чином:

▪ перевага «читачів» – якщо є дозвіл на читання, «письменники» очікують, коли не буде жодного «читача» (нові «читачі» одразу ж починають читати);

▪ перевага «письменників» – як тільки «письменник» встановить ознаку бажання записувати, жоден новий «читач» не повинен починати читання (але необхідно дочекатися, поки всі «читачі» закінчать операцію читання).

Передбачити завершення всіх процесів (серверних та клієнтських) за обраним ключовим словом, що записується в буфер.

Таблиця 5.1 – Індивідуальні завдання

Номер варіанту	Номер задачі	Механізм синхронізації	Додаткові умови	
			Кількість	Перевага
1	II	семафори	2 «письменника», 2 «читача»	«письменники»
2	I	семафори	1 «виробник», 2 «споживача»	
3	II	події	2 «письменника», 2 «читача»	«письменники»
4	I	події	1 «виробник», 2 «споживача»	
5	II	м’ютекси	2 «письменника», 2 «читача»	«письменники»
6	I	семафори	2 «виробника», 1 «споживач»	
7	II	семафори	1 «письменник», 3 «читача»	«читачі»
8	I	події	2 «виробника», 1 «споживач»	
9	II	події	1 «письменник», 3 «читача»	«читачі»
10	I	семафори	1 «виробник», 2 «споживача»	
11	II	м’ютекси	2 «письменника», 1 «читач»	«читачі»
12	I	події	2 «виробника», 1 «споживач»	
13	II	семафори	3 «письменника», 1 «читач»	«письменники»
14	I	семафори	2 «виробника», 2 «споживач»	
15	II	події	1 «письменник», 3 «читача»	«письменники»
16	I	семафори	3 «виробника», 3 «споживач»	
17	II	м’ютекси	3 «письменника», 1 «читач»	«письменники»
18	I	події	2 «виробника», 2 «споживач»	
19	II	семафори	3 «письменника», 1 «читач»	«читачі»

6. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ МЕХАНІЗМУ ПОВІДОМЛЕНЬ В ОС WINDOWS

Мета: вивчити можливості по використанню механізму обміну повідомленнями для передачі даних між процесами в ОС Microsoft Windows.

6.1. Теоретичні відомості

Механізм обміну на основі передачі повідомлень заснований на виділенні ділянки пам'яті одного процесу, який виконує передачу, іншому, що виконує прийом. Цей механізм вважається найпростішим з засобів IPC в ОС Microsoft Windows.

Для передачі даних між процесами можна використовувати декілька типів повідомлень, серед яких розглянемо WM_SETTEXT, WM_GETTEXT та WM_COPYDATA. Перші два призначені для маніпуляцій з заголовками вікон програм, а третій – для передачі порцій даних або власних повідомлень, які не відомі в системі. Для роботи з усіма цими повідомленнями необхідно використовувати тільки функцію **SendMessage** (використання другої функції для передачі – **PostMessage** не допускається). Це пов'язано з тим, що за допомогою функції **PostMessage** повідомлення ставиться в чергу повідомлень для нитки, що створила вікно, і одразу виконується повернення без очікування відповіді, а функція **SendMessage** після відправлення повідомлення очікує завершення обробки переданого повідомлення і тільки після цього виконується повернення. Необхідність очікування завершення обробки повідомлення описана нижче.

Повідомлення WM_SETTEXT.

Синтаксис функції **SendMessage** для встановлення нового заголовку вікна:

```
SendMessage(
 (HWND) hWnd, // дескриптор вікна, якому
              // передається повідомлення
 WM_SETTEXT,  // тип – встановити
```

```
                // заголовок вікна
  (WPARAM) wParam, // не використовується і
                   // повинен бути рівним 0
  (LPARAM) lParam  // адреса рядка з
                   // новим заголовком
);
```

Тип параметру `lParam` повинен бути LPCTSTR і містить адресу рядка в адресному просторі процесу, який викликає цю функцію.

Механізм обробки повідомлення WM_SETTEXT дозволяє процесу–приймачу повідомлення отримати доступ до рядка за адресою, що міститься в параметрі `lParam` без проблем з доступом до адресного простору процесу, що викликав функцію **SendMessage**. На відміну від інших типів повідомлень при передачі даних між процесами за допомогою повідомлення WM_SETTEXT створюється файл, що відображується в пам'ять, в який копіюється рядок з адресного простору передавача і робиться відображення цього файла доступним для приймача. Після цих дій виконується безпосередня передача повідомлення приймачу. При обробці повідомлення WM_SETTEXT приймач визначає адресу відображення файла, що містить копію рядка, та в параметр `lParam` заноситься саме ця адреса (тобто адреса в параметрі `lParam` для передавача і приймача буде різною). Одразу ж після обробки повідомлення відображення файла знищується.

Повідомлення WM_GETTEXT.

Синтаксис функції **SendMessage** для зчитування заголовка вікна:

```
SendMessage(
  (HWND) hWnd, // дескриптор вікна, якому
               // передається повідомлення
 WM_GETTEXT,  // тип – прочитати заголовок вікна
  (WPARAM) wParam, // число символів для копіювання
  (LPARAM) lParam  // адреса буфера для прийому даних
);
```

При зчитуванні заголовка вікна іншої програми операційною системою виконуються такі дії: одразу після виклику функції **SendMessage** з параметром WM_GETTEXT система спочатку надсилає повідомлення WM_GETTEXTLENGTH, у результаті обробки якого отримується кількість символів у заголовку вікна і отримане число використовується в якості розміру для створення відображення файла. Далі для занесення даних у файл, що відображений в пам'ять, посилається повідомлення WM_GETTEXT. Потім управління повертається процесу, що передав повідомлення WM_GETTEXT, і виконується копіювання даних з файла, що відображено в пам'ять, у буфер, адреса якого задана в параметрі `lParam`. Наприкінці виконується повернення з функції **SendMessage**.

Повідомлення WM_COPYDATA.

Для передачі повідомлення WM_COPYDATA необхідно заповнити структуру COPYDATASTRUCT, адреса якої передається через параметр `lParam` функції **SendMessage**:

```
SendMessage(
 (HWND) hWnd, // дескриптор вікна,
              // якому передається повідомлення
 WM_COPYDATA, // тип – передати дані
 (WPARAM) wParam, // дескриптор вікна,
                  // яке передає повідомлення
 (LPARAM) lParam  // покажчик на
                  // структуру COPYDATASTRUCT
);
```

Структура COPYDATASTRUCT описана в заголовному файлі `Winuser.h`:

```
typedef struct tagCOPYDATASTRUCT {
 ULONG_PTR dwData; // будь яке 32-бітне значення
```

```
 DWORD cbData; // обсяг даних для передачі в байтах
 PVOID lpData; // покажчик на перший байт даних
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

У полі `dwData` можна записувати будь-яке значення, яке можна використовувати за своїм розсудом (наприклад, вказувати тип даних, що передаються). Покажчик `lpData` вказує на адресу, що знаходиться в адресному просторі процеса, що викликає функцію **SendMessage** з параметром WM_COPYDATA.

При виклику функції **SendMessage** створюється відображення файла розміром `cbData` байтів і дані копіюються в нього з адреси, на яку посилається поле `lpData`. Після цих дій надсилається повідомлення процесу-приймачу. У приймачі при прийомі повідомлення WM_COPYDATA параметр `lParam` вказує на адресу структури COPYDATASTRUCT, яка знаходиться в адресному просторі приймача. Поле `lpData` цієї структури вказує на початок відображення файла в адресному просторі приймача. Так само як і при обробці повідомлення WM_SETTEXT одразу ж після завершення обробки відображення файла знищується.

Приклад 6.1. Використання повідомлень для передачі даних. Програма-передавач реалізована без використання графічного інтерфейсу. У ній виконуються наступні дії: пошук вікна програми-приймача, зчитування та виведення на екран заголовка вікна приймача, встановлення нового заголовка у вікні приймача, далі в циклі – передача рядків, введених користувачем з клавіатури. Якщо було введено слово «*quit*» – робота програми завершується (при цьому введене слово передається приймачу і приймач також завершує свою роботу).

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
void main()
{
char szBuf1[100], szBuf2[256];
```

```
COPYDATASTRUCT data;
HWND hwnd = FindWindow("MyWindowApp",
      "MessageReceiver");
if (hwnd) {
      printf("Receiver Found\n");
      SendMessage(hwnd, WM_GETTEXT,
            sizeof(szBuf2), (LPARAM)szBuf2);
      printf("Found Window With Caption: %s\n",
            szBuf2);
      SendMessage(hwnd, WM_SETTEXT, 0,
            (LPARAM)"Ready To Message Receive!");
      printf("Caption Changed\n");
      printf("Enter <quit> to end program\n");
      do {
            scanf_s("%99s", szBuf1,
                  (unsigned)_countof(szBuf1));
            data.cbData = strlen(szBuf1);
            data.lpData = szBuf1;
            SendMessage(hwnd, WM_COPYDATA,
                  (WPARAM)GetFocus(),
                  (LPARAM)&data);
      } while (strcmp(szBuf1, "quit"));
}
}
```

Програма-приймач реалізована з використанням графічного інтерфейсу. Це викликано необхідністю обробляти повідомлення WM_COPYDATA. У разі отримання повідомлення приймач виводить його по центру свого вікна. Для коректної роботи першим необхідно запускати на виконання програму-приймач, другою – програму-передавач. Як можна побачити, більша частина коду призначена лише для організації функціонування процесу з графічним віконним інтерфейсом. Частина, що відповідає за роботу з повідомленнями (прийом та відображення) – відокремлена коментарями.

```
#include <windows.h>
char              Buf[100];
PCOPYDATASTRUCT   pMyCDS;
RECT              rect;
static char szAppName[] = "MyWindowApp";
static char szWinCaption[] = "MessageReceiver";
LRESULT CALLBACK WndProc (HWND, UINT, WPARAM, LPARAM);

int WINAPI WinMain (HINSTANCE hInstance,
      HINSTANCE hPrevInstance, PSTR szCmdLine,
      int iCmdShow)
{
HWND        hwnd;
MSG         msg;
WNDCLASSEX  wndclass;
wndclass.cbSize = sizeof (wndclass);
wndclass.style = CS_HREDRAW | CS_VREDRAW;
wndclass.lpfnWndProc = WndProc;
wndclass.cbClsExtra = 0;
wndclass.cbWndExtra = 0;
wndclass.hInstance = hInstance;
wndclass.hIcon = LoadIcon (NULL, IDI_APPLICATION);
wndclass.hCursor = LoadCursor (NULL, IDC_ARROW);
wndclass.hbrBackground=
      (HBRUSH) GetStockObject(COLOR_BACKGROUND);
wndclass.lpszMenuName = NULL;
wndclass.lpszClassName = szAppName;
wndclass.hIconSm = LoadIcon (NULL, IDI_APPLICATION);
RegisterClassEx (&wndclass);
hwnd = CreateWindow
            (szAppName, szWinCaption,
            WS_OVERLAPPEDWINDOW,
```

```
            GetSystemMetrics (SM_CXFULLSCREEN) /2-170,
            GetSystemMetrics (SM_CYFULLSCREEN) /2-100,
            340, 200, NULL, NULL, hInstance, NULL);
ShowWindow (hwnd, iCmdShow);
UpdateWindow (hwnd);
while (GetMessage (&msg, NULL, 0, 0))
      {
      TranslateMessage (&msg);
      DispatchMessage (&msg);
      }
return msg.wParam;
}

LRESULT CALLBACK WndProc (HWND hwnd, UINT iMsg,
      WPARAM wParam, LPARAM lParam)
{
HDC         hdc;
PAINTSTRUCT ps;
switch (iMsg)
      {
      case WM_CREATE :
            return 0;
            break;
// тут починається реалізація роботи з повідомленнями
      case WM_COPYDATA : //прийом повідомлення
            pMyCDS = (PCOPYDATASTRUCT) lParam;
            strcpy_s(Buf,(char *)pMyCDS->lpData);
            Buf[pMyCDS->cbData] = 0;
// тут можна використовувати реалізацію порівняння
// через CompareString з прикладу в попередній темі
            if (!strcmp(Buf,"quit"))
                  SendMessage (hwnd,WM_DESTROY,0,0);
            InvalidateRect (hwnd, NULL, 1);
```

```
            break;
      case WM_PAINT :
            hdc = BeginPaint (hwnd, &ps);
            GetClientRect (hwnd, &rect);
            DrawText (hdc, Buf, -1, &rect,
                  DT_SINGLELINE|DT_CENTER|DT_VCENTER);
            EndPaint (hwnd, &ps);
            return 0;
            break;
// тут закінчується реалізація роботи з повідомленнями
      case WM_DESTROY :
            PostQuitMessage (0);
            return 0;
      }
return DefWindowProc (hwnd, iMsg, wParam, lParam);
}
```

Зауваження щодо використання передачі повідомлень.

Як було сказано вище, використання функції **PostMessage** не допускається. Основна причина – виконання цієї функції асинхронне по відношенню до приймача повідомлення, і при обробці вказаних типів повідомлень можлива наступна ситуація: відображення файла буде створено, але не буде знищено після обробки повідомлення. Це пов'язано з тим, що операційна система не може визначити час, коли необхідно знищувати відображення і, відповідно, звільнити пам'ять.

При обробці повідомлення WM_COPYDATA виконується копіювання даних у відображуваний файл, тобто витрачається деякий час, тому не допускається зміна даних у буфері, який передається, до завершення роботи функції **SendMessage**. Якщо інші нитки процесу-передавача під час виконання функції **SendMessage** виконають зміну даних у буфері – можлива часткова або повна зміна даних, і процес-приймач отримає пошкоджену інформацію.

У даних, що передаються за допомогою механізму повідомлень, не слід розміщувати абсолютні адреси, покажчики або інші посилання на об'єкти, навіть коли вони також знаходяться в тому ж самому буфері і будуть також передані процесу-приймачу. В адресному просторі процеса-приймача такі дані будуть мати інші адреси (навіть якщо передаються в тому ж самому буфері) або будуть недоступні (якщо їх передача не виконується через буфер). Так, якщо передається список з декількох структур, і всі елементи списку розміщено в одному буфері і, відповідно, передача виконується одним викликом функції **SendMessage,** то в процесі-приймачі покажчики на наступні елементи списку більше не будуть показувати на відповідні структури.

Дані, які передаються за допомогою повідомлення WM_COPYDATA, доступні процесу-приймачу тільки для читання. Звільнювати пам'ять, в якому знаходяться дані, і на яку посилається поле `lpData` структури COPYDATASTRUCT, не допускається. Слід зауважити, що відображуваний файл, в якому знаходяться дані, буде знищено після обробки повідомлення, і в разі необхідності отримання доступу к переданим даним, їх необхідно скопіювати в локальний буфер процеса-приймача.

Використання функції **SendMessage** може привести до зупинки виконання процесу. Це пов'язано з тим, що вказана функція виконується синхронно по відношенню к процесу, якому посилається повідомлення. Повернення із функції буде виконане тільки тоді, коли повідомлення буде оброблене. Якщо приймач не відповідає, очікування може бути нескінченним. Для того, щоб мати можливість виходу з функції передачі повідомлень, необхідно використовувати функцію **SendMessageTimeout**, за допомогою якої задається інтервал часу, на протязі якого очікується відповідь на повідомлення. Функція описується таким чином:

```
LRESULT SendMessageTimeout(
 HWND hWnd,             // перші 4 параметри ідентичні
 UINT Msg,              // функції
 WPARAM wParam,         // SendMessage
 LPARAM lParam,         //
```

```
 UINT fuFlags,          // прапорці передачі
 UINT uTimeout,         // час очікування відповіді
 PDWORD_PTR lpdwResult // результат обробки
);
```

Прапорці `fuFlags` дозволяють встановити наступні режими передачі повідомлення: SMTO_ABORTIFHUNG (в тому разі, якщо процес-приймач не відповідає – одразу повернути управління. В ОС процес вважається таким, що не відповідає, якщо більше п’яти секунд він не оброблює повідомлення), SMTO_BLOCK (не дає виконувати обробку синхронних повідомлень процесу-передавачу. При використанні цього прапорця можлива ситуація взаємного блокування процесів), SMTO_NORMAL (задається, коли не задані інші прапорці. Його значення дорівнює 0), SMTO_NOTIMEOUTIFNOTHUNG (ігнорування часу очікування відповіді, якщо процес-приймач оброблює повідомлення).

У наведеній програмі процеса-передавача можна замінити виклик функції **SendMessage** на **SendMessageTimeout** таким чином:

```
DWORD dwRes;
SendMessageTimeout(hwnd, WM_COPYDATA,
      (WPARAM)GetFocus(), (LPARAM)&data,
      SMTO_ABORTIFHUNG, 1000, &dwRes);
```

6.2. Контрольні питання

1. Що відбудеться, якщо в прикладі 6.1 використання повідомлень для передачі даних першою запустити програму-передавач? Чому?

2. Чим відрізняється передача повідомлень через **SendMessage** та **PostMessage**?

3. Яким чином прийняти повідомлення WM_COPYDATA?

4. Яким чином ще можна перевірити відповідність знайденої програми за ім’ям?

5. Чи можна надсилати повідомлення передачі даних та зміни заголовка вікна самому собі?

6.3. Індивідуальні завдання

1) Розробити програму, що розсилає повідомлення всім своїм запущеним на виконання копіям.

2) Розробити програму, що змінює заголовки всім вікнам запущених на виконання програм, при завершенні роботи програми вернути заголовки на місто.

3) Розробити програму, що виконує передачу створеного в програмі списку до іншої, також розробленої, програми.

4) Розробити програми, одна з яких отримує рядки символів від інших програм, переводить їх до верхнього регістру та відправляє назад.

5) Розробити програми, такі, що програма-приймач приймає дані від програм-передавачів та виводить на екран ці дані та дескриптори вікон програм-передавачів.

6) Розробити дві програми, одна з яких отримує перелік всіх заголовків вікон у системі та передає його другій програмі.

7) Розробити дві програми, одна з яких виводить коди символів, що вводяться в другій програмі.

8) Розробити дві програми, одна з яких посилає повідомлення другій, а друга змінює в першій програмі заголовок вікна на прийняте повідомлення з переставленими символами в зворотному порядку.

9) Розробити дві програми, одна з яких отримує список заголовків вікон всіх інших програм у системі та з інтервалом 10 секунд змінює заголовок на кожний з отриманих другій програмі та передає їй текстове повідомлення про зміну.

10) Розробити програму, яка виконує керування другою, також розробленою, програмою (імітація натискань на її кнопки). Для виконання завдання зміст повідомлення трактувати як повідомлення користувача, по яким пізнаються натискання.

*Примітка***:** для виконання індивідуального завдання необхідно визначити тип програм (консольні або віконні) відповідно до поставленого завдання. Розроблені програми не повинні впливати на працездатність ОС,

після їх завершення можливий вплив на інші програми повинен бути скасованим.

Для деяких програм може знадобитися перегляд вікон всіх процесів у системі. Для реалізації перегляду можна скористуватися наступним фрагментом програми.

```
BOOL CALLBACK EnumWndProc( HWND hWnd, LPARAM lParam )
{
char szWindowName[71];
char szClassName[31];
// отримуємо заголовок і клас вікна
GetWindowText (hWnd, szWindowName, 70);
GetClassName (hWnd, szClassName, 30);
// перевіряємо отримані заголовок та клас вікна з
// szWinCaption та szAppName
if((strcmp (szWindowName,szWinCaption)==0) &&
      (strcmp (szClassName,szAppName)==0))
      { //в разі знайденого вікна - тут необхідні дії
      }
return (TRUE);
}
```

Реентерабельна функція (зворотного виклику) **EnumWndProc** викликається один раз за допомогою функції WinAPI **EnumWindows**:

```
EnumWindows ((WNDENUMPROC)EnumWndProc, NULL);
```

Ця функція виконується доти, доки не будуть передані у функцію **EnumWndProc** всі вікна, що знаходяться на екрані або доки функція зворотного виклику **EnumWndProc** не поверне значення FALSE.

7. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ АНОНІМНИХ КАНАЛІВ В ОС WINDOWS

Мета: вивчити можливості використання механізму обміну даними між процесами в ОС Windows на основі анонімних каналів.

7.1. Теоретичні відомості

Канал зв'язку (pipe) – це засіб, за допомогою якого можна організувати обмін даними між процесами. Принцип дії каналу оснований на механізмі введення–виведення: процес, що передає дані, діє так, мовби записує дані у файл, а процес, що приймає – так, ніби читає дані з файла. Насправді, канали є буферною пам'яттю, що організована за принципом FIFO.

Через канал можна передавати дані тільки між двома процесами. Один з цих процесів створює канал, другий відкриває його. Після цього обидва процеси можуть передавати дані через канал в одну або в обидві сторони.

Існують два різновиди каналів – іменовані (Named Pipes) та анонімні (Anonymous Pipes).

Іменованому каналу при створенні привласнюється ім'я, яке є доступним для інших процесів. Знаючи ім'я якої-небудь робочої станції в мережі, процес може отримати доступ до каналу, створеному на цій робочій станції.

Анонімні канали зазвичай використовуються для організації передачі даних між батьківськими та дочірніми процесами, запущеними на одній і тій самій робочій станції або на «окремо розміщеному» комп'ютері.

Анонімні канали дозволяють виконувати передачу даних між процесами тільки в межах локальної робочої станції в одному напрямку (напівдуплексний режим). Кожний канал має два дескриптори – один на читання і один на запис. У тому випадку, якщо необхідно організувати двоспрямовану передачу даних, необхідно створити два канали. Анонімні канали неможливо створити між двома будь-якими процесами – вони створюються тільки для обміну між дочірніми процесами. При цьому значення дескриптора передається або через рядок виклику процесу або через змінну середовища виконання програми. Анонімні канали часто використовуються для

заміни стандартного введення або виведення (цей механізм використовується операційними системами UNIX та Windows для організації конвеєрів команд, наприклад, конвеєри команд « **ls | sort | more »** (для UNIX та Linux) та « **dir | sort | more »** (для Windows) дозволяють отримати перелік файлів у каталозі, відсортувати його та вивести на екран посторінково, при цьому « | » створює анонімний канал для організації конвеєра).

Створення анонімного канала виконується за викликом функції **CreatePipe**:

```
BOOL CreatePipe(
PHANDLE hReadPipe,      // покажчик на дескриптор
                        // сторони читання канала
PHANDLE hWritePipe,     // покажчик на дескриптор
                        // сторони запису канала
LPSECURITY_ATTRIBUTES lpPipeAttributes, // покажчик на
                        // структуру атрибутів безпеки
DWORD nSize       // бажаний розмір канала
);
```

Структуру атрибутів безпеки для організації можливостей наслідування необхідно ініціювати таким чином:

```
SECURITY_ATTRIBUTES saAttr;
saAttr.nLength = sizeof(SECURITY_ATTRIBUTES);
saAttr.bInheritHandle = TRUE;
saAttr.lpSecurityDescriptor = NULL;
```

або еквівалентний коротший запис

```
SECURITY_ATTRIBUTES saAttr =
      {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
```

Розмір каналу, що задається параметром `nSize` в байтах, не обов'язково буде саме таким, як вказано – на основі заданого параметра в системі обчислюється розмір буфера для обміну. Якщо параметр дорівнює 0 – використовується розмір буфера за замовчанням.

У разі успішного створення каналу дескриптори обох його сторін, покажчики на які (`hReadPipe` та `hWritePipe`) повертає функція створення, передаються процесам, що виконуватимуть обмін даними. Ці дескриптори використовуються в першому параметрі (дескриптор файла) функцій **ReadFile** та **WriteFile** для читання та запису даних у канал відповідно.

Приклад 7.1 Наведені тексти двох програм, перша програма (**APipeF**) виконує такі дії: отримує дескриптори стандартних потоків введення та виведення та в нескінченному циклі за допомогою функції **ReadFile** зчитує рядки символів зі стандартного введення, переставляє символи в зчитаних рядках у зворотному порядку, та модифіковані рядки виводить за допомогою функції **WriteFile** на стандартний пристрій виведення. Для завершення роботи програми необхідно натиснути комбінацію клавіш <Control+Z>.

```
#include <windows.h>

char szBuf[256];
DWORD dwRead, dwWritten;
HANDLE hStdin, hStdout;
BOOL fSuccess;

int main()
{
hStdout = GetStdHandle (STD_OUTPUT_HANDLE);
hStdin = GetStdHandle (STD_INPUT_HANDLE);
if ((hStdout == INVALID_HANDLE_VALUE) ||
            (hStdin==INVALID_HANDLE_VALUE))
             ExitProcess(1);
for (;;) {
```

```
      fSuccess=ReadFile(hStdin,szBuf,256,
            &dwRead,NULL);
      if (!fSuccess || dwRead == 0) break;
      for(int i=0, j=dwRead-2;i<j;i++,j--) {
            char c = szBuf[i]; szBuf[i] = szBuf[j];
            szBuf[j] = c;}
      fSuccess = WriteFile (hStdout,szBuf,dwRead,
            &dwWritten,NULL);
      if (!fSuccess) break;
      }
ExitProcess (0);
return 0;
}
```

У другій програмі (**APipeC**) виконуються подібні операції за винятком перетворення рядка символів: тут всі символи переводяться до верхнього регістру за допомогою функції **CharUpperBuff**.

```
#include <windows.h>

char szBuf[256];
DWORD dwRead, dwWritten;
HANDLE hStdin, hStdout;
BOOL fSuccess;

int main()
{
hStdout = GetStdHandle (STD_OUTPUT_HANDLE);
hStdin = GetStdHandle (STD_INPUT_HANDLE);
if ((hStdout == INVALID_HANDLE_VALUE) ||
            (hStdin==INVALID_HANDLE_VALUE))
             ExitProcess(1);
for (;;) {
```

```
      fSuccess=ReadFile(hStdin,szBuf,256,
            &dwRead,NULL);
      if (!fSuccess || dwRead == 0) break;
      CharUpperBuff (szBuf, strlen (szBuf));
      fSuccess = WriteFile (hStdout,szBuf,dwRead,
            &dwWritten,NULL);
      if (!fSuccess) break;
      }
ExitProcess (0);
return 0;
}
```

Для організації конвеєру команд у консолі операційної системи необхідно запустити програму **cmd** та в ній ввести такий рядок:

```
APipeF.exe | APipeC.exe
```

У такому разі потік стандартного виведення першої програми пов'язується за допомогою анонімного канала з потоком стандартного введення другої програми (« | » – створює анонімний канал для організації конвеєру). Введені рядки символів у першій програмі будуть переставлятися в зворотному порядку, передаватись другій програмі, і там переводитись на верхнього регістру та виводитись на екран.

Приклад 7.2. У програмі виконується створення анонімного канала для реалізації такого ж механізму конвеєризації команд (аналог « | »). У програмі спочатку налаштовуються стандартні потоки введення/виведення та створюється процес з програми прикладу 7.1 **APipeF** вже з перевизначеним потоком стандартного виведення на анонімний канал. Далі налаштовуються потоки введення/виведення та виконується запуск другого процесу з прикладу 7.1 – **APipeC** з перевизначеним потоком введення на анонімний канал. Розроблена програма буде працювати абсолютно ідентично наведеному конвеєру команд як з точки зору отриманих результатів так і з точки

зору використаного механізму передачі даних. *Примітка*: всі три програми повинні знаходитися в одному каталозі.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
HANDLE hRead, hWrite;
STARTUPINFO si;
PROCESS_INFORMATION pi;
HANDLE Proc[2];

int main()
{
SECURITY_ATTRIBUTES saAttr =
      {sizeof (SECURITY_ATTRIBUTES), NULL, TRUE};
// Створюємо анонімний канал
// Якщо виникла помилка – завершуємо роботу програми
if (!CreatePipe (&hRead, &hWrite, &saAttr, 512)) {
      printf ("Error Pipe Creating!!!\n");
      _getch ();
      return 0;
}
// Запускаємо перший процес
ZeroMemory (&si, sizeof(si));
si.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
si.hStdError = GetStdHandle(STD_ERROR_HANDLE);
si.hStdOutput = hWrite;
// встановлюємо стандартні потоки введення, виведення
// та помилок згідно з заданими вище дескрипторами
si.dwFlags = STARTF_USESTDHANDLES;
si.cb = sizeof (si);
ZeroMemory (&pi, sizeof(pi));
if (!CreateProcess(NULL,(LPSTR)"APipeF.exe",NULL,NULL,
```

```
      TRUE,0/*CREATE_NEW_CONSOLE*/,NULL,NULL,&si,&pi))
            {
            printf ("Error APipeF Starting!!!");
            _getch ();
            return 0;
            }
CloseHandle (pi.hThread);
CloseHandle (hWrite);
Proc[0] = pi.hProcess;
// Запускаємо другий процес
ZeroMemory (&si, sizeof(si));
si.hStdInput = hRead;
si.hStdError = GetStdHandle (STD_ERROR_HANDLE);
si.hStdOutput = GetStdHandle (STD_OUTPUT_HANDLE);
si.dwFlags = STARTF_USESTDHANDLES;
si.cb = sizeof(si);
ZeroMemory (&pi, sizeof(pi));
if (!CreateProcess(NULL,(LPSTR)"APipeC.exe",NULL,NULL,
      TRUE,0/*CREATE_NEW_CONSOLE*/,NULL,NULL,&si,&pi))
            {
            printf ("Error APipeC Starting!!!");
            _getch ();
            return 0;
            }
CloseHandle (pi.hThread);
CloseHandle (hRead);
Proc[1] = pi.hProcess;
WaitForMultipleObjects (2, Proc, TRUE, INFINITE);
CloseHandle (Proc[0]);
CloseHandle (Proc[1]);
return 0;
}
```

Якщо в програмі замінити параметр `dwCreateFlags` у зверненнях до функції **CreateProcess** з 0 на CREATE_NEW_CONSOLE (вказано в коментарях у тексті програми), тоді для кожної програми буде створено власну консоль, і дані, що виводяться програмою **APipeC.exe**, будуть відображуватися в іншому вікні від вікна програми **APipeF.exe**.

7.2. Контрольні питання

1. Яким чином можна отримати доступ до анонімного канала?
2. Чому анонімні канали не можуть використовуватись для передачі даних у локальній мережі?
3. Чи може анонімний канал використовуватись для двоспрямованої передачі даних?
4. Які механізми всередині ОС реалізовані на анонімних каналах?
5. Яким чином можна передавати дескриптори анонімних каналів між неспорідненими процесами?

7.3. Індивідуальні завдання

Організувати передачу рядків символів, що вводяться з клавіатури, між процесами 1, 2, 3, …, N за заданою схемою (символ ➔ означає напрямок передачі даних) з використанням анонімних каналів. Передбачити можливість передачі будь-якої кількості рядків. Завершення роботи програм виконувати за командою з клавіатури. Автоматичне переспрямування означає миттєву передачу в канал щойно прийнятого рядка символів з іншого канала. Допускається перетворення даних програмами за власним розсудом. Якщо вказані одразу кілька номерів процесів після символу ➔, це означає, що передачу одних і тих же даних необхідно виконати до всіх вказаних процесів. Для вирішення індивідуального завдання рекомендується розробити програму-менеджер запуску процесів, яка запускається першою, у ній створюються всі необхідні канали та виконується запуск всіх інших програм, яким передаються дескриптори необхідних каналів як параметри запуска. Для кожної програми передавача та приймача на екрані обов’язково відображувати крім самих даних також схему передачі даних (наприклад, «1➔2: переданий рядок» для програми передавача 1, та «1➔2:

прийнятий рядок» для програми приймача 2). Кожна програма повинна виконуватись у власній консолі.

У кожній програмі прийом та передача даних повинні бути реалізовані в окремих нитках: наприклад, якщо програма 2 отримує дані від програми 1 та програми 3, та надсилає дані в програму 1, то вона повинна мати: окрему нитку для читання даних з каналу від 1 програми, окрему нитку для читання даних з каналу від 3 програми та окрему нитку запису в канал для 1 програми.

1) 1➔4; 2➔4; 3➔4

2) 1➔2; 2➔3; 3➔4

3) 1➔2,3; 2➔1,3; 3➔1,2

4) 1➔2,3,4; 3➔2; 4➔1

5) 1➔2; 2➔3; 3➔4 (автоматичне переспрямування)

6) 1➔2; 2➔1

7) 1➔2,3; 2➔3; 3➔1

8) 1➔4; 2➔4; 3➔4

9) 1➔2,3; 2➔3 (автоматичне переспрямування)

10) 1➔2,3; 2➔1 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

11) 1➔2,3,4; 2➔3,4; 3➔4

12) 1➔2,3; 2➔1,3; 3➔1,2

13) 1➔2; 3➔4; 2➔3 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

14) 1➔2,3,4 (автоматичне переспрямування); 2➔1; 3➔1; 4➔1

15) 1➔2; 2➔3; 3➔2 (автоматичне переспрямування)

16) 1➔3; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування)

17) 1➔2,3; 2➔1,3 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

18) 1➔2,3,4; 2➔1; 3➔1; 4➔1

19) 1➔2,3,4; 4➔2,3,4

20) 1➔2; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

8. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ ІМЕНОВАНИХ КАНАЛІВ В ОС WINDOWS

Мета: вивчити можливості використання механізму обміну даними між процесами в ОС Windows на основі іменованих каналів.

8.1. Теоретичні відомості

Іменовані канали на відміну від анонімних мають імена, за якими різні процеси можуть отримувати доступ до них. Також іменовані канали дозволяють виконувати двоспрямовану передачу даних. На відміну від анонімних каналів іменовані канали дозволяють виконувати передачу даних у мережах та використовувати асинхронне введення/виведення. У разі, якщо виконується з'єднання декількох процесів через один канал – створюється декілька екземплярів цього каналу.

Імена каналів у загальному випадку формуються в форматі UNC (Universal Naming Convention – повні системні імена для позначення ресурсів мережі) та мають такий вигляд:

```
\\ІмʼяСерверу\pipe\ІмʼяКаналу
```

Якщо процес відкриває канал, створений на іншій робочій станції, він повинен указати ім'я сервера. Якщо ж процес створює канал або відкриває його на своїй робочій станції, замість імені сервера достатньо вказати символ крапки:

```
\\.\pipe\ІмʼяКаналу
```

У будь якому випадку процес може створити канал тільки на тій робочій станції, де він був запущений. Тому під час створення каналу ім'я сервера ніколи не вказується. В іменах каналів регістр не враховується, тому наступні два імені описують один і той же канал:

```
\\.\pipe\MyPipe
```

```
\\.\PIPE\myPipE
```

Для роботи з іменованими каналами використовуються такі функції: **CreateNamedPipe** (створення іменованого каналу), **ConnectNamedPipe** (встановлення з'єднання з каналом з боку сервера. Після того як серверний процес створив канал, він може перейти в режим з'єднання з клієнтським процесом), **CreateFile** (встановлення з'єднання з каналом з боку клієнта. За замовчанням канал відкривається в режимі потоку байтів), **DisconnectNamedPipe** (відключення серверного процесу від клієнтського), **CallNamedPipe** (підключення до іменованого каналу в режимі повідомлень, запис у нього, читання відповіді та відключення від каналу), **GetNamedPipeHandleState** та **GetNamedPipeInfo** (отримання інформації про вказаний іменований канал), **PeekNamedPipe** (читання даних з каналу без їх знищення в ньому), **SetNamedPipeHandleState** (встановлення режиму роботи каналу), **TransactNamedPipe** (оформлення запису в канал та читання відповіді з нього як єдиної операції), **WaitNamedPipe** (якщо клієнту неможливо підключитися до каналу, очікування вказаного часу, поки канал не стане доступним для підключення).

Параметри деяких вказаних функцій:

```
BOOL CallNamedPipe(
 LPCTSTR lpNamedPipeName, // ім'я каналу
 LPVOID lpInBuffer,    // покажчик на буфер запису
 DWORD nInBufferSize, // розмір буфера запису в байтах
 LPVOID lpOutBuffer,   // покажчик на буфер читання
 DWORD nOutBufferSize, // розмір буфера читання
 LPDWORD lpBytesRead,  // число прочитаних байтів
 DWORD nTimeOut   // час очікування виконання операції
);

BOOL ConnectNamedPipe(
 HANDLE hNamedPipe,   // дескриптор іменованого каналу
 LPOVERLAPPED lpOverlapped   // адреса структури
```

```
//для асинхронного введення/виведення з перекриттям
);

HANDLE CreateNamedPipe(
 LPCTSTR lpName,      // ім'я каналу
 DWORD dwOpenMode,   // режим відкриття каналу (режими
      // читання, запису, захисту
 DWORD dwPipeMode,  // режим каналу (потік байтів,
      // повідомлення, блокування)
 DWORD nMaxInstances, // кількість екземплярів каналу
 DWORD nOutBufferSize, // розмір буфера виведення
 DWORD nInBufferSize,  // розмір буфера введення
 DWORD nDefaultTimeOut, // час очікування виконання
      // операції за замовчанням
 LPSECURITY_ATTRIBUTES lpSecurityAttributes //покажчик
      // на структуру атрибутів безпеки
);

BOOL PeekNamedPipe(
 HANDLE hNamedPipe,   // дескриптор іменованого каналу
 LPVOID lpBuffer,     // покажчик на буфер читання
 DWORD nBufferSize,   // розмір буфера в байтах
 LPDWORD lpBytesRead, // число прочитаних байтів
 LPDWORD lpTotalBytesAvail, // покажчик на загальне
      // число байтів, доступних у каналі
 LPDWORD lpBytesLeftThisMessage // покажчик на число
      // байтів ще доступних у каналі
);

BOOL TransactNamedPipe(
 HANDLE hNamedPipe,   // дескриптор іменованого каналу
 LPVOID lpInBuffer,   // покажчик на буфер запису
 DWORD nInBufferSize, // розмір буфера запису в байтах
```

```
 LPVOID lpOutBuffer,  // покажчик на буфер читання
 DWORD nOutBufferSize,// розмір буфера читання
 LPDWORD lpBytesRead, // число прочитаних байтів
 LPOVERLAPPED lpOverlapped  // адреса структури
// для асинхронного введення/виведення з перекриттям
);

BOOL SetNamedPipeHandleState(
 HANDLE hNamedPipe,   // дескриптор іменованого каналу
 LPDWORD lpMode,      // новий режим каналу
 LPDWORD lpMaxCollectionCount, // максимальний розмір
      // порції даних для передачі
 LPDWORD lpCollectDataTimeout // час очікування
      // виконання операції
);

BOOL WaitNamedPipe(
 LPCTSTR lpNamedPipeName,  // ім'я каналу
 DWORD nTimeOut   // час очікування виконання операції
);
```

Читання та запис даних у відкритому каналі виконується за допомогою функцій **WriteFile** та **ReadFile**, подібно записуванню та читанню в звичайному файлі. Для закриття ідентифікатора каналу використовується функція закриття дескриптора **CloseHandle**.

Приклад 8.1. Передача даних (синхронна) від програми-клієнта програмі-серверу за допомогою іменованого каналу на одній і тій самій робочій станції. Після отримання даних від клієнта, сервер відправляє клієнту відповідь і робота закінчується.

Текст програми–сервера:

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

```
#include <conio.h>
BOOL fConnected;
HANDLE hNamedPipe;
LPSTR lpszPipeName=(LPSTR)"\\\\.\\pipe\\$MyFirstPipe";
char szBuf[512];
DWORD cbRead, cbWritten;

int main()
{
      // Створення каналу з ім'ям lpszPipeName
      hNamedPipe = CreateNamedPipe(lpszPipeName,
            PIPE_ACCESS_DUPLEX, PIPE_TYPE_MESSAGE |
            PIPE_READMODE_MESSAGE | PIPE_WAIT,
            PIPE_UNLIMITED_INSTANCES,
            512, 512, 5000, NULL);
      // Якщо виникла помилка – завершити роботу
      if (hNamedPipe == INVALID_HANDLE_VALUE)
      {
            printf("Error Pipe Creating!!!\n");
            _getch();
            return 0;
      }
      // Очікування підключення до каналу клієнта
      fConnected = ConnectNamedPipe(hNamedPipe, NULL);
      // Якщо виникла помилка – завершити роботу
      if (!fConnected)
      {
            printf("Error Pipe Connecting!!!\n");
            CloseHandle(hNamedPipe);
            _getch();
            return 0;
      }
      // Отримання даних з каналу
```

```
      if (ReadFile(hNamedPipe,szBuf,512,&cbRead,NULL))
            printf("Received %d bytes: <%s>\n",
                  cbRead, szBuf);
      else printf("Error Data Transfer!!!\n");
      strcpy_s(szBuf, "OK");
      // Запис даних у канал
      if (!WriteFile(hNamedPipe,szBuf,strlen(szBuf)+1,
            &cbWritten, NULL))
            printf("Error Data Transfer!!!\n");
      else printf("Replied %d bytes: <%s>\n",
            cbWritten, szBuf);
      CloseHandle(hNamedPipe);
      _getch();
      return 0;
}
```

Текст програми клієнта:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
HANDLE hNamedPipe;
DWORD cbWritten, cbRead;
char szBuf[512];
LPSTR lpszPipeName=(LPSTR)"\\\\.\\pipe\\$MyFirstPipe";

int main()
{
      // Цикл встановлення з'єднання з каналом
      while (1) {
            hNamedPipe = CreateFile(lpszPipeName,
                  GENERIC_READ | GENERIC_WRITE, 0,
                  NULL, OPEN_EXISTING, 0, NULL);
            // Помилки нема – вихід з циклу
```

```
        if (hNamedPipe != INVALID_HANDLE_VALUE)
              break;
        // Помилка є, якщо канал не зайнятий-вихід
        if (GetLastError() != ERROR_PIPE_BUSY) {
              printf("Error Pipe Opening!!!\n");
              _getch();
              return 0;
        }
        // Канал зайнятий-очікуємо його звільнення
        if (!WaitNamedPipe(lpszPipeName,
              NMPWAIT_WAIT_FOREVER))
        {
              printf("Error Pipe Opening!!!\n");
              _getch();
              return 0;
        }
}
// Встановлення режиму передачі повідомлень
DWORD dwPipeState = PIPE_READMODE_MESSAGE;
SetNamedPipeHandleState(hNamedPipe,&dwPipeState,
      NULL, NULL);
// Передача даних серверу
strcpy_s(szBuf, "Test Pipe Connection");
if (!WriteFile(hNamedPipe,szBuf,strlen(szBuf)+1,
      &cbWritten, NULL))
      printf("Error Data Transfer!!!\n");
else printf("Transferred %d bytes: <%s>\n",
      cbWritten, szBuf);
if (ReadFile(hNamedPipe,szBuf,512,&cbRead,NULL))
      printf("Received %d bytes: <%s>\n",
            cbRead, szBuf);
else printf("Error Data Transfer!!!\n");
CloseHandle(hNamedPipe);
```

```
	_getch();
	return 0;
}
```

У даному прикладі виконується однократна передача та прийом відповіді, програму клієнта можна модифікувати– виконати заміну операцій запису та читання на одну функцію **TransactNamedPipe**:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
HANDLE hNamedPipe;
DWORD cbRead;
char szBuf[512];
LPSTR lpszPipeName=(LPSTR)"\\\\.\\pipe\\$MyFirstPipe";

int main()
{
	hNamedPipe = CreateFile(lpszPipeName,
		GENERIC_READ | GENERIC_WRITE, 0, NULL,
		OPEN_EXISTING, 0, NULL);
	if (hNamedPipe == INVALID_HANDLE_VALUE)
	{
		printf("Error Pipe Creating!!!\n");
		_getch();
		return 0;
	}
	DWORD dwPipeState = PIPE_READMODE_MESSAGE;
	SetNamedPipeHandleState(hNamedPipe,&dwPipeState,
		NULL, NULL);
	strcpy_s(szBuf, "Test Pipe Connection");
	printf("Client will send: <%s>\n", szBuf);
	if (TransactNamedPipe(hNamedPipe, szBuf,
```

```
            sizeof(szBuf), szBuf, sizeof(szBuf),
            &cbRead, NULL))
            printf("Received %d bytes: <%s>\n",
                  cbRead, szBuf);
      CloseHandle(hNamedPipe);
      _getch();
      return 0;
}
```

У клієнтській частині даного прикладу можна також відмовитись від операцій відкриття та закриття каналу і використовувати функцію **CallNamedPipe**:

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
DWORD cbRead;
char szBuf[512];
LPSTR lpszPipeName=(LPSTR)"\\\\.\\pipe\\$MyFirstPipe";

int main()
{
      strcpy_s(szBuf, "Test Pipe Connection");
      printf("Will send: <%s>\n", szBuf);
      if (CallNamedPipe(lpszPipeName, szBuf,
            sizeof(szBuf), szBuf, sizeof(szBuf),
            &cbRead, NMPWAIT_WAIT_FOREVER))
            printf("Received %d bytes: <%s>\n",
                  cbRead, szBuf);
      _getch();
      return 0;
}
```

8.2. Контрольні питання

1. Яким чином можна отримати доступ до іменованого канала?
2. Яка можлива проблема є при створенні іменованих каналів на відміну від анонімних?
3. Які дії виконує функція **TransactNamedPipe**?
4. Які дії виконує функція **CallNamedPipe**?
5. Порівняйте між собою анонімні та іменовані канали.

