МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЧЕРКАСЬКИЙ ДЕРЖАВНИЙ ТЕХНОЛОГІЧНИЙ УНІВЕРСИТЕТ

ЧЕРКАСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ БОГДАНА ХМЕЛЬНИЦЬКОГО

ХАРКІВСЬКИЙ УНІВЕРСИТЕТ ПОВІТРЯНИХ СИЛ ІМЕНІ ІВАНІ КОЖЕДУБА

В. М. Рудницький, С. В. Бєсєдіна, Г. А. Кучук

ДОСЛІДЖЕННЯ І ПРОЕКТУВАННЯ ДРАЙВЕРІВ ОПЕРАЦІЙНИХ СИСТЕМ

Навчальний посібник

Черкаси, Харків, 2010

УДК 004ю454
ББК 32 973я73
Р 64

Р е ц е н з е н т и: ***О. Г. Корченко*** – доктор технічних наук, професор, завідувач кафедри безпеки інформаційних технологій Національного авіаційного університету, Київ; ***О. А. Кочкарьов*** – доктор технічних наук, професор кафедри інформатики та інформаційної безпеки Черкаського державного технологічного університету; ***І. В. Шостак*** – доктор технічних наук, доцент, професор кафедри програмної інженерії Національного аерокосмічного університету ім. М.Є. Жуковського "ХАІ".

Рекомендовано до видання рішенням Вченої ради Черкаського державного технологічного університету, протокол №1 від 23.09.2010 р.

Рудницький В. М.

Р64 Дослідження і проектування драйверів операційних систем: навч. посібник / В.М. Рудницький, С.В. Бєсєдіна, Г.А. Кучук. – Черкаси: Черкаський державний технологічний університет, 2010. – 216 с.

ISBN 978-966-676-362-5

У навчальному посібнику авторами висвітлюються найсуттєвіші питанн, пов'язані з дослідженням і проектуванням драйверів на прикладі операційної системи Windows NT 5. Матеріал посібника подано у формі стислого викладу тематичного матеріалу, містить контрольні запитання та задачі для освоєння лекційного курсу, тестові завдання для самоконтролю. Окрім того, в кінці кожної теми додається список необхідної літератури.

У цілому навчальний посібник "Дослідження і проектування драйверів операційних систем" значно поширює поле досліджень при вивченні сучасних операційних систем, формує базу для подальшого поглибленого вивчення системного програмування, і тому, без сумніву, буде корисним для викладачів та студентів, а також зацікавить науковців та спеціалістів-практиків.

ISBN 978-966-676-362-5

ЗМІСТ

СПИСОК СКОРОЧЕНЬ

ACPI – Advanced Configuration and Power Interface
API – Application Programming Interfaces
APC – Asynchronous Procedure Call
AWE – Address Windowing Extension
CSR – Control and Status Register
DDK – Driver Development Kit
DMA – Direct Memory Access
DPC – Deferred Procedure Call
FIDO – Filter Device Object
GDI – Graphical Device Interface
GUI – Graphical User Interface
HAL – Hardware Abstraction Layer
HCT – Hardware Compatibility Tests
HID – Human Interface Device
IOCTL – Input/Output Control Code
IRP – Input/Output Request Packet
IRQL – Interrupt ReQuest Level
ISR – Interrupt Service Routine
KMDF – Kernel Mode Driver Framework
LPC – Local Procedure Call
NT – New Technology
OHCI – Open Host Controller Interface
PCMCIA – Personal Computer Card International Association
PAE – Physical Address Extension
PDB – Program Database
PIO – Programmed I/O
PNP – Plug and Play
RPC – Remote Procedure Call
SCM – Service Control Manager
UMDF – User Mode Driver Framework
VDM – Virtual DOS Machine
VxD – Virtual Device Driver
WDM – Windows Driver Model
WDI – Windows Management and Instrumentations
WHQL – The Windows Hardware Quality
WOW – Windows on Windows
ДВВ – диспетчер введення/виведення
ОС – операційна система
ПЗП – постійний запам’ятовуючий пристрій

ВСТУП

Актуальним завданням суспільства на сьогодні є подальший розвиток інформаційних технологій. Стрімкий розвиток відповідної апаратної та програмної бази ставить ряд питань, серед яких питання, пов'язані з дослідженням, проектуванням та розробкою драйверів операційних систем, безсумнівно необхідні для системних програмістів та системних адміністраторів комп'ютерних мереж.

В основу навчального посібника "Дослідження і проектування драйверів операційних систем" закладено зміст нормативної дисципліни "Дослідження і проектування драйверів операційних систем", яка є базовою для підготовки магістрів з таких спеціальностей: 8.0501201 – «Компютерні системи та мережі»; 8.05010202 – «Системне програмування».

Основними завданнями посібника є такі: ознайомити читачів із сутністю та теоретичними засадами дослідження та проектування драйверів; навчити правильно застосовувати відповідний методичний інструментарій; сформувати теоретичну та методологічну базу, необхідну для подальшого оволодіння практикою розробки драйверів.

У посібнику розкриті основні положення, що стосуються дослідження і проектування драйверів операційних систем, згідно з програмою відповідної нормативної дисципліни послідовно розкриті такі питання: архітектура родини операційних систем WINDOWS NT; апаратне забезпечення персонального комп'ютера; інтерфейс NATIVE API WINDOWS NT; драйверна модель WDM; основні прийоми програмування драйверів в режимі ядра; структура LEGACY –драйвера та його основні процедури; використання диспетчера керування службами для завантаження/вивантаження драйверів; методики тестування та відладки драйверів; використання INF-файлів для інсталяції драйверів. Також додані глосарій та базові терміни розробки драйверів під операційною системою.

Матеріал посібника подано у формі стислого викладу тематичного матеріалу, містить контрольні запитання, а в кінці кожної теми подається список необхідної літератури. Зміст посібника викладено з урахуванням особливостей вітчизняного та зарубіжного досвіду у напряму дослідження, проектування та розробки драйверів операційних систем.

Даний посібник буде корисним для використання студентами денної, заочної та дистанційної форм навчання, а також факультетів перепідготовки і підвищення кваліфікації. Сподіваємось, що даний посібник зацікавить науковців та спеціалістів-практиків.

ЧАСТИНА 1. ДОСЛІДЖЕННЯ ДРАЙВЕРІВ ОПЕРАЦІЙНИХ СИСТЕМ

РОЗДІЛ 1. АРХІТЕКТУРА РОДИНИ ОПЕРАЦІЙНИХ СИСТЕМ WINDOWS NT

План

- Фундаментальні цілі створення операційної системи Windows NT
- Архітектура операційної системи Windows NT
- Різниця між версіями операційної системи Windows NT

У даному розділі розглядаються основні цілі створення операційної системи Windows NT, характерні архітектурні особливості, а саме режим ядра, виконувана частина, особливості роботи ядра, виконавчі компоненти, підсистеми середовища, а також проведено порівняльну характеристику між різними версіями операційної системи Windows NT.

1.1. Фундаментальні цілі створення операційної системи Windows NT

На початку 1989 року концепції Windows NT (New Technology) почали набувати рис реальності, де і сьогодні незмінними залишаються п’ять фундаментальних завдань NT:

1. *Сумісність*. Операційна система (ОС) повинна підтримувати максимально можливу кількість програмного й апаратного забезпечення.
2. *Переносимість*. ОС повинна функціонувати на максимально можливій кількості існуючих та очікуваних в перспективі апаратних платформ.
3. *Розширюваність*. Оскільки вимоги ринків постійно ростуть, ОС повинна уміти з легкістю розширювати набір своїх можливостей і перелік підтримуваної апаратури при мінімальному втручанні в свій внутрішній код.
4. *Надійність і стійкість*. ОС повинна бути стійкою до ненавмисного або навмисного неправильного використання компонентів. Призначені для користувача застосування не повинні мати можливості приведення системи до фатальних збоїв.
5. *Продуктивність*. ОС повинна забезпечувати хорошу продуктивність на всіх апаратних платформах, що підтримуються.

Розглянемо дані завдання операційної системи Windows NT більш детально.

1.1.1. Сумісність

Для досягнення стійкості в роботі системи, розробники NT обрали для побудови ядра архітектуру “клієнт-сервер”. Дане застосування призначене для користувача, яке є клієнтом служб ОС. Воно функціонує в спеціальному режимі (щодо апаратного забезпечення), який називається “user mode” – режим користувача. В межах цього режиму, код застосування обмежений виконанням “нешкідливих” інструкцій. Тобто, якщо раптом застосуванню потрібно буде виконати будь-що з числа заборонених для нього дій, воно повинне запитати відповідну службу ОС.

Код самої ОС виконується в “kernel mode” – режимі ядра (режимі рівня ядра). Код режиму ядра має право виконати будь-яку процесорну інструкцію, не виключаючи інструкцій введення/виведення. Пам’ять, що належить будь-якому застосуванню, може бути доступна коду режиму ядра, звичайно, якщо сторінкова пам’ять застосування в даний момент не скинута на жорсткий диск.

1.1.2. Переносимість

Сучасні процесори реалізують декілька форм привілейованого режиму на відміну від непривілейованого. Код режиму ядра виконується в привілейованому контексті, тоді як призначений для користувача код виконується в непривілейованому середовищі. Оскільки різні процесори (і платформи на їх основі) реалізують привілейовані режими по-різному, то, для забезпечення переносимості, розробники ОС ввели особливі абстрактні елементи програмування, які дозволяють розмежовувати призначений для користувача режим і режим ядра. Код ОС використовує їх для перемикання привілейованого/непривілейованого контексту, отже, при перенесенні ОС тільки код цих додаткових елементів необхідно буде переписувати конкретно під специфіку нової апаратної платформи. На платформі Intel призначений для користувача режим реалізується з набору інструкцій Ring 3 (Кільця 3), тоді як режим ядра реалізований з використанням Ring 0 (Кільця 0).

Драйвери рівня ядра працюють в привілейованому контексті. Відповідно, погано написаний драйверний код може виявитися шкідливим для ОС. Розробник повинен з особливою увагою відноситися до створюваного коду, щоб не зруйнувати всю архітектуру ОС. Фірма Microsoft намагається вирішити проблему надійності драйверів, що поставляються у складі дистрибутива Windows, через механізм тестування і підписання драйверів, про що буде розглядатися в другій частині посібника у розділі 8.

Як спосіб рішення задачі переносимості конструктори Windows NT вибрали багатошарову архітектуру (рис. 1.1).

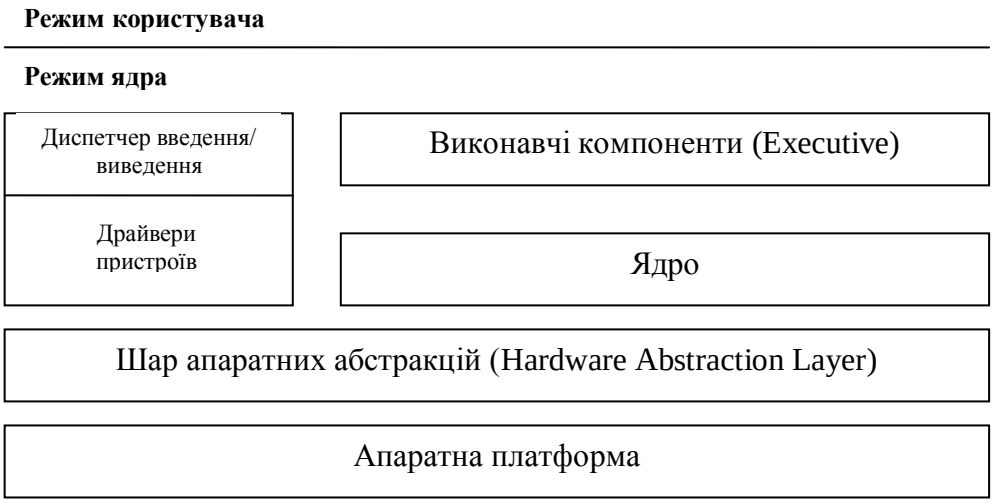

Рис. 1.1. Шари операційної системи Windows NT

Шар апаратних абстракцій (Hardware Abstraction Layer, HAL) ізолює процесорні й платформенні особливості від коду ОС. Тому саме цими послугами необхідно користуватися і розробникові драйверного коду. Якщо драйвер як програмна одиниця жорстко прив'язаний і до свого пристрою, і до конкретного процесора, і до конкретної платформи, то цілком можливо так написати його, що для перенесення його на іншу платформу потрібно буде хіба що перекомпілювати його. Драйвер повинен звернутися до використання засобів рівня HAL для взаємодії з апаратними регістрами і апаратною підтримкою шин. В окремих випадках розробник драйвера може посилатися на код, що надається Диспетчером введення/виведення, для роботи із апаратними ресурсами, що спільно використовуються.

1.1.3. Розширюваність

На рис. 1.1 позначена і ще одна важлива особливість представленої архітектури – ядро відокремлене від шару, який носить назву "Виконавчі компоненти" (Executive), що будуть розглядатися у п. 1.2.4.

В даному випадку, ядро несе відповідальність за планування активності програмних потоків (threads). Потік є лише "незалежною стежиною" у виконанні програмного коду. Щоб зберегти незалежність від діяльності інших потоків, для кожного з них необхідно зберігати унікальний потоковий контекст (thread context), який складається із стану регістрів процесора, включаючи також ізольований стек і лічильник інструкцій (Program Counter), збереженого ID (ідентифікатора потоку, Thread ID або TID), значення пріоритету, розподіли пам'яті, що пов'язані з потоком (Thread Local Storage), й іншої інформації, що має відношення до даного потоку.

Обов'язком планувальника потоків є визначення, який потік повинен виконуватися в даний момент. У середовищі з єдиним процесором, в кожен момент часу тільки один потік отримує в своє розпорядження процесор.

У багатопроцесорній конфігурації потоки можуть виконуватися на різних процесорах, реалізуючи справжнє паралельне виконання коду. Планувальник в більшості випадків потоку видає процесор на фіксований часовий інтервал, відомий під назвою thread time quantum (потоковий часовий квант), який надається процесором на основі пріоритету потоку.

Окрім основним завданням ядра є управління потоками, робота по управлінню пам'яттю, питаннями доступу (security) і діями з введенням/виведенням покладено на інші компоненти ОС. Вони конфігуровані як модулі програмного забезпечення (хоча диспетчер введення/виведення сам є тим включенням з цього правила).

1.1.4. Продуктивність

Особлива увага з боку розробників Windows NT була приділена швидкій міжшаровій взаємодії. Деякі шари виконуються в одному і тому ж режимі – режимі ядра. Отже, між шарами і виклики не використовують інших засобів, ніж звичайна процесорна CALL. З тих же, що надаються рівнем HAL, в основному є макро визначеннями. До того ж, розробники зробили багато зусиль, направлених на те, щоб змусити працювати паралельно максимально можливе число програмних потоків виконавчих компонентів. Допоміжні процедури рідко блокуються або входять в очікування, що дозволяє часто надання процесора.

Питання підвищення продуктивності в значній мірі стосуються і розробників драйверів. Тоді, коли прикладне для користувача застосування або системний потік посилає запит до обслуговуваного драйвером пристрою (об'єкту пристрою), життєво важливо, щоб драйверний код не блокував виконання запиту. Якщо ж запит не може бути оброблений негайно (наприклад, якщо пристрій зайнятий або повільно працює), запит повинен бути поставлений в чергу для подальшої обробки. Диспетчер введення/виведення надає необхідні для цього процедури.

1.2. Архітектура операційної системи Windows NT

До основних особливостей ОС Windows NT належать:

– модель з мікроядром;
– можливість емуляції інших ОС (на рівні прикладних систем);
– багатопоточність;
– інтегрована підтримка мережі.

Windows NT має модульну структуру й складається з двох основних рівнів – компонентів, що працюють у режимі користувача й компонентів режиму ядра. Програми й підсистеми, що працюють у режимі користувача мають обмеження на доступ до системних ресурсів. Режим ядра має необмежений доступ до системної пам'яті й зовнішніх пристроїв. Ядро системи

NT називають гібридним ядром або макроядром. Архітектура містить у собі саме ядро, рівень апаратних абстракцій, драйвера й ряд служб (Executives), які працюють у режимі ядра або в режимі користувача (див. рис. 1.1).

На рис. 1.2 приведена загальна архітектура Windows NT і її компонентів. Елементи представлені у блоці “User mode” є процесами призначеного для користувача режиму, а у блоці “Kernel mode” розташовуються процеси ОС, що виконуються ядром. Потоки, що призначені для режиму користувача виконуються в захищеному адресному просторі. Проте, під час їх виконання в режимі ядра, вони дістають доступ до системного простору. Таким чином, системні процеси, процеси сервера (служби), підсистема середовища або застосування призначені для користувача мають свій власний адресний простір.

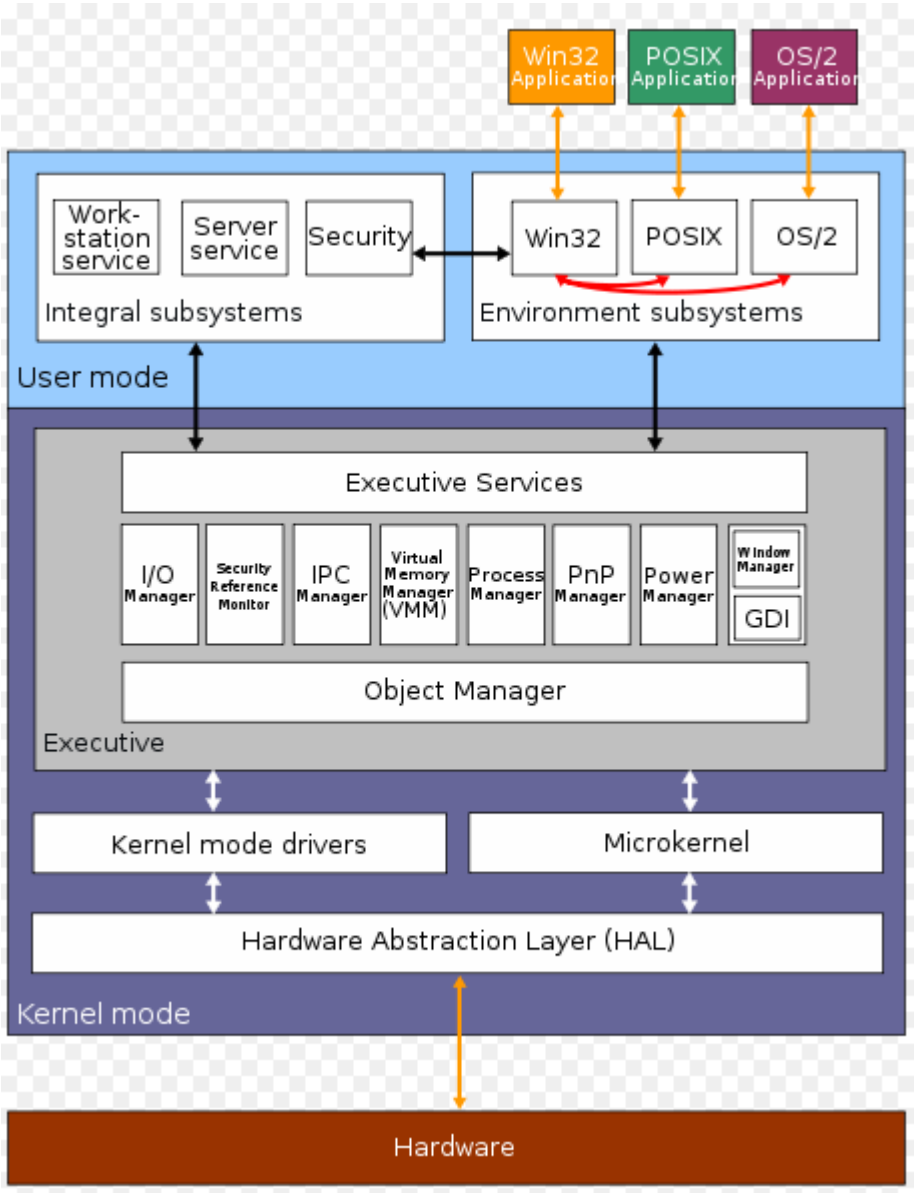

Рис. 1.2. Архітектура Windows NT

Режим користувача Windows NT складається з підсистем, що передають запити введення/виведення відповідному драйверу режиму ядра за допомогою

менеджера введення/виведення. Є дві підсистеми на рівні користувача: підсистема оточення (запускає додатки написані для різних ОС) й інтегрована підсистема (управляє особливими системними функціями від імені підсистеми оточення). Режим ядра має повний доступ до апаратної частини й системних ресурсів комп'ютера. І також запобігає доступ до критичних зон системи з боку користувальницьких служб і додатків. Більш детально компоненти архітектури Windows NT будуть розглянуті в п. 1.2.4.

1.2.1. Режим ядра операційної системи Windows NT

Важливі для продуктивності ОС компоненти виконуються в режимі ядра, де вони взаємодіють з устаткуванням і один з одним без використання перемикачів контексту і зміни режимів. Наприклад, менеджер пам'яті, менеджер кеш-пам'яті, менеджер об'єктів, менеджер системи безпеки, мережеві протоколи, файлові системи, управління потоками і процесами працюють в режимі ядра. Всі ці компоненти повністю захищені від виконуваних додатків, які не мають прямого доступу до коду і даних з привілейованої частини ОС.

Компоненти режиму ядра Windows NT спроектовані на основі принципів побудови об'єктно-орієнтованих систем. Наприклад, вони не працюють безпосередньо із структурами даних, підтримуваних індивідуальними компонентами. Замість цього для передачі параметрів, доступу і/або модифікації структур даних вони використовують формальний інтерфейс. Разом з тим, не дивлячись на повсюдне використання об'єктів для представлення системних ресурсів, що розділяються, Windows NT не є об'єктно-орієнтованою системою в точному сенсі цього поняття, оскільки основна частина коду системи написана на мові С з міркувань забезпечення високої швидкості виконання і переносимості.

У режимі ядра виконуються наступні компоненти ОС:

- виконувана частина NT, яка включає управління пам'яттю, процесами, потоками, безпекою, введенням/виведення, міжпроцесорними обмінами;
- ядро Windows NT виконує низькорівневі функції ОС: диспетчеризація потоків, переривань і виключень, синхронізація процесорів. Ядро також включає набір процедур і базових об'єктів, що використовуються виконуваною частиною для створення високорівневих конструкцій;
- шар абстракції від устаткування, ізолює ядро, драйвери пристроїв і виконувану частину NT від апаратних платформ, на яких повинна працювати ОС;
- драйвери пристроїв включають як файлову систему, так і апаратні драйвери, які транслюють призначені для користувача виклики функцій введення/виведення в запити фізичних пристроїв введення/виведення;
- функції графічного інтерфейсу користувача працюють з вікнами, елементами управління і рисунками.

1.2.2. Виконувана частина операційної системи Windows NT 5

Виконувана частина Windows NT – верхній шар програми – ядра NTOSKRNL.EXE (само ядро – це нижній шар) містить такі компоненти:

1. *Менеджер процесів і потоків* управляє процесами і потоками. Виконувана частина додає додаткову семантику і функції до цих об'єктів нижнього рівня.

2. *Менеджер віртуальної пам'яті* використовує схему управління, при якій кожен процес отримує власний достатньо великий адресний простір, захищений від дії інших процесів. Менеджер пам'яті також забезпечує низькорівневу підтримку для менеджера кеш-пам'яті.

3. *Монітор безпеки* проводить політику забезпечення заходів безпеки на локальному комп'ютері, охороняючи системні ресурси і виконуючи процедури аудиту і захисту об'єктів.

4. *Система введення/виведення* використовує незалежне від пристроїв введення/виведення і відповідає за пересилку даних відповідним драйверам для подальшої обробки.

5. *Менеджер кеш-пам'яті* покращує продуктивність системи введення/виведення файлів, розміщуючи прочитані з диска дані в основній пам'яті для прискорення доступу до них, а також відкладаючи на короткий час запис змінених даних на диск.

Крім того, виконувана частина включає чотири головні групи функцій, які використовують вище перераховані компоненти:

1. *Менеджер об'єктів*, який створює, видаляє об'єкти і абстрактні типи даних, а також управляє ними. Об'єкти використовуються в Windows NT для представлення таких ресурсів ОС, як процеси, потоки і об'єкти синхронізації.

2. *Локальний процедурний виклик* (Local Procedure Call, LPC) передає повідомлення між клієнтським процесом і процесом сервера на тому ж самому комп'ютері. По суті, LPC – це оптимізована версія відомої процедури видаленого виклику RPC (Remote Procedure Call), стандарту для організації взаємодії процесів в архітектурі клієнт/сервер.

3. *Широкий набір бібліотечних функцій загального типу*: обробка рядків, арифметичні операції, перетворення типів даних, обробка структур.

4. *Процедури розподілу пам'яті*, взаємообмін між процесами через пам'ять, два спеціальні типи об'єктів синхронізації – ресурси і об'єкти fast mutex.

1.2.3. Ядро операційної системи Windows NT 5

Ядро NTOSKRNL.EXE виконує більшість основних операцій NT, що визначають порядок використання процесора: диспетчеризація потоків; диспетчеризація і обробка виключень; синхронізація роботи процесорів; забезпечення базових об'єктів ядра, які використовуються виконуваною частиною (і в деяких випадках експортуються в режим користувача).

На відміну від решти виконуваної частини ОС, ядро ніколи не вивантажується з оперативної пам'яті, його виконання ніколи не уривається іншими потоками. Код ядра написаний в основному на мові С, а частини, що дають найбільше навантаження на процесор, на мові Асемблері.

Об'єкти ядра. Одна з функцій ядра – забезпечення низькорівневої бази для добре певних примітивів ОС, які забезпечують роботу компонентів вищого рівня. Ядро ізолює само себе від решти частини ОС, що дозволяє винести схвалення політичних рішень з ядра, за винятком диспетчеризації потоків. Ядро використовує набір простих об'єктів, які називаються об'єктами ядра. Вони дозволяють управляти роботою центрального процесора і порядком створення обчислюваних об'єктів. Один з наборів об'єктів називається об'єктами управління і включає об'єкт процесу ядра, об'єкт диспетчера викликів асинхронних процедур (Asynchronous Procedure Call, APC), об'єкт процедури відкладеного виклику (Deferred Procedure Call, DPC) і декілька об'єктів, що використовуються системою введення/виведення (наприклад, об'єкт обробки переривання).

Інший набір об'єктів ядра – об'єкти диспетчеризації, включає об'єкти синхронізації потоків, потік ядра, mutex, об'єкти події, семафора, таймера, таймера очікування і ряд інших.

Підтримка устаткування. Іншим найголовнішим завданням ядра є абстрагування (або ізоляція) виконуваної частини і драйверів пристроїв від відмінностей мікропроцесорних платформ, на яких здатна працювати Windows NT: x86 і Alpha AXP. Специфічними для архітектури є функції (такі, як контекстне перемикання потоку) реалізовані в ядрі. Вони реалізовані у складі HAL і можуть відрізнятися від машини до машини.

Ядро підтримує набір інтерфейсів, семантично ідентичних для всієї архітектури. Деякі з інтерфейсів по-різному реалізовані для різної архітектури, проте, й ідентичні зовні інтерфейси реалізовані за допомогою специфічного для архітектури коду. Незалежний від архітектури інтерфейс може бути викликаний на будь-якій машині, і його семантика буде тією ж, не дивлячись на те, чи залежить код від архітектури чи ні. Деякі інтерфейси ядра (реалізовані в HAL, оскільки їх реалізація може змінюватися навіть усередині одного сімейства комп'ютерів. Як приклад залежного від архітектури коду можна назвати підтримку кеша центрального процесора.

Абстракція від устаткування. Завантажуваний модуль ядра HAL забезпечує низькорівневий інтерфейс з апаратною платформою, що дозволяє приховати такі залежні від апаратури деталі, як інтерфейс введення/виведення, контролери переривань, механізм обміну даними між процесорами – будь-які апаратно-залежні і специфічні для архітектури функції.

Драйвери пристроїв. Драйвери пристроїв – це завантажувальні модулі, які працюють в режимі ядра, забезпечуючи інтерфейс між системою введення/виведення і відповідним устаткуванням. Назви цих модулів зазвичай мають розширення .SYS. Всі вони, як правило, написані на мові С (іноді С++) із використанням викликів процедур HAL і можуть бути переносними на рівні

двійкового коду між платформами, що підтримуються NT. Розрізняють декілька типів драйверів пристроїв, а саме:

1. *Драйвери, що маніпулюють пристроями* (з використанням HAL) для запису вихідних даних або отримання вхідних даних від фізичних пристроїв або через мережу.
2. *Драйвери файлової системи*, що приймають запити на файлове введення/виведення і транслюють їх в запити введення/виведення, які пов'язані з конкретними пристроями.
3. *Драйвери-фільтри*. Прикладом можуть бути драйвери підтримки дзеркальних дисків, шифрування даних, перехоплення введення/виведення для додаткової обробки даних перед передачею їх на наступний рівень і так далі.
4. *Мережеві драйвери*, які передають і приймають видалені запити на введення/виведення.

Оскільки установка драйверів пристроїв є єдиним способом додати до системи призначений для користувача код, що працює в режимі ядра, то деякі програмісти можуть розглядати написання драйверів пристроїв як спосіб доступу до внутрішніх функцій і структур даних ОС, недоступних з режиму користувача.

1.2.4. Виконавчі компоненти операційної системи Windows NT 5

Як видно з рис. 1.2, у режимі ядра (kernel mode) найнижчим є апаратний рівень. Наступним йде HAL, вище – диспетчер введення/виведення і драйвера пристроїв в одній зв'язці, а також ядерце разом з виконавчими компонентами. Ядро відокремлене від виконавчих компонентів, які реалізовані по модульній схемі, але декілька компонентів дану схему не підтримують, що в свою чергу веде до розширюваності системи. До найбільш важливих виконавчих компонентів можна віднести:

1. Інтерфейс системних служб (System Service Interface).
2. Менеджер конфігурації (Configuration Manager).
3. Диспетчер введення/виведення, ДВВ (I/O Manager).
4. Менеджер віртуальної пам'яті (Virtual Memory Manager,VMM).
5. Локальний процедурний виклик (Local Procedure Call, LPC).
6. Диспетчер процесів (Process Manager).
7. Менеджер об'єктів (Object Manager).

Розглянемо дані компоненти більш детально.

Інтерфейс системних служб забезпечує точки переходу з призначеного для користувача режиму в код режиму ядра, що дозволяє застосуванням користувача (потокам цих застосувань) безпечно здійснювати виклики системних сервісів (процедур режиму ядра). Залежно від платформи, перехід з призначеного для користувача режиму до коду режиму ядра може бути і простою процесорною інструкцією, і достатньо складним перемикачем контексту Save або Restore.

Менеджер конфігурації конструює модель всієї доступної апаратури і всього інстальованого програмного забезпечення, яке є на комп'ютері. Для зберігання образу цієї моделі використовується база даних, яку часто називають Системним Реєстром. Драйвери пристроїв використовують інформацію Реєстру для уточнення великої кількості характеристик оточення, в якому їм доведеться працювати. З введенням специфікації PNP роль Системного Реєстру для драйверів, реалізованих по WDM моделі, істотно знизилася.

Диспетчер (менеджер) введення/виведення є виконавчим компонентом, який реалізований у вигляді множини процедур рівня ядра. ДВВ представляє для процесів призначеного для користувача режиму одноманітний підхід до операцій введення/виведення. Пропонована модель не робить відмінностей, чи отримує призначений для користувача процес доступ до клавіатури, комунікаційного порту або файлу на диску – спосіб звернення оформлений однаково.

ДВВ представляє запити від процесів призначеного для користувача режиму драйверним процедурам у формі пакету запиту на введення/виведення, тобто пакету IRP. Пакет IRP є свого роду робочим рецептом, створеним ДВВ, який передається в драйверні процедури. Робота ж драйвера полягає в тому, щоб належним чином цей запит обробити.

Фактично, ДВВ є інтерфейсом між кодом режиму користувача і драйверами пристроїв. Таким чином, він є першим і найважливішим системним компонентом, з яким взаємодіє драйвер. За посередництва ДВВ з драйвером можуть спілкуватися і компоненти рівня ядра, наприклад, можуть звертатися й інші драйвери.

Менеджер віртуальної пам'яті. У ОС Windows 2000/XP/2003, як і в Windows 9x, адресний простір процесу є безперервним і таким, що безперервно адресується (flat). Розмір простору, що адресується таким чином, в 32-розрядній версії Windows складає 4 Гбайти, що відповідає використанню 32-розрядні покажчики. При цьому тільки нижні 2 Гбайти доступні для використання кодом в режимі користувача. Програмний код і дані призначених для користувача програм повинні розміщуватися в цій нижній половині адресного простору. У випадку, якщо призначені для користувача програми використовують спільно бібліотеки (DLL), що динамічно підключаються, то цей бібліотечний код також повинен розміщуватися в цих двох гігабайтах адресного простору.

Верхні 2 Гбайти адресного простору кожного процесу містять код і дані, доступ до яких можливий тільки з програмного коду, що виконується на рівні ядра. Верхні 2 Гбайти використовуються кодом рівня ядра спільно від процесу до процесу. Якщо виконати друк адрес будь-яких об'єктів, змінних або процедур у вікні DebugView або DebugPrint, то код драйвера буде відображатися на адресному просторі вище 2 Гбайти (адреси перевищують 0x80000000).

Менеджер віртуальної пам'яті здійснює управління пам'яттю від імені всієї ОС. Для звичайної програми режим користувача означає виділення пам'яті

і управління адресним простором та фізичною пам'яттю нижче за межу 2 Гбайт. У тому випадку, коли процесу режиму користувача не вистачає фізичної пам'яті, VMM створює ілюзію наявності пам'яті шляхом віртуалізації запиту. Необхідна пам'ять виділяється сторінками, тобто блоками відповідного розміру (для Intel платформи – 4 Кбайт) на жорсткому диску (що називається – paging). У міру необхідності доступу до неї з боку потоків процесу виділені сторінки переміщаються у фізичну пам'ять. Таким чином, фізична пам'ять стає ресурсом, що спільно використовується усіма процесами.

