Лабораторна робота №5

Тема: Керування пристроями вводу/виводу.

Мета роботи: навчитися розробляти програми керування пристроями вводу/виводу.

Обладнання та програмне забезпечення: ПК, операційна система Windows, середовище розробки ПЗ Visual Studio.

Вказівки для самостійної підготовки

Під час підготовки необхідно повторити теоретичний матеріал:

1. Способи виконання операцій введення-виведення.
2. Підсистема введення-виведення ядра.
3. Введення-виведення у режимі користувача.
4. Таймери і системний час.
5. Керування введенням-виведенням: Windows.

Теоретичні відомості

1 Способи виконання операцій введення-виведення

1. Зовнішній пристрій взаємодіє із комп'ютерною системою через точку зв'язку, яку називають ***портом***. Якщо кілька пристроїв з'єднані між собою і обмінюються повідомленнями відповідно до визначеного протоколу, то вони використовують ***шину***.
2. Пристрої зв'язуються із комп'ютером через контролери. Є два базові способи зв'язку із контролером: через ***порт введення-виведення*** і ***відображувану пам'ять***. У першому випадку дані пересилають за допомогою спеціальних інструкцій, у другому – взаємодіють з контролером. Деякі пристрої застосовують обидві технології відразу.
3. Спілкування із контролером через порт використовує чотири регістри. Команди записують у *керуючий регістр*, дані - у *регістр виведення*, інформація про стан контролера зчитується із *регістра статусу*, дані від контролера - із *регістра введення*.
4. Ядро ОС реєструє всі порти введення-виведення і діапазони відображуваної пам'яті, а також інформацію про використання пристроєм порту і діапазону.

1.1 Опитування пристроїв

Контролер повідомляє, що він зайнятий, увімкнувши біт ***busy*** регістра статусу. Програма повідомляє про команду запису вмиканням біта ***write*** командного регістра, а про те, що є команда — за допомогою біта ***cready*** того самого регістра. Послідовність кроків протоколу взаємодії з контролером:

1. Застосування у циклі зчитує біт ***busy***, поки він не буде вимкнутий.
2. Застосування вмикає біт ***write*** керуючого регістра і відсилає байту регістр виведення.
3. Застосування вмикає біт ***cready***.
4. Коли контролер зауважує, що біт ***cready*** увімкнутий, то вмикає біт ***busy***.
5. Контролер зчитує значення керуючого регістра і бачить команду ***write***. Він зчитує регістр виведення, отримує із нього байт і передає пристрою.
6. Контролер очищує біти ***cready*** і ***busy***, показуючи, що операція завершена.

На першому етапі програма займається опитуванням пристрою, вона перебуває в циклі активного очікування. Одноразове опитування здійснюється швидко, для нього досить трьох інструкцій процесора — читання регістра, виділення біта статусу і переходу за умовою, пов'язаною з цим бітом. Проблеми з'являються, коли опитування потрібно повторювати багаторазово.

У більшості випадків реалізується введення-виведення, кероване перериваннями.

1.2 Введення-виведення, кероване перериваннями

Механізм переривань дозволяє процесору відповідати на асинхронні події. Подія може бути згенерована контролером після закінчення введення-виведення або у разі помилки, після чого процесор зберігає стан і переходить до виконання оброблювача переривання, встановленого ОС. Цим знімають необхідність опитування пристрою — система може продовжувати виконання після початку операції введення-виведення.

Рівні переривань

Переривання поділяють на рівні відповідно до їхнього пріоритету (***Interrupt Request Level, IRQL*** - рівень запиту переривання). Окремі фрагменти коду ОС можуть маскувати переривання, нижчі від певного рівня, скасовуючи їхнє отримання. Виділяють немасковані переривання, отримання яких не можна скасувати (апаратний збій пам'яті тощо).

Встановлення оброблювачів переривань

Контролер переривань здійснює роботу з перериваннями на апаратному рівні. Він є мікросхемою, що відсилає сигнал процесору різними лініями. Процесор вибирає оброблювач переривання, на який потрібно перейти, на підставі номера лінії, що нею прийшов сигнал. Її називають лінією запиту переривання (IRQ line). Сучасні системи мають розширені програмовані контролери переривань (Advanced Programmable Interrupt Controllers, APIC), які реалізують багато ліній переривання (для APIC фірми Intel ліній 255) і коректно розподіляють переривання між процесорами за умов багатопроцесорних систем.

Ядро ОС зберігає інформацію про всі доступні лінії переривань. Драйвер пристрою дає запит каналу переривання (IRQ) перед використанням і вивільняє після використання. Різні драйвери можуть спільно використовувати лінії переривань. Оброблювач переривання встановлюється під час ініціалізації драйвера або першого доступу до пристрою. Драйвер визначає, яку лінію переривання буде використовувати пристрій (чому дорівнює *номер переривання* для цього пристрою).

Відкладена обробка переривань

Основною вимогою до оброблювачів є ефективність реалізації. Два критерії (швидкість і обсяг роботи) конфліктують один із одним. Сучасні ОС вирішують проблему поділом коду оброблювача переривання навпіл.