8.3. Індивідуальні завдання

Організувати передачу рядків символів, що вводяться з клавіатури, між процесами 1, 2, 3, …, N за заданою схемою (символ ➔ означає напрямок передачі даних) з використанням іменованих каналів. Передбачити можливість передачі будь-якої кількості рядків. Завершення роботи програм виконувати за командою з клавіатури. Автоматичне переспрямування означає миттєву передачу в канал щойно прийнятого рядка символів з іншого каналу. Допускається перетворення даних програмами за власним розсудом. Якщо вказані одразу кілька номерів процесів після символу ➔, це означає, що передачу одних і тих же даних необхідно виконати до всіх вказаних процесів. Для вирішення індивідуального завдання рекомендується розробити програму-менеджер запуску процесів, яка запускається першою, у ній створюються всі необхідні канали та виконується запуск всіх інших програм, яким передаються імена необхідних каналів як параметри запуска. Для кожної програми передавача та приймача на екрані обов'язково відображувати крім самих даних також схему передачі даних (наприклад, «1➔2: переданий рядок» для програми передавача 1, та «1➔2: прийнятий рядок» для програми приймача 2). Кожна програма повинна виконуватись у власній консолі.

У кожній програмі прийом та передача даних повинні бути реалізовані в окремих нитках: наприклад, якщо програма 2 отримує дані від програми 1 та програми 3, та надсилає дані в програму 1, то вона повинна мати: окрему нитку для читання даних з каналу від 1 програми, окрему нитку для читання даних з каналу від 3 програми та окрему нитку запису в канал для 1 програми.

1) 1➔4; 2➔4; 3➔4

2) 1➔2; 2➔3; 3➔4

3) 1➔2,3; 2➔1,3; 3➔1,2

4) 1➔2,3,4; 3➔2; 4➔1

5) 1➔2; 2➔3; 3➔4 (автоматичне переспрямування)

6) 1➔2; 2➔1

7) 1➔2,3; 2➔3; 3➔1

8) 1➔4; 2➔4; 3➔4

9) 1➔2,3; 2➔3 (автоматичне переспрямування)

10) 1➔2,3; 2➔1 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

11) 1➔2,3,4; 2➔3,4; 3➔4

12) 1➔2,3; 2➔1,3; 3➔1,2

13) 1➔2; 3➔4; 2➔3 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

14) 1➔2,3,4 (автоматичне переспрямування); 2➔1; 3➔1; 4➔1

15) 1➔2; 2➔3; 3➔2 (автоматичне переспрямування)

16) 1➔3; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування)

17) 1➔2,3; 2➔1,3 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

18) 1➔2,3,4; 2➔1; 3➔1; 4➔1

19) 1➔2,3,4; 4➔2,3,4

20) 1➔2; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

9. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ ПОШТОВИХ СКРИНЬОК В ОС WINDOWS

Мета: вивчити можливості використання механізму обміну даними між процесами в ОС Windows на основі поштових скриньок.

9.1. Теоретичні відомості

Поштові скриньки (mailslots) реалізують односпрямований протокол обміну даними між процесами як на локальній станції, так і в мережі. Обмін виконується на основі датаграм, що не забезпечує надійність – немає гарантії доставки пакетів (процес, який виконує передачу, не отримує інформацію про доставку). Поштові скриньки дозволяють виконувати тільки односпрямовану передачу даних від одного або декількох клієнтів до одного або декількох серверів.

Головна особливість поштових скриньок полягає в тому, що вони, на відміну від каналів, дозволяють передавати дані в широкомовному режимі. Це означає, що на комп'ютері або в мережі можуть робити декілька серверних процесів, здатних отримувати повідомлення через поштові скриньки. При цьому один клієнтський процес може посилати повідомлення одразу всім цим серверним процесам. Прикладом використання поштових скриньок є реалізація служби обміну повідомленнями Messenger в операційних системах Windows на базі ядра NT. Коли ця служба запущена на виконання, на робочій станції створюється поштова скринька з ім'ям «Messngr». Коли використовується передача повідомлення за допомогою команди NET SEND, клієнтська машина записує його в поштову скриньку з ім'ям «Messngr» для вказаної робочої станції або для всіх одразу (у разі широкомовного режиму передачі повідомлення).

Створення поштової скриньки. Поштова скринька, тобто Mailslot, створюється серверним процесом за допомогою функції **CreateMailslot**. Після створення поштової скриньки серверний процес отримує ідентифікатор Mailslot, користуючись яким, сервер може читати повідомлення, які посилають у поштову скриньку клієнтські процеси. Але сервер не може виконувати над Mailslot операцію запису, тому що ця поштова скринька

призначена тільки для односпрямованої передачі даних – від клієнта до сервера.

Ім'я «Mailslot» задається подібно імені каналу «Pipe» (нижче наведений синтаксис для створення поштової скриньки на своїй робочій станції):

```
\\.\mailslot\[Шлях]Ім'яПоштовоїСкриньки
```

Щоб відкрити Mailslot, створений на іншій робочій станції в мережі, рядок імені повинен мати такий вигляд:

```
\\Ім'яРобочоїСтанції\mailslot\[Шлях]Ім'яСкриньки
```

Можна відкрити Mailslot для передачі повідомлень усім робочим станціям заданого домена. Для цього необхідно задати ім'я за таким зразком:

```
\\Ім'яДомена\mailslot\[Шлях]Ім'яПоштовоїСкриньки
```

Для передачі повідомлень одночасно всім робочим станціям мережі основного домена ім'я задається таким чином:

```
\\*\mailslot\[Шлях]Ім'яПоштовоїСкриньки
```

В останніх двох випадках розмір повідомлень обмежується 400 байтами. Параметри функції створення поштової скриньки:

```
HANDLE CreateMailslot(
 LPCTSTR lpName,     // ім'я поштової скриньки
 DWORD nMaxMessageSize,//максимальний розмір Message
 DWORD lReadTimeout, // максимальний інтервал
      // очікування читання
 LPSECURITY_ATTRIBUTES lpSecurityAttributes //покажчик
      // на структуру атрибутів безпеки
);
```

Запис/читання даних у поштових скриньках здійснюється подібно запису/читанню в каналах.

Визначення стану Mailslot. Серверний процес може визначити поточний стан Mailslot за його ідентифікатором за допомогою функції **GetMailslotInfo**. Параметри функції **GetMailslotInfo**:

```
BOOL GetMailslotInfo(
 HANDLE hMailslot,     // дескриптор поштової скриньки
 LPDWORD lpMaxMessageSize,  // максимальний розмір
      // повідомлення
 LPDWORD lpNextSize, // розмір наступного повідомлення
      // (якщо воно є, або 0)
  LPDWORD lpMessageCount,    // кількість непрочитаних
      // повідомлень у поштовій скриньці
 LPDWORD lpReadTimeout      // максимальний інтервал
      // очікування читання
);
```

Для встановлення нового максимального інтервалу очікування читання використовується функція **SetMailslotInfo**. Її параметри:

```
BOOL SetMailslotInfo(
 HANDLE hMailslot,    // дескриптор поштової скриньки
 DWORD lReadTimeout   // максимальний інтервал
      // очікування читання
);
```

Приклад 9.1. Передача одного рядка текстових даних від клієнта серверу за допомогою поштової скриньки на одній і тій самій робочій станції. Після передачі даних робота програм завершується.

Текст програми–сервера.

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

```
#include <stdio.h>
#include <conio.h>
BOOL fReturnCode; // Код повернення з функцій
DWORD cbMessages; // Розмір повідомлення
DWORD cbMsgNumber; // Кількість повідомлень у Mailslot
HANDLE hMailslot;
LPSTR lpszMailslotName =
(LPSTR)"\\\\.\\mailslot\\$MyFirstMailslot";
char szBuf[512]; // Буфер для прийому даних
DWORD cbRead; // Кількість байтів даних, прийнятих
      // через Mailslot

int main()
{
// Створення Mailslot, з ім'ям lpszMailslotName
hMailslot = CreateMailslot(lpszMailslotName, 0,
      MAILSLOT_WAIT_FOREVER, NULL);
// Якщо виникла помилка – вихід з програми
if (hMailslot == INVALID_HANDLE_VALUE)
      {
      printf("Error MailSlot Creating!!!\n");
      _getch();
      return 0;
      }
while (1)
      { // Визначення стану Mailslot
      fReturnCode = GetMailslotInfo(hMailslot, NULL,
            &cbMessages, &cbMsgNumber, NULL);
      if (!fReturnCode)
      {
            printf("Get MailSlotInfo Error!!!\n");
            _getch();
            return 0;
```

```
      }
      // Якщо в Mailslot є повідомлення, читаємо перше
      if (cbMsgNumber != 0)
      {
            if (ReadFile(hMailslot, szBuf, 512,
                  &cbRead, NULL))
                  printf("Received %d bytes: <%s>\n",
                        cbRead, szBuf);
            else printf("Error Data Transfer!!!\n");
            break;
      }
}
_getch();
CloseHandle(hMailslot);
return 0;
}
```

Текст програми–клієнта.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
HANDLE hMailslot;
char szBuf[512]; // Буфер для передачі даних
DWORD cbWritten; // Кількість байтів, переданих
      // через Mailslot
LPSTR lpszMailslotName =
(LPSTR)"\\\\.\\mailslot\\$MyFirstMailslot";
int main()
{
// Відкриття Mailslot
hMailslot = CreateFile(lpszMailslotName,
      GENERIC_WRITE, FILE_SHARE_READ, NULL,
      OPEN_EXISTING, 0, NULL);
```

```
// Якщо виникла помилка – вихід з програми
if (hMailslot == INVALID_HANDLE_VALUE)
{
      printf("CreateFile Error!!!\n");
      _getch();
      return 0;
}
// Передача даних через Mailslot
strcpy_s(szBuf, "Test MailSlot Connection");
if (!WriteFile(hMailslot, szBuf, strlen(szBuf) + 1,
      &cbWritten, NULL))
      printf("Error Data Transfer!!!\n");
else printf("Transferred %d bytes: <%s>\n",
      cbWritten, szBuf);
CloseHandle(hMailslot);
_getch();
return 0;
}
```

9.2. Контрольні питання

1. Який процес (передавач або приймач) може створювати поштову скриньку?

2. Чи можлива двоспрямована передача даних через поштову скриньку?

3. Чи можлива широкомовна передача даних через поштові скриньки? Яким чином?

4. Порівняйте між собою анонімні канали та поштові скриньки.

5. Порівняйте між собою іменовані канали та поштові скриньки.

6. Яким чином організувати можливість прийому даних в одну поштову скриньку від кількох передавачів?

9.3. Індивідуальні завдання

Організувати передачу рядків символів, що вводяться з клавіатури, між процесами 1, 2, 3, …, N за заданою схемою (символ ➔ означає напрямок передачі даних) з використанням поштових скриньок. Передбачити можливість передачі будь-якої кількості рядків. Завершення роботи програм виконувати за командою з клавіатури. Автоматичне переспрямування означає миттєву передачу в поштову скриньку щойно прийнятого рядка символів з іншої поштової скриньки. Допускається перетворення даних програмами за власним розсудом. Якщо вказані одразу кілька номерів процесів після символу ➔, це означає, що передачу одних і тих же даних необхідно виконати до всіх вказаних процесів. Для вирішення індивідуального завдання рекомендується розробити програму-менеджер запуску процесів, яка запускається першою, у ній виконується запуск всіх інших програм, в яких (якщо виконується прийом даних) створюються поштові скриньки або (якщо ведеться передача даних) відкриваються поштові скриньки. Імена необхідних поштових скриньок та режими роботи з ними передаються програмам як параметри запуска. Для кожної програми передавача та приймача на екрані обов'язково відображувати крім самих даних також схему передачі даних (наприклад, «1➔2: переданий рядок» для програми передавача 1, та «1➔2: прийнятий рядок» для програми приймача 2). Кожна програма повинна виконуватись у власній консолі.

У кожній програмі прийом та передача даних повинні бути реалізовані в окремих нитках: наприклад, якщо програма 2 отримує дані від програми 1 та програми 3, та надсилає дані в програму 1, то в ній потрібно реалізувати окрему нитку для читання даних з поштової скриньки від 1 програми, окрему нитку для читання даних з поштової скриньки від 3 програми та окрему нитку запису в поштову скриньку для 1 програми.

1) 1➔4; 2➔4; 3➔4
2) 1➔2; 2➔3; 3➔4
3) 1➔2,3; 2➔1,3; 3➔1,2
4) 1➔2,3,4; 3➔2; 4➔1
5) 1➔2; 2➔3; 3➔4 (автоматичне переспрямування)

6) 1➔2; 2➔1

7) 1➔2,3; 2➔3; 3➔1

8) 1➔4; 2➔4; 3➔4

9) 1➔2,3; 2➔3 (автоматичне переспрямування)

10) 1➔2,3; 2➔1 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

11) 1➔2,3,4; 2➔3,4; 3➔4

12) 1➔2,3; 2➔1,3; 3➔1,2

13) 1➔2; 3➔4; 2➔3 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

14) 1➔2,3,4 (автоматичне переспрямування); 2➔1; 3➔1; 4➔1

15) 1➔2; 2➔3; 3➔2 (автоматичне переспрямування)

16) 1➔3; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування)

17) 1➔2,3; 2➔1,3 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

18) 1➔2,3,4; 2➔1; 3➔1; 4➔1

19) 1➔2,3,4; 4➔2,3,4

20) 1➔2; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

10. РОБОТА З ФАЙЛОВОЮ СИСТЕМОЮ В ОС WINDOWS

Мета: навчитись створювати прикладні програми в операційній системі Windows для виконання роботи з файловою системою на рівні використання функцій WinAPI.

10.1. Теоретичні відомості

ОС Windows підтримує такі базові типи файлових систем:

▪ *FAT12, FAT16, FAT32 (File Allocation Table) та exFAT* – файлові системи, що використовувались в ОС MS-DOS та Windows 95, 98. Їх підтримка зумовлена необхідністю забезпечувати сумісність з іншими операційними системами, для підтримки застарілих програм та для формату гнучких дисків (FAT12). Теоретично FAT32 дозволяє використовувати розділи розміром до 8 ТБайтів, але в Windows розмір розділу обмежується 32 ГБайтами (теоретично може підтримуватися і більший розмір, але створювати можна тільки до 32 ГБайтів). Максимальний розмір файла у файловій системі FAT32 дорівнює 4 ГБайтам, що обумовлено 32-розрядним числом, яке задає розмір файла в елементі каталогу.

В основі цієї файлової системи лежить таблиця, яка діє як покажчик її вмісту. Загальна структура цієї ФС складається з трьох окремих областей:

- завантажувальний сектор – перший сектор будь-якого розділу, відформатованого у FAT, який містить важливу інформацію про його організацію;

- таблиця розміщення файлів (*File Allocation Table* або FAT), а також її резервна копія, до якої можна отримати доступ, якщо виникне проблема з читанням оригінальної таблиці;

- область зберігання даних – розділена на кластери. Кластер складається з суміжних секторів і використовується як мінімальна одиниця розподілу пам’яті. Його розмір є фіксованим, але може коливатися від 512 байтів до 64 кілобайтів, залежно від розміру розділу та версії FAT. Файл, незалежно від того, наскільки він малий, займає весь кластер, а незайнятий простір, таким чином, витрачається даремно. Якщо для файла потрібні кілька кластерів, система може зайняти послідовний ланцюжок кластерів або

розмістити його в кластери, розкидані по всьому розділу, що призводить до фрагментації файла. Кожен кластер має відповідний запис у таблиці розміщення файлів. Нульове значення в запису означає, що кластер на даний момент не використовується, тоді як не нульове може вказувати на наступний кластер того самого файла або спеціальний індикатор його кінця.

Каталоги, як і файли, знаходяться в області зберігання даних. Вони складаються з записів каталогу довжиною 32 байти, кожен з яких описує файл, що зберігається в цьому каталозі (або його підкаталозі). Окрім імені, розміру та інших атрибутів файла, запис каталогу містить інформацію про перший кластер файла, а будь-який наступний кластер можна знайти через таблицю розміщення файлів, використовуючи її як список з посиланнями.

FAT32 все ще активно використовується, головним чином завдяки своїй широкій сумісності. До неї можна отримати доступ майже з будь-якої операційної системи, включаючи macOS і Linux, що робить її гарною альтернативою для портативних пристроїв, таких як карти пам’яті та USB-накопичувачі. Цей формат також підтримується смартфонами, цифровими камерами, ігровими консолями та іншими гаджетами.

exFAT (*Extended File Allocation Table*, Розширена таблиця розміщення файлів) не має суттєвих обмежень щодо розміру та часто використовується на зовнішніх жорстких дисках, твердотільних накопичувачах, флеш-накопичувачах USB великої ємності тощо.

▪ *NTFS (New Technology File System)* – файлова система для операційних систем, починаючи з Windows NT. Операційні системи лінійки Windows Server також використовують цей формат. Максимальний розмір розділу з файловою системою NTFS досягає 16 ЕкзаБайтів (16 мільярдів ГБайтів), але Windows обмежує розмір до 128 ТБайтів, що пов’язано з використанням 32-розрядної адресації кластерів. Також у NTFS підтримуються захист файлів та каталогів, шифрування та стиснення. Файлова система забезпечує безпеку даних та високу стійкість.

NTFS досить надійна завдяки журналюванню та пропонує контроль доступу, шифрування, стиснення файлів тощо. Крім того, вона використовує вдосконалені методи організації даних, які дозволяють краще викорис-

товувати простір для зберігання та роблять її набагато менш схильною до фрагментації. Вся файлова система спирається на кілька службових файлів:

- файл $Boot;
- файл $MFT (*Master File Table*, Головна файлова таблиця);
- файл $Bitmap;
- $LogFile та інші.

Файл $Boot бере участь у процесі завантаження ОС і містить багато важливих параметрів ФС.

Головна файлова таблиця (MFT) має запис для кожного файла у файловій системі. Ці записи називаються атрибутами і можуть містити будь-яку інформацію: від назви файла, розміру, дозволів, часу створення/останньої зміни до фактичного вмісту. Якщо цей вміст завеликий, щоб поміститися в запис Головної файлової таблиці (розмір запису становить 1024 байти), NTFS виділяє для нього кластери за межами Таблиці і створює покажчики на їх розташування. Інші атрибути також можуть бути занадто великими для запису в MFT, наприклад, довгі імена файлів. Такі атрибути також отримують окремі кластери.

Кластери зазвичай виділяються послідовностями, які називаються екстентами. NTFS завжди намагається розмістити вміст в одному екстенті. Проте, якщо суміжні кластери недоступні, вона створює новий екстент в іншому місці, розділяючи файл на фрагменти. Каталоги в NTFS зберігаються як файли, але замість типового вмісту (даних) у таких файлах зберігаються списки імен файлів і посилань, що ідентифікують ці файли.

Файлова система NTFS підтримує потоки даних: без імені – основний, його розмір можна отримати при перегляді змісту каталогу, та будь яку кількість іменованих потоків, які може створювати користувач. Для звернення до іменованого потоку необхідно в імені файла вказати через двокрапку ім’я потоку.

Розглянемо простий приклад – створення іменованого потоку для файла. За допомогою текстового редактору створимо звичайний файл TEST.TXT з текстом – «MAIN STREAM». Далі, за допомогою команди ECHO запишемо рядок «MY STREAM. DON’T TOUCH THIS» у потік «My»:

```
ECHO MY STREAM. DON'T TOUCH THIS > TEST.TXT:MY
```

Тепер виведемо зміст основного потоку на екран:

```
MORE < TEST.TXT
```

Отримаємо результат: MAIN STREAM

Виведемо зміст додаткового потоку:

```
MORE < TEST.TXT:MY
```

Результат: MY STREAM. DON'T TOUCH THIS

Але будь який менеджер файлів показує розмір файла – 11 байтів (розмір основного потоку). Це дає можливість в іменованих потоках зберігати додаткову інформацію про файл, наприклад, дані про захист від несанкціонованого копіювання, контрольну суму та ін.

Файл $Bitmap відстежує стан кластерів. Кожен біт у ньому представляє один кластер і може мати значення 1, коли кластер зайнятий, або 0, коли він вільний.

Перш ніж змінити будь-яку зі своїх ключових структур, NTFS спочатку записує ці зміни до $LogFile. Цей журнал дає можливість відновити їх у разі будь-яких невідповідностей, які можуть бути спричинені збоєм під час оновлення. Якщо під час нормальної роботи виникає помилка, NTFS визначає несправний кластер, занотовує його у файл $BadClus і копіює дані в інше місце.

NTFS добре підходить для внутрішнього використання на комп'ютерах з ОС Windows. З іншого боку, такі пристрої, як карти пам'яті чи флеш-накопичувачі USB, можуть потребувати більш легкої файлової системи, яка до того ж залишатиметься доступною навіть за межами середовища Windows. Одна з причин – досить великі обсяги структур даних, які використовуються для підтримки роботи цієї файлової системи.

▪ *ReFS* (*Resilient File System*, гнучка файлова система) – остання розробка Microsoft, випущена разом з Windows Server 2012 і пізніше додана до Windows 8.1. Зараз вона також доступна на Windows 11. ReFS була розроблена для усунення певних недоліків NTFS, зокрема тих, що стосуються пошкодження даних. Завдяки механізму копіювання при записуванні (*Copy-on-Write (CoW)*) вона має набагато вищу стійкість до збоїв. Під час редагування існуючих метаданих ReFS зберігає їх копію в іншій області носія даних і замість того, щоб перезаписувати їх прямо на місці, оновлює копію та створює посилання з нової копії на відповідний файл. Таким чином, у різних місцях на носії зберігається значна кількість старих копій, що дозволяє легко відновити цілісність файлової системи та запобігти втраті даних. ReFS також використовує контрольні суми (*checksums*), які дозволяють швидко виявляти будь-які можливі пошкодження даних.

Архітектура ReFS кардинально відрізняється від архітектури інших файлових систем Windows. Вона використовує B+-дерева як основну структуру на диску для представлення як метаданих, так і даних файлів. Таке дерево складається з кореня, внутрішніх вузлів і листя. Кожен вузол дерева має впорядкований список ключів або покажчиків на вузли нижчого рівня (листя).

Така конструкція робить ReFS оптимальним форматом для систем зберігання великого розміру та високої доступності. Але, незважаючи на свої очевидні переваги, вона не настільки стабільна, як NTFS, і не може забезпечити сумісність з іншими пристроями на базі Windows.

▪ *HPFS* (*High Performance File System*, високопродуктивна файлова система) була створена корпорацією Microsoft у співпраці з IBM і представлена разом з OS/2 1.20 у 1989 році як файлова система для серверів, яка може забезпечити набагато кращу продуктивність порівняно з FAT.

На відміну від FAT, HPFS намагається розміщувати файли в суміжних блоках або принаймні таким чином, щоб їх фрагменти (так звані екстенти) були розташовані максимально близько один до одного.

На початку HPFS є три блоки керування, що займають 18 секторів: завантажувальний блок, суперблок і запасний блок. Решта простору зберігання поділена на порції суміжних секторів, які називаються смугами, по 8

МБ кожна. Кожна смуга має власну бітову мапу (bitmap) розподілу секторів, що показує, які сектори зайняті (1 – зайнятий, 0 – вільний).

Кожен файл і каталог має власний F-вузол, розташований поруч із ним на диску – ця структура містить інформацію про місцезнаходження файла та його розширені атрибути. Для зберігання каталогів використовується спеціальна смуга каталогів, розташована в центрі диска, а сама структура каталогів являє собою збалансоване дерево із записами в алфавітному порядку. HPFS має значні обмеження і з часом застаріла. Вона перестала підтримуватись в ОС Windows, починаючи з Windows NT 4.

▪ *CDFS (CD-rom File System)* – призначена для доступу до даних, записаних на компакт диски. Режим роботи з файловою системою – тільки для читання. Існує два базові різновиди CDFS: формат ISO-9660 та Joliet. У першому випадку імена файлів можуть бути тільки в коді ASCII у верхньому регістрі і довжина імені не перевищує 32 символи. У другому випадку використовується код UNICODE і довжина імен не обмежується. Загальні обмеження на CDFS – максимальний розмір файла не більше 4 ГБайтів (тому в цій файловій системі неможливо записати на DVD диск файл розміром більш ніж 4 Гбайти) та кількість каталогів не повинна перевищувати 65535.

▪ *UDF (Universal Disk Format)* – файлова система для магнето-оптичних носіїв типу DVD дисків. Розмір файла обмежується 64-розрядним значенням, довжина імен файлів та каталогів обмежується 254 символами ASCII або 127 символами UNICODE.

При роботі з файлами в ОС Windows використовуються такі правила:

▪ повне ім’я файла містить символьне позначення диску;

▪ ім’я файла можна задати у вигляді UNC імені;

▪ в якості розділювача використовується символ зворотної риски « \ » але в параметрах функцій WinAPI можна використовувати і символ прямої риски « / »;

▪ в іменах файлів не допускається використання символів з ASCII кодами від 1 до 31, а також « < », « > », « **:** », « " », « | », « **?** », « ***** », « \ », « / »;

▪ в іменах можуть бути присутні пробіли, для правильної інтерпретації таких імен файлів у командному рядку ім’я необхідно брати в лапки;

▪ регістр в іменах файлів не розрізняється, але він зберігається на диску;

▪ довжина імені файла та каталогу не перевищує 255 символів;

▪ ім’я файла та його розширення розділюються символом крапки;

▪ один символ крапки та два символи крапки в імені файла позначають поточний та батьківський каталог.

Для роботи з файловою системою використовуються наступні функції (їх значно більше, але для вивчення теми достатньо використовувати вказані): **CreateFile** (створення або відкриття об’єкта ядра «файл»); **GetVolumeInformation** (отримання інформації про розділ, де розміщений вказаний кореневий каталог); **FindFirstFile** (пошук файла в каталозі з заданим іменем. Ім’я можна задати за допомогою маски); **FindFirstFileEx** (пошук файла в каталозі з заданим іменем і атрибутами); **FindNextFile** (продовження пошуку файла. Викликається після функції **FindFirstFile** або **FindFirstFileEx**); **GetBinaryType** (визначення, чи являється файл виконуваним, і, якщо це так, то визначення його типу. Визначається, для виконання в якій системі він призначений: SCS_32BIT_BINARY – 32-бітна Windows програма, SCS_64BIT_BINARY – 64-бітна Windows програма, SCS_DOS_BINARY – MS-DOS програма, SCS_OS216_BINARY – 16-бітна OS/2 програма, SCS_PIF_BINARY – PIF файл, що виконується як MS-DOS програма, SCS_POSIX_BINARY – POSIX програма, SCS_WOW_BINARY – 16-бітна Windows програма); **GetCurrentDirectory** (отримання повного імені поточного каталогу для поточного процеса); **GetDiskFreeSpace** (отримання інформації про вільне місце, розмір кластера, розмір сектора для розділу, де розміщений вказаний кореневий каталог); **GetDriveType** (отримання інформації про тип диска: DRIVE_REMOVABLE – змінний носій, DRIVE_FIXED – жорсткий диск, DRIVE_CDROM – CD-ROM, DRIVE_RAMDISK – RAM-диск, DRIVE_REMOTE – мережевий диск, DRIVE_UNKNOWN – диск не можна визначити, DRIVE_NO_ROOT_DIR – диск не має кореневого запису, наприклад: не змонтований диск); **GetFileAttributes** (отримання атрибутів для вказаного файла або каталогу); **GetFileInformationByHandle** (отримання докладної інформації про файл за його дескриптором); **GetFileSize** (отримання розміру вказаного файла);

GetFileSizeEx (отримання розміру вказаного файла. Для зберігання розміру треба, щоб тип був більше ніж DWORD: значення може бути більше 4 ГБ); **GetFileType** (отримання типу вказаного файла: FILE_TYPE_DISK – блоковий (дисковий), FILE_TYPE_CHAR – символьний (LPT або консоль), FILE_TYPE_PIPE – канал, FILE_TYPE_UNKNOWN – невідомий тип); **GetFullPathName** (отримання повного шляху та імені вказаного файла); **GetLogicalDrives** (отримання бітової маски з інформацією про відображення логічних дисків); **GetLogicalDriveStrings** (отримання рядка з інформацією про відображення логічних дисків. Рядок подається у вигляді символьних позначень розділів, поділених між собою символами з кодом 0 (<0>), та символом з кодом 0 у кінці. Наприклад: «c:\<0>d:\<0><0>»); **GetLongPathName** (перетворення вказаного шляху в його довгу форму подання); **GetShortPathName** (перетворення вказаного шляху в його коротку форму подання); **SearchPath** (пошук вказаного файла за схемою: каталог, з якого була запущена на виконання програма, поточний каталог, системний каталог ОС, каталог ОС, каталоги, що перераховані в системній змінній PATH); **SetCurrentDirectory** (зміна поточного каталогу).

Параметри деяких функцій:

```
HANDLE CreateFile(
 LPCTSTR lpFileName,     // ім'я файла
 DWORD dwDesiredAccess,  // режим доступу
 DWORD dwShareMode,      // режим сумісного доступу
 LPSECURITY_ATTRIBUTES lpSecurityAttributes,//покажчик
      // на структуру атрибутів безпеки
 DWORD dwCreationDisposition, // режим створення
      // (OPEN_EXISTING – відкрити існуючий файл)
 DWORD dwFlagsAndAttributes,  // базові атрибути файла
 HANDLE hTemplateFile    // дескриптор шаблону файла
      // для встановлення додаткових атрибутів
);

HANDLE FindFirstFile(
```

```
 LPCTSTR lpFileName,                // ім'я файла
 LPWIN32_FIND_DATA lpFindFileData  // буфер з
      // інформацією про знайдений файл
);

BOOL FindNextFile(
 HANDLE hFindFile,                // дескриптор пошуку
 LPWIN32_FIND_DATA lpFindFileData // буфер з
      // інформацією про знайдений файл
);

BOOL GetBinaryType (
 LPCTSTR lpApplicationName,  // повне ім'я файла
 LPDWORD lpBinaryType        // бінарний тип файла
);

DWORD GetCurrentDirectory(
 DWORD nBufferLength,  // розмір буфера для імені
 LPTSTR lpBuffer       // буфер з іменем каталогу
);

UINT GetDriveType(
 LPCTSTR lpRootPathName   // кореневий каталог
);

DWORD GetFileAttributes(
 LPCTSTR lpFileName   // ім'я файла або каталогу
);

DWORD GetFileSize(
 HANDLE hFile,           // дескриптор файла
 LPDWORD lpFileSizeHigh  // розмір файла
);
```

```
DWORD GetFileType(
 HANDLE hFile   // дескриптор файла
);

DWORD GetFullPathName(
 LPCTSTR lpFileName,  // ім'я файла
 DWORD nBufferLength, // розмір буфера
 LPTSTR lpBuffer,   // буфер для шляху
 LPTSTR *lpFilePart // адреса початку імені файла
      // в буфері
);

DWORD GetLogicalDriveStrings(
 DWORD nBufferLength,  // розмір буфера
 LPTSTR lpBuffer       // буфер з символьними
      // позначеннями дисків
);

DWORD GetLongPathName(
 LPCTSTR lpszShortPath, //коротке ім'я файла зі шляхом
 LPTSTR lpszLongPath,   //буфер для довгого шляху
 DWORD cchBuffer        //розмір буфера
);

DWORD GetShortPathName(
 LPCTSTR lpszLongPath,  // шлях
 LPTSTR lpszShortPath,  // буфер для короткої форми
 DWORD cchBuffer        // розмір буфера
);

DWORD SearchPath(
 LPCTSTR lpPath,      // шлях пошуку
```

```
 LPCTSTR lpFileName,  // ім'я файла
 LPCTSTR lpExtension, // розширення файла
 DWORD nBufferLength, // розмір буфера
 LPTSTR lpBuffer,     // буфер із знайденим шляхом
      // і ім'ям файла
 LPTSTR *lpFilePart   // адреса початку імені файла
      // в буфері
);

BOOL SetCurrentDirectory(
 LPCTSTR lpPathName  // ім'я нового поточного каталогу
);
```

Якщо заданий розмір буфера в функціях **GetCurrentDirectory**, **GetFullPathName**, **GetLogicalDriveStrings**, **GetLongPathName**, **GetShortPathName** та **SearchPath** менше, ніж необхідно для збереження результату, повертається необхідний розмір. Необхідно викликати вказані функції повторно з отриманим розміром від першого виклику.

Приклад 10.1. Рекурсивний перегляд дерева каталогів на логічному диску та виведення їх змісту. У зв'язку з великою ємністю сучасних носіїв інформації рекомендується використовувати цей приклад для USB-накопичувача або для віртуального диска. Для створення віртуального диска перед запуском даної програми необхідно виконати команду **subst**. Синтаксис та параметри команди:

```
subst [<drive1>: [<drive2>:]<path>]
subst <drive1>: /d
```

де `<drive1>:` – віртуальний диск, якому необхідно призначити шлях; `[<drive2>:]<path>` – фізичний диск і шлях, який необхідно призначити віртуальному диску; `/d` – видалення віртуального диска.

Створимо, наприклад, віртуальний диск `k:` для реального шляху `d:\temp` з невеликою кількістю файлів та вкладених каталогів:

```
subst k: d:\temp\
```

Після виконання даної команди в системі з'явиться новий диск `k:` зі змістом шляху `d:\temp\` (в тому випадку, якщо такий диск існував до виконання команди, доречно обрати іншу літеру). Після завершення виконання програми створений віртуальний диск можна видалити:

```
subst k: /d
```

Дану програму не рекомендується запускати для реальних дисків у зв'язку з великою кількістю файлів на них та через це – дуже довгим виконанням програми.

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

void GetDIR(char* dir)
{
WIN32_FIND_DATA FileData;
HANDLE hSearch;
BOOL fFinished = FALSE;
int lngth = strlen(dir) + 4;
char* path = new char[lngth];
// виведення назви каталогу
printf("ENTERING TO DIR > %s\n", dir);
strcpy_s(path, lngth, dir);
strcat_s(path, lngth, "*.*");
hSearch = FindFirstFile(path, &FileData);
while (!fFinished)
```

```
{
      // якщо каталог
      if (FileData.dwFileAttributes ==
            FILE_ATTRIBUTE_DIRECTORY &&
            (FileData.cFileName[0] != '.'))
      {
            lngth = strlen(dir) +
                  strlen(FileData.cFileName)+2;
            char* newdir = new char[lngth];
            strcpy_s(newdir, lngth, dir);
            strcat_s(newdir, lngth,
                  FileData.cFileName);
            strcat_s(newdir, lngth, "/");
            GetDIR(newdir); //рекурсивний виклик
            delete newdir;
      }
      if ((FileData.cFileName[0] != '.') &&
            (FileData.dwFileAttributes !=
                  FILE_ATTRIBUTE_DIRECTORY))
                  // не каталог
                  // виведення імені файла
            printf("%s%s\n", dir, FileData.cFileName);
      if (!FindNextFile(hSearch, &FileData))
      {
            if (GetLastError() == ERROR_NO_MORE_FILES)
            // файлів більше нема - вихід з рекурсії
            {
                  fFinished = TRUE;
            }
      }
}
}
```

```
int main()
{
char StartDir[] = "k:/";
GetDIR(StartDir);
_getch();
return 0;
}
```

Недоліки наведеної програми – використання рекурсії та одна нитка виконання. Використання рекурсії в разі дуже великої кількості рекурсивних викликів може привести до переповнення стеку та аварійного завершення роботи. Одна нитка виконання – неефективне використання ресурсів операційної системи: при кожному зверненні до файлової системи буде відбуватись переривання виконання процесу – він буде переміщений в стан очікування, після завершення файлової операції – процес перейде до стану готовності і через деякий час продовжить своє виконання. Якщо використовувати кілька ниток – поки одна або кілька ниток знаходяться в стані очікування, інша нитка або декілька ниток будуть активними. Час виконання процесу буде менший. До того ж, якщо існує значна вкладеність каталогів, у програмі з переглядом на нитках буде задіяний менший обсяг пам'яті для збереження шляхів: при закінченні перегляду поточного каталогу в нитці одразу звільнюється пам'ять, а в рекурсії пам'ять буде звільнено лише після перегляду всіх підкаталогів у поточному каталозі.

Приклад 10.2. Нерекурсивний перегляд дерева каталогів на логічному диску та виведення їх змісту. Реалізація виконана на нитках.