Менеджер віртуальної пам'яті виступає також і в ролі відповідального за розподіл пам'яті в тому сенсі, що управляє "купою" (heap area) для програмного коду рівня ядра. Для своїх потреб драйвери через відповідні виклики (наприклад, виклик ExAllocatePool, що оброблюється VMM) можуть запитувати виділення областей пам'яті в сторінковій або несторінковій пам'яті.

Засоби локальних процедурних викликів є механізмом викликів між процесами на одному комп'ютері. Оскільки міжпроцесний виклик (interprocess call) може проходити між різними адресними просторами, то існують і відповідний виконавчий компонент, який робить цю дію можливою і ефективною. Як правило, драйверному коду не потрібно здійснення виклику іншого процесу і, відповідно, LPC засобу.

Менеджер процесів надає функції, що починаються з префікса Ps (наприклад, PsGetVersion – отримати версію ОС). Функції PsXxx можуть виконуватися на різних рівнях IQRL, що слід уточнювати в документації DDK. Робота менеджера процесів полягає в наступному: процес є середовищем, в якому існує (виконується) потік. Кожен процес визначає власний адресний простір і містить елементи ідентифікації для визначення прав доступу (security identity). Важливо відзначити, що в Windows процес не виконується; виконується потік, який є одиницею виконання, в той час, як процес є "фігурою" власності. Процес володіє одним або декількома потоками.

Менеджер процесів Windows 2000/XP/2003 є виконавчим компонентом, який управляє створенням процесу і надає йому оточення, в якому працюють програмні потоки. Менеджер процесів в своїй роботі спирається на інші виконавчі компоненти (наприклад, менеджера об'єктів і менеджера віртуальної пам'яті), представляючи собою верхній рівень абстрагування над іншими системними сервісами нижчого рівня.

Драйвери рідко контактують безпосередньо з менеджером процесів. Як правило, вони спираються на інші системні сервіси для доступу до середовища процесу (process environment). Наприклад, драйвер хоче бути упевнений, що буферна область пам'яті в області адрес, що знаходяться у володінні процесу, утримується протягом всього часу передачі даних. Системні процедури з префіксом Mm, що відносяться до менеджера пам'яті, надають драйверу засоби для такого утримання.

Менеджер (диспетчер) об'єктів відповідає за централізоване створення та знищення всіх системних об'єктів (файли, процеси, потоки, події, секції пам'яті і навіть підрозділи Системного Реєстру). Це дозволяє отримати уніфікацію

(одноманітність) доступу до об'єктів, контролю над їх часом життя і забезпечувати безпеку і права доступу до них.

В табл. 1.1 подаються основні префікси виконавчих компонентів.

Таблиця 1.1

Префікси виконавчих компонентів

Префікс	Характеристика
Cc	Диспетчер кеша
Dbg	Підтримка відладки
Ex	Підтримка виконуючої підсистеми Executive
FsRtl	Бібліотека часу виконання для підтримки файлової системи (File System Run-Time Library)
Hal	Диспетчер рівня апаратних абстракцій, HAL
Inbv	Драйвер ініціалізації системи/завантаження VGA
Init	Ініціалізація системи
Interlocked	Потокобезпечна операція змінними
Io	Диспетчер введення/виведення (I/O Manager)
Kd	Підтримка Kernel Debugger
Ke	Підпрограми ядра (kernel)
Ki	Обробка ядра
Ldr	Завантажувач образу
Lpc	Локальний виклик процедур (Local Procedure Call)
Lsa	Local Security Authority
Mm	Менеджер пам'яті (Менеджер Віртуальної пам'яті, VMM)
Nls	Лінгвістична підтримка (National Language Support)
Nt	NT Native API
Ob	Менеджер об'єктів (Object Manager)
Pfx	Обробка префіксів
Po	Менеджер електроживлення
Ps	Підтримка процесів (Process Structure)
Rtl	Бібліотека часу виконання (Run-Time Library)
Se	Управління безпекою і забезпечення привілеїв
Zw	Альтернативний інтерфейс Native API
інші	Допоміжні функції і бібліотека часу виконання

Примітка. *У першому стовпчику вказані скорочення, що зазвичай є першими символами імен функцій, які даний компонент надає розробникові для використання при програмуванні в режимі ядра. Наприклад, функція RtlCopyMemory, що є аналогом відомої функції режиму користувача memcpy, надається бібліотекою часу виконання Rtl (Run Time Library).*

1.2.5. Підсистеми середовища операційної системи Windows NT 5

Як видно з рис. 1.2, Windows NT має три підсистеми середовища (Win32, Posix і OS/2), які працюють тільки на платформі x86. Підсистема Win32 специфічна для Windows NT і не може працювати поза нею. Розглянемо більш детально кожну із цих підсистем середовища.

Кожна з підсистем забезпечує додаткам користувача доступ до різних піднаборів служб Windows NT. Це означає, що деякі речі можуть бути зроблені з додатку, побудованого на одній підсистемі, і не можливі з додатку, побудованого в іншій підсистемі. Так, додаток для Win32 не може використовувати функцію fork підсистеми Posix.

Кожен модуль, що виконується, зв'язується з однією і лише однією підсистемою. Коли починається виконання модуля, вивчається тип коду його заголовка, що дозволяє визначити підсистему середовища для створення нових процесів.

Процеси користувача безпосередньо не викликають служби NT, а використовують бібліотеки динамічних зв'язків (DLL) відповідної підсистеми середовища. Роль бібліотек, що належать підсистемі середовища, в тому, щоб транслювати документовані функції середовища у відповідні виклики недокументованих служб NT. Ці бібліотеки DLL експортують документований інтерфейс, який можуть викликати пов'язані з підсистемою програми. Наприклад, бібліотеки DLL підсистеми Win32 використовує функції Win32 API. Бібліотека DLL підсистеми Posix використовує функції Posix 1003.1 API.

Підсистема Win32. Головними компонентами підсистеми Win32 є процес підсистеми середовища і драйвер режиму ядра.

Процес підсистеми середовища підтримує:

- консольні (текстові) вікна;
- створення і видалення процесів і потоків;
- роботу віртуальної 16-розрядної машини DOS;
- інші функції (GetTempFile, DefineDosDevice, ExitWindowsEx тощо).

Драйвер режиму ядра підтримує:

- менеджер вікон, який управляє відображенням вікон, виведенням на екран, введенням з клавіатури, від миші і інших пристроїв, а також передачею призначених для користувача повідомлень додаткам;
- інтерфейс графічних пристроїв (Graphical Device Interface, GDI) – бібліотека функцій для виведення на графічні пристрої, для малювання тексту, ліній, фігур і маніпуляцій графічними об'єктами;
- залежні від пристроїв драйвери графіки, принтера і відеопорту;
- декілька бібліотек DLL, які транслюють документовані функції Win32 API у відповідні недокументовані виклики NTOSKRNL.EXE і WIN32K.SYS.

Додатки викликають стандартні функції для створення вікон і кнопок на дисплеї. Менеджер вікон передає ці запити драйверам графічних пристроїв через інтерфейс графічних пристроїв GDI, де вони форматуються для виведення засобами конкретних пристроїв. GDI забезпечує набір стандартних

функцій, що дозволяють додаткам спілкуватися з графічними пристроями, включаючи дисплеї і принтери, без конкретних знань про них. GDI інтерпретує запити додатків на графічний виведення і посилає їх драйверам графічних дисплеїв. Цей інтерфейс дозволяє створювати код додатку, незалежний від конкретних пристроїв і їх драйверів.

NTDLL.DLL – це спеціальна система підтримки DLL-бібліотек. Вона містить два типи функцій.

Перша група функцій забезпечує інтерфейс до служб NT, які можуть бути викликані з режиму користувача. Існує більше 200 таких функцій, наприклад, NtCreateFile, NtSetEvent і таке інше. Для кожної з них є точка входу в NTDLL.DLL з тим же ім'ям. Внутрішній код функції містить специфічні для архітектури команди, які викликають перехід в режим ядра для звернення до реальних служб NT, код яких міститься в NTOSKRNL.EXE.

Друга група функцій містить велику кількість функцій підтримки: завантажувач виконуваних модулів, комунікаційні функції для процесів підсистеми Win32, бібліотека функцій реального часу та диспетчер викликів асинхронних процедур APC в режимі користувача, диспетчер виключень.

Виконавчі компоненти Windows 2000/XP/2003 визначають і представляють основні сервіси ОС. Проте ці сервіси ніколи не надаються програмам призначеного для користувача режиму безпосередньо. Замість цього розробники з Microsoft визначили декілька інтерфейсів прикладного програмування (Application Programming Interfaces, API). Він дозволяє звертатися прикладним програмам до системних сервісів через їх спеціальні абстракції. Ці інтерфейси формують різні середовища (environmental subsystems), в яких і мешкають прикладні програми. В даний час в Windows NT є присутніми п'ять підсистем:

1. *Підсистема Win32*, що є власним (native-mode) API для 32-розрядних версій Windows. Вся решта середовищ використовує цю підсистему для виконання своєї роботи. Всі нові додатки 32-розрядних Windows 2000/XP/2003, а також і всі перенесені, вважаються на Win32 як у середовищі свого функціонування. Проте, в 64-розрядній версії Windows (версії XP/Server 2003) вона сама стає клієнтом WOW 64.

2. *Віртуальна машина DOS* (Virtual DOS Machine, VDM) підсистема забезпечує 16-розрядне операційне середовище MS DOS для старих додатків DOS. Не дивлячись на запевнення в сумісності, безліч існуючих програм DOS в цьому середовищі не працюють належним чином. Відбувається це з тієї причини, що Microsoft, проповідуючи консервативний підхід, надає емуляцію апаратури замість можливості безпосереднього звернення до неї. У результаті, прямий доступ до апаратури приводить до обмеження з боку ОС і, часто, додатки DOS відмовляють працювати.

3. *Підсистема "Windows on Windows"* (WOW) підтримує операційне середовище для можливості роботи старих 16-бітових додатків Windows (наприклад, Windows 3.x). У 64-розрядних клонах Windows XP/Server 2003

підсистема WOW 64 служить для запуску створених раніше 32-розрядних додатків, що перенесені на нові апаратні платформи.

4. *Підсистема POSIX* забезпечує виконання Unix-додатків, які задовольняють стандарту POSIX 1003.1. На жаль, більшість перенесених додатків Unix-подібних систем не працюють належним чином в цій підсистемі. В даному випадку, більшість Unix-додатків переносяться під Windows шляхом переписування під підсистему Win32 або вони спочатку створюються з використанням спеціальних програмних пакетів типу MainWin, Motif і OpenMotif.

5. *Підсистема OS/2* створює середовище виконання для 16-розрядних програм операційної системи OS/2 – принаймні, тих з них, які не використовують в своїй роботі сервісів такого компоненту OS/2, як Presentation Manager. На цю підсистему можна розраховувати тільки у версії Windows для платформи Intel (x86).

Кожний додаток однозначно пов’язаний з одним середовищем виконання. Додатки не можуть здійснювати API виклики до інших виконавчих середовищ. Крім того, підсистема Win32 є основною в 32-розрядних версіях Windows NT 5.x. Інші підсистеми емулюють відповідні властивості середовищ, що реалізовуються, через засоби і методи Win32. Відповідно, параметри виконання програм в цих середовищах деградують і істотно поступаються аналогічним програмам для Win32, тому дану систему розглянемо більш детально.

Оскільки підсистема Win32 має особливий статус серед решти підсистем, а внаслідок цього до неї пред’являються підвищені вимоги, то і реалізація цієї підсистеми істотно відрізняється. Зокрема, підсистема Win32 розділена на декілька компонентів, частина з яких працює в режимі користувача, а інша у режимі ядра. Функції Win32 можна розділити на три категорії, а саме:

1. Функції, які надаються користувачеві для управління вікнами, меню, діалогами і елементами контролю (кнопками, смугами прокрутки, перемикачами, закладками тощо).

2. Функції Graphical User Interface (GUI), що реалізовують промальовування зображень на фізичних пристроях, екрані, принтері, графічному пристрої.

3. Функції, які управляють неграфічними ресурсами, такими як процеси, програмні потоки, файли і об’єкти синхронізації. Kernel-функції тісно пов’язані з системними службами виконавчих компонентів.

З часів NT 4.0 велика частина функцій перших двох категорій з приведеної вище класифікації була реалізована в режимі ядра. Призначені для користувача процеси, які запрошують послуги GUI, звертаються безпосередньо до коду режиму ядра при використанні інтерфейсу системних служб). Код, який представляє ці функції і ядра працює в режимі локалізованому в модулі WIN32K.SYS.

Функції третьої категорії при обробці запитів від призначених для користувача процесів спираються на стандартний серверний процес CSRSS.exe

(Client-Server Runtime Subsystem), який і звертається власне до коду виконавчих компонентів для завершення обробки цих звернень.

До підсистем, що реалізують середовище виконання кодів DOS, Windows, POSIX і OS/2, можна віднести ще декілька ключових системних компонентів, які реалізовані як процеси режиму користувача. Серед них основними є:

1. *Підсистема безпеки* (Security Subsystem), яка управляє локальною і видаленою безпекою (захистом від неправомірного доступу) з використанням ряду процесів і динамічних бібліотек. Частина роботи Active Directory протікає якраз серед цієї логічної підсистеми.

2. *Менеджер управління сервісами* (Service Control Manager, SCM), управляє процесами-демонами (сервісами), і драйверами пристроїв.

3. *Процеси підтримки RPC викликів*, які подають підтримку додаткам, що поширюються по мережі. Удаючись до використання викликів видалених процедур, додатки можуть виконувати свою роботу з використанням безлічі мережевих комп'ютерів.

Підсистема введення/виведення вносить корегування в список завдань нових систем Windows, основними з яких є:

1. *Конфігурація*, як з точки зору апаратури, так і програмного забезпечення. Для драйверів Windows це означає повну підтримку специфікації для шин і пристроїв (Plug and Play**,** PNP).

2. *Пріоритетність і переривчастість*. Код, написаний для обслуговування введення/виведення, ніколи не повинен блокуватися і повинен містити безпечні програмні потоки.

3. *Безпека при використанні на багатопроцесорних платформах*. Один і той же драйверний код повинен безпомилково працювати і на однопроцесорних і на багатопроцесорних комп'ютерах.

4. *Об'єктна орієнтованість*. Послуги, що надаються кодом, повинні "формулюватися" в термінах цілком певних структур даних, які служать виконанню дозволених операцій.

5. *Пакетне управління*. Запити, зроблені до підсистеми введення/виведення формулюються, передаються і відстежуються за допомогою чіткого формату, відомого як IRP-пакет (I/O Request Packet, пакет запиту на введення/виведення).

6. *Підтримка асинхронного введення/виведення*. Підсистема введення/введення повинна дозволяти коду програми, по зверненню якої створений запит IRP, виконуватися паралельно тому програмному коду, який здійснює обробку запиту. Також повинен існувати механізм сповіщення ініціатора виклику про повне завершення обробки запиту.

Крім вже вказаних завдань, існує необхідність забезпечення повторного використання коду, що називається реєнтерабельністю. Це означає необхідність не тільки складної структуризації власне коду, що відповідає за операції введення/виведення в одному пристрої, але й іншого погляду на взаємодію шарів всієї підсистеми введення/виведення. Наприклад, код, що відповідає за управління шиною, повинен бути реалізований автономно від коду, що

відповідає за конкретний пристрій, підключений до шини. Виконання цієї вимоги дозволяє повторно використовувати код шинного драйвера програмними блоками (наприклад, драйверами), що відповідають за інші пристрої на цій шині, які можуть підключатися і відключатися.

1.3. Різниця між версіями операційної системи Windows NT 5.x

Відмінності версій Windows 2000 (NT 5.0), XP (NT 5.1) і Server 2003 (NT 5.2), зрозуміло, існують. І стосуються вони не тільки призначеного для користувача інтерфейсу і наповнення сервісними програмами, але і наборів системних викликів, що надаються в режимі ядра.

Коротко відзначимо найбільш важливі для розробника драйверів відмінності. Перш за все, слід розрізняти 32 і 64-розрядні клони. Якщо 64-розрядна Windows 2000 була зібрана лише один раз в тестовому режимі, то Windows XP (і вже тим більше, Server 2003) в 64-розрядній збірці є реальним комерційним продуктом. Організація адресного простору пам'яті в 64-розрядній версії сильно відрізняється від організації віртуального простору в 32-розрядній версії. Додатки, створені як 32-розрядні програми, запускаються під управлінням підсистеми Win32 в рамках моделі WOW, аналогічно тому, як 16-розрядні додатки працюють в 32-розрядному середовищі.

Драйвер в 64-розрядній версії компілюється як 64-розрядний код – зокрема, з відповідною довжиною покажчиків. У даній ситуації драйверу актуально знати, від якого додатку він отримав запит – від нового 64-розрядного або старого 32-розрядного. Ця проблема не представляє великої складності, оскільки 64-розрядний драйвер може з'ясувати тип процесу за допомогою системної функції IoIs32bitProcess, доступної для додатку в 64-розрядних драйверах.

Наприклад:

```
if( IoIs32bitProcess(Irp)== TRUE )
{
    #if DBG
        DbgPrint("IRP request is made by 32 bit process.");
    #endif
}
else
{
    #if DBG
        DbgPrint("IRP request is made NOT by 32 bit process.");
    #endif
}
```

Відповідно до відповіді драйвер може скорегувати деякі свої операції.

З якою конкретно версією ОС має справу драйвер, можна легко визначити по тому, яку версію моделі WDM підтримує система. Це можна дізнатися за допомогою системного виклику IoIsWdmVersionAvailable**,** наприклад:

```
if (IoIsWdmVersionAvailable(1, 0x30)) {
    // Windows Server 2003
} else if (IoIsWdmVersionAvailable(1, 0x20)) {
    // Windows XP
} else if (IoIsWdmVersionAvailable(1, 0x10)) {
    // Windows 2000
} else if (IoIsWdmVersionAvailable(1, 0x05)) {
    // Windows Me
} else {
    //    Windows 98
}
```

Оскільки найбільш істотні зміни відбулися в 32-розрядних версіях, то розглянемо їх більш детально:

1. При переході від Windows 2000 до Windows XP був переписаний завантажувач NTLDR, внаслідок чого процес завантаження прискорився в 4-5 разів.

2. Модуль `NTOSKRNL.EXE` Windows 2000 експортував 1129 функцій (більшість з яких – системні виклики, доступні з драйверів режиму ядра), Windows XP експортує 1464, а Windows Server 2003 – вже 1525.

3. Модуль HAL.DLL експортує, 95, 92 і 92 виклики, відповідно. Причому, по непрямих ознаках помітно, що від версії 2000 до XP його спіткала значна переробка, а від версії XP до Server 2003 – лише процес "шліфування".

4. Поповнення системних викликів в Server 2003 відбулося, в основному, за рахунок викликів з суфіксом Ex, тобто функцій, що розширюють існуючі системні виклики Windows XP. Наприклад, IoCsqInitializeEx вийшов з виклику IoCsqInitialize.

5. При переході від Windows 2000 до Windows XP було пом'якшено багато обмежень на граничні розміри.

6. ОС Windows 2000 обмежувала загальний розмір адресного простору під драйверами 220 Мегабайтами, в XP ця межа починається з 960. Таким же він залишився і в 32-розрядній версії Server 2003.

7. Граничний загальний розмір файлів Системного Реєстру Windows XP жорстко не фіксований, тоді як в Windows 2000 він не повинен був перевищувати 376 Мбайт. Збільшений розмір системних сторінкових таблиць, внаслідок чого системний віртуальний адресний простір збільшився до 1.3 Гбайт – проти 660 Мбайт в Windows 2000. При цьому 960 Мбайт в системному віртуальному адресному просторі XP безперервні, в інших же його "місцях" є розриви (64-розрядна версія підтримує розмір віртуального системного адресного простору, що дорівнює 128 Гбайт.) Мабуть, що такі ж параметри залишилися і у Windows Server 2003 (точні відомості відсутні).

8. Оскільки, сторінкові таблиці і велика частина Системного Реєстру розміщуються резидентно у фізичній пам'яті, зрозуміло, що тоді ціна такого "послаблення" буде нове підвищення вимог до мінімального розміру встановленої на комп'ютері оперативної пам'яті – до 128 Мбайт.

9. Збільшено число процесорів, які може підтримувати ОС в симетричній багатопроцесорній конфігурації (SMP). У редакції Datacenter Windows Server 2003 може підтримувати до 64 процесорів (у 64-розрядній версії). 32-розрядна версія (як і Datacenter Windows 2000) підтримує як і раніше до 32 процесорів. Зрозуміло, така архітектура зобов'язана бути належно протестована, і Microsoft анонсувала особливий підхід до продаж таких "важких" серверів: ОС поставлятиметься тільки разом з апаратурою ЕОМ постачальниками.

10.Відповідь на питання, що ж означають величезні цифри підтримуваної оперативної пам'яті в 32-розрядних версіях: від 2 Гбайт (редакція Web Windows Server 2003) до 64 Гбайт (редакція Datacenter Windows Server 2003) криється в двох абревіатурах: AWE (Address Windowing Extension) і PAE (Physical Address Extension). ОС, підкоряючись параметру PAE, що заданий у файлі boot.ini, завантажується в модифікованій конфігурації, яка підтримує режим роботи PAE. У такому режимі можлива маніпуляція фізичними адресами оперативної пам'яті (тип PHYSICAL_ADDRESS) за межами 4 Гбайтного простору функціями категорії MmAllocatePagesForMdl. Найбільш простим додатком даного розширення, що знаходиться "на поверхні", є створення драйверів, які реалізуютьі RAM диск. Правда, ускладнює справу висока ціна устаткування. На сьогодні частка материнських плат, що підтримують розмір оперативної пам'яті більше 8 Гбайт, не перевищує 1,5% ринку.

Отже, як і раніше всі розглянуті версії продовжують підтримувати всі три файлові системи: FAT, FAT32, NTFS. Іншими словами, Windows Server 2003 Datacenter Edition цілком встановлюється на логічному диску FAT32, займаючи при цьому трохи більше 1,3 Гбайт. До того ж, Microsoft остаточно (у Windows Server 2003) відійшла від підтримки підсистем POSIX і OS/2.

Проте, залишилося незмінним найважливіше – драйверна модель, закладена в Windows 2000, забезпечує практично повну сумісність програмного коду (а іноді – і бінарного) у всіх даних версіях. Тому для подальшого розуміння необхідно розглянути особливості апаратного забезпечення персонального комп'ютера.

Висновки

- Основними цілями створення операційної системи Windows NT є сумісність, переносимість, розширюваність та продуктивність. Її основними головними особливостями вважається: модель змінюваного мікроядра, наявність різних підсистем, багатопоточність та інтегрована підтримка мережі.
- Архітектура Windows NT складається із двох основних рівнів (режимів) – режиму користувача та режиму ядра.
- До виконавчих компонентів ОС Windows NT відносять: інтерфейс системних служб, менеджер конфігурації, диспетчер введення/виведення,

менеджер віртуальної пам'яті, локальний процедурний виклик, диспетчер процесів, менеджер об'єктів.

- ✦ Відмінності між версіями Windows полягають в призначеному для користувача інтерфейсі, в наповненні сервісними програмами та в різниці наборів системних викликів, що надаються в режимі ядра.

Контрольні запитання та завдання

1. Які основні цілі створення ОС Windows NT залишаються актуальними і сьогодні?
2. Поясніть, чому для досягнення стійкості в роботі системи ОС Windows NT для побудови ядра було обрано архітектуру "клієнт-сервер".
3. Охарактеризуйте основні виконавчі компоненти ОС Windows NT.
4. За що відповідає підсистема Win32?
5. Назвіть основні функції підсистеми Win32.
6. Яка різниця між AWE та PAE. Без якого із цих параметрів операційна система Windows NT буде функціонувати без збоїв, поясніть свій вибір.
7. Назвіть та охарактеризуйте основні драйвери пристроїв ОС Windows NT.

Тести для самоконтролю

1. Як називають властивість ОС Windows NT, яка повинна підтримувати максимально можливу множину програмного та апаратного забезпечення?
   a. сумісність
   b. переносність
   c. розширюваність
   d. продуктивність
2. В якому режимі виконується код самої ОС Windows NT?
   a. code mode
   b. kernel mode
   c. user mode
3. Що означає шар ОС Windows NT – HAL?
   a. виконавчі компоненти
   b. диспетчер введення/виведення
   c. рівень апаратних абстракцій
   d. служби, які працюють у режимі ядра
4. Що означає параметр Executive?
   a. потік
   b. диспетчер введення/виведення
   c. виконавчий компонент
   d. планувальник

5. Як називається виконавчий компонент ОС Windows NT, що забезпечує точки переходу з режиму користувача в код режиму ядра?
    a. Configuration Manager
    b. I/O Manager
    c. Local Procedure Call
    d. System Service Interface
6. Що в ОС Windows NT є інтерфейсом між кодом режиму користувача і драйверами пристроїв?
    a. диспетчер введення/виведення
    b. менеджер віртуальної пам’яті
    c. менеджер конфігурації
    d. локальний процедурний виклик
7. Яка підсистема ОС Windows NT забезпечує виконання UNIX-додатків?
    a. Win32
    b. VDM
    c. WOW
    d. POSIX
    e. OS/2
8. Який системний виклик використовує ОС Windows NT для визначення версії ОС?
    a. IoIs32bitProcess
    b. IoIsWdmVersionAvailable
    c. IoCsqInitializeEx
    d. MmAllocatePagesForMdl
9. Як називають завантажувальні модулі, які працюють в режимі ядра, забезпечуючи інтерфейс між системою введення/виведення і відповідним устаткуванням?
    a. драйвери пристроїв
    b. абстракція від устаткування
    c. об’єкти ядра

Теми для самостійного опрацювання

1. Особливості операційних систем Windows XP та Windows NT.
2. Функціональні компоненти ОС Windows NT.
3. Виконавча система (The Executive) ОС Windows NT.
4. Реєстр ОС Windows NT.
5. Порівняльна характеристика підсистем Win32 та Win64.
6. Архітектура UNIX та Windows.
7. Загальна характеристика WOW64.

Список використаної літератури

1. *David A. Solomon* The Windows NT Kernel Architecture / *David A. Solomon*. – IEEE Computer, October 1998. – Pp. 40-47.
2. *Кокорева О.* Реестр Windows XP / О. Кокорева. - СПб.: БХВ-Петербург, 2003. – 560 с.
3. *Комиссарова В.* Программирование драйверов для Windows / В. Комиссарова. – СПб.: БХВ-Петербург, 2007. – 256 с.
4. *Немнюгин С. А.* Эффективная работа: UNIX / С. А. Немнюгин, А. В. Комолкин, М. П. Чаунин. – СПб.: Питер, 2001. – 688 с.
5. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
6. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.
7. *Руссинович М.* Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс / М. Руссинович, Д. Соломон; пер. с англ. – [4-е изд.]. – М.: Русская редакция, СПб.: Питер, 2005. – 992 с.
8. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО “Бином-Пресс”, 2004. – 480 с.
9. *Сорокина С. И.* Программирование драйверов и систем безопасности : учебное пособие / С. И. Сорокина, А. Ю. Тихонов, А. Ю. Щербаков. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2002. – 256 с.
10. *Шеховцов В. А.* Операційні системи / В. А. Шеховцов. – К.: Видавнича група BHV, 2008. – 576 с.
11. *Шрайбер С.* Недокументированный возможности Windows 2000 : Библиотека программиста / Свен Шрайбер. – СПб.: Питер, 2002. – 544 с.

РОЗДІЛ 2. АПАРАТНЕ ЗАБЕЗПЕЧЕННЯ ПЕРСОНАЛЬНОГО КОМП'ЮТЕРА

План

- ✦ Загальні відомості про апаратне забезпечення
- ✦ Сигнали, пріоритети та вектори переривань
- ✦ Механізми передачі даних
- ✦ Шини в комп'ютерних системах
- ✦ Корисні поради щодо роботи з апаратурою

В даному розділі розглядаються основні поняття та терміни апаратного забезпечення персонального комп'ютера, сигнали, пріоритети та вектори переривань, механізми передачі даних та наводиться характеристика шин в комп'ютерних системах.

2.1. Загальні відомості про апаратне забезпечення

Не дивлячись на велике різноманіття типів і областей застосування пристроїв, потреба в підключенні яких до комп'ютера виникає на практиці, можна виділити декілька загальних рис, про які обов'язково необхідно знати розробникові драйвера:

1. Чи підтримує пристрій специфікацію PNP і як пристрій може сповістити систему про своє існування при підключенні?
2. Як відбувається доступ до регістрів стану і контролю пристрою?
3. Як пристрій може генерувати сигнал переривання?
4. Як пристрій бере участь в передачі даних?
5. Чи використовує пристрій будь-яку вбудовану пам'ять?
6. Як конфігурується пристрій? Як це можна зробити програмно?

Розглянемо більш детально деякі із цих понять.

Автоматичне розпізнавання і конфігурація

За кожним апаратним пристроєм, підключеним до комп'ютера, закріплюються системні ресурси, які можуть включати:

- діапазон адрес введення/виведення (портів введення/виведення);
- діапазон відведених адрес пам'яті;
- номер переривання IRQ;
- DMA канал.

Оскільки різні пристрої були виготовлені в різний час різними постачальниками, то конфлікти ресурсів абсолютно неминучі. Перші

персональні комп'ютери вимагали чималої кмітливості від користувача при конфігурації пристроїв, що підключалися, правильного, а часом – єдино вірного, виставляння перемичок або DIP перемикачів, що визначають унікальну настройку ресурсів для даної системи. Помилки в такій ручній конфігурації були частими, а результатом були або неможливість завантажити систему, або непередбачувані зависання системи і непрацюючі пристрої.

Для подолання цих проблем були введені нові принципи побудови шинної архітектури, яка б підтримувала автоматичне розпізнавання і конфігурації пристроїв. Автоматичне розпізнавання повинне відбуватися у момент завантаження/перезавантаження системи або безпосередньо у момент підключення пристрою до комп'ютера ("гаряче" підключення). Можливості ОС, відповідно, повинні бути такі, щоб відповідне супровідне програмне забезпечення вступало в роботу без загального перезавантаження системи.

Автоматична конфігурація дозволяє програмному забезпеченню встановлювати прийнятні значення ресурсів для пристроїв, що допускають програмну настройку. Ці удосконалення дозволяють відійти від використання перемичок на пристроях, що підключаються, а кінцевому користувачеві більше немає потреби знати про всі особливості конфігурації системи і нового пристрою в ній.

Регістри пристроїв

Драйвери взаємодіють з пристроями, що підключаються, шляхом читання з регістрів або запису в їх внутрішні регістри. Кожен внутрішній регістр пристрою зазвичай реалізує одну з функцій, перерахованих нижче:

1. *Регістр стану*. Зазвичай читається драйвером, коли тому необхідно отримати інформацію про поточний стан пристрою.

2. *Регістр команд*. Біти цього регістра управляють пристроєм деяким чином, наприклад, починаючи або припиняючи передачу даних. Драйвер зазвичай проводить запис в такі регістри.

3. *Регістри даних*. Зазвичай, вони використовуються для передачі даних між пристроєм і драйвером. У вихідні (output) регістри, регістри виведення, драйвер проводить запис, тоді як інформація вхідних (input) регістрів та регістрів введення читається драйвером.

Доступ до регістрів пристрою досягається в результаті виконання інструкцій доступу до портів введення/виведення (port address) або звернення до певних адрес в адресному просторі оперативної пам'яті (memory-mapped address), що і інтерпретуються системою як доступ до апаратних регістрів.

Прості пристрої мають невелике число асоційованих регістрів. В той же час, складне апаратне забезпечення (наприклад, графічні адаптери) може мати значно більше регістрів. Число і призначення регістрів визначається розробниками апаратного забезпечення і повинно бути повно і однозначно описане в документації. Проте часто така однозначність так і залишається недосяжною мрією, а розробникові драйвера доводиться визначати реальне призначення потрібних бітів в пристроях, використовуючи випадково здобутий

тестовий програмний приклад невідомого автора або метод власних проб і власних помилок. Більш того, часто з'ясовується, що біти оголошені в документації як "зарезервовані" (reserved) можуть бути шкідливими.

Доступ до регістрів пристроїв

Коли функціонування пристрою стало майже зрозумілим, залишиться невелика проблема: як програмно дістати доступ до регістрів пристрою.

Як правило, регістри слідують один за одним в своєму адресному просторі. Отже, спершу, необхідна адреса першого з них. На жаль, значення терміну "адреса" сильно варіюється при використанні його щодо віртуальних адресних просторів на різних платформах.

Варіант перший. Використовуємо специфічну для даного процесора інструкцію введення/виведення в порт, а значить, потрібна адреса порту введення/виведення.

Варіант другий. Особливі області в пам'яті суміщені з адресами доступу до пристрою. Запис за адресою в пам'яті означає перенесення даних в пристрій. Найпоширеніший прийом для доступу до відеопам'яті – доступ по спеціальних адресах з використанням стандартних звернень до пам'яті.

Якщо говорити в термінах асемблера, то інструкції для доступу до простору пам'яті – це MOV, а для доступу до простору введення/виведення використовуються інструкції типу IN або OUT.

Простір введення/виведення

У деяких реалізаціях процесорної архітектури доступ до регістрів пристроїв здійснюється за допомогою спеціальних команд процесора – інструкцій введення/виведення. Вони посилаються на спеціальні набори виводів процесора і визначають окремий шинно-адресний простір для пристроїв введення/виведення. Адреси на цих шинах широко відомі як порти (ports) і не мають ніякого відношення до адресації пам'яті. У архітектурі Intel x86 адресний простір введення/виведення має розмір 64 КБ (16 розрядів), а в мові асемблера визначено дві інструкції для читання і запису в цьому просторі: IN і OUT (точніше, дві групи інструкцій, усередині яких відмінність має місце по розрядності зчитуваних/записуваних даних).