Верхня половина - це безпосередньо оброблювач переривання, що виконується у відповідь на прихід сигналу відповідною лінією. У верхній половині здійснюють мінімально необхідну обробку, після чого вона планує до виконання другу частину.

Нижня половина не виконується негайно у відповідь на переривання, ядро планує її до виконання пізніше, у безпечніший час.

Таку технологію називають *відкладеною обробкою переривань,* вона реалізована в усіх сучасних ОС, у Windows XP - *відкладеними викликами процедур*.

1.3 Прямий доступ до пам'яті

Процесор бере участь у кожній операції читання і запису, пересилаючи дані від пристрою у пам'ять і назад. Якщо переривання йдуть часто, то час їхньої обробки може бути порівняний із часом, відпущеним для решти робіт. Бажано пересилати дані між пристроєм і пам'яттю більшими блоками і без участі процесора, а його у цей час зайняти більш продуктивними операціями.

Це призвело до розробки *контролерів прямого доступу до пам'яті* (direct memory access, **DMA**). Такий контролер сам керує пересиланням блоків даних від пристрою безпосередньо у пам'ять, не залучаючи до цього процесора. Блоки даних, які пересилають, набагато більші, ніж розрядність процесора, можуть бути завдовжки 4 Кбайт.

Схема введення-виведення:

1. процесор дає команду DMA - контролеру виконати читання блоку від пристрою, разом із командою він відсилає контролеру адресу буфера для введення-виведення;
2. DMA - контролер починає пересилання, процесор у цей час може виконувати інші інструкції;
3. після завершення пересилання всього блоку DMA - контролер генерує переривання;
4. оброблювач переривання (нижня половина) завершує обробку операції читання, переміщуючи дані із фізичного буфера у сторінкову пам'ять. Процесор тут бере участь тільки на початку операції та в кінці - за все інше відповідає контролер прямого доступу до пам'яті.

2 Підсистема введення-виведення ядра

2.1 Планування операцій введення-виведення

Планування введення-виведення реалізоване як середньо-термінове планування. З кожним пристроєм пов'язують чергу очікування, під час виконання блокувального виклику потік поміщають у чергу для відповідного пристрою, з якої його вивільняє оброблювач переривання. Різним пристроям можуть присвоювати різні пріоритети.

2.2 Буферизація

Технологією підвищення ефективності обміну даними між пристроєм і застосуванням або між двома пристроями є буферизація. Виділяють спеціальну ділянку пам'яті, яка зберігає дані під час цього обміну і є буфером. Залежно від того, скільки буферів використовують і де вони перебувають, розрізняють кілька підходів до організації буферизації.

Способи реалізації буферизації

Першим способом є ***одинарна буферизація в ядрі***. У ядрі створюють буфер, куди копіюють дані в міру їхнього надходження від пристрою. Коли буфер заповнюється, весь його вміст за одну операцію копіюють у буфер, що перебуває у просторі користувача. Аналогічно під час виведення даних їх спочатку копіюють у буфер ядра, після чого вже ядро відповідатиме за їхнє виведення на пристрій.

Коли буфер переповнений, нові дані нікуди помістити. Технологія ***подвійної буферизації*** у пам'яті ядра створює два буфери. Коли перший з них заповнений, дані починають надходити в другий, і до моменту, коли заповниться другий, перший уже буде готовий прийняти нові дані і т. д.

Буферизація і кешування

Основна відмінність між ними полягає в тому, що буфер може містити єдину наявну копію даних, тоді як кеш зберігає у більш швидкій пам'яті копію даних з іншого місця.

Ділянку пам'яті в деяких випадках можна використати і як буфер, і як кеш. Якщо після виконання операції введення із використанням буфера надійде запит на таку саму операцію, дані можуть бути отримані із буфера, який при цьому буде частиною кеша. Сукупність таких буферів називають буферним кешем.

Використання буферного кеша дає можливість накопичувати дані для збереження їх на диску великими обсягами за одну операцію, що сприяє підвищенню ефективності роботи підсистеми введення-виведення.

2.3 Введення-виведення із розподілом та об'єднанням

Способом оптимізації операцій введення-виведення із записом великих обсягів даних протягом однієї операції, є введення-виведення з розподілом та

об'єднанням. В системі дозволяется використовувати під час введення-виведення набір непов'язаних ділянок пам'яті. При цьому можливі дві дії:

1. дані із пристрою відсилають у набір ділянок пам'яті за одну операцію введення (операція розподілу під час введення);
2. усі дані набору ділянок пам'яті відсилають пристрою для виведення за одну операцію (операція об'єднання під час виведення).

Виконання цих дій дає змогу обійтися без додаткових операцій доступу до пристрою, які потрібно було б виконувати, коли всі ділянки пам'яті були використані для введення-виведення по одній.

Win32 API надає для введення із розподілом функцію ***ReadFileScatter(),*** а для виведення з об'єднанням - ***WriteFileGather().***

2.4 Спулінг

Спулінг - технологія виведення даних із використанням буфера, що працює за принципом FIFO. Буфер називають ***спулом*** або ***ділянкою спула***.