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

void GetDIR(char* dir)
{
```

```
WIN32_FIND_DATA FileData;
HANDLE hSearch;
BOOL fFinished = FALSE;
char szBuf[512];
int lngth = strlen(dir) + 4;
char* path = new char[lngth];
// виведення назви каталогу
printf("ENTERING TO DIR > %s\n", dir);
strcpy_s(path, lngth, dir);
strcat_s(path, lngth, "*.*");
hSearch = FindFirstFile(path, &FileData);
while (!fFinished)
{
      // якщо каталог
      if (FileData.dwFileAttributes ==
            FILE_ATTRIBUTE_DIRECTORY &&
            (FileData.cFileName[0] != '.'))
      {
            lngth = strlen(dir) +
                  strlen(FileData.cFileName) + 2;
            char* newdir = new char[lngth];
            strcpy_s(newdir, lngth, dir);
            strcat_s(newdir, lngth,
                  FileData.cFileName);
            strcat_s(newdir, lngth, "/");
            CreateThread(NULL, 0,
                  (LPTHREAD_START_ROUTINE)GetDIR,
                  newdir, 0, NULL);
      }
      if ((FileData.cFileName[0] != '.') &&
            (FileData.dwFileAttributes !=
                  FILE_ATTRIBUTE_DIRECTORY))
                  // не каталог
```

```
                  // виведення імені файла
      printf("%s%s\n", dir, FileData.cFileName);
      if (!FindNextFile(hSearch, &FileData))
      {
            if (GetLastError() == ERROR_NO_MORE_FILES)
            // файлів більше нема - вихід з циклу
            {
                  fFinished = TRUE;
                  delete dir;
            }
      }
}
}

int main()
{
char* StartDir = new char[4];
strcpy_s(StartDir, 4, "k:/");
int cnt = 0;
CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)GetDIR,
      StartDir, 0, NULL);
_getch();
return 0;
}
```

На відміну від рекурсії при вході в кожний новий каталог створюється нова нитка для його перегляду, після завершення перегляду – нитка завершує роботу.

10.2. Контрольні питання

1. Чим відрізняються результати виконання прикладів програм перегляду дерева каталогів та виведення списку знайдених файлів у них на рекурсії (приклад 10.1) та на нитках (приклад 10.2)? Чому?

2. Яким чином визначити, що елемент каталогу є також каталогом?

3. На що посилаються каталоги з іменами крапка (.) та двокрапка (..)?

4. Що станеться, якщо видалити оператор **_getch()** перед оператором **return 0** у кінці текста прикладу 10.1 для перегляду каталогів на рекурсії? Чому?

5. Що станеться, якщо видалити оператор **_getch()** перед оператором **return 0** у кінці текста прикладу 10.2 для перегляду каталогів на нитках? Чому?

6. Чому в прикладі 10.2 перегляду каталогів на нитках звільнення пам'яті винесено в кінець роботи функції, а не одразу після оператора переходу до підкаталога, як у прикладі 10.1 з рекурсією?

10.3. Індивідуальні завдання

Розробити віконну прикладну програму в середовищі програмування Visual C, яка вирішує поставлену задачу у відповідності зі своїм варіантом. Реалізація перегляду дерева каталогів повинна бути або на рекурсії або на нитках. Для зберігання проміжних результатів можна використовувати функції WinAPI, які підтримують роботу з пам'яттю. Визначити кращій варіант реалізації перегляду дерева каталогів (на рекурсії або на нитках). Пояснити, чому не підходить варіант, від якого відмовились. Для тестування розробленої програми створити необхідну кількість віртуальних дисків з невеликим наповненням файлами та каталогами. Працювати тільки з віртуальними дисками. Вкладеність каталогів повинна бути не менше чотирьох, у кожному каталозі від п'яти до двадцяти файлів.

1) Визначити, яких виконуваних файлів на всьому віртуальному диску більше за все (прикладні програми DOS, WIN16, WIN32, WIN64, POSIX).

2) Визначити кількість файлів на жорсткому диску розміром більше 40 КБайтів, які мають розширення TXT.

3) Визначити, який логічний диск містить більшу кількість виконуваних файлів.

4) Визначити середню довжину шляху до файла на жорсткому диску.

5) Знайти всі каталоги з однаковими короткими іменами на жорсткому диску.

6) Виконати порівняння вмісту двох указаних каталогів, вивести всю інформацію про відмінності.

7) У відсотках до загального зайнятого простору визначити, який об'єм займають на диску всі файли, що мають розширення TXT.

8) Визначити всі файли на жорсткому диску, які мають однакові довгі та короткі імена.

9) Визначити на всіх логічних дисках кількість файлів, розмір яких є кратним розміру сектора, кластера.

10) Визначити кількість файлів на жорсткому диску, які відрізняються в імені тільки першою буквою.

11) Визначити, який логічний диск містить більшу кількість виконуваних файлів у форматі DOS.

12) Визначити наявність однакових повних імен каталогів на різних логічних дисках, підключених у системі.

13) Визначити кількість файлів, які мають імена, що співпадають з іменами виконуваних файлів на жорсткому диску.

14) Визначити кількість каталогів на жорсткому диску, які містять однакову кількість файлів.

15) Визначити кількість файлів на жорсткому диску, які мають більш ніж один розділовий символ крапки « **.** ».

16) Визначити загальне число секторів на жорсткому диску, які зайняті виконуваними файлами.

17) Визначити кількість системних файлів на жорсткому диску.

18) Визначити, які файли з однаковими іменами найбільш часто зустрічаються на жорсткому диску. Чи являються ці файли копіями одного і того ж файла?

19) Вивести перелік виконуваних файлів на жорсткому диску, які мають однакові імена, але являються різними типами прикладних програм (DOS, WIN16, WIN32, WIN64, POSIX).

20) Визначити усі можливі шляхи до каталогів, довге представлення яких співпадає з коротким.

11. СПОВІЩЕННЯ У ФАЙЛОВІЙ СИСТЕМІ В ОС WINDOWS

Мета: навчитись створювати прикладні програми в операційній системі Windows для виконання роботи з файловою системою на рівні використання функцій WinAPI.

11.1. Теоретичні відомості

Для відстеження змін у файловій системі в ОС Windows використовується система сповіщень. Щоб почалось відстеження змін у каталогах, необхідно створити дескриптор контролю змін і встановити початкові умови фільтрації оповіщень про зміни за допомогою функції **FindFirstChangeNotification.** Під час виклику цієї функції можна задати каталог для відстеження, необхідність аналізу підкаталогів і види змін, які мають відстежуватись. Можливі значення:

FILE_NOTIFY_CHANGE_FILE_NAME – зміна імені файла, формується при операціях перейменування, створення, видалення файлів;

FILE_NOTIFY_CHANGE_DIR_NAME – зміна імені каталогу, формується при операціях перейменування, створення, видалення каталогів;

FILE_NOTIFY_CHANGE_ATTRIBUTES – зміна атрибутів файлів або каталогів;

FILE_NOTIFY_CHANGE_SIZE – зміна розміру файлів при запису файла на диск ОС;

FILE_NOTIFY_CHANGE_LAST_WRITE – зміна мітки останнього запису для файлів або каталогів;

FILE_NOTIFY_CHANGE_SECURITY – зміна дескриптора захисту.

Значення, яке повертає ця функція, це дескриптор виявленого об’єкта контролю змін. Цей дескриптор використовується під час контролю змін за допомогою функцій **WaitForSingleObject** або **WaitForMultipleObjects**.

Параметри функції **FindFirstChangeNotification**:

```
HANDLE FindFirstChangeNotification(
 LPCTSTR lpPathName, //ім’я каталогу
 BOOL bWatchSubtree, //ознака моніторингу підкаталогів
```

```
 DWORD dwNotifyFilter//параметри моніторингу
);
```

У разі необхідності повторного очікування сповіщення треба використати функцію **FindNextChangeNotification**:

```
BOOL FindNextChangeNotification(
 HANDLE hChangeHandle   // дескриптор сповіщення
);
```

Для припинення відстеження змін для заданого дескриптора треба використати функцію **FindCloseChangeNotification**:

```
BOOL FindCloseChangeNotification(
 HANDLE hChangeHandle   // дескриптор сповіщення
);
```

Приклад 11.1. Виведення вмісту каталогу у випадку зміни імен файлів у ньому. Закоментована частина в даному прикладі не використовується, але якщо треба організувати повторне очікування сповіщення, то необхідно код розкоментувати.

```
#include <windows.h>
#include <stdio.h>
HANDLE hFind;
HANDLE g_hndl;
WIN32_FIND_DATA fd;
BOOL  bRet = TRUE;

int main()
{
// Ініціалізація дескриптора сповіщень
g_hndl = FindFirstChangeNotification("D:/TEMP",
```

```
      FALSE, FILE_NOTIFY_CHANGE_FILE_NAME);
if (g_hndl == INVALID_HANDLE_VALUE)
{
      printf("Invalid Handle\n");
}
// Очікування сповіщень
if (WaitForSingleObject(g_hndl, INFINITE) !=
      WAIT_FAILED)
{
      printf("File Name Changed!");
      // Виведення списку файлів у каталозі
      hFind = FindFirstFile("D:\\TEMP\\*.*", &fd);
      while (hFind != INVALID_HANDLE_VALUE && bRet)
      {
            if ((fd.dwFileAttributes &
                  FILE_ATTRIBUTE_DIRECTORY) == 0)
            {
                  printf("%s\n", fd.cFileName);
            }
            bRet = FindNextFile(hFind, &fd);
      }
      FindClose(hFind);
      // код для повторного очікування сповіщення
      // if(FindNextChangeNotification(g_hndl)==FALSE)
      // {
      //    printf("Find Next failed\n");
      // }
}
FindCloseChangeNotification(g_hndl);
return 0;
}
```

Для виконання моніторингу змін у файловій системі також використовується (більше того – вона рекомендується для використання) функція **ReadDirectoryChangesW** (тільки в системах Windows NT, 2000 та вище). Функція підтримує тільки UNICODE символи. У параметрі `dwNotifyFilter` задається тип події, яку необхідно відстежувати. Тип співпадає з типами функції **FindFirstChangeNotification** за винятком доданих двох типів подій:

FILE_NOTIFY_CHANGE_LAST_ACCESS – зміна часу останнього доступу до файла;

FILE_NOTIFY_CHANGE_CREATION – зміна часу створення файла.

```
BOOL ReadDirectoryChangesW(
 HANDLE hDirectory,  //дескриптор каталогу
 LPVOID lpBuffer,    //буфер результатів
 DWORD nBufferLength, //розмір буфера результатів
 BOOL bWatchSubtree, //ознака моніторингу підкаталогів
 DWORD dwNotifyFilter, //параметри моніторингу
 LPDWORD lpBytesReturned,//кількість повернутих байтів
 LPOVERLAPPED lpOverlapped, //буфер для асинхронного
      // виклику функції
 LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine
      //покажчик на функцію, що викликається після
      // завершення операції
);
```

Для отримання дескриптора каталогу необхідно скористатися функцією **CreateFile** з такими значеннями параметрів:

```
hDir = CreateFile(
 DirName,   // ім'я каталогу
 FILE_LIST_DIRECTORY,//режим доступу - перегляд файлів
 FILE_SHARE_READ|FILE_SHARE_WRITE,//режим розподілення
 NULL, // покажчик на дескриптор безпеки
```

```
 OPEN_EXISTING, // тільки відкриття існуючого каталогу
 FILE_FLAG_BACKUP_SEMANTICS, // атрибут резервного
      // копіювання
 NULL  // шаблон файла
);
```

Функція **ReadDirectoryChangesW** повертає результат у параметрі `lpBuffer`, який описаний таким чином:

```
typedef struct _FILE_NOTIFY_INFORMATION {
 DWORD NextEntryOffset; // зсув до наступної порції
                        // даних у буфері
 DWORD Action; // тип події, що відбулася
 DWORD FileNameLength; // довжина імені файла в байтах
 WCHAR FileName[1]; // ім'я файла UNICODE
} FILE_NOTIFY_INFORMATION, *PFILE_NOTIFY_INFORMATION;
```

Розмір буфера повинен бути достатнім для збереження декількох порцій інформації, але не більше ніж 64 КБайти. Параметр `Action` означає, яка саме подія відбулася:

FILE_ACTION_ADDED – файл було додано в каталог,

FILE_ACTION_REMOVED – файл було видалено з каталогу,

FILE_ACTION_MODIFIED – файл було змінено (також фіксуються окремо зміни атрибутів та часу),

FILE_ACTION_RENAMED_OLD_NAME – файл було перейменовано з вказаного в параметрі `FileName` імені,

FILE_ACTION_RENAMED_NEW_NAME – файл було перейменовано в указане в параметрі `FileName` ім'я.

Недоліками використання функції **ReadDirectoryChangesW** є: кінцевий розмір буфера (64 КБайти), куди розміщуються дані про зміни – черга подій може переповниться і частина подій буде втрачена; події переміщення файлів посилаються раніше, ніж ці зміни стають видимими у файловій системі – може знадобиться організація затримки для їх обробки.

Приклад 11.2. Відстеження одного перейменування файла в каталозі. Як тільки файл перейменовано – програма завершує свою роботу. Для роботи програми необхідно встановити підтримку в програмі символів UNICODE. Для цього визначаємо константи UNICODE та _UNICODE. Для коректного відображення символів кирилиці необхідно для консолі встановити підтримку кодової сторінки 866. Також замість головної функції **main** необхідно використовувати функцію **wmain**. Для роботи з текстовими константами потрібно використовувати макрос TEXT або L.

```
#define _WIN32_WINNT 0x0500
#define _UNICODE
#define UNICODE
#include <windows.h>
#include <conio.h>
#include <stdio.h>
#include <locale.h>

int wmain(int argc, char* argv[])
{
setlocale(LC_CTYPE, "rus_rus.866");
DWORD nBufSize = 32 * 1024;
FILE_NOTIFY_INFORMATION* pBuffer =
      (FILE_NOTIFY_INFORMATION*)calloc(1, nBufSize);
FILE_NOTIFY_INFORMATION* pBufferCurrent;
WCHAR fname[100];

HANDLE hDir = CreateFile(L"d:/temp",
      FILE_LIST_DIRECTORY,
      FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
      OPEN_EXISTING,FILE_FLAG_BACKUP_SEMANTICS,NULL);
DWORD BytesReturned;
while (true)
```

```
{
      int ret = ReadDirectoryChangesW(hDir, pBuffer,
            nBufSize, TRUE,
            FILE_NOTIFY_CHANGE_FILE_NAME,
            &BytesReturned, NULL, NULL);
      pBufferCurrent = pBuffer;
      while (pBufferCurrent)
      {
            wcsncpy_s(fname,
                  pBufferCurrent->FileName,
                  pBufferCurrent->FileNameLength / 2);
            fname[pBufferCurrent->FileNameLength/2]=0;
            switch (pBufferCurrent->Action)
            {
            case FILE_ACTION_RENAMED_OLD_NAME:
                  wprintf(L"RENAMED FROM: %s\n",
                        fname);
                  break;
            case FILE_ACTION_RENAMED_NEW_NAME:
                  wprintf(L"RENAMED TO : % s\n",
                        fname);
                  CloseHandle(hDir);
                  _getch();
                  return 0;
                  break;
            }
            // пошук наступної порції даних у буфері
            if (pBufferCurrent->NextEntryOffset)
                  pBufferCurrent =
                  (FILE_NOTIFY_INFORMATION*)
                  (((BYTE*)pBufferCurrent) +
                        pBufferCurrent->
                        NextEntryOffset);
```

```
            else
                  pBufferCurrent = NULL;
      }
}
}
```

11.2. Контрольні питання

1. У чому полягає основна відмінність результатів виклику функцій **FindFirstChangeNotification** та **ReadDirectoryChangesW**?

2. Які істотні недоліки існують при використанні функцій **FindFirstChangeNotification** / **FindNextChangeNotification**?

3. Яким чином можна позбавитись недоліків роботи з функцією **ReadDirectoryChangesW**?

4. Чи можна викликом однієї функції сповіщення відстежувати одразу кілька різних подій?

5. Яким чином можна відстежувати зміни в каталозі без використання функцій роботи зі сповіщеннями?

11.3. Індивідуальні завдання

Розробити програму в середовищі програмування Visual C, яка відстежує зміни у файловій системі та виводить повідомлення, у відповідності з заданим варіантом. Програма повинна відстежувати зміни постійно в циклі. Продемонструвати працездатність програми з кількістю подій не менше п’яти. Завершення програми виконувати по команді з клавіатури. Інтерфейс програми – за бажанням: командний рядок або вікно. Для зберігання проміжних результатів можна використати засоби WinAPI для роботи з пам’яттю.

1) Ім’я каталогу вводиться з клавіатури. Підкаталоги не аналізуються. Виводити імена змінюваних файлів.

2) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги аналізуються. При зміні часу останнього записування у файл – виводити ім’я файла і час записування.

3) Ім’я каталогу вводиться з клавіатури. Підкаталоги аналізуються. При зміні розміру файла виводити старе і нове значення.

4) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги не аналізуються. При зміні імені файла виводити час зміни та старе і нове ім’я файла.

5) Ім’я каталогу вводиться з клавіатури. Підкаталоги не аналізуються. Виводити імена видалених файлів та час їх видалення.

6) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги аналізуються. При зміні імен каталогів виводити старі і нові їх імена та час перейменування.

7) Ім’я каталогу вводиться з клавіатури. Підкаталоги аналізуються. Виводити імена створюваних файлів та час їх створення.

8) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги не аналізуються. Виводити імена файлів, розміри яких були збільшені.

9) Ім’я каталогу вводиться з клавіатури. Підкаталоги не аналізуються. Виводити час записування інформації у файли.

10) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги аналізуються. Виводити імена каталогів, в яких відбуваються будь-які зміни з визначенням, які саме зміни відбуваються.

11) Ім’я каталогу вводиться з клавіатури. Підкаталоги аналізуються. Відстежувати появу файлів з однаковими іменами в різних підкаталогах.

12) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги не аналізуються. Відстежувати появу файлів з нульовою довжиною.

13) Ім’я каталогу вводиться з клавіатури. Підкаталоги не аналізуються. Виводити журнал створення та видалення файлів у каталозі (ім’я, час).

14) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги аналізуються. Виводити журнал створення підкаталогів у каталозі (ім’я, час).

15) Ім’я каталогу вводиться з клавіатури. Підкаталоги аналізуються. Виводити імена файлів, які видаляються, і час їх видалення.

16) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги не аналізуються. Виводити імена файлів, розміри яких були зменшені.

17) Ім’я каталогу вводиться з клавіатури. Підкаталоги не аналізуються. Вести статистику зміни сумарного об’єму файлів у каталозі.

18) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги аналізуються. Визначити появу каталогів з однаковими короткими іменами.

19) Ім’я каталогу вводиться з клавіатури. Підкаталоги аналізуються. Вести статистику по зміні кількості підкаталогів.

20) Ім’я каталогу передається як параметр під час запуску програми. Підкаталоги не аналізуються. Вести статистику по зміні кількості файлів.

12. РОБОТА З РЕЄСТРОМ В ОС WINDOWS

Мета: вивчити основні принципи роботи з системним реєстром в ОС Windows.

12.1. Теоретичні відомості

У системному реєстрі зберігається інформація про конфігурацію системи Windows (в ОС Windows 3.1 використовувались INI файли, в ОС UNIX аналогічна інформація зберігається в каталозі /etc). Структура системного реєстру подібна структурі каталогів: еквівалентом каталогу є ключ (key), а файла – значення (value). Ключ містить підключі (subkeys) та значення. У цілому системний реєстр має деревоподібну структуру. Основними гілками системного реєстру являються 6 розділів:

HKEY_CLASSES_ROOT – містить визначення типів документів, зв'язків з файлами та інтерфейсу командного процесора;

HKEY_CURRENT_USER – містить параметри налаштування поточного користувача;

HKEY_LOCAL_MACHINE – зберігає апаратні конфігурації, мережеві протоколи та класи програмного забезпечення;

HKEY_USERS – використовується для зберігання вибраних користувачами глобальних параметрів (колір, звук і т.і.), а також містить параметри налаштування робочого столу;

HKEY_CURRENT_CONFIG – встановлює зв'язок з ключем відображення, який входить до складу вибраної конфігурації config із **HKEY_LOCAL_MACHINE**;

HKEY_PERFORMANCE_DATA (Windows 2000 і вище) або **HKEY_DYN_DATA** (Windows 9x) – зберігає дані про кожний апаратний компонент системи та дані про продуктивність системи.

У значеннях реєстру зберігаються дані наступних основних типів:

- REG_SZ – рядковий (Unicode або ANSI), закінчується символом з кодом 0;
- REG_BINARY – двійковий в будь-якій формі;
- REG_DWORD – подвійне слово (32 розряди);

▪ REG_LINK – дозволяє одному розділу реєстру вказувати на інший (наприклад, **HKEY_CURRENT_USER** посилається на підрозділ поточного користувача з **HKEY_USERS**, **HKEY_CURRENT_CONFIG** посилається на **HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ HardwareProfiles \ Current**, **HKEY_CLASSES_ROOT** посилається на **HKEY_LOCAL_MACHINE \ SOFTWARE \ Classes**);

▪ REG_DWORD_LITTLE_ENDIAN – подвійне слово в малобайтовому форматі (наприклад, 0x12345678 зберігається в пам'яті з найменшого байта як 0x78 0x56 0x34 0x12);

▪ REG_DWORD_BIG_ENDIAN – багатобайтове значення зберігається в пам'яті від самого більшого байта (наприклад, 0x12345678 зберігається як 0x12 0x34 0x56 0x78);

▪ REG_EXPAND_SZ – рядок, що завершується нульовим значенням, містить нерозширені посилання на змінні середовища (наприклад, %PATH%);

▪ REG_MULTI_SZ – послідовність рядків, що закінчуються значенням NULL, що в свою чергу також завершується порожнім рядком (\0 – один символ з кодом 0) (наприклад, String1\0String2\0String3\0LastString\0\0);

▪ REG_NONE – нема певного типу значення;

▪ REG_QWORD – 64-розрядне число;

▪ REG_QWORD_LITTLE_ENDIAN – 64-розрядне число в малобайтовому форматі;

▪ REG_RESOURCE_LIST – список ресурсів драйвера пристрою (повний опис наведено за посиланням https://learn.microsoft.com/uk-ua/windows/win32/wmisdk/describing-a-resource-for-the-registry).

Фізично реєстр на диску зберігається в декількох файлах. Кожен з файлів називається кущем (hive). Перелік кущів, з яких сформовано реєстр, можна знайти в підрозділі реєстру **HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ Hivelist**, в якому вказані ключі реєстру та файли на диску, що їх містять. Більшість кущів знаходяться в каталозі **\%SystemRoot%\System32\Config** (файли DEFAULT, SAM, SOFTWARE, SECURITY і SYSTEM), де **%SystemRoot%** – каталог, в якому встановлено операційну систему, та декілька – у каталогах користувачів **\Documents**

and Settings (**%USERPROFILE%\NTUSER.DAT**, де **%USERPROFILE%** – каталог поточного користувача).

Будь який кущ можна завантажити в реєстр (тільки в ключі **HKEY_LOCAL_MACHINE** та **HKEY_USERS**) як нові підключі за допомогою утілити роботи з реєстром **Regedit** (пункт меню – *Load Hive*). Максимальний розмір реєстру обмежується 376 МБайтів. Крім основних кущів реєстру існують також реєстраційні кущі (log hives) (файли з розширенням log, що знаходяться в каталозі **\%SystemRoot%\System32\Config**). У реєстраційних кущах знаходяться бітові масиви, кожен біт яких відповідає за один сектор куща (512 байтів) та вказує на його модифікацію. При змінах в реєстрі у відповідному реєстраційному кущі встановлюються ознаки модифікації, і через п’ять секунд після модифікації активується процедура синхронізації – збереження відповідних частин реєстру на диску (але не раніше ніж через п’ять секунд після останньої синхронізації).

Ще одна особливість організації реєстру: через ключ **HKEY_PERFORMANCE_DATA** можна отримати доступ до чисельних лічильників продуктивності системи. Через утиліту **Regedit** доступ до лічильників неможливий, тільки за допомогою функцій WinAPI роботи з реєстром. Деякі лічильники можна отримати за допомогою утиліти **Task Manager**. Перелік лічильників вказано в ключі реєстру **HKEY_LOCAL_MACHINE \ SOFTWARE \ Microsoft \ Windows NT \ CurrentVersion \ Perflib \ langid**, де **langid** – ідентифікатор мови (наприклад, 009 – англійська): у параметрі *Counter* вказані назви значень лічильників (мають тільки парні номери), у параметрі *Help* наведена коротка допомога: призначення кожного лічильника (мають тільки непарні номери). Відповідність значень та допомоги наступна: для кожного парного номеру значення, наступний непарний номер допомоги задає його опис. Значення *Counter* та *Help* мають тип MULTI_SZ. Для їх зчитування необхідно заготовити буфер достатнього розміру: кількість лічильників сягає кількох тисяч.

Для роботи з реєстром використовуються такі функції (перераховані найбільш часто використовувані): **RegCloseKey** (закриття відкритого ключа системного реєстру), **RegCreateKeyEx** (створення нового підключа),

RegDeleteKey (видалення ключа із системного реєстру), **RegDeleteValue** (видалення значення із системного реєстру), **RegEnumKeyEx** (перелік всіх підключів даного ключа), **RegEnumValue** (перелік всіх значень даного ключа), **RegFlushKey** (примусовий запис змін, які стались у системному реєстрі), **RegNotifyChangeKeyValue** (відстежування моменту зміни ключа або значення в системному реєстрі: REG_NOTIFY_CHANGE_ATTRIBUTES – зміна будь яких атрибутів у ключі та підключах, REG_NOTIFY_CHANGE_LAST_SET – зміна після операції останнього запису, REG_NOTIFY_CHANGE_NAME – зміна імен ключів, їх створення або видалення, REG_NOTIFY_CHANGE_SECURITY – зміна в дескрипторі захисту), **RegOpenKeyEx** (відкриття існуючого ключа системного реєстру), **RegQueryInfoKey** (отримання інформації про ключ), **RegQueryValueEx** (отримання значення ключа), **RegSetValueEx** (привласнення ключу значення).

Параметри виклику деяких з вказаних функцій:

```
LONG RegCreateKeyEx(
 HKEY hKey,        // ключ, в якому створюється підключ
 LPCTSTR lpSubKey, // ім'я підключа
 DWORD Reserved,   // зарезервовано
 LPTSTR lpClass,   // ім'я класу ключа
 DWORD dwOptions,  // параметри зберігання ключа
      //(REG_OPTION_VOLATILE – не зберігати ключ)
 REGSAM samDesired, // режим доступу до ключа
 LPSECURITY_ATTRIBUTES lpSecurityAttributes,
      // атрибути захисту ключа
 PHKEY phkResult,   // дескриптор нового підключа
 LPDWORD lpdwDisposition  // результат створення
);

LONG RegDeleteKey(
 HKEY hKey,        // дескриптор відкритого ключа
 LPCTSTR lpSubKey // ім'я підключа
```

```
);

LONG RegDeleteValue(
 HKEY hKey,            // дескриптор відкритого ключа
 LPCTSTR lpValueName   // назва значення
);

LONG RegEnumKeyEx(
 HKEY hKey,            // дескриптор відкритого ключа
 DWORD dwIndex,        // індекс підключа (перший – 0)
 LPTSTR lpName,        // буфер з ім'ям підключа
 LPDWORD lpcName,      // розмір буфера lpName
 LPDWORD lpReserved,   // зарезервовано
 LPTSTR lpClass,       // ім'я класу ключа
 LPDWORD lpcClass,     // розмір буфера lpClass
 PFILETIME lpftLastWriteTime // дата і час останнього
      // запису в підключ
);

LONG RegEnumValue(
 HKEY hKey,            // дескриптор відкритого ключа
 DWORD dwIndex,        // індекс значення (початок – 0)
 LPTSTR lpValueName,   // буфер для імені значення
 LPDWORD lpcValueName,// розмір буфера lpValueName
 LPDWORD lpReserved,   // зарезервовано
 LPDWORD lpType,       // буфер для типу значення
 LPBYTE lpData,        // зміст значення
 LPDWORD lpcbData      // розмір буфера lpData
);

LONG RegNotifyChangeKeyValue(
 HKEY hKey,             // дескриптор відкритого ключа
 BOOL bWatchSubtree,    // аналіз підключів (так/ні)
```

```
 DWORD dwNotifyFilter, // ознаки сповіщення
 HANDLE hEvent,       // дескриптор події про зміни
 BOOL fAsynchronous  // ознака асинхронного виклику
);

LONG RegOpenKeyEx(
 HKEY hKey,          // дескриптор відкритого ключа
 LPCTSTR lpSubKey,  // ім'я підключа
 DWORD ulOptions,   // зарезервовано
 REGSAM samDesired, // маска прав доступу
 PHKEY phkResult // дескриптор ключа, що відкривається
);

LONG RegQueryInfoKey(
 HKEY hKey,              // дескриптор відкритого ключа
 LPTSTR lpClass,         // ім'я класу ключа
 LPDWORD lpcClass,       // розмір буфера lpClass
 LPDWORD lpReserved,     // зарезервовано
 LPDWORD lpcSubKeys,      // кількість підключів
 LPDWORD lpcMaxSubKeyLen, // найдовше ім'я підключа
 LPDWORD lpcMaxClassLen,  // найдовше ім'я класу
 LPDWORD lpcValues,       // кількість значень
 LPDWORD lpcMaxValueNameLen, // найдовше ім'я значення
 LPDWORD lpcMaxValueLen,     // найдовші дані значення
 LPDWORD lpcbSecurityDescriptor, // кількість байтів
      // дескриптора захисту
 PFILETIME lpftLastWriteTime  // час останнього запису
      // (в 100 нс інтервалах від 1 січня 1601 року)
);

LONG RegQueryValueEx(
 HKEY hKey,             // дескриптор відкритого ключа
 LPCTSTR lpValueName,  // ім'я значення
```

```
 LPDWORD lpReserved,   // зарезервовано
 LPDWORD lpType,       // буфер для типу значення
 LPBYTE lpData,        // зміст значення
 LPDWORD lpcbData      // розмір буфера lpData
);

LONG RegSetValueEx(
 HKEY hKey,           // дескриптор відкритого ключа
 LPCTSTR lpValueName, // ім'я значення
 DWORD Reserved,      // зарезервовано
 DWORD dwType,        // тип значення
 CONST BYTE *lpData,  // зміст значення
 DWORD cbData         // розмір буфера lpData
);
```

Приклад 12.1. Програма при відсутності ключа в реєстрі створює його, а при наявності – виводить всі значення ключа та видаляє його.

```
#include <windows.h>
#include <stdio.h>
HKEY hk;
CHAR szBuf[80];
DWORD dwData, dwDsp, dwValues, dwMaxValueNameLen,
      dwMaxValueData, dwNameLen, dwValueData, dwIndex,
      dwType;
LONG lReturn;
LPTSTR lpszVN1, lpszVN2;

int main()
{
if (RegCreateKeyEx(HKEY_CURRENT_USER,
      (LPSTR)"Software\\MyLabSPO", 0, (LPSTR)"",
      REG_OPTION_NON_VOLATILE, // зберегти ключ
```

```
      KEY_ALL_ACCESS, NULL, &hk, &dwDsp))
{
      printf("Key Creation Error\n"); return(0);
}
if (dwDsp == REG_OPENED_EXISTING_KEY)
{
      printf("Key Found\n");
      // закриття дескриптора ключа
      RegCloseKey(hk);
      // відкриття ключа (приведено для прикладу) –
      // у даному випадку достатньо проаналізувати
      // dwDsp==REG_OPENED_EXISTING_KEY
      if (RegOpenKeyEx(HKEY_CURRENT_USER,
            "Software\\MyLabSPO", 0,
            KEY_ALL_ACCESS, &hk))
      {
            printf("Key Open Error\n"); return(0);
      }
      // Визначення необхідних параметрів для читання
      RegQueryInfoKey(hk, NULL, NULL, NULL, NULL,
            NULL, NULL,
            &dwValues, // Кількість значень у ключі
            &dwMaxValueNameLen, // максимальна
                  // довжина імені значення
            &dwMaxValueData, // максимальна довжина
                  // значення
            NULL, NULL);
      printf("Values:%d,Max Value Name Length:%d,
            Max Data Length : % d\n", dwValues,
            dwMaxValueNameLen, dwMaxValueData);
      // так як рядки завершуються 0 - додамо 1 байт
      dwMaxValueNameLen++;
      dwMaxValueData++;
```

```
// виділення пам'яті для збереження рядків
lpszVN1 = (LPTSTR)HeapAlloc(GetProcessHeap(),
      HEAP_ZERO_MEMORY, dwMaxValueNameLen);
lpszVN2 = (LPTSTR)HeapAlloc(GetProcessHeap(),
      HEAP_ZERO_MEMORY, dwMaxValueData);
// читання значень ключа
do
{
      dwNameLen = dwMaxValueNameLen;
      dwValueData = dwMaxValueData;
      lReturn = RegEnumValue(hk, dwIndex,
            lpszVN1, &dwNameLen, 0, &dwType,
            (unsigned char*)lpszVN2,
            &dwValueData);
      if (lReturn != ERROR_NO_MORE_ITEMS)
      {
            printf("%s=%s\n", lpszVN1, lpszVN2);
            dwIndex++;
      }
} while (lReturn != ERROR_NO_MORE_ITEMS);
// звільняємо пам'ять
if(lpszVN1)HeapFree(GetProcessHeap(),0,lpszVN1);
if(lpszVN2)HeapFree(GetProcessHeap(),0,lpszVN2);
// видаляємо значення ключа
if (RegDeleteValue(hk, "File Name"))
      printf("Value 1 Delete Error\n");
if (RegDeleteValue(hk, "Application Run"))
      printf("Value 2 Delete Error\n");
// видаляємо ключ
if (RegDeleteKey(hk, ""))
{
      printf("Key Delete Error\n");
}
```

```
      else
      {
            printf("Key Deleted Successfull\n");
      }
}
else
{
      printf("Key Not Found\nCreated Successfull\n");
      strcpy_s(szBuf, "My Program");
      // створення значення ключа
      if (RegSetValueEx(hk, "File Name", 0, REG_SZ,
            (LPBYTE)szBuf, strlen(szBuf) + 1))
      {
            printf("Value 1 Set Error\n");
            RegCloseKey(hk); return(0);
      }
      dwData = 1;
      if (RegSetValueEx(hk, "Application Run", 0,
            REG_DWORD, (LPBYTE)&dwData,
            sizeof(DWORD)))
      {
            printf("Value 2 Set Error\n");
            RegCloseKey(hk); return(0);
      }
}
RegCloseKey(hk);
return 0;
}
```

12.2. Контрольні питання

1. Які переваги використання реєстру для налаштувань програм по зрівнянню з файлами налаштувань INI?

2. Які гілки реєстру будуть відрізнятися між собою для різних користувачів на одній робочій станції?

3. За допомогою якої утиліти можна отримати доступ до реєстру?

4. Необхідно записати дані в реєстр та одразу їх прочитати. Які при цьому можуть бути проблеми? Чому?

5. Чому помилкові зміни в реєстрі можуть призвести до помилок у роботі ОС?

12.3. Індивідуальні завдання

Попередження: Ніколи не змінюйте вміст реєстру, якщо не знаєте його призначення!!! Некоректна зміна даних може привести до збоїв у роботі або повної непрацездатності ОС Windows!!!

Примітка: У даній роботі необхідно використати тільки СВОЇ унікальні ключі в реєстрі, причому тільки в розділі HKEY_CURRENT_USER. Зміна або видалення будь-яких існуючих ключів ЗАБОРОНЕНІ!!!

Розроблена програма повинна мати графічний інтерфейс. Всі необхідні вікна програми повинні бути розроблені за допомогою функцій WIN API, взаємодія між ними – за допомогою повідомлень (детальна інформація вказана в темі «Віконні програми в ОС Windows»).

1) Написати програму, кількість запусків якої на даній машині обмежується п’ятьма. Під час запуску програми необхідно видавати користувачеві інформацію про кількість запусків, які ще не відбулись.

2) Написати програму, яка видає дату та час свого попереднього запуску.

3) Написати програму, яка інформує користувача про факт наявності змін у реєстрі для вказаної гілки.

4) Написати програму, яка у випадку зміни значень певного ключа повертає їх у початковий стан.

5) Написати програму, яка сигналізує про своє некоректне завершення під час попереднього сеансу роботи.

6) Написати програму, яка зберігає налаштування положення та розмірів вікна, які використовує під час свого наступного запуску.

7) Написати програму, яка фіксує загальний час (за всі сеанси) своєї роботи.

8) Написати програму, яка запобігає видаленню певних ключів у реєстрі.

9) Написати програму, яка виводить інформацію про самий довгий сеанс роботи з нею.

10) Написати програму, повторні запуски якої виконуються тільки за паролем, заданим під час першого її запуску.

11) Написати програму, яка виконує пошук вказаних ключів у реєстрі та виводить всі їх значення у відповідності з їх типами.

12) Написати програму, яка зберігає п’ять останніх введених рядків у полі введення та видає їх під час кожного наступного запуску.

13) Написати програму, яка виводить час свого попереднього завершення з точністю до однієї хвилини.

14) Написати програму, яка виводить список програм, що знаходяться в списку автозапуску системи (ключі в гілках **HKEY_LOCAL_MACHINE** та **HKEY_CURRENT_USER Software \ Microsoft \ Windows \ CurrentVersion \ Run**) і перевіряє фізичну присутність таких файлів на диску.

15) Написати програму, яка відстежує зміну дати на комп’ютері в сторону зменшення.

16) Написати програму, яка під час старту виконує запуск вказаної в своєму ключі в реєстрі прикладної програми. Прикладна програма повинна задаватись цією ж програмою.

17) Написати програму, тривалість роботи якої визначається залежно від кількості її запусків.

18) Написати програму, яка фіксує та виводить на екран усі каталоги, із яких вона запускалась.

13. МІНІМІЗАЦІЯ ПРОГРАМ У СЕРЕДОВИЩІ MICROSOFT VISUAL STUDIO

Мета: виконати мінімізацію програм, розроблених у середовищі програмування Microsoft Visual Studio.

13.1. Теоретичні відомості

Налаштування системи програмування за замовчанням та використання стандартних бібліотечних функцій не завжди дозволяє отримати програми, розмір яких був би відповідним до тих дій, які вони виконують. Виконання операцій по мінімізації дозволяє за рахунок деяких погіршень характеристик цих програм (наприклад, зменшення швидкості завантаження програми в пам’ять та погіршення сумісності та переносимості) досягти зменшення обсягу в десятки або в сотні разів. При цьому, функціонування програм залишається без змін. Для оптимізації програм за розміром необхідно виконати як модифікацію тексту програми, так і встановити нові, більш ефективні налаштування середовища програмування.

Налаштування середовища програмування.

Всі кроки по змінам налаштувань необхідно робити поступово, залишаючи ті, які дійсно дають виграш за розміром, та відкидаючи ті, що в даному випадку не приводять до покращення (або навіть погіршують стан).

Для оптимізації розміру програм необхідно скористатися змінами в наступних налаштуваннях:

1) встановити режим проєкта «Release»,

2) задати налаштування компілятора (compiler),

3) задати налаштування компонувальника (linker),

4) замінити частину кода згідно зроблених налаштувань.

В якості демонстрації можливостей по мінімізації програм використовуються консольна програма з прикладу 1.4 теми «Процеси, нитки та волокна в ОС Windows» та віконна програма з прикладу 3.1, розробка якої велась у темі «Віконні програми в ОС Windows».

Необхідність встановлення режиму створення проєкта «Release» полягає в наступному. У версії «Debug» компілятором у код програми додається додаткова інформація, що дозволяє виконувати дії по налагодженню програми. Відповідно, швидкість виконання програми за рахунок майже постійних перевірок на наявність помилок буде меншою, чим у режимі «Release». Але для коректної роботи програми в режимі «Release» необхідно виконувати додаткові дії по зрівнянню з режимом «Debug» (в режимі «Debug» ці дії автоматично виконує додатковий код): обнуління структур, виділення пам'яті змінним, перевірка значень функцій на помилки і т.і.

Для програми, взятої в якості прикладу, розмір файла для виконання в режимі «Debug» складає 40 кБайтів (для різних версій середовищ програмування ці дані можуть відрізнятися).

Як було вказано в темі «Процеси, нитки та волокна в ОС Windows» – для всіх проєктів, які розроблювались, було застосовано налаштування компілятору

▪ Проєкт → Властивості → Властивості Конфігурації → Додатково → Набір Символів → Не Задано

Це дозволило відключити використання кодування символів у форматі Wide Char. У цьому форматі кожен символ кодується двома байтами, відключення такого кодування примушує використовувати кодування Ansi Char (один байт), відповідно тільки це налаштування дозволило вже зменшити розмір символьних даних у два рази.

Для обробки кожного з типів кодування існує два види функцій WinAPI (що закінчуються символом A або W). Наприклад: **CreateFileA** і **CreateFileW**: перша приймає рядкові параметри як Ansi Char кодування, а друга – як Wide Char кодування. Виклик тієї або іншої функції залежить від налаштування компілятора щодо набору символів, тому, коли в коді зустрічається виклик **CreateFile**, викликається або **CreateFileA** або **CreateFileW**. Сам виклик **CreateFile** є лише define-токеном:

```
#ifdef UNICODE
#define CreateFile  CreateFileW
#else
```

```
#define CreateFile  CreateFileA
#endif // !UNICODE
```

Для явної вказівки, яку саме функцію викликати – краще одразу в коді писати саме Ansi Char варіант, якщо такий є.

Також, для того щоб прибрати частину зайвого коду, який автоматично додається в проєкт, можна скористатись налаштуванням так само, як це було вказано в темі «Процеси, нитки та волокна в ОС Windows»:

▪ Проєкт → Властивості → Властивості Конфігурації → C/C++ → Попередньо Відкомпільовані Заголовки → Не Використовувати Попередньо Відкомпільовані Заголовки

У режимі «Release» розмір виконуваного файла прикладу з відключенням двобайтового набору символів та попередньо відкомпільованих заголовків вже складає 9216 байтів.

Наступна частина змін пов'язана з налаштуваннями оптимізації. Для зменшення розміру виконуваного файла необхідно встановити такі параметри в гілці налаштувань «**Проєкт → Властивості → Властивості Конфігурації → C/C++ → Оптимізація**»:

▪ Оптимізація → Максимальна Оптимізація (Пріоритет Розміру) (/O1)

▪ Розгортати Функції, Що Підставляються → За Замовчанням

▪ Підключити Функції, Що Підставляються → Ні

▪ Віддавати Перевагу Розміру Або Швидкості Коду → Віддавати Перевагу Стислості Коду (/Os)

▪ Оптимізація Всієї Програми → Так (/GL)

Оптимізація може вплинути на:

▪ локальні змінні (можуть бути видалені оптимізатором або переміщені до місць, недоступних для налагодження);

▪ код усередині функцій (змінюється розташування при злитті оптимізатором блоків коду);

▪ імена функцій в кадрах стека викликів (можуть виявитися некоректними при злитті оптимізатором двох функцій);

▪ фрейми в стеку викликів майже завжди вірні, але за наявності символів для всіх кадрів. Фрейми в стеку викликів можуть бути некоректними при пошкодженні стека, при наявності функцій на асемблері, або у фреймах операційної системи за відсутності відповідних символів у стеку викликів;

▪ глобальні та статичні змінні завжди відображаються правильно. Це також стосується структурних типів. Якщо існує покажчик на структуру та його значення вірне, кожна змінна-елемент структури також має правильне значення.

У зв'язку з цими обмеженнями слід по можливості налагоджувати неоптимізовані версії програми. За замовчуванням оптимізація вимкнена в конфігурації «Debug» програми C++ та включена до конфігурації «Release». Проте, помилка може з'явитися і в оптимізованій версії програми.

Оптимізація всієї програми дозволяє компілятору виконувати оптимізацію з інформацією про всі модулі програми. Без оптимізації всієї програми оптимізація виконується на основі кожного модуля (компіляції). Маючи інформацію про всі модулі, компілятор може:

▪ оптимізувати використання регістрів за межами функцій;

▪ краще відстежувати зміни глобальних даних, дозволяючи зменшити кількість їх завантажень і збережень;

▪ відстежувати можливий набір елементів, змінених розіменуванням покажчиків, зменшуючи необхідну кількість їх завантажень і збережень;

▪ вбудовувати функції в поточні модулі, навіть якщо функції визначені в інших модулях.

Наступна частина оптимізації – подальша робота з видалення коду, що додається автоматично під час компіляції. Корегування відбувається в гілці налаштувань «**Проєкт → Властивості → Властивості Конфігурації → C/C++ → Створення Коду**»:

▪ **Підключити Об'єднання Рядків → Так (/GF)**

▪ **Підключити C++ Виключення → Ні**

▪ **Бібліотека Часу Виконання → Багатониткова (/MT)**

▪ **Перевірка Безпеки → Відключити Перевірку Безпеки (/GS-)**

Об'єднання рядків дозволяє компілятору створити одну копію ідентичних рядків в образі програми та в пам'яті під час виконання. Такі рядки будуть доступні тільки для читання. При спробі змінити рядки виникне помилка. Об'єднання рядків дозволяє тому, що передбачалося як кілька покажчиків на кілька буферів, бути кількома покажчиками на один буфер. Наприклад, у наступному фрагменті об'єднання рядків приведе до того, що покажчики `s` та `t` будуть вказувати на одну й ту саму пам'ять:

```
char *s = "Це символьний буфер";
char *t = "Це символьний буфер";
```

Обробка виняткових ситуацій використовується в C++ за допомогою конструкції try-catch, якщо не використовувати таку обробку, та не використовувати засоби C++, то можна від такої обробки відмовитись.

Параметр /MT примушує програму використовувати багатониткову статичну версію бібліотеки часу виконання замість стандартної CRT бібліотеки. Змушує компілятор помістити назву бібліотеки LIBCMT.lib у файл OBJ, щоб компонувальник використовував LIBCMT.lib для розрішення зовнішніх символів.

Параметр компілятора /GS захищає наступні елементи:

- адресу виклику функції;
- адресу обробника виключень для функцій;
- вразливі параметри функції.

Ключ Visual Studio /GS захищає від переповнення буфера в стеці. Якщо відключення цього ключа призводить до зміни поведінки програми, то в програмі присутній «зрив» стека і запис в адреси за його межами. Таким чином, необхідно дуже ретельно ставитись до розробки коду програми, щоб уникати можливого переповнення стека.

Окремим пунктом є необхідність встановити в гілці «**Проєкт → Властивості → Властивості Конфігурації → C/C++ → Додатково**» налаштування

- **Компілювати Як → Компілювати Як C Код (/TC)**

Параметр /TC вказує, що всі файли, як аргументи командного рядка компілятора, є вихідними файлами C, навіть якщо вони не мають розширення C (за замовчуванням файли з розширенням .c компілюються як C).

Друга частина налаштувань середовища відноситься до компонувальника «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник**».

- **Файл Маніфеста → Створювати Маніфест → Ні (/MANIFEST:NO)**
- **Налагодження → Створювати Налагоджувальну Інформацію → Ні**
- **Додатково → Внесення Випадковості В Базову Адресу → Відключити Внесення Випадковості В Образ (/DYNAMICBASE:NO)**
- **Додатково → Фіксована Базова Адреса → Так (/FIXED)**
- **Додатково → Базова Адреса → 0x02000000**

Маніфест програми – це XML-файл, який описує та ідентифікує загальні та приватні паралельні збірки, з якими програма має зв'язуватися під час виконання. Маніфести програм також можуть описувати метадані для файлів, які є приватними для програм. Одні з самих відомих застосувань маніфесту – вказівка версії використовуваних бібліотек DLL, налагодження вигляду вікна, контроль за рівнем прав програми (взаємодія з UAC – User Account Control, або служба захисту користувачів – компонент та технологія безпеки Microsoft Windows). Маніфест порівняно великий, у завантажувачі PE (Portable Executable) створюється розділ `.rsrc`. Взагалі, кожна додаткова секція в PE-файлі – це мінімум 512 байтів завдяки вирівнюванню.

При створенні налагоджувальної інформації (параметр /DEBUG) дані налагодження із залученням файлів об'єктів і бібліотек розміщуються у файл бази даних програм (PDB), який оновлюється кожен раз під час останнього збору програм. Виконуваний файл (EXE або DLL-файл), створений для налагодження, містить ім'я та шлях до відповідного PDB. Відключення створення налагоджувальної інформації дозволить ненабагато зменшити розмір виконуваного файла, крім того, у режимі «Release», для якого оптимізується розмір, налагоджувальна інформація вже не потрібна.

Параметр /DYNAMICBASE змінює заголовок виконуваного образу, файла DLL або EXE, щоб вказати, чи потрібно довільно перебазувати програму під час завантаження, і вмикає випадковий розподіл віртуальних адрес, що впливає на розміщення віртуальної пам'яті куп, стеків та на інші розподіли операційної системи. При відключенні внесення випадковості в адресу компонувальник не буде створювати секцію `.reloc` (relocation table), тобто таблицю налаштування адрес у програмі, якщо вона буде завантажена за адресою, відмінною від `ImageBase`. Для виконуваних файлів EXE, на відміну від DLL, фактично це не відбувається ніколи, тому доцільно від цієї секції відмовитись.

Параметр /FIXED пов'язаний з попереднім параметром та вказує операційній системі завантажувати програму лише за бажаною базовою адресою. Для файлів DLL цей параметр встановлено за замовчанням у /FIXED:NO. Якщо вказано /FIXED, компонувальник не створює секцію переміщення `.reloc` у програмі. Якщо під час виконання операційна система не може завантажити програму за вказаною адресою, вона видає повідомлення про помилку та не завантажує програму.

Параметр компонувальника /BASE встановлює базову адресу для програми. Він замінює розташування за замовчуванням для файла EXE або DLL. Базовою адресою за замовчуванням для файла EXE є 0x400000 для 32-розрядних образів або 0x140000000 для 64-розрядних образів. Для DLL базовою адресою за замовчуванням є 0x10000000 для 32-розрядних образів або 0x180000000 для 64-розрядних образів. Якщо встановлено параметр /DYNAMICBASE:NO, операційна система спочатку намагається завантажити програму за вказаною базовою адресою або адресою за замовчуванням. Якщо там недостатньо місця, система переміщує програму. Щоб запобігти переміщенню, використовується параметр /FIXED. Компонувальник видає помилку, якщо адреса не кратна 64K. За бажанням можна вказати розмір програми. Компонувальник видає попередження, якщо програма не вміщується у вказаний розмір. Для мінімізації можна залишити базову адресу за замовчанням або обрати яку-небудь нестандартну, наприклад, 0x02000000.

І нарешті, ще два налаштування, ефект від яких буде значним, але вони можуть потребувати також значних змін у тексті програми.

▪ **Введення → Ігнорувати Всі Стандартні Бібліотеки → Так (/NODEFAULTLIB)**

▪ **Додатково → Точка Входу →MyMain**

Після заборони використання стандартної бібліотеки функцій C Run-Time (CRT) за допомогою налаштування компонувальника /NODEFAULTLIB можливе виникнення помилок типу

```
unresolved external symbol X referenced in function Y
```

Ця ситуація виникає, коли явно або неявно викликаються функції з бібліотеки CRT, які мають звертання до коду ініціалізації стартової функції програми. Якщо така помилка виникає – необхідно ще раз перевірити, чи було встановлено параметр компілятора **«Відключити Перевірку Безпеки (/GS)**» та виконати заміну функцій з цієї бібліотеки на аналогічні виклики WinAPI.

Якщо використовується /NODEFAULTLIB для збірки програм без бібліотеки CRT, також потрібно вказати параметр /ENTRY, щоб задати нову функцію точки входу в програму.

Заміна точки входу до програми насправді не замінює функцію **main** або **WinMain**, що є головною для програми та з неї починається виконання написаного коду. Але, програма для ОС Windows, що розроблена у Visual C++ , починає роботу не з функції **main** (**WinMain**), а з виконання спеціальної процедури ініціалізації. Адреса цієї процедури міститься в полі `AddressOfEntryPoint` заголовка виконуваного файла PE. Вона є звичайною функцією мови C, описаною з угодою про виклики __stdcall. Залежно від налаштувань проєкта, у Visual C++ ця функція може називатися **mainCRTStartup** або **wmainCRTStartup** (для консольних програм з використанням Ansi Char символів та Wide Char (Unicode) символів відповідно), **WinMainCRTStartup** або **wWinMainCRTStartup** (для віконних програм з використанням ANSI-символів та Unicode-символів відповідно) або **_DllMainCRTStartup** (для бібліотечних файлів DLL). Також під час ініціа-

лізації виконується функція **___security_init_cookie**. Файл `cookie` глобальної безпеки використовується для захисту від переповнення буфера в коді, скомпільованому з параметром /GS (перевірка безпеки буфера), і в коді, який використовує обробку винятків. Під час входу до функції, захищеної від переповнення буфера, файл `cookie` розміщується в стек, а при виході – значення в стеку порівнюється з глобальним файлом `cookie`. Будь-яка різниця між ними вказує на те, що відбулося переповнення буфера, і функція викликає негайне завершення програми.

Якщо обійти ініціалізацію CRT (використання параметру /ENTRY для визначення точки входу), тоді доведеться самостійно викликати **__security_init_cookie**. Якщо **__security_init_cookie** не викликається, для файла `cookie` глобальної безпеки встановлюється значення за замовчуванням, і захист від переповнення буфера відключається. Рекомендується завжди викликати **__security_init_cookie** при визначенні власної точку входу в програму, але в нашому випадку це значно збільшить розмір виконуваного файла:

```
#include <process.h>
void main(void) {
__security_init_cookie();
// основний текст програми
...
}
```

Під час виконання процедури ініціалізації виконуються такі дії:

▪ ініціалізація змінних CRT (такі, як `errno` і `osver` – від їх використання доведеться відмовитись);

▪ ініціалізація динамічної пам’яті (купи);

▪ ініціалізація середовища обробки помилок в обчисленнях з плаваючою крапкою;

▪ отримання значень аргументів командного рядка програми і змінних середовища;

▪ ініціалізація консолі і прив'язка стандартного виведення до файлових дескрипторів С. При старті виконуваного файла, в якого в заголовку PE значення поля `Subsystem` дорівнює 3 (Windows character-mode executable), створюється консоль;

▪ виклик функцій ініціалізації CRT і конструкторів глобальних змінних;

▪ і нарешті – виклик функції **main** або **WinMain**.

У налаштуваннях точки входу /ENTRY можна вказати і **main** або **WinMain** (тільки необхідно вказати, що функції не приймають жодного параметра), тоді ці функції дійсно стануть точками входу, а всі попередні дії будуть пропущені.

Завершення програми при виклику функції **exit** також спочатку виконує послідовність дій з очищення, а тільки потім відбувається саме завершення за допомогою функції **ExitProcess**, тому **exit** необхідно замінити на виклик безпосередньо **ExitProcess**. *Примітка*: функція аварійного завершення **abort** виводить повідомлення, а далі звертається до функції **exit**. Виклик **return** також доведеться замінити на **ExitProcess**, так як результат виконання **return** приймає саме бібліотека CRT.

Текст програми після заміни точки входу повинен виглядати наступним чином (замість функції **MyMain** можна змінити точку входу на будь-яку іншу):

```
#include <windows.h>
void MyMain(void)
{
...
ExitProcess(0);
}
```

Вказані вище налаштування компілятора та компонувальника можна зробити через налаштування проєкта або через командний рядок. Деякі налаштування можна задати шляхом додавання прагм у початок файла з текстом програми, наприклад:

```
#pragma optimize("s", on)
#pragma comment(linker, "/NODEFAULTLIB")
```

Після виконаних дій з налаштуваннями компілятора та компонувальника розмір виконуваного файла з прикладу зменшився до 2048 байтів.

Використання бібліотечних функцій.

Вище було вказано про відключення стандартної бібліотеки CRT, та про неможливість після цього користуватися деякими стандартними функціями мови програмування С (які власне знаходяться в цій бібліотеці).

Також, використання API функцій замість стандартних функцій мови програмування дозволить зменшити код програми за рахунок того, що реалізація API функцій виконана не в програмі, а в ОС, хоча звернення до таких функцій виконується за довший час, бо потребує в більшості ситуацій перемикання ОС у режим ядра. У таблиці 13.1 наведені часто вживані функції з бібліотеки CRT та їх еквіваленти у WinAPI.

Таблиця 13.1 – Еквівалентні API функції

Стандартна функція	Еквівалент Windows API
malloc	HeapAlloc / VirtualAlloc
free	HeapFree / VirtualFree
strcpy	StringCchCopy / StringCbCopyA
strcat	StringCchCatA / StringCbCatA
strstr	StrStrA / StrStrIA
strchr	StrChrA / StrChrIA
strlen	lstrlenA
stricmp, strcmpi	StrCmpIA/ StrCmpNA / StrCmpNIA
strcmp	lstrcmpA
sprintf	wsprintf
vsprintf	wvsprintf
printf	wsprintf + WriteConsole
scanf	ReadConsole + розбір рядка

Якщо в програмі необхідно отримати параметри командного рядка, можна скористатись функціями API **GetCommandLineW** для отримання рядка виклику програми та **CommandLineToArgvW** для його синтаксичного розбору. Вказані функції – призначені для роботи у форматі Wide Char. Окремо можна використовувати функцію **GetCommandLine**A, але тоді синтаксичний розбір результату необхідно робити самостійно – варіанта функції **CommandLineToArgvW** для формата Ansi Char немає. Приклад використання цих двох функцій (необхідно замітити, що точка входу в програму **MyMain** так і залишилась – без аргументів):

```
void MyMain(void)
{
char buf[256];
LPWSTR* szArglist;
int nArgs;
int i;

szArglist = CommandLineToArgvW(GetCommandLineW(),
            &nArgs);
if (NULL == szArglist)
      {
      WriteConsole(GetStdHandle(STD_OUTPUT_HANDLE),
            "CommandLineToArgvW failed\n",
            lstrlen("CommandLineToArgvW failed\n"),
            &t, NULL);
      ExitProcess(0);
      }
      else for (i = 0; i < nArgs; i++) {
            wsprintfA(buf,"%d:%ws\n",i,szArglist[i]);
            WriteConsole(
                  GetStdHandle(STD_OUTPUT_HANDLE),
                  buf, lstrlenA(buf), &t, NULL);
      }
```

```
// звільнення пам'яті, що було виділено при виклику
// CommandLineToArgvW
LocalFree(szArglist);
ExitProcess(0);
}
```

Нажаль, не всі функції з бібліотеки CRT можна замінити аналогічними або схожими функціями Windows API, серед яких також трапляються функції, що використовують у свою чергу звернення до бібліотеки CRT. Якщо такі функції залишаться в тексті програми, то при компонуванні буде виникати помилка виду **«unresolved external symbol _memset…»**.

Реалізацію частини функцій CRT можна знайти у встановленій Visual Studio в каталозі, який має в повному імені частину шляху «\VC\Tools\MSVC\версія збірки\crt\src\», де можна «запозичити» вихідні тексти необхідних функцій. Але частину функцій доведеться реалізовувати самостійно, або відмовитись від їх використання зовсім.

Оптимізація секцій виконуваного файла.

За замовчуванням компілятор мови С у Visual Studio створює такі основні секції:

`.text` – секція коду;

`.rdata` – секція імпорту;

`.data` – секція даних.

Також створюються секції експорту, ресурсів, переміщення (від секції `.reloc` вже позбавилися за допомогою параметра /DYNAMICBASE:NO) і т.і.

Кожна секція перед записом у файл вирівнюється за розміром на 512 байтів (розмір секції завжди кратний буде 512 байтам). Тобто, якщо ми маємо дані (наприклад тільки глобальні змінні) розміром чотири байти, то все одно секція даних займатиме 512 байтів мінімум. Для трьох вказаних секцій мінімальний їх розмір складатиме 512*3=1536 байтів через вирівнювання розміру, навіть якщо в них буде зайнято лише по одному байту. Для гарантованого вирівнювання секцій не більше ніж по 512 байтів необхідно

задати параметр /ALIGN у розділі компонувальника «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»**

▪ **Додатково → Вирівнювання Розділу → 512**

Вирівнювання можна експериментальним шляхом зменшити, але необхідно дотримуватись кратності ступеню двійки. Найменше значення, яке можна встановити – 16.

Цей параметр також можна задати за допомогою прагми, наприклад:

```
#pragma comment(linker, "/ALIGN:16")
```

У разі неможливості виконати вирівнювання за меншим значенням під час компонування буде видане повідомлення, що одна з секцій повинна мати більший розмір, наприклад, при додаванні виклику функції **__security_init_cookie** видається помилка, і вирівнювання секцій необхідно встановити в значення 64:

```
вирівнювання секції 0x8 (64) більше значення в /ALIGN
```

Крім вирівнювання можна виконати об'єднання кількох секцій в одну. Практично завжди об'єднанню піддаються секції коду, імпорту та даних. Об'єднання можна зробити за допомогою параметра /MERGE в розділі компонувальника «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»**

▪ **Додатково → Об'єднати Розділи → .rdata=.data=.text**

– виконується об'єднання секцій `.rdata`, `.data` і `.text` у секцію `.text`, або через додавання прагм у початок файла з текстом програми:

```
#pragma comment(linker, "/merge:.rdata=.text")
#pragma comment(linker, "/merge:.data=.text")
```

Секція коду має права RE (читання та виконання), секція даних – RW (читання та запис), а секція імпорту – R (тільки читання). Після об'єднання всіх трьох секцій в одну вона повинна мати загальні права для всіх секцій,

які в неї увійшли, інакше можна зіткнутися з помилками при включеному DEP (Data Execution Prevention – запобігання виконанню даних – вбудована в ОС Windows технологія, яка захищає від запуску виконуваного коду з місць, де він не передбачається: зі сторінок даних, таких як купи, стеки і пули пам'яті) та інших захистах. Для встановлення прав необхідно змінити параметр /SECTION у розділі компонувальника «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»**

▪ **Загальні → Задати Атрибути Секції → .text,EWR**

Або після прагм об'єднання секцій прописати:

```
#pragma comment(linker, "/SECTION:.text,EWR")
```

Оптимізація заголовка PE виконуваного файла.

PE-формат (Portable Executable) – це формат всіх 32- і 64-розрядних виконуваних файлів в ОС Windows. Формат PE є спеціалізацією COFF для зберігання модулів, що виконуються. Назва Portable Executable пов'язана з тим, що цей формат не залежить від архітектури процесора, для якого побудований виконуваний файл. Зараз існує два формати PE-файлів: PE32 та PE32+. Обидва вони обмежують адресний простір програми розміром 4 Гбайти (0xFFFFFFFF), але PE32 використовує 32-бітові адреси (архітектура Win32), а PE32+ – 64-бітові адреси (архітектура Win64).

Будь-який PE-файл складається з кількох заголовків та кількох (від 1 до 96) секцій (рис. 13.2). Заголовки містять службову інформацію, що описує різні властивості виконуваного файла та його структуру.

Оскільки і програми DOS, і застосунки Windows мають розширення EXE, всі виконувані файли Windows використовують схему подвійного завантаження. Вона полягає в тому, що файл починається із заголовка DOS, за яким слідує заглушка (stub), тобто невеликий EXE-файл формату DOS. При спробі завантажити файл в ОС DOS виконується заглушка, а при завантаженні файла з Windows завантажувач аналізує заголовок DOS і витягує з нього зсув до справжнього заголовка файла.

Зазвичай заглушка DOS виводить на екран повідомлення типу «Ця програма вимагає Microsoft Windows» і закінчує роботу. Однак при розро-

бці програми можна встановити в якості заглушки будь-який інший виконуваний файл DOS. Це дозволяє створювати «дуальні» програми, що працюють і в DOS, і Windows.

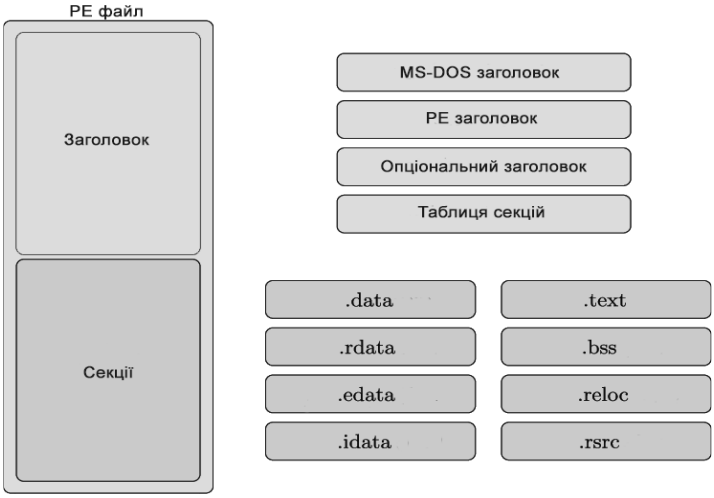


Рисунок 13.2 – Формат файла PE

За допомогою параметра компонувальника /STUB «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»** можна вказати іншу заглушку (повинна міститись у файлі з заданим ім'ям **filename**):

▪ **Командний Рядок → /STUB:filename**

Для мінімізації розміру програми необхідно в якості заглушки обрати код мінімальної довжини. У мережі Інтернет можна знайти кілька варіантів заглушок з довжиною від 64 до 12 байтів, але у Visual Studio без додаткового редагування коду отриманого файла можна скористатись варіантом заглушки довжиною 64 байти, код якої наведено нижче:

```
000000   4D 5A 00 00 01 00 00 00  02 00 00 00 FF FF 00 00
000010   40 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00
000020   B4 4C CD 21 00 00 00 00  00 00 00 00 00 00 00 00
000030   00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
```

У таблиці 13.2 наведено детальний розбір коду заглушки.

Таблиця 13.2 – Заглушка DOS

Зсув	Значення	Призначення	Коментар
1	2	3	4
+0	5A4D	Підпис exe-файла	MZ
+2	0000	Довжина останньої неповної сторінки образу, байтів	Ігнорується операційною системою
+4	0001	Довжина образу, сторінок (по 512 байтів)	Програма займає менше однієї сторінки
+6	0000	Число елементів у таблиці переміщення	В цій програмі елементів, що переміщуються, немає
+8	0002	Розмір exe-заголовка, параграфів (по 16 байтів)	Вказується розмір базової частини заголовка. З урахуванням інших значень параметрів у даному випадку означає, що виконуваний код у файлі починається зі зсуву 20h, і стартова адреса знаходиться на початку виконуваного кода
+0ah	0000	Мінімум необхідної пам’яті після кінця програми, параграфів	В даному випадку не має сенсу
+0ch	FFFF	Максимум необхідної пам’яті після кінця програми, параграфів	Традиційно відводиться вся доступна пам’ять

Продовження табл. 13.2

1	2	3	4
+0eh	0000	Сегментний зсув сегмента стека	Для встановлення регістра ss
+10h	0040	Значення регістра sp (покажчик на стек) при запуску	В цій програмі стек не має значення
+12h	0000	Контрольна сума виконуваного модуля	Не використовується
+14h	0000	Значення регістра ip (лічильника команд) при запуску	Точка входу співпадає з початком кодового сегмента
+16h	0000	Сегментний зсув кодового сегмента	Для встановлення регістра cs
+18h	0040	Зсув у файлі 1-го елемента переміщення	В цій програмі жодного елемента переміщення немає, виконуваний код знаходиться всередині заголовка, тому дане значення співпадає з кінцем програми
+1ah	0000	Номер оверлею	Не використовується
+1eh	0000 0000	Резерв – 4 байта	

+20h	4CB4 21CD	Виконуваний код:
		`mov ah,4ch`
		`int 21h`
			Програма повертає управління операційній системі

+3ch	0000 0000	Зарезервоване подвійне слово	Використовується компонувальником Windows для розташування зсуву PE-заголовка

У наведеній заглушці максимальний розмір виконуваного кода 28 байтів (розташовано зі зсувом 20h...3bh). Виконуваний код з метою економії

розміру програми знаходиться в області, яку відведено під заголовок виконуваного файла EXE.

Наведений код необхідно сформувати за допомогою будь-якого редактора бінарних файлів (можна скористатись і он-лайн версією, наприклад https://hexed.it/), та зберегти його у файл з іменем, наприклад, **dosstub.bin**.

Файл з новою заглушкою необхідно розташувати в каталозі, в якому міститься вихідний текст програми, та підключити його за допомогою параметра «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»**

- **Командний Рядок → /STUB:dosstub.bin**

Оптимізація тексту програми.

Незважаючи на те, що зроблені налаштування проєкта орієнтовані на зменшення розміру виконуваного файла, але під час написання програми також слід приділяти увагу певним моментам. Збільшенню обсягу сприяє використання великої кількості глобальних змінних. Тому слід відмовитись від глобальних змінних, а особливо ініціалізованих, та буферів вигляду

```
char buf[1024] = {0};
```

Такий код гарантовано збільшить розмір файла на 1024 байти. Тому ініціалізацію змінних, а особливо буферів, треба виконувати дуже розсудливо. Якщо потрібен глобальний буфер, то краще виділити під нього пам'ять на початку роботи програми. Таку саму проблему дає використання великої кількості локальних змінних, які описані як статичні (static).

Також слід виконати перевірку тексту програми на наявність повторювань ділянок коду: наприклад, якщо є ділянка коду, що складається з викликів кількох функцій, і використовується в програмі багато разів, то доцільно оформити таку ділянку як окрему функцію. Також необхідно виконувати згортку великих ділянок однакового коду в цикли.

Використання однакових та незмінних рядків необхідно оформити у вигляді констант (префікс `const`), або, що ще краще – використовувати `#define`, що дозволить компілятору виконати їх гарантоване об'єднання.

У невеликих програмах значний обсяг (по відношенню до загального розміру програми) займають ресурси. Тому бажано або відмовитись від їх використання (як це робилось в усіх прикладах вище), або виконати їх оптимізацію. Так, піктограми, якщо вони багатоколірні та багаторозмірні, займають також багато простору – для його зменшення слід по можливості використовувати системні піктограми, велика кількість яких зберігається у файлі **shell32.dll**. Власні піктограми краще обмежити двома розмірами 16*16 та 32*32 точки та 16 кольорами.

Також слід позбавитись від інформаційного діалогового вікна «About» з інформацією про автора та версію – всі необхідні дані краще відображати в робочому вікні програми.

У результаті змін текст програми приклада 1.4 буде таким:

```
#include <windows.h>

unsigned ThreadFunc(void* arg)
{
      char** str = (char**)arg;
      MessageBoxA(0, str[0], str[1], 0);
      ExitThread(0);
      return 0;
}

void main(void)
{
DWORD t;
char buf[256];
const char* InitStr1[2] = { "First thread","11111" };
const char* InitStr2[2] = { "Second thread","22222" };
unsigned long uThreadIDs[2];
HANDLE hThreads[2];
hThreads[0] = CreateThread(NULL, 0,
      (LPTHREAD_START_ROUTINE)ThreadFunc,
```

```
      InitStr1, 0, &uThreadIDs[0]);
hThreads[1] = CreateThread(NULL, 0,
      (LPTHREAD_START_ROUTINE)ThreadFunc,
      InitStr2, 0, &uThreadIDs[1]);
// Очікування завершення роботи ниток
WaitForMultipleObjects(2, hThreads, TRUE,
      INFINITE);
// Закриття дескрипторів
CloseHandle(hThreads[0]);
CloseHandle(hThreads[1]);
ExitProcess(0);
}
```

Розмір виконуваного файла вдалося зменшити до 1136 байтів.

Аналогічним чином виконується мінімізація віконних програм, за винятком того, що необхідно змінити параметр /SUBSYSTEM зі значення `CONSOLE` на значення `WINDOWS`. Це робиться в налаштуванні компонувальника «**Проєкт → Властивості → Властивості Конфігурації → Компонувальник»**

- **Система → /SUBSYSTEM:WINDOWS**

Необхідно звернути увагу, що для створення мінімальної за розміром віконної програми не можна користатись раніше розробленим проєктом або створювати проєкт як «Класичний застосунок Windows». Треба обов'язково створювати «Порожній проєкт», а далі вже додавати в нього файл з текстом програми через меню «**Проєкт → Додати Існуючий Елемент**», в іншому випадку досягти мінімального розміру не вдасться.

Так, для прикладу 3.1 **«Створення кнопки та робота з нею»** з теми «Віконні програми в ОС Windows» при обробці існуючого проєкта вдалося зменшити розмір виконуваного файла зі 104 кБайтів лише до 90 кБайтів, а при створенні «Порожнього проєкта» – до 2240 байтів.

При мінімізації цього прикладу були зроблені додаткові зміни тексту програми (крім заміни точки входу):

1) для отримання дескриптора екземпляра процесу `hInstance` (так як він більше не передається як параметр до головної функції програми) на початку роботи програми необхідно написати виклик

```
HINSTANCE hInstance = GetModuleHandle (NULL);
```

2) для коректного завершення роботи програми виклик **return msg.wParam** у головній функції необхідно замінити на **ExitProcess**.

Також слід зазначити, що використання в програмах бібліотеки MFC (Microsoft Foundation Class) потребує наявності коду ініціалізації, тому виконати повну мінімізацію для таких програм неможливо.

13.2. Контрольні питання

1. Які обмеження виникають при відмові від використання символів у форматі Wide Char у програмі?
2. На якому етапі розробки програми можна робити мінімізацію за розміром виконуваного файла?
3. Якими способами можна встановити налаштування проєкта у Visual Studio?
4. З якою метою в заголовку PE виконуваного файла розміщується заглушка DOS?
5. Яким чином можна замінити функції з бібліотеки CRT?

13.3. Індивідуальні завдання

Виконати мінімізацію консольної та віконної програми. В якості індивідуальних завдань взяти свої раніше виконані завдання з попередніх тем: одна програма консольна та одна віконна. Оцінити розміри отриманих мінімізованих виконуваних файлів розроблених програм та порівняти їх з розмірами виконуваних файлів, отриманих раніше.

14. ПРОЦЕСИ В ОС LINUX

Мета: оволодіння системними програмними засобами породження та завершення процесів в ОС Linux.

14.1. Теоретичні відомості

Системний інтерфейс Linux визначається як бінарний (двійковий) інтерфейс застосунків та інтерфейс програмування застосунків, що надається завдяки взаємодії ядра Linux (центру операційної системи), бібліотеки GNU C (glibc) та компілятора GNU C. Всі теми щодо програмування в Linux, а також приклади та індивідуальні завдання орієнтовані для роботи з інтерфейсом командного рядка ОС.

Далі всі приклади наведено для ОС Ubuntu. У тому випадку, якщо цю ОС встановлено як гостьову в Oracle VM Virtual Box, доцільно підключити можливості обміну інформацією з основною системою – спільний буфер обміну та підтримка спільної технології Drag and Drop. Для цього необхідно підключити образ диску доповнень гостьової ОС (в меню «Пристрої») та запустити скрипт встановлення доповнень:

```
sudo /media/dir/VBox_Gas_6.1.20/VBoxLinuxAdditions.run
```

де `dir` – каталог користувача (наприклад, `user`), `VBox_Gas_6.1.20` – каталог підключеного диску доповнень (може бути іншим у конкретному випадку). Після встановлення доповнень ОС необхідно перезавантажити.

Компілятор.

Одним з перших програмних продуктів, створених у рамках проєкта GNU (рекурсивне скорочення, *GNU's Not UNIX*), є компілятор мови C з відкритим кодом. Цей компілятор включається в поставку всіх версій ОС Linux. У разі, якщо його не встановлено, необхідно виконати команди:

```
sudo apt install gcc
```

та

```
sudo apt install g++
```

У розпорядження розробнику надається 4 компілятори на вибір:

- **cc** – стандартний компілятор мови С;
- **c++** – стандартний компілятор мови С++;
- **gcc** – GNU-компілятор мови С (і не тільки);
- **g++** – GNU-компілятор мови С++ (і не тільки).

Способи запуску й переважна більшість опцій всіх зазначених компіляторів ідентичні, вибір того або іншого компілятора відбивається тільки на імені команди. Наведені приклади буди протестовані за допомогою компілятора **cc**, тому в опису далі буде основна увага приділятись саме йому.

Компілятор мови С виконує як власне компіляцію – переклад початкового тексту на машинну мову, результатом чого є об'єктний модуль, так і редагування зв'язків – збірку з декількох об'єктних модулів (у тому числі, і бібліотечних) виконуваного модуля.

Файли з початковими текстами С-програм повинні мати розширення .c, наприклад: **hello.c**. Результатом компіляції є файл, що містить об'єктний модуль, його ім'я збігається з ім'ям початкового модуля, а розширення – .o, наприклад: **hello.o**. Для файла, що містить виконуваний модуль, стандартного розширення не існує. При компіляції програми, що складається з єдиного початкового модуля, об'єктний модуль автоматично видаляється після створення компілятором виконуваного модуля.

Загальний формат команда виклику компілятора має такий вигляд:

```
cc [опції] [вихідний_файл] файл1 [файл2]
```

Найбільш часто вживані опції компілятора наступні:

`-c` – подавляє фазу редагування зв'язків, створює об'єктний модуль для кожного початкового модуля з перерахованих у параметрах виклику. `вихідний_файл` із цією опцією не задається. Опція може застосовуватися разом з опцією `-I`;

`-o` – компіляція і редагування зв'язків. Файли, що перераховані в параметрах виклику й мають розширення .c, розглядаються як початкові модулі та компілюються зі створенням об'єктного модуля для кожного початкового модуля; а файли, що мають розширення .o, розглядаються як об'єктні модулі й підключаються при редагуванні зв'язків. Параметр `вихідний_файл` задає ім'я файла виконуваного модуля. Опція може застосовуватися разом з опціями `-L`, `-l`, `-I`;

`-L`*`каталог`* – додати *`каталог`* у список каталогів, які містять об'єктні бібліотечні модулі;

`-l`*`бібліотека`* – при редагуванні зв'язків підключити модулі з *`бібліотеки`*;

`-I`*`каталог`* – шукати файли, що підключаються (#include), імена яких не починаються з /, спочатку в *`каталозі`*, а лише потім – у стандартних каталогах для файлів, що підключаються;

`-E` – виконати обробку зазначених початкових модулів тільки препроцесором, результат направляється в стандартний вивід. `вихідний_файл` із цією опцією не задається. Опція може застосовуватися разом з опцією `-I`;

`-w` – подавити видачу застережливих повідомлень.

Приклади:

1) Компіляція початкового модуля **hello.c** з видачею повідомлень про помилки на пристрій стандартного виведення. Файл об'єктного модуля не створюється.

```
cc hello.c
```

2) Компіляція початкового модуля **hello.c** з видачею повідомлень про помилки на пристрій стандартного виведення. При успішній компіляції об'єктний модуль записується у файл `hello.o`.

```
cc -c hello.c
```

3) Редагування зв'язків для об'єктного модуля **hello.o**, виконуваний модуль записується у файл **hello**.

```
cc –o hello hello.o
```

4) Створення виконуваного модуля у файлі **hello** з об'єктного **hello.o** та модуля **hello1.c** (останній модуль є початковим, він компілюється попередньо).

```
cc –o hello hello.o hello1.c
```

5) Створення виконуваного модуля у файлі **hello** з об'єктних модулів **hello.o** та **hello1.o** з підключенням об'єктних модулів з бібліотеки **hellolib**.

```
cc –o hello hello.o hello1.o -l hellolib
```

Процеси.

Процес в UNIX/Linux являє собою одиницю роботи обчислювальної системи, якій операційна система виділяє ресурси. Із задовільним ступенем наближення можна визначити процес як виконувану програму. Кожний процес у системі має свій унікальний ідентифікатор процесу (PID), що представляється цілим числом.

Кожному процесу в операційній системі відповідає запис у таблиці процесів і адресний простір процесу. Запис у таблиці процесів (і її розширення в адресному просторі процесу) містить керуючу інформацію про ресурси, виділені процесу, і про стан процесу. Адресний простір містить коди та дані процесу.

Процеси можуть запускатися для різних цілей:

- виконання якихось одноразових дій (наприклад, скрипти);
- виконання завдань під управлінням користувача (інтерактивні процеси);
- безперервної роботи у фоновому режимі (сервіси або демони).

Процес-«демон» (daemon) – це процес, який запускається для довгострокової роботи у фоновому режимі, відключається від терміналу, який його запустив (його стандартні потоки введення-виведення закриваються або перенаправляються в лог-файл), і змінює свої права доступу на мінімально необхідні. Управління таким процесом здійснюється за допомогою механізму сигналів ОС.

Процес може породжувати інший процес. Породження нового процесу в UNIX/Linux реалізовано копіюванням запису таблиці процесів, таким чином, процес-«нащадок» у момент свого породження являє собою точну копію батьківського процесу. Процес-«предок» і «нащадок» далі виконуються паралельно, але батьківський процес може й очікувати завершення дочірнього процесу.

Процеси в UNIX/Linux виконуються в режимі поділу часу, це означає, що час центрального процесора рівномірно (з урахуванням пріоритетів) розподіляється між всіма готовими до виконання процесами. Навіть якщо процес не переходить у стан очікування (наприклад, очікування виконання операції введення-виведення) зі своєї ініціативи, після закінчення виділеного процесу кванта часу, виконання процесу буде перервано операційною системою, і процесор буде відданий більш пріоритетному процесу. UNIX/Linux застосовує динамічне переобчислення пріоритетів, пріоритет процесу, що виконується, знижується, а пріоритети процесів, що очікують, підвищуються.

Породження процесів. Новий процес породжується системним викликом **fork**, який створює дочірній процес – копію батьківського. У дочірньому процесі виконується та ж програма, що й у батьківському, і коли дочірній процес починає виконуватися, він виконується із точки повернення із системного виклику **fork**. Системний виклик **fork** повертає батьківському процесу PID дочірнього процесу, а дочірньому процесу – 0. По коду повернення виклику **fork** дочірній процес може «усвідомити» себе як дочірній. Свій PID процес може одержати за допомогою системного виклику **getpid**, а PID батьківського процесу – за допомогою системного виклику **getppid**. Якщо потрібно, щоб у дочірньому процесі виконувалася програма,

відмінна від програми батьківського процесу, процес може перемінити виконувану в ньому програму за допомогою одного із системних викликів сімейства **exec**. Всі виклики цього сімейства завантажують для виконання в процесі програму із заданого у виклику файла й відрізняються один від одного способом передачі параметрів цій програмі. Таким чином, найпоширеніший контекст застосування системного виклику **fork** виглядає приблизно так (в прикладі не наведено аналіз помилки породження процесу):

```
/* породження дочірнього процесу і
   запам'ятовування його PID */
 if (!(ch_pid=fork())
    /* завантаження іншої програми в
       дочірньому процесі */
    exec(програма);
 else
    продовження батьківського процесу
```

Пріоритет процесу. За інших рівних умов процесорний час розподіляється між процесами, що виконуються, порівну, але процес може встановити добавку до пріоритету. Добавка ця, проте, не підвищує, а знижує пріоритет процесу в сенсі використання процесора. Тільки процеси суперкористувача можуть одержувати негативну добавку до пріоритету, тобто, реально підвищувати свій пріоритет.

Процес може змінити свій пріоритет за допомогою системного виклику **nice**, а пріоритет іншого процесу може бути змінений системним викликом **setpriority**. Системний виклик **getpriority** дозволяє взнати пріоритет процесу.

Завершення процесу. Нормальне завершення процесу відбувається при досягненні кінця функції **main** або при виконанні системного виклику **exit**. При цьому процес установлює деякий код свого завершення, який може бути прочитаний процесом-«предком».

Примусове завершення процесу ззовні може бути виконане за допомогою системного виклику **kill**, що посилає процесу сигнал. Відзначимо, що гарантовано «вбити» процес, що має `PID = p`, можна системним викликом:

```
kill(p,SIGKILL)
```

Процес при завершенні звільняє всі свої ресурси (за винятком PID) і стає «зомбі» (zombie) – порожній запис у таблиці процесів, що зберігає код завершення для батьківського процесу. Дочірній процес завжди спочатку стає «зомбі» перед видаленням із таблиці процесів.

Система повідомляє батьківському процесу про завершення дочірнього за допомогою сигналу SIGCHLD (механізм сигналів буде розглянуто пізніше). Передбачається, що після отримання SIGCHLD він зчитує код завершення за допомогою системного виклику **wait** або **waitpid**, після чого запис «зомбі» буде видалений зі списку процесів. Якщо батьківський процес ігнорує SIGCHLD (а він ігнорується за замовчуванням), то «зомбі» залишаються до його завершення.

Процеси «зомбі» не займають пам'яті (як процеси «сироти»), але блокують записи в таблиці процесів, розмір якої обмежений для кожного користувача і системи в цілому. Один з найпростіших способів «вбити зомбі» – це завершити батьківський процес. Якщо в системі завершується який-небудь процес, то спеціальний «демон» **init** наслідує всіх нащадків завершеного процесу і видаляє їх, якщо вони вже завершили своє виконання. Можлива ситуація, коли батьківський процес виконує якісь необхідні дії, і, видаливши його, можна втратити дані.

Вочевидь, що краще запобігти виникненню процесів «зомбі», ніж з ними боротися. Одне з рішень – використання функцій очікування завершення процесу.

Приклад 14.1. Створення процесу «зомбі». У батьківському процесі створюється дочірній процес, який одразу завершує свою роботу, батьківський процес чекає 50 секунд. На цей час дочірній процес стає процесом «зомбі». Якщо запустити цю програму у фоновому режимі та перевірити

стан процесу (команда **ps**), можна побачити, що стан дочірнього процесу буде з позначкою «Z - defunct».

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    // fork повертає PID батьківському процесу
    pid_t child_pid = fork();
    if (child_pid > 0)
    // батьківський процес: чекаємо 50 секунд
        sleep(50);
    else
    // дочірній процес: одразу завершуємо роботу
        exit(0);
    return 0;
}
```

Процес, батьківський процес якого більше не існує, тобто або завершено, або припинено, не чекаючи завершення цього дочірнього процесу, називається процесом «сиротою» (orphan). Такий процес довго не існує, процес «сирота» незабаром приймається процесом **init**, коли його батьківський процес «вмирає».