Оскільки при створенні драйвера слід уникати прив'язки до апаратної платформи, Microsoft рекомендує уникати і використання реальних інструкцій IN/OUT. Замість цього слід використовувати макровизначення HAL. Відповідність між традиційними інструкціям DOS/Windows асемблера і макровизначеннями HAL, наприклад, такі:

1. Читання одного байта з порту введення/виведення:

– асемблер x86:
```
IN AL,DX
IN AL,port
```
– аналог HAL:
```
READ_PORT_UCHAR
```

2. Запис 32-розрядного слова в порт введення/виведення:

– асемблер x86:
```
OUT DX,EAX
```
– аналог HAL:
```
WRITE_PORT_ULONG
```

```
OUT port,EAX
```

Доступ через адресацію в пам'яті

Не завжди була доцільним організація "портового" доступу до регістрів пристроїв через адресний простір введення/виведення. У альтернативному підході доступ до регістрів пристроїв здійснювався шляхом звернення до певних адрес в просторі пам'яті (memory address space). В деяких випадках (наприклад, для відеоадаптерів в архітектурі Intel x86) допускаються обидва способи доступу відразу: і через простір введення/виведення, і через адресний простір.

У NTDDK визначені макроси HAL для доступу до таких memory mapped (тобто з доступом за допомогою адресації в пам'яті) регістрам, наприклад READ_REGISTER_UCHAR – читання одного значення з регістра введення/виведення. Оскільки ці макровизначення за змістом відрізняються від HAL макровизначень для операцій з портами введення/виведення, драйверний код повинен бути розроблений так, щоб компіляція для різних платформ (з різними методами доступу до регістрів) проходила коректно. Хорошим прийомом є складання таких макровизначень умовної компіляції, які указували б на один з потрібних макросів HAL залежно від ключів компіляції.

2.2. Сигнали, пріоритети та вектори переривань

2.2.1. Сигнали переривань

Як правило, пристрої функціонують паралельно і асинхронно щодо дій центрального процесора. Тому, для ситуацій, коли їм потрібна увага з боку драйвера, код якого виконується на центральному процесорі, була вироблена тактика використання переривань**,** коли пристрої сигналізують про необхідність обслуговування, тобто генерують переривання. Різні процесори реалізують різні способи. Але є і загальне місце в цих способах: завжди задіяний один (або декілька) з виводів процесора (або допоміжної мікросхеми). За цей вивід пристрій може "посмикати", коли буде потрібно участь процесора.

У обов'язку процесора, якщо він має намір перейти до обслуговування по сигналу, що поступив, входить збереження стану процесора, контексту виконуваного програмного потоку. Лише після цього процесор (а точніше, операційна система) може виконати перехід до програмного коду процедури обслуговування переривання (Interrupt Service Routine, ISR), яку реєструє драйвер, що узяв на себе роботу за уявленням даного пристрою в ОС.

Зазвичай пристрої генерують сигнали переривань в ситуаціях:

1. Пристрій завершив обробку попереднього запиту (що поступив від драйвера) і тепер готовий до обробки нового.

2. Буфер або черга FIFO пристрою майже повні (під час введення) або майже порожні (під час виведення). Таке переривання дозволяє драйверу отримати з буфера пристрою або вивести в буфер пристрою даних для забезпечення його безперервної роботи.

3. Пристрій зіткнувся з нештатною або помилковою ситуацією під час виконання операції, і переривання може бути спеціальне для того призначеною формою завершення операції.

Пристрої, які не здатні генерувати переривання, можуть створити ситуацію серйозної деградації ОС. У зв'язку з тим, що величезну кількість потоків, що виконуються, використовує центральний процесор спільно, неприпустимо дозволяти якому-небудь драйверу таку розкіш, як очікування повного завершення "його" поточної операції. При роботі з подібними пристроями, що не уміють "розмовляти перериваннями", можна застосовувати тактику опиту через певні проміжки часу ("polling").

З ускладненням апаратного забезпечення виникли конфігурації, де шини почали підключатися до інших шин через інтерфейсні елементи, що називаються мостами. В результаті, джерелами переривань (а практично – пристроями) виявляються декілька апаратних шарів, що утворилися навколо процесора, і методи визначення пріоритетів і передачі сигналів процесору зазнали зміни. Проте, залишилося і щось загальне.

2.2.2. Пріоритети переривань

Достатньо часто виникають ситуації, коли пристрої вимагають уваги до себе практично одночасно. Виникає питання: як визначити черговість їх обслуговування? Напевно, найважливіший пристрій, який менше останніх може чекати на обслуговування, повинен мати максимальний пріоритет. Якщо пристрій може бути обслужений із затримкою, то йому привласнюється найнижчий пріоритет переривання. В окремих випадках можна призначити пріоритет пристрою при установці програмного забезпечення для нього.

Для чого потрібний пріоритет переривання? В той час, коли процесор зайнятий виконанням програмного коду у відповідь на переривання від низькопріоритетного пристрою (наприклад, виконує його ISR-процедуру обробки переривань), цілком може поступити сигнал переривання від пристрою з високим пріоритетом. В даному випадку, центральний процесор отримає два переривання – друге над першим – і перейде до обробки того, яке має більший пріоритет, зберігаючи, зрозуміло, контекст відкладеного "убік" потоку. І навпаки, обробка переривання низького рівня, що поступив при обслуговуванні високопріоритетного, затримується до моменту, поки високопріоритетне переривання не буде оброблене повністю і оголошене завершеним.

Важливо відзначити, що в режимі ядра програмний потік будь-якого пріоритету не може бути перерваний задля виконання коду потоку навіть рівного з ним пріоритету – на відміну від режиму користувача, коли навіть самі

низькопріоритетні потоки коли-небудь дістають можливість попрацювати. Цим, зокрема і пояснюються численні і "настирливі" рекомендації в літературі з програмування драйверів не затримуватися в коді процедур обробки переривань (з високим пріоритетом), оскільки ігнорування цього правила може приводити до швидкої і незворотної деградації системи. Ця вимога стає особливо явною, якщо скористатися системним програмним забезпеченням, що дає інформацію про кількість переривань в секунду, – воно міняється від півсотні до тисячі. Зрозуміло, якщо розробник драйверів не дотримуватиметься правила "менше працюй на рівнях DIRQL", то система не зможе нормально працювати, а драйвер ніколи не отримає цифровий підпис Microsoft.

2.2.3. Вектори переривань

Деякі пристрої і деяка процесорна архітектура допускають механізм автоматичної диспетчеризації – переходів "по вектору", тобто адресі програмних процедур, під час вступу сигналів про переривання. При іншому методі векторизація не застосовується, і для всіх типів переривань обслуговування надається узагальненою процедурою. Така процедура проглядає в порядку пріоритетності весь список пристроїв, які могли б викликати переривання, і визначає, який пристрій вимагає обслуговування насправді. Зрозуміло, що в сучасних комп'ютерних системах, де генерується від декількох десятків до декількох тисяч переривань в секунду, істотно ефективнішим виявляється векторний підхід.

2.2.4. Передача сигналів переривань

При генерації переривань використовуються дві основні стратегії. Старіший і менш важливий механізм носить назву edge-triggered (керований перепадом, керований фронтом) або latched-переривання. Пристрої, які генерують такі переривання, сигналізують про потребу в обслуговуванні здійсненням переходу на фізично існуючому провіднику (наприклад, на одному з провідників шини або сполучного кабелю) з одного логічного стану в інший, скажімо, з "1" в "0". Як тільки цей перехідний процес пройшов, пристрій може відпустити лінію переривання, відновивши параметри електричного сигналу, в даному випадку, що позначають логічною "1". Іншими словами, лінія переривання такого пристрою "пульсує" під його керівництвом, а процесор повинен звернути увагу на імпульси, що проходять.

Цей тип переривань є потенційним джерелом помилкових спрацьовувань, оскільки шум на лінії може виглядати як сигнал переривання. Крім того, складніша проблема виникає, коли два пристрої з таким типом переривань використовують одну лінію. У випадку, якщо два пристрої "просигналізують" одночасно, процесор зможе розрізнити лише одне переривання, і той факт, що переривання згенерували два пристрої, буде втрачений безповоротно.

Класичним прикладом устаткування, де трапляються втрати такого типу переривань, є послідовні COM-порти. За традицією, COM1 і COM3 використовують одне визначене по фронту переривання, а саме IRQ4. У результаті, неможливо одночасно підключити до цих двох портів устаткування, яке використовує кероване перериваннями програмне забезпечення. Спроби включити мишу в COM1 і модемі в COM3 рано чи пізно, але – неминуче приводе до зупинки одного з драйверів, який чекає "зниклий" сигнал про переривання.

Ці обмеження долаються реалізацією другого механізму, відомого як level-sensitive, чутливий до рівня, або level-triggered, керований рівнем. Пристрої утримують загальну апаратну лінію (спільно використовуваний провідник) у відповідному стані доти, доки потреба кожного не буде задоволена. Тепер для процесора немає проблем в тому, щоб зареєструвати обидва переривання, оскільки лінія утримується в цьому стані до моменту завершення обслуговування і першого, і другого пристрою – це дозволяють робити спеціальні типи цифрових схем. У випадку, якщо два переривання відбуваються одночасно, обслуговується пристрій з вищим пріоритетом, а після нього інший пристрій, оскільки воно продовжує сигналізувати, утримуючи лінію у відповідному стані і навіть після закінчення обробки запиту першого високопріоритетного пристрою.

У тих випадках, коли апаратне забезпечення системи містить більш ніж один процесор, зростає значення питання: як будуть оброблятися переривання? Як підключена лінія переривання пристрою – до одного процесора або до всіх?

Як правило, існують деякі компоненти апаратного забезпечення, які дозволяють драйверу конфігурувати і розподіляти сигнал переривання. Якщо окремий процесор обслуговує переривання, що поступають тільки від певного пристрою, то про такі переривання говорять, що вони мають "спорідненість до процесора" (processor affinity). Призначення конкретного процесора для обробки конкретного переривання може бути використане для контролю збалансованості завантаження окремого пристрою, якщо в системі присутні декілька процесорів і декілька керованих пристроїв.

2.3. Механізми передачі даних

Не дивлячись на різноманіття комп'ютерної техніки, існує три основні механізми, за допомогою яких пристрій може обмінюватися з центральним процесором або, в широкому сенсі, з комп'ютером:

1. Програмоване введення/виведення.
2. Прямий доступ до пам'яті (Direct Memory Access, DMA).
3. Області пам'яті, які спільно використовуються.

При виборі розробником апаратури механізму передачі даних, використовуваного для зв'язку з пристроєм, слід виходити з швидкості, з якою потрібно передавати дані, і середнім розміром безперервного блоку даних, що передається.

2.3.1. Програмоване введення/виведення

Пристрої, що використовують програмоване введення/виведення (Programmed I/O, PIO), здійснюють передачу даних безпосередньо через регістри пристрою. Драйвер повинен використовувати інструкцію введення/виведення (I/O instruction) для читання з регістра або запису в регістр пристрою кожного байта даних. При великих об'ємах драйвер повинен підтримувати адресацію буферної області й лічильник переданих даних.

Оскільки реальна швидкість передачі даних таких пристроїв набагато менше, ніж можливості процесора з читання або запису в регістри даних, то пристрої з таким підходом зазвичай генерують переривання для кожного байта (слова) даних, що передаються. Послідовний COM порт є одним з прикладів PIO пристроїв. Досконаліші пристрої використовують накопичувальні FIFO, що дозволяє їм в одне переривання передавати 4 або 16 байт даних. Та все ж кількість переривань щодо переданого об'єму даних вельми велика, тому цей метод придатний лише для повільних пристроїв.

2.3.2. Прямий доступ до пам'яті

Прямий доступ до пам'яті (Direct Memory Access, DMA) використовує вигоди від залучення до роботи вторинного процесора, що називається контролером DMA або DMA controller (DMAC). Цей контролер є допоміжним процесором з обмеженим набором можливостей і обов'язків, але при цьому з достатнім інтелектом і статусом, щоб виконувати передачу даних між пристроєм і оперативною пам'яттю. Контролер DMA працює паралельно з основним процесором (процесорами), і зазвичай його діяльність майже непомітна в загальному функціонуванні системи.

При ініціації операції введення/виведення із залученням DMA методу драйвер повинен виконати установку потрібного стану DMA контролера (запрограмувати його), визначивши адресу початку буферної області та кількість даних, що передається. По надходженню від драйвера вказівки почати роботу DMA контролер приступає до виконання передачі даних між пристроєм і системною оперативною пам'яттю. Коли DMA контролер завершує передачу даних повністю, генерується переривання. Таким чином, драйверний код виконується тільки на початку операції передачі даних, і потім, лише після її завершення, звільняючи центральний процесор для виконання інших завдань.

Слід звернути особливу увагу на те, що в Windows NT при DMA операціях завжди використовуються логічні адреси (трюк з пам'яттю, схожий з віртуальною адресацією, але призначений спеціально для DMA операцій), що

дозволяють “обдурити” DMA контролер, для якого логічні адреси здаються безперервними, хоча і можуть представляти фізично розривні області пам’яті.

Високошвидкісні пристрої, які вимушені регулярно займатися передачею крупних блоків даних, є першими кандидатами на використання методу DMA. В порівнянні з операціями програмованого введення/виведення, інтенсивність використання сигналів переривань і активність драйвера істотно зменшуються. Жорсткі диски, пристрої мультимедіа, мережеві карти є прикладами пристроїв, що використовують DMA.

Необхідно відзначити, що реальна DMA передача даних не є безумовно вигідною. Вторинний процесор, яким є DMAC, конкурує з центральним процесором у використанні пропускної здатності системної оперативної пам’яті і шин. У випадку якщо центральний процесор звертається до системної пам’яті часто, то хтось з них – або центральний процесор, або DMAC – вимушені чекати, доки попередній цикл звернення до пам’яті не закінчиться. Можна лише сподіватися, що при великих розмірах кеш-пам’яті сучасних процесорів, у останнього нечасто виникає крайня необхідність у використанні максимальної пропускної здатності пам’яті. Але все таки, система з великою кількістю пристроїв, що реалізовують bus master DMA (один з різновидів операцій прямого доступу до пам’яті), може виявити, що можливості одночасного доступу до пам’яті вичерпані.

Первинна специфікація персонального комп’ютера по IBM (і стандарти, що послідували) включали основну плату, що називається “материнською”, з набором загальних DMA котролерів. Кожен DMA контролер надає DMA канали (DMA channel), і даний пристрій може бути конфігурований так, щоб використовувати один (або більше) доступних каналів. Спочатку існували чотири канали, які були розширені до семи при введенні специфікації AT. Системний DMA (system DMA) відомий ще як slave DMA.

Перевага використання системного DMA полягає в тому, що кількість апаратної логіки для реалізації DMA в пристрої зменшується. До недоліків слід віднести те, що у разі сумісного використання каналу даним пристроєм, воно дістає можливість участі в DMA передачі даних тільки один раз за певний часовий інтервал. У кожен конкретний момент часу DMA канал знаходиться “у власності” тільки одного пристрою, решта спроб використання цього каналу з боку інших пристроїв відкладається до моменту, коли перший пристрій “відмовиться” від власності на канал. Таке сумісне використання каналу дає погані результати при використанні його для двох високошвидкісних пристроїв. Контролер флопі дисководів в більшості персональних комп’ютерів якраз є пристроєм, що реалізовує операції slave DMA.

Складніші пристрої, які не бажають бути залежними від системних контролерів DMA, містять власні апаратні засоби забезпечення DMA. Оскільки ці засоби належать власне пристрою, то передача відбувається по “волі” пристрою – якщо, зрозуміло, протокол шини підтримує такі дії. Для виконання своєї операції пристрою необхідно отримати “контроль над шиною” в результаті процедури, яка описується в протоколах відповідних шин.

Пам'ять, відведена пристрою

Третій з механізмів передачі даних полягає в тому, що для доступу до пристрою використовуються діапазони адрес пам'яті, відкриті для сумісного використання.

В деяких випадках ці діапазони визначені жорстко, як у випадку з діапазоном адрес 0x0A0000-0x0AFFFF буфера адаптера VGA.

Пам'ять, відведена пристрою, є ресурсом, що надається драйверу пристрою операційною системою, і зазвичай з ним можна ознайомитися в аплеті властивостей драйвера в настройках системи (Пуск / Настройка / Панель управління / Система / Пристрої / Диспетчер пристроїв / Пристрій / Ресурси).

Правильно спроектований пристрій повинен ідентифікувати (проявити) себе і надати системі перелік ресурсів, які воно споживає. Це перелік, що формулюється в деяких позиціях власне пристроєм, а в деяких – його драйвером, повинен включати:

- ідентифікатор виробника (Manufacturer ID);
- ідентифікатор типу пристрою (Device type ID);
- вимоги до простору введення/виведення (I/O space requirements);
- вимоги по використанню переривань;
- вимоги по використанню каналів DMA;
- вимоги щодо пам'яті, відведеної пристрою.

У разі PNP пристроїв, ідентифікатори виробника і типу пристрою є критерієм вибору драйвера при завантаженні системи або ж при підключенні пристрою (якщо воно було підключене після завантаження).

2.4. Шини в комп'ютерних системах

Шина (bus) представляє собою сукупність даних, адрес і ліній (провідників на друкарській платі або в кабелі чи шлейфі) сигналів контролю, які дозволяють пристроям організувати повідомлення між собою. Деякі шини є "широкими", забезпечуючи одночасну (паралельну) передачу багатьох бітів даних і бітів контролю. Інші є всього лише парою провідників, які дозволяють пристроям передавати дані сигнали, що управляються послідовно. Деякі шини дозволяють зв'язуватися з будь-яким іншим пристроєм на шині. Інші потребують наявність "господаря шини", master controller (центральний процесор або контролер введення/виведення), виступаючого в ролі одержувача або відправника даних.

Від того, якими властивостями володіє головна шина комп'ютера залежить продуктивність всієї системи. В середині 80-х років, наприклад, робоча станція Apollo на базі процесора 68020 була орієнтована на шину ISA як на стандарт. У даний час, персональні комп'ютери з Intel-архітектурою експлуатують шину PCI, що хоча і зазнала модифікації, але є головною.

Архітектура драйверів в ОС Windows NT 5 ефективно підтримує введення нових шин, підтримка багатьох популярних шин вбудована в стандартне постачання Windows. Тому розглянемо найбільш поширені з них.

2.4.1. ISA: Industry Standard Architecture

Шина ISA була запропонована фірмою IBM для PC/AT на початку 80-х років. Вона підтримувала підключення і 8-розрядних, і 16-розрядних пристроїв. З тієї причини, що ця шина була винайдена ще два десятиліття тому, вона не відрізнялася ні швидкістю, ні простотою конструкції. Проте є дві причини, через які про цю шину забути неможливо.

По-перше, такі популярні і до цих пір обов'язкові пристрої настільного комп'ютера, якими є паралельний порт LPT і послідовні COM порти, є пристроями ISA шини. Ця ISA шина через міст підключається до шини PCI, і існує в комп'ютері навіть тоді, коли на материнській платі немає жодного ISA роз'єму.

По-друге, при удосконаленні комп'ютерів, які ISA шина спочатку цілком влаштовувала, виявилося стільки незручностей і обмежень, що стало явним – до ухвалення шинних протоколів слід підходити відповідальніше. Невисока пропускна здатність (максимум 8 Мбайт/с), відсутність стандарту на використання адрес введення/виведення, неможливість сумісного використання переривань (їх всього 16, велика частина яких вже зайнята системними пристроями) наполегливо вказували на необхідність нових рішень. Наприклад, старі плати ISA не можуть використовувати все 16 розрядів адрес введення/виведення, а декодують тільки перші 10 розрядів адреси, внаслідок чого відгукуються по адресах через 0x400 адрес. Пристрій з адресою 0x300 відповідає також і за адресою 0x700. Поява таких пристроїв в системі приводить до пустування більшої частини 64 Кбайт адресного простору введення/виведення. І, нарешті, шина ISA має можливість всього лише 24-розрядної адресації, що визначає обмеження на виконання перенесення даних тільки в нижні 16 Мбайт фізичної пам'яті. Цей артефакт доставляє Windows особливі проблеми при роботі з ISA шиною, якщо виникає необхідність застосувати DMA спосіб передачі даних.

Як вже було сказано, пристрої ISA шини не оголошували себе, вони не надавали списку необхідних ресурсів, і вони не потребували динамічної конфігурації з боку програмного забезпечення. Конфігурація цих пристроїв виконувалася вручну і зазвичай зводилося до необхідної установки перемичок (jumpers) або перемикачів DIP.

Сучасні пристрої ISA намагаються скоректувати цю ситуацію, пропонуючи специфікацію PNP розширення до стандарту ISA. Такі пристрої значно розширили свою популярність з введенням ОС Windows 95. Версії NT, що існували до Windows 2000, реально не підтримували специфікацію PNP, чому цим новим пристроям ISA доводилося повністю покладатися на спеціальні програми інсталяції, які забезпечували їх належне функціонування під NT, і

послуги Системного Реєстру. Проте з виходом Windows 2000 підтримка таких пристроїв стала цілком коректною.

2.4.2. EISA: Extended Industry Standard Architecture

Специфікація шини EISA є розширенням архітектури ISA. Розширення EISA було спробою усунути обмеження ISA і при цьому уникнути проблем несумісності з існуючими пристроями ISA (платами ISA), відмовитися зовсім від яких на той момент було неможливо.

Зрозуміло, вимога на забезпечення сумісності обмежувала модернізацію шини по багатьом напрямам. Наприклад, в той час, як розрядність даних, що передається (data bus width), була збільшена до 32 бітів, тактова частота залишилася рівною 8 МГц. Відповідно, максимальна швидкість передачі даних по шині почала складати близько 32 Мбайт/с. Оскільки роз'єми EISA на материнській платі повинні були приймати ще й власне плати ISA, то з'явилися істотні складнощі в усуненні електричних шумів, обумовлених топологією розводки з'єднань ISA.

Пристрої EISA вже цілком забезпечували можливості автоматичної ідентифікації, хоча процедура ця була не дуже проста. В усякому разі, специфікація EISA здала свої позиції протоколу PCI достатньо легко, варто лише тому з'явитися. А ось прощання з ISA ще продовжується і сьогодні.

2.4.3. PCI: Peripheral Component Interconnect

Швидкі мережеві додатки, високоякісне відео, 16 і 20-розрядний звук і музичні додатки, що швидко розвиваються, дисплеї з глибиною кольору 24 бітів – всі ці нововведення зажадали введення нового стандарту шини з високими швидкостями передачі даних. Шина PCI з'явилася спробою задовольнити всі ці збільшені потреби. І хоча первинна конструкція зародилася в Intel (перша специфікація версії 1.0 з'явилася в червні 1992 р.), шині PCI надана підтримка з боку інших комп'ютерних виробників. Вона є незалежною від процесорних конфігурацій і успішно використовується тепер в комп'ютерах Alpha (DEC) і POWERPC (Motorola).

Підвищивши тактову частоту до 33 МГц (пізніше і до 66 МГц) і застосувавши безліч технічних прийомів, розробники PCI досягли того, що пропускна здатність PCI шини зросла до 132 Мбайт/с при передачі даних по 32-розрядній шині. При передачі даних по 64-розрядній шині пропускна здатність збільшується удвічі. До основних принципів, які дозволили забезпечити такий значний прогрес, можна віднести такі:

1. Протокол PCI має на увазі, що кожна передача даних відбувається як пакетно-монопольна операція (burst operation), внаслідок чого для швидких пристроїв, що використовують великі об'єми даних, дійсно досягаються фізично граничні показники PCI.

2. Протокол PCI підтримує режим багатьох “господарів шини” (bus master) і вирішує пряму передачу даних пристрій-пристрій (без проміжного використання пам’яті), що може бути використане для підвищення ступеня паралельності між операціями введення/виведення і роботою процесора.

3. Центральний шинний арбітр (central bus arbiter) суміщає операції арбітрування і власне перенесення даних, чим зменшується час затримок. Це дозволяє наступній операції стартувати відразу ж, як тільки поточна операція закінчується.

4. Міст PCI, наділений істотними здібностями, між головним процесором і власне шиною представляє різноманітні можливості кешування і виконання операцій попереджуючого читання даних (readahead functions). Все це дозволяє зменшити час, який процесор витрачає на очікування даних.

5. Міст PCI є пристроєм шини, що дозволяє підключати вторинні шини PCI. У такому разі міст називається “PCI-to-PCI bridge”. Зокрема, раніше це було вирішенням проблеми збільшення числа слотів PCI, якщо їх потрібно більше 4.

Архітектура PCI дозволяє підключити до шини до 32 фізичних одиниць, які називаються пристроями (devices). Кожна з цих одиниць може містити до 8 окремих функціональних одиниць, які називаються функціями (functions). Після відкидання однієї адреси функції для генерації широкомовних повідомлень на одній шині PCI (broadcast messages – повідомлень, призначених для усіх пристроїв) залишається адресованих 255 одиниць-функцій.

Доступ до регістрів

Шина PCI дозволяє використовувати 32-розрядні адреси, але системний простір введення/виведення все ще обмежений адресами до 64 Кбайт (16 розрядів). Відповідно, там повинні перебувати і всі регістри PCI. Більш того, в системах, що містять шини EISA або ISA, розробникам слід стримуватися і від використання адрес, які могли б використовувати застарілі пристрої, що підключаються до цих шин, по причинах, вказаних вище. Разом з адресацією простору введення/виведення і адресацією пам’яті, архітектура PCI визначає діапазон адрес у внутрішній пам’яті пристрою, що називається конфігураційним простором (configuration space).

Механізми переривань

Шина PCI використовує чотири рівно-пріоритетні лінії запитів на переривання INTA-INTD. Ці лінії є активними по низькому рівню (сигналом переривання є низький рівень в лінії), що перемикаються по рівню (інформативним значенням є логічний рівень – на відміну від того випадку, коли інформативним є фронт сигналу, тобто перехід від одного рівня до іншого). Дані лінії допускають можливість сумісного використання (це, можливо, стане технічно зрозумілішим, якщо сказати, що така лінія переривань реалізується по електронній схемі “Open drain”, N-МОП аналог “відкритого колектора” для ТТЛ логіки). Пристрій, що підключається до PCI і представляє одну функцію, повинен використовувати тільки лінію INTA.

Багатофункціональні пристрої можуть використовувати комбінації з чотирьох ліній, починаючи з INTA. Єдине обмеження полягає в тому, що кожна з восьми функцій (можливих в одному пристрої) може використовувати тільки одну лінію переривання. Відповідно, пристрій з внутрішніми вісьма функціями може задіяти наявні лінії INTA, INTB, INTC, INTD таким чином:

- всі вісім функцій підключено до INTA;
- сім підключені до INTA, одна до INTB;
- дві підключені до INTA, дві до INTB, дві до INTC і дві до INTD;
- чотири підключені до INTA, чотири до INTB і так далі.

Специфікація PCI відносно байдужа до пріоритетів переривань. Пріоритети, в даному випадку, залежать від зовнішнього контролера, який переадресує запит на переривання PCI пристрою у відповідну лінію системних переривань. Наприклад, на персональних комп'ютерах редиректор може перетворити запит функціональної одиниці PCI по лініях INTA-INTD в запит по одній з ліній IRQ0-IRQ15. Щоб можна було зробити це, будь-яка функція усередині PCI пристрою, яка генерує переривання, повинна задіювати два конфігураційні регістри (у своєму конфігураційному просторі):

- регістр виведення переривання (Interrupt Pin Register, Int PIN) – регістр "тільки для читання", який ідентифікує лінію переривання PCI (INTA-INTD), що використовується даною функцією даного пристрою PCI;
- регістр лінії переривання (Interrupt Line Register, Int Line) – регістр "тільки для запису", що указує пріоритет і вектор (номер системного переривання), які редиректор переривань повинен привласнити даній функції. У персональних комп'ютерах Intel x86 значення 0x00-0x0F відповідають IRQ0-IRQ15.

Така схема є достатньо гнучкою, оскільки не нав'язує жодних обмежень, обумовлених специфікою механізму переривань в системі, конструктору пристроїв або системи, що робить можливим використання цієї архітектури в процесорних середовищах, відмінних від Intel x86.

Можливості DMA

Специфікація PCI не включає поняття slave DMA. Замість цього, власні функції PCI (функціональні одиниці) є або "господарями шини", що виконують своє власне DMA перенесення даних, або вони використовують програмоване введення/виведення. Єдиним типом пристроїв, які можуть використовувати DMA перенесення даних з використанням системних контролерів (slave DMA), є не-PCI плати, підключені через ISA міст.

У DMA операціях власне PCI шини, учасники називаються агентами (agents), і в кожній транзакції завжди беруть участь два з них, а саме:

- ініціатор (Initiator) – це "господарів шини", який виграв право доступу до шини і хоче визначити операцію перенесення;
- мета (Target) – це PCI функція (функціональна одиниця PCI пристрою), що адресується зараз ініціатором з метою виконання передачі даних.

Оскільки, будь-який “господар шини” PCI може бути ініціатором, то можливо передача даних безпосередньо між двома PCI пристроями без проміжної зупинки даних в оперативній пам’яті. Ця потужна можливість просто призначена для використання у високошвидкісних мережевих і відео додатках.

Необхідно також згадати, що специфікація PCI не визначає стратегію, яка повинна бути використана при арбітруванні доступу до шини. Визначені тільки тимчасові діаграми арбітрованих сигналів в шині. Метод по знаходженню того “хто буде наступним” у використанні шини, визначається конкретною системою, де реалізовані шини PCI.

Пам’ять, відведена пристроям

Відведена пам’ять, що використовується PCI функціями (функціональними одиницями PCI пристроїв), може бути розміщена в будь-якому місці 32-розрядного адресного простору. Ця можливість може бути використана при роботі в базисі “функція пристрою” – “функція пристрою”.

Цікава можливість специфікації PCI полягає в тому, що в конфігураційному просторі передбачено місце для вказівки посилання на область розширення постійного запам’ятовуючого пристрою (ПЗП, Extension ROM Base Address), вбудованого в даний пристрій PCI. У цьому ПЗП можуть бути записані фрагменти кодів ініціалізації, призначені для ініціалізації пристрою на різних апаратних платформах, що дає можливість поставляти один і той же продукт для різних комп’ютерних платформ. Специфікація PCI визначає стандартний заголовний формат для цих блоків ПЗП, тому програмне забезпечення, що виконує ініціалізацію, може визначити необхідний фрагмент, який зберігається в ПЗП і завантажити саме його для виконання ініціалізації на існуючій платформі.

Автоматичне розпізнавання і конфігурація

Специфікація PCI зобов’язує кожну окрему функцію (функціональну одиницю пристрою, а значить, і шини) мати свою власну область для зберігання конфігураційних даних, розмір якої дорівнює 256 байтам. Ця область відома як конфігураційний простір функціональних одиниць PCI (PCI function’s configuration space).

Перші 64 байти такого конфігураційного простору називаються заголовком, мають зумовлену структуру, тоді як останні 192 байти можуть бути використані за бажанням розробника даної плати PCI. Системне програмне забезпечення може використовувати цей заголовок для ідентифікації функціональної одиниці PCI і виділення їй ресурсів.

Відображати всі 256 адрес конфігураційного простору в пам’яті або просторі введення/виведення не має сенсу, тому ОС дає можливість доступу до конфігураційного простору шляхом звернення до наступних двох регістрів (адреси яких встановлюються ОС):

- конфігураційний адресний регістр, який визначає номер шини (bus number), пристрій, функцію і адресу доступу до даних в конфігураційному просторі;
- конфігураційний регістр даних діє як буфер між процесором і конфігураційним простором. Після установки значення в адресному регістрі, читання або запис в цей регістр даних приводить до власне перенесення інформації із/в конфігураційний простір.

ОС надає HAL функції для доступу до конфігураційних даних. Для цього пропонується використовувати функції HalGetBusData, HalSetBusData і HalAssignSlotResources. Проте, в WDM моделі вони вважаються застарілими (навіть їх опис в DDK відсутній) – рекомендовано працювати через запити до нижнього драйвера стека.

Для ідентифікації драйвера, що завантажується ОС для обслуговування даного пристрою, специфікація PCI рекомендує розробникам ОС використовувати інформацію з конфігураційних регістрів Vendor ID, Device ID, Revision ID, Class Code, Subsystem Vendor ID, Subsystem ID (останні два стали обов'язковими тільки в специфікації версії 2.2).

2.4.4. IEEE 1394: Firewire Bus

Запропонована і реалізована Apple Computers, а згодом визначена інститутом IEEE як високошвидкісна однорангова (peer-to-peer) послідовна шина, IEEE 1394 призначена для додатків, коли USB шина версії 1.1 не могла використовуватися із-за низької швидкості передачі даних. Специфікація IEEE 1394 (на даний момент – IEEE 1394a-2000) описує три стандартні швидкості передачі даних 100, 200 і 400 Мбіт/с. Хоча специфікація IEEE 1394b підтримує великі швидкості, проте навіть при таких швидкостях, шина IEEE 1394 вже в змозі переносити більше 10 Мбайт/с, що перевищує показники ISA.

Найменування Firewire є торговою маркою Apple Computers. Термін 1394 зазвичай використовується для того, щоб позначити приналежність цієї специфікації до апаратного забезпечення персональних комп'ютерів. Фірма Sony й інші виробники відеокамер використовують для позначення своєї модифікації 1394 термін i.Link(TM).

Кожен пристрій IEEE 1394 може бути підключений до шини кабелем 4,5 м, що містить 6 або 4 провідників. До шини може бути підключене до 63 пристроїв шлейфовим чином (daisy-chained) при загальній довжині з'єднання 72 м. Можна використовувати шинні мости і пристрої розгалуження, що дозволяє підключити додатково до 62 пристроїв. При використанні шинних мостів можна задіювати до 1024 шин. В результаті, число пристроїв, яке теоретично можна підключити до комп'ютера в конфігурації 1394, складає 64 Кбайт. При підключенні пристрою, йому привласнюється 16-розрядний ідентифікатор (Node ID). Шестипровідниковий кабель містить два провідники для передачі даних, два для передачі сигналів синхронізації і два для енергопостачання пристроїв, що підключаються. Кабель екранований і

закінчується роз'ємом, який отриманий модифікацією роз'єму ігрової приставки Nintendo Gameboy.