Спулінг використовують коли виведення даних має виконуватися неподільними порціями (роботами). Неподільність робіт полягає в тому, що їхній вміст під час виведення не перемішується (тільки після виведення всіх даних однієї роботи має починатися виведення наступної). Прикладом виведення є робота із розподілюваним принтером, запити на друкування документів приходять від багатьох процесів у довільному порядку, але друкуватися документи можуть тільки по одному (тут роботою є документ).

Роботи надходять у спул і вишиковуються у FIFO - чергу (нові роботи додаються у її хвіст). Як тільки пристрій вивільняється, роботу із голови черги передають пристрою для виведення.

Спулінг пов'язаний із повільними пристроями, найчастіше ділянку спула організовують на жорсткому диску, а роботи відображають файлами.

3 Введення-виведення у режимі користувача

3.1 Синхронне введення-виведення

У більшості випадків введення-виведення на рівні апаратного забезпечення кероване перериваннями, а отже є асинхронним. Однак використати

асинхронну обробку даних завжди складніше, ніж синхронну, тому найчастіше введення-виведення в ОС реалізоване у вигляді набору блокувальних або синхронних системних викликів. Під час виконання такого виклику поточний потік призупиняють, переміщуючи в чергу очікування для цього пристрою. Після завершення операції введення-виведення і отримання всіх даних від пристрою потік переходить у стан готовності та може продовжити своє виконання.

3.2 Багатопотокова організація введення-виведення

Цей підхід полягає в тому, що за необхідності виконання асинхронного введення-виведення у застосуванні створюють новий потік, у якому виконуватиметься звичайне, синхронне введення-виведення. При блокуванні цього потоку вихідний потік продовжуватиме своє виконання.

Такий підхід має багато переваг і може бути рекомендований для використання у багатьох видах застосувань. Приклад розробки багатопотокового сервера за принципом «потік для запиту».

У таких серверах є головний потік, який очікує безпосередніх запитів клієнта на отримання даних. Після отримання кожного запиту головний потік створює новий робочий потік для обробки його запиту, після чого продовжує очікувати подальших запитів. Робочий потік обробляє запит і завершується. Такий підхід застосовують для зв'язку без збереження стану, коли обробка одного запиту не залежить від обробки іншого.

3.3 Введення-виведення у режимі користувача

Доцільно організувати одночасне очікування отримання даних із кількох дескрипторів.

Виконання введення-виведення поділяють на кілька етапів:

1. Спеціальний системний виклик (виклик повідомлення) визначає, чи можна виконати синхронне введення-виведення хоча б для одного дескриптора із заданого набору без блокування потоку.
2. Як тільки хоча б один дескриптор із набору стає готовий до введення-виведення без блокування, виклик повідомлення повертає керування; при цьому поточний потік може визначити, для яких саме дескрипторів може бути виконане введення-виведення або які з них змінили свій стан (тобто отримати повідомлення про стан дескрипторів).

3. Потік, що викликає, може у циклі обійти всі дескриптори, визначені внаслідок повідомлення на етапі 2, і виконати введення-виведення для кожного з них, блокування поточного потоку ця операція в загальному випадку не спричинить.

3.4 Порти завершення введення-виведення

Технологія ***порт завершення введення-виведення*** поєднує багатопотоковість із асинхронним введенням-виведенням для вирішення проблем розробки серверів, що обслуговують велику кількість одночасних запитів.

Одним із варіантів є організація ***пула потоків***. При цьому в момент запуску серверного застосування заздалегідь створюють набір потоків (пул), кожен із яких готовий обслуговувати запити. Коли приходить запит, перевіряють, чи є в пулі вільні потоки, якщо є, з нього вибирають потік, який починає обслуговувати запит. Після виконання запиту потік повертають у пул. Коли з появою нового запиту вільних потоків у пулі немає, запит поміщають у чергу, і він там очікує, поки не вивільниться потік, що може його обслужити. При цьому можна керувати розміром пула, досягаючи того, щоб кількість активних потоків збігалася із кількістю процесорів у системі.

Для підтримки організації такого пула потоків і було розроблено технологію портів завершення введення-виведення. Такі порти дотепер реалізовані лише у системах лінії Windows XP.

4 Таймери і системний час

4.1 Керування системним часом

Апаратний таймер - пристрій, що генерує переривання таймера через певний проміжок часу. Як пристрій використовується для відстеження поточного системного часу.

Завдання розв'язують просто: створюють лічильник, який збільшують для кожного переривання таймера. Основною проблемою є розмір цього лічильника, а саме:

1. 32-бітне значення не може зберігати великий проміжок часу (переповнення лічильника за частоти переривання таймера 60 Гц настане упродовж двох років);

2. 64-бітне значення на 32-бітному процесорі (в архітектурі IA-32) оброблятиметься неефективно.

Визначення системного часу у Windows XP

Для визначення поточного системного часу у Windows XP, використовується функція ***GetSystemTime():***

VOID GetSystemTime(LPSYSTEMTIME time);

time - покажчик на структуру SYSTEMTIME із полями, що задають елементи системного часу: wYear (рік), wMonth (місяць від одиниці) і т. д. аж до wMilliseconds (мілісекунди).

```
SYSTEMTIME ctime;
GetSystemTime(&ctime);
printf( "Зараз %02d/02d/%/d%02d/%02d\n",  ctime,wDay, ctime,wMonth, ctime,wYear, ctime,wHour, ctime,wMinute);
```

Windows XP постійно коригує системний час за годинником комп'ютера, тому використовувати результат виконання цієї функції для визначення проміжку часу не рекомендовано.