Приклад 14.2. Створення процесу «сироти»:

```
#include<stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    // створення дочірнього процесу
```

```
    int pid = fork();
    if (pid > 0)
    // батьківський процес –
    // виводимо повідомлення і завершуємо роботу
        printf("in parent process");
    // якщо PID = 0 – дочірній процес
    // якщо PID < 0 – помилка створення процесу
    else if (pid == 0)
    {
    // дочірній процес - чекаємо 30 секунд,
    // виводимо повідомлення і завершаємо роботу
        sleep(30);
        printf("in child process");
    }
    return 0;
}
```

Очікування завершення процесу. Батьківський процес може очікувати завершення дочірнього процесу (або процесів) за допомогою системних викликів **wait** або **waitpid**. Якщо дочірній процес ще не завершився, батьківський процес переводиться таким системним викликом у стан очікування до завершення дочірнього процесу (втім, процес може й не очікувати завершення нащадка, а тільки перевірити, чи завершився він). Ці системні виклики дозволяють також батьківському процесу взнати код завершення нащадка.

14.2. Контрольні питання

1. Чим відрізняється процес «зомбі» від процесу «сироти»?
2. Яка відмінність створення нового процесу в Linux від Windows?
3. Чи є аналог виклику **fork** у Windows? Як він працює?
4. У чому полягають основні відмінності між **cc** та **gcc**?
5. Назвіть можливі варіанти отримання PID поточного процесу, батьківського процесу та дочірнього процесу.

6. Що відбувається з процесом, коли він викликає функцію **sleep**?

7. В якому процесі (батьківський або дочірній) після повернення з першого виклику **fork** буде виконано другий його виклик у наведеному фрагменті тексту програми? Поясніть.

```
...
while(fork()==0);
...
```

14.3. Індивідуальні завдання

Розробити програми для виконання поставленого завдання. Всі дії, що стосуються як батьківського процесу, так і породжених дочірніх процесів, виконувати за допомогою різних виконуваних файлів. Всі вихідні дані вводяться в батьківському процесі з клавіатури. Робота дочірніх процесів має виконуватися паралельно. Сигналом про закінчення роботи дочірнього процесу вважати його завершення. Результати своєї роботи дочірні процеси передають батьківському процесу шляхом запису структурованої інформації у файли, індивідуальні кожному процесу (наприклад: «Son=1, x=5, x!=120»).

Використання файлів для передачі даних між процесами наведено у наступному прикладі.

Приклад 14.3. У батьківському процесі створюється дочірній процес та очікується його завершення. Після цього з файла **son.txt** зчитується рядок, виводиться на екран та файл видаляється.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

FILE *fl;
pid_t pw;
```

```
char str1[256];

int main()
{
// створюємо дочірній процес
pw=fork();
if (pw==0)
{
      // дочірній процес – запускаємо файл son
      if (execl("./son"," ",NULL)<0)
      {
            perror(str1);
            exit(0);
      }
}
// чекаємо завершення дочірнього процесу
waitpid(pw,NULL,0);
// відкриття файла на читання
if ((fl=fopen("./son.txt","r"))==NULL)
{
      perror(str1);
      printf("Error file opening!\a\n");
      exit(1);
}
// читання рядка з файла
fgets(str1,256,fl);
printf("Father read about son: %s\n", str1);
fclose(fl);
// видалення файла
unlink("son.txt");
return 0;
}
```

У дочірньому процесі зчитується власний PID, створюється файл **son.txt** та записується в нього рядок тексту з PID.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>

FILE *fl;
pid_t pwmy;
char str[]="I`m son, my pid is: ";

int main()
{
// отримання власного PID
pwmy = getpid();
// створення файла son.txt
if ((fl=fopen("./son.txt","a+"))==NULL)
{
      perror(str);
      printf("Error file opening!\a\n");
      exit(1);
}
// запис рядка у файл
fprintf(fl,"%s%d",str,pwmy);
fclose(fl);
exit(0);
}
```

Остаточний результат індивідуального завдання одержати в батьківському процесі. Хід роботи процесів відображати на екрані повідомленнями (початок роботи, обробка даних, завершення роботи тощо).

1) Обчислити для n (n>3) клієнтів 12 відсотків від заданої користувачем суми кредиту. У «нащадку» 1 обчислити 12-відсоткову виплату першої половини клієнтів [1 – n/2], а в «нащадку» 2 – для другої половини клієнтів [(n/2+1) – n].

2) Виконати породження та перевірку статусу завершення двох дочірніх процесів. Виконання «нащадка» 1 завершити викликом **exit()**, а виконання «нащадка» 2 – за допомогою сигналу, надісланого з терміналу.

3) Виконати породження «Сина», «Онука» та «Правнука». У тілі кожного процесу визначити інформацію про процес: ідентифікатор та покоління (батько – 1 покоління, син – 2 покоління тощо). Кожен «нащадок» надсилає повідомлення про народження шляхом запису у файл інформації про себе. Кожен «Батько», отримавши повідомлення, виводить на екран інформацію лише про свого «Сина».

4) Обчислити число поєднань C(k,n)=n!/(k!·|n-k|!), де n>0, k>0, n≠k. Обчислення n!, k!, | n-k |! виконати в трьох процесах-«нащадках» відповідно.

5) Виконати породження 7 процесів-«нащадків». У кожному з процесів, що виконуються, визначити значення свого і батьківського ідентифікатора. Від кожного з «батьків» виконати передачу повідомлення «Hello, Son! My Father ID =, My ID =, Your ID = !» до свого «сину» через файл. Завершувати кожен процес лише після завершення всіх його «нащадків».

6) Обчислити щільність нормального розподілу в точці x за формулою $f(x)=Exp(-x^2/2)/Sqrt(2\cdot x)$, де x>0. Обчислення експоненти та квадратного кореня виконати в двох процесах-«нащадках» відповідно.

7) Обчислити щільність опуклого розподілу в точці x за формулою $f(x)=(1-Cos(x))/(x^2)$, де x>0. Обчислення Cos(x) і x^2 виконати в двох процесах-«нащадках» відповідно.

8) Обчислити значення функції $f(x)=((-1)^x+x^{(2x-1)})/(2x-1)!$, де x>0. Обчислення $(-1)^x$, $x^{(2x-1)}$, (2x-1)! виконати в трьох процесах-«нащадках» відповідно.

9) Обчислити значення функції f(x,k)=(x(2k+1)+x(5k-1))/(2k+1). Обчислення x(2k+1) і x(5k-1) виконати в двох процесах-«нащадках» відповідно.

10) Обчислити значення функції $f(x,k)=(1!\cdot 2!\cdot 3!\cdot\ldots\cdot k!)/x!$, де $x>0$, $k>1$, $k<8$. Для обчислення кожного факторіалу виконати породження процесу «нащадка».

11) Визначити, сума цифр якого цілого числа більша від заданих трьох. Підрахунок суми цифр кожного числа виконати в окремому процесі-«нащадку».

12) Здійснити пошук максимального з 8 чисел за допомогою пошуку більшого з двох чисел. Пошук більшого із двох для кожної пари чисел виконати в окремому процесі-«нащадку».

13) Обчислити площу кільця за значеннями внутрішнього та зовнішнього радіусів, використовуючи функцію обчислення площі кола. Обчислення площі кола з внутрішнім та зовнішнім радіусом виконати в двох процесах-«нащадках» відповідно.

14) Обчислити суму факторіалів усіх парних чисел від m до n включно. Для обчислення факторіалу кожного парного числа в діапазоні [m, n] виконати породження процесу-«нащадка». При цьому $m<10$, $n<22$.

15) Виконати породження трьох дочірніх процесів кожні 2 секунди. Кожен із «нащадків» виконується заданий батьком час, посилаючи перед закінченням «батькові» повідомлення через файл. «Батько» виводить на екран усі повідомлення про зміни в процесах.

15. НИТКИ І СЕМАФОРИ НИТОК В ОС LINUX

Мета: оволодіння системними засобами породження та спільного виконання ниток в ОС Linux.

15.1. Теоретичні відомості

Нитки.

В операційних системах нитки є паралельними потоками виконання в складі одного процесу. Деякі перекладачі перекладають термін «thread» як «нитка», деякі як «потік».

У складі процесу може бути запущено кілька ниток, які виконуються паралельно (або квазіпаралельно – у режимі поділу часу процесора). Можна вважати (і в деяких операційних системах це дійсно так), що в будь-якому процесі мається принаймні одна нитка – головна – та, в якій виконується функція **main**. Головна нитка може породжувати інші нитки. У програмі процесу нитка має вигляд процедури/функції, що викликається спеціальним системним викликом і після виклику виконується паралельно з ниткою, що її запустила.

Нитки, що належать одному процесу, спільно використовують майже всі ресурси свого процесу. Однак, у кожної нитки є й деякі власні ресурси. Очевидно, що першим таким власним ресурсом нитки є процесорний час. Два інших власних ресурси нитки – вектор стану та стек. Наявність у нитки власного вектора стану дозволяє системі переривати й поновлювати виконання ниток, тобто, планувати нитки в режимі багатозадачності з витісненням. Наявність у нитки власного стека дозволяє запускати на паралельне виконання кілька екземплярів однієї й тієї ж процедури/функції. Оскільки локальні змінні функції розміщуються в стеці, кожна нитка має свій набір локальних змінних, які незалежні від інших ниток. Статичні ж змінні програми – загальні для всіх ниток, також всі нитки використовують той самий екземпляр коду функції.

При введенні поняття ниток розробники систем мали на увазі два можливих призначення:

▪ по-перше, створення ниток відбувається швидше, ніж створення процесу, таким чином, нитки підходять для задач, що вимагають швидкого розпаралелювання з мінімальними накладними витратами (наприклад, розпаралелювання виконання запиту до бази даних);

▪ по-друге, оскільки нитки спільно використовують ресурси свого процесу, вони являються зручним способом програмування тісно зв'язаних паралельних робіт, тобто таких, які використовують великий обсяг спільних даних та інших ресурсів.

У клонах операційної системи UNIX мається два підходи до забезпечення механізму ниток:

▪ у деяких операційних системах цього сімейства (наприклад, Open UNIX) в ядро системи включені «віртуальні процесори», так звані, легковагові процеси, що спеціально підтримують багатонитковість. Відповідно, системні виклики, що забезпечують роботу з нитками, звертаються безпосередньо до ядра та використовують механізм легковагових процесів;

▪ розробники інших операційних систем (наприклад, FreeBSD) виходили з того, що в клонах UNIX створення «повноцінного» процесу – операція сама по собі швидка, таким чином, вони бачили своє основне завдання в забезпеченні можливості спільного використання ресурсів нитками. Цю задачу вони вирішили введенням в ядро такого розширення системного виклику **fork**, який забезпечує спадкування дочірнім процесом будь-яких (на вибір програміста) ресурсів батьківського процесу. Для сумісності з раніше розробленим програмним забезпеченням сам системний виклик **fork** залишається без змін, а його розширення одержує інше ім'я (в FreeBSD – **pfork**). Нитка в таких системах фактично являє собою окремий процес, що використовує ресурси батьківського процесу разом з ним.

В операційних системах UNIX/Linux при розробці програм мовою програмування C/C++ забезпечено стандарт API ниток POSIX (**pthread**) для всіх пов'язаних з нитками функціями. Нитки найбільш ефективні в багатопроцесорних або багатоядерних системах, де вони можуть бути реалізовані на рівні ядра для досягнення швидкості виконання. Виграш також можна отримати в однопроцесорних системах, використовуючи затримку в

операціях введення/виведення або інших системних функціях, які можуть призупинити процес.

Системні виклики UNIX/Linux. В операційній системі Linux механізм реалізації ниток дотримується (потребує) підходу BSD: у системі мається виклик **clone**, який безпосередньо звертається до ядра системи та являє собою розширену версію **fork**, а стандартний API забезпечується функціями бібліотеки **pthread**, які є надбудовами над викликом **clone**.

Нижче перераховані найбільш часто вживані функції бібліотеки ниток, що відповідають стандарту та забезпечують основні пов'язані з роботою ниток операції в Linux.

Виконання ниток. При розробці програми необхідно підключити файл заголовка **pthread.h**, щоб використовувати всі функції бібліотеки **pthread**. У деяких випадках також необхідно явно вказати бібліотеку під час компіляції файла:

```
cc -pthread file.c
```

Функція **pthread_create** створює нову нитку. Цій функції передається покажчик на атрибути виконання нитки (більшість ниток виконується зі стандартними атрибутами), покажчик на потокову функцію (функцію, виконувану в нитці), параметр потокової функції, адреса змінної типу **pthread_t**, в яку **pthread_create** записує ідентифікатор створеної нитки. Функція **pthread_create** запускає потокову функцію в новій нитці, та вона виконується паралельно з іншими нитками процесу.

Потокова функція має прототип:

```
void * ім'я_функції(void *);
```

Параметр, що передається у функцію, та значення, що з неї повертається – покажчики, таким чином функція може приймати й повертати будь-яку інформацію.

Синтаксис функції **pthread_create**:

```
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

Параметри:

`thread` – покажчик на беззнакове ціле число, яке повертає ідентифікатор створеної нитки;

`attr` – покажчик на структуру, яка використовується для визначення атрибутів нитки, таких як відокремлений стан, політика планування, адреса стеку тощо. Значення NULL використовується для атрибутів нитки за замовчуванням;

`start_routine` – покажчик на підпрограму, яка виконується ниткою. Функція має один атрибут, але якщо функції потрібно передати кілька значень, слід використовувати структуру;

`arg` – покажчик на `void`, що містить аргументи функції, визначеної в попередньому аргументі.

Нитка завершується при завершенні виконання потокової функції або при виконанні функції **pthread_exit**. Якщо використовувати **exit** замість **pthread_exit** для завершення нитки, весь процес із усіма пов'язаними нитками буде припинено, навіть якщо деякі з ниток можуть усе ще працювати.

Синтаксис:

```
void pthread_exit(void *retval);
```

функція приймає обов'язковий параметр `retval`, який є покажчиком на ціле число, яке зберігає статус повернення завершеної нитки. Область цієї змінної має бути глобальною, щоб будь-яка нитка, яка очікує на приєднання до цієї нитки, могла прочитати статус повернення.

Функція **pthread_join** застосовується для ниток так само, як системний виклик **wait** застосовується для процесів: вона змушує нитку, що її ви-

кликала, очікувати завершення зазначеної у виклику нитки. Функція дозволяє нитці, що її викликала, одержати значення, яке повернула завершена нитка.

В якомусь сенсі **pthread_join** схожа на виклик **waitpid**, але з деякими відмінностями. По-перше, всі нитки однорангові, у тому числі відсутній ієрархічний порядок, тоді як процеси утворюють дерево і підпорядковані ієрархії відносин «батько – нащадок». Тому можлива ситуація, коли нитка А, що створила нитку Б, що в свою чергу створила нитку В, але потім після виклику функції **pthread_join** А може чекати нитку В, або навпаки, В може чекати нитку А. По-друге, не можна дати вказівку одній нитці очікувати завершення групи ниток, як це можливо з викликом **waitpid(-1, &status, options)**. Також неможливо здійснити виклик **pthread_join** без блокування поточної нитки.

Синтаксис:

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

Функція приймає такі параметри:

`th` – ідентифікатор нитки, яку очікує поточна нитка;

`thread_return` – покажчик на місце, де зберігається статус виходу нитки, заданої в `th`.

Виконання нитки може бути примусово припинене з іншої нитки за допомогою функції **pthread_cancel**, якій задається ідентифікатор нитки, яку треба завершити. Важливо розуміти, що незважаючи на те, що **pthread_cancel** повертається одразу і може завершити нитку достроково, її не можна назвати засобом примусового завершення ниток. Справа в тому, що нитка не тільки може самостійно вибрати момент завершення у відповідь на виклик **pthread_cancel**, а й зовсім його ігнорувати. Виклик функції **pthread_cancel** слід розглядати тільки як запит виконання дострокового завершення нитки. Тому, якщо важливо, щоб нитка була видалена, потрібно дочекатися її завершення функцією **pthread_join**.

Синтаксис:

```
int pthread_cancel(pthread_t thread);
```

Обов'язковий параметр `thread` – ідентифікатор нитки, до якої надсилається запит на скасування.

Функція **pthread_self** використовується для отримання ідентифікатора поточної нитки.

Синтаксис:

```
pthread_t pthread_self(void);
```

Функція **pthread_equal** порівнює, чи однакові дві нитки чи ні. Якщо дві нитки рівні, функція повертає ненульове значення, інакше нуль. Функція **pthread_equal** необхідна, оскільки формат ідентифікаторів ниток слід вважати «чорним ящиком»: не рекомендується виконувати пряме порівняння двох безпосередньо значень ідентифікаторів ниток. Синтаксис:

```
int pthread_equal(pthread_t t1,
                  pthread_t t2);
```

Параметри:

`t1` – ідентифікатор першої нитки;

`t2` – ідентифікатор другої нитки.

Приклад 15.1. Створення двох ниток, що виводять передану їм інформацію.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

void* func(void* arg)
{
```

```
char* str = (char*)arg;
printf("Inside the %s thread\n",str);
pthread_exit(NULL);
}

int main()
{
pthread_t ptid1, ptid2;
pthread_create(&ptid1, NULL, &func, "First");
pthread_create(&ptid2, NULL, &func, "Second");
// очікування завершення ниток
pthread_join(ptid1, NULL);
pthread_join(ptid2, NULL);
return 0;
}
```

Функція **pthread_detach** використовується для від'єднання нитки. Відокремлена нитка не потребує приєднання нитки після завершення. Ресурси нитки автоматично звільняються після завершення, якщо нитку від'єднано. Синтаксис:

```
int pthread_detach(pthread_t thread);
```

Приймає обов'язковий параметр `thread`, який є ідентифікатором нитки, яку потрібно від'єднати. При успішному завершенні **pthread_detach** повертає код 0, ненульове значення сигналізує про помилку.

Від'єднану нитку вже не перехопити за допомогою виклику **pthread_join**, наприклад, щоб отримати статус завершення. Також не можна скасувати від'єднаний стан. У цьому випадку дочірня нитка виконується асинхронно по відношенню до батьківської і остання вже не може отримати статусу завершення дочірньої нитки.

Приклад 15.2. Створення від'єднаної нитки. Головна нитка завершує свою роботу до закінчення роботи створеної нитки, але процес не завершується, поки нитка також не закінчить роботу.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

void* func (void* arg)
{
for (int i = 10; i > 0; i--)
{
      sleep(1);
      printf("CountDown %d\n", i);
}
printf("Stop\n");
pthread_exit(NULL);
}

int main()
{
pthread_t tid;
pthread_create(&tid, NULL, &func, NULL);
pthread_detach(tid);
pthread_exit(NULL);
}
```

Порівняння ниток та процесів.

Переваги паралельних ниток в одному процесі:

▪ нитки досить просто обмінюються даними по зрівнянню з процесами;

▪ створювати нитки для ОС простіше та швидше, чим створювати процеси.

Недоліки:

▪ при програмуванні програми з множинними нитками необхідно забезпечити потокову безпеку функцій (thread safety). Програми, що виконуються через безліч процесів, не мають таких вимог.

▪ одна нитка може зашкодити іншим, оскільки нитки ділять загальний адресний простір. Процеси більш ізольовані один від одного;

▪ нитки конкурують між собою в адресному просторі. Стек та локальне сховище нитки розташовуються в частині віртуального адресного простору процесу, що робить цю частину недоступним для інших ниток. Для вбудованих пристроїв таке обмеження може мати важливе значення.

Семафори ниток.

Оскільки статична пам'ять програми – загальна для всіх ниток, при роботі зі спільно використовуваними змінними можливі конфлікти доступу. Конфлікти можуть виникати при одночасному звертанні до таких змінних. Проявляються такі конфлікти не у відмові в доступі або у фатальних помилках, а в можливому порушенні цілісності спільно використовуваних даних. Відповідальність за збереження такої цілісності лежить на програмісті. Програміст повинен забезпечити, щоб логічно закінчені операції над спільно використовуваними даними виконувалися як транзакції, тобто, під час їхнього виконання інші нитки, що виконуються паралельно, не могли змінювати значення цих же даних. Ділянки коду програми, що виконують таку роботу зі спільно використовуваними ресурсами, називаються критичними секціями. Для забезпечення цілісності потрібно, щоб дві або більше нитки не могли перебувати в своїх критичних секціях одночасно. Це – проблема, що найбільш часто виникає при програмуванні багатониткових застосунків, однак, задачі синхронізації й обліку ресурсів у таких застосунках також виникають. Для вирішення цієї проблеми можна застосовувати звичайні семафори – як і в аналогічних задачах для процесів. Однак для цілей роботи з нитками в багатьох операційних системах вводяться спеціальні засоби, призначені для забезпечення взаємодій тільки в межах одного про-

цесу. Це «полегшені» семафори-лічильники ресурсів, виключаючі семафори та сигналізуючі семафори. Фактично всі ці засоби являють собою звичайні семафори, використовувані тільки в межах одного процесу, однак, інтерфейси цих засобів відрізняються від інтерфейсу семафорів процесів та являються більше специфічними й зручними в застосуванні.

Приклад 15.3. Асинхронна робота. Створюються 2 нитки. Перша нитка, яка починає працювати отримує номер 1 (використовується глобальна змінна), виводить інформацію про початок роботи, працює (для емуляції викликається **sleep**), виводить інформацію про закінчення. Друга нитка отримує номер 2, та виконує аналогічні дії. При запуску програми можна побачити, що робота ниток перекривається в часі, і номер нитки 1 змінюється на 2.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    counter += 1;
    printf("\n Job %d has started\n", counter);
    sleep(3);
    printf("\n Job %d has finished\n", counter);
    pthread_exit(NULL);
}
```

```
int main(void)
{
int i = 0;
int error;
while (i < 2) {
   error=pthread_create(&(tid[i]),NULL,&trythis,NULL);
   if (error != 0)
        printf("\nThread can't be created :[%s]",
               strerror(error));
   i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
return 0;
}
```

Виключаючі семафори. Виключаючий семафор або м'ютекс по суті є тим самим, чим є бінарний семафор (тобто семафор з двома станами: «зайнятий» і «не зайнятий»). Але термін «mutex» частіше використовується, щоб описати схему, яка оберігає два процеси від одночасного використання загальних даних/змінних. У той час як термін «бінарний семафор» найчастіше використовується для опису конструкції, яка обмежує доступ до одного ресурсу. Тобто бінарний семафор використовують там, де один процес займає семафор, а інший його звільняє. Тоді як м'ютекс звільняється тим самим процесом/ниткою, який зайняв його.

Виключаючий семафор представляється в Linux-програмі змінною типу `pthread_mutex_t`. Семафор повинен бути ініціалізований за допомогою функції **pthread_mutex_init**. Як правило, застосункам задовольняють стандартні параметри ініціалізації виключаючих семафорів. Синтаксис:

```
int pthread_mutex_init (pthread_mutex_t *mutex,
            const pthread_mutexattr_t *attr);
```

Функція ініціалізує м'ютекс `mutex` атрибутом `mutexattr`. Якщо `mutexattr` дорівнює NULL, то м'ютекс ініціалізується значенням за замовчанням. У разі успішного виконання функції (код повернення 0), м'ютекс вважається ініціалізованим та «вільним».

Типові помилки, які можуть виникнути:

- EAGAIN – недостатньо необхідних ресурсів (крім пам'яті) для ініціалізації м'ютексу;
- ENOMEM – недостатньо пам'яті;
- EPERM – немає прав для виконання операції;
- EBUSY – спроба ініціалізувати м'ютекс, який вже був ініціалізований, але не знищений;
- EINVAL – значення `mutexattr` не вірно.

Приклад створення м'ютекса:

```
pthread_mutex_t lock;
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // завжди перевіряємо успіх
```

Якщо необхідно створити м'ютекс з атрибутами за замовчуванням, можна користуватися макросом PTHREAD_MUTEX_INITIALIZER:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Функції **pthread_mutex_lock** та **pthread_mutex_unlock** аналогічні семафорним операціям P і V відповідно. Синтаксис:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Функція **pthread_mutex_lock**, якщо `mutex` ще не зайнятий, то займає його, стає його власником і одразу виходить. Якщо м'ютекс зайнятий, то блокує подальше виконання процесу і чекає на звільнення м'ютексу.

Коди повернення для **pthread_mutex_lock**:

▪ EINVAL – м'ютекс неправильно ініціалізований;

▪ EDEADLK – м'ютекс вже зайнятий поточним процесом;

pthread_mutex_unlock звільняє зайнятий м'ютекс.

Коди повернення для **pthread_mutex_unlock**:

▪ EINVAL – м'ютекс неправильно ініціалізований;

▪ EPERM – процес, що викликає, не є власником м'ютексу.

Функція **pthread_mutex_trylock** аналогічна **pthread_mutex_lock** але при неможливості захопити семафор не переводить нитку в очікування, а повертає відповідну ознаку. Синтаксис:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Коди повернення для **pthread_mutex_trylock**:

▪ EBUSY – м'ютекс вже зайнятий

▪ EINVAL – м'ютекс неправильно ініціалізований

Після використання м'ютексу його необхідно знищити за допомогою функції **pthread_mutex_destroy**. Синтаксис:

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

При цьому, на момент виклику цієї операції знищуваний м'ютекс повинен бути вільний, тобто не захоплений жодною з ниток, у тому числі тою, що виконує цю операцію. Якщо м'ютекс був ініціалізований за допомогою макросу PTHREAD_MUTEX_INITIALIZER, то необхідність в його явному знищенні відсутня.

Приклад 15.4. Синхронна робота. Нитки працюють строго одна за одною. Яка почне працювати першою – отримає номер 1.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
```

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* trythis(void* arg)
{
    pthread_mutex_lock(&lock);
    counter += 1;
    printf("\n Job %d has started\n", counter);
    sleep(3);
    printf("\n Job %d has finished\n", counter);
    pthread_mutex_unlock(&lock);
    pthread_exit(NULL);
}

int main(void)
{
int i = 0;
int error;
if (pthread_mutex_init(&lock, NULL) != 0) {
   printf("\n mutex init has failed\n");
   return 1;
}
while (i < 2) {
   error=pthread_create(&(tid[i]),NULL,&trythis,NULL);
   if (error != 0)
        printf("\nThread can't be created :[%s]",
               strerror(error));
   i++;
}
pthread_join(tid[0], NULL);
pthread_join(tid[1], NULL);
```

```
pthread_mutex_destroy(&lock);
return 0;
}
```

Сигналізуючі семафори. Сигналізуючий семафор (умовна змінна, conditional variable) представляється в Linux-програмі змінною типу `pthread_cond_t`. Фактично такий семафор являє собою звичайний двійковий семафор з початковим значенням 0. Семафор `cond` повинен бути ініціалізований за допомогою функції **pthread_cond_init** з атрибутами `attr` (або NULL, якщо ініціалізація з атрибутами за замовчанням). Синтаксис:

```
int pthread_cond_init( pthread_cond_t *cond,
                       const pthread_condattr_t *attr);
```

Так само як і м'ютекс, ініціалізацію можна провести статичним ініціалізатором:

```
pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
```

Значення, що повертаються:

- EOK – успішне завершення;
- EAGAIN – усі об'єкти синхронізації в ядрі вже використовуються;
- EBUSY – об'єкт `cond` вже ініціалізований – перед повторною ініціалізацією необхідно його звільнити;
- EFAULT – при доступі до `cond` або `attr` з боку ядра виникла помилка;
- EINVAL – параметр `cond` неправильний;
- ENOMEM – недостатньо пам'яті для ініціалізації умовної змінної.

Інші базові функції (перераховано тільки найбільш часто вживані). Функція **pthread_cond_wait** реалізує на сигнальному семафорі семафорну

операцію P. Вона блокує нитку, що її викликала, доти, поки не буде отриманий сигнал про установку семафора в 1. Синтаксис:

```
int pthread_cond_wait( pthread_cond_t *cond,
                       pthread_mutex_t *mutex );
```

де `cond` – покажчик на умовну змінну, `mutex` – м'ютекс, який буде розблоковано.

Коди повернення для **pthread_cond_wait:**

▪ EAGAIN – недостатньо системних ресурсів для виконання очікування на умовну змінну;

▪ EFAULT – виникла помилка при доступі до наданих буферів;

▪ EINVAL – виконано одну або кілька умов: один або обидва примітиви синхронізації `cond` і `mutex` пошкоджені або не є дійсними; конкуруючі блокування в очікуванні умовної змінної `cond` використовують різні м'ютекси.

▪ EPERM – поточна нитка не володіє м'ютексом.

Функція **pthread_cond_signal** аналогічна семафорній операції V, вона розблокує одну з ниток, що очікують на семафорі. Якщо таких ниток немає, то нічого не відбувається. Якщо їх кілька, то розблокується одна, але це може бути будь-яка нитка. Синтаксис:

```
int pthread_cond_signal( pthread_cond_t *cond );
```

Значення, що повертаються:

▪ EOK – успішне завершення;

▪ EFAULT – при зверненні до буферів сталася помилка;

▪ EINVAL – умовна змінна `cond` не є коректною.

Функція **pthread_cond_broadcast** розблокує всі нитки, що очікують на семафорі. Значення, що повертаються – такі самі. Синтаксис:

```
int pthread_cond_broadcast( pthread_cond_t *cond );
```

Функція **pthread_cond_destroy** знищує умовну змінну `cond`. Після цього її повторне використання можливе тільки після чергової ініціалізації через **pthread_cond_init**. Синтаксис:

```
int pthread_cond_destroy( pthread_cond_t *cond );
```

Приклад 15.5. Створення нитки, яка блокується виконанням функції **pthread_cond_wait** над умовною змінною. У головній нитці виконується розблокування цієї змінної за допомогою функції **pthread_cond_signal** для продовження роботи нитки.

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

void* blockedThread()
{
    // займаємо м’ютекс lock
    pthread_mutex_lock(&lock);
    // виконуємо операцію P над cond
    printf("Waiting on cond\n");
    pthread_cond_wait(&cond, &lock);
    // звільняємо м’ютекс lock
    pthread_mutex_unlock(&lock);
    printf("Returning thread\n");
    pthread_exit(NULL);
}

int main()
{
```

```
    pthread_t tid;
    // створюємо нитку
    pthread_create(&tid, NULL, blockedThread, NULL);
    // затримка на 1 сек, даємо шанс
    // запуститись створеній нитці
    sleep(1);
    // виконуємо операцію V над cond
    printf("Signaling cond\n");
    pthread_cond_signal(&cond);
    // чекаємо на завершення створеної нитки
    pthread_join(tid, NULL);
    return 0;
}
```

Лічильники ресурсів. Лічильники ресурсів для ниток – це звичайні загальні семафори. Такий семафор представляється в програмі змінною типу `sem_t`. Такий семафор повинен бути ініціалізований за допомогою функції **sem_init**, а потім до нього можуть застосовуватися функції **sem_wait** та **sem_post**, виконуючи відповідно P- та V-операції.

Семафор ініціалізується за допомогою **sem_init** (для процесів або потоків) або **sem_open** (для IPC).

```
sem_init(sem_t *sem, int pshared, unsigned int value);
```

де `sem` – визначає семафор, який потрібно ініціалізувати; `pshared` – визначає, чи буде нещодавно ініціалізований семафор спільним для процесів або між нитками. Ненульове значення означає, що семафор використовується між процесами, а нульове значення означає, що він використовується між нитками; `value` – значення, яке потрібно призначити щойно ініціалізованому семафору.

Коди помилок:

- EAGAIN – ресурс, необхідний ініціалізації семафору, вичерпано;
- EBUSY – семафор був раніше ініціалізований та не був знищений;

▪ EINVAL – аргумент `value` перевищує SEM_VALUE_MAX;

▪ EPERM – процес немає прав для ініціалізації семафора;

▪ ENOSPC – ресурс, необхідний ініціалізації семафору, вичерпаний;

▪ ENOSYS – функція **sem_init** не підтримується.

Щоб заблокувати семафор або почекати, використовується функція **sem_wait**:

```
int sem_wait(sem_t *sem);
```

Коди помилок:

▪ EDEADLK – виявлено стан взаємного блокування;

▪ EINVAL – неприпустимий дескриптор `sem`;

▪ EINTR – виклик перервано сигналом.

Щоб звільнити або сигналізувати семафор, використовується функція **sem_post**:

```
int sem_post(sem_t *sem);
```

Коди помилок:

▪ EINVAL – неприпустимий дескриптор семафора `sem`;

▪ ENOSYS – функція **sem_post** не підтримується.

Щоб знищити семафор, використовується **sem_destroy**. Можлива помилка: EINVAL – неприпустимий дескриптор семафору `sem`. Синтаксис:

```
sem_destroy(sem_t * sem);
```

Функція **sem_getvalue** дозволяє довідатися про поточне значення семафора. Можлива помилка: EINVAL – неприпустимий дескриптор семафора `sem`. Синтаксис:

```
int sem_getvalue( sem_t *sem, int *value );
```

де `sem` – покажчик на семафор `sem_t`, значення якого потрібно отримати; `value` – покажчик на змінну, де функція може зберегти значення семафора (0 відповідає заблокованому семафору, позитивне значення – розблокованому). Це значення може змінитися будь-якої миті. Воно дійсне лише в момент виклику функції **sem_getvalue**.

Ця функція призначена лише для налагодження коду, який використовує семафор.

Приклад 15.6. Створення двох ниток з інтервалом в дві секунди. Кожна нитка має критичний код, де вона «працює» чотири секунди (емуляція за допомогою очікування). За допомогою семафорів захищається критичний код. Для відстеження станів семафора використовується функція **sem_getvalue**. Можна побачити, що друга нитка вимушена чекати приблизно дві секунди, перш ніж почати «працювати».