Доступ до регістрів

Специфікація IEEE 1394 задовольняє стандарту IEEE 1212 архітектури Control and Status Register (CSR). Стандарт CSR визначає фіксовану 64-розрядну схему адресації, що включає 10-розрядний шинний номер і 6-розрядний ідентифікатор вузла (Node ID) з 48-розрядним полем, що залишається, для використання за бажанням пристрою.

Як у випадку із специфікацією USB, регістри призначені для управління і зберігання інформації про стан пристрою, а також для передачі невеликих (до 64 байт) пакетів даних. Синхронна передача використовується в тих випадках, коли необхідно витримати вказану пропускну здатність шини (наприклад, при роботі з відеокамерою), асинхронна передача даних використовується при гарантованому надходженні даних (наприклад, при читанні даних з жорсткого диска.)

Механізми переривань

Шина 1394 симулює переривання пристроїв (втім, як і шина USB). Пристрій повинен послати пакет даних для того, щоб повідомити хост-контролера про свій стан, коли потрібне втручання ОС. Драйвер, що відповідає за даний пристрій, повинен відреагувати на такий пакет даних, який розміщується IEEE 1394 інтерфейсом в системному адресному просторі.

Сімейство стандартів 1394 включає Open Host Controller Interface (OHCI), який є найбільш значущим стандартом для розробників драйверів, забезпечує загальний механізм роботи з перериваннями і DMA передачею даних.

Можливості DMA

Хост-контролер IEEE 1394 використовує DMA для передачі даних і команд в/із системної пам'яті. Пристрої, підключені до шини 1394, не можуть мати прямого доступу до системної пам'яті, тому адаптери OHCI надають діапазон адрес (у системній пам'яті),с якими, власне, і працюють прикладні програми і контролер DMA. Таким чином, виникає ілюзія можливості DMA передачі даних для кожного пристрою, підключеного до шини.

Автоматичне розпізнавання і конфігурація

Специфікація шини 1394 була розроблена з безпосередньою підтримкою специфікації PNP. Кожен пристрій при підключенні до шини сигналізує про своє існування при виконанні шинної операції "reset", що нагадує спосіб поводження з шиною USB і її пристроями. Хост-комп'ютер (або інші вузли) виконують операцію "перерахування" конфігураційних ПЗП для того, щоб виявити даний пристрій і забезпечити запуск (а при необхідності – і інсталяцію) відповідного драйвера.

2.4.5. USB: Universal Serial Bus

Специфікація USB була розроблена консорціумом компаній, включаючи Intel і Microsoft. Метою нового стандарту було забезпечення організації недорогої середньошвидкісної шини в таких областях застосування, як передача цифрових зображень, комп'ютерна телефонія та мультимедійні ігри. Поточними версіями специфікації USB є версія 1.1 і версія 2.0 (зрозуміло, в другу закладені вищі швидкісні характеристики), і список компаній-членів, що беруть участь в консорціумі, постійно росте.

Гранична швидкість передачі даних по шині USB специфікації 1.1 складає 12 Мбіт/с (Full Speed). Повільні використовують низьку швидкість передачі 1,5 Мбіт/с (Low Speed). Стандарт USB версії 2.0 підтримує фізичну швидкість передачі 480 Мбіт/с (High Speed). Реально ж максимальна швидкість з'єднання USB-пристрою комп'ютеар досягла 32 Мбайт/с. Дані передаються послідовно по парі провідників. Живлення для деяких пристроїв доступно по окремих провідниках живлення і заземлення (для пристроїв з невеликим енергоспоживанням).

Пристрої USB можуть бути підключені 5-метровим кабелем (а практично – і довшим). Використанні USB хаба (hub – мережевий концентратор), дозволяє збільшити дальність розміщення пристроїв від хост-комп'ютера і збільшити кількість пристроїв, що підключаються до однієї шини USB. Послідовно можна включити до п'яти пристроїв-хабів, забезпечивши довжину з'єднання 30 метрів.

Слід відмітити, що пристрої High Speed (для USB 2.0) повинні підтримувати функціонування і на швидкості Full Speed. Більш того, процес "перерахування" проводиться якраз на цій швидкості.

При конфігурації пристроїв при використанні зовнішніх USB хабів слід пам'ятати, що хаб, призначений для роботи на деякій швидкості (наприклад, Full Speed), обмежить швидкість роботи пристроїв, що стоять в ланцюжку пристроїв далі від хост-комп'ютера (наприклад, High Speed пристрої), і останні не зможуть працювати на швидкості, що перевищує швидкість цього повільного хаб-концентратора. За відсутності можливості працювати на швидкості Hi Speed пристрої Hi Speed працюють на швидкості Full Speed.

Робота програміста, що створює драйвер зовнішнього (що не знаходиться на материнській платі) пристрою USB зводиться до того, щоб скористатися програмним інтерфейсом системних драйверів шини USB, спілкування з яким відбувається за допомогою пакетів, що називається URB (USB Request Block) пакетами. Робота з регістрами USB контролерів на материнській платі тепер стала долею вузького кола фахівців – розробників материнських плат й ОС. Решті всіх розробників USB пристроїв з управлінням програм під Windows пропонується достатньо розвинений програмний інтерфейс до системних драйверів WDM, які беруть на себе всі апаратно-орієнтовані операції.

До хост-комп'ютеру можна підключити до 127 пристроїв, шинна адреса яких встановлюється динамічно при підключенні пристроїв.

Доступ до регістрів

Доступ до регістрів пристроїв (з боку хост-контролера і системних драйверів) здійснюється із застосуванням спеціальних команд USB, які застосовуються для конфігурації, управління енергоспоживанням і перенесення невеликих об'ємів даних. Проходження команд (призначених для конкретного пристрою) сильно залежить від пристрою, так що число і призначення команд визначається кожним пристроєм індивідуально шляхом заяви (передачі в хост-контролер) конфігураційних даних по відповідних запитах на початку роботи.

Механізм передачі даних є асинхронним і блоковим. У кожному блоці даних, що передається, який називають USB-фреймом, може передаватися до 1280 байт даних. Кожен фрейм займає фіксований часовий інтервал 1 мс.

Операція командами і блоками даних реалізується за допомогою такої логічної абстракції, як USB pipe-канал. Pipe-канал є каналом зв'язку між хост-контролером і кінцевою точкою (endpoint – ще одна логічна абстракція специфікації USB) зовнішнього пристрою, який зазвичай є буфером деякої протяжності усередині зовнішнього пристрою. Відкритий pipe-канал можна порівняти з відкритим файлом. До речі сказати, з ним асоційований дескриптор відкритого каналу.

Для передачі команд (і даних, що входять до складу команд) використовується pipe-канал за умовчанням (default pipe), тоді як для передачі даних може бути використана будь-яка кількість потокових pipe-каналів (stream pipes) і pipe-каналів повідомлень (message pipes).

Концепція "кінцевих точок" (endpoints) USB сильно нагадує концепцію функцій пристроїв PCI, оскільки після конфігурації пристрою за кінцевими точками усередині нього закріплюється і певна функція (0 – тільки функції управління і контролю, останні – за задумом розробника, але – на весь час роботи в системі).

Механізми переривань

Для шини USB справжнього механізму переривань, строго кажучи, не визначено. Замість цього, USB інтерфейс хост-комп'ютера опитує підключені пристрої на предмет наявності даних про переривання. Опитування відбувається у фіксовані інтервали часу, зазвичай кожні 1-32 мілісекунди. Пристрою дозволяється посилати у момент опитування до 64 байт даних.

З погляду драйвера, можливості роботи з перериваннями і DMA передачею даних фактично визначаються USB хост-контролером, який і забезпечує підтримку фізичної реалізації USB інтерфейсу. Достатньо багато зусиль було зроблено для стандартизації інтерфейсу хост-адаптера, внаслідок чого з'явилися два типи інтерфейсів: Open Host Controller Interface та Universal Host Controller Interface. Власне хост-контролер і підтримує загальноприйняті механізми переривань і DMA передачі даних. Зовнішні призначені для користувача USB-пристрої повинні лише пасивно і відповідно до протоколу брати участь в обміні даними.

Можливості DMA

Пристрої USB шини не мають прямого доступу до системної пам’яті. Вони ізольовані від системних ресурсів USB інтерфейсом хост-комп’ютера і не підтримують способу DMA передачі даних в звичному сенсі. Проте, USB інтерфейс хост-комп’ютера забезпечує “ілюзію” підтримки DMA для логічних ріре-каналів, що забезпечують доступ до кінцевих точок, тобто буферам, усередині підключених до шини пристроїв. У міру отримання даних від підключених пристроїв USB, інтерфейс хост-контролера використовує DMA доступ для того, щоб помістити отримані з пристрою дані в системну пам’ять. Таким чином, USB інтерфейс, що складається із зовнішнього пристрою, контролера в хост-комп’ютері та системних драйверах, підтримує можливість DMA передачі даних — або, строго кажучи, її ілюзію.

Автоматичне розпізнавання і конфігурація

Специфікація USB була розроблена з безпосередньою підтримкою специфікації PNP (іноді здається, що – з відвертою орієнтацією на WDM модель драйверів Windows). Кожний USB-пристрій при підключенні до шини USB сигналізує про своє існування і повідомляє ідентифікатор виробника (vendor ID) й ідентифікатор пристрою (device ID). Ці ідентифікатори є визначальною інформацією у виборі завантажуваного драйвера – відповідна інформація повинна бути присутньою в Системному Реєстрі (якщо нічого відповідного там не виявляється, то ініціюється процес установки нового драйвера).

2.4.6. PC Card (PCMCIA)

Близько десяти років тому, декілька компаній спільно розробили стандарт шинної архітектури для мобільних пристроїв. Спочатку, увагу було сфокусовано на платах оперативної пам’яті, і ця група отримала назву Personal Computer Card International Association (PCMCIA). Мобільні пристрої обмежені в розмірах і енергоресурсах, і на цьому зроблений акцент стандартів даної групи. На сьогодні більше 300 компаній є членами PCMCIA.

Спочатку, стандарт PC Card визначав 68-вивідний інтерфейс при одній з трьох величин товщини друкарської плати – типу I, II або III. Даний стандарт визначає швидкість передачі даних практично таку ж, що і для ISA шини. Але в даному випадку вищий пріоритет був встановлений для мінімізації енергоспоживання і розмірів, а не для поліпшення швидкісних параметрів шини.

Термін PCMCIA часто використовується замість терміну PC Card, і навпаки. Таке використання створює таку суперечність: PCMCIA є організацією, в той час, як PC Card визначає шинний інтерфейс. В даний час PCMCIA визначає, принаймні, три стандарти: PC Card, DMA і CardBus.

Спочатку, тактова частота шини PC Card була встановлена рівній тактовій частоті шини ISA, тобто 8 Мгц, що може бути використане при роботі з 8 і 16

розрядними пристроями. Відповідно, максимальна пропускна здатність шини при роботі 16-розрядними картами досягає 16 Мбайт/с.

Архітектура CardBus дозволяє використовувати 32-розрядні пристрої. Крім того, тактова частота збільшена до величини, визначеної для шини PCI, тобто 33 МГц, що збільшує пропускну здатність CardBus більше ніж 128 Мбайт/с.

Доступ до регістрів

Стандарт PC Card визначає 26 розрядний адресний діапазон (64 Мбайт). З іншого боку, адресація відбувається по схемі, схожою зі схемою, що використовується в ISA архітектурі. У архітектурі CardBus схема використання простору, що адресується 32-розрядною адресою, аналогічна схемі, яка застосовується в архітектурі PCI.

Механізми переривань

У стандартах PC Card і CardBus визначений один провідник (лінія) для переривань – IREQ, або CINT. Ця лінія переривань управляється рівнем (level sensitive), отже, може бути використана спільно декількома картами на одній і тій же шині. Проте при використанні багатофункціональних карт (якщо лінія переривань використовується спільно декількома функціями карти) PCMCIA повинно бути забезпечене арбітрування засобами програмного забезпечення.

Можливості DMA

Спочатку запропонований стандарт PC Card не припускав можливості DMA передачі даних. Пізніший стандарт, прийнятий в 1995 році, ввів DMA розширення в стандарт, який називається “DMA”. Це розширення стандарту дозволяє виконувати передачу 16-розрядних слів в манері, схожій із стандартом ISA. Цей стандарт має на увазі, що всі пристрої, підключені до шини, виступають в ролі “Bus slave” пристроїв, спільно використовуючих DMA контролери. Так само, як і в архітектурі ISA, важко реалізувати режим “Bus master DMA”.

Стандарт CardBus дозволяє виконувати операції DMA передачі даний майже таким же чином, як це виконується в архітектурі PCI. Дане доповнення, що дозволяє працювати з пристроями в ролі “Bus master”, є вельми важливим. 16 і 32 розрядних операції DMA передачі даних в стандарті CardBus можуть виконуватися при частоті 33 МГц. Правда, чинник обмеженості геометричних розмірів вимагає використання компонентів вищого ступеня інтеграції.

Автоматичне розпізнавання і конфігурація

Стандарт шини PC Card задовольняє специфікації PNP повністю, хоча ще продовжується внесення змін, що стосуються “гарячого підключення” пристроїв і автоматичної конфігурації.

Програмне забезпечення для стандартів PC Card або CardBus розділяється на два крупні фрагменти: сервіси сокетів (Socket Services) і служби карт (Card Services). Перший представляє програмне забезпечення рівня BIOS, і управляє в

системі одним або більш роз'ємами. Це програмне забезпечення відповідає за виявлення і оголошення факту підключення нового пристрою. Служби карт управляють апаратними ресурсами даної карти.

2.5. Корисні поради щодо роботи з апаратурою

Перед початком розробки нового драйвера, необхідно дізнатися про новий пристрій якомога більше. Намагайтеся виділити інформацію, що стосується, принаймні, наступних питань:

1. Шинна архітектура, що використовується.
2. Регістри управління і спосіб доступу до них з програмного забезпечення.
3. Спосіб повідомлення про поточний стан пристрою і помилки.
4. Поведінка пристрою у зв'язку з використанням переривань.
5. Механізми передачі даних.
6. Пам'ять, реалізована в пристрої.

Архітектура шини

Шинна архітектура, в яку повинен вписатися новий пристрій, матиме визначальне значення для конструкції нового драйвера – чи зможе він спиратися на лежачих нижче в стеку драйвера або змушений буде реалізовувати весь фізичний інтерфейс заново. Повинні бути зрозумілі всі особливості, що стосуються автоматичного розпізнавання і конфігурації нового пристрою. Слід розраховувати на те, що новий пристрій братиме участь в загальних механізмах PNP і обміну повідомленнями про зміни енергопостачання, що пропонуються Windows 2000/XP/Server 2003.

Регістри управління

Повинна бути відома схема адресації, а також розмір регістрів пристрою. Слід повністю проаналізувати призначення кожного регістра управління, кожного регістра стану і кожного регістра даних, так само як і можливі варіанти їх вмісту. Необхідно розглянути варіанти можливої незвичайної поведінки, наприклад:

- окремі регістри можуть бути регістрами тільки для читання або тільки для запису;
- один і той же регістр може представляти одні функції при операціях читання і абсолютно інші при операціях запису;
- регістри даних або стану можуть містити невірні дані до закінчення деякого тимчасового інтервалу після надходження команди-запиту;
- доступ до регістра (регістрів) може проходити в специфічній послідовності.

Отримання інформації про стан пристрою і про помилки

Слід визначити протокол, який пристрій використовує для повідомлення про свій поточний стан і для повідомлення про виниклі помилки. Необхідно володіти механізмом упевненого виявлення збоїв у функціонуванні пристрою.

Поведінка, пов'язана з використанням переривань

Необхідно достовірно з'ясувати, які сигнали переривань і за яких умов генерує дане апаратне забезпечення, і чи буде даний пристрій використовувати більше ніж один вектор переривань. При роботі з контролером, пов'язаним з багатьма пристроями, переривання можуть поступати і від самого контролера, так що слід з'ясувати механізм, як точно визначити пристрій-ініціатор переривання – на предмет, а чи до даного пристрою відноситься сигнал, що поступив.

Механізми передачі даних

Драйвери для роботи з пристроями, що підтримують програмоване введення/вивід, сильно відрізняються від драйверів, що реалізовують методи DMA передачі даних. Деякі пристрої підтримують обидва механізми введення/виведення. У разі DMA пристроїв, слід з'ясувати чи потрібна реалізація DMA механізму, в якому пристрій виступає в ролі "Bus master", або механізму, коли пристрій виконує дії з сценарію "Bus slave". Слід визначити, чи є обмеження на інтервал адрес фізичного буфера пам'яті, який може бути задіяний при цих операціях.

Використовуйте інтелект нового пристрою

Деякі периферійні пристрої містять власні процесори, які виконують операції діагностики і управління. Ці процесори можуть працювати під управлінням вбудованого програмного коду (прошитого в ПЗП), але можливо, що драйвер самостійно виконує завантаження коду, який управляє оперативною пам'яттю пристрою у момент його ініціалізації (як це має місце при роботі з мікросхемами USB контролерів фірми Cypress).

Використання вбудованих можливостей інтелектуальних пристроїв може дати істотно кращі результати в роботі і в діагностиці такої апаратури.

Тестування апаратури

Нова апаратура нечасто може бути вільною від помилок. З цієї причини її завжди слід піддавати попередньому тестуванню. Крім виявлення помилок, цей етап є періодом, коли автор пристрою дізнається багато нового про поведінку своєї розробки. Зазвичай використовують базисні та спеціальні тести.

Базисні тести означають, що перш за все, слід упевнитися, що всі пристрої і всі сполучні кабелі є сумісними і їх підключення виконане правильно і надійно. Наприклад, неекранований кабель USB задовільно працюватиме з USB 1.1 (Low і Full Speed), але, можливо, буде причиною збоїв при роботі на швидкості Hi Speed в USB 2.0. Що стосується підключення зовнішніх пристроїв до паралельного порту, то абсолютно однакові зовні кабелі LPT можуть

створювати абсолютно різні умови підключення, а при спробі підвищити швидкість передачі в режимах EPP або ECP неякісні кабелі можуть бути причиною різноманітних збоїв.

Після збірки всієї конфігурації слід виконати просте перезавантаження ОС. Її вдале завершення дасть упевненість, що новий пристрій не заважає роботі інших системних компонентів.

Спеціальні тести включають, при можливості, написання програмного коду, який ретельно тестує новий пристрій разом з тим, що міститься в ньому вбудованим або завантаженим програмним забезпеченням.

Нарешті, слід випробувати вбудовані засоби діагностики, ввівши пристрій в помилковий стан, що передбачається діагностичними засобами. Вбудоване програмне забезпечення повинне виявити цю помилку і видати належне повідомлення.

Висновки

- За умов багатозадачності виникає необхідність сповіщати процеси про події, що відбуваються в системі або інших процесах. Найпростішим механізмом такого сповіщення є сигнали. Основним механізмом, що забезпечує функціонування Ос є система переривань.
- Існує три механізми за допомогою яких, пристрій може обмінюватися з центральним процесором: програмоване введення/виведення, прямий доступ до пам’яті та спільно використовувана пам’ять.
- Архітектура драйверів в ОС Windows NT 5 ефективно підтримує введення нових шин, підтримка багатьох популярних шин вбудована в стандартне постачання Windows. Найбільш поширеними є ISA, EISA, PCI, IEEE1394, USB, PC Card.

Контрольні запитання та завдання

1. Виділіть та охарактеризуйте декілька загальних рис підключення пристроїв до комп’ютера.
2. Наведіть відповідність між традиційними інструкціям DOS/Windows асемблера і макровизначеннями HAL.
3. Сигнали, пріоритети та вектори переривань.
4. Проведіть порівняльну характеристику шини ISA та EISA в комп’ютерних системах.
5. Проведіть порівняльну характеристику шини PCI та IEEE 1394 в комп’ютерних системах.
6. Проведіть порівняльну характеристику шини USB та PC Card в комп’ютерних системах.
7. Які поради можна дати розробнику драйверів щодо роботи з апаратурою?

8. Як буде представлено на асемблері x86 читання чотирьох байтів з порту введення/виведення?
9. За допомогою, якого макровизначення HAL буде проведено запис 16-розрядного слова в порт введення/виведення?

Тести для самоконтролю

1. Як називаються спеціальні команди процесора, за допомогою яких здійснюється доступ до регістрів пристроїв?
    a. інструкції введення/виведення
    b. IN/OUT
    c. порти введення/виведення
    d. адресний простір введення/виведення
2. Яка із команд повинна зареєструвати callback-функцію, що називається процедурою відкладеного виклику для обслуговування переривання?
    a. Input Request Packet
    b. Interrupt Service Routine
    c. Interrupt ReQuest Level
3. Як називається метод обміну даними між пристроєм і оперативною пам’яттю без участі центрального процесора?
    a. Deferred Procedure Call
    b. Direct Memory Access
    c. Remote Procedure Call
4. Як розшифровується назва шини EISA?
    a. Extended Industry Standard Architecture
    b. Extended Industry Standard American
    c. Extended Interconnect Standard Architecture
    d. Extended Interconnect Standard American
5. Яка шина зобов’язує кожну окрему функцію мати свою власну область для зберігання конфігураційних даних, розмір якої дорівнює 256 байт?
    a. ISA
    b. EISA
    c. PCI
    d. IEEE 1394
    e. USB
    f. PC Card

Теми для самостійного опрацювання

1. Система пріоритетів.
2. Контролери переривань на платформі I64.
3. Трасування повідомлень ядра.

4. Система пріоритетів ОС Windows NT.
5. Планування процесів і потоків в операційній системі Windows NT.
6. Механізми управління ОС Windows NT.
7. Механізми передачі даних в ОС Windows XP.

Список використаної літератури

1. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
2. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.
3. *Руссинович М.* Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс / М. Руссинович, Д. Соломон; пер. с англ. – [4-е изд.]. – М.: Русская редакция, СПб.: Питер, 2005. – 992 с.
4. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО “Бином-Пресс”, 2004. – 480 с.
5. *Сорокина С. И.* Программирование драйверов и систем безопасности : учебное пособие / С. И. Сорокина, А. Ю. Тихонов, А. Ю. Щербаков. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2002. – 256 с.
6. *Шеховцов В. А.* Операційні системи / В. А. Шеховцов. – К.: Видавнича група BHV, 2008. – 576 с.
7. *Харт Джонсон М.* Системное программирование в среде Windows / Джонсон М. Харт; пер. с англ. – [3-е изд.]. – М.: Издательский дом “Вільямс”, 2005. – 592 с.

РОЗДІЛ 3. ІНТЕРФЕЙС NATIVE API WINDOWS NT 5

План

- Особливості механізмів перехоплення функцій API в режимі користувача
- Перехоплення викликів API в системах Windows NT і Windows 9X
- Створення і виконання віддаленого коду
- Зміна таблиць імпорту
- Методи встановлення глобальних перехоплювачів
- Призупинення потоків
- Особливості Native API
- Перехоплення функцій NtOpenFile і NtCreateFile

В даному розділі розглядаються особливості механізмів перехоплення функцій API в режимі користувача та перехоплення викликів та функцій API, наводиться огляд самої структури Native API.

3.1. Особливості механізмів перехоплення функцій API в режимі користувача

За часів MS DOS жодна серйозна програма не обходилася без перехоплень переривань – сервісів системи для установки на них своїх процедур-обробників. Це було необхідно, наприклад, для забезпечення “псевдо-багатозадачності” (pop up), реакції на таймер в режимі реального часу, отримання розширеної інформації про одночасно натиснуті користувачем клавіші і т.п. Установка своїх обробників могла здійснюватися навіть без відома системи, – правда, тоді була ймовірність того, що DOS виділить пам’ять, в якій знаходиться цей обробник для якої-небудь програми, що загрожує крахом системи. Для дрібних резидентів це було не страшно, а великі “захищалися” кількома способами – наприклад, маркуванням області DOS, яка використовується в системних цілях (установка маркера 80h). Резиденти могли привести систему в нестабільний стан. Щоб цього не сталося, розробники повинні були враховувати дуже велику кількість нюансів і “підводних каменів” ОС і застосовувати відповідні заходи. До того ж, обробники переривань писалися найчастіше на мові Java з метою економії пам’яті та збільшення швидкості роботи. Звідси випливає, що якісні резиденти – дуже необхідні в програмуванні під DOS техніку – писали тільки програмісти досить високої кваліфікації.

Системи Win32 претендують на те, що жоден користувальницький процес не може порушити або вплинути на роботу іншого або викликати крах системи.

Саме поняття “резидент” в Win32 втрачає свій сенс, тому що кожен процес працює в своєму контексті пам’яті. Тому розробники цих ОС відмовилися від такого механізму, залишивши тільки можливість обробки віконних повідомлень (можливо, навіть чужого процесу). Обробка віконних повідомлень – це єдиний метод отримання інформації про дії користувача для GUI програм.

Розробники Win32, звичайно, надали сервіси ОС для здійснення того, що в DOS могло бути зроблено тільки шляхом установки резидента. Частина з них реалізована через систему віконних повідомлень, частина – через сервіси 3-го кільця (user mode), а решта – у вигляді низькорівневих сервісів, які можуть використовуватися тільки драйверами (kernel mode). Тобто, єдиний в минулому механізм розділився на абсолютно різні по суті підсистеми ОС. Це накладає великі обмеження на його використання.

Наприклад, користувач NT (Win2k) з привілеями користувача або гостя хоче захиститися від атак з мережі на відмову, небажаних з’єднань, або перевірити, чи не передається від нього будь-яка інформація без його відома. Це може бути вирішено шляхом установки драйвера – аналізатор трафіку (sniffer) його мережевої карти, тобто персонального файрволла (firewall). Проблема полягає в тому, що він не може встановити свій драйвер в систему, тому що не має адміністраторських прав. Це обмеження дійсно необхідне – адже в іншому випадку навіть гість може зробити з системою все, що завгодно, оскільки код драйвера виконується в нульовому кільці. Гість, наприклад, може отримати адміністраторські права, використовуючи Winlogon.exe.

Однак, це ж завдання може бути вирішене без використання драйверів з практично тією ж ефективністю.

Розробка механізмів перехоплення функцій API безпосередньо в режимі користувача включає декілька методів, розглянемо їх більш детально.

Метод 1: Безпосередня модифікація таблиць імпорту/експорту процесу, шляхом заміни в них покажчиків на оригінальні функції покажчиками на власні функції-заглушки.

До основних переваг даного методу відносять:

1. Досить проста схема реалізації. Гарантія неможливості пошкодження вмісту адресного простору випадкового процесу.

2. У разі помилки помирає тільки піддослідний процес/процеси, а не вся система.

До недоліків даного методу можна віднести:

1. Використання складних методів впровадження власного коду в адресний простір чужого процесу (установка глобальних хуків (SetWindowsHookEx ()), завантаження DLL через реєстр, безпосереднє впровадження коду в адресний простір піддослідного процесу (WriteProcessMemory (), CreateRemoteThread () ...).

2. Неможливість провести дані маніпуляції з будь-яким процесом, оскільки Windows NT забезпечує практично всі системні об’єкти дескрипторами безпеки, що визначають можливість і ступінь володіння одного об’єкта іншим.

3. Що, якщо системна функція А, що перехоплюється базується на іншій системній функції В і процесу раптом необхідно напряму звернутися до функції В? Доведеться шукати такі ж функції В і теж створювати для них перехоплювач.

4. Можливість організувати спостереження за функціями, що викликаються лише процесами, що обробляються, а не всіма процесами.

Метод 2: Модифікація частини коду бібліотек, що містять функції, які відслідковуються (метод сплайсінгу коду, наприклад, шляхом впровадження в початку функцій машинної інструкції Jmp 0xXXXXXXXX на тіло обробника).

Перевагою даного методу є те, що не потрібно проводити модифікацію таблиць імпорту модулів процесу. У Windows 9X метод дає можливість відстежувати звернення до функцій системних бібліотек з усіх процесів.

До недоліків даного методу можна віднести:

1. Ризик привести систему до недієздатного стану, оскільки в момент модифікації коду бібліотек може відбутися перемикання контексту, і, у випадку, якщо модифікація не була завершена, а ця функція була викликана іншим процесом або потоком, то, існує велика ймовірність того, що спрацює виключна ситуація, а в гіршому випадку система видасть BSOD. Те ж саме може відбутися, якщо код перехоплювача даної функції знаходиться в контексті процесу, відмінного від того, який викликається.

2. Для здійснення даного методу необхідно деяким чином модифіковані сегменти коду бібліотек, що мають атрибути “ReadExecute”.

І, нарешті, ще один варіант – це використання спеціально створеного налагоджувального механізму Windows (Debug API).

3.2. Перехоплення викликів API в системах Windows NT і Windows 9X

Не можна стверджувати, що адреси функцій навіть в системних бібліотеках (наприклад, Kernel32.dll) не змінюються в залежності від версії ОС, її збірки або навіть конкретної ситуації. Це відбувається через те, що бажана база образу бібліотеки (dll preferred imagebase) є константою, яку можна змінювати при компіляції. Більш того, зовсім не обов’язково, що dll буде завантажена саме за бажаною адресою, – цього може не відбутися в результаті колізії з іншими модулями, динамічно виділеної пам’яттю і т.п. Тому статичний імпорт функцій відбувається на ім’я модуля та ім’я функції, що надається цим модулем. Завантажувач PE файлу аналізує його таблицю імпорту та визначає адреси імпортованих функцій. У випадку, якщо в таблиці імпорту вказана бібліотека, не присутня в контексті, відбувається її відображення в необхідний контекст, настроювання її образу і ситуація рекурсивно повторюється. У результаті в потрібному місці певної секції PE файлу (що має атрибут “readable”) заповнюється масив адрес імпортованих функцій. В процесі роботи кожен модуль звертається до свого масиву для визначення точки входу в яку-

небудь функцію. Звідси випливає, що для перехоплення функцій в рамках одного з контексту можна діяти двома способами.

Перший спосіб полягає в наступному. Визначається адреса обробника функції, розташованого, наприклад, у бібліотеці в контексті даного процесу. Визначається справжня адреса функції і проводиться заміщення перших 5 байт її коду на стрибок до обробника. Для виклику вихідного коду функції обробник повинен відновити колишні байти коду, зробити виклик функції, і після повернення управління відновити опкод JMP на початку функції, інакше наш обробник ніколи не отримає управління знову. Цей метод називається "сплайсінг". Розглянемо його сильні і слабкі сторони.

До позитивних сторін відносять:

1. Так як виправляється тільки пам'ять модуля, в якому проводиться перехоплення, то код буде викликатися в результаті викликів функції як старими модулями, так і тими, що динамічно підвантажуються, так як адреса перехоплюваних функцій не змінилася – зроблена лише "врізка" декількох байтів. Коли викликається функція, що перехвачується, код обов'язково буде викликатися.

2. Будуть перехоплюватися виклики навіть усередині обробленого модуля, тому що неважливо, яким чином перехоплена точка входу функції отримала управління.

3. Стає елементарною деінсталяція слухача – достатньо відновити початкові байти.

До негативних сторін відносять:

1. У системах Win95/98 деякі системні бібліотеки (kernel32, user32 та ін) завантажуються в адресний простір, що проектується на всі контексти, присутні в системі. Це ускладнює процес перехоплення їх функцій, так як вміст пам'яті понад 2 Гбайти не може бути змінено стандартними документованими API функціями. Встановити дозвіл на запис у цю область пам'яті може функція _PageModifyPermissions, що викликається за допомогою kernel32! VxDCall0 і має номер 1000dh. Після встановлення атрибуту "writable" на необхідний регіон пам'яті і здійснення запису в нього зміни відбудуться в усіх присутніх контекстах одночасно. Значить, код обробника повинен розташовуватися за однаковими адресами у всіх контекстах. Це можливо тільки тоді, коли код буде знаходитися вище 2 Гбайти – цього можна досягти декількома способами. Ось найпоширеніші з них: безпосередня вказівка необхідного imagebase лінкеру динамічної бібліотеки, що завантажується при компіляції; розташування коду в міжсекційному просторі будь-якого модуля, завантаженого вище 2 Гбайти; виділення пам'яті, що спроектована на всі контексти (наприклад, з файлу підкачки) і копіювання туди свого коду. Таким чином, обробник буде отримувати управління в результаті виклику функції, що перехоплюється в будь-якому контексті. Якщо ж необхідно здійснити перехоплення функції тільки в певному контексті, обробник повинен викликати GetCurrentProcessId () для отримання відомостей про абонента. Якщо ж перехоплення звужується до

певного модуля, то крім ідентифікації поточного процесу необхідний аналіз стека для визначення модуля, що викликається.

2. Якщо обробник отримав управління, відновив вихідні байти і зайнявся чимось, то інший потік в цей момент може викликати цю функцію, тобто виклик перехоплений не буде – реєнтерабельність обробників не гарантується навіть при правильній організації коду. Ця проблема в загальному випадку не має рішення.

Інший метод виглядає так. Визначається точка входу функції, що перехоплюється. Складається список модулів, що в даний момент завантажені в контекст необхідного процесу. Потім перебираються дескриптори імпорту цих модулів для пошуку адрес функції, що перехоплюється. У разі збігу ця адреса змінюється на адресу обробника.

Позитивні сторони цього методу включають:

1. Здійснюється реєнтерабельність обробника, так як код функції, що перехоплюється не змінюється взагалі.

2. Стає більш гнучким вибір модуля/процесу, в якому необхідно здійснити перехоплення. Тіло обробника може знаходитися за різними адресами у різних контекстах.

Негативні сторони полягають:

1. Вже ініціалізовані модулі можуть зберегти справжню адресу функції і згодом викликати саме його, пропускаючи обробник.