4.2 Керування таймерами відкладеного виконання

Іноді процесу потрібно, щоб система сповістила його про закінчення заданого проміжку часу.

Для процесів у системі створюють таймери відкладеного виконання, їх об'єднують у чергу, на початку якої перебуває таймер, що має спрацювати першим (поточний таймер); у ньому зберігають число, яке показує, скільки переривань таймера залишилося до його спрацювання. Кожний наступний таймер у черзі містить число, яке вказує, скільки переривань таймера залишиться до його спрацювання після того, як спрацював попередній.

Для кожного переривання таймера ОС зменшує на одиницю число, котре зберігають у поточному таймері. Коли воно досягає нуля — таймер спрацьовує і його вилучають із черги, а поточним стає наступний за ним.

Аналогічні таймери використовують у ядрі для керування деякими апаратними пристроями. Наприклад, дисковід гнучких дисків не можна використати відразу після ввімкнення двигуна, йому потрібен час для розгону.

Для розв'язання цього завдання драйвер диска встановлює таймер після включення двигуна так, щоб він спрацював через час, необхідний для розгону. Після спрацювання такого сторожового таймера вважають, що дисковід готовий до роботи.

Таймери очікування у Win32

Аналогами інтервальних таймерів у Win32 є ***таймери очікування***. Такі таймери є синхронізаційними об'єктами, із ними можна використовувати функції очікування. Сигналізація таймерів очікування відбувається через заданий час (можна періодично).

Є два види таймерів очікування: таймери синхронізації і таймери із ручним скиданням. Таймер синхронізації у разі сигналізації переводить у стан готовності до виконання всі потоки, які на ньому очікували, а таймер із ручним скиданням - тільки один потік.

Для створення таймера очікування необхідно використовувати функцію ***CreateWaitableTimer():***

HANDLE CreateWaitableTimer(LPSECURITY_ATTRIBUTES psa, BOOL manual_reset, LPCTSTR name);

manual_reset визначає тип таймера (TRUE - таймер із ручним скиданням). Ця функція повертає дескриптор створеного таймера.

Після створення таймер перебуває в неактивному стані. Для його активізації і керування станом використовують функцію ***SetWaitableTimer():***

*BOOL SetWaitableTimer(HANDL E ht, const LARGEJNTEGER *endtime, LONG period, PTIMERAPCROUTINE pfun, LPVOID pfun_arg, BOOL resume);*

ht - дескриптор таймера;

endtime - час, коли спрацює таймер (за негативного значення — задано відносний інтервал, за позитивного — абсолютний час, вимірюваний у 10^7 с);

period - період наступних спрацювань таймера (у мілісекундах), нуль - якщо таймер повинен спрацювати один раз;

pfun - функція користувача, яку викликатимуть у разі спрацювання таймера.

5 Керування введенням-виведенням: Windows XP

5.1 Основні компоненти підсистеми введення-виведення

Базовим компонентом підсистеми введення-виведення Windows XP є менеджер введення-виведення.

Передавання даних між рівнями підсистеми

Обмін даними між рівнями підсистеми введення-виведення є асинхронним. Більшу частину даних подано у вигляді пакетів, які передають від одного компонента підсистеми до іншого, можливо, змінюючи на ходу. Підсистема Windows XP є *керованою пакетами* . Такі пакети називають *пакетами запитів введення-виведення*, для стислості називатимемо їх *пакетами IRP*.

Менеджер введення-виведення створює пакет IRP, що відображає операцію введення-виведення, передає покажчик на нього потрібному драйверу і вивільняє пам'ять з-під нього після завершення операції. Драйвер, у свою чергу, отримує такий пакет, виконує визначену в ньому операцію і повертає його назад менеджерові введення-виведення як індикатор завершення операції або для передавання іншому драйверу для подальшої обробки.

Категорії драйверів пристроїв

Windows XP дає змогу використати кілька категорій драйверів режиму ядра. Найбільше поширення останнім часом набули ***WDM-драйвери*.**

Такі драйвери мають відповідати вимогам стандарту, який називають *Windows Driver Model* (WDM). Розрізняють три типи WDM-драйверів.

1 ***Драйвери шини*** керують логічною або фізичною шиною (PCI, USB, ISA). Драйвер відповідає за виявлення пристроїв, з'єднаних із певною шиною.
2 ***Функціональні драйвери*** керують пристроєм конкретного типу. Драйвери шини надають пристрої функціональним драйверам. Тільки функціональний драйвер працює з апаратним забезпеченням пристрою, саме він дає змогу системі 11використати пристрій.
3 ***Драйвери-фільтри*** доповнюють або змінюють поведінку інших драйверів.

Windows підтримує також драйвери режиму користувача. До них належать драйвери принтерів, які перетворюють незалежні від пристрою запити GDI-підсистеми в команди відповідного принтера і передають ці команди WDM-

драйверу (наприклад, драйверу паралельного порту або універсальному драйверу USB-принтера).