```
#include <stdio.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

sem_t sem;
int state;

void* thread(void* arg)
{
    // чекаємо на звільнення семафора
    // та одразу захоплюємо його
    sem_getvalue(&sem,&state);
    printf("\nsem value before CS=%d...\n",state);

    sem_wait(&sem);
    printf("\nEntered in CS..\n");

    sem_getvalue(&sem,&state);
```

```
    printf("\nsem value inside CS=%d...\n",state);

    // захищена (критична) секція
    // емуляція роботи зі спільними ресурсами
    sleep(4);

    // вихід з критичної секції
    // збільшуємо семафор на 1
    printf("\nExiting from CS...\n");
    sem_post(&sem);

    sem_getvalue(&sem,&state);
    printf("\nsem value after CS=%d...\n",state);

    pthread_exit(NULL);
}

int main()
{
    sem_init(&sem, 0, 1);

    sem_getvalue(&sem,&state);
    printf("\nsem value after init=%d...\n",state);

    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);

    sem_getvalue(&sem,&state);
    printf("\nsem value after work=%d...\n",state);
```

```
    sem_destroy(&sem);
    return 0;
}
```

15.2. Контрольні питання

1. Що станеться, якщо в тексті програми приклада 15.1 видалити один рядок з викликом **pthread_join**? Якщо обидва? Які при цьому ситуації можуть зрідка відбуватись?

2. Порівняйте поведінку роботи програми приклада 15.1 з поведінкою програми в прикладі 1.4 з теми «Процеси, нитки та волокна в Windows». Що спільного? Чому саме така поведінка при кількох запусках програми на виконання?

3. Програма в прикладі 15.4 працює в два рази довше ніж у прикладі 15.3. Поясніть, чому?

4. З якими проблемами можна зіштовхнутись при розробці програм на нитках?

5. Чим відрізняється виключаючий семафор (м'ютекс) від сигналізуючого семафора (умовної змінної)?

6. Чим відрізняється виключаючий семафор (м'ютекс) від загального семафора (лічильника ресурсів)?

7. Чим відрізняється сигналізуючий семафор (умовна змінна) від загального семафора (лічильника ресурсів)?

8. Яку операцію не можна виконати над семафором в ОС Windows на відміну від Linux?

15.3. Індивідуальні завдання

Необхідно вирішити одну з двох класичних задач на основі вказаних засобів синхронізації на нитках та семафорах-ниток.

І. Задача «виробники-споживачі». Має бути вирішена проблема роботи декількох ниток з одним буфером. Частина ниток – це «виробники»: у випадкові моменти часу вони виконують записування інформації в буфер. Друга частина ниток – це «споживачі»: у випадкові моменти часу вони зчи-

тують інформацію з буфера (після зчитування інформація в буфері губиться). Необхідно так організувати виконання ниток, щоб не було колізій при сумісній роботі «виробників» і «споживачів».

II. Задача «читачі-письменники». Є дані, які сумісно використовуються декількома нитками. Є декілька ниток, які тільки читають ці дані («читачі») і декілька ниток, які тільки записують дані або змінюють їх («письменники»). При цьому повинні враховуватись наступні вимоги:

▪ будь яке число «читачів» може одночасно читати дані;

▪ записувати дані в означений момент часу може тільки один «письменник».

▪ коли «письменник» записує дані, жодний «читач» не може їх у цей час читати.

Індивідуальні завдання наведені в таблиці 15.1.

Таблиця 15.1 – Індивідуальні завдання

Номер варіанту	Номер задачі	Механізм синхронізації	Додаткові умови
1	2	3	4
1	задача I	загальний семафор (лічильник ресурсів)	циклічний буфер
2	задача II	загальний семафор (лічильник ресурсів)	Перевага в «письменників»
3	задача I	виключаючий семафор (м'ютекс)	лінійний буфер
4	задача II	виключаючий семафор (м'ютекс)	перевага в «письменників»
5	задача I	сигналізуючий семафор (умовна змінна)	буфер – стек
6	задача II	сигналізуючий семафор (умовна змінна)	перевага в «письменників»
7	задача I	загальний семафор (лічильник ресурсів)	лінійний буфер

Продовження таблиці 15.1.

1	2	3	4
8	задача II	загальний семафор (лічильник ресурсів)	перевага в «читачів»
9	задача I	загальний семафор (лічильник ресурсів)	буфер – стек
10	задача II	виключаючий семафор (м'ютекс)	перевага в «читачів»
11	задача I	виключаючий семафор (м'ютекс)	циклічний буфер
12	задача II	виключаючий семафор (м'ютекс)	перевага в «письменників»
13	задача I	виключаючий семафор (м'ютекс)	буфер – стек
14	задача II	загальний семафор (лічильник ресурсів)	перевага в «читачів»
15	задача I	сигналізуючий семафор (умовна змінна)	лінійний буфер
16	задача II	виключаючий семафор (м'ютекс)	перевага в «читачів»
17	задача I	сигналізуючий семафор (умовна змінна)	циклічний буфер
18	задача II	виключаючий семафор (м'ютекс)	перевага в «читачів»

Перевага «читачів» або «письменників» визначається таким чином:

▪ перевага «читачів» – якщо є дозвіл на читання, «письменники» очікують, коли не буде жодного «читача» (нові «читачі» одразу ж починають читати);

▪ перевага «письменників» – як тільки «письменник» встановить ознаку бажання записувати, жоден новий «читач» не повинен починати читання (але необхідно дочекатися, поки всі «читачі» закінчать операцію читання).

16. СИГНАЛИ В ОС LINUX

Мета: освоєння способів використання сигналів для синхронізації роботи процесів в ОС Linux.

16.1. Теоретичні відомості

Сигнал або віртуальне переривання є повідомленням, що система посилає процесу або один процес посилає іншому. Коли процес одержує сигнал, виконання програми процесу переривається, і керування передається на підпрограму (функцію) – обробник сигналу. Після завершення обробника сигналу виконання перерваної програми відновлюється з тієї точки, на якій вона була перервана. Починаючи з ядра Linux версії 2.6 слід зазначити, що більшість сигналів переривають лише одну нитку, на відміну від попередньої практики переривання всієї програми.

В операційній системі передбачена велика кількість типів сигналів, але більшість із цих типів зарезервовано для системних цілей – це сигнали, які операційна система посилає процесу. Однак є й сигнали, якими процеси можуть обмінюватися між собою.

За замовчуванням реакція на більшість сигналів – припинення процесу, що одержав сигнал, тобто, якщо процес одержує сигнал, обробка якого в ньому не передбачена, то процес-одержувач сигналу завершується. Кожен сигнал має один із трьох можливих станів (сигнальних масок):

- для сигналу можна встановити власний обробник сигналів;
- для обробки сигналу можна використовувати стандартний обробник. Кожен сигнал має обробника за замовчуванням, яке він виконує. Наприклад, програма буде закрита обробником SIGINT за замовчуванням;
- сигнал можна не помітити – ігнорування сигналу.

При написанні програм обробники сигналів за замовчуванням призначаються бібліотекою С. Це означає, що навіть якщо ігнорувати сигнали, створене програмне забезпечення все одно оброблятиме їх і діятиме відповідно до встановлених бібліотекою обробників, як зазвичай.

Якщо процес перебуває в стані «добровільної» призупинки (викликаної, наприклад, виконанням системного виклику **sleep**), то одержання сиг-

налу «пробуджує процес від сну» – незалежно від того, у чому складалася обробка сигналу, системний виклик **sleep** закінчується негайно.

Деякі типи сигналів. Типи сигналів ідентифікуються числовими номерами, але при програмуванні часто використовуються символьні імена сигналів, які визначені в системних файлах. Наприклад:

```
#define SIGHUP 1  /* Зависання процесу */
#define SIGINT 2  /* Переривання процесу */
#define SIGQUIT 3 /* Вийти з процесу */
#define SIGILL 4  /* Недозволена інструкція */
#define SIGTRAP 5 /* Перехоплення трасування */
#define SIGABRT 6 /* Перервати */
```

Отримати повний перелік сигналів, що підтримує певна система можна за допомогою команди:

```
kill -l
```

В ОС Ubuntu 22.04.3, наприклад, зареєстровано більше 60 сигналів. Нижче наведені деякі найбільше часто вживані імена сигналів:

▪ SIGHUP – цей сигнал вказує на те, що керуючий термінал було закрито. HUP – це абревіатура, що означає «повісити трубку». Цей сигнал виходить, коли процес виконується з терміналу, і цей термінал раптово завершується;

▪ SIGINT – це сигнал, який створюється, коли користувач натискає Ctrl + C на клавіатурі;

▪ SIGQUIT – це сигнал, який генерується, коли користувач натискає Ctrl + \ на клавіатурі;

▪ SIGILL – сигнал для забороненої інструкції. Це сигнал переривання, який надсилає операційна система процесу, коли вона виявляє заборонені інструкції в програмі. Також викликається в тому випадку, коли процес завантажує пошкоджену динамічну бібліотеку;

▪ SIGABRT – сигнал переривання вказує на те, що було використано API функцію **abort()** у програмі. Використовується для завершення процесу;

▪ SIGFPE – виключення для чисел з плаваючою комою. Сигнал генерується операційною системою, коли процес викликає виключення;

▪ SIGPIPE – «зламана трубка» (broken pipe). Інформування про те, що дані, які процес відправляє в канал, не будуть прочитані через відсутність процесу-«читача» або невідкриття ним каналу зв'язку;

▪ SIGSEGV – сигнал виключення. Коли процес намагається отримати доступ до пам'яті, яка йому не належить, операційна система подає процесу такий сигнал;

▪ SIGALRM – сигнал тривоги, надісланий системною функцією **alarm()** до процесу. Системний виклик **alarm()** по суті діє як таймер, який дозволяє отримати SIGALRM через встановлений проміжок часу (існують також інші API таймерів, які є більш точними);

▪ SIGTERM – цей сигнал наказує процесу завершити роботу. Видачу цього сигналу, наприклад, містить у собі команда (не системний виклик!) **kill**. Мається на увазі, що процес, який одержав цей сигнал, повинен завершитися, однак процес може встановити ігнорування цього сигналу або призначити для нього власний обробник;

▪ SIGCHLD – інформує про те, що дочірній процес завершився або зупинився. Реакція на цей сигнал, установлена за замовчуванням, – ігнорування. Батьківський процес може не піклуватися про обробку цього сигналу, якщо тільки він не хоче використати його для синхронізації свого виконання з дочірнім процесом;

▪ SIGUSR1 і SIGUSR2 – два невизначені сигнали, які надаються на розсуд програміста. Ці сигнали можна використовувати для зв'язку чи синхронізації програмного забезпечення з іншими процесами;

▪ SIGKILL – цей сигнал приводить до завершення того процесу, що його одержав. Це єдиний сигнал, що не може ігноруватися й для якого не можна призначити власний обробник.

Визначені користувачем обробники сигналів. Процес може замінити обробник сигналу за замовчуванням для майже всіх сигналів (крім SIGKILL та SIGSTOP) власною функцією обробника.

Функція обробки сигналів може мати будь-яке ім'я, але повинна мати тип повернення `void` і один параметр `int`.

Наприклад:

```
void sigchld_handler(int sig);
```

Коли обробник сигналу виконується, параметр `sig`, який йому передається, є номером сигналу. Програміст може використовувати одну функцію обробки сигналів для обробки кількох сигналів. У цьому випадку в обробнику потрібно буде перевірити параметр, щоб визначити, який сигнал було надіслано. З іншого боку, якщо функція обробника сигналу обробляє лише один сигнал, немає потреби перевіряти параметр, оскільки це завжди буде необхідним номером сигналу.

Так як невідомо, в який момент виконання процесу буде викликана функція обробки сигналів, необхідно виявляти особливу обережність при зверненні до глобальних структур даних цієї функції. Для функцій, що обробляють нитки, існує ще одна важлива вимога – реентерабельність. Оскільки обробник сигналу може бути викликаний в будь-якій точці виконання програми (а при деяких умовах під час обробки одного сигналу може бути викликаний інший обробник сигналу), в обробниках повинні використовуватися функції, які задовольняють вимогам реентерабельності, тобто, можуть бути викликані в той час, коли вони вже викликані десь у іншій точці програми. Фактично, вимога реентерабельності зводиться до того, щоб функція не використовувала жодних глобальних ресурсів, не подбавши про синхронізацію доступу до цих ресурсів. Деякі функції введення-виведення, у тому числі, функція **printf()**, не є реентерабельними. Це означає, що виведення однієї функції **printf()** може перешкодити виведенню іншої функції. Нижче наведено список реентерабельних функцій, які безпечно викликати з обробників сигналів.

```
accept()            access()            aio_error()
aio_return()        aio_suspend()       alarm()
bind()              cfgetispeed()       cfgetospeed()
cfsetispeed()       fsetospeed()        chdir()
chmod()             chown()             clock_gettime()
close()             connect()           creat()
dup()               dup2()              execle()
execve()            _Exit()             _exit()
fchmod()            fchown()            fcntl()
fdatasync()         fork()              fpathconf()
fstat()             fsync()             ftruncate()
getegid()           geteuid()           getgid()
getgroups()         getpeername()       getpgrp()
getpid()            getppid()           getsockname()
getsockopt()        getuid()            kill()
link()              listen()            lseek()
lstat()             mkdir()             mkfifo()
open()              pselect()           posix_trace_event()
raise()             read()              readlink()
recv()              recvfrom()          recvmsg()
rename()            sendto()            setgid()
setpgid()           setsid()            setsockopt()
setuid()            shutdown()          sigaction()
sigaddset()         sigdelset()         sigemptyset()
sigfillset()        sigismember()       signal()
sigpause()          sigpending()        sigprocmask()
sigqueue()          sigset()            sigsuspend()
sleep()             socket()            socketpair()
stat()              symlink()           sysconf()
tcdrain()           tcflow()            tcflush()
tcgetattr()         tcgetpgrp()         tcsendbreak()
tcsetattr()         tcsetpgrp()         time()
timer_getoverrun()  timer_gettime()     times()
timer_settime()     umask()             uname()
unlink()            utime()             wait()
waitpid()           write()
```

Для встановлення власного обробника сигналу, для його скасування або для установки ігнорування сигналу використовується системний виклик **signal**. Він приймає два аргументи: посилання на код обробника сигналу та номер сигналу.

Синтаксис:

```
signal (signum, sig_handler);
```

Виклик **signal** дозволяє вказати обробник сигналу за замовчуванням, який буде використовуватися для певного сигналу. Також можна наказати системі ігнорувати певний сигнал: параметру `sig_handler` надати значення SIG_IGN. Для обробки за замовчанням параметру `sig_handler` можна надати значення SIG_DFL.

При надходженні сигналу з номером `signum` відбувається наступне: сигнал ігнорується, якщо для відповідного обробника встановлено значення SIG_IGN; дія сигналу за замовчуванням виконується, якщо обробник встановлено на SIG_DFL; нарешті, якщо обробником є функція під назвою `sig_handler`, `sig_handler` виконується після скидання обробника до SIG_DFL або блокування сигналу, залежно від реалізації.

Функція **signal** повертає SIG ERR у разі помилки, або повертає попереднє значення обробника сигналу.

Приклад 16.1. У програмі виконано перехоплення сигналів для завершення процесу SIGINT (натискання комбінації клавіш Ctrl + C) та SIGQUIT (натискання комбінації клавіш Ctrl + \). Слід приділити увагу, що використання функції **printf** всередині обробника не рекомендується (по причині, вказаній вище).

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
```

```
char buf1[]="Caught signal SIGINT\n";
char buf2[]="Caught signal SIGQUIT\n";

void handle_signal(int sig)
{
switch (sig){
      case SIGINT:
            write(1, buf1, sizeof(buf1)); break;
      case SIGQUIT:
            write(1, buf2, sizeof(buf2)); break;
}
}

int main()
{
signal(SIGINT, handle_signal);
signal(SIGQUIT, handle_signal);
while (1) ;
return 0;
}
```

Використання функції **signal** не рекомендується, тому що:

▪ функція не блокує отримання інших сигналів, поки виконується поточний обробник, він буде перерваний і почне виконуватися новий обробник (проблема з реентерабельністю);

▪ після першого отримання сигналу (для якого встановлено власний обробник), його обробник буде скинутий на SIG_DFL.

Слід дотримуватися наступних рекомендацій:

▪ повідомлення про помилки повинні виводитися системним викликом **write** (або іншою безпечною функцією), а завершення роботи застосування – системним викликом **_exit**;

▪ глобальні змінні, які змінюються усередині обробника, повинні мати тип **volatile sig_atomic_t**, а повернення в застосування повинно проводитись за умови, що для сигналу встановлено прапор `SA_RESTART`.

Крім того, цей системний виклик має проблему з переносимістю – він діє по-різному в різних операційних системах. Існує новіший системний виклик, який виконує всі функції **signal**, надаючи додатково трохи більше деталей про сам сигнал, його походження тощо:

```
int sigaction(int signum, const struct sigaction *act,
                  struct sigaction *oldact);
```

Номер сигналу вказується в його першому аргументі. Покажчики на структуру `sigaction` надаються як другий і третій аргументи. Ця структура описує, як даний сигнал має оброблятися процесом. Другий аргумент – це новий опис для сигналу, через третій повертається старе значення. Структура `struct sigaction` має такі поля:

`sa_handler` – аналог `sig_handler` у функції **signal**;

`sa_mask` – маска сигналів, які будуть блоковані поки виконується обробник (за замовчанням блокується і отриманий сигнал);

`sa_mask` – дозволяє задати додаткові дії при обробці сигналу.

Використання даної функції: встановлення обробника для сигналів SIGUSR1 і SUGUSR2, а також вказівка, що необхідно блокувати ці сигнали поки виконується обробник.

```
struct sigaction act;
memset(&act, 0, sizeof(act));
act.sa_handler = hdl;
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGUSR1);
sigaddset(&set, SIGUSR2);
act.sa_mask = set;
sigaction(SIGUSR1, &act, 0);
```

```
sigaction(SIGUSR2, &act, 0);
```

У наведеному фрагменті використані функції з групи функцій управління наборами сигналів:

▪ функція **sigemptyset** ініціалізує набір сигналів, що задається `set`, порожнім значенням, тобто всі сигнали виключені з набору;

▪ функція **sigfillset** ініціалізує `set` максимальним значенням, тобто всі сигнали входять до набору;

▪ функції **sigaddset** і **sigdelset**, відповідно, додають та видаляють сигнал `signum` з набору;

▪ функція **sigismember** перевіряє, чи `signum` є членом набору `set`.

Обробник сигналів встановлюється на весь процес і всі нитки, що в ньому породжені, одразу. Немає можливості для кожної нитки встановити свій обробник сигналів. Але слід розуміти, що коли сигнал адресується процесу, обробник викликається саме для головної нитки (що представляє процес). Якщо сигнал адресується для нитки, то обробник викликається з контексту цієї нитки.

Блокування сигналів. У процесі також є можливість призупинити обробку сигналів, що надходять, тимчасово, на період виконання будь-якої важливої операції. У традиційній термінології зупинка отримання певних сигналів називається блокуванням. Якщо для сигналу, що надійшов, було встановлено блокування, сигнал буде переданий процесу, як тільки він розблокує даний тип сигналів. Цим блокування відрізняється від ігнорування сигналу, у якому сигнали відповідного типу ніколи не передаються процесу.

Щоб заблокувати деякі сигнали для процесу, необхідно додати їх у маску сигналів даного процесу. Для цього використовується функція

```
int sigprocmask(int how, const sigset_t *set,
                  sigset_t *oldset);
```

Можна до вже існуючої маски сигналів додати нові сигнали (SIG_BLOCK), можна з цієї маски прибрати частину сигналів (SIG_UNBLOCK), а також повністю встановити маску сигналів (SIG_SETMASK).

Для роботи з маскою сигналів усередині нитки використовується функція

```
int pthread_sigmask(int how, const sigset_t *set,
                    sigset_t *oset);
```

яка дозволяє зробити такі самі дії, але вже для кожної нитки окремо.

За допомогою цих функцій неможливо заблокувати сигнали SIGKILL або SIGSTOP. Спроби це зробити ігноруватимуться.

Функція **sigwait** дозволяє призупинити виконання процесу (або нитки) до отримання потрібного сигналу (або одного з маски сигналів). Особливістю цієї функції є те, що при отриманні сигналу не буде викликано функції обробника сигналу.

Приклад 16.2. Використання функції **sigwait** та блокування сигналу `SIGINT` замість зайнятого циклу `while(1)`, в якому процес знаходився в прикладі 16.1.

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

int main() {
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigprocmask(SIG_BLOCK, &set, NULL);
printf("Just try to Ctrl-C!\n");
```

```
while (1) {
      int delivered;
      sigwait(&set, &delivered);
      printf("\nReceived signal %d: %s\n",
            delivered, strsignal(delivered));
      }
return 0;
}
```

Посилання сигналу. Процес може послати сигнал будь-якому іншому процесу, PID якого йому відомий, за допомогою системного виклику **kill**. У деяких випадках процесу буває потрібно послати сигнал самому собі, це можна зробити за допомогою системного виклику **raise**.

```
int kill (pid_t pid, int sig);
int raise (int sig);
```

Другий виклик по суті рівносильний

```
kill(getpid(), signal);
```

де **getpid()** повертає PID поточного процесу.

Для того, щоб надіслати сигнал до окремої нитки, використовується функція

```
int pthread_kill(pthread_t thread, int sig);
```

Приклад 16.3. Перехоплення сигналів SIGINT та SIGQUIT та посилання сигналів поточному процесу через виклики **kill** та **raise**.

```
#include<stdio.h>
#include<signal.h>
#include <unistd.h>
```

```
#include <stdlib.h>
#include <string.h>

char buf1[]="Caught signal SIGINT\n";
char buf2[]="Caught signal SIGQUIT\n";

void handle_signal(int sig)
{
switch (sig){
      case SIGINT:
            write(1, buf1, sizeof(buf1)); break;
      case SIGQUIT:
            write(1, buf2, sizeof(buf2)); break;
      }
}

int main() {
sigset_t set;
struct sigaction act;
memset(&act, 0, sizeof(act));
act.sa_handler = handle_signal;
sigemptyset(&set);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGQUIT);
act.sa_mask = set;
sigprocmask(SIG_UNBLOCK, &set, NULL);
sigaction(SIGINT, &act, 0);
sigaction(SIGQUIT, &act, 0);
kill(getpid(), SIGINT);
raise(SIGQUIT);
while(1);
return 0;
}
```

16.2. Контрольні питання

1. Чим відрізняється виклик **signal** від **sigaction**?

2. Що буде виведено в результаті виконання даної програми? Відповідь поясніть (в програмі для скорочення тексту використано виклик **signal** замість рекомендованого **sigaction**).

```
#include<wait.h>
#include<stdio.h>
#include<signal.h>
#include <unistd.h>
#include <stdlib.h>
volatile sig_atomic_t a = 5;

void handler(int sig) { a += 3; }

int main()
{ pid_t pid;
signal(SIGCHLD, handler);
if ((pid = fork()) == 0)
{     a -= 2;
      exit(0); }
waitpid(pid, NULL, 0);
printf("%d\n", a);
exit(0); }
```

3. Що буде виведено в результаті виконання даної програми? Відповідь поясніть (у програмі для скорочення тексту використано **signal** та функція **printf,** використання якої в реальних програмах всередині обробників сигналів не рекомендується).

```
#include<stdio.h>
#include<wait.h>
#include<signal.h>
```

```
#include <unistd.h>
#include <stdlib.h>
pid_t pid;
volatile sig_atomic_t counter = 0;
void handler1(int sig) {
counter++;
printf("counter = %d\n", counter);
fflush(stdout);
kill(pid, SIGUSR1); }

void handler2(int sig) {
counter += 3;
printf("counter = %d\n", counter);
_exit(0); }

int main() {
pid_t p;
int status;
signal(SIGUSR1, handler1);
if ((pid = fork()) == 0) {
      signal(SIGUSR1, handler2);
      kill(getppid(), SIGUSR1);
      while(1); }
if ((p = wait(&status)) > 0) {
      counter += 4;
      printf("counter = %d\n", counter); }
}
```

4. Чим відрізняється блокування від ігнорування сигналів?

5. Чим сигнал відрізняється від апаратного переривання? Від програмного переривання?

6. Як виглядає обробник сигналу?

16.3. Індивідуальні завдання

Розробити програми для виконання поставленого завдання. Для створення власних обробників сигналів бажано використовувати виклик **sigaction**.

Хід обміну сигналами та даними відображати на екрані. У тілі обробника сигналу використовувати мінімум програмного коду. Для виведення повідомлень на екран використовувати запис у стандартний потік виведення за допомогою функції **write**. Програмний код батьківського процесу та породжених процесів реалізувати в різних виконуваних файлах (крім варіанта 5).

Для індивідуальних завдань, що містять обчислення математичних виразів, модифікувати програми, розроблені за темою «Процеси в Linux»: процеси-«нащадки» чекають від батьківського процесу сигнал – дозвіл на початок розрахунку; батьківський процес чекає від процесів-«нащадків» сигнал – повідомлення про закінчення розрахунку. Остаточний результат розрахунку отримати в батьківському процесі лише після отримання сигналів від усіх процесів-«нащадків».

1) Обчислити для n (n>3) клієнтів 12 відсотків від заданої користувачем суми кредиту. У «нащадку» 1 обчислити 12-відсоткову виплату першої половини клієнтів [1 – n/2], а в «нащадку» 2 – для другої половини клієнтів [(n/2+1) – n].

2) Виконати породження завершення двох дочірніх процесів та перевірку їх статусу завершення з допомогою обробника сигналу SIGCHLD. Виконання «нащадка» 1 завершити з кодом 200, а виконання «нащадка» 2 – за допомогою сигналу, надісланого батьківським процесом після того, як батьківський процес отримав від другого «нащадка» сигнал – повідомлення про можливість завершення. Батьківський процес наприкінці своєї діяльності йде в цикл очікування, вихід із якого здійснити лише після завершення обох процесів-«нащадків» (реалізувати за допомогою сигналів).

3) Виконати породження «Сина», «Онука» та «Правнука». Кожен «нащадок» відправляє сигнал – повідомлення про народження своєму «батькові» та переходить у режим очікування, «батько», отримавши сигнал,

виводить повідомлення про народження «нащадка» і надсилає «нащадку» сигнал на завершення. «Нащадок» завершується лише після отримання сигналу SIGKILL з трьох можливих. У тілі кожного процесу визначити та вивести на екран інформацію про процес: ідентифікатор та покоління («батько» – 1 покоління, «син» – 2 покоління тощо).

4) Обчислити число поєднань $C(k,n)=n!/(k!\cdot|n-k|!)$, де $n>0$, $k>0$, $n\neq k$. Обчислення n!, k!, | n-k |! виконати в трьох процесах-«нащадках» відповідно.

5) Використовуючи сигнали, розробити програмну модель наступної ситуації. Пізно ввечері молодий «філософ» вивчає праці всесвітньо відомих філософів. Подивившись на годинник, «філософ» замислюється над тим, що вже час йти спати, оскільки завтра треба рано вставати, проте, йому хочеться продовжити вивчення. Так «філософ» перетворюється на стан протиріччя із собою: він нагадує собі, що час йти спати, то продовжує вивчення. Тільки після п'ятого нагадування він погоджується з тим, що треба закінчувати вивчення та йде спати. Всі дії виконати в рамках одного файла, що виконується.

6) Обчислити щільність нормального розподілу в точці x за формулою $f(x)=Exp(-x^2/2)/Sqrt(2\cdot x)$, де $x>0$. Обчислення експоненти та квадратного кореня виконати в двох процесах-«нащадках» відповідно.

7) Обчислити щільність опуклого розподілу в точці x за формулою $f(x)=(1-Cos(x))/(x^2)$, де $x>0$. Обчислення $Cos(x)$ і x^2 виконати в двох процесах-«нащадках» відповідно.

8) Обчислити значення функції $f(x)=((-1)^x+x^{(2x-1)})/(2x-1)!$, де $x>0$. Обчислення $(-1)^x$, $x^{(2x-1)}$, $(2x-1)!$ виконати в трьох процесах-«нащадках» відповідно.

9) Обчислити значення функції $f(x,k)=(x(2k+1)+x(5k-1))/(2k+1)$. Обчислення $x(2k+1)$ і $x(5k-1)$ виконати в двох процесах-«нащадках» відповідно.

10) Розробити програмну модель за допомогою обміну сигналами між процесами. Батько має трьох синів, яких він «відправив на прогулянку». Через деякий час батько просить кожного сина завершити прогулянку, проте сини відповідають відмовою протягом часу, що залежить від віку (що молодший син, тим він швидше відповідає). Отримавши відмову, батько

повторює своє прохання, однак його діти погоджуються з ним лише з третього разу. Якщо за певний час якийсь із синів не погодився з батьком, останній перестає вважати його своєю дитиною.

11) Визначити, сума цифр якого цілого числа більша від заданих трьох. Підрахунок суми цифр кожного числа виконати в окремому процесі-«нащадку».

12) Здійснити пошук максимального з восьми чисел за допомогою пошуку більшого з двох чисел. Пошук більшого із двох для кожної пари чисел виконати в окремому процесі-«нащадку».

13) Обчислити площу кільця за значеннями внутрішнього та зовнішнього радіусів, використовуючи функцію обчислення площі кола. Обчислення площі кола з внутрішнім та зовнішнім радіусом виконати в двох процесах-«нащадках» відповідно.

14) Виконати обмін сигналами між батьківським та дочірнім процесами за наступною схемою. «Батько» після народження посилає «нащадку» сигнал на завершення SIGTERM, проте «нащадок» ігнорує його. Через деякий час «батько» перевіряє статус «нащадка» і якщо той не завершився, посилає сигнал SIGINT. «Нащадок», отримавши сигнал SIGINT, не завершується, а надсилає «батькові» сигнал SIGUSR1 – відмова від завершення. Отримавши сигнал SIGUSR1, «батько» примусово завершує «нащадка».

15) Виконати породження трьох дочірніх процесів. Кожен із «нащадків» виконується заданий «батьком» випадковий час (реалізувати за допомогою сигналу SIGALRM). Про завершення «нащадка» «батько» дізнається за допомогою сигналу SIGCHLD. Якщо через деякий час «батько» не отримав від «нащадка» сигнал SIGCHLD, він відправляє «нащадку» сигнал на завершення, проте, «нащадок» реагує на цей сигнал лише з другого разу.

17. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ НЕІМЕНОВАНИХ КАНАЛІВ В ОС LINUX

Мета: освоєння неіменованих програмних каналів як способу обміну даними між процесами в ОС Linux.

17.1. Теоретичні відомості

Програмний канал в UNIX/Linux являє собою один із засобів взаємодії між процесами. Сама назва (pipe, дослівно – трубка) досить точно передає зміст функціонування цього засобу. Канал подібний до трубопроводу, прокладеному між двома процесами, і по цьому трубопроводі процеси можуть пересилати один одному дані. Подібно трубопроводу, канал має власну ємність, дані, спрямовані в канал процесом-відправником, не обов'язково повинні бути негайно прочитані процесом-одержувачем, але можуть накопичуватися в каналі. Як і в трубопроводі, ємність каналу кінцева, коли вона буде вичерпана, запис у канал стає неможливим: або заблокується або завершиться з помилкою – залежно від налаштувань. У різних реалізаціях ОС різні обмеження на ємність каналу. Програми не повинні покладатися на певну величину: їх потрібно розробляти так, щоб процес, що читає, переробляв дані як тільки вони з'являються, щоб процес, який розміщує дані в канал, не блокувався.

Довідкова інформація: до Linux 2.6.11 ємність каналу дорівнювала розміру системної сторінки (наприклад, 4096 байтів на i386). Починаючи з Linux 2.6.11, ємність каналу становить 16 сторінок (тобто 65536 байтів у системі з розміром сторінки 4096 байтів). Починаючи з Linux 2.6.35, ємність каналу за замовчуванням становить 16 сторінок, але ємність можна змінити. У будь-якому випадку розмір однієї порції інформації не перевищує 4096 байтів (одна сторінка).

Так само як в ОС Windows, системи UNIX/Linux дозволяють з'єднувати потоки `stdout` команди з `stdin` іншої команди (конвеєр). Для того, щоб це зробити, використовується символ « | ». Конвеєр використовується для об'єднання двох або більше команд, і в цьому випадку вихідні дані однієї команди діють як вхідні дані для іншої команди, а вихідні дані

цієї команди можуть діяти як вхідні дані для наступної команди і так далі. Конвеєр також можна візуалізувати як тимчасовий зв'язок між двома чи більше командами/програмами/процесами. Програми командного рядка, які виконують подальшу обробку, називаються фільтрами.

Цей прямий зв'язок між командами/програмами/процесами дозволяє їм працювати одночасно та дозволяє безперервно передавати дані між ними замість того, щоб передавати їх через тимчасові файли або через екран дисплея. Канали є односпрямованими, тобто дані проходять зліва направо через конвеєр. Синтаксис:

```
команда_1 | команда_2 | команда_3 | .... | command_N
```

Операційні системи UNIX/Linux надають у розпорядження програмістів два види каналів – іменовані й неіменовані. Робота з обома видами багато в чому подібна до роботи з файлами.

Неіменовані канали. Неіменований канал є засобом взаємодії між зв'язаними процесами – батьківським і дочірнім. Батьківський процес створює канал за допомогою системного виклику:

```
int pipe(int fd[2]);
```

Масив із двох цілих чисел є вихідним параметром цього системного виклику. Якщо виклик виконався нормально, то цей масив містить два файлових дескриптори. `fd[0]` є дескриптором для читання з каналу, `fd[1]` – дескриптором для запису в канал (рис. 17.1):

```
int fd[2];
pipe(fd);
```

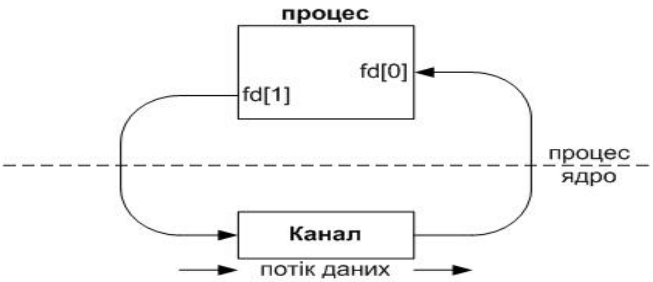


Рисунок 17.1 – Канали ріре

Системний виклик **pipe** знаходить перші дві доступні позиції в таблиці відкритих файлів процесу та виділяє їх для дескрипторів каналу: читання та запису.

Коли процес породжує інший процес, дескриптори батьківського процесу успадковуються дочірнім процесом, і, таким чином, прокладається «трубопровід» між двома процесами (рис. 17.2):

```
pid_t childpid;
childpid = fork();
```

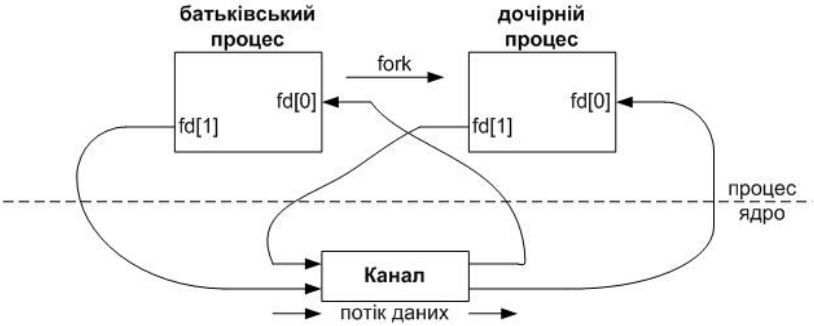


Рисунок 17.2 – Наслідування каналів

Природно, що один із процесів використовує канал тільки для читання, а інший – тільки для запису. Тому, якщо, наприклад, через канал по-

винні передаватися дані з батьківського процесу в дочірній, батьківський процес одразу після запуску дочірнього процесу закриває дескриптор каналу для читання, а дочірній процес закриває дескриптор для запису (рис. 17.3):

```
if (childpid == 0)     { /* дочірній процес */
      close(fd[1]);
     ...
    } else {/* батьківський процес */
      close(fd[0]);
     ...
}
```

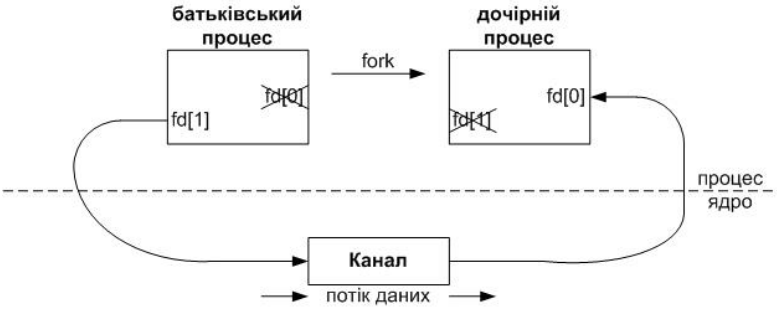


Рисунок 17.3 – Односпрямований канал

Якщо потрібний двоспрямований обмін даними між процесами, то батьківський процес створює два канали, один з яких використовується для передачі даних в одну сторону, а другий – в іншу (рис. 17.2).

Після одержання процесами дескрипторів каналу для роботи з каналом використовуються файлові системні виклики:

```
int read(int pipe_fd, void *area, int cnt);
int write(int pipe_fd, void *area, int cnt);
```

Перший аргумент цих викликів – дескриптор каналу, другий – покажчик на область пам'яті, з якою відбувається обмін, третій – кількість байтів. Обидва виклики повертають число переданих байтів або -1 – при помилці.

Виконання цих системних викликів може переводити процес у стан очікування. Це відбувається, якщо процес намагається читати дані з порожнього каналу або писати дані в переповнений канал. Процес виходить із очікування, коли в каналі з'являються дані або коли в каналі з'являється вільне місце, відповідно.

При завершенні використання каналу процес виконує системний виклик:

```
int close(int pipe_fd);
```

Якщо батьківський процес, що створив канал, породжує кілька дочірніх процесів, то всі дочірні процеси будуть підключені до іншого кінця каналу. Якщо, наприклад, батьківський процес виводить дані в канал, то вони «дістануться» тому дочірньому процесу, що раніше виконає системний виклик **read**.

Приклад 17.1. Батьківський процес створює односпрямований канал, записує в нього два рядки. Дочірній процес зчитує рядки з каналу. Обидва процеси реалізовані в одній програмі.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";

int main()
{
```

```
  char inbuf[MSGSIZE];
  int p[2], pid, nbytes;
  if (pipe(p) < 0) exit(1);
  if ((pid = fork()) > 0) {
     // батьківський процес
     close(p[0]);
     write(p[1], msg1, MSGSIZE);
     write(p[1], msg2, MSGSIZE);
     close(p[1]);
     wait(NULL);
  }
  else {
     // дочірній процес
     close(p[1]);
     while ((nbytes = read(p[0], inbuf, MSGSIZE)) > 0)
         printf("%s\n", inbuf);
     if (nbytes != 0) exit(2);
     close(p[0]);
     printf("Finished reading\n");
  }
  return 0;
}
```

17.2. Контрольні питання

1. Чим відрізняється робота з неіменованими каналами в ОС Linux по зрівнянню з анонімними каналами в ОС Windows?

2. Що станеться, якщо в прикладі 17.1 не закрити дескриптор каналу `p[1]` у батьківському процесі? Чому?

3. Яким чином забезпечити двоспрямовану передачу даних через неіменовані канали?

4. Чому бажано одразу виконувати читання даних з каналу, як тільки вони там з’являються?

5. Яка класична задача з синхронізації процесів може бути легко вирішена на каналах?

6. Для чого використовуються канали? Яка перевага від їх використання в порівнянні з використанням функції копіювання з пам'яті в пам'ять всередині одного процесу?

17.3. Індивідуальні завдання

Організувати передачу рядків символів, що вводяться з клавіатури, між процесами 1, 2, 3, …, N за заданою схемою (символ ➔ означає напрямок передачі даних) з використанням неіменованих каналів. Передбачити можливість передачі будь-якої кількості рядків. Завершення роботи програм виконувати за командою з клавіатури. Автоматичне переспрямування означає миттєву передачу в канал щойно прийнятого рядка символів з іншого каналу. Допускається перетворення даних програмами за власним розсудом. Якщо вказані одразу кілька номерів процесів після символу ➔, це означає, що передачу одних і тих же даних необхідно виконати до всіх вказаних процесів. Для вирішення індивідуального завдання рекомендується розробити програму-менеджер запуску процесів, яка запускається першою, у ній створюються всі необхідні канали та виконується запуск всіх інших програм, яким передаються дескриптори необхідних каналів в якості параметрів запуска. Для кожної програми передавача та приймача на екрані обов'язково відображувати крім самих даних також схему передачі даних (наприклад, «1➔2: переданий рядок» для програми передавача 1, та «1➔2: прийнятий рядок» для програми приймача 2).

У кожній програмі прийом та передача даних повинні бути реалізовані в окремих нитках: наприклад, якщо програма 2 отримує дані від програми 1 та програми 3, та надсилає дані в програму 1, то в ній потрібно реалізувати окрему нитку для читання даних з каналу від 1 програми, окрему нитку для читання даних з каналу від 3 програми та окрему нитку запису в канал для 1 програми.

Кожна програма повинна бути в окремому виконуваному файлі (обов'язкове використання для створення процесів пари викликів **fork** та **exec**).

Для додаткових операцій з синхронізації використовувати механізм сигналів.

1) 1➔4; 2➔4; 3➔4
2) 1➔2; 2➔3; 3➔4
3) 1➔2,3; 2➔1,3; 3➔1,2
4) 1➔2,3,4; 3➔2; 4➔1
5) 1➔2; 2➔3; 3➔4 (автоматичне переспрямування)
6) 1➔2; 2➔1
7) 1➔2,3; 2➔3; 3➔1
8) 1➔4; 2➔4; 3➔4
9) 1➔2,3; 2➔3 (автоматичне переспрямування)
10) 1➔2,3; 2➔1 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)
11) 1➔2,3,4; 2➔3,4; 3➔4
12) 1➔2,3; 2➔1,3; 3➔1,2
13) 1➔2; 3➔4; 2➔3 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)
14) 1➔2,3,4 (автоматичне переспрямування); 2➔1; 3➔1; 4➔1
15) 1➔2; 2➔3; 3➔2 (автоматичне переспрямування)
16) 1➔3; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування)
17) 1➔2,3; 2➔1,3 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)
18) 1➔2,3,4; 2➔1; 3➔1; 4➔1
19) 1➔2,3,4; 4➔2,3,4
20) 1➔2; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

18. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ ІМЕНОВАНИХ КАНАЛІВ В ОС LINUX

Мета: освоєння іменованих програмних каналів як способу обміну даними між процесами в ОС Linux.

18.1. Теоретичні відомості

Іменовані канали (в UNIX їх іноді називають FIFO) можуть використовуватись як засіб взаємодії між неспорідненими та навіть віддаленими процесами. Такий канал має зовнішнє ім'я, що включається в простір імен файлової системи. Тому іменований канал ще більш схожий на файл, чим неіменований. У системі канал представляється спеціальним файлом і створюється спеціальним системним викликом:

```
int mknod(char *name, int mode, int dev);
```

Цей системний виклик може використовуватися також і для створення звичайних файлів, каталогів і інших спеціальних файлів. Параметр `name` цього виклику є покажчиком на символьний рядок, що містить ім'я каналу (ім'я може містити в собі також і шлях). Параметр `mode` визначає тип створюваного файла та режим доступу до нього. Старші 7 біт цього числа визначають тип створюваного файла (для іменованого каналу він може кодуватися макроконстантою S_IFIFO, молодші 9 біт визначають права доступу «rwx» для власника (старша трійка), для групи (середня трійка) та для всіх інших (молодша трійка). Так, наприклад, для каналу, що буде доступний тільки для власника, код параметра `mode` буде S_IFIFO|0x140, а для каналу, доступного для «всіх-всіх-всіх» – S_IFIFO|0x1B6. (Природно, право «x» для каналу не визначається). Третій параметр при створенні каналу задається 0.

Параметр `mode` можна також задати за допомогою констант:

- S_IRUSR, S_IRGRP, S_IROTH – читання для власника, групи та всіх інших відповідно;
- S_IWUSR, S_IWGRP, S_IWOTH – запис для власника, групи та всіх інших відповідно.

У сучасних системах також можна скористатись викликом, який на відміну від **mknod** простіше (**mknod** дозволяє створювати не тільки FIFO, але й каталоги, якщо вказати параметр S_IFDIR, та спеціальні блочні та символьні пристрої), та є стандартизованим та переносимим:

```
int mkfifo(const char *name, mode_t mode);
```

Ідентифікатор власника FIFO при цьому встановлюється рівним ефективному ідентифікатору користувача процесу, а ідентифікатор групи FIFO встановлюється рівним ефективному ідентифікатору групи процесу.

Також FIFO можна створити не тільки програмно, але й за допомогою команди для створення файлів (FIFO – теж файл) або окремої команди **mkfifo**:

```
mknod ім'я_файла p
mkfifo ім'я_файла
```

Далі при роботі з іменованим каналом використовуються файлові системні виклики:

```
int open(int *name, int oflag);
int read(int pipe_fd, void *area, int cnt);
int write(int pipe_fd, void *area, int cnt);
int close(int pipe_fd);
```

Необхідно звернути увагу на те, що при відкритті файла-каналу можуть бути задані прапорці відкриття, серед яких може бути прапорець **O_NDELAY**. Якщо іменований канал відкритий з цим прапорцем, то процес, що працює з іменованим каналом, не переходить в очікування в тих випадках, які призводять до призупинки процесу, що працює з неіменованим каналом, – замість цього системні виклики **read** та **write** закінчуються з ознакою помилки.

Якщо кілька процесів пишуть у той самий FIFO, необхідно звернути увагу на те, щоб одразу не записувалося більше, ніж PIPE_BUF байтів. Це необхідно, щоб дані не змішувалися один з одним.

Іменований канал є постійним об'єктом, він зберігається навіть після завершення процесу, що його створив, і при необхідності такий канал повинен знищуватися явно – за допомогою системного виклику:

```
int unlink(char *name);
```

Фактично, іменований канал є правильним замінником для обміну даними через тимчасові файли, оскільки обмін через файли володіє наступними недоліками:

- використання повільного диска замість більш швидкої пам'яті;
- витрата місця на диску (у той час як при обміні даними через FIFO після зчитування вони видаляються); більш того, місце на диску може закінчитися;
- у процесу може не бути прав створити файл або ж файл може бути зіпсований/видалений іншим процесом.