2. Необхідно якось поширювати дані обробника на модулі, що динамічно завантажуються. Це в свою чергу можна здійснити кількома способами:

– у модулі, що містить код функції, який перехоплюється, змінюється таблиця експорту – точніше, відносна віртуальна адреса (RVA) функції, що перехоплюється. Завантажувач PE буде вказувати ці значення (+ imagebase) в таблиці імпорту нових модулів. Також функція GetProcAddress повертатиме адреси обробників. Цей спосіб має істотний недолік. Розглянемо приклад: нехай перехоплюється деяка функція з kernel32. Процесом був викликаний GetProcAddress (kernel32_base, & kernel_function_name) для отримання її адреси. Так як, kernel32 завантажений у всіх контекстах з однією і тою ж адресою, то адресу, яка повертається GPA, можна використовувати для виклику функції віддаленим кодом. Далі процес виділяє пам’ять, наприклад, в контексті свого дочірнього процесу, і після цього копіює віддалений код в цю область пам’яті. Після цього він викликає CreateRemoteThread для створення віддаленого потоку. Але, так як перехоплено одну з функцій, що викликається цим потоком, то її адреса більше не належить регіону kernel32. Навіть якщо ця функція перехоплена і в дочірньому процесі, то не можна гарантувати однозначність адреси обробника в обох контекстах. Тобто, дуже ймовірно, що при виконанні віддаленого потоку виникне ситуація передачі управління не на обробник – наприклад, на неініціалізовану сторінку, яка не має атрибут “executable” або взагалі в

чужий код. Будь-який з цих випадків приведе до GPF і процес-мішень буде аварійно завершено;

– щоб уникнути ситуації, описаної вище, не можна перехоплювати функції, що часто використовуються як початкова точка віддаленого потоку – LoadLibraryA, LoadLibraryW. Вони є всього лише перехідниками до більш потужних функцій – LoadLibraryExA, LoadLibraryExW. Але їх перехоплення не вирішить проблему, тому що перехід до них у kernel32 робиться коротким – по відносному зсуву, не використовуючи таблицю імпорту. Але при більш детальному вивченні цих функцій виявляється, що LoadLibraryExA зводиться до LoadLibraryExW, а та, в свою чергу, до недокументованої функції ntdll! LdrLoadDll. Її перехоплення необхідне для вирішення двох завдань: інсталяції перехоплення функцій на динамічні бібліотеки та можливості встановлення обробників в ланцюжок.

3.3. Створення і виконання віддаленого коду

Віддалений код – код, який виконується поза контекстом, що спочатку його містить. Читання-запис процесом пам'яті здійснюється функціями kernel32! ReadProcessMemory і, відповідно, kernel32! WriteProcessMemory. Синтаксис виклику цих функцій ідентичний; одним з параметрів є хендл процесу, над яким виконується операція – процес "відкривається" для будь-яких (певних) дій. Проводиться це функцією kernel32! OpenProcess.

В операційних системах, які підтримують систему привілеїв (NT/Win2k) можливість успішного відкриття процесу залежить від поточного рівня привілеїв процесу, що виконується. Зокрема, функція OpenProcess по відношенню до процесу winlogon.exe виконається тільки при включеному привілеї SE_DEBUG_PRIVILEGE. Її в більшості випадків мають тільки адміністратори, так як вона дозволяє маніпулювати з усіма без винятку процесами системи, що загрожує крахом самої ОС або її системи безпеки. Таким чином, можливість читання-запису пам'яті чужого процесу обмежена привілеями користувача. Так як читання-запис пам'яті – головна проблема всієї ідеології перехоплень API, то явним є те, що існує клас завдань, нездійсненних користувачем, що не мають, наприклад, адміністраторських прав. Однак, такий користувач може застосувати перехоплення функцій до будь-якого процесу, запущеному (прямо чи опосередковано) ним самим (CreateProcess і аналоги), що в більшості випадків і потрібно.

Розглянемо варіанти впровадження динамічних бібліотек в чужий контекст. Як відомо, після ініціалізації dll відбувається виконання її точки входу (як PE файлу) з трьома параметрами. Саме в цій процедурі буде знаходитися код, який здійснює перехоплення функцій:

1. *Використання стандартної функції SetWindowsHookEx*. Ця функція встановлює хук на віконні повідомлення. Вона призначена для відстеження повідомлень вікон як свого процесу, так і чужого. У цьому випадку код

обробника повинен знаходитися в dll. Результати роботи функції в загальних рисах такі:

- вона проектує бібліотеку на всі контексти, які задовольняють таким умовам: їх потоки мають в поточний момент GUI-вікно, що приймає повідомлення від користувача, а їх процеси можуть бути успішно відкриті користувачем функцією OpenProcess з параметром PROCESS_ALL_ACCESS;
- дякуючи такому способу установлений глобальний обробник вловлює все, що можна вловити. Найбільш цікавим моментом є те, що функція автоматично проектує бібліотеку і на нові GUI-процеси – процеси, яких не було в момент її виклику.

Наприклад, в результаті виконання нижченаведеного коду бібліотека myhookingdll буде спроектована на всі GUI-вмісні контексти, і всі вони отримають повідомлення DLL_PROCESS_ATTACH:

```
call LoadLibraryA, offset myhookingdll, 0
call GetProcAddress, eax, offset mydummyhook, eax
call SetWindowsHookExA, WH_CBT, eax
```

2. *Використання віддалених потоків*. У WinNT і Win2k мають місце функції для створення віддалених потоків. З них документована одна – kernel32! CreateRemoteThread. Як параметри до неї передаються: описувач процесу, в якому буде створений потік, його стартова адреса в чужому контексті, аргумент функції потоку (її прототип – DWORD WINAPI ThreadFunc (PVOID pvParam)), початковий розмір стека та інші. Найпростіший спосіб впровадження бібліотеки в чужий контекст з використанням CreateRemoteThread – вказати стартову адресу потоку як адресу функції LoadLibraryA (або W) і помістити як параметр вказівник на ім'я бібліотеки. Для цього потрібно виконати такі кроки:

- відкрити чужий процес функцією OpenProcess як мінімум з правами PROCESS_CREATE_THREAD PROCESS_VM_WRITE;
- виділити пам'ять в чужому контексті для розміщення імені бібліотеки з шляхом (функція VirtualAllocEx);
- скопіювати туди повне ім'я бібліотеки (функція WriteProcessMemory);
- знайти адресу функції LoadLibraryA (W);
- виконати CreateRemoteThread з початковим EIP, рівним отриманій адресі LoadLibrary *;
- закрити процес і потік.

Вищесказане ілюструє нижченаведений код:

```
call OpenProcess, PROCESS_ALL_ACCESS, 1, PID
xchg eax, ebx
call VirtuallAllocEx, ebx, 0, mydllnamesize, MEM_COMMIT, \
       PAGE_READWRITE, ebx
call WriteProcessMemory, ebx, eax, \
       offset mydll, mydllnamesize, 0, eax, 0,0
call GetModuleHandleA, offset kernel32
call GetProcAddress, eax, offset _LoadLibrary
```

```
  call CreateRemoteThread, ebx, 0,0, eax
  call WaitForSingleObject, eax, INFINITE, eax
  call CloseHandle
  call CloseHandle
```

3.4. Зміна таблиць імпорту

Коли компілятор зустрічає у вихідному тексті виклик функції, яка присутня не у виконуваному файлі, а в деякому іншому – частіше за все, в dll, у простому випадку він генерує “call” на цей символ. Згодом лінкер виправляє цей псевдовиклик на виклик перехідника (“stub”), використовуючи бібліотеку імпорту, що містить перехідники для всіх експортованих символів в зазначених бібліотеках. Такі перехідники складаються з однієї інструкції – “jmp [x]”, де x – адреса подвійного слова в таблиці імпорту PE файлу. Ці адреси завантажувач PE файлу заповнює коректними значеннями при ініціалізації модуля, спираючись на дані, наведені в таблиці імпорту.

У складніших випадках (при безпосередній вказівці імпортованої функції) компілятор генерує “call [x]”, минаючи перехідник. Таблиця (директорія) імпорту повинна розташовуватися в секції, що має атрибути “ініціалізовані дані” і “що читається” (IMAGE_SCN_CNT_INITIALIZED_DATA і IMAGE_SCN_MEM_READ). Таблиця імпорту складалася з масиву структур – дескрипторів імпорту (IMAGE_IMPORT_DESCRIPTOR), завершальним елементом якого є нульова структура. Дескриптор імпорту виглядає наступним чином:

```
IMAGE_IMPORT_BY_NAME STRUC
IBN_Hint DW?
IBN_Name DB 1 DUP (?); Довжина не фіксована
IMAGE_IMPORT_BY_NAME ENDS

IMAGE_THUNK_DATA STRUC
UNION
TD_AddressOfData DD IMAGE_IMPORT_BY_NAME PTR?
TD_Ordinal DD?
TD_Function DD BYTE PTR?
TD_ForwarderString DD BYTE PTR?
ENDS
IMAGE_THUNK_DATA ENDS

IMAGE_IMPORT_DESCRIPTOR STRUC
UNION
ID_Characteristics DD?
ID_OriginalFirstThunk DD IMAGE_THUNK_DATA PTR?
ENDS
ID_TimeDateStamp DD?
ID_ForwarderChain DD?
ID_Name DD BYTE PTR?
ID_FirstThunk DD IMAGE_THUNK_DATA PTR?
```

```
IMAGE_IMPORT_DESCRIPTOR ENDS
```

де:

– ID_OriginalFirstThunk і ID_FirstThunk містять відносні віртуальні адреси (RVA) структур IMAGE_THUNK_DATA, що описують імпортовані функції;
– ID_TimeDateStamp містить "штамп часу" модуля-експортера і використовується при технології "bound imports". Якщо імпорт не пов'язаний, то значення цього поля = 0;
– ID_ForwarderChain містить RVA першого форвардера у списку імпортованих функцій. Якщо форвардери відсутні, то значення цього поля дорівнює "-1". ID_Name містить RVA імені імпортованого модуля;
– масиви ID_OriginalFirstThunk і ID_FirstThunk йдуть паралельно. Два масиви необхідні для збереження інформації про імпортовані функції – масив ID_OriginalFirstThunk залишається незмінний, а масив ID_FirstThunk буде заповнений адресами необхідних функцій (RVA і [illegible] функцій [illegible]).

Модуль може експортувати функції, реалізація яких у ньому не присутня, а лише імпортується з іншого модуля. Цей прийом називається "форвардинг функцій". Його можна прослідкувати, наприклад, у wsock32.dll. Ця бібліотека експортує безліч функцій, частина з яких є перехідниками до функцій ws2_32.dll, а частина реалізована в модулі.

Суть перехоплення функцій методом виправлення таблиці імпорту полягає в наступному:

– визначається адреса перехоплюваних функцій;
– виходячи з даних PE заголовка модуля-жертви обчислюється адреса його таблиці імпорту;
– серед дескрипторів імпорту шукається той, який описує імпорт з модуля, що містить шукані перехоплювані функції;
– перебираються всі структури IMAGE_THUNK_DATA, починаючи з RVA = ID_FirstThunk знайденого дескриптора у пошуках полів TD_Function, що містять адреси перехоплюваних функцій;
– знайдені адреси замінюються на адресу обробника перехопленої функції.

Код функції, що здійснює перехоплення вищеописаним способом, може виглядати так (форвардинг функцій не враховується):

```
hook_api proc modbase: dword, modname: dword, \
                             procname: dword, hook_proc: dword
                        local oldproc: dword
                          local dummy: dword
            pushad
            @ SEH_SetupFrame "jmp bad_exit"
            call IsBadCodePtr, hook_proc; перевірка коректності
                                             ; виклику
            test eax, eax
            jnz bad_exit
```

```
              push procname; крок 1 – дізнаємося адресу
              call GetModuleHandleA, modname; перехоплюваних Функцій
              call [realGetProcAddress], eax
              test eax, eax
              mov oldproc, eax
              jz bad_exit
              mov edi, modbase
              call IsBadReadPtr, edi, 40h
              test eax, eax
              jnz bad_exit
              cmp word ptr [edi], 'ZM'
              jnz bad_exit
              mov ebx, [edi.MZ_lfanew]
              push 0F8h
              add ebx, edi
              call IsBadReadPtr, ebx
              test eax, eax
              jnz bad_exit
              cmp dword ptr [ebx], 'EP'
              jnz bad_exit
              mov esi, [ebx.NT_OptionalHeader \; крок 2: одержання
              . OH_DirectoryEntries \; адреси
              . DE_Import \; таблиці імпорту
              . DD_VirtualAddress]
              or esi, esi
              jz bad_exit
             add esi, edi
             cmp esi, ebx
             jz bad_exit
             stc
             mov eax, [esi.ID_Name]
             test eax, eax
             jz no_imps
next_imp_desc:; крок ь4: перебираємо дескриптори імпорту
             push esi
             push edi
nxtchar__: call patchthisidesk

             pop edi
             pop esi
             mov eax, [(esi \
                       + IMAGE_SIZEOF_IMPORT_DESCRIPTOR). ID_Name]
             add esi, IMAGE_SIZEOF_IMPORT_DESCRIPTOR
             test eax, eax
             jnz next_imp_desc
no_imps: popad
             xor eax, eax
             jc simpleret
             mov eax, oldproc
simpleret: @ SEH_RemoveFrame
             ret

bad_exit: @ SEH_RemoveFrame
             popad
             xor eax, eax
             stc
             ret
```

```
bad_exit4proc:
              pop eax
              jmp mismatch

patchthisidesk:; крок 3: шукаємо вхід
              add esi, 0ch; перехоплюваних функцій
              call IsBadReadPtr, esi, 8; в дескрипторі імпорту
              sub esi, 0ch
              test eax, eax
              jnz bad_exit4proc
              mov eax, [esi.ID_Name]
              test eax, eax
              jz bad_exit4proc
              mov esi, [esi.ID_FirstThunk]
              add esi, edi
              call IsBadReadPtr, esi, 4
              test eax, eax
              jnz bad_exit4proc
              mov eax, [esi]
              test eax, eax
              jz bad_exit4proc
              mov edi, oldproc

next_thunk:
              cmp eax, edi
              jnz next_thunk__
              call VirtualProtect, esi, 1000h, 4, offset dummy, esi
              pop esi

              mov eax, hook_proc
              mov [esi], eax; крок 5: замінюємо адрес на свій
              clc
next_thunk__:
              mov eax, [esi +4]; перевіримо TD_Function
                               ; наступного блоку на 0
              add esi, 4
              test eax, eax
              jnz next_thunk
              db 0c3h; ret
hook_api endp
```

2.4. Сплайсінг функцій ядра Windows 9X

У Win9x, на відміну від WinNT/Win2k, пам’ять, починаючи з 2 Гбайт, спроектована на всі контексти, присутні в системі, тобто її вміст однаковий для всіх процесів. У цьому регіоні пам’яті знаходяться модулі ядра, об’єкти ядра і призначені для користувача об’єкти, доступні всім контекстам – проекції файлів. Документований Windows API не надає засобів для внесення змін в пам’ять 2 Гбайт. VirtualProtect + WriteProcessMemory завершується невдало. Проте цю проблему можна вирішити й без написання власного VxD. Достатньо викликати VxDCall0 _PageModifyPermissions. VxDCall0 – це завжди перша експортована функція Kernel32.dll. Нижченаведений код дозволить читати/записувати першу сторінку kernel32:

```
    call GetModuleHandleA, offset kernel32
   mov ebx, eax
   mov eax, [ebx.MZ_lfanew]
   lea edi, [eax.ebx]
   mov esi, [edi.NT_OptionalHeader. \
                  OH_DirectoryEntries.DE_Export. \
                  DD_VirtualAddress]
   mov esi, [esi.ebx.ED_AddressOfFunctions]
   mov ecx, [esi.ebx]
   add ecx, ebx; ecx == VxDCall0
   shr ebx, 12
   push 020060000h
   push 00h
   push 01h
   push ebx
   push 001000dh; _PageModifyPermissions
   call ecx
```

Треба враховувати проектованість пам’яті 2 Гбайт, адже якщо певний регіон буде дозволений для запису, а потім його пам’ять буде змінена, то ці зміни відбудуться одночасно у всіх контекстах. Таким чином, якщо вдасться змінити ядро для передачі управління деяких функцій іншим обробником, розташованим в пам’яті 2 Гбайт (пам’ять користувальницького процесу), то при виклику функції, що перехоплюється процесом іншого контексту відбудеться збій – за вказаною адресою не виявиться відповідного коду обробника. Отже, код глобальних обробників функцій ядра необхідно мати у своєму розпорядженні також в пам’яті 2 Гбайт . Код в пам’яті 2 Гбайт можна розташувати кількома способами. Якщо розмір коду невеликий, то його можна вмістити в міжсекційних простір будь-якого модуля ядра (розмір коду не більше різниці між фізичною та віртуальною довжиною секції PE модуля). Наступний код намагається знайти місце в регіоні, зайнятому kernel32, що задовольняє цій вимозі і, в разі успіху, копіює в знайдений простір код обробника:

```
            call GetModuleHandle, offset kernel32
            mov ebx, eax
            mov eax, [ebx.MZ_lfanew]
            movzx ecx, word ptr [eax.ebx.NT_FileHeader. \
                              FH_NumberOfSections]
            lea esi, [eax.ebx + SIZE IMAGE_NT_HEADERS]
try_next_section:
            mov eax, [esi.SH_Characteristics]
            and eax, IMAGE_SCN_MEM_WRITE \
                              + IMAGE_SCN_MEM_READ \
                              + IMAGE_SCN_CNT_INITIALIZED_DATA
            cmp eax, IMAGE_SCN_MEM_WRITE \
                              + IMAGE_SCN_MEM_READ \
                              + IMAGE_SCN_CNT_INITIALIZED_DATA
            jne next_section
            mov eax, [esi.SH_SizeOfRawData]
            mov edi, [esi.SH_VirtualSize]
```

```
            sub eax, edi
            cmp eax, CODE_SIZE
            jb next_section
            add edi, [esi.SH_VirtualAddress]
            add edi, ebx
            jmp copy_code
next_section:
            add esi, IMAGE_SIZEOF_SECTION_HEADER
            loop try_next_section
            jmp section_not_found
copy_code:
            mov esi, offset IMPLANT_CODE
            mov ecx, CODE_SIZE
            cld
            rep movsb; скопіювати код в знайдений простір
```

Перевагою цього методу є те, що процес, який здійснив запис у міжсекційний простір, ніяк не пов'язаний з цією пам'яттю. Він може бути закритий, і пам'ять не буде звільнена.

Однак найчастіше вільної пам'яті, знайденої таким шляхом, не вистачає. У такому випадку можна "розкидати" частини свого обробника по всіх міжсекційних просторах модулів, що завантажені вище 2 Гбайт, а потім "склеювати" частини обробника на льоту. Цей метод використовується, наприклад, у вірусі Win9X.cih (він розподіляє свій код в міжсекційних просторах зараженого модуля – при цьому фізичний розмір модуля на диску не змінюється). Таким способом можна розміщувати обробники розміром не більше 10 Кбайт. Зрозуміло, що будь-який серйозний проект буде перевищувати цю межу.

Існує більш простий і надійний метод виділення пам'яті більше 2 Гбайт. Можна виділити пам'ять під об'єкт – проекцію файлу, наприклад, з своп-файлу. Вміст подібного об'єкта буде розташовано системою більше 2 Гбайт, всі виділені сторінки можуть бути помічені атрибутом "виконуються":

```
  call CreateFileMappingA, 0ffffffffh, NULL, \
                  PAGE_READWRITE, 0, IMPLANT_SIZE, 0
  call MapViewOfFile, eax, FILE_MAP_WRITE + \
                  SECTION_MAP_EXECUTE, 0,0, IMPLANT_SIZE
  mov edi, eax
  mov esi, offset IMPLANT
  mov ecx, IMPLANT_SIZE
  cld
  rep movsb
```

За допомогою цього методу можна виділити пам'ять будь-якого розміру, але вона буде мати господаря – процес, який виділив її. Тому якщо процес, який встановив глобальний оброблювач в пам'ять 2 Гбайт завершиться, то пам'ять, їм виділена, звільниться; будь-який виклик у ядрі, що приводить до передачі керування на цю пам'ять в кращому випадку призведе до аварійного завершення процесу, чий потік зробив "незаконну" дію, а в гіршому – до краху

системи. Отже, глобальний перехоплювач функція ядра не повинен завершитися.

Загальним недоліком цих методів є невизначеність базової адреси коду, що копіюється. У будь-якому випадку, очевидно, що базову адресу, вказану лінкувальником при компіляції модуля-перехоплювача, не співпаде з його справжньою базою в пам'яті. Також виникає питання про виклики API функцій таким обробником – таблиця імпорту відсутня.

Найпростіше рішення першої проблеми полягає в застосуванні техніки базонезалежного коду. Ідея в тому, що в самому коді зберігаються лише відносні адреси даних. При ініціалізації коду він визначає свою базу і, додаючи її до відносних адрес, обчислює абсолютні адреси даних.

Приклад базонезалежного коду:

```
            call delta
delta: pop ebp
            sub ebp, offset delta-code_start
            lea esi, [ebp + (offset _data-code_start)]
            ...
_data db 'Text', 0
```

Розглянемо рішення іншої проблеми відносно функцій ядра. Так як модулі ядра присутні у всіх контекстах з однією й тією ж адресою, то для викликів їх функцій з обробників досить побудувати в них перехідники, а необхідні адреси функцій у них будуть записуватися ще до копіювання коду обробника в регіон більше 2 Гбайт. Типовий код перехідника і коду, що його заповнює, має вигляд:

```
 ; Код в секції коду процесу-перехоплювача
            call GetModuleHandleA, offset user32
            call GetProcAddress, eax, offset msgboxa
            mov msgboxa_, eax
            ...
 ; Код в секції даних процесу-перехоплювача:
 ; Він буде скопійований в пам'ять більше 2Gb
_MessageBoxA: Mov eax, 12345678h
            org $ -4
msgboxa_ dd 0
            jmp eax
```

Використовуючи вищенаведену техніку, виклик функцій ядра стає тривіальним і буде виглядати таким чином:

```
   mov eax, [ebp + (offset _uType-code_start)]
   push eax
   lea eax, [ebp + (offset _caption-code_start)]
   push eax
   lea eax, [ebp + (offset _text-code_start)]
   push eax
   mov eax, [ebp + (offset _hWnd-code_start)]
   push eax
   call _MessageBoxA
```

Але як бути у випадку, коли необхідно викликати функцію з бібліотеки, що розташована нижче 2 Гбайт? У різних контекстах вона може бути завантажена (якщо завантажена) в результаті колізій за різними адресами – отже, її адресу не можна заздалегідь записати в перехідник. У цьому випадку спочатку потрібно визначити адресу бібліотеки в поточному контексті (якщо бібліотека туди не завантажена, завантажити її), а потім знайти адресу потрібної функції.

Для надійності не слід використовувати функції LoadLibrary *, тому що якщо бібліотека вже перебувала в контексті, то ця операція збільшує її лічильник використання. Якщо виявиться так, що ця dll була завантажена в контекст єдиний раз, то процес-господар, виконавши FreeLibrary, мав би вивантажити її, але в результаті непередбаченого виклику LoadLibrary * FreeLibrary лише декрементує її лічильник, і може порушитися логіка виконання програми-господаря. Таким чином, для більш “невидимого” втручання в чужий процес необхідно спочатку перевірити присутність потрібного модуля функцією kernel32! GetModuleHandle, і лише у разі його відсутності скористатися LoadLibrary * (після виконання функцій dll слід вивантажити за допомогою FreeLibrary з тих же причин).

```
lea ebx, [ebp + (offset ws2_32-offset code_start)]
call _GetModuleHandleA, ebx
push ebp
xor ebp, ebp
or eax, eax
jnz dllloaded
call _LoadLibrary, ebx
dllloaded: lea ecx, [ebp + (offset clsock-offset code_start)]
call _GetProcAddress, ebx, ecx
mov ecx, [ebp + (offset hSocket-offset code_start)]
call eax, ecx
call _FreeLibrary, ebp; пройде успішно, якщо
pop ebp; dll завантажили
```

Отже, будь-яка пам’ять доступна нам для читання/запису, код може коректно виконуватися незалежно від його бази, і викликати будь-які API функції. Тепер можна приступити до виправлення коду функцій ядра для передачі управління обробнику.

Ebp вказує на початок знайденої пам’яті. Для початку збережемо перших п’яти байт функції, що перехоплюється:

```
call GetProcAddress, ebx, offset lla
mov edi, eax
lea esi, [ebp + (offset lla_code-offset code_start)]
push edi
xchg esi, edi
movsb
movsd
```

Потім змінимо прапор дозволу запису атрибуту сторінки, яка містить початок функції, на істину:

```
   mov eax, [esp]
   pushad
   shr eax, 12
   push 020060000h
   push 00h
   push 01h
   push eax
   push 001000dh; _PageModifyPermissions
   mov eax, [vxdcall0]
   call eax
   popad
   pop edi
```

Побудуємо 5-байтовий JMP на початку функції:

```
   mov al, 0e9h
   stosb
   stosd; додамо 4 до edi

   lea eax, [ebp + (offset lla_entry-offset code_start)]
   sub eax, edi
   mov [edi-4], eax
```

При такому методі організації сплайсінгу обробник перехопленої функції може виглядати так (синтаксис функції – перше подвійне слово):

```
lla_entry: call swap_lla
             push dword ptr [esp +4]
             call _lla; виклик цієї функції
             call swap_lla
             ... ; якісь дії
             ret 4

_lla: mov eax, 12345678h
             org $ -4
lla_ dd 0
             jmp eax

swap_lla: push esi edi ebp
             call delta
delta: pop ebp
             sub ebp, offset delta-code_start

             lea esi, [ebp + (offset lla_code-code_start)]
             mov edi, [ebp + (offset lla_-code_start)]
             push eax
             mov eax, [edi]
             xchg [esi], eax
             mov [edi], eax
             mov al, [edi +4]
             xchg [esi +4], al
             mov [edi +4], al
             pop ebp edi esi
             ret
```

3.5. Методи встановлення глобальних перехоплювачів

Часто буває необхідно перехопити будь-яку функцію глобально, тобто у всіх поточних процесах разом. Наприклад, для досягнення невидимості в NT/w2k через функцію ntdll! NtQuerySystemInformation процес-привид повинен перебрати всі інші процеси і поширити свій обробник на кожен з них. Крім цього, процес повинен подбати і про нові процеси – ті, які будуть створені після інсталяції його обробника NtQuerySystemInformation. Це нетривіальне завдання, тому що процес може бути запущений безліччю способів: shell32! ShellExecute *, kernel32! CreateProcess *, ntdll! NtCreateProcess та іншими. Більшість з них зводиться до NtCreateProcess. Також виникає проблема – процес з привілеями гостя не може змінити пам'ять контексту, що працює з привілеями System або Local Administrator – наприклад, Winlogon.exe, а саме він запускає найпопулярніший менеджер завдань для NT/w2k/XP: taskmgr.exe (за допомогою функції msgina! WlxStartApplication). Таким чином, на Winlogon.exe обробник NtCreateProcess поширити не вдасться, і при натисканні користувачем ctrl-shift-esc запущений taskmgr "побачить" захований процес.

Ця проблема залишається нерозв'язною. Проте ряд використовує той факт, що taskmgr – GUI додаток, а отже, встановивши глобальний хук повідомлень – наприклад, типу WH_CBT – модуль автоматично буде завантажено до taskmgr.exe і буде викликаним першим з причиною DLL_PROCESS_ATTACH ще до того, як основний модуль taskmgr отримає управління.

Перша проблема може бути вирішена наступним чином. Перехопимо ntdll! NtCreateThread і ntdll! CsrClientCallServer. Процес не має свого ідентифікатора до створення першого потоку, і якщо в обробнику NtCreateThread є така ситуація, то:

- процес-господар не має ідентифікатора;
- створений потік призупинено;
- після створення потоку процес отримує ідентифікатор, що означає створення нового процесу. У цьому випадку запам'ятовуємо його ідентифікатор і чекаємо виклику CsrClientCallServer з командою 10000h – ініціалізації процесу. Потім перевіряємо, чи був раніше збережений ідентифікатор. Якщо так, то після виклику цього CsrClientCallServer процес готовий до застосування до нього обробників.

Код обробників NtCreateThread і CsrClientCallServer може виглядати так:

```
myNtCreateThread proc lpThreadHandle, DesiredAccess, \
              lpObjectAttributes, ProcessHandle, lpClientId, \
              lpInitialContext, lpUserStackDescriptor, \
              CreateSuspended
            mov eax, pbi2
            and [eax.UniqueProcessId], 0
            call NtQueryInformationProcess, ProcessHandle, \
                 ProcessBasicInformation, eax, pbisize, NULL
            push eax
            call [realNtCreateThread], lpThreadHandle, \
```

```
                    DesiredAccess, lpObjectAttributes, \
                    ProcessHandle, lpClientId, lpInitialContext, \
                    lpUserStackDescriptor, CreateSuspended
            pop ecx
            pop eax
            or ecx, ecx
            jl nctexit
            test eax, eax
            jl nctexit
            cmp CreateSuspended, FALSE
            je nctexit
            mov eax, pbi
            cmp [eax.UniqueProcessId], 0
            jne nctexit
            mov eax, pbi2
            call NtQueryInformationProcess, ProcessHandle, \
               ProcessBasicInformation, eax, pbisize, NULL
nctexit: pop eax
            ret
myNtCreateThread endp

myCsrClientCallServer proc lpStruc, Par1, dwCommand, StrucSize

            call [realCsrClientCallServer], lpStruc, Par1, \
                            dwCommand, StrucSize
            cmp dwCommand, 10000h
            jne cccsexit
            mov edx, lpStruc
            cmp dword ptr [edx +20 h], 0
            jl cccsexit
            mov eax, pbi2
            mov ecx, [eax.UniqueProcessId]
            jecxz cccsexit
            pushad
            ... ; установка обробників
            popad
cccsexit: ret
myCsrClientCallServer endp
```

При використанні методу виправлення таблиць імпорту в w9x можна вчинити аналогічно, перехопивши kernel32! GetStartupInfoA.

При використанні сплайсінгу модулів вище 2 Гбайт проблема вирішується сама собою: ця пам’ять проектується на всі контексти, отже, виклики функцій з будь-якого процесу будуть перехоплені.

3.6. Призупинення потоків

Розглянемо метод зміни таблиці імпорту модуля з метою перехоплення функцій, що імпортується.

При ініціалізації бібліотеки (виклик DllMain з причиною DLL_PROCESS_ATTACH), як було обумовлено раніше, відбувається виконання коду-інсталятора обробників. Кінцевою його метою є заміщення всіх

входів таблиці імпорту модуля, що містять адреси функцій, які перехоплюються на адреси відповідних обробників. Якщо необхідно перехоплювати функції з усіх модулів процесу, то ці дії будуть виконуватися "не моментально". Виникає проблема синхронізації: у момент перебору модулів процесу оброблені модулі будуть містити одні адреси функцій, що перехоплюються, а необроблені – інші. Це може позначитися на логіці виконання процесу.

Щоб уникнути цього, необхідно припиняти потоки перед маніпуляціями з таблиці імпорту, а після їх завершення знову "запускати" потоки. Існують документовані функції kernel32 SuspendThread (HANDLE hThread) і ResumeThread (HANDLE hThread), які дозволяють зупиняти і запускати потоки (вірніше, їх код в user mode). Ці функції оперують з лічильником зупинок потоку (suspend count). Функція SuspendThread збільшує цей лічильник. Якщо його значення більше нуля, то потік зупинений, якщо значення перевищує допустимий (MAXIMUM_SUSPEND_COUNT) – SuspendThread повертає помилку (ERROR_SIGNAL_REFUSED) та інкрементування не проводиться. ResumeThread декрементує лічильник зупинок; якщо він досяг нуля, потік знову стає запланованим. Для роботи обох функцій необхідний описувач (handle) потоку, відкритий з прапором THREAD_SUSPEND_RESUME.

Проблема полягає в тому, що документований API Win32 надає описувач потоку тільки в декількох випадках: при створенні потоку функцією kernel32! CreateThread, при налагодженні процесу як повідомлення відладчику (повертається при старті налагодження або після створення потоку налагоджувати додатком через kernel32! WaitForDebugEvent), після дуплікування наявного описувача і при виклику GetCurrentThread.

Незважаючи на це, всі Win32 системи дозволяють перераховувати всі ідентифікатори потоків (аналогічно ідентифікаторів процесів). У SDK пропонується використовувати ідентифікатори потоків для складання запитів процесу-творцеві потоку з метою отримання описувача. За SDK, якщо процес не підтримує віддалене маніпулювання його потоками, то ідентифікатор потоку взагалі втрачає сенс.

Однак, у всіх Win32 системах існує спосіб отримання описувача потоку за його ідентифікатором. Достатньою умовою отримання описувача потоку з прапором THREAD_SUSPEND_RESUME є можливість відкрити процес з прапором PROCESS_ALL_ACCESS.

3.6.1. Отримання описувача потоку за його ідентифікатором в Windows 9X

При уважному вивченні коду функції OpenProcess з'ясовується, що вона зводиться до більш потужної функції, в якості параметрів до якої передаються прапори, прапор вкладеності і деяке значення, отримане з ідентифікатора процесу шляхом операції XOR його з якимсь подвійним словом. Результатом цієї операції є адреса структури, яка описує процес (Process Data Block).

Далі йде перевірка – чи описує структура процес за адресою, отриману від покажчика. Якщо це не процес, то відбувається вихід з OpenProcess. Тому виникає підозра, що ідентифікатори потоку і процесу мало чим відрізняються за своєю сутністю.