Підтримка конкретного пристрою може бути розділена між кількома драйверами. Залежно від рівня цієї підтримки виділяються додаткові категорії драйверів.

Клас-драйвери. Реалізують інтерфейс обробки запитів введення - виведення, специфічних для конкретного класу пристроїв, наприклад драйвери дисків або пристроїв DVD.

Порт-драйвери. Реалізують інтерфейс обробки запитів введення - виведення, специфічних для певного класу портів введення-виведення; зокрема до цієї категорії належить драйвер підтримки SCSI.

Мініпорт-драйвери. Керують реальними пристроями (наприклад, SCSI-адаптерами конкретного типу) і реалізують інтерфейс, наданий клас-драйверами і порт-драйверами.

Структура драйвера пристрою

Основні процедури драйвера:

Процедура ініціалізації. Її виконує менеджер введення-виведення під час завантаження драйвера у систему, і вона здійснює глобальну ініціалізацію структур даних драйвера.

Процедура додавання пристрою. Менеджер Plug and Play викликає цю процедуру, якщо знаходить пристрій, за який відповідає драйвер. У ній створюють структуру даних, відображувану пристроєм (об’єкт пристрою).

Набір процедур диспетчеризації. Ці процедури реалізують дії, допустимі для пристрою (відкриття, закриття, читання, запису тощо). Їх викликає менеджер введення-виведення під час виконання запиту.

Процедура обробки переривання. Вона є оброблювачем переривання від пристрою, виконується із високим пріоритетом; основне її завдання – запланувати для виконання нижню половину оброблювача (DPC- процедуру).

Процедура відкладеної обробки переривання, DPC- процедура, відповідає коду нижньої половини оброблювача переривання. Вона виконує більшу частину роботи, пов’язаної з обробкою переривання, після чого сигналізує про необхідність переходу до коду завершення введення-виведення.

5.2 Виконання операції введення-виведення для пристрою

У Windows на внутрішньому рівні всі операції введення-виведення, відображені пакетами IRP, є асинхронними. Будь-яку операцію синхронного введення-виведення відображають у вигляді сукупності асинхронної операції й операції очікування.

Процес обробки запиту синхронного введення-виведення до однорівневого драйвера зводиться до такого:

Запит введення-виведення перехоплює динамічна бібліотека підсистеми (наприклад, підсистема Win32 перехоплює виклик функції ***WriteFile()*.**

Динамічна бібліотека підсистеми викликає внутрішню функцію ***NtWriteFile(),*** що звертається до менеджера введення-виведення.

Менеджер введення-виведення розміщує у пам'яті пакет IRP, що описує запит, і відсилає його відповідному драйверу пристрою викликом функції ***IoCallDriver().***

5.3 Передавання параметрів драйверу пристрою

У WinAPI є функція, що викликає команду драйвера пристрою і передає йому необхідні параметри. Це функція ***DeviceIoControl().***

BOOL DeviceIoControl(HANDLE hd, DWORD ioc_code, LPVOID in_buf, DWORD in_bufsize, LPVOID out_buf, DWORD out_bufsize, LPDWORD pbytes_returned, LPOVERLAPPED ov);

hd - дескриптор пристрою, відкритого ***CreateFile();*** це може бути том, каталог тощо;

ioc_code - код команди драйвера;

in_buf - буфер із вхідними даними для виклику, і*n_bufsize* - його довжина;

out_buf - буфер з вихідними даними виклику, *out_bufsize*- його довжина;

pbytes_returned — покажчик на пам'ять, у яку буде збережено дані, поміщені в *out_buf*.

Приклади розробки програм

Приклад 1.Асинхронний запис у файл

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
```

```
#include <iostream.h>
//#include <vcl.h>

int main()
{
 HANDLE  hFile;     //дескриптор файлу
 OVERLAPPED  ovl;   //структура керування асинхронним доступом
                    //до файлу
 //ініціалізуємо структуру OVERLAPPED
 ovl.Offset = 0;      //молодша частина зміщення рівна 0
 ovl.OffsetHigh = 0;  //старша частина зміщення рівна 0
 ovl.hEvent = 0;      //події немає

 //створюємо файл для запису даних
 hFile = CreateFile(
       "C:\\demo_file.txt",   //ім'я файлу
       GENERIC_WRITE,         //запис у файл
       FILE_SHARE_WRITE,      //спільний доступ до файлу
       NULL,                  //захисту немає
       OPEN_ALWAYS,           //відкриваємо або створюємо новий файл
       FILE_FLAG_OVERLAPPED,  //асинхронний доступ до файлу
       NULL                   // шаблона немає
 );
 //перевіряємо на успішне створення
 if (hFile == INVALID_HANDLE_VALUE)
 {
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish.";
  cin.get();
  return 0;
 }
 //пишемо дані в файл
 for (int i = 0; i < 10; ++i)
 {
  DWORD  dwBytesWrite;
  DWORD  dwRet;
```