Деяку складність може являти синхронізація роботи процесів з каналами. При цьому потрібно враховувати наступні обставини:

- якщо процес пробує відкрити канал для читання, але канал ще не відкритий для запису, процес переводиться в стан очікування;
- якщо процес пробує відкрити канал для запису , але канал ще не відкритий для читання, процес переводиться в стан очікування;
- системний виклик відкриття не існуючого (ще не створеного) каналу завершується з помилкою;
- спроба читання з закритого каналу та запису в закритий канал викликає виключну ситуацію **«Broken pipe»** та призводить до аварійного завершення процесу.

При наявності двох (чи більше) процесів та двох каналів, можлива ситуація класичного тупику, яка показана на рисунку 18.1.

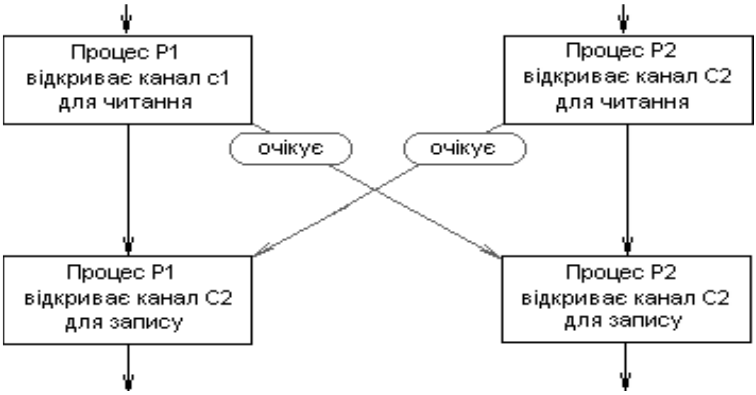


Рисунок 18.1 – Тупикова ситація при роботі з каналами

Приклад 18.1. Перша програма створює іменований канал, створює другий процес та записує в канал рядок. Після цього канал видаляється. Друга програма **l5_2** відкриває канал та читає з нього рядок. У програмах не реалізовані жодні додаткові засоби синхронізації.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <string.h>
#include <fcntl.h>
char* msg1 = "hello, world #1";

int main()
{
mknod("pipe1",S_IFIFO|S_IRUSR|S_IWUSR,0);
pid_t pw2 = fork();
if (pw2==0)
{
```

```
      if (execl("./l5_2"," ",NULL)<0)
      {
            perror("./l5_2");
            exit(0);
      }
} else {
      int pipe1 = open("pipe1",O_WRONLY);
      write(pipe1,msg1,strlen(msg1));
      printf("\nSent: %s\n",msg1);
      unlink("pipe1");
      }
return(0);
}
```

Текст програми ***l5_2****.*

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/stat.h>

int main()
{
int pipe1 = open("pipe1",O_RDONLY|O_NDELAY);
char buffer[BUFSIZ];
read(pipe1,&buffer,BUFSIZ);
printf("\nReceived: %s\n",buffer);
return(0);
}
```

Деякі особливості використання каналів FIFO та неіменованих каналів:

1. при читанні меншого числа байтів, ніж у каналі, повертається необхідне число байтів, залишок зберігається для наступних читань;

2. при читанні більшого числа байтів, ніж у каналі, повертається доступне число байтів. Процес, який читає з каналу, повинен відповідним чином обробити ситуацію, коли прочитано менше, ніж замовлено;

3. якщо канал порожній та жоден процес не відкрив його на запис, під час читання з каналу буде отримано 0 байтів. Якщо один або більше процесів відкрили канал для запису, виклик **read** буде заблокований до появи даних (якщо не встановлено прапор відсутності блокування O_NDELAY);

4. запис числа байтів, менше ніж ємність каналу, гарантовано атомарний. Це означає, що в разі коли кілька процесів одночасно записують у канал, порції даних від цих процесів не перемішуються;

5. при запису більшої кількості байтів, ніж це дозволяє канал, виклик **write** блокується до звільнення потрібного місця. У цьому випадку атомарність операції не гарантується. Якщо процес намагається записати дані в канал, не відкритий жодним процесом на читання, процесу генерується сигнал SIGPIPE, а виклик **write** повертає 0 із встановленням помилки (`errno`=EPIPE) (якщо процес не встановив обробку сигналу SIGPIPE – проводиться обробка за умовчанням, і процес завершується).

18.2. Контрольні питання

1. Чим відрізняються іменовані канали від неіменованих?

2. Порівняйте роботу з іменованими каналами в ОС Windows та Linux.

3. Що відбудеться, якщо в прикладі 18.1 видалити рядок з викликом **unlink** з тексту першої програми? Чи можна буде повторно запустити програми без помилки етапу виконання?

4. Чи можна реалізувати конвеєр команд на іменованих каналах?

5. Опишіть основні кроки по вирішенню задачі «виробники-споживачі» на іменованих каналах.

18.3. Індивідуальні завдання

Організувати передачу рядків символів, що вводяться з клавіатури, між процесами 1, 2, 3, …, N за заданою схемою (символ ➔ означає напрямок передачі даних) з використанням іменованих каналів. Передбачити можливість передачі будь-якої кількості рядків. Завершення роботи програм виконувати за командою з клавіатури. Автоматичне переспрямування означає миттєву передачу в канал щойно прийнятого рядка символів з іншого каналу. Допускається перетворення даних програмами за власним розсудом. Якщо вказані одразу кілька номерів процесів після символу ➔, це означає, що передачу одних і тих же даних необхідно виконати до всіх вказаних процесів. Для вирішення індивідуального завдання рекомендується розробити програму-менеджер запуску процесів, яка запускається першою, у ній створюються всі необхідні канали та виконується запуск всіх інших програм, яким передаються імена необхідних каналів в якості параметрів запуску. Для кожної програми передавача та приймача на екрані обов'язково відображувати крім самих даних також схему передачі даних (наприклад, «1➔2: переданий рядок» для програми передавача 1, та «1➔2: прийнятий рядок» для програми приймача 2).

У кожній програмі прийом та передача даних повинні бути реалізовані в окремих нитках: наприклад, якщо програма 2 отримує дані від програми 1 та програми 3, та надсилає дані в програму 1, то в ній потрібно реалізувати окрему нитку для читання даних з каналу від 1 програми, окрему нитку для читання даних з каналу від 3 програми та окрему нитку запису в канал для 1 програми.

Кожна програма повинна бути в окремому виконуваному файлі (обов'язкове використання для створення процесів пари викликів **fork** та **exec**).

Для додаткових операцій з синхронізації використовувати механізм сигналів.

1) 1➔4; 2➔4; 3➔4
2) 1➔2; 2➔3; 3➔4
3) 1➔2,3; 2➔1,3; 3➔1,2

4) 1➔2,3,4; 3➔2; 4➔1

5) 1➔2; 2➔3; 3➔4 (автоматичне переспрямування)

6) 1➔2; 2➔1

7) 1➔2,3; 2➔3; 3➔1

8) 1➔4; 2➔4; 3➔4

9) 1➔2,3; 2➔3 (автоматичне переспрямування)

10) 1➔2,3; 2➔1 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

11) 1➔2,3,4; 2➔3,4; 3➔4

12) 1➔2,3; 2➔1,3; 3➔1,2

13) 1➔2; 3➔4; 2➔3 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

14) 1➔2,3,4 (автоматичне переспрямування); 2➔1; 3➔1; 4➔1

15) 1➔2; 2➔3; 3➔2 (автоматичне переспрямування)

16) 1➔3; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування)

17) 1➔2,3; 2➔1,3 (автоматичне переспрямування); 3➔1 (автоматичне переспрямування)

18) 1➔2,3,4; 2➔1; 3➔1; 4➔1

19) 1➔2,3,4; 4➔2,3,4

20) 1➔2; 2➔3 (автоматичне переспрямування); 3➔4 (автоматичне переспрямування); 4➔1 (автоматичне переспрямування)

19. СИНХРОНІЗАЦІЯ ПРОЦЕСІВ ЗА ДОПОМОГОЮ СЕМАФОРІВ В ОС LINUX

Мета: освоєння семафорів як способу комунікацій між процесами в ОС Linux.

19.1. Теоретичні відомості

У теорії операційних систем семафор являє собою невід'ємну цілу змінну, над якою можливі два види операцій: P та V.

▪ P-операція над семафором являє собою спробу зменшення значення семафора на 1. Якщо перед виконанням P-операції значення семафора було більше 0, P-операція виконується без затримок. Якщо перед виконанням P-операції значення семафора було 0, процес, що виконує P-операцію, переводиться в стан очікування доти, поки значення семафора не стане більшим 0.

▪ V-операція над семафором являє собою збільшення значення семафора на 1. Якщо при цьому є процеси, затримані на виконанні P-операції на цьому семафорі, один із цих процесів виходить із стану очікування і може виконати свою P-операцію.

Семафори є гнучким і зручним засобом для синхронізації і взаємного виключення процесів, обліку ресурсів, сховані семафори також використаються в операційних системах як основа для інших засобів взаємодії процесів. Як одна з форм міжпроцесної взаємодії (IPC), семафори не призначені для обміну великими обсягами даних, як у випадку FIFO або черг повідомлень. Натомість вони виконують функцію, що повністю відповідає своїй назві – дозволяти або забороняти процесу використання того чи іншого ресурсу, що розділяється.

Для нормальної роботи необхідно забезпечити виконання таких умов:

1) значення семафора має бути доступне різним процесам. Тому семафор перебуває не в адресному просторі процесу, а в адресному просторі ядра;

2) операції перевірки та зміни значення семафора повинні бути реалізовані у вигляді однієї атомарної по відношенню до інших процесів опера-

ції. В іншому випадку можлива ситуація, коли після перевірки значення семафора виконання процесу буде перервано іншим процесом, який в свою чергу перевірить семафор та змінить його значення.

Єдиним способом гарантувати атомарність критичних ділянок операцій є виконання цих операцій в режимі ядра. Таким чином, семафори є системним ресурсом, дії над яким здійснюються через інтерфейс системних викликів.

Працюючи з семафорами взаємодіючі процеси мають «домовитися» про їх використання і кооперативно проводити операції над семафорами. Операційна система не накладає обмежень використання семафорів. Зокрема, процеси вільні вирішувати, яке значення семафора є вирішальним, на яку величину змінюється значення семафора тощо.

Особливості реалізації семафорів у System V. IPC-семафори ввійшли в її будову як невід'ємна частина. Слід зазначити, що набір операцій над IPC-семафорами в System V відрізняється від класичного набору операцій {P, V}, запропонованого Дейкстрою. Він включає три операції:

- A(S, n) – збільшити значення семафора S на величину n;
- D(S, n) – поки значення семафора S < n, процес блокується. Далі S = S - n;
- Z(S) – процес блокується доти, поки значення семафора S не стане рівним 0.

Всі IPC-семафори ініціалізуються нульовим значенням. Тобто класичній операції P(S) відповідає операція D(S,1), а класичній операції V(S) відповідає операція A(S,1). Аналогом ненульової ініціалізації значенням n семафорів Дейкстри може служити виконання операції A(S,n) одразу після створення семафора S із забезпеченням атомарності створення семафора і її виконання за допомогою іншого семафора. Отже, класичні семафори реалізуються через IPC-семафори System V. Зворотна дія не є вірною. Використовуючи операції P(S) і V(S), ми не зможемо реалізувати операцію Z(S).

Взаємне виключення на семафорі. Для реалізації взаємного виключення, наприклад, запобігання можливості одночасної зміни двома або бі-

льше процесами загальних даних, створюється двійковий (з можливими значеннями 0 і 1) семафор S. Початкове значення цього семафора – 1. Критичні секції коду (секції, які можуть одночасно виконуватися тільки одним процесом) обрамляються «дужками» P(S) (на початку секції) і V(S) (наприкінці секції). Процес, що входить у критичну секцію, виконує операцію P(S) і переводить семафор у 0. Якщо в критичній секції вже перебуває інший процес, то значення семафора вже 0, тоді другий процес, що бажає ввійти в критичну секцію, блокується в своїй P-операції доти, поки процес, що перебуває в критичній секції зараз, не вийде з неї, виконавши на виході операцію V(S).

Синхронізація на семафорі. Для забезпечення синхронізації створюється двійковий семафор S з початковим значенням 0. Значення 0 означає, що подія ще не наступила. Процес, що сигналізує про настання події, виконує операцію V(S), що встановлює семафор в 1. Процес, що очікує настання події, виконує операцію P(S). Якщо до цього моменту подія вже відбулася, процес, що очікує, продовжує виконуватися, якщо ж подія ще не відбулася, процес переводиться в стан очікування доти, поки процес, що сигналізує, не виконає V(S).

Семафор як лічильник ресурсів. Якщо є N одиниць деякого ресурсу, то для контролю його розподілу створюється загальний семафор S з початковим значенням N. Виділення ресурсу супроводжується операцією P(S), звільнення – операцією V(S). Значення семафора, таким чином, відображає число вільних одиниць ресурсу. Якщо значення семафора 0, тобто вільних одиниць більше не залишається, то черговий процес, що запитує одиницю ресурсу буде переведено в очікування в операції P(S) доти, поки який-небудь із процесів, що використовують ресурс, не звільнить одиницю ресурсу, виконавши при цьому V(S).

Таким чином, при роботі з семафорами процеси використовують різні комбінації операцій, визначених системою, трактуючи значення семафорів.

Системні виклики UNIX/Linux. В ОС UNIX/Linux механізм семафорів обслуговується трьома системними викликами: **semget** (скорочення від semaphore-get), **semctl** і **semop**.

З метою економії системних ресурсів операційна система Linux дозволяє створювати не по одному семафору для кожного конкретного значення ключа, а зв'язувати з ключем цілий масив семафорів (в ОС Linux – до 500 семафорів у масиві, хоча ця кількість може бути зменшена системним адміністратором).

Але основною причиною для запровадження масових операцій над семафорами було прагнення дати програмістам можливість уникати тупикових ситуацій в зв'язку із семафорною синхронізацією. Це забезпечується тим, що системний виклик **semop**, яким би довгим він не був (через потенційно необмежену довжину масиву семафорів), виконується як атомарна операція, тобто під час виконання **semop** жоден інший процес не може змінити значення будь-якого семафора з масиву.

Системний виклик **semget** створює масив семафорів або повертає ідентифікатор вже існуючого масиву семафорів. Цей ідентифікатор використовується при подальших операцій з семафорами. Системні виклики працюють з масивами семафорів, це зроблено лише для того, щоб мати загальну ідентифікацію для всіх семафорів однієї задачі. Системні виклики **semctl** і **semop** дають можливість окремо оперувати з кожним семафором масиву.

```
int semget(key_t key, int nsems, int semflg);
```

Семафори в UNIX/Linux не мають зовнішніх імен. При отриманні ідентифікатора семафора процес користується числовим ключем `key`. Розробники незв'язаних процесів можуть домовитися про загальне значення ключа, який вони будуть використовувати, але в них немає гарантії в тому, що це ж значення ключа не буде використане кимось ще. Як значення цього параметра може використовуватися значення ключа, отримане за допомогою функції **ftok**, або спеціальне значення IPC_PRIVATE – гарантовано унікальний масив семафорів, але такий ключ не може бути зовнішнім – він не може бути отриманий за допомогою функції **ftok** ні при одній комбінації

її параметрів. Тому семафори використовуються, як правило, родинними процесами, які мають можливість передавати один одному ідентифікатори семафорів, наприклад, через наслідувані ресурси або через параметри виклику дочірньої програми.

Параметр `nsems` визначає кількість семафорів у створюваному або вже існуючому масиві. У випадку, якщо масив із зазначеним ключем уже є, але його розмір не збігається із зазначеним у параметрі `nsems`, констатується виникнення помилки.

Параметр `semflg` – прапорці – відіграє роль тільки при створенні нового масиву семафорів і визначає права різних користувачів при доступі до масиву, а також необхідність створення нового масиву і поведінку системного виклику при спробі створення. Він є комбінацією прав доступу та наступних визначених значень:

▪ IPC_CREAT – якщо масиву для зазначеного ключа не існує, він повинен бути створений;

▪ IPC_EXCL – застосовується разом із прапорцем IPC_CREAT. При спільному їхньому використанні і існуванні масиву із зазначеним ключем, доступ до масиву не відбувається і констатується помилка, при цьому змінна **errno** прийме значення EEXIST.

Права доступу мають наступні комбінації:

▪ 0400 – дозволене читання для користувача, що створив семафор;

▪ 0200 – дозволений запис для користувача, що створив семафор;

▪ 0040 – дозволене читання для групи користувача, що створив семафор;

▪ 0020 – дозволений запис для групи користувача, що створив семафор;

▪ 0004 – дозволене читання для всіх інших користувачів;

▪ 0002 – дозволений запис для всіх інших користувачів.

Загальна поведінка всіх викликів множини «**get**» (**semget**, **shmget** та **msgget**) для IPC визначається правилами.

При використанні ключа IPC_PRIVATE: так як нащадкам доступний вже готовий дескриптор створеного об'єкта, вони можуть безпосередньо працювати з ним, не звертаючись до виклику «**get**». Таким чином, створе-

ний ресурс може спільно використовуватись батьківським та породженими процесами. Однак, важливо розуміти, що якщо один з цих процесів повторно зробить виклик «**get**» з ключем IPC_PRIVATE, у результаті буде отримано інший, абсолютно новий ресурс, що розділяється, так як при зверненні до «**get**» з ключем IPC_PRIVATE щоразу створюється новий об'єкт потрібного типу.

Якщо під час звернення до «**get**» вказаний ключ, відмінний від IPC_PRIVATE, відбувається таке:

- відбувається пошук об'єкта з заданим ключем серед уже існуючих об'єктів необхідного типу. Якщо об'єкт з вказаним ключем не знайдено, і серед прапорців вказано прапорець IPC_CREAT, буде створено новий об'єкт. При цьому значення параметра прапорців має містити побітове додавання прапорця IPC_CREAT і константи, що вказує права доступу для об'єкта, який створюється;
- якщо об'єкт із заданим ключем не знайдений, і серед переданих прапорців відсутній прапорець IPC_CREAT, «**get**» поверне -1, а змінній `errno` буде встановлено значення ENOENT;
- якщо об'єкт із заданим ключем знайдено серед існуючих, «**get**» поверне дескриптор для цього об'єкта, тобто фактично відбувається підключення до вже існуючого об'єкту по заданому ключу. Якщо процес очікував створення нового об'єкта за вказаним ключем, то для нього така поведінка може виявитися небажаною, оскільки це означає, що в результаті випадкового збігу ключів він підключився до чужого ресурсу. Щоб уникнути такої ситуації, слід вказати в параметрі прапорців поряд з прапорцем IPC_CREAT та правами доступу ще й прапорець IPC_EXCL – у цьому випадку «**get**» поверне -1, якщо об'єкт із таким ключем вже існує (змінна `errno` буде встановлена в значення EEXIST).

Слід зазначити, що при підключенні до об'єкта, що вже існує, додатково перевіряються права доступу до нього. У випадку, якщо процес, що запросив доступ до об'єкта, не має на те прав, «**get**» поверне -1, а змінній `errno` буде встановлено значення EACCESS.

Як було вказано вище – для генерації ключа можна скористатись викликом функції **ftok**, який дозволяє з певною ймовірністю згенерувати майже унікальний ключ:

```
key_t ftok(char *filename, char proj);
```

Ця функція генерує значення ключа за деяким рядком символів `filename` і додатковим символом `proj`, що передаються як параметри. Гарантується, що отримане таким чином значення відрізнятиметься від усіх інших значень, згенерованих функцією **ftok** з іншими значеннями параметрів, і в той же час, при повторному запуску **ftok** з тими ж параметрами, буде отримано те саме значення ключа. Сенс другого аргументу функції **ftok** – додаткового символу – у тому, що він дозволяє генерувати різні значення ключа по тому самому значенню першого параметра – рядка. Це дозволяє програмісту підтримувати кілька версій своєї програми, які будуть використовувати один і той же рядок, але різні додаткові символи для генерації ключа, і тим самим отримають можливість у рамках однієї системи працювати з різними ресурсами, що розділяються. Слід зазначити, що функція **ftok** не є системним викликом, а надається бібліотекою.

Системний виклик **semctl** дозволяє виконувати керуючі операції над масивом семафорів й окремих його елементів: читати й встановлювати значення, знищувати масив семафорів.

```
int semctl(int semid, int semnum, int cmd, arg);
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
   } arg;
```

Параметр `semid` є дескриптором для масиву семафорів, тобто значенням, яке повернув системний виклик **semget** при створенні масиву або при

його пошуку за ключем. Аргумент `semnum` задає номер семафора в масиві. Семафори нумеруються з нуля.

Призначення аргументу `arg` залежить від дії, яка визначається значенням аргументу `cmd`. Допустимі наступні дії:

- GETVAL – отримати значення семафора і видати його як результат;
- SETVAL – встановити значення семафора, що дорівнює `arg.val`;
- GETPID – отримати ідентифікатор процесу, що останнім виконував операцію над семафором, і видати його як результат;
- GETNCNT – отримати кількість процесів, що очікують збільшення значення семафора, і видати її як результат;
- GETZCNT – отримати кількість процесів, що очікують обнулення значення семафора, і видати її як результат;
- GETALL – прочитати значення всіх семафорів масиву та помістити їх у масив, на який вказує `arg.array`;
- SETALL – встановити значення всіх семафорів масиву рівними значенням елементів масиву, на який вказує `arg.array`;
- IPC_STAT – помістити інформацію про стан масиву семафорів, що міститься в структурі даних, асоційованій з ідентифікатором `semid`, у структуру користувача, на яку вказує `arg.buf`;
- IPC_SET – у структурі даних, асоційованій з ідентифікатором `semid`, переустановлення значення діючих ідентифікаторів користувача та групи, а також прав на операції. Потрібні значення витягуються із структури даних, на яку вказує `arg.buf`;
- IPC_RMID – видалити із системи ідентифікатор `semid`, ліквідувати масив семафорів та асоційовану з ним структуру даних.

Щоб виконати керуючу дію IPC_SET або IPC_RMID, процес повинен мати діючий ідентифікатор користувача, що дорівнює ідентифікаторам творця або власника масиву, або ідентифікатору суперкористувача. Для виконання дій SETVAL і SETALL потрібно право на зміну, а для виконання інших дій – право на читання.

Системний виклик **semop** виконує прикладні семафорні операції: аналоги P- і V-операцій, а також перевірки стану семафора.

```
int semop (int semid, struct sembuf *sops,
           unsigned int nsops);
struct sembuf {
   short sem_num;
   short sem_op;
   short sem_flg;
   };
```

Як аргумент `semid` повинен виступати ідентифікатор масиву семафорів, попередньо отриманий за допомогою системного виклику **semget**. Аргумент `sops` (масив структур) визначає, над якими семафорами виконуватимуться операції та які саме. Номер семафора `sem_num` задає конкретний семафор у масиві, над яким має бути виконана операція. Операція `sem_op`, що виконується, визначається наступним чином:

▪ позитивне значення наказує збільшити значення семафора на величину `sem_op`;

▪ негативне значення визначає зменшення семафора на абсолютну величину `sem_op`. Операція не може бути успішно виконана, якщо в результаті вийде негативне число;

▪ нульове значення наказує порівняти значення семафора з нулем. Операція не може бути успішно виконана, якщо значення семафора відмінне від нуля.

Допустимі значення прапорців операцій (поле `sem_flg`):

▪ IPC_NOWAIT – якщо будь-яка операція, для якої заданий прапорець IPC_NOWAIT, не може бути успішно виконана, системний виклик завершується невдачею, причому в жодного з семафорів не буде змінено значення;

▪ SEM_UNDO – задає перевірочний режим виконання операції; він наказує анулювати її результат навіть у разі успішного завершення системного виклику **semop**. Інакше кажучи, блокування всіх операцій (зокрема й тих, котрим заданий прапор SEM_UNDO) виконується звичайним чином,

але коли нарешті всі операції можуть бути успішно виконані, операції з прапором SEM_UNDO ігноруються.

Аргумент `nsops` специфікує кількість структур у масиві. Максимально допустимий розмір масиву визначається системним параметром SEMOPM, тобто в кожному системному виклику **semop** можна виконати не більше операцій SEMOPM.

Приклад 19.1 Моделювання доступу 3 процесів до спільного ресурсу в монопольному режимі (імітація доступу в критичній секції, що обмежено дужками початку (P) та кінця (V)). Батьківський та всі дочірні процеси реалізовані в одному програмному коді. Як допоміжні засоби використовуються сигнали та семафори ниток.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <string.h>
#include <semaphore.h>

int cnt=0;
sem_t sem;
union semun {
      int val;
      struct semid_ds *buf;
      unsigned short *array;
      #if defined(__linux__)
          struct seminfo *__buf;
      #endif
      } arg;
```

```
// обробка сигналу, чекаємо на завершення нащадків
void handle_signal(int sig)
{
cnt++;
if (cnt==3) {
    // всі нащадки завершено – розблокуємо sem
    sem_post(&sem);
   }
}

// реалізація операції P
int p (int semid)
{
struct sembuf p_buf[1];
p_buf[0].sem_num = 0;
p_buf[0].sem_op = -1;
p_buf[0].sem_flg = 0;
if (semop (semid, p_buf, 1)==-1)
   {
   printf("\nError p(semid)\n");
   perror("semop");
   exit(EXIT_FAILURE);
   }
return(0);
}

// реалізація операції V
int v (int semid)
{
struct sembuf v_buf[1];
v_buf[0].sem_num = 0;
v_buf[0].sem_op = 1;
```

```
v_buf[0].sem_flg = 0;
if (semop (semid, v_buf, 1)==-1)
   {
   printf ("\nError v(semid)\n");
   perror("semop");
   exit(EXIT_FAILURE);
   }
return(0);
}

// створення або відкриття масиву семафорів з
// ключем semkey
int initsem(key_t semkey)
{
int status, semid;

status = 0;
// спроба створити масив з 1 семафора
semid = semget(semkey, 1,
               0600 | IPC_CREAT | IPC_EXCL);
if (semid == -1)
   {
   if (errno == EEXIST)
      {
      // масив вже існує – відкриваємо його
      semid = semget (semkey, 1, 0);
      }
   }
   else
   {
      // якщо тільки створили масив – встановлюємо
      // значення семафора в 1
      unsigned short val[1] = {1};
```

```
      arg.array = val;
      status = semctl(semid, 0, SETALL, arg);
   }
if ((semid == -1) || (status == -1))
   {
   printf ("\nError semget\n");
   exit(EXIT_FAILURE);
   }
return(semid);
}
// основна функція дочірніх процесів
void worker (key_t skey)
{
int semid, pid;
pid = getpid();
// створюємо або відкриваємо масив семафорів
// (з 1 елементу - нам більше не потрібно)
semid = initsem(skey);
if (semid < 0) {  exit(1); }
printf("-->Process %d try to enter in CS\n", pid);
// спроба входу до критичної секції
p(semid);
printf("+++Process %d entered to CS\n", pid);
printf("***Process %d working in CS\n", pid);
          // емуляція роботи з
sleep(3); // ресурсом у монопольному
          // режимі
printf("<--Process %d leave CS\n", pid);
// вихід з критичної секції
v(semid);
printf("   Process %d after CS\n", pid);
// надсилаємо сигнал батьківському процесу
// про закінчення своєї роботи
```

```
kill(getppid(),SIGUSR1);
exit(0);
}

int main(int argc, char *argv[])
{
int semid;
// створюємо семафор нитки в блокованому стані
sem_init(&sem, 0, 0);
// реєструємо обробник сигналу
signal(SIGUSR1,handle_signal);
// формуємо ключ для семафора з назви програми та 1
key_t semkey=ftok(argv[0], 1);
printf("\nSTART\n");
// запуск 3 процесів нащадків
for(int i=1; i<=3; i++)
 {
   if (fork() == 0) {
   worker(semkey);
   }
 }
// чекаємо на розблокування sem та знищуємо його
sem_wait(&sem);
sem_destroy(&sem);
printf("FINISH\n");
// отримуємо semid (користуємось можливостями
// розробленої функції для процесів-нащадків)
semid = initsem(semkey);
// видаляємо з системи масив семафорів з semid
semctl(semid, 0, IPC_RMID , arg);
exit(0);
}
```

У процесі розробки та налагодження програм можуть виникати ситуації, коли програма буде аварійно закінчуватися або перериватися перш, ніж вона знищить створені нею семафори (в наведеному прикладі 19.1 за це відповідає передостанній рядок коду з викликом **semctl**). Такі семафори не видаляються в системі автоматично й можуть накопичуватися протягом багатьох днів. Нагромадження таких «забутих» семафорів може привести до того, що буде вичерпаний системний ліміт на кількість семафорів, і черговий виклик **semget** закінчиться з помилкою. Для того, щоб цього не відбувалося, необхідно регулярно виконувати процедуру очищення IPC за допомогою команди для семафора зі вказаним ідентифікатором `semaphoreid` масиву

```
ipcrm -s <semaphoreid>
```

або для видалення всіх семафорів

```
ipcrm -asem
```

Список створених семафорів можна побачити, виконавши наступну команду:

```
ipcs -s
```

19.2. Контрольні питання

1. Що відбудеться, якщо видалити з прикладу 19.1 передостанній рядок коду з викликом **semctl**? Які зміни відбудуться в системі та яким чином їх можна виправити?

2. Чим відрізняється робота з семафорами в Linux та Windows?

3. З якою метою в Linux створюється та виконується робота одразу з масивом семафорів на відміну від створення та роботи з семафорами поодинці в ОС Windows?

4. Назвіть, які операції над семафорами однакові в ОС Linux та Windows, а які відрізняються?

5. Чим відрізняється результат створення семафора в ОС Linux та Windows?

19.3. Індивідуальні завдання

Розробити комплекс програм, за допомогою яких організовано обмін даними між батьківським процесом і дочірніми процесами відповідно до індивідуального завдання за допомогою спільного сегменту пам'яті, що розділяється, з урахуванням наступних вимог:

- організувати синхронізовану роботу процесів за допомогою семафорів;
- хід роботи процесів відображати на екрані;
- батьківський процес та дочірні процеси повинні бути реалізовані в окремих виконуваних файлах.

В роботі необхідно виконати розробку основи комплексу з імітацією обміну даними. Реалізація самого процесу обміну – виконується в темі «Обмін даними між процесами за допомогою спільних ділянок пам'яті в ОС Linux».

1) Обчислити для n (n>3) клієнтів 12 процентів від заданої користувачем суми кредиту. У «нащадку» 1 обчислити 12-відсоткову виплату першої половини клієнтів [1 – n/2], а «нащадку» 2 – для другої половини клієнтів [(n/2+1) – n].

2) «Нащадок» розміщує дані в спільну область пам'яті, «батько» – читає (1 письменник, 1 читач). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

3) «Батько» розміщує дані в спільну область пам'яті, «нащадки» – забирають дані, очищуючи пам'ять (1 виробник, 2 споживача). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

4) Виконати породження двох «нащадків». У кожному з процесів визначити значення свого і батьківського ідентифікатора. Від кожного «бать-

ка» виконати передачу повідомлення «Hello, Son! My Father ID =, My ID =, Your ID = !» своєму «нащадку».

5) «Нащадок» розміщує дані в спільну область пам'яті, «батько» – забирає дані, очищуючи пам'ять (1 виробник, 1 споживач). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

6) Обчислити щільність нормального розподілу в точці x за формулою $f(x)=Exp(-x^2/2)/Sqrt(2\cdot x)$, де $x>0$. Обчислення експоненти та квадратного кореня виконати в двох процесах-«нащадках» відповідно.

7) Обчислити щільність опуклого розподілу в точці x за формулою $f(x)=(1-Cos(x))/(x^2)$, де $x>0$. Обчислення $Cos(x)$ і x^2 виконати в двох процесах-«нащадках» відповідно.

8) «Батько» розміщує дані в спільну область пам'яті, «нащадки» – читають (1 письменник, 2 читача). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

9) Обчислити значення функції $f(x,k)=(x^{(2k+1)}+x^{(5k-1)})/(2k+1)$. Обчислення $(2k+1)$ і $x^{(5k-1)}$ виконати в двох процесах-«нащадках» відповідно.

10) «Нащадки» розміщують дані в спільну область пам'яті, «батько» – читає (2 письменника 1 читач). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

11) «Нащадки» розміщують дані в спільну область пам'яті, «батько» – забирає, очищуючи пам'ять (2 виробника 1 споживач). Операції читання та запису – асинхронні, багаторазові та з різною довжиною за часом. Для моделювання асинхронної роботи користатися затримкою з випадковою тривалістю.

12) Два дочірніх процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через один і той же сегмент пам'яті, що розділяється, батьківському процесу черговий рядок деякого вірша, при цьому перший процес передає непарні рядки, другий – парні. Цикли робіт процесів не збалансовані за часом. Батьківський процес компонує з фраг-

ментів, що передаються, закінчений вірш і виводить його після завершення роботи дочірніх процесів.

13) Чотири дочірні процеси виконують деякі цикли робіт, передаючи після закінчення чергового циклу через один і той же сегмент пам'яті, що розділяється, батьківському процесу черговий рядок деякого вірша, при цьому перший процес передає 1-й, 5-й, 9-й і т.д. рядки, другий – 2-й, 6-й, 10-й і т.д. рядки, третій – 3-й, 7-й, 11-й і т.д. рядки, четвертий – 4-й, 8-й, 12-й і т.д. рядки. Цикли робіт процесів не збалансовані за часом. Батьківський процес компонує з переданих фрагментів закінчений вірш і виводить його після завершення роботи всіх дочірніх процесів.

14) Батьківський процес розміщує в сегмент пам'яті імена програм з попередніх лабораторних робіт, які можуть бути запущені. Виконуючи деякі цикли робіт, породжені процеси випадково вибирають імена програм з таблиці сегмента, пам'яті що розділяється, запускають ці програми, і продовжують свою роботу. За допомогою апарату семафорів має бути забезпечено, щоб не було одночасно запущено дві однакові програми.

15) Розв'язати задачу з варіанта 14 за умови, що має бути забезпечено, щоб не було одночасно запущено дві програми від одного процесу.

20. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ СПІЛЬНИХ ДІЛЯНОК ПАМ’ЯТІ В ОС LINUX

Мета: освоєння спільних ділянок пам’яті як способу комунікацій між процесами в ОС Linux.

20.1. Теоретичні відомості

В операційних системах UNIX і Linux є можливість створювати сегменти пам’яті, доступні декільком процесам. При використанні цього засобу ділянки віртуального адресного простору різних процесів відображаються на ті самі адреси реальної пам’яті.

При роботі зі спільним сегментом пам’яті один із процесів повинен створити сегмент поділюваної (спільної) пам’яті, указавши необхідний розмір сегмента, і одержати його ідентифікатор. Цей ідентифікатор потім використовується процесом, що створив цей сегмент, для виконання керуючих операцій з поділюваною пам’яттю.

Одержавши ідентифікатор поділюваного сегмента пам’яті (створивши сегмент або одержавши його ідентифікатор від іншого процесу), процес повинен «приєднати» сегмент. Операція приєднання повертає адресу поділюваного сегмента у віртуальному адресному просторі процесу. Віртуальна адреса того самого сегмента може бути різною для різних процесів. Далі процеси ведуть читання й запис даних у сегменті поділюваної пам’яті, використовуючи для цього ті ж самі машинні команди, що й при роботі зі звичайною пам’яттю.

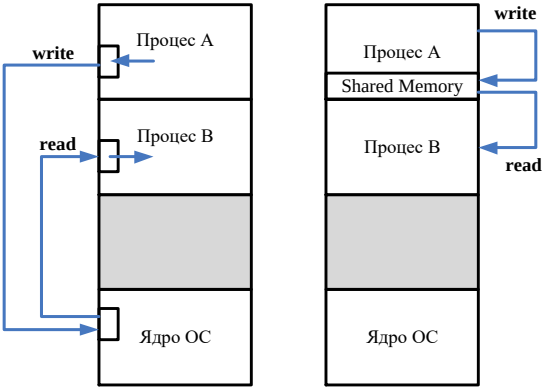
Процес, що закінчив роботу із сегментом поділюваної пам’яті повинен «від’єднати» його, а по закінченні використання сегмента всіма процесами – він повинен бути знищений.

При одночасній роботі процесів зі спільною областю пам’яті, можливо, потрібна синхронізація їхнього доступу до області. За забезпечення такої синхронізації повністю відповідає програміст, що може використати для цього алгоритми, що виключають конфлікти одночасного доступу або системні засоби, наприклад, семафори.

Основна проблема при роботі з неіменованими каналами, FIFO та чергами повідомлень (черги повідомлень розглядаються в темі «Обмін даними між процесами за допомогою черг повідомлень в ОС Linux») полягає в тому, що коли два процеси обмінюються інформацією між собою, ця інформація має проходити через ядро ОС (рис. 20.1 (а)), наприклад, коли необхідно передати дані з файла сервера до файла клієнта:

- сервер читає дані з вхідного файла;
- сервер записує ці дані в повідомлення за допомогою неіменованого каналу, FIFO або черги повідомлень (**write**);
- клієнт зчитує дані з каналу IPC, знову вимагаючи копіювання даних із буфера IPC ядра в буфер клієнта (**read**);
- нарешті, дані копіюються з буфера клієнта.

Всього потрібно робити чотири копії даних (2 для читання та 2 для запису). На відміну від цих механізмів, спільна пам'ять дозволяє двом або більше процесам спільно використовувати сегмент пам'яті (рис. 20.1 (б)): при такій самій передачі даних, але за допомогою спільної пам'яті, дані копіюються лише двічі – із вхідного файла в спільну пам'ять і зі спільної пам'яті у вихідний файл.



а) б)

Рисунок 20.1 – Канали, FIFO, черги повідомлень (а) та спільні сегменти (б)

Системні виклики UNIX/Linux. В ОС UNIX/Linux механізм поділюваних сегментів пам'яті забезпечується чотирма системними викликами: **shmget**, **shmctl**, **shmat**, **shmdt**.

Системний виклик **shmget** створює поділюваний сегмент або повертає ідентифікатор уже існуючого сегмента. Цей ідентифікатор використається при подальших операціях із сегментом.