Справді, операція XOR ідентифікатора потоку з “незрозумілим” подвійним словом дає адреса TDB, – таким чином, для перекладу ідентифікатора потоку в його описувач достатньо “вручну” робити XOR ідентифікатора потоку з тим подвійним словом, розміщувати результат у eax і викликати [OpenProcess 24 h]. Так як код OpenProcess не змінився від Win95 до Win98, то зсув збережеться для обох систем.

```
OpenProcess proc near

dwFlags = dword ptr 4
inheritance = dword ptr 8
pid = dword ptr 0Ch

              push [esp + pid]
              call xorbyobsfucator
              test eax, eax
              jnz short pidconverted
              xor eax, eax
              jmp short bad_exit

pidconverted: cmp byte ptr [eax], 6; (1)
              jz short OpenThread
              push 57h
              call sub_BFF7C991
              mov ecx, 0FFFFFFFFh
              jmp short loc_BFF95CA1
OpenThread:
              mov ecx, 0; (2)
              mov edx, [esp + dwFlags]
              cmp [esp + inheritance], 1
              adc ecx, 0FFFFFFFFh
              and edx, 1F0FFFh
              and ecx, 80000000h
              or ecx, edx
              mov edx, dword_BFFC9CDC
              push ecx
              push eax
              push dword ptr [edx]
              call SomePowerfulFunction
              ...
```

Залишається тільки довідатися про “magic dword”, який використовується для отримання адреси PDB. Це можна зробити кількома способами. Самий безпечний і швидкий – це скористатися відомим для поточного процесу PDB, адреса якого знаходиться за адресою fs: [30h]. Потрібно використовувати XOR на ньому і значення, що повертається GetCurrentProcessId. Тоді можливий код функції w9x_OpenThread буде виглядати наступним чином (передбачається використання TASM як компілятора, тому що нижче проводиться читання

адреси OpenProcess з таблиці стрибків (jump table), що генерується TASM за умовчанням; MASM генерує код іншого характеру – виклики функцій API виробляються шляхом виконання call [x], де x – адреса входу необхідної функції в таблиці імпорту модуля):

```
w9x_OpenThread proc flags, inheritance, tid: dword
             local w9xopenthread: dword

             pushad
             call GetCurrentProcessId
             xor eax, fs: 30h
             mov ebx, eax
             lea esi, OpenProcess 2; jmp far [xxxx]
             lodsd; xxxx
             xchg eax, esi
             lodsd; [xxxx] = Kernel32! OpenProcess
             lea esi, [eax +24 h]
             lodsd; [OpenProcess +24 h]
             mov edi, esi
             cmp eax, 0b9h; mov ecx, dword ptr 0
             jnz bad_exit
             sub edi, 4
             mov w9xopenthread, edi
             xor ebx, tid
             lea esi, [ebx +2]
             call IsBadWritePtr, esi, 2
             or eax, eax
             jnz bad_exit
             xchg eax, ebx
             mov eax, w9xopenthread
             call eax, flags, inheritance, tid
             mov [esp.Pushad_eax], eax
             popad
             ret
bad_exit: popad
             call SetLastError, ERROR_ACCESS_DENIED
             xor eax, eax
             ret
w9x_OpenThread endp
```

3.6.2. Отримання описувача потоку за його ідентифікатором в Windows NT

У Windows NT (починаючи з версії 3.51) присутня недокументована функція ntdll! NtOpenThread, що дозволяє отримувати описувач потоку за ідентифікатором процесу – господаря й ідентифікатором самого потоку. Функція перевіряє привілеї потоку, що викликав її, тому що вона не загрожує стабільності системи. Код функції перебуває цілком у ntoskrnl.exe, що розташовується вище 2 Гбайт і виконуються у режимі ядра, тому звичайний процес не може змінити логіку виконання цієї функції. Її прототип кілька нестандартний для WINAPI (це перехідник на сервіс ntoskrnl.exe, отже, прототип – NTKERNELAPI):

```
NTKERNELAPI
NTSTATUS
NtOpenThread (
OUT PHANDLE ThreadHandle,
IN ACCESS_MASK DesiredAccess,
IN POBJECT_ATTRIBUTES ObjectAttributes,
IN PCLIENT_ID ClientId OPTIONAL
);   де

DWORD ClientId [2];
ClientId [0] = TargetPID;
ClientId [1] = TargetTID;
```

Структура Object_Attributes повинна розташовуватися в пам'яті за адресою 1000h. В іншому випадку функція поверне помилку – STATUS_DATATYPE_MISALIGNMENT. Для виділення пам'яті з початковим адресою, що задовольняє такій умові, можна скористатися стандартною функцією kernel32! VirtualAlloc.

Відмінною рисою цієї функції від її аналога в w9x є необхідність вказівки ідентифікатора процесу, але в більшості випадків використання цієї функції потрібно повністю припинити роботу якого-небудь процесу (може, за винятком якогось потоку), а при перерахуванні потоків потрібно знати ідентифікатор процесу, так що вказана особливість не стає великою перешкодою у використанні NtOpenThread. Додатковий код доведеться писати лише в тому випадку, коли необхідно отримати описувач потоку без початкових даних про його приналежність. Цю інформацію можна отримати за допомогою NtQuerySystemInformation.

Можливий код функції nt_OpenThread може мати вигляд:

```
nt_OpenThread proc flags, inheritance, tid, pid: DWORD
             local thandle: DWORD
             local pntot: dword
             local _tid: dword
             local _pid: dword

             mov eax, tid
             mov _tid, eax
             mov eax, pid
             mov _pid, eax

             call GetModuleHandleA, offset ntdll
             call GetProcAddress, eax, offset ntopenthread
             or eax, eax
             jz baderror
             mov pntot, eax

             call VirtualAlloc, 0,100,1000 h, 40h
             mov ebx, eax
             xchg eax, edi
             push 18h
             pop eax
             stosd
```

```
            xor eax, eax
            push 5
            pop ecx
            rep stosd

            lea ecx, thandle
            lea edx, _pid
            and dword ptr [ecx], 0
            call [pntot], ecx, 1f0000h, ebx, edx
            mov eax, thandle
            call VirtualFree, ebx, 0,8000 h, eax
            call GetCurrentProcess
            pop ebx
            lea ecx, thandle
            and dword ptr [ecx], 0
            call DuplicateHandle, eax, ebx, eax, \
                              ecx, 1f0fffh, inheritance, \
                              DUPLICATE_CLOSE_SOURCE
            mov eax, thandle
            ret
baderror: xor eax, eax
            ret
nt_OpenThread endp
```

У kernel32 ОС Windows ME/2000/XP присутня документована функція OpenThread, використання якої робить завдання тривіальним. Синтаксис функції такий (він практично дублює синтаксис OpenProcess):

```
HANDLE OpenThread (
DWORD dwDesiredAccess, / / access right
BOOL bInheritHandle, / / handle inheritance option
DWORD dwThreadId / / thread identifier
);
```

3.7. Особливості Native API

Як відомо, Windows 2000, була спроектована таким чином, що б дати можливість виконуватися не тільки додаткам Win32, але так само додаткам Win16, MsDos, Os/2v 1.x і POSIX. ОС має три вбудованих за замовчуванням підсистеми, кожна з яких виконує свої образи (рис. 3.1). Реально, ніхто не заважає додати в систему підтримку додатків інших операційних систем, всього лише потрібно написати та інтегрувати модуль необхідної підсистеми.

Якщо програма, що запускається не є додатком Win32, а є програмою іншої ОС, тоді Win2k буде шукати для неї відповідний спосіб підтримки, що зрозуміє і запустить дану програму. Якщо не знайде, то поверне помилку відповідно. Будь-який виклик, наприклад, програми OS/2 буде конвертований модулем підтримки даного додатку у виклик відповідної функції рідної бібліотеки Win32-підсистеми Kernel32.dll, яка у свою чергу направить виклик одній з функції модуля NTDLL.DLL.

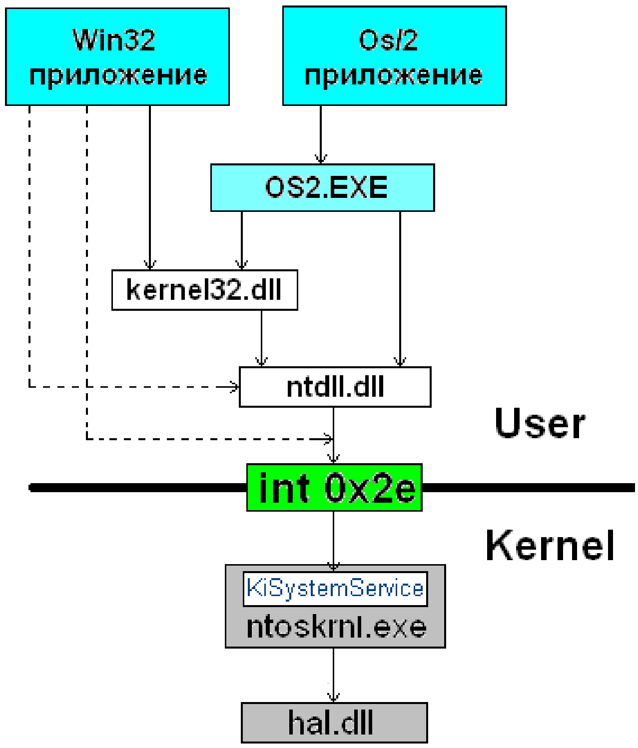

Рис 3.1. Підсистеми ОС

Хоча, наприклад, ніщо не заважає додатку Win32 напряму звертатися до функцій модуля ntdll.dll або навіть минаючи останній, безпосередньо до шлюзу int 0x2e.

Так, це має бути дуже специфічна програма і звичайно вона повинна знати, що робить. Проте, програми в WinNT нерідко називаються вони ми, точніше, додатками. Прикладом може служити відомий Svchost.exe. Такі програми мають повне право ... WinNT. У Win9x вони працювати не можуть. Розглянемо більш детально таблицю ... (рис 3.2).

Крім даної структури, що одночасно експортується модулем Ntoskrnl.exe в WinNT є ще одна, що називається ServiceDescriptorTableShadow, яка не експортується ... модуля.

Якщо ... функції не експортуються, то це не означає, що не можна їх використати. Існує декілька простих способів отримати доступ до них. Необхідно скористатися одним з ..., що входять в комплект DDK – Kernel Debugger (i386kd.exe), WinDbg (windbg.exe) або SoftIce, але, попередньо встановивши налагоджувальні символи, для дизасемблювання та аналізу бінарників краще використати IDA, яка при завантаженні ... (меню Edit -> Plugins-> Load PDB File або Ctrl + F12) без усяких ... з файлом .PDB.

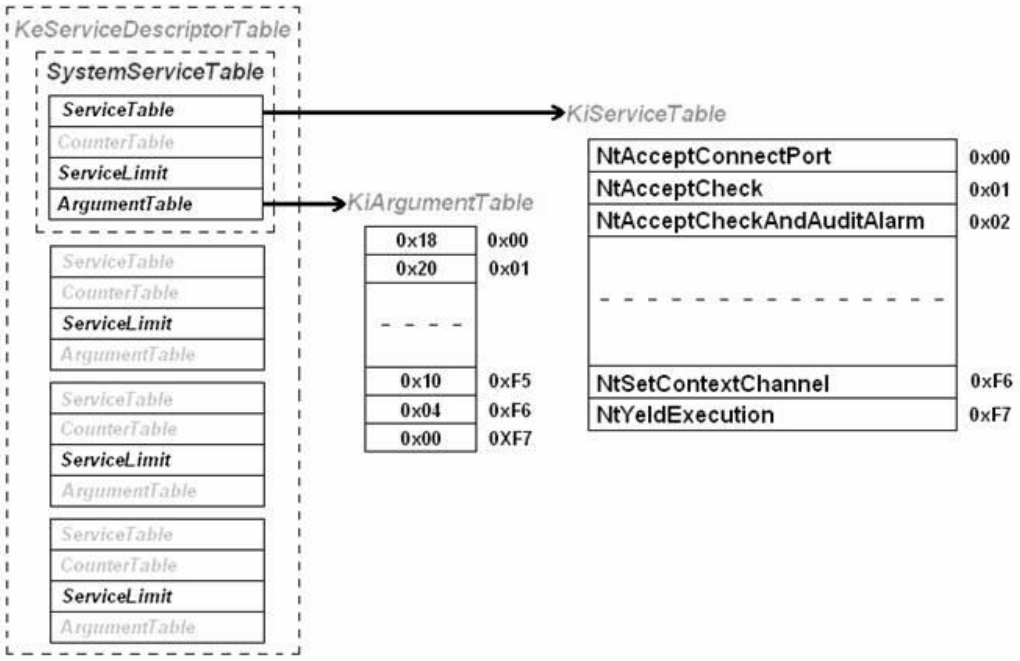

Рис. 3.2. Сервіси та аргументи ОС Win9x

На рис. 3.2 поля виділені сірим кольором містять нульові покажчики або нулі, а на мові С це має такий вигляд:

```
typedef struct _SERVICE_DESCRIPTOR_TABLE
(
SYSTEM_SERVICE_TABLE NtoskrnlTable; / / ntoskrnl.exe (native api)
SYSTEM_SERVICE_TABLE Table2; / / вільна
SYSTEM_SERVICE_TABLE Table3; / / використовується Internet Information
Services
SYSTEM_SERVICE_TABLE Table4; / / вільна
)
        SERVICE_DESCRIPTOR_TABLE,
     * PSERVICE_DESCRIPTOR_TABLE,
    ** PPSERVICE_DESCRIPTOR_TABLE;

typedef struct _SYSTEM_SERVICE_TABLE
 (
 PNTPROC ServiceTable; / / вказівник на масив точок входу в
                        / /обробники
 PDWORD CounterTable; / / лічильник викликів сервісів
                      / / (не використовується)
 DWORD ServiceLimit; / / кількість підтримуваних сервісів
 PBYTE ArgumentTable; / / вказівник на масив аргументів сервісів
)
        SYSTEM_SERVICE_TABLE,
     * PSYSTEM_SERVICE_TABLE,
    ** PPSYSTEM_SERVICE_TABLE;
```

У Windows 2000 знайти її можна безпосередньо за структурою ServiceDescriptorTable. Однак не всі версії Windows NT слідують цим правилом. Цікава дана структура тим, що, окрім поля NtoskrnlTable, друге поле має не

нульові значення, а містить покажчик на структуру SystemServiceTable, посилається на масив точок входу функцій GDI, що надаються драйвером Win32k.sys. Як відомо, у WinNT, графіка “прихована” в ядрі, для прискорення відповідних процесів. Але, в більшості випадків, на цей факт мало звертають увагу, оскільки перехоплення графічних функцій знову ж таки мало кому цікаве.

Тепер детальніше розповімо про дані структури. Структура KeServiceDescriptorTable включає в себе чотири абсолютно однакових структури SystemServiceTable. Значення має тільки перша з них, яку умовно називають NtoskrnlTable. Структура SystemServiceTable знову ж таки в свою чергу складається з чотирьох полів. Розглянемо їх більш детально:

- *ServiceTable* – вказівник на масив покажчиків на точки входу в обробники системних сервісів. Де одні точки входу сервісів вказують на реально існуючі функції, що експортуються модулем Ntoskrnl.exe, а інші – ні. Наприклад, функція NtCreateProcess не експортується, а точка входу, тим паче, вказує на деякий код, який і реалізує останню. Тому, виникають складнощі з реалізацією цієї функції всередині, наприклад, драйвера, оскільки не маємо можливості ні скористатися готовим експортом, ні простим універсальним інтерфейсом, окрім знову ж таки того, що надається функцією KiSystemService. Але, NtCreateProcess – це лише вершина айсберга, більша частина підготовчих операцій, пов’язаних зі створенням процесу заховані всередині модулів Kernel32.dll і Ntdll.dll. І це не єдина функція подібного роду. Так що, якщо необхідно реалізувати подібну операцію всередині драйвера, доведеться спочатку розібратися з тим, як це робить WinNT. Завдання це потребує серйозних знань, оскільки що код може бути сильно розмазаний по різним модулям, а не локалізований десь всередині певної третьої функції. Проте у Windows 9x операція зі створення процесу була чітко визначена в режимі ядра і виражена певною функцією, що “експортується” драйвером Shell.vxd – це ShellExecute;
- *CounterTable* – покажчик на змінну ядра, яка використовується як лічильник кількості звернень до KiSystemService. Має значення тільки в налагоджувальній версії. У вільній версії він завжди дорівнює NULL;
- *ServiceLimit* – кількість функцій-сервісів, що реалізовуються даною версією Windows NT. Наприклад в Windows 2000 build 2195 це число 248 (0xF8), а в Windows XP build 2600 вже 284 (0x11C). Як видно, їх кількість зростає;
- *ArgumentTable* – вказівник на масив байтів, що вказує на кількість аргументів, що передаються функції в байтах. Тобто, якщо функція NtAcceptConnectPort приймає на стек 0x18 байт, виходить шість подвійних слів. Індекс подвійного слова в масиві ServiceTable зіставляється елементу з тим же індексом в таблиці ArgumentTable.

В структурі ServiceDescriptorTable при самому найкращому розкладі зайнятими можуть бути тільки дві структури SystemServiceTable (крім Ntoskrnl.exe, друга може бути зайнята сервісами IIS). Тобто, залишається ще дві

структури, які можуть бути віддані користувачу для додавання в систему власних сервісів, шляхом виклику функції KeAddSystemServiceTable, описаної в DDK.

Код _KiSystemService буде мати вигляд:

```
; Завантажуємо в CL відповідний елемент масиву ArgumentTable,
; Для того, щоб точно таке ж число байт перенести в стек ядра
. text: 00465070 xor ecx, ecx
. text: 00465072 mov cl, [eax + ebx]
. text: 00465075 mov edi, [edi]
; Завантажуємо в регістр EBX точку входу у функцію відповідного
; сервісу
. text: 00465077 mov ebx, [edi + eax * 4]
. text: 0046507A sub esp, ecx
; Передача подвійними словами
. text: 0046507C shr ecx, 2
. text: 0046507F mov edi, esp
; Перевіряємо, чи знаходиться користувальницький стек в
; призначеному для користувача адресному просторі?
. text: 00465081 cmp esi, MmUserProbeAddress; = 0x7FFFFFFF
. text: 00465087 jnb loc_465271
. text: 0046508D
. text: 0046508D loc_46508D:
. text: 0046508D
; Передаємо параметри в стек функції, що викликається
. text: 0046508D repe movsd
; коли всі налаштування та перевірки позаду, викликаємо сам сервіс
. text: 0046508F call ebx
; Пам'ятайте, регістр EDX у функції
; NtDll! ZwWriteFile? Підказка: його вміст залишився незмінним
; і це означає, що ним ще скористаємося.
```

3.8. Перехоплення функцій NtOpenFile і NtCreateFile

Розглянемо драйвер, який реалізує перехоплення двох досить цікавих функцій: NtOpenFile і NtCreateFile. Вони цікаві тим, що, перевівши все це мовою об'єктної абстракції, виходить, що це не просто якісь функції, а методи, що застосовуються до більшості об'єктів системи. Таким чином, методи застосовуються до тих об'єктів системи, над даними яких вони можуть виконуватися. А це – файли, пристрої тощо. Генеруємо шаблон, вказавши шлях і слово mfilemon як ім'я проекту. Необхідно правильно ініціалізувати деякі поля у файлі w2k_wiz.ini, інакше компілятор може ігнорувати драйвер. Оригінальний текст складається з декількох модулів, найбільш цікавий, де, власне, і розташовується основний код – fmonitor.c.

Модуль "mfilemon.c"

Алгоритм роботи драйвера наступний: спочатку перевіряємо версію ядра, чи можна працювати:

```
if (* NtBuildNumber == 2195); else / / 2k
if (* NtBuildNumber == 2600); else / / XP
```

```
if (* NtBuildNumber == 3790); else / / 2003
```

При ініціалізації (DriverInitialize ()) викликаємо функцію IoRegisterShutdownNotification (gpDeviceObject) для того, щоб зареєструвати callback-функцію (DriverShutdown), яку система викличе безпосередньо перед перезавантаженням:

```
IoRegisterShutdownNotification (gpDeviceObject); / / реєструємо
                              / /подію завершення роботи системи
if (! SetFMonHandler ()) / / встановлюємо наші перехоплювачі
(
    IoDeleteDevice (pDeviceObject);
    return ns;
)
SystemGetLocalTime (); / / запитуємо час
)
if    ((ns    =    IoInitializeTimer(pDeviceObject,&TimerProc,NULL))    ==
STATUS_SUCCESS) IoStartTimer(pDeviceObject);
  else IoDeleteDevice (pDeviceObject);
 }
   return ns;
 }
```

Слід за нею, викликом функції SetFMonHandler () створюємо і відкриваємо файл, для запису в нього протоколу перехоплення і одночасно встановлюємо перехоплювачі.

Тепер необхідним є регістр EDX. Через нього _KiSystemService отримує покажчик на користувальницький стек. Оскільки вміст регістра не змінюється протягом всіх перевірок аж до інструкції CALL EBX. Індекси функцій в таблиці SST є різними в різних версіях ядра. Проте, якщо необхідно отримати відповідні дані, то, всього-на-всього варто звернутися за допомогою до відладчика або дизасемблера.

Модуль "fmonitor.c"

Функція SetFMonHandler (). Викликом функції SystemDeleteFile () затираємо логічний файл на диску, якщо той залишився від попереднього запуску драйвера. Потім створюємо чистий файл протоколу, відкриваємо його і зберігаємо описувач. Описувач файлу повинен бути описувачем ядра, оскільки в такому випадку він не буде прив'язаний до конкретного процесу. Потім, залежно від версії ядра правим відповідні елементи в таблиці SDT, шляхом запису в них адрес перехоплювачів, попередньо зберігши системні. Ще один цікавий момент:

```
_asm mov eax, CR0
_asm mov CR0Reg, eax
_asm and eax, 0xFFFEFFFF / / скинути WP bit
_asm mov cr0, eax
```

Це називається захист від модифікації ядерних сторінок. При скинутому біті WP, в режимі ядра можлива модифікація користувальницьких сторінок, що

мають атрибут readonly і системних сторінок пам'яті без будь-якої генерації #GP. Якщо цей біт піднятий, то при записі в системні сторінки пам'яті, що мають атрибут захисту від запису, буде згенеровано виключення #GP, і це в свою чергу призведе до BSOD. Оскільки вважається, що було зроблено спробу зруйнувати системні дані. Все це залежить від того, який обсяг оперативної пам'яті безпосередньо доступний системі. У разі Windows 2000 за наявності оперативної пам'яті 128 Мбайт і вище, ці сторінки мають розмір 4 Мбайт. У Windows XP і Windows 2003 Server бар'єр піднятий вже до 256 Мб. Отже, драйвер (.Sys), та й практично всі виконуючі модулі Windows NT мають формат PE. А, як відомо, секції всередині даного файла вирівняні по межі 4 Кбайт. У разі, якщо розмір системної сторінки пам'яті становить 4 Мбайт, захист вимикається (скидається біт WP регістру CR0), оскільки в дану сторінку необхідно умістити як код, так і дані, до того ж ще й не одного драйвера. А вже дані в окремих випадках не модифікуються.

Функція WriteToLogFile (PVOID Stack). Отже, функція приймає як параметр покажчик на користувальницький стек аргументів, той самий покажчик, що передається в регістрі EDX. Перехоплювач використовує один і той самий код для обох NtCreateFile і NtOpenFile функцій, оскільки структура їх стеків майже повністю ідентична. Код функції вкладений у блок _try _except. Оскільки маємо справу з функціями (RtlUnicodeStringToAnsiString ()), які у свою чергу працюють з пам'яттю, рано чи пізно тут може виникнути виключна ситуація.

Далі розглянемо стеки функцій NtCreateFile і NtOpenFile:

```
NTSYSAPI
NTSTATUS
NTAPI
NtCreateFile (
  OUT PHANDLE FileHandle,
  IN ACCESS_MASK DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN PLARGE_INTEGER AllocationSize OPTIONAL,
  IN ULONG FileAttributes,
  IN ULONG ShareAccess,
  IN ULONG CreateDisposition,
  IN ULONG CreateOptions,
  IN PVOID EaBuffer OPTIONAL,
  IN ULONG EaLength);

NTSYSAPI
NTSTATUS
NTAPI
NtOpenFile (
  OUT PHANDLE FileHandle,
  IN ACCESS_MASK DesiredAccess,
  IN POBJECT_ATTRIBUTES ObjectAttributes,
  OUT PIO_STATUS_BLOCK IoStatusBlock,
  IN ULONG ShareAccess,
  IN ULONG OpenOptions);
```

Як видно, до певних меж, функції однакові інтерес становить тільки структура OBJECT_ATTRIBUTES. Покажчик на дану структуру розташовується в стеку третім за вкладеністю:

```
(PBYTE) Stack = (PBYTE) Stack 8;
pObjectAttributes = * (PDWORD) Stack;
```

Таким чином, покажчик на структуру OBJECT_ATTRIBUTES дає можливість отримати ім'я об'єкта, який буде відкрито.

У кінцевому підсумку роботи драйвера, в корені диска C знаходитиметься файл з ім'ям KiSystemService.log. Зазирнувши в нього звичайним текстовим редактором можна буде побачити приблизно наступне:

```
0 16:38:51 3a4 40c C: \ WINNT \ Explorer.EXE \ Device \ (56A954E7-
 28ED-471A-B406-2936BB2363B3)
1 16:38:52 3a4 40c C: \ WINNT \ Explorer.EXE \ Device \ (56A954E7-
 28ED-471A-B406-2936BB2363B3)
2 16:38:53 3a4 2dc C: \ WINNT \ Explorer.EXE \?? \ C: \
3 16:38:53 3a4 2dc C: \ WINNT \ Explorer.EXE \?? \ C: \ Program
 Files \ desktop.ini
4 16:38:53 3a4 2dc C: \ WINNT \ Explorer.EXE \?? \ C: \ Program
 Files \ desktop.ini
```

Висновки

- Розробка механізмів перехоплення функції API в режимі користувача включає два основні методи: безпосередню модифікацію таблиць імпорту/експорту процесу, шляхом заміни в них покажчиків на оригінальні функції покажчиками на власні функції-заглушки та модифікацію частини коду бібліотек, що містять функції, які відслідковуються (метод сплайсінгу коду).
- В ОС, які підтримують систему привілеїв можливість успішного відкриття процесу залежить від поточного рівня привілеїв процесу, що виконуються. Читання-запис пам'яті є головною проблемою всієї ідеології перехоплень API. Тому існує безліч методів та функцій, що вирішують цю проблему, які необхідно знати кожному розробнику драйверів.

Контрольні запитання та завдання

1. Назвіть основні особливості механізмів перехоплення функцій API в режимі користувача.
2. Назвіть особливості перехоплення викликів API в системах Windows NT і Windows 9X.

3. Що входить до складу наборів функцій інтерфейсу Native API Windows 2000?
4. Назвіть етапи створення і виконання віддаленого коду.
5. Особливості таблиці імпорту.
6. Назвіть особливості структури KeServiceDescriptorTable.
7. Охарактеризуйте функції перехоплення.

Тести для самоконтролю

1. Що таке API?
    a. опис програмної моделі інтерфейсів користувача
    b. набір визначень взаємодії різнотипного програмного забезпечення, метод абстракції між низькорівневим та високорівневим програмним забезпеченням
    c. програмна модель взаємодії прикладного програмного забезпечення
2. Що таке служби Windows?
    a. особливий вид драйверів
    b. процеси режиму користувача, для запуску та функціонування яких реєстрація інтерактивного користувача в системі не потрібна
    c. програми, що забезпечують розширення функціоналу ОС
3. В чому позитивна сторона методу сплайсінгу коду?
    a. не потрібно проводити модифікацію таблиць імпорту модулів процесу. У Win 9X метод дає можливість відстежувати звернення до функцій системних бібліотек з усіх процесів
    b. не потрібно модифікувати частини коду бібліотек, що містять функції, що відслідковуються
    c. не потрібно взагалі відстежувати виклики системних функцій, що нас цікавлять
4. Що виконує функція SetWindowsHookEx?
    a. стає елементарною деінсталяцією слухача – достатньо відновити початкові байти
    b. встановлює покажчик на обробник функції, що виконується
    c. проектує бібліотеку на всі контексти, які задовольняють таким умовам: їх потоки мають в поточний момент GUI - вікно, що приймає повідомлення від користувача, а їх процеси можуть бути успішно відкриті користувачем функцією OpenProcess з параметром PROCESS_ALL_ACCESS
5. Що виконує функція CreateRemoteThread?
    a. створює віддалений потік
    b. створює покажчик на віддалений об’єкт
    c. відкриває чужий процес
6. Що повертає функція ServiceTable?

   a. вказівник на масив покажчиків на точки входу в обробники системних сервісів
   b. вказівник на таблицю сервісів
   c. назву сервісу відповідно до його індексу в таблиці сервісів
7. Що повертає функція CounterTable?
   a. вказівник на таблицю індексів системних служб Windows
   b. масив значень таблиці сервісів
   c. покажчик на змінну ядра, який використовується як лічильник кількості звернень до KiSystemService

Теми для самостійного опрацювання

1. Диспетчер системних викликів. Огляд взаємозв'язків системних модулів у режимі користувача та режиму ядра.
2. Таблиці дескрипторів системних викликів. Обробник системних викликів INT 2EH.
3. Роль асемблера у розробці драйвера слідкування за системними викликами функцій Native API.
4. Процес створення диспетчера перехоплення системних викликів.
5. Типи даних в Win32 API.
6. Особливості програмування в середовищі Win32 та Win64.

Список використаної літератури

1. *Побегайло А. П.* Системное программирование в Windows / Побегайло А. П. – СПб.: БХВ-Петербург, 2006. – 1056 с.
2. *Рихтер Дж.* Windows для профессионалов: создание ефективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Джеффри Рихтер; пер. с англ. – [4-е изд.]. – СПб: Питер; М.: Издательско-торговый дом "Русская Редакція", 2003. – 752 с.
3. *Рудаков П. И.* Язык ассемблера: уроки программирования / П. И. Рудаков, К. Г. Финогенов. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.
4. *Руссинович М.* Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс / М. Руссинович, Д. Соломон; пер. с англ. – [4-е изд.]. – М.: Русская редакция, СПб.: Питер, 2005. – 992 с.
5. *Сорокина С. И.* Программирование драйверов и систем безопасности : учебное пособие / С. И. Сорокина, А. Ю. Тихонов, А. Ю. Щербаков. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2002. – 256 с.

РОЗДІЛ 4. ДРАЙВЕРНА МОДЕЛЬ WDM

План

- Типи драйверів операційної системи Windows NT 5
- Основи методології Plug & Play
- Основи драйверної моделі WDM
- Порівняльна характеристика драйверних моделей WDM та WDF

В даному розділі розглядаються основні типи драйверів ОС Windows NT 5, принципи функціонування технології Plug & Play, особливості драйверної архітектури WDM, проведено порівняльну характеристику між моделями WDM та WDF, виділено їх недоліки та переваги.

4.1. Типи драйверів операційної системи Windows NT 5

Драйвер – це фрагмент коду ОС, який відповідає за взаємодію із апаратурою. Де під терміном “апаратура” розуміють як реальні фізичні пристрої (наприклад, набори мікросхем на материнських платах сучасних персональних комп’ютерів), так і віртуальні або логічні.

У минулі часи розробник драйвера міг вивчити нову апаратуру, інтерфейс спілкування ОС і драйверів, зосередитися і створити драйвер. Добре це або погано, але пора монолітних драйверів практично пройшла. Сьогодні розробник драйверів повинен вивчити архітектуру апаратних шин, архітектуру багатошарової підсистеми введення/виведення для того лише, щоб визначити мотиви вибору конструкції драйвера. Визначення**,** якого типу драйвер слід написати, вже само по собі важливе рішення. Вибір, чи слід наново реалізувати або можна повторно використовувати будь-який вже існуючий в підсистемі введення/виведення шар, є ще одним важливим рішенням.

Умовно драйвери, що використовуються у Windows NT 5, можна поділити на дві групи: драйвери режиму користувача і режиму ядра. Перші є системними програмними кодами, що функціонують в режимі користувача. Як приклад можна назвати драйвери-симулятори (віртуалізатори) для уявної апаратури або нових виконавчих підсистем (WOW, POSIX, OS/2).

Драйвери режиму ядра (kernel mode drivers) цілком складаються з коду системного рівня, що виконується в режимі ядра. Оскільки коду режиму ядра дозволено працювати безпосередньо з апаратурою, такі драйвери мають прямий доступ до управління пристроями, що містяться в комп’ютері або підключеними до комп’ютера. Зрозуміло, ніщо не може перешкодити такому драйверу представляти апаратур в режимі користувача або режимі ядра.

Обмежившись категорією драйверів режиму ядра, розглянемо код режиму ядра. На цьому рівні можна виконати ділення драйверів ще на дві категорії: успадковані (legacy, що дісталися як спадок від Windows NT 3.5, 4) і WDM-драйвери. Драйвери типу legacy (якщо не говорити про завдання проникнення в режим ядра із завданнями чистого програмування або дослідження) призначені для роботи з тими пристроями, які не підтримують PNP специфікацію, оскільки тільки розробник драйвера знає, як розрізнити його присутність в системі, не говорячи вже про прийоми роботи з ним.

Отже, все, що стосується успадкованих драйверів NT, повністю застосовуються до моделі WDM, по якій можна побудувати драйвери, які працюють в Windows 2000/XP/Server 2003 (і Windows 98, Me).

Здатність драйверів WDM працювати по PNP специфікації включає:

- участь в управлінні енергопостачанням системи;
- автоматична конфігурація пристрою;
- можливість його “гарячого” підключення.

Коректно написаний драйвер WDM може бути використаний і під Windows NT 5.x і під Windows 98/Me, хоча Microsoft не гарантує бінарної сумісності (простого перенесення .SYS файлу до іншої ОС). В більшості випадків все ще необхідна перекомпіляція під Windows 98 DDK.

Успадковані та WDM-драйвери можна розділити також і на інші три категорії: високорівневі, середньо і низькорівневі драйвери. Де високорівневі драйвери залежать від драйверів середнього і низького рівня у виконанні своїх завдань. Драйвери середнього рівня (intermediate, проміжні), відповідно, в своїй роботі залежать від функціонування драйверів низького рівня (low-level drivers).

До високорівневих драйверів, наприклад, відносяться драйвери файлових систем (file system drivers, FSDs). Такі драйвери надають ініціаторам запитів нефізичну абстракцію одержувача, і вже ці запити транслюються в специфічні запити до драйверів, що знаходяться нижче. Необхідність в створенні високорівневих драйверів виникає тоді, коли основні послуги апаратури вже реалізовані драйверами нижнього рівня, і потрібно тільки створити нову фігуру абстрагування, яка необхідна для пред’явлення ініціаторові запитів (клієнтові драйвера).