```
  if (!WriteFile(
              hFile,          //дескриптор файла
              &i,             //адреса буфера, звідки йде запис
              sizeof(i),      //кількість байтов, що записуються
              &dwBytesWrite,  //кількість записаних байтів
              &ovl            //запис асинхронний
   ))
  {
   dwRet = GetLastError();
   if (dwRet == ERROR_IO_PENDING)
    cout << "Write file pending." << endl;
   else
   {
    cout << "Write file failed." << endl
     << "The last error code: " << dwRet << endl;
    return 0;
   }
  }
      //чекаємо, поки завершиться асинхронна операція запису
  WaitForSingleObject(hFile, INFINITE);
      //збільшуємо зміщення у файлі
  ovl.Offset += sizeof(i);
 }
 //закриваємо дескриптор файла
 CloseHandle(hFile);

 cout << "The file is written." << endl;

 return 0;
}
```

Приклад 2.Асинхронний запис у файл з використанням події

```
//асинхронний запис у файл
#define _WIN32_WINNT 0x0400
#include <windows.h>
```

```
#include <iostream.h>

int main()
{
  HANDLE  hFile;     //дескриптор файлу
  HANDLE  hEndWrite; //дескриптор події
  OVERLAPPED  ovl;   //структура керування асинхронним
                     //доступом к файлу
  //створюємо подію з автоматичним скиданням
  hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEndWrite == NULL)
    return GetLastError();

  //ініціалізуємо структуру OVERLAPPED
  ovl.Offset = 0;          //молодша частина зміщення рівна 0
  ovl.OffsetHigh = 0;      //старша частина зміщення рівна 0
  ovl.hEvent = hEndWrite;  //подія для оповіщення завершення
                           //запису
  //створюємо файл для запису даних
  hFile = CreateFile(
        "C:\\demo_file.txt",   //ім'я файлу
        GENERIC_WRITE,         //запис у файл
        FILE_SHARE_WRITE,      //спільний доступ до файлу
        NULL,                  //захисту немає
        OPEN_ALWAYS,           //відкриваємо або створюємо новий файл
        FILE_FLAG_OVERLAPPED,  //асинхронний доступ до файлу
        NULL                   // шаблону немає
  );
  //перевіряємо на успішне створення
  if (hFile == INVALID_HANDLE_VALUE)
  {
    cerr << "Create file failed." << endl
      << "The last error code: " << GetLastError() << endl;
    CloseHandle(hEndWrite);

    cout << "Press any key to finish.";
    cin.get();
```

```
  return 0;
}
//пишемо дані у файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwBytesWrite;
  DWORD  dwRet;

  if (!WriteFile(
              hFile,          //дескриптор файлу
              &i,             //адреса буфера, звідки йде запис
              sizeof(i),      //кількість байтів, що записуються
              &dwBytesWrite,  //кількість записаних байтів
              &ovl            //запис асинхронний
    ))
  {
    dwRet = GetLastError();
    if (dwRet == ERROR_IO_PENDING)
      cout << "Write file pending." << endl;
    else
    {
      cout << "Write file failed." << endl
        << "The last error code: " << dwRet << endl;
              return 0;
    }
  }
      //чекаємо, поки завершиться асинхронна операція запису
  WaitForSingleObject(hEndWrite, INFINITE);
      //збільшуємо зміщення у файлі
  ovl.Offset += sizeof(i);
}
//закриваємо дескриптори
CloseHandle(hFile);
CloseHandle(hEndWrite);

cout << "The file is written." << endl;
```

```
  return 0;
}
```

Приклад 3. Визначення стану асинхронного запису даних

```
#define _WIN32_WINNT 0x0400
#include <windows.h>
#include <iostream.h>

int main()
{
  HANDLE  hFile;       //дескриптор файлу
  HANDLE  hEndWrite;   //дескриптор події
  OVERLAPPED  ovl;     //структура керування асинхронним доступом
                       //до файлу
  //створюємо подію з автоматичним скиданням
  hEndWrite = CreateEvent(NULL, FALSE, FALSE, NULL);
  if (hEndWrite == NULL)
    return GetLastError();

   //ініціалізуємо структуру OVERLAPPED
  ovl.Offset = 0;          //молодша частина зміщення рівна 0
  ovl.OffsetHigh = 0;      //старша частина зміщення рівна 0
  ovl.hEvent = hEndWrite;  //подія для повідомлення завершення
                           //запису
  //створюємо файл для запису даних
  hFile = CreateFile(
        "C:\\demo_file.dat",   //ім/я файлу
        GENERIC_WRITE,         //запис у файл
        FILE_SHARE_WRITE,      //спільний доступ до файлу
        NULL,                  //захисту немає
        OPEN_ALWAYS,           //відкриваємо або створюємо новий файл
        FILE_FLAG_OVERLAPPED,  //асинхронний доступ до файлу
        NULL                   //шаблону немає
  );
```

```
//перевіряємо на успішне створення
if (hFile == INVALID_HANDLE_VALUE)
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;