```
int shmget (key_t key, int size, int shmemflg);
```

Аргументи цього виклику: `key` – ключ для доступу до пам'яті, що розділяється; `size` задає розмір області пам'яті, до якої хоче отримати доступ. Якщо в результаті виклику **shmget** буде створено нову область пам'яті, що розділяється, то її розмір буде відповідати значенню `size`. Якщо ж процес підключається до існуючої області пам'яті, що розділяється, то значення `size` має бути не більше її розміру, інакше виклик поверне -1. Зауважимо, що якщо процес при підключенні до існуючої області пам'яті вказав в аргументі `size` значення, менше її фактичного розміру, то згодом він зможе отримати доступ тільки до перших `size` байтів цієї області.

Зазначимо, що в заголовному файлі `<sys/shm.h>` визначено константи SHMMIN і SHMMAX, що задають мінімально можливий і максимально можливий розмір області пам'яті, що розділяється. Якщо процес намагається створити область пам'яті, що розділяється, розмір якої не задовольняє цим межам, системний виклик **shmget** закінчиться невдачею.

Третій параметр `shmemflg` визначає прапорці, які керують поведінкою виклику (комбінація наступних прав доступу та прапорців IPC_CREAT та IPC_EXCL):

- 0400 – дозволене читання для користувача, що створив сегмент;
- 0200 – дозволений запис для користувача, що створив сегмент;
- 0040 – дозволене читання для групи користувача, що створив сегмент;
- 0020 – дозволений запис для групи користувача, що створив сегмент;
- 0004 – дозволене читання для всіх інших користувачів;

▪ 0002 – дозволений запис для всіх інших користувачів.

Докладніше алгоритм створення/підключення ресурсу сумісного ресурсу за допомогою множини викликів «**get**» було описано в темі «Синхронізація процесів за допомогою семафорів в ОС Linux».

У разі успішного завершення виклик повертає позитивне число – дескриптор області пам'яті, у разі невдачі -1.

Системний виклик **shmctl** дозволяє виконувати керуючі операції над сегментом: одержувати інформацію про його стан, змінювати права доступу до нього, знищувати сегмент.

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Цей виклик використовується для отримання або зміни процесом керуючих параметрів, пов'язаних з областю пам'яті, що розділяється, накладання і зняття блокування на неї і її знищення. Аргументи виклику – дескриптор області пам'яті `shmid`, команда `cmd`, яку потрібно виконати, та структура `buf`, яка описує керуючі параметри області пам'яті. Тип `shmid_ds` описаний в заголовному файлі <sys/shm.h>, і є структурою, у полях якої зберігаються права доступу до області пам'яті, її розмір, кількість процесів, приєднаних до неї в даний момент, і статистика звернень до області пам'яті:

```
struct shmid_ds {
  struct ipc_perm shm_perm;    /* володар і права */
  size_t shm_segsz;   /* розмір сегменту в байтах*/
  time_t shm_atime;   /* час останнього підключення */
  time_t shm_dtime;   /* час останнього відключення */
  time_t shm_ctime;   /* час останньої зміни */
  pid_t shm_cpid;     /*PID того,хто створив сегмент*/
  pid_t shm_lpid;     /* PID останнього, хто виконував
                         shmat/shmdt */
  shmatt_t shm_nattch; /* поточна кіл-ть підключень */
  ... };
```

Структура `ipc_perm` визначена наступним чином (значення полів встановлюються за допомогою команди `cmd` IPC_SET):

```
struct ipc_perm {
  key_t __key;    /* ключ, що передається в shmget */
  uid_t uid;      /* діючий UID володаря */
  gid_t gid;      /* діючий GID володаря */
  uid_t cuid;     /* діючий UID творця */
  gid_t cgid;     /* діючий GID творця */
  unsigned short mode; /* права + прапорці SHM_DEST і
                          SHM_LOCKED */
  unsigned short __seq;/* порядковий номер */
};
```

Можливі значення аргументу `cmd`:

▪ IPC_STAT – скопіювати структуру, що описує керуючі параметри області пам'яті за адресою, вказаною в параметрі `buf`;

▪ SHM_STAT (є тільки в Linux) – повертає структуру `shmid_ds` як в операції IPC_STAT. Але аргумент `shmid` є не ідентифікатором сегмента, а індексом у внутрішньому масиві ядра, в якому міститься інформація про всі спільні сегменти пам'яті в системі;

▪ IPC_SET – замінити структуру, яка описує керуючі параметри області пам'яті на структуру, що знаходиться за адресою, вказаною в параметрі `buf`. Виконати цю операцію може процес, в якого діючий ідентифікатор користувача збігається з власником чи творцем, або процес з правами привілейованого користувача, при цьому процес може змінити лише власника області пам'яті та права доступу до неї;

▪ IPC_RMID – видалити сегмент. Видалити сегмент може лише процес, у якого діючий ідентифікатор користувача збігається з власником чи творцем, або процес із правами привілейованого користувача. Сегмент буде видалено тільки після того, як від нього відключиться останній процес

(тобто коли поле shm_nattch з пов'язаної структури shmid_ds буде рівним нулю);

▪ SHM_LOCK, SHM_UNLOCK – блокувати або розблокувати область пам'яті. Є тільки в Linux. Виконати цю операцію може лише процес із правами привілейованого користувача. При блокуванні забороняється підкачування сторінок з зовнішньої пам'яті.

▪ IPC_INFO (є тільки в Linux) – повертає параметри і інформацію про системні максимальні значення спільної пам'яті в структуру buf. Ця структура має тип shminfo (тобто, потрібне приведення типів) і визначена в <sys/shm.h>, якщо визначено макрос тестування властивостей _GNU_SOURCE:

```
struct shminfo {
  unsigned long shmmax; /* макс.розмір сегмента */
  unsigned long shmmin; /* мін.розмір сегмента =1 */
  unsigned long shmmni; /* макс.кількість сегментів */
  unsigned long shmseg; /* макс. кількість сегментів
                  до яких може підключитись;
                  процес не використовується в ядрі */
  unsigned long shmall; /* макс.кількість сторінок
                       спільної пам'яті в системі */
};
```

Значення shmmni, shmmax і shmall можна змінити за допомогою файлів у /proc з такими ж іменами;

▪ SHM_INFO (є тільки в Linux) – повертає структуру shm_info, де поля містять інформацію про системні ресурси, що використовуються спільною пам'яттю. Структура визначена в <sys/shm.h>, якщо визначено макрос тестування властивостей _GNU_SOURCE:

```
struct shm_info {
  int used_ids; /* кількість сегментів, що
                   використовуються в даний момент */
```

```
  unsigned long shm_tot;  /* загальна кількість
                          спільних сторінок пам'яті */
  unsigned long shm_rss;  /* кількість спільних
                сторінок, що знаходяться в пам'яті */
  unsigned long shm_swp;  /* кількість сторінок
                   пам'яті в просторі підкачування */
  unsigned long swap_attempts; /*не використовується*/
  unsigned long swap_successes;/*не використовується*/
};
```

Системні виклики **shmat** і **shmdt** виконують приєднання й від'єднання сегмента відповідно. Перш ніж використовувати спільний сегмент пам'яті, необхідно приєднатися до нього за допомогою **shmat**.

```
char *shmat(int shmid, char *shmaddr, int shmfl);
```

За допомогою цього виклику процес приєднує область пам'яті, що розділяється, дескриптор якої вказаний в `shmid`, до свого віртуального адресного простору. Після виконання цієї операції процес зможе читати і модифікувати дані, що знаходяться в області пам'яті, що розділяється, адресуючи її як будь-яку іншу область у своєму власному віртуальному адресному просторі.

Як другий аргумент `shmaddr` процес може вказати віртуальну адресу в своєму адресному просторі, починаючи з якого необхідно приєднати пам'ять, що розділяється. Найчастіше, однак, як значення цього аргументу передається 0, що означає, що система сама може вибрати адресу початку пам'яті, що розділяється. Передача конкретної адреси в цьому параметрі має сенс у тому випадку, якщо, наприклад, у пам'ять, що розділяється, записуються покажчики на неї ж (наприклад, у ній зберігається пов'язаний список) – у цій ситуації для того, щоб використання цих покажчиків мало сенс і було коректним для всіх процесів, підключених до пам'яті, важливо, щоб у всіх процесах адреса початку області пам'яті, що розділялася, збігалася. Третій аргумент `shmfl` є комбінацією прапорців. Як значення цього

аргументу може бути вказаний прапорець SHM_RDONLY, який вказує на те, що область, яка приєднується, буде використовуватися тільки для читання.

Функція **shmat** повертає адресу, починаючи з якої відображатиметься пам'ять, що приєднується. У разі невдачі виклик повертає -1.

Після закінчення роботи з сегментом спільної пам'яті, програма повинна від'єднатися від нього за допомогою **shmdt**:

```
int shmdt(char *shmaddr)
```

Параметр `shmaddr` – адреса прикріпленої до процесу пам'яті, отримана під час виклику **shmat**. У разі успішного виконання функція повертає 0, у разі невдачі -1.

Поділювані сегменти пам'яті в UNIX/Linux (як і семафори) не мають зовнішніх імен. При одержанні ідентифікатора сегмента процес користується числовим ключем. Розробники незв'язаних процесів можуть домовитися про загальне значення ключа, що вони будуть використати, але в них немає гарантії в тому, що це ж значення ключа не буде використане кимсь ще. Гарантовано унікальний сегмент можна створити з використанням ключа IPC_PRIVATE, але такий ключ не може бути зовнішнім. Тому сегменти використовуються, як правило, родинними процесами, які мають можливість передавати один одному їхні ідентифікатори, наприклад, через наслідувані ресурси або через параметри виклику дочірньої програми. Для генерації ключа можна скористатись викликом **ftok**, який дозволяє з певною ймовірністю згенерувати майже унікальний ключ. Опис наведено в темі «Синхронізація процесів за допомогою семафорів в ОС Linux».

Приклад 20.1 Наведені 2 програми (**shmexmpl1** та **shmexmpl2**), перша з яких створює спільний сегмент пам'яті, записує в нього дані – рядок, введений з клавіатури, та від'єднується від нього. Друга програма відкриває створений спільний сегмент пам'яті, зчитує з нього дані та виводить на екран, від'єднується від нього і наприкінці видаляє створений сегмент. Ключ для спільного сегменту в обох програмах однаковий, що дозволяє

спільно його використовувати другій програмі навіть після завершення першої.

Текст програми ***shmexmpl1***.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
// формуємо ключ для сегмента з рядка та символа
key_t shmkey=ftok("shmexmpl", 1);
// створюємо (або відкриваємо) спільний сегмент
int shmid = shmget(shmkey, 1024, 0666 | IPC_CREAT);
// приєднання до сегмента,
// змінна str має адресу сегмента
char* str = (char*)shmat(shmid, (void*)0, 0);
printf("\nWrite Data : ");
scanf("%s\n", str);
// від'єднання від сегмента
shmdt(str);
return 0;
}
```

Текст програми ***shmexmpl2***.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main()
{
// формуємо такий самий ключ для сегмента
// як у першій програмі
```

```
key_t shmkey=ftok("shmexmpl", 1);
// створюємо (або відкриваємо) спільний сегмент
int shmid = shmget(shmkey, 1024, 0666 | IPC_CREAT);
// приєднання до сегмента,
// змінна str має адресу сегмента
char* str = (char*)shmat(shmid, (void*)0, 0);
printf("\nData read from memory: %s\n",str);
// від'єднання від сегмента
shmdt(str);
// видалення сегмента
shmctl(shmid, IPC_RMID, NULL);
return 0;
}
```

У процесі налагодження програми можуть виникати ситуації, коли програма буде аварійно закінчуватися або перериватися перш, ніж вона знищить створені нею спільні сегменти пам'яті. Такі області не видаляються в системі автоматично й можуть накопичуватися протягом багатьох днів. Нагромадження таких «забутих» спільних областей може привести до того, що буде вичерпаний системний ліміт на кількість спільних сегментів, і черговий виклик **shmget** закінчиться з помилкою. Для того, щоб цього не відбувалося, необхідно регулярно виконувати процедуру очищення IPC за допомогою команди (видалити можна всі спільні сегменти, в яких `nattch` дорівнює 0)

```
ipcrm -m <shmid>
```

або для видалення всіх спільних сегментів

```
ipcrm -ashm
```

Список створених спільних сегментів можна побачити, виконавши команду:

```
ipcs -m
```

20.2. Контрольні питання

1. Чим відрізняється робота по обміну даними між процесами за допомогою файлів, що відображуються в пам'ять, в ОС Windows та спільними сегментами в ОС Linux?

2. Що відбудеться, якщо виконати наступну послідовність команд для приклада 20.1? Чому?

```
$ ./shmexmpl1
$ ipcrm -ashm
$ ./shmexmpl2
```

3. За рахунок чого є можливість прочитати дані зі спільного сегмента, коли процес, який його створив та записав у нього дані, вже завершився?

4. Що відбудеться, якщо видалити рядок `shmctl` з другої програми в прикладі 20.1?

5. Які є переваги використання спільних сегментів по відношенню до використання каналів для обміну даними між процесами?

20.3. Індивідуальні завдання

Розробити комплекс програм, за допомогою яких організовано обмін даними між батьківським процесом і дочірніми процесами відповідно до індивідуального завдання за допомогою загального сегменту пам'яті, що розділяється, з урахуванням наступних вимог:

- організувати синхронізовану роботу процесів за допомогою семафорів;
- хід роботи процесів відображати на екрані;
- батьківський процес та дочірні процеси повинні бути реалізовані в окремих виконуваних файлах.

У роботі необхідно завершити розробку з теми «Синхронізація процесів за допомогою семафорів в ОС Linux» з повноцінним обміном даними.

21. ОБМІН ДАНИМИ МІЖ ПРОЦЕСАМИ ЗА ДОПОМОГОЮ ЧЕРГ ПОВІДОМЛЕНЬ В ОС LINUX

Мета: освоєння використання механізму черг повідомлень як способу комунікацій між процесами в ОС Linux.

21.1. Теоретичні відомості

Механізм черг дає процесам можливість посилати іншим процесам потоки форматованих даних. Процес має можливість послати повідомлення в певну чергу й прийняти повідомлення з черги. Передача даних у черзі відбувається завжди повідомленнями, причому кожне повідомлення має заголовок і тіло. Заголовок завжди має фіксований для даної системи формат. У нього обов'язково входить довжина повідомлення, а інша інформація залежить від специфікацій конкретної системи: це може бути пріоритет повідомлення, тип повідомлення, ідентифікатор процесу, що послав повідомлення, та інше. Тіло повідомлення інтерпретується за правилами, встановленими самими процесами: відправником і одержувачем. Заголовок і тіло являють собою істотно різні структури даних і розташовуються в різних місцях у пам'яті. Властиво черга ОС становить із заголовків повідомлень. В елементи черги включаються покажчики на тіла повідомлень, що розташовуються в пам'яті системи.

Завдяки наявності типізації повідомлень, черга повідомлень надає можливість мультиплексувати повідомлення від різних процесів, при цьому кожна пара процесів, що взаємодіють через чергу, може використовувати свій тип повідомлень, і таким чином, їх дані не будуть змішуватися. Завдяки наявності типів повідомлень, чергу можна інтерпретувати подвійно – розглядати її або як наскрізну чергу невиразних за типом повідомлень, або як деяке поєднання підчерг, кожна з яких містить елементи певного типу.

Черги повідомлень дозволяють процесам обмінюватися структурованими даними, що мають такі атрибути:

- тип повідомлення (дозволяє мультиплексувати повідомлення в одній черзі);
- довжина даних повідомлення в байтах (може бути нульовою);

▪ власне дані (якщо довжина ненульова, можуть бути структурованими).

Черги повідомлень розташовуються в адресному просторі ядра операційної системи у вигляді односпрямованих списків і мають обмеження за обсягом інформації, що зберігається в кожній черзі. Для кожної черги ядро створює заголовок черги (`msgid_ds`), де міститься інформація про права доступу до черги (`msg_perm`), її поточний стан (`msg_cbytes` – число байтів та `msg_qnum` – число повідомлень у черзі), а також покажчики на перше (`msg_first`) та останнє (`msg_last`) повідомлення, що зберігаються у вигляді пов'язаного списку. Кожен елемент списку є окремим повідомленням.

Завдяки структурованості вибірка повідомлень із черги може здійснюватися трьома способами:

1. у порядку FIFO незалежно від типу повідомлення;

2. у порядку FIFO для повідомлень конкретного типу;

3. першим вибирається повідомлення з мінімальним типом, що не перевищує деякого заданого значення, яке прийшло раніше інших повідомлень із тим же типом.

Черги повідомлень, як і інші засоби IPC (семафори та спільні сегменти), дозволяють організувати взаємодію процесів, що не перебувають одночасно в обчислювальній системі.

Системні виклики UNIX/Linux. В ОС UNIX/Linux механізм черг повідомлень забезпечується чотирма системними викликами: **msgget**, **msgctl**, **msgsnd**, **msgrcv**.

Системний виклик **msgget** створює нову чергу або повертає ідентифікатор вже існуючої черги. Як і інші засоби взаємодії між процесами, черги в UNIX/Linux (як і семафори) не мають зовнішніх імен. При роботі з чергою один із процесів створює чергу й одержує її ідентифікатор. Цей ідентифікатор потім передається іншим процесам (можливо дочірнім) і використовується процесами для виконання операцій із чергою. Синтаксис виклику:

```
int msgget (key_t key, int msgflag);
```

При одержанні ідентифікатора черги процес користується числовим ключем `key`. Гарантовано унікальну чергу можна створити з використанням ключа IPC_PRIVATE. Як значення цього параметра може бути використане значення ключа, отримане за допомогою функції **ftok**. Використання значення IPC_PRIVATE завжди приводить до спроби створення нової черги повідомлень із ключем, що не збігається із значенням ключа ні однієї із вже існуючих черг і не може бути отриманий за допомогою функції **ftok** ні при одній комбінації її параметрів.

Параметр `msgflag` – прапорці – відіграє роль тільки при створенні нової черги повідомлень і визначає права різних користувачів при доступі до черги, а також необхідність створення нової черги і поведінки системного виклику при спробі створення. Він є деякою комбінацією (за допомогою операції «побітове або») визначених значень і вісімкових прав доступу аналогічно, як і для **semget** та **shmget**. Опис наведено в темі «Синхронізація процесів за допомогою семафорів в ОС Linux».

У випадку успішного завершення, повертається дескриптор IPC для цієї черги (ціле невід'ємне число, що однозначно характеризує чергу повідомлень всередині обчислювальної системи і використовується надалі для інших операцій з нею).

Черга повідомлень має обмеження на загальну кількість збереженої інформації, яка може бути змінена адміністратором системи. Поточне значення обмеження можна довідатися за допомогою команди

```
ipcs -l
```

Приклад результату виконання команди для ОС Ubuntu 22.04.4 LTS (виводиться також інформація про обмеження для семафорів та спільних сегментів):

```
------ Messages Limits --------
max queues system wide = 32000
```

```
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) =18446744073709551612
min seg size (bytes) = 1

------ Semaphore Limits --------
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

Системний виклик **msgctl** дозволяє виконувати керуючі операції над чергою: отримувати інформацію про статус черги повідомлень, змінювати деякі з пов'язаних з чергою обмежень або видаляти чергу з системи. Він виконує контрольну операцію, задану в `cmd`, над чергою повідомлень, заданою дескриптором `msgid`, отриманим у результаті виклику **msgget**:

```
int msgctl (int msgid, int cmd, struct msgid_ds *buf);
```

Структура даних `msqid_ds` визначена в `<sys/msg.h>` наступним чином:

```
struct msqid_ds {
  struct ipc_perm msg_perm;   /* володар і права */
  time_t msg_stime;   /* час останнього msgsnd */
  time_t msg_rtime;   /* час останнього msgrcv */
  time_t msg_ctime;   /* час останньої зміни */
  unsigned long __msg_cbytes; /* поточна кількість
```

```
                                  байтів у черзі */
  msgqnum_t msg_qnum; /* поточна кількість
                       повідомлень у черзі */
  msglen_t msg_qbytes; /* максимальна кількість байтів
                        для черги */
  pid_t msg_lspid;    /* PID останнього msgsnd */
  pid_t msg_lrpid;    /* PID останнього msgrcv */
};
```

Структура `ipc_perm` визначена наступним чином (значення полів встановлюються за допомогою IPC_SET):

```
struct ipc_perm {
  key_t __key; /* ключ, що передається в msgget */
  uid_t uid; /* діючий UID володаря */
  gid_t gid; /* діючий GID володаря */
  uid_t cuid; /* діючий UID творця */
  gid_t cgid; /* діючий GID творця */
  unsigned short mode; /* права */
  unsigned short __seq; /* порядковий номер */
};
```

Можливі значення `cmd`:

▪ IPC_STAT – копіює інформацію зі структури даних ядра, пов'язаної з `msqid`, у структуру `msqid_ds`, розташовану за адресою `buf`. Необхідно мати права на читання черги повідомлень;

▪ IPC_SET – записує значення деяких полів структури `msqid_ds`, адреса якої вказана в `buf`, у структуру даних ядра, пов'язану з цією чергою повідомлень, оновлюючи при цьому її поле `msg_ctime`. Оновлюються такі поля структури: `msg_qbytes`, `msg_perm.uid`, `msg_perm.gid` і `msg_perm.mode` (молодші 9 біт). Діючий UID викликаючого процесу повинен збігатися з ідентифікатором власника (`msg_perm.uid`) або творця

(msg_perm.cuid) черги повідомлень, або той хто її викликає, повинен мати привілеї. Для вказівки значення msg_qbytes, більшого ніж значення системного параметра MSGMNB, також потрібні відповідні привілеї;

▪ IPC_RMID – негайно видаляє чергу повідомлень, «пробуджуючи» всі процеси, що чекають запису або читання цієї черги (при цьому повертається помилка, а змінна errno набуває значення EIDRM). Процес повинен мати відповідні привілеї або його діючий ідентифікатор користувача повинен збігатися з ідентифікатором творця або власника черги повідомлень. Третій аргумент msgctl ігнорується;

▪ IPC_INFO (є тільки в Linux) – повертає параметри та інформацію про системні максимальні значення черги повідомлень у структурі, зазначеній у buf. Дана структура має тип msginfo (тобто потрібно приведення типів) і визначена в <sys/msg.h>, якщо визначено макрос тестування властивостей _GNU_SOURCE:

```
struct msginfo {
  int msgpool; /* розмір у кілобайтах буферного пула,
                  що використовується для зберігання
                  даних повідомлення */
  int msgmap;  /* максимальна кількість елементів у
                  карті повідомлень */
  int msgmax;  /* максимальна кількість байтів, які
          можуть бути записані в одне повідомлення */
  int msgmnb;  /* максимальна кількість байтів, які
          можна записати в чергу; використовується
          для ініціалізації msg_qbytes під час
          створення черги msgget */
  int msgmni;  /* макс. кількість черг повідомлень */
  int msgssz;  /* розмір сегмента повідомлення */
  int msgtql;  /* максимальна кількість повідомлень в
          усіх чергах у системі */
  unsigned short int msgseg;
                /* максимальна кількість сегментів */
```

```
};
```

Значення `msgmni`, `msgmax` і `msgmnb` можна змінити за допомогою файлів у /proc з такими ж іменами;

▪ MSG_INFO (є тільки в Linux) – повертає структуру `msginfo`, яка містить таку ж інформацію, що й IPC_INFO, за винятком того, що значення полів містять інформацію про системні ресурси, які споживають черги повідомлень: у полі `msgpool` – кількість черг повідомлень, які в поточний час є в системі; у полі `msgmap` – загальна кількість повідомлень в усіх чергах системи; у полі `msgtql` – загальна кількість байтів в усіх повідомленнях у всіх чергах системи;

▪ MSG_STAT (є тільки в Linux) – повертає структуру `msqid_ds` як в операції IPC_STAT, але аргумент `msqid` є не унікальним ідентифікатором, а індексом у внутрішньому масиві ядра, в якому міститься інформація про всі черги повідомлень у системі.

При успіху IPC_STAT, IPC_SET і IPC_RMID повертається 0. При успішному виконанні операції IPC_INFO або MSG_INFO повертається індекс найостаннішого використаного елемента у внутрішньому масиві ядра, в якому записана інформація про всі черги повідомлень. При успішному виконанні операції MSG_STAT повертається ідентифікатор черги, чий індекс був вказаний в `msqid`. У разі помилки повертається -1, а в `errno` записується значення помилки:

▪ EACCES – значення аргументу `cmd` дорівнює IPC_STAT або MSG_STAT, але процес, що викликає, не має прав на читання черги повідомлень `msqid`;

▪ EFAULT – значення аргументу `cmd` дорівнює IPC_SET або IPC_STAT, але адреса, вказана в `buf`, недоступна;

▪ EIDRM – чергу повідомлень було видалено;

▪ EINVAL – невірне значення `cmd` або `msqid` або для операції MSG_STAT значення індексу, вказаного в `msqid`, посилається на елемент масиву, який на даний момент не використовується;

▪ EPERM – значення аргументу `cmd` дорівнює IPC_SET або IPC_RMID, але діючий ідентифікатор користувача процесу не дорівнює ідентифікатору творця (`msg_perm.cuid`) або власника (`msg_perm.uid`) черги повідомлень, і той, хто викликає, не має прав;

▪ EPERM – була зроблена спроба (IPC_SET) збільшити `msg_qbytes` і порушити межі системного параметра MSGMNB, але той, хто викликає, не має прав.

Системні виклики **msgsnd** і **msgrcv** виконують посилку й прийом повідомлення відповідно. При посилці й прийомі повідомлень процеси оперують числовим типом повідомлення й текстом повідомлення. Тип повідомлення може виконувати роль пріоритету: процес може вибирати із черги повідомлення заданого типу або повідомлення, що мають найменше число типу. Якщо вибірка по типу не застосовується, повідомлення вибираються за принципом FIFO.

Системний виклик **msgsnd** копіює користувальницьке повідомлення в чергу повідомлень, задану IPC-дескриптором:

```
int msgsnd(int msqid, struct msgbuf *ptr,
           int length, int flag);
```

Повідомлення міститься в структурі `struct msgbuf`, описаній у файлі `<sys/msg.h>` – шаблоні, визначеному користувачем наступної форми:

```
struct msgbuf {
long mtype;
char mtext[1];
};
```

Перший елемент `mtype` обов'язково має тип `long` і містить тип повідомлення, а далі розміщується інформативна частина теоретично довільної довжини (практично в Linux вона має обмеження, як було вказано вище),

яка містить саме повідомлення. При цьому інформація зовсім не обов'язково є текстовою (`char mtext`) – це можуть бути будь-які типи даних.

Значення поля `mtype` можна використовувати для розбиття повідомлень на категорії. При цьому значущими є лише позитивні значення; негативні або нульові не можуть використовуватись. Довжина повідомлення, що надсилається, задається параметром `length` виклику **msgsnd** і може бути в діапазоні від нуля до меншого з двох значень: розміру `mtext` і максимального розміру повідомлення, визначеного в системі. Параметр `flag` виклику **msgsnd** може нести лише один прапорець: IPC_NOWAIT. При невстановленому параметрі IPC_NOWAIT процес призупинить роботу, якщо для надсилання повідомлення недостатньо системних ресурсів: якщо повна довжина повідомлень у черзі перевищить максимум, заданий для черги чи всієї системи. Якщо прапорець IPC_NOWAIT встановлено, тоді при неможливості надіслати повідомлення повернення з виклику відбудеться негайно. Значення, що повертається буде -1, і змінна `errno` матиме значення EAGAIN, що означає необхідність повторення спроби. Виклик **msgsnd** також може завершитись помилкою через встановлені права доступу. Наприклад, якщо ні діючий ідентифікатор користувача, ні діючий ідентифікатор групи процесу не пов'язані з чергою, і встановлено код доступу до черги 0660, виклик **msgsnd** для цієї черги завершиться невдачею.

Слід ще раз звернути увагу на кілька особливостей **msgsnd**:

▪ тип даних `struct msgbuf` не є типом даних для користувальницьких повідомлень, а є лише шаблоном для створення таких типів. Користувач сам повинен створити структуру для своїх повідомлень, у якій першим полем повинна бути змінна типу `long` , яка містить позитивне значення типу повідомлення;

▪ довжина повідомлення `length` – вказується не вся довжина структури даних, що відповідає повідомленню, а тільки довжина корисної інформації, тобто інформації, яка розташовується в структурі даних після типу повідомлення. Це значення може бути рівним 0 у випадку, коли вся корис-

на інформація полягає в самому факті приходу повідомлення (повідомлення використовується як сигнальний засіб зв'язку);

▪ як правило, використовується нульове значення прапорця системного виклику `flag`, яке приводить до блокування процесу при відсутності вільного місця в черзі повідомлень.

Для читання із черги, заданої ідентифікатором `msqid`, використовується виклик **msgrcv**. Читання дозволено, якщо процес має право на доступ до черги на читання. Успішне читання повідомлення призводить до видалення його з черги.

```
int msgrcv(int msqid, struct msgbuf *ptr,
           int length, long type, int flag);
```

Структура `struct msgbuf` використовується для зберігання отриманого повідомлення, а параметр `length` визначає максимальну довжину повідомлень, які можуть знаходитися в цій структурі. Успішний виклик повертає довжину отриманого повідомлення.

Параметр `type` визначає тип повідомлення, що приймається, він допомагає вибрати потрібне з повідомлень, що знаходяться в черзі. Якщо параметр `type` дорівнює нулю, з черги зчитується перше повідомлення, тобто те, що було надіслано першим. При позитивному ненульовому значенні параметра `type` зчитується перше повідомлення з черги із заданим типом повідомлення. Наприклад, якщо черга містить повідомлення зі значеннями `mtype` 999, 5 та 1 поля структури `struct msgbuf`, а параметр `type` у виклику **msgrcv** має значення 5, то зчитується повідомлення типу 5. І, нарешті, якщо параметр `type` має ненульове негативне значення, то зчитується перше повідомлення з найменшим значенням `mtype`, яке менше або дорівнює модулю параметра `type`. Цей алгоритм здається складним, але демонструє просте правило: якщо повернутися до попереднього прикладу з трьома повідомленнями зі значеннями `mtype` 999, 5 і 1, то при значенні параметра `type` у виклику **msgrcv** рівному -999 і триразовому виклику, повідомлення будуть отримані в порядку 1, 5, 999.

Останній параметр `flag` містить керуючу інформацію. У цьому параметрі можуть бути незалежно встановлені два прапорці – IPC_NOWAIT та MSG_NOERROR. Прапорець IPC_NOWAIT має звичайний зміст – якщо він не заданий, процес буде призупинено за відсутності в черзі відповідних повідомлень, і повернення з виклику відбудеться після надходження повідомлення відповідного типу. При встановленні прапорця IPC_NOWAIT системний виклик у цій ситуації не блокується, повернення з виклику за будь-яких обставин відбудеться негайно, та констатується виникнення помилки із встановленням значення змінної `errno`, описаної у файлі `<errno.h>`, рівним EAGAIN.

При встановленому прапорці MSG_NOERROR повідомлення буде усічено, якщо його довжина більша, ніж `length` байтів. Дізнатися про те, що усічення мало місце, неможливо. Якщо реальна довжина корисної частини інформації в обраному повідомленні перевищує значення, зазначене в параметрі `length` і прапорець MSG_NOERROR не встановлений, то вибірка повідомлення не здійснюється, і фіксується наявність помилкової ситуації.

При використанні виклику **msgrcv** потрібно звернути увагу на наступні моменти:

- тип даних `struct msgbuf`, як і для виклику **msgsnd**, є лише шаблоном для користувацького типу даних;
- спосіб вибору повідомлення задається нульовим, додатним або від'ємним значенням параметра `type`. Точне значення типу обраного повідомлення можна визначити з відповідного поля структури, в яку системний виклик скопіює повідомлення;
- системний виклик повертає довжину тільки корисної частини скопійованої інформації, тобто інформації, розташованої в структурі після поля типу повідомлення `type`;
- обране повідомлення вилучається із черги повідомлень;
- як параметр `length` вказується максимальна довжина корисної частини інформації, що може бути розміщена в структурі, адресованій параметром `ptr`;

▪ як правило, використовується нульове значення прапорців для системного виклику, що приводить до блокування процесу у випадку відсутності в черзі повідомлень із потрібним типом і до помилкової ситуації у випадку, коли довжина інформативної частини обраного повідомлення перевищує довжину, вказану в параметрі `length`.

Приклад 21.1. Наводяться дві програми для одноразової передачі інформації через чергу повідомлень. Перша програма (**msgexmpl1**) створює чергу повідомлень та надсилає повідомлення – рядок, введений з клавіатури. Друга програма (**msgexmpl2**) відкриває чергу повідомлень, приймає з неї одне повідомлення та виводить його на екран, після чого видаляє створену чергу повідомлень. Ключ для черги повідомлень в обох програмах однаковий, що дозволяє спільно її використовувати другій програмі навіть після завершення першої.

Текст програми ***msgexmpl1***.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX 10

struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
key_t key;
int msgid;
key = ftok("msgexmpl", 1);
msgid = msgget(key, 0666 | IPC_CREAT);
message.mesg_type = 1;
```

```
printf("Write Data : ");
fgets(message.mesg_text,MAX,stdin);
msgsnd(msgid, &message, sizeof(message), 0);
printf("Data send is : %s \n", message.mesg_text);
return 0;
}
```

Текст програми ***msgexmpl2***.

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct mesg_buffer {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
key_t key;
int msgid;
key = ftok("msgexmpl", 1);
msgid = msgget(key, 0666 | IPC_CREAT);
msgrcv(msgid, &message, sizeof(message), 1, 0);
printf("Data Received is : %s \n", message.mesg_text);
msgctl(msgid, IPC_RMID, NULL);
return 0;
}
```

У процесі налагодження програми можуть виникати ситуації, коли програма буде аварійно закінчуватися або перериватися перш, ніж вона знищить створені нею черги. Такі черги не видаляються в системі автоматично й можуть накопичуватися протягом багатьох днів. Нагромадження

таких «забутих» черг може привести до того, що буде вичерпаний системний ліміт на кількість черг, і черговий виклик **msgget** закінчиться з помилкою. Для того, щоб цього не відбувалося, необхідно регулярно виконувати процедуру очищення IPC за допомогою команди

```
ipcrm msg <msqid>
```

або для видалення всіх створених черг

```
ipcrm -amsg
```

Список створених черг можна побачити, виконавши команду:

```
ipcs -q
```

21.2. Контрольні питання

1. Як зміниться поведінка роботи програм у прикладі 21.1, якщо змінити порядок їх запуску на зворотній? Відповідь поясніть.

2. Як зміниться поведінка роботи програм у прикладі 21.1, якщо спочатку першу програму запустити 3 рази, а потім другу програму запустити 3 рази? Відповідь поясніть.

3. Чим відрізняється робота зі спільними сегментами пам'яті та чергами повідомлень?

4. Чи можна використовувати одну чергу повідомлень для обміну даними між кількома групами процесів (наприклад, серед 6 процесів двоспрямований обмін повідомленнями ведеться тільки в групах між 1 та 2 процесом, між 3 та 4 процесом та між 5 та 6 процесом)? Якщо так, то яким чином організувати обмін?

5. Чи є схожі риси при роботі з повідомленнями в ОС Windows та з чергами повідомлень в ОС Linux? Що відрізняється?

6. Що відбудеться при роботі програми в прикладі 21.1, якщо спробувати передати повідомлення розміром `mesg_text[100]`, як описано в структурі `mesg_buffer`? Чому?

21.3. Індивідуальні завдання

Розробити комплекс програм, за допомогою яких організовано протокол обміну повідомленнями між процесом-«батьком» та процесами-«нащадками» відповідно до індивідуального завдання за допомогою черги повідомлень з урахуванням наступних вимог: вихідні дані вводити з клавіатур; хід роботи процесів відобразити на екрані; всі процеси повинні створюватись з окремих виконуваних файлів.

1) Обчислити для n (n>5) клієнтів 12 відсотків від суми кредиту. У «нащадку» 1 обчислити 12 відсоткову виплату частини клієнтів [1 – k] (n>k), а «нащадку» 2 – для клієнтів [(k+1) – n]. Задати k і n у батьківському процесі та передати повідомленням конкретному «нащадку». Результати розрахунку «нащадки» передають повідомленням батькові.

2) Організувати обмін текстовими повідомленнями між процесом-«батьком» та двома процесами-«нащадками». Процеси-«нащадки» виступають у ролі клієнтів, процес-«батько» – сервер, який може визначити, від якого клієнта отримано повідомлення.

3) Виконати породження «Сина», «Онука» та «Правнука». У тілі кожного процесу визначити інформацію про процес: ідентифікатор та покоління («батько» – 1 покоління, «син» – 2 покоління тощо). Кожен «нащадок» надсилає повідомлення з інформацією про себе. Кожен «батько» отримує повідомлення з черги тільки від свого «сина».

4) Обчислити число поєднань C(k,n)=n!/(k!·|n-k|)!), де n>0, k>0, n≠k. Обчислення n!, k!, | n-k |! виконати в трьох процесах-«нащадках» відповідно. Задати k і n у батьківському процесі і передати повідомленням конкретному «нащадку». Результати розрахунку «нащадки» передають повідомленнями «батькові».

5) Виконати породження 3 процесів-«нащадків». У кожному з них визначити значення свого і батьківського ідентифікатора. Від кожного «батька» виконати передачу повідомлення «Hello, Son! My Father ID =, My ID =, Your ID = !» своєму «нащадку».

6) Обчислити щільність нормального розподілу в точці x за формулою $f(x)=Exp(-x^2/2)/Sqrt(2\cdot x)$, де x>0. Обчислення експоненти та квадратного кореня виконати в двох процесах-«нащадках» відповідно. Задати x у бать-

ківському процесі та передати повідомленням процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

7) Обчислити щільність опуклого розподілу в точці x за формулою $f(x)=(1-Cos(x))/(x^2)$, де $x>0$. Обчислення $Cos(x)$ і x^2 виконати в двох процесах-«нащадках» відповідно. Задати x у батьківському процесі та передати повідомленням процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

8) Обчислити значення функції $f(x)=((-1)^x+x^{(2x-1)})/(2x-1)!$, де $x>0$. Обчислення $(-1)^x$, $x^{(2x-1)}$, $(2x-1)!$ виконати в трьох процесах-«нащадках» відповідно. Задати x і k у батьківському процесі та передати повідомленням процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

9) Обчислити значення функції $f(x,k)=(x^{(2k+1)}+x^{(5k-1)})/(2k+1)$. Обчислення $(2k+1)$ і $x^{(5k-1)}$ виконати в двох процесах-«нащадках» відповідно. Задати x і k у батьківському процесі та передати повідомленням процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

10) Обчислити значення функції $f(x,k)=(1!\cdot 2!\cdot 3!\cdot \ldots \cdot k!)/x!$, де $0>x<30$, $1>k<10$. Для обчислення кожного факторіалу виконати породження процеса-«нащадка». Задати x і k у батьківському процесі та передати повідомленнями процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

11) Визначити, сума цифр якого цілого числа більша від заданих трьох. Підрахунок суми цифр кожного числа виконати в окремому процесі-«нащадку». Задати x і k у батьківському процесі та передати повідомленнями процесам-«нащадкам». Результати розрахунку «нащадки» передають повідомленням «батькові».

12) Організувати обмін текстовими повідомленнями між «батьком» (1) та «нащадками» (2-3) за такою схемою: 1→2,3; 2→3,1 (символ → означає напрям передачі).

13) Організувати обмін текстовими повідомленнями між процесом-«батьком» та процесом-«нащадком». При цьому кожен одержувач після

отримання повідомлення виводить на екран інформацію про стан черги повідомлень.

14) Організувати обмін текстовими повідомленнями між «батьком» (1) та «нащадками» (2-3) за такою схемою: 2→1; 1→3; 3→1 (символ → означає напрям передачі).

15) Організувати двосторонній обмін текстовими повідомленнями між двома процесами, які не пов’язані прямими спорідненими відносинами. При цьому значення ключа, з яким пов’язана черга повідомлення, отримати за ім’ям однієї з програм.

СПИСОК ЛІТЕРАТУРИ

1. Billimoria K. Linux Kernel Programming. Part 2 – Char Device Drivers and Kernel Synchronization: Create user-kernel interfaces, work with peripheral I/O, and handle hardware interrupts / K. Billimoria. – Packt Publishing, 2021. – 452 p.

2. Billimoria K. Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization / K. Billimoria. – Packt Publishing, 2021. – 754 p.

3. Carver R.H. Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs 1st Edition / R.H. Carver, K.-C. Tai. – Wiley-Interscience, 2005. – 480 p.

4. Cody I.D. C++ and Linux Operating System 2 Bundle Manuscript Essential Beginners Guide on Enriching Your C++ Programming Skills and Learn the Linux Operating System / I.D. Cody. – CreateSpace Independent Publishing Platform, 2017. – 198 p.

5. Hart J.M. Windows System Programming. 4th edition / J.M. Hart. – Addison-Wesley Professional, 2015. – 609 p.

6. Kerrisk M. The Linux Programming Interface: A Linux and UNIX System Programming Handbook / M Kerrisk. – No Starch Press, 2010. – 1580 p.

7. Love R. Linux Kernel Development / R. Love. – Addison-Wesley Professional, 2010. – 468 p.

8. Love R. Linux System Programming: Talking Directly to the Kernel and C Library. 2nd Edition / R. Love. – O'Reilly Media, 2013. – 454 p.

9. Petzold C. Programming Windows: The Definitive Guide To The Win32 API. 5th edition / C. Petzold. – Wiley, 2011. – 1512 p.

10. Windows Internals, 7th Edition. Part 1. System architecture, processes, threads, memory management, and more / Yosifovich P., Ionescu A., Russinovich M.E., Solomon D.A. – Microsoft Press, 2017. – 800 p.

11. Windows Internals, 7th Edition. Part 2 (Developer Reference) / Allievi A., Russinovich M.E., Ionescu A., Solomon D.A. – Microsoft Press, 2021. – 912 p.

12. Yosifovich P. Windows 10 System Programming, Part 1 / P. Yosifovich. – Independently published, 2021. – 528 p.

13. Yosifovich P. Windows 10 System Programming, Part 2 / P. Yosifovich. – Independently published, 2021. – 500 p.

14. Yosifovich P. Windows Kernel Programming / P. Yosifovich. – Independently published, 2023. – 625 p.

15. Авраменко В.С., Авраменко А.С. Основи операційних систем. Навчальний посібник / В.С. Авраменко, А.С. Авраменко. – Черкаси: ЧНУ імені Богдана Хмельницького, 2018. – 524 с.

16. Архітектура системного програмного забезпечення: підручн. для студ. спец. 121 «Інженерія програмного забезпечення» / Л.О. Левченко, Н.Г. Кучук, Ю.А. Тарнавський, В.П. Колумбет; КПІ ім. Ігоря Сікорського. – Київ: КПІ ім. Ігоря Сікорського, 2022. – 497 с.

17. Зайцев В.Г. Операційні системи: навчальний посібник для студентів спеціальності 123 «Комп'ютерна інженерія» / В.Г. Зайцев, І.П. Дробязко. – Київ: КПІ ім. Ігоря Сікорського, 2019. – 240 с.

18. Операційні системи: навчальний посібник. / Федотова-Півень І.М., Миронець І.В., Півень О.Б., Сисоєнко С.В., Миронюк Т.В.; Черкаський державний технологічний університет. – Харків : ТОВ «ДІСА ПЛЮС», 2019. – 216 с.

19. Системне програмування: розробка багатопотокових програм в операційній системі Linux: навчальний посібник / Гоменюк С.І., Чопоров С.В., Лісняк А.О., Кудін О.В., Гребенюк С.М. – Запоріжжя: Запорізький національний університет, 2021. – 120 с.

Навчальне видання

ПАНЧЕНКО Володимир Іванович
КОЛОМІЙЦЕВ Олексій Володимирович
МЕЖЕРИЦЬКИЙ Сергій Геннадійович

СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ. ПРОГРАМУВАННЯ СИСТЕМНИХ МЕХАНІЗМІВ ОС

Навчальний посібник
для студентів спеціальності
123 «Комп'ютерна інженерія»
денної та заочної форм навчання

Відповідальний за випуск проф. Олександр ЗАКОВОРОТНИЙ

Роботу до видання рекомендував проф. Микола ЗАПОЛОВСЬКИЙ

В авторській редакції

План 2024 р., поз. 91

Підписано до друку р.. Формат 60×84 1/16. Папір офсетний.
Різо-друк. Гарнітура Таймс. Ум. друк. арк. .

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
м. Харків, вул. Кирпичова, 2.

Електронна версія