Фірма Microsoft поставляє комплект програмного забезпечення Installable File System (IFS) Kit, який розповсюджується окремо від MSDN або інших продуктів. Пакет IFS Kit потребує наявності пакету DDK (і деяких інших засобів) для того, щоб зайнятися розробкою файлової системи серйозно. При цьому існують численні обмеження на те, які типи файлових систем можуть бути отримані за допомогою цього пакету.

Драйвери середнього рівня можуть бути проілюстровані такими прикладами, як драйвери дзеркальних дисків, класові драйвери (class drivers), міні-драйвери (mini drivers) і фільтр-драйвери (filter drivers). Ці драйвери позиціонують себе між абстракціями високорівневих драйверів і засобами фізичної підтримки на нижніх рівнях. Наприклад, драйвер дзеркальних дисків

отримує запит від високорівневого драйвера файлової системи, транслює цей запит в два запити до двох різних дискових драйверів нижчого рівня. При цьому немає жодної необхідності в тому, щоб хто-небудь з драйверів верхнього або нижнього рівнів був в курсі, як насправді відбулася “дзеркалізація” і чи була вона взагалі.

Класові драйвери є можливістю повторного використання коду в межах драйверної моделі. Оскільки багато драйверів певного типу можуть мати багато загального, то програмний код, що описує загальні місця, може бути поміщений в загальний для даного типу класовий драйвер, окремо від специфічного для обслуговування конкретних пристроїв коду. Наприклад, драйвери HID (Human interface device, пристрої введення по шині USB). Драйвери специфічних пристроїв HID могли б бути, в такій ситуації, реалізовані як міні-драйвери, що взаємодіють з класовими драйверами. Вони відрізняються тим, що спілкуються тільки з іншими драйверами верхнього рівня, не виходячи на “прямий контакт” із диспетчером введення/виведення. Міні-драйвер і його клієнт вгорі мають заздалегідь обумовлений протокол спілкування, і, зазвичай, міні-драйвер експортує набір функцій свого інтерфейсу по запиту драйвера-оболонки, після чого можливе спілкування між драйверами, обминаючии диспетчер введення/виведення, що істотно прискорює роботу.

Фільтр-драйвери є драйверами середнього рівня, які позиціонують себе під час завантаження драйвера, що цікавлять їх, і перехоплюють запити, що йдуть до нього або від нього. Вони, як правило, призначені для модифікації запиту до існуючого драйвера або для реалізації якоїсь додаткової функції, спочатку не закладеної в існуючому драйвері (у простому випадку це може бути підрахунок пропущених пакетів IRP). У ОС фільтр-драйвери з великою часткою вірогідності можна пізнати по відсутності імен у об’єктів пристроїв, створених такими драйверами (при використанні програми DeviceTree).

В документації DDK також впроваджена така категорія (по відношенню до WDM драйверів) як функціональні драйвери **(**functional drivers**),** що можуть бути або класовими, або міні-драйверами. Такі драйвери завжди працюють як інтерфейс між абстрактним запитом введення/виведення і кодом низькорівневого фізичного драйвера, “фізичного”, – в тому сенсі, що він пов’язаний безпосередньо з пристроєм і в його функціонуванні добре є видимими особливості цього пристрою. Характерна в даному випадку наступна деталь. Коли типовий WDM-драйвер нормального пристрою PNP збирається підключити себе до стека пристроїв (це повинно відбуватися в процедурі AddDevice), він виконує підключення функціонального об’єкту пристрою (FDO), створеного ним самим, до фізичного об’єкту пристрою (PDO), наданого йому батьківським драйвером. Як правило, батьківським є драйвер шини, який спочатку виявив підключення даного пристрою, ініціювавши потім звернення до даного WDM-драйвера пристрою. Ось тут і виявляється функціональність останнього: він отримує загальні функціональні запити, а перетворює їх на

низькорівневі, зрозумілі шинному драйверу і пристрою, втілюючи при цьому для свого клієнта функції пристрою, а не його конструкцію і внутрішню логіку.

4.2. Основи методології Plug & Play

Технологія Plug and Play (PNP) підтримується комбінацією апаратного і програмного забезпечення, що дозволяє розпізнавати і настроювати апаратні зміни в конфігурації майже без втручання користувача. Можна динамічно додавати і видаляти пристрою без необхідності реконфігурації системи і знання складного комп'ютерного устаткування.

Вперше концепція PNP була реалізована в ОС Windows 95, але з того часу ця технологія отримала істотний розвиток в плані управління системою, конфігурації пристроїв і управління енергоспоживанням, особливо завдяки ініціативній проектній групі OnNow. Одним з результатів роботи цієї групи стала специфікація ACPI (Advanced Configuration and Power Interface) версії 1.0, що визначила новий дизайн материнських плат і BIOS, що забезпечує управління енергоспоживанням і нові конфігураційні можливості під повним управлінням ОС.

Методи розпізнавання устаткування, визначені специфікацією ACPI, не залежать від ОС або типу центрального процесора. ACPI визначає інтерфейс функцій PNP і управління енергоспоживанням на рівні регістрів і дає описовий інтерфейс для нових можливостей устаткування. Це дозволяє проектувальникам створювати широкий діапазон нових пристроїв з використанням тих же драйверів ОС. ACPI забезпечує також типовий механізм управління PNP і управління енергоспоживанням, що базується на системних подіях.

При проектуванні архітектури PNP в Windows NT 5.0 ставилися дві основні цілі:

- розширення існуючої інфраструктури введення/виведення для підтримки PNP і управління енергоспоживанням для роботи з устаткуванням по стандартах PNP;
- створення єдиних інтерфейсів для драйверів пристроїв, що підтримують PNP і управління енергоспоживанням, для класів пристроїв, які працюють під Windows NT 5.0 і Windows 98

Підтримка PNP в 5.0 включає початкову інсталяцію системи, розпізнавання змін устаткування подіям між окремими завантаженнями системи, відгук системи на такі події часу виконання, як підключення/відключення комп'ютера, під'єднання/від'єднання периферійних пристроїв.

Драйвери PNP не надають пристроям власні ресурси. Замість цього необхідні ресурси ідентифікуються системою під час розпізнавання пристроїв. Ґрунтуючись на запитах пристроїв на ресурси, диспетчер PNP призначає відповідні апаратні ресурси (наприклад, порти введення/виведення, канали DMA тощо), розподіляє пам'ять. Диспетчер PNP реконфігурує призначені

ресурси з потреби, коли пристрої додаються до системи і запрошують вже розподілені ресурси. Крім того, диспетчер PNP визначає, які драйвери потрібні для підтримки конкретних пристроїв, і завантажує ці драйвери.

Одна з ключових рис PNP і системи управління енергоспоживанням – це динамічна обробка подій. Прикладами таких подій можуть бути додавання або видалення пристроїв, включення/виключення режиму зниженого енергоспоживання. PNP і система управління енергоспоживанням, використовуючи функції WDM, обробляють динамічні події схожим чином.

Головною перевагою використання методології PNP є забезпечення автоматичної підтримки інсталяції й видалення системних пристроїв. Щоб досягти цього, необхідно виконати кілька умов, основними з яких є:

- пристрої повинні бути орієнтовані на виконання програмного налаштування. Повинна існувати можливість установки портів введення/виведення, задіяних переривань і ресурсів DMA з керуючого програмного забезпечення, крім механічного налаштування за допомогою перемичок, що переставляються, і DIP-перемикачів на платах;
- система повинна забезпечувати надійне автоматичне виявлення нового пристрою або факт видалення існуючого. Пристрій і шина, до якої пристрій підключений (було підключено), повинні інформувати керуюче програмне забезпечення про те, що з апаратною конфігурацією відбулися часткові зміни;
- необхідні драйвери для нових пристроїв повинні завантажуватися автоматично, у міру виявлення цих пристроїв ОС (втручання користувача все-таки потрібно, але тільки лише для установки драйверів для ніколи раніше не присутніх у системі нестандартних пристроїв);
- у тих випадках, коли пристрій і його інтерфейсна шина дозволяють, ОС повинна підтримувати “гаряче” приєднання апаратури (при включеному живленні). Тобто повинна бути можливість підключення/відключення пристрою безпосередньо в “живу” систему без створення для неї стресових ситуацій.

До елементів системи підтримки PNP в ОС входять:

1. PNP-менеджер, що складається із двох частин – частини, яка працює в режимі ядра й частини, яка виконується в режимі користуча. Частина з режиму ядра взаємодіє з апаратурою й іншими програмними компонентами, що функціонують у режимі ядра, забезпечуючи керування правильним визначенням і налаштуванням апаратури. Частина з режиму користувача взаємодіє з компонентами користувальницького інтерфейсу, дозволяючи інтерактивній програмі робити запити й змінювати конфігурацію інстальованого PNP програмного забезпечення.
2. Менеджер керування енергоживленням (Power Manager), що визначає й обробляє події енергозабезпечення.
3. Системний Реєстр Windows, який є базою даних установленого апаратного й програмного забезпечення, що підтримує специфікацію PNP.

Вміст реєстру допомагає драйверам й іншим компонентам при визначенні ресурсів, що використовуються будь-яким конкретним пристроєм.

4. Inf-файли. Кожний пристрій повинне бути повністю описано файлом, що використовується при інсталяції керуючого їм драйвера. Inf-файл є рецептом, як і яку інформацію про пристрій заносити, зокрема, до Системного Реєстру.

5. Драйвери для PNP пристроїв, які можна розділити на дві категорії: WDM і NT-драйвера. Останні є "успадкованими" від NT-драйверами, які спираються на деякі аспекти PNP архітектури, але, з іншого боку, не повністю задовольняють моделі WDM. Наприклад, вони можуть використовувати сервіси PNP менеджера, щоб одержати інформацію про конфігурацію, але при цьому не обробляють IRP пакети повідомлення з кодом IRP_MJ_PNP. Драйвери WDM моделі, по визначенню, повністю відповідають вимогам взаємодії за правилами PNP.

Тому для більшої наочності представимо розглянуті програмні PNP-компоненти у Windows NT на рис. 4.1.

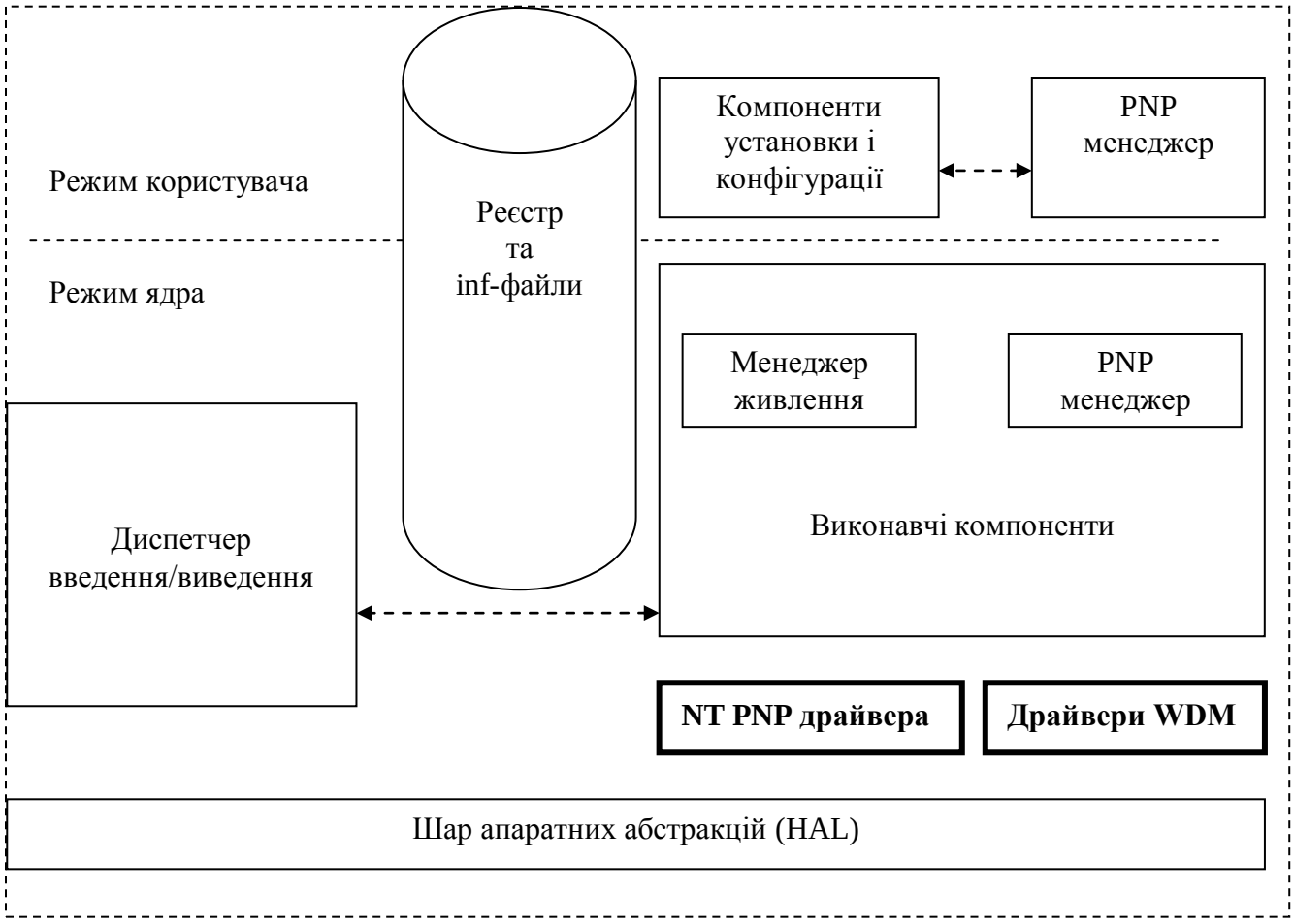

Рис. 4.1. Програмні PNP-компоненти у ОС Windows NT

Підтримка PNP в ОС Windows NT 5.0 внесла ряд змін до раніше розробленої моделі драйверів Windows NT, а саме:

1. Драйвери шини відокремлені від HAL для координації із змінами і розширеннями існуючих компонентів режиму ядра (виконувана частина,

драйвери ядра, HAL). Драйвери шини управляють шиною введення/виведення, включаючи управління окремими слотами, незалежними від пристроїв.

2. Нові методи і можливості для підтримки установки пристроїв і конфігурації внесли зміни до таких існуючих компонентів режиму користувача, як диспетчер буфера виведення, установники класів, застосувань Control Panel, Setup. Додані нові компоненти для підтримки PNP як в режимі ядра, так і в режимі користувача.

3. Додані інтерфейси PNP API для читання і запису інформації з реєстру (registry). Тепер структура регістра підтримує PNP і дозволяє вносити вдосконалення в майбутніх версіях NT, в той же час забезпечуючи сумісність знизу догори.

Windows NT 5.0 підтримує успадковані драйвери, але їх використання зменшує можливості системи в області підтримки PNP. Виробникам, охочим реалізувати повні можливості PNP для свого устаткування і використовувати одні й ті ж драйвери під NT та Windows 98, необхідно розробити нові драйвери, інтегруючи останні досягнення технології PNP і управління енергоспоживанням.

4.3. Основи драйверної моделі WDM

WDM (Windows Driver Model) – це драйверна модель від Microsoft для ОС Windows, що прийшла на зміну попередньому середовищу написання драйверів для ОС Windows – VxD (Virtual Device Driver). Сьогодні вона є однією з наважливіших концепцій у написанні драйверів, розбиратися в якій обов'язково необхідно будь-якому розроблювачеві драйверів під Windows.

Отже, спробуємо розглянути архітектуру WDM, а в її контексті – основні функції драйвера та їхнє призначення.

До головних особливостей драйверної моделі WDM відносять:

- сумісність на рівні двійкових кодів між драйверами для ОС Windows 98 і NT;
- підтримка керування живленням пристрою (power management), що надає системі можливість значно зберігати електричну енергію шляхом вибіркового відключення живлення декільком або усім пристроям у системі. Технологія керування живленням використовує WDI (Windows Management and Instrumentations), і тому останній необхідний для WDM-драйверів пристроїв;
- підтримка PNP;
- підтримка "просунутого" шинного керування (advanced bus management).

До компонентів режиму ядра, що забезпечують підтримку керування живленням відносять:

- BIOS, що підтримує ACPI;
- наявність драйвера ACPI;
- менеджер керування живленням;

– підтримка цього механізму з боку драйвера (якщо драйверу це потрібно, звичайно).

Якщо розглянути в спрощеному вигляді життєвий цикл “середньостатистичного” WDM-драйвера, то він буде складатися із таких етапів:

1. Драйвер шини визначає пристрій.
2. PNP-менеджер визначає місцезнаходження ключа пристрою у гілці Enum реєстру. Цей ключ містить вказівник на інший ключ реєстру, що визначає функціональний драйвер пристрою, який управляє окремим пристроєм і є його основним драйвером). PNP-менеджер динамічно завантажує функціональний драйвер.
3. PNP-менеджер викликає функцію драйвера AddDevice для того, щоб створити DRIVER_OBJECT. Якщо драйвер відповідає більше, ніж одному фактичному пристрою, PNP-менеджер викликає AddDevice для кожного з них. З цього моменту вся комунікація драйвера із зовнішнім світом здійснюється з використанням IRP(I/O Request: Packet) пакетів.
4. PNP-менеджер виділяє всі необхідні драйверу ресурси введення/виведення (запити на переривання, номери портів тощо) і надсилає запит для ініціалізації пристрою.
5. Деякі пристрої можуть бути вилучені із системи без вимикання комп’ютера. Якщо пристрій – один з таких, то PNP-менеджер посилає драйверу спеціальний IRP-пакет, в результаті чого створений функцією AddDevice об’єкт пристрою знищується.
6. Коли всі пристрої вилучені, то менеджер введення/виведення (I/O Manager) викликає функцію DriverUnload, яка видаляє образ драйвера з пам’яті.

До головних функцій драйвера в контексті архітектури WDM можна віднести:

– DriverEntry, заповнює вказівники на функції усередині об’єкта драйвера. Якщо на даному етапі потрібні ще будь-які глобальні ініціалізації, то ця функція виконує їх;
– DispatchPnp і DispatchPower – функції призначені для обробки PNP і запити Power I/O відповідно;
– DriverUnload – функція вивантаження драйвера, повинна очистити всі глобальні ініціалізації, зроблені функцією DriverEntry (звільнити ресурси і т.д.);
– AddDevice – цю функцію система викликає, щоб повідомити про появу пристрою, для якого необхідно управління, PNP-менеджер викликає цю функцію для кожного пристрою, який керується драйвером;
– IoCreateDevice – функція для створення об’єкта пристрою й екземпляра об’єкта розширення пристрою;
– IoActtichDeviceToDeviceStack – функція для включення нового пристрою в стек пристроїв.

В драйверній моделі WDM для обробки PNP і Power IRP PNP-менеджер використовує IRP для прямого старту, зупинки й видалення пристроїв і для

запиту драйверів про їхні пристрої. Всі PNP JRP мають головний функціональний код – IRP_MJ_PNP.

Драйвери повинні обробляти PNP IRP у функції XxxDispatchPnp, де Xxx – це префікс, що ідентифікує пристрій. Драйвер встановлює адресу функції DispatchPnp в DriverObject->MajorFunction[IRP_MJ_PNP] під час ініціалізації драйвера в його функції DriverEntry. PNP-менеджер, за допомогою I/O-менеджера, викликає функцію DispatchPnp. Слід відмітити, що всі запити керування живленням мають IRP-код IRP_MJ_POWER. Тому деякі головні функції драйвера в контексті архітектури WDM розглянемо нижче.

WDM-функція DriverEntry заповнює вказівники на функції усередині об'єкта драйвера. Якщо на даному етапі потрібні ще будь-які глобальні ініціалізації, то функція *DriverEntry* виконує їх:

```
    NTSTATUS
    DriverEntry{
    IN PDRIVER_OBJECT DriverObject,
    It PUNICODE_STRING RegistryPath
);
{
    DriverObject->DriverUnload = DriverUnload;
    DriverObject->Drive lExt ension->AddDevice = AddDevice;
    DriverObject->MajorFunction[IRP_MJ_PNP] = DispatchPnp;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
     …
    return STATUS_SUCCESS;
}
```

Кожний драйвер має обробляти PNP і запити Power I/O. Для цього будуть використовуватись функції *DispatchPnp* і *DispatchPower.*

Функція *DriverUnload* – функція вивантаження драйвера повинна очистити всі глобальні ініціалізації, зроблені функцією *DriverEntry* (звільнити ресурси і т.д.):

```
    VOID
    XxxUnload {
    IN PDRIVER_OBJECT DriverObject
    }
```

Потрібно відзначити, що в драйверах не-PNP функція *DriverUnload* виконує значно більшу роботу, ніж така ж у PNP-драйверах.

Далі розглянемо функцію *AddDevice*. Вона прийшла разом з архітектурою WDM. Цю функцію система викликає, щоб повідомити про появу пристрою, для якого необхідно управління, PNP-менеджер викликає цю функцію для кожного пристрою, який керується драйвером:

```
    NTSTATUS
    XxxAddDevice{
    IN PDRIVER_OBJECT DriverObject,
    It PDEVICE_OBJECT PhysicalDeviceObject
    );
```

Параметр DriverObject вказує на той самий об'єкт драйвера, що був проініціалізований в функції DriverEntry. Аргумент PDO – PhysicalDeviceObject – указує на фізичний об'єкт пристрою, який розташований на самому дні стеку пристроїв. Основні завдання AddDevice в функціональному драйвері – створити об'єкт пристрою й "прикріпити" його до стеку. Це реалізовується наступною послідовністю кроків:

1. Викликається функція IoCreateDevice для створення об'єкта пристрою й екземпляра об'єкта розширення пристрою.
2. Реєструється один або кілька інтерфейсів пристрою, для того щоб програмні додатки знали про його існування. Можливий інший варіант: давти об'єкту пристрою ім'я й створити на нього символічне посилання.
3. Провести ініціалізацію об'єкта розширення пристрою й поля Flags (прапорці) об'єкта пристрою.
4. Викликати IoActtichDeviceToDeviceStack для включення нового пристрою в стек пристроїв.

Кожен фільтр-драйвер і функціональний драйвер зобов'язані створювати стек об'єктів пристроїв, починаючи з PDO. Все це легко реалізовується за допомогою функції IoAttachDeviceToDeviceStack:

```
PDEVICE_OBJECT
IoAttachDeviceToDeviceStack{
IN PDRIVER_OBJECT SourceDevice,
It PDEVICE_OBJECT TargetDevice
);
```

Як може бути використана ця функція усередині *AddDevice*, показано нижче:

```
WTSTATUS AddDevice ...
{
PDEVICE_OBJECT fdo;
IoCreateDevice{..., &fdo);
pdx->LowerDeviceObject = loAttachDeviceToDeviceStack(fdo,pdo);
}
```

Перший аргумент функції loAttachDeviceToDeviceStack – це адреса недавно створеного об'єкта пристрою. Другий аргумент – це адреса PDO, що був одержаний в якості аргументу функції AddDevice. Значення, що повертається, – це адреса будь-якого об'єкта пристрою, що перебуває внизу стека після об'єкта. Це може бути PDO або ж адреса будь-якого більш низького фільтра-об'єкта пристрою.

Перейдемо до інсталяції драйвера. Інсталювати драйвер можна досить великою кількістю способів. Всі наведене вище відноситься до звичайних драйверів. А зараз розглянемо особливий драйвер – фільтр-драйвер.

WDM фільтр-драйвер – це особливий тип драйвера, що перебуває над яким-небудь драйвером і перехоплює запити, які йдуть до нього. Фільтр-драйвер працює за допомогою приєднання свого об'єкта пристрою до об'єкту пристрою більш низького драйвера. Фільтр-драйвер, що перебуває вище

функціонального драйвера, називається Upper Filter Driver, а той, що перебуває нижче – Lower Filter Driver. Структура обох типів драйверів практично однакова. Застосувань фільтр-драйверів може бути багато – перехоплення запитів введення/виведення, додаткова їхня обробка тощо.

Розглянемо скелет WDM фільтр-драйвера. Фільтр-драйвер, так само, як будь-який інший драйвер, має функції DriverEntry і AddDevice.

Функція DriverEntry фільтр-драйвера практично ідентична такій же функції звичайного драйвера. Але на відміну від звичайного драйвера, фільтр-драйвер повинен мати функції для всіх типів IRP, а не тільки для тих, які збирається обробляти, наприклад:

```
NTSTATUS DriverEntry (PDRIVER_OBJECT DriverObject,
FUNICODE_STRING RegistryPath)
{
  DriverObject->DriverUnload = DriverUnload;
       DriverObject->DriverExtension->AddDevice = AddDevice;
    for {int 1 = 0; i MajorFunction) ++i)
DriverObject->MajorFunction[i] = DispatchAny;
    DriverObject->MajorFunction[IRP_MJ_POWER] = DispatchPower;
  DriverObject->MajorFunctioin[IRP_MJ_PNP] = DispatchPnp;
  return STATUS_SUCCESS;
}
```

У функції AddDevice викликається IoCreateDevice для створення безіменного об’єкту пристрою, а потім IoAttachDeviceToDeviceStack для включення його в стек. Також необхідно скопіювати деякі налаштування об’єкту пристрою, який знаходиться під об’єктом драйвера, тому дана функція має вигляд:

```
NTSTATUS AddDevice(PDRIVER_OBJECT DriverObject, PDEVICE_OBJECT pdo)
{
  PDEVICE_OBJECT fido;
  NTSTATUS status = IoCreateDevice(DriverObject,
sizeof(DEVICE_EXTENSION), NULL, FILE_DEVICE_UNKNOWN,0, FALSE, &fido); //
створюємо пристрій
  if (!NT_SUCCESS(status)) return status;
  PDEVICE_EXTENSION pdx=(PDEVICE_EXTENSION) fido->DeviceExtension;
// розширення пристрою
 {
   pdx->DeviceObject = fido;
   pdx->Pdo - pdo;
   IoInitializeRemoveLock(&pdx->RemoveLock, 0, 0, 255);
   PDEVICE_OBJECT fdo = IoAttachDeviceToDeviceToStack(fido, pdo);
   // назначаємо пристрій до стеку
   pdx->LowerDeviceObject = fdo; // вказівник на нижчий об’єкт
   fido->Flagi = fdo->Flags & (DO_DIRECT_IO  DO_BUFFERED_IO
 DO_POWER_PAGABLE  DO_POWER_INRUSH);
   fido->DeviceType = fdo->DeviceType;
   fido->Characteristics = fdo->Characteristics;
   fido->AlignmentRequirement = fdo->AlignmentRequirement;
   fido->Flags  &- ~DO_DEVICE_INITIALIZING;
 }
     }
```

Оскільки фільтр-драйвер використовується в основному для того, щоб змінити поведінку пристрою необхідні будуть функції, які обробляють вхідні IRP пакети, перш ніж передати їх далі. Далі будемо просто передавати більшість IRP вниз по стеку. Розглянемо головну функцію обробки IRP-функції – DispatchAny, яка призначена для обробки всіх IRP, за винятком IRP_MJ_PNP і IRP_MJ_POWER:

```
NTSTATUS DispatchAny(PDEVICE_OBJECT fido, PIRP Irp)
{
  PDEVICE_EXTENSION pdx=(PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status,0);
  IoSkipCurrentIrpStackLocation(Irp); //копіюємо параметр IRP-стека
                                      //с поточної позиції
    //до вищестоячого драйвера
  status = IoCallDriver(pdx->LowerDeviceObject, Irp};
    //посилаємо запит
    //вишестоячому драйверу
  IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
}
```

IoAcquireemoveLock перешкоджає видаленню об’єкта пристрою в той час, коли відбувається звернення до полів всередині його й всередині об’єкта розширення пристрою, який “прикріплений” до нього, CompleteRequest – це допоміжна функція, що працює з механізмом завершення IRP.

Розглянемо обробку окремих IRP: IRP_MJ_PNP і IRP_MJ_POWER.

Обробка IRP_MJ_PNP буде мати вигляд:

```
NTSTATUS DispatchPnp(PDEVICE_OBJECT fido, PIRP Irp)
{
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0) ;
  // отримуємо поточну позицію стека
  PIO_STACK_LOCATION stack = IoGetCurrentIrpStackLocation(Irp);
  ULONG fcn = stack->MinorFunction;
  IoSkipCurrentIrpStackLocation(Irp);
  status = IoCallDriver(&pdx->LowerDeviceObject, Irp);
  if (fcn == IRP_MN_REMOVE_DEVICE)
{
IoReleaseRemoveLockAndWait(&pdx->RemoveLock, Irp);
  IoDetachDevice(pdx->LowerDeviceObject); //від’єднуємо пристрій
  IoDeleteDevice(fido);                   // видаляємо пристрій
     }
  else IoReleaseRemoveLock(&pdx->RemoveLock, Irp);
  return status;
}
```

Обробка IRP_MJ_POWER можна представити таким чином:

```
NTSTATUS DispatchPower(PDEVICE_OBJECT fido, PIRP Icp)
{
```

```
  PoStartNextPowerIrp(Irp); // інформуємо Power Manager проте,
   // що драйвер готовий обробляти наступний запит IRP-Power
  PDEVICE_EXTENSION pdx = (PDEVICE_EXTENSION) fido->DeviceExtension;
  NTSTATUS status = IoAcquireRemoveLock(&pdx->RemoveLock, Irp);
  if (!NT_SUCCESS(status)) return CompleteRequest(Irp, status, 0);
  IoSkipCurrentIrpStackLocation(Irp);
  status = PoCallDriver(pdx->LowerDeviceObject, Irp); // відправляє
// запит до вищестоячого драйвера
  IoReleaseRemveLock(&pdx->RemoveLock, Irp);
  return status;
}
```

Дві основні відмінності від функції – обробка DispatchAny в тому, що необхідно викликати функції PoStartNextPowerIrp в момент отримання IRP (це потрібно зробити в будь-якому випадку) і PoCallDriver (замість IoCallDriver), щоб передати IRP наступному драйверу.

4.4. Порівняльна характеристика драйверних моделей WDM та WDF

WDF – це нова драйверна модель Microsoft, яка працює в ОС із родини Microsoft Windows Vista, Microsoft Windows Server 2003, Microsoft Windows XP та Microsoft Windows 2000.

Драйвери WDF використовуються для тих же цілей, що і драйвери WDM: вони здійснюють взаємодію між Windows та пристроєм. Хоча WDF представляє нову драйверну модель, вона не є окремою від WDM. WDF виконує функцію шару абстракції між моделлю WDM та WDF-драйвером, який спрощує задачу створення надійних, безпечних та ефективних драйверів. WDF надає інфраструктуру, яка виконує ключові задачі драйверів WDM: отримує і оброблює пакети IRP, керує змінами в стані PNP та енергоживлення, і т. ін.

До складу драйверної моделі WDF входить три компоненти, перші дві називаються інфраструктурою:

1. Середовище для написання драйверів режиму ядра (Kernel Mode Driver Framework – KMDF).
2. Середовище для написання драйверів режиму користувача (User Mode Driver Framework, UMDF).
3. Інструменти для перевірки та відладки драйверів.

Хоча WDF підтримує дві перші інфраструктури, проте конструкція та функціональність обох цих інфраструктур дуже схожа.

UMDF підтримує користувальницькі типи драйверів, а KMDF підтримує такі типи kernel-драйверів:

- шинні драйвери для стеку пристроїв PNP;
- фільтр-драйвери для пристроїв
- legacy-драйвери для пристроїв, що не включені в стек PNP;
- функціональні драйвери для пристроїв PNP.

WDF представляє загальну драйверну модель для широкого кола типів пристроїв. Найбільш суттєвими особливостями моделі є такі:

1. Підтримка драйверів, як користувальницького режиму, так і режиму ядра.
2. Добре спроектована об’єктна модель.
3. Об’єктна ієрархія, яка спрощує управління часом життя об’єктів та синхронізацію запитів введення/виведення.
4. Модель введення/виведення, в якій інфраструктури управляють взаємодіями із ОС та потоком запитів введення/виведення, включаючи реагування на повідомлення PNPта енергоживлення.
5. Реалізація механізму PNPта управління енергоживленням, яка надає надійне управління станами та стандартне інтелектуальне управління переходами між станами. Драйвери WDF оброблюють лише ті переходи між станами, які мають відношення до їх пристроїв.

Інфраструктура пропонує набір об’єктів, які надають різні, пов’язані із Windows та драйверами, базові структурні компоненти, такі як драйвер, запит введення/виведення, черга та т. ін. Драйвери WDF використовують об’єкти інфраструктури для реалізації різних аспектів функціональності драйвера. Кожний об’єкт надає програмний інтерфейс, який драйвер використовує для доступу до властивостей об’єкта або щоб дати об’єкту вказівку виконати задачу. Кожен об’єкт також підтримує одне або більше повідомлень, які викликаються у відповідь на такі явища, як зміна системного часу або прибуття запиту введення/виведення.

Деякі об’єкти інфраструктури створюються самою інфраструктурою та передаються драйверу WDF. Інші об’єкти інфраструктури створюються тоді, коли у драйвера WDF виникає в них потреба. Задача розробника полягає в тому, щоб зібрати відповідні об’єкти у структуру, що підтримує відповідні вимоги пристрою. В залежності від конкретних вимог пристрою, структури відрізняються в деталях реалізації, але всі драйвери WDF створюються із одних і тих же базових блоків.