  CloseHandle(hEndWrite);

  cout << "Press any key to finish.";
  cin.get();

  return 0;
}
//пишемо дані у файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwBytesWrite;
  DWORD  dwNumberOfBytesTransferred;
  DWORD  dwRet;

  if (!WriteFile(
              hFile,        //дескриптор файлу
              &i,           //адреса буфера, звідки йде запис
              sizeof(i),    //кількість байтів, що записуються
              &dwBytesWrite, //кількість записаних байтів
              &ovl          //запис асинхронний
    ))
  {
    dwRet = GetLastError();
    if (dwRet == ERROR_IO_PENDING)
      cout << "Write file pending." << endl;
    else
    {
      cout << "Write file failed." << endl
        << "The last error code: " << dwRet << endl;

      return 0;
```

```
    }
  }
      //перевіряємо стан асинхронної операції запису
  if (!GetOverlappedResult(
            hFile,      //дескриптор файлу
            &ovl,       //адреса структури для асинхронної работи
            &dwNumberOfBytesTransferred,  //кількість переданих
                  //байтів
            FALSE       //не чекати завершення операції запису
   ))
  {
    cout << "Get overlapped result failed." << endl
      << "The last error code: " << GetLastError() << endl;

    return 0;
  }
  else
    cout << "Number of bytes transferred: "
        << dwNumberOfBytesTransferred << endl;
      //чекаємо завершення асинхронної операції запису
  WaitForSingleObject(hEndWrite, INFINITE);
      //збільшуємо зміщення у файлі
  ovl.Offset += sizeof(i);
 }
 //закриваємо дескриптори
 CloseHandle(hFile);
 CloseHandle(hEndWrite);

 cout << "The file is written." << endl;
 cin.get();

 return 0;
}
```

Приклад 4. Робота з портом завершення

```
#define _WIN32_WINNT 0x0400
```

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

HANDLE  hCompletionPort;   // порт завершення

DWORD WINAPI thread(LPVOID)
{
  int  i = 0;                              //кількість отриманих пакетів
  DWORD  dwNumberOfBytes;      //кількість переданихх байтів
  ULONG    ulCompletionKey;       //ключ файлу
  LPOVERLAPPED  lpOverlapped;        //покажчик на структуру типу
                          //OVERLAPPED
  cout << "The thread is started." << endl;

  //підключаємо потік до порту
  while (GetQueuedCompletionStatus(
        hCompletionPort,   //дескриптор порту завершення
        &dwNumberOfBytes,  //кількість переданихх байтів
        &ulCompletionKey,  //ключ файлу
        &lpOverlapped,     //покажчик на структуру типу OVERLAPPED
        INFINITE))        //нескінченне очікування
//  {---тіло циклу---

        // перевіряємо пакет на завершення виведення
    if (ulCompletionKey == 0)
    {
      cout << endl << "The thread is finished." << endl;
      break;
    }
    else
      cout << "\tPacket: " << ++i << endl
        << "Number of bytes: " << dwNumberOfBytes << endl
        << "Completion key: " << ulCompletionKey << endl;
//  }---кінець тіла циклу---
  return 0;
}
```

```
int main()
{
 HANDLE  hFile;     //дескриптор файлу
 OVERLAPPED  ovl;   //структура керування асинхронним доступом к
файлу
 ULONG    ulKey;    //ключ файлу
 HANDLE  hThread;   //масив для дескрипторів потоків
 DWORD  dwThreadID; //масив для ідентифікаторів потоків

 //ініціалізуємо структуру OVERLAPPED
 ovl.Offset = 0;      //молодша частина зміщення рівна 0
 ovl.OffsetHigh = 0;  //старша частина зміщення рівна 0
 ovl.hEvent = 0;      //події немає

 //запитуємо ключ файлу
 cout << "Input a number for file key (not zero): ";
 cin >> ulKey;
 if (ulKey == 0)
 {
  cout << "The file key can't be equal to zero." << endl
    << "Press any key to exit." << endl;

  return 0;
 }

 //створюємо файл для запису даних
 hFile = CreateFile(
       "C:\\demo_file.dat",   //ім'я файлу
       GENERIC_WRITE,        //запис у файл
       FILE_SHARE_WRITE,     //спільний доступ до файлу
       NULL,                 //захисту немає
       OPEN_ALWAYS,          //відкриваєм або створюємо новий файл
       FILE_FLAG_OVERLAPPED,  //асинхронний доступ до файлу
       NULL                  //шаблона немає
 );
 //перевіряємо на успішне створення
 if (hFile == INVALID_HANDLE_VALUE)
```

```
{
  cerr << "Create file failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish." << endl;

  cin.get();
  return 0;
}

// створюємо порт завершення та підключаємо до нього файл
hCompletionPort = CreateIoCompletionPort(
        hFile,   //дескриптор файлу
        NULL,    //новий порт
        ulKey,   //ключа файлу
        1        //один потік
  );

//перевіряємо на успішне створення
if (hCompletionPort == NULL)
{
  cerr << "Create completion port failed." << endl
    << "The last error code: " << GetLastError() << endl;
  cout << "Press any key to finish." << endl;

  cin.get();
  return 0;
}

//запускаємо потік
hThread = CreateThread(NULL, 0, thread, NULL, 0, &dwThreadID);

//пишемо дані у файл
for (int i = 0; i < 10; ++i)
{
  DWORD  dwBytesWrite;
  DWORD  dwRet;
```

```
 if (!WriteFile(
            hFile,      //дескриптор файлу
            &i,         //адреса буфера, звідки йде запис
            sizeof(i),  //кількість байтів, що записуються
            &dwBytesWrite,  //кількість записаних байтів
            &ovl        //запис асинхронна
  ))
 {
  dwRet = GetLastError();
  if (dwRet == ERROR_IO_PENDING)
   cout << "Write file pending." << endl;
  else
  {
   cout << "Write file failed." << endl
    << "The last error code: " << dwRet << endl;
   return 0;
  }
 }
     //чекаємо, поки завершиться асинхронна операція запису
 WaitForSingleObject(hFile, INFINITE);
     //збільшуємо зміщення у файлі
 ovl.Offset += sizeof(i);
}

//посилаємо пакет з командою на завершення потоку
PostQueuedCompletionStatus(
     hCompletionPort,  //дескрипор потоку
     0,     //немає передачі
     0,     //ключ завершення
     NULL);  //немає структури типу OVERLAPPED

//чекаємо завершення потоку
WaitForSingleObject(hThread, INFINITE);
//закриваємо дескриптори
CloseHandle(hFile);
CloseHandle(hCompletionPort);
CloseHandle(hThread);
```

```
  cout << "The file is written." << endl;

  return 0;
}
```

Приклад 5. Створення та установка очікуючого таймера

```
// #define _WIN32_WINNT 0x0400
#include <windows.h>
#include <iostream.h>
#define _SECOND 10000000          //одна секунда для очікуючого таймера

HANDLE  hTimer;                   //очікуючий таймер

DWORD WINAPI thread(LPVOID)
{
  //чекаємо сигналу від очікуючого таймера
  WaitForSingleObject(hTimer, INFINITE);
  //виводимо повідомлення
  cout << "\aThe timer is signaled." << endl;
  return 0;
}
int main()
{
  HANDLE  hThread;
  DWORD  IDThread;

  DWORD  qwTimeInterval;    //час затримки для таймера

  //створюємо очікуючий таймер
  hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
  if (hTimer == NULL)
    return GetLastError();

  //час затримки для таймера = 3 секунди
  qwTimeInterval = -3 * _SECOND;
```

```
 //ініціалізуємо таймер
 if (!SetWaitableTimer(
        hTimer,  //дескриптор таймера
        (LARGE_INTEGER*)&qwTimeInterval,   //часовий інтервал
        0,      //не періодичний таймер
        NULL,    //процедури завершення немає
        NULL,    //параметрів до цієї процедури немає
        FALSE    //режим збереження енергії не установлювати
   ))
 {
   cout << "Set waitable timer failed." << endl
     << "The last error code: " << GetLastError() << endl;

   return 0;
 }
 //запускаємо потік
 hThread = CreateThread(NULL, 0, thread, NULL, 0, &IDThread);
 if (hThread == NULL)
   return GetLastError();

 //чекаємо, поки потік закінчить роботу
 WaitForSingleObject(hThread, INFINITE);
 // закриваємо дескриптори
 CloseHandle(hThread);
 CloseHandle(hTimer);

 cin.get()  ;

 return 0;
}
```

Завдання для студентів

1. Розробити програми роботи керування пристроями введення-виведення. Для програмної реалізації задач використовувати середовище програмування CodeGear C++ Builder. Програми розробляти у консольному режимі.

2. Скласти звіт про виконання лабораторної роботи.
   Зміст звіту:
   - опис функцій для роботи з файловою системою;
   - постановка задачі;
   - програмний код.
   - схеми алгоритмів для кожної програми.
3. При захисті лабораторної роботи підготувати відповіді на контрольні питання.

Контрольні питання

1. Як виконується опитування пристроїв?
2. Як виконується введення-виведення, кероване перериваннями?
3. Як виконується встановлення оброблювачів переривань?
4. Як виконується відкладена обробка переривань?
5. Охарактеризувати прямий доступ до пам'яті.
6. Охарактеризувати планування операцій введення-виведення.
7. Як виконується буферизація введення-виведення?
8. Охарактеризувати введення-виведення із розподілом та об'єднанням.
9. Охарактеризувати спулінг.
10. Як виконується синхронне введення-виведення?
11. Охарактеризувати багатопотокову організацію введення-виведення.
12. Охарактеризувати введення-виведення у режимі користувача.
13. Охарактеризувати порти завершення введення-виведення.
14. Як виконується керування системним часом?
15. Як виконується визначення системного часу у Windows?
16. Охарактеризувати керування таймерами відкладеного виконання.
17. Охарактеризувати таймери очікування у Win32.
18. Назвати основні компоненти підсистеми введення-виведення.
19. Охарактеризувати структуру драйвера пристрою.
20. Охарактеризувати виконання операцій введення-виведення для пристрою.
21. Як виконується передавання параметрів драйверу пристрою?

ЛІТЕРАТУРА

1. Бондаренко М.Ф., Качко О.Г. Операційні системи. - К.: СМІТ, 2008. - 432с.
2. Бэкон Дж., Харрис Т. Операционные системы. – К.: Издат. группа BHV; СПб.: Питер, 2004. – 800 с.
3. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001. – 736 с.
4. Танненбаум Э. Операционные системы. – СПб.: Питер, 2002. – 1040 с.
5. Таненбаум Э. Современные операционные системы. – СПб.: Питер, 2007. – 1040 с.
6. Шеховцов В.А. Операційні системи. - К.: Видавнича група BHV. 2005. – 576 с.