Деякі об’єкти постійні або мають дуже довгий час життя, тобто вони існують на протязі часу життя даного пристрою. Наприклад, при фізичному підключенні пристрою драйвер WDF створює об’єкт пристрою для надання пристрою та один чи декілька об’єктів черги для управління запитами введення/виведення. Інші ж об’єкти не довговічні; вони служать для рішення визначених короткочасних задач та знищуються одразу ж після рішення цієї задачі. По умовчанню інфраструктура надає обробку для всіх повідомлень, тому драйвера WDF не обов’язково оброблювати їх. Але в результаті цієї цілковитої залежності від інфраструктури для стандартної обробки повідомлень буде отримано функціональний, але відносно нецікавий драйвер. Щоб замінити стандартну обробку повідомлень інфраструктурою своєї обробкою, драйвер WDF повинен явним чином зареєструвати спеціальну функцію оберненого виклику. Інфраструктура викличе цю функцію оберненого виклику, щоб повідомити драйвер WDF про виникнення повідомлення та дозволить йому виконати необхідне обслуговування. Саме ця модель опійної обробки повідомлень драйвером є ключовим аспектом WDF. Драйвери WDF реєструють функції оберненого виклику лише для тих повідомлень, для яких стандартна

обробка, що надається інфраструктурою, не є достатньою. Всі інші повідомлення оброблюються інфраструктурою, без втручання драйверу.

Як зазначалось вище, підтримка технологій управління живленням та PNP в драйверній моделі WDM здійснюється досить непросто, тобто необхідно писати багато рядків коду, рішати велику кількість проблем тощо. WDF пропонує такі концепції для рішення цих проблем:

- драйвер не повинен оброблювати і/або відповідати на всі вхідні запити;
- дії WDF повинні в будь-який момент бути чітко визначені та передбачені;
- середовище повинно надавати стандартні обробники для великої кількості можливостей PNP;
- драйвер повинен мати можливість перевизначити будь-які встановлені середовищем значення по замовчуванням.

WDF інтегрує технологію PNPі управління живленням із технологією організації черг I/O-запитів, а останню, в свою чергу, із технологією відміни запитів.

Отже, можна виділити основні недоліки та переваги драйверних моделей WDM та WDF.

До недоліків драйверної моделі WDM можна віднести такі:

- складність написання драйверів;
- велика кількість існуючих різних моделей портів;
- більшість драйверів, написаних із застосуванням старих технологій, можуть виконуватися лише в режимі ядра;
- наявність різних драйверних моделей призводить до труднощів при тестуванні та верифікації.

До основних переваг драйверної моделі WDF можна віднести:

- простота та гнучкість;
- розширюваність, версійність, незалежність від основних компонентів ОС;
- можливість роботи багатьох драйверів у користувальницькому інтерфейсі;
- написання драйверів на мовах високого рівня;
- вміння ізолювати драйвер, тобто створювати окреме “захисне оточення”.

Висновки

- ✦ Драйвер – це фрагмент коду операційної системи, який вілповідає за взаємодію із апаратурою. У ОС Windows NT 5 виділяють драйвери режиму користувача та драйвери режиму ядра.
- ✦ Технологія PNP підтримується комбінацією апаратного та програмного забезпечення, що дозволяє розпізнавати і настроювати апаратні зміни в конфігурації майже без втручання користувача. Головною перевагою використання методології PNP є забезпечення автоматичної підтримки інсталяції й видалення системних пристроїв.

- Оскільки кінцевою версією драйверної моделі на сьогодні стала WDM модель. Були обговорені специфічні аспекти розробки драйверних процедур у аспектах нової моделі й розглянуті приклади використання стандартних системних викликів режиму ядра в процедурах WDM-драйверів, що реалізують багатошарову методологію сучасної драйверної моделі Windows.

Контрольні запитання та завдання

1. Проведіть порівняльну характеристику між класовими драйверами, міні-драйверами та фільтр-драйверами.
2. Призначення технології PNP.
3. Охарактеризуйте елементи підтримки PNP, що входять до складу ОС.
4. Призначення драйверної моделі WDM.
5. Життєвий цикл “середньостатистичного” WDM-драйвера.
6. Назвіть головні функції драйвера в контексті архітектури WDM.
7. Проведіть порівняльну характеристику драйверних моделей WDM та WDF.
8. Об’єкти драйверу WDF.

Тести для самоконтролю

1. Що таке драйвер?
   a. програма, за допомогою якої операційна система отримує доступ до керування апаратним забезпеченням
   b. єдина та неподільна частина ОС, що виконує певні специфічні функції
   c. прикладна програма з розширеними можливостями
2. Драйверна концепція описує взаємодію:
   a. користувача з користувачем
   b. комп’ютер з комп’ютером
   c. системи з пристроями
3. На даний момент найбільш актуальною і поширеною є драйверна модель:
   a. WDF
   b. WDM
   c. VxD
   d. SuperDrM

4. Як розшифровується WDM?
   a. Windows Driver Model
   b. Windows Driver Master
   c. Windows Driver Machine
5. Що таке WMI?

a. це набір модулів до WDM-моделі, що надає ОС сукупність інструментів, за допомогою яких вона може запускати ті чи інші заплановані служби
b. це набір спеціальних розширень до WDM-моделі, що надає інтерфейс ОС, за допомогою якого компоненти мають можливість надавати різну інформацію й оповіщення
c. це спеціальний WDM-драйвер, що надає інтерфейс ОС, за допомогою якого інші драйвери можуть працювати в режимі ядра

6. Фільтр-драйвери – це:
   a. драйвери, які входять до складу антивірусного програмного забезпечення і є базовим компонентом
   b. фільтрують запити на введення/виведення, які находять до обслуговуючого пристрою і захищають буфер від переповнення
   c. забезпечують модифікацію запиту на введення/виведення перед передачею його драйверам більш низьких рівнів
7. Основна функція, яка виступає початковою точкою стикування ОС з драйвером – це:
   a. AddDevice
   b. DriverEntry
   c. MyPNP_Handler

Теми для самостійного опрацювання

1. Об’єктна модель KMDF.
2. Об’єктна модель UMDF.
3. Модель WDM та методологія WDF.
4. Архітектура WDF.
5. Робота із нижніми шарами драйверного стеку.

Список використаної літератури

1. *Агуров П. В.* Интерфейсы USB. Практика использования и программирования / П. В. Агуров. – СПб.: БХВ-Петербург, 2004. – 576 с.
2. *Комиссарова В.* Программирование драйверов для Windows / В. Комиссарова. – СПб.: БХВ-Петербург, 2007. – 256 с.
3. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
4. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.
5.

ЧАСТИНА 2. ПРОЕКТУВАННЯ ДРАЙВЕРІВ ОПЕРАЦІЙНИХ СИСТЕМ

РОЗДІЛ 5. ОСНОВНІ ПРИЙОМИ ПРОГРАМУВАННЯ ДРАЙВЕРІВ В РЕЖИМІ ЯДРА

План

- Порівняльна характеристика драйверів різних операційних систем
- Особливості і прийоми програмування в режимі ядра
- Архітектура драйверу
- Переривання та рівні IRQL
- Структура драйверу
- Найпростіший драйвер
- Поради щодо програмування в режимі ядра

В даному розділі проводиться огляд драйверів різних операційних систем, розглядаються особливості та прийоми програмування в режимі ядра, а також структура самого драйверу.

5.1. Порівняльна характеристика драйверів різних операційних систем

З моменту своєї появи і по сьогодні, драйвер постійно еволюціонував, і процес цей досі ще не завершився. Концепція драйвера, як окремого змінного модуля сформувалася не одразу. Деякі версії UNIX і сьогодні практикують повну перекомпіляцію ядра при заміні будь-якого драйвера, що абсолютно не схоже на поводження з драйверами в операційних системах Linux, Windows і MS DOS. Саме MS DOS ввела в масовий обіг поняття драйвера, що дозволило після чергового перезавантаження поліпшити якість життя користувача.

Розглянемо основні характерні риси драйвера (ядра, що працює з повноваженнями компоненту) для різних ОС:

1. В операційних системах MS DOS, Windows, Unix і всіх клонах Linux прийнятий спосіб роботи з драйверами як з файлами. Тобто при доступі до драйвера використовуються функції або співпадаючі (лексично), або дуже схожі на функції для роботи з файлами (open, close, read, write, CreateFile...).

Даний порядок недивний для систем родини Unix, оскільки в них вся дійсність сприймається у вигляді файлів (що є початковою концепцією даної гілки ОС). Наприклад, директорію (каталог файлів) можна відкрити як файл і

прочитувати звідти блоки даних, що відповідають інформації про кожен файл, що зберігається в цій директорії.

В ОС Windows пропонується такий же механізм. Для доступу до драйвера зі свого додатку користувач використовує функцію CreateFile**.** Проте цей файл для "відкриття" на жорсткому диску не існує, його знайти можна лише в надрах ОС і виглядає він, наприклад, як "\.\myDevice" (ОС розуміє його як символьне посилання для ідентифікації конкретного драйвера).

2. Драйвери стали легко замінюваною запасною частиною в операційній системі. Якщо раніше і були відмінності між продуктами Microsoft і Unix системами (драйвери в ОС Microsoft спочатку були "рухомо-змінними" але в UNIX і ранніх версіях Linux при їх заміні треба було наново виконувати перекомпіляцію ядра), то сьогодні такі відмінності зникли. При збереженні деяких особливостей інсталяції, драйвери тепер всюди можуть бути видалені/додані в систему редагуванням одного запису в спеціальних системних файлах. Більш того, завантаження "на вимогу" (по запиту призначеної для користувача програми) стає практично загальною межею Windows/Unix/Linux. Навіть ОС реального часу, наприклад, QNX, також використовують методику змінних драйверів.

3. Концепція існування режиму ядра (з великими функціональними можливостями і відносній безконтрольності) і призначеного для користувача режиму (з жорстким контролем з боку системи) присутня в Windows/Unix. Якщо у Linux драйвер реалізується як лише модуль ядра, що має якесь (додаткове) віддзеркалення у вигляді файлу в директорії /dev/. То в операційній системі Windows драйвер (режиму ядра) представляється не просто драйвером, а можливістю увійти до режиму ядра зі своїм програмним кодом.

Проте у всіх популярних ОС загальним при порівнянні драйверів залишається механізм дії на драйвером за допомогою IOCTL (Input/Output Control Code, код управління введенням/виведенням) запитів.

Ядро Windows не призначене для самостійної взаємодії з пристроями. Для виявлення приєднаних пристроїв, обміну інформацією з пристроями і надання можливостей драйвера клієнтам ядро Windows покладається на драйвери пристроїв. Windows же надає абстрактний інтерфейс для підтримки пристроїв, що називається драйверною моделлю. Завдання розробника драйверів полягає в реалізації даного інтерфейсу, який буде підтримувати конкретні вимоги апаратного пристрою. Тобто, завданням драйвера є надання зв'язку і контроль взаємодії між додатками і пристроєм. У багатьох відношеннях робота драйверів схожа з роботою сервісів. Так, наприклад, драйвери:

- виконуються у фоновому режимі, окремо від процесів додатків, і доступні багатьом користувачам;
- мають довгий час життя – таке ж, як і час життя відповідних ним пристроїв. Драйвер пристрою запускається, коли Windows виявляє пристрій, і припиняє виконання, коли пристрій видалений;
- відповідають на зовнішні запити введення/виведення. Ці запити зазвичай виходять від додатків Windows та іноді від інших драйверів;

- не мають призначеного для користувача інтерфейсу. Користувачі зазвичай взаємодіють з драйвером, указуючи додатку видати запит введення/виведення;
- виконуються в іншому адресному просторі, ніж адресний простір додатку, що видало запит введення/виведення.

Але драйвери відрізняються від сервісів в багатьох важливих аспектах. Так, вони:

- взаємодіють з основними сервісами Windows і пристроями за допомогою своїх власних спеціальних інтерфейсів, які називаються DLL;
- засновані на моделі введення/виведення Windows, абсолютно іншій, чим модель, що використовується для сервісів і додатків;
- можуть безпосередньо взаємодіяти з компонентами режиму ядра. Драйвери режиму ядра виконуються повністю в режимі ядра. Але навіть драйвери UMDF іноді повинні взаємодіяти з базисними драйверами режиму ядра для обміну даними з пристроєм.

5.2. Особливості і прийоми програмування в режимі ядра

Програмування в режимі ядра має масу особливостей, для прикладників дуже незвичних, а для новачків досить складних. По-перше, для коду, що виконується в режимі ядра, має дуже велике значення його рівень IRQL, оскільки додаткам, що виконуються на високих рівнях IRQL, недоступні багато функцій, до яких мають доступ додатки низьких рівнів IRQL, і навпаки. Все це необхідно враховувати. По-друге, в режимі ядра є свої додаткові описувачі типів. Повний їх список можна знайти в заголовному файлі ntdef.h. Його зміст приблизно такий:

```
typedef unsigned char USHAR
typedef unsigned short USHORT
typedef unsigned long ULONG
.............
```

Це потрібно для уніфікації стилю класичних С – типів даних і нововведень – таких, як WCHAR (двобайтовий символ Unicode), LARGE_INTEGER (який, насправді, є об’єднанням) і так далі А також для уніфікації початкових кодів для 32-розрядних платформ і що насуваються 64-розрядних.

В початкових кодах драйверів часто зустрічаються макровизначення IN, OUT, OPTIONAL, які введені вони тільки для підвищення легкості читання початкового коду. OPTIONAL позначає необов’язкові параметри, наприклад, IN – параметри, що передаються всередину функції, OUT – відповідно навпаки. А ось IN OUT означає, що параметр передається всередину функції, а потім повертається з неї назад.

При програмуванні драйверів використовується ще один тип – NT_STATUS. Він включає інформацію про код завершення функції

(визначення цього типу можна подивитися у файлі ntdef.h). NT_STATUS є перевизначеним типом long integer. Ненегативні значення змінних цього типу відповідають успішному завершенню функції, негативні, – навпаки (файл NTSTATUS містить символьні позначення всіх кодів повернення). Повідомлення про вдале завершення має код 0 і символьне позначення STATUS_SUCCESS. Решта кодів повернення, відповідних різноманітним варіантам помилок, транслюються в системні коди помилок і передаються програмі. Для роботи з типом NT_STATUS існує декілька макровизначень (описані у файлі ntdef.h), наприклад NT_SUCCESS(), перевіряючий код повернення на успішність завершення.

Функції драйвера, за винятком DriverEntry, можуть називатися як завгодно, проте, існують певні “правила хорошого тону” при розробці драйверів, у тому числі і для іменування процедур: наприклад, всі функції, що відносяться до HAL, бажано передувати префіксом HAL і так далі. А імена типів даних і макровизначення в лістингах DDK написані суцільно заголовними буквами, що й необхідно роботи при розробці драйверів, це і справді у багато разів підвищує зручність роботи з лістингом.

Для подальшого розуміння основних відмінностей між програмуванням в режимі користувача і системному режимі, необхідно розглянути основні функції для роботи з пам’яттю і реєстром в режимі ядра.

Почнемо з функцій для роботи з пам’яттю, власне про пристрій і роботу з пам’яттю в Windows. Єдиний 4-х гігабайтний адресний простір пам’яті Windows (для 32-х розрядних версій Windows) ділиться на дві частини: 2 Гбайт для призначеного для користувача простору і 2 Гбайт для системного. 2 Гбайт системного простору доступні для всіх потоків режиму ядра.

Видів адрес в режимі ядра три: фізичні (реально вказують на область фізичної пам’яті), віртуальні (які перед використанням транслюються у фізичні), і логічні (використовуються HAL рівнем при спілкуванні з пристроями; він же і відповідає за роботу з такими адресами). Функції режиму ядра, що відповідають за виділення і звільнення віртуальної пам’яті, відрізняються від таких в режимі користувача. Також, знаходячись на рівні режиму ядра, стає можливим використовувати функції виділення і звільнення фізично безперервної пам’яті. Розглянемо всі ці функції детальніше:

1. PVOID ExAllocatePool (рівень IRQL, на якому може виконуватися ця функція, – < DISPATCH_LEVEL) – виділяє область пам’яті. Приймає два параметри: параметр POOL_TYPE, в якому міститься значення, що означає, якого типу область пам’яті потрібно виділити: PagedPool – сторінкова, NonPagedPool – несторінкова (в цьому випадку функцію можна викликати з будь-якого IRQL рівня). Другий параметр ULONG – розмір запрошуваної області пам’яті. Функція повертає покажчик на виділену область пам’яті, і NULL, якщо виділити пам’ять не вдалося.

2. VOID ExFreePool (IRQL<DISPATCH_LEVEL) – звільняє область пам’яті. Приймає параметр PVOID – покажчик на область пам’яті, що

звільняється. Якщо вивільняється несторінкова пам'ять, то дана функція може бути викликана з DISPATCH_LEVEL. Значення, що повертається, – void.

3. PVOID MmAllocateContiguousMemory (IRQL==PASSIVE_LEVEL) – виділяє фізично безперервну область пам'яті. Приймає два параметри. Перший параметр ULONG – розмір запитуваної області пам'яті, другий параметр PHYSICAL_ADDRESS, що означає верхню межу адрес для запитуваної області. Значення, що повертається: віртуальна адреса виділеної області пам'яті або NULL (при невдачі).

4. VOID MmFreeContiguousMemory (IRQL==PASSIVE_LEVEL) – звільняє фізично безперервну область пам'яті. Приймає єдиний параметр PVOID – покажчик на область пам'яті, виділену раніше з використанням функції MmAllocateContiguousMemory. Значення, що повертається, – void.

5. BOOLEAN MmIsAddressValid (IRQL<=DISPATCH_LEVEL) – робить перевірку віртуальної адреси. Приймає параметр PVOID – віртуальну адресу, що потребує перевірки. Функція повертає TRUE, якщо адреса "валідна", тобто присутня у віртуальній пам'яті, і FALSE – адреса відсутня.

6. PHYSICAL_ADDRESS MmGetPhysicalAddress (IRQL – будь-який) – визначає віртуальну фізичну адресу. Приймає параметр PVOID, що містить аналізовану віртуальну адресу. Значення, що повертається, – отримана фізична адреса.

Для подальшого розуміння розглянемо функції для роботи з реєстром, а саме функції доступу до реєстру, що надаються диспетчером введення/виведення, потім драйверні функції прямого доступу до реєстру, а також функції для роботи з реєстром – Zw~.

Драйверні функції, що надаються диспетчером введення/виведення:

1. IoRegisterDeviceInterface – дана функція реєструє інтерфейс пристрою. Диспетчер введення/виведення створює підрозділи реєстру для всіх зареєстрованих інтерфейсів. Після цього можна створювати і зберігати в цьому підрозділі потрібні драйверу параметри за допомогою виклику функції IoOpenDeviceInterfaceRegistryKey, яка повертає дескриптор доступу до підрозділу реєстру для зареєстрованого інтерфейсу пристрою.

2. IoGetDeviceProperty – дана функція запрошує з реєстру настановну інформацію про пристрій.

3. IoOpenDeviceRegistryKey – повертає дескриптор доступу до підрозділу реєстру для драйвера або пристрою по покажчику на його об'єкт.

4. IoSetDeviceInterfaceState – за допомогою даної функції можна вирішити або заборонити доступ до зареєстрованого інтерфейсу пристрою. Компоненти системи можуть діставати доступ тільки до дозволених інтерфейсів.

Драйверні функції для прямого доступу до реєстру:

1. RtlCheckRegistryKey – перевіряє, чи існує вказаний підрозділ усередині підрозділу, переданого першим параметром. Якщо він існує, то повертається STATUS_SUCCESS.

2. RtlCreateRegistryKey – створює підрозділ усередині розділу реєстру, вказаного другим параметром. Далі – все те ж саме, що і у RtlCheckRegistryKey.

3. RtlWriteRegistryValue – записує значення параметра реєстру. Перший параметр – куди пишемо, другий – в який підрозділ (якщо його немає, то він буде створений), а третій – який параметр створюємо.

4. RtlDeleteRegistryValue – видаляє параметр з підрозділу. Параметри ті ж самі, що і у RtlWriteRegistryValue (звичайно тільки з необхідними поправками).

5. RtlQueryRegistryValues – дана функція дозволяє за один виклик набути значень відразу декількох параметрів вказаного підрозділу.

До функцій для роботи з реєстром сімейства Zw~ входять:

1. ZwCreateKey – відкриває доступ до підрозділу реєстру. Якщо такого немає – створює новий. Повертає дескриптор відкритого об'єкту.

2. ZwOpenKey – відкриває доступ до існуючого підрозділу реєстру.

3. ZwQueryKey – повертає інформацію про підрозділ.

4. ZwEnumerateKey – повертає інформацію про вкладені підрозділи вже відкритого раніше підрозділу.

5. ZwEnumerateValueKey – повертає інформацію про параметри і їх значення відкритого раніше підрозділу.

6. ZwQueryValueKey – повертає інформацію про значення параметра у відкритому раніше розділі реєстру. Повнота інформації, що повертається, визначається третім параметром, що передається функції, який може приймати наступні значення: KeyValueBasicInformation, KeyValuePartialInformation і KeyValueFullInformation.

7. ZwSetValueKey – створює або змінює значення параметра у відкритому раніше підрозділі реєстру.

8. ZwFlushKey – примусово зберігає на диск зміни, зроблені у відкритих підрозділах функціями ZwCreateKey і ZwSetValueKey.

9. ZwDeleteKey – видаляє відкритий підрозділ з реєстру.

10. ZwClose – закриває дескриптор відкритого раніше підрозділу реєстру, заздалегідь зберігши зроблені зміни на диску.

Практично всі вище перераховані функції для роботи з реєстром повинні викликатися з рівня IRQL PASSIVE_LEVEL.

Отже, у всіх вище перерахованих функцій є маса нюансів в застосуванні. Проте функцій режиму ядра нітрохи не менше, ніж в режимі користувача. Тому, проведене порівняння відмінностей функцій режиму ядра, від таких в режимі користувача, дозволяє перейти до розгляду архітектури та структури драйверу.

5.3. Архітектура драйвера

Архітектура драйверів Windows спроектована для підтримки чотирьох ключових можливостей, а саме:

1. Модульна багаторівнева архітектура. Пристрій може обслуговувати декілька драйверів, організованих в стек.

2. Пакетний механізм введення/виведення. Всі запити обробляються в пакетах, якими передаються між системою і драйвером, і від одного до іншого драйвера в стеку.

3. Асинхронне введення/виведення. Драйвери можуть почати обробляти інші запити, не чекаючи завершення обробки поточного запиту.

4. Динамічне завантаження і вивантаження. Час життя драйвера прив'язаний до часу життя його пристрою. Драйвер вивантажується тільки після того, як всі зв'язані пристрої були видалені.

Розглянемо більш детально об'єкти та стек пристрою.

Запит введення/виведення, направлений підсистемою ядра пристрою, обробляється одним або декількома драйверами. Кожен драйвер має асоційований з ним об'єкт пристрою, який представляє участь драйвера в обробці запитів введення/виведення даного пристрою. Об'єкт пристрою – це структура даних, що містить покажчики на робочі процедури, які дозволяють менеджерові введення/виведення взаємодіяти з драйвером.

Об'єкти пристроїв організовані в стек пристрою, при цьому для кожного пристрою створюється окремий стек. Зазвичай під терміном "стек пристрою" розумівють стек об'єктів пристрою з їх зв'язаними драйверами. Але стек пристрою асоціюється тільки з одним пристроєм, тоді як набір драйверів може обслуговувати декілька стеків пристроїв. Набір драйверів іноді називається стеком драйверів.

Отже, до стеку пристрою входять такі компоненти:

1. *Драйвер шини і об'єкт фізичного пристрою* (PDO). Внизу стека знаходиться PDO, який асоційований з драйвером шини. Пристрої зазвичай підключаються до стандартних апаратних шин, таким як, наприклад, PCI або USB. Драйвер шини зазвичай управляє декількома апаратними пристроями, підключеними до фізичної шини. Наприклад, після установки драйвер шини перераховує пристрої, підключені до шини, і запрошує ресурси для цих пристроїв. PNP-менеджер використовує цю інформацію для виділення ресурсів кожному пристрою. Кожен пристрій має власний об'єкт PDO. PNP-менеджер ідентифікує драйвери для кожного пристрою і створює відповідний стек пристроїв вверху кожного об'єкту PDO.

2. *Драйвер функції і об'єкт функціонального пристрою* (FDO). Основним компонентом стека пристроїв є FDO, який асоціюється з драйвером функції. Драйвер функції перетворить абстракцію пристрою, створювану Windows, в справжні команди, необхідні для обміну даними з фізичним пристроєм. Він надає "верхню межу" (іншими словами, інтерфейс пристрою) для взаємодії з додатками і пристроями, і зазвичай визначає, яким чином драйвер реагує на зміни в стані PNP або енергоспоживання. "Нижня межа" драйвера пристрою обробляє взаємодію з пристроєм або іншими драйверами, такими як, наприклад, розташований нижче драйвер фільтру або драйвер шини.

3. *Драйвери фільтрів і об'єкти пристрою фільтру* (Filter Device Object, FIDO). Стек пристрою може мати декілька об'єктів пристроїв фільтру, які можуть розташовуватися навколо об'єкту FDO. З кожним об'єктом FIDO

асоціюється драйвер фільтру. Драйвери фільтрів не є обов'язковими, але часто присутні. Сторонні постачальники можуть забезпечувати стек пристрою драйверами фільтрів з метою надання йому додаткових можливостей, таким чином, підвищуючи його вартість. Зазвичай драйвер фільтру служить для модифікації деяких запитів введення/виведення, коли вони проходять через стек пристроїв, в значній мірі подібно до того, як аудіофільтр модифікує аудіопотік.

Наприклад, за допомогою драйверів фільтру можна зашифровувати або розшифровувати запити на читання і запис. Драйвери фільтру можна також застосовувати для виконання завдань, що не потребують модифікації запитів введення/виведення, наприклад, таких як відстежування запитів і надання інформації назад відстежуючому додатку.

Ці три типи об'єктів пристроїв відрізняються в деталях, але працюють по забезпеченню системи можливостями обробки запитів введення/виведення в основному однаково.

Шини комп'ютерних систем зазвичай організовані в декілька рівнів. До шини найнижчого рівня може бути підключена одна або декілька інших шин, а також індивідуальні пристрої. До цих шин, у свою чергу, теж можуть підключатися інші шини і незалежні пристрої, і так далі Тому більшість драйверів шин повинні грати дві ролі: драйвера функції для нижчележачией шини і перечислителя та менеджера шини для будь-яких підключених пристроїв.

При завантаженні системи PNP-менеджер починає роботу з шини найнижчого рівня і виконує такі операції:

1. Завантажує драйвер шини, який перераховує об'єкти PDO для всіх пристроїв, підключених до його фізичної шини, і запрошує ресурси для цих пристроїв.
2. Виділяє ресурси кожному пристрою.
3. Опитує свою базу даних, щоб визначити асоціювання драйверів з пристроями.
4. Створює стек пристроїв поверх кожного об'єкту PDO.
5. Запускає всі пристрої.
6. Опитує кожен пристрій на можливу наявність у нього об'єктів PDO для перерахування.
7. При необхідності повторює кроки 2-6.

Результати цього процесу представляються ієрархічною структурою, яка називається деревом пристроїв. Кожен стек пристрою представляється в дереві пристроїв вузлом, який називається devnode. Вузол devnode пристрою використовується PNP-менеджером для зберігання інформації про конфігурацію і відстежування пристрою. Основа дерева пристроїв – це абстрактний пристрій, що називається кореневим пристроєм.

Для виконання своїх завдань драйвери залежать від різних об'єктів і структур даних ядра. Ці об'єкти і структури даних управляються ядром

Windows та інкапсулюють різні системні ресурси, такі як пристрої, запити введення/виведення, потоки, події тощо.

Об'єкти управляються менеджером об'єктів і відстежуються методом підрахунку всіх посилань на них (reference count). Наприклад, об'єкт \KernelObjects \LowMemoryCondition – це стандартний подієвий об'єкт, який сигналізує про нестачу пам'яті. Менеджер об'єктів також управляє процесами і потоками. Багато об'єктів можуть мати дескриптори, за допомогою яких додатки можуть маніпулювати об'єктом.

5.4. Переривання та рівні IRQL

Для ефективного реагування на апаратні події операційні системи забезпечуються механізмом, які називаються перериванням. Кожній апаратній події призначається переривання. Наприклад, одне переривання може ініціювати запуск годинника і планувальника, інше – драйвери клавіатури і таке інше. Коли відбувається апаратна подія, Windows доставляє відповідне переривання процесору.

Коли процесор отримує переривання, система не створює новий потік для його обслуговування. Замість цього Windows перериває існуючий потік на короткий час, необхідний для обробника переривання, щоб обслужити переривання. По завершенню обслуговування переривання система повертає контроль над потоком його оригінальному власникові.

Переривання необов'язково ініціюються безпосередньо апаратними подіями. Windows також підтримує програмні переривання у формі процедур відкладеного виклику (DPC). Windows планує процедуру DPC, доставляючи переривання відповідному процесору. Драйвери режиму ядра використовують процедури DPC для обробки аспектів апаратних переривань, що потребують великої витрати часу.

З кожним перериванням асоціюється рівень, що називається рівнем IRQL (Interrupt ReQuest Level, рівень запиту на переривання), який визначає велику частину програмування в режимі ядра. Система використовує різні значення рівнів IRQL, щоб в першу чергу забезпечити час обробки найбільш важливих і залежних переривань. По рівню IRQL виконується процедура обслуговування, призначена відповідному перериванню. Коли відбувається переривання, система знаходить відповідну процедуру обслуговування і призначає її на виконання процесору.

Поточний рівень IRQL процесора визначає, коли процедура обслуговування запускається на виконання і чи може вона переривати виконання потоку, що виконується процесором в даний час. В цьому відношенні основне правило полягає в тому, що пріоритет має найвищий рівень IRQL. Коли процесор отримує переривання, відбувається наступне:

1. Якщо рівень IRQL переривання вищий, ніж рівень IRQL процесора, система підвищує рівень IRQL процесора до рівня IRQL переривання.

Виконання поточного коду на даному процесорі припиняється до тих пір, поки не завершиться виконання процедури обслуговування і рівня IRQL процесора не повернеться до свого первинного значення. Виконання процедури обслуговування може бути, у свою чергу, перервано іншою процедурою обслуговування, що має вищий рівень IRQL.

2. Якщо рівень IRQL переривання такий же, як і рівень IRQL процесора, процедура обслуговування повинна чекати завершення виконання всіх попередніх процедур з таким же рівнем IRQL.

Після цього виконується процедура поточного переривання. Її виконання може бути перерване перериванням з ще вищим рівнем IRQL.

3. Якщо рівень IRQL переривання, що прибуло, нижче поточного рівня IRQL процесора, її процедура обслуговування повинна чекати завершення виконання всіх попередніх процедур з вищим рівнем IRQL.

Ці варіанти виконання процедур обслуговування переривань з різними рівнями IRQL дійсні, коли вони виконуються на одному процесорі. Але сучасні системи зазвичай обладнані двома або більше процесорами. Цілком можлива ситуація, коли процедури драйвера з різними рівнями IRQL виконуються одночасно на різних процесорах. Якщо процедури належним чином не синхронізовані, це може викликати взаємоблокування.

Рівні IRQL – зовсім інше поняття, ніж пріоритети потоків. Пріоритети потоків використовуються системою для управління плануванням потоків під час штатної роботи системи. Переривання ж за визначенням є щось що виходить за рамки штатної роботи і вимагає як можна швидкого обслуговування. Процесор повертається до нормальної обробки процесів тільки після завершення обслуговування всіх переривань, що чекають на виконання.

Кожному рівню IRQL призначається чисельне значення. Але оскільки ці значення є різними для різної процесорної архітектури, рівням IRQL зазвичай привласнюються ще й імена. Драйвери використовують тільки деякі зі всіх наявних рівнів IRQL; решта рівнів IRQL зарезервована для використання системою.

Розглянемо найбільш широко використовувані драйверами рівні IRQL, починаючи з рівня з найнижчим значенням:

- PASSIVE_LEVEL – це найнижчий рівень IRQL. Він призначається за умовчанням штатній обробці потоків. Це єдиний рівень IRQL, який не асоційований з будь-яким перериванням. Всі додатки призначеного для користувача режиму виконуються з рівнем IRQL = PASSIVELEVEL так само, як і низькопріоритетні процедури драйверів. Процедури, що виконуються з рівнем IRQL = PASSIVELEVEL, можуть звертатися до всіх сервісів ядра Windows;
- DISPATCHLEVEL – це найвищий рівень IRQL, який призначається програмним перериванням. Процедури DPC і інші процедури підвищеного пріоритету виконуються з рівнем IRQL = DISPATCH_LEVEL. Процедури, що виконуються з рівнем IRQL = DISPATCHLEVEL, можуть мати доступ тільки до обмеженої підмножини базових сервісів Windows;

– DIRQL – цей рівень IRQL вищий, ніж рівень DISPATCHLEVEL, і є самим вищим рівнем IRQL, з яким драйверам зазвичай доводиться мати справу. Насправді, це збірний термін для сукупності рівнів IRQL, що асоціюються з апаратними перериваннями, які також називаються рівнями IRQL пристроїв (device IRQL, DIRQL). PNP-менеджер призначає рівень DIRQL кожному пристрою і передає його відповідному драйверу при запуску. Зазвичай знати точний рівень не так вже і важливо. Для більшості завдань потрібно лише знати, що виконання відбувається на рівні DIRQL і що процедура обслуговування блокує виконання на даному процесорі практично будь-якого іншого завдання. Процедури, що виконуються з рівнем DIRQL, можуть мати доступ тільки до дуже невеликої підмножини базових сервісів Windows.

Однією з основ програмування якісних драйверів є відстежування рівня IRQL, з яким виконуються або повинні виконуватися ваші процедури. Недбалість в цій області може призвести до повного збою в системі. Набір WDK містить інструменти, такі як, наприклад, Driver Verifier і SDV, що допомагають виловлювати такі помилки.

Рівні IRQL, з якими можуть виконуватися драйверні процедури, і рівні IRQL, з яких можна викликати процедури DDI, описуються в документації WDK.

5.5. Структура драйвера

Отже, драйвер можна представити просто як набір процедур, що періодично викликаються зовнішніми програмами. Не дивлячись на те, що процедури драйверів для різних пристроїв сильно відрізняються, є загальна структура і загальні функції для всіх драйверів. Головні з них розглянемо нижче.

Вхідна точка будь-якого драйвера – функція DriverEntry, яка фактично грає ту ж саму роль для драйвера, що і main для програми на мові С. Ця функція викликається при завантаженні драйвера (неважливо, чи завантажується він динамічно або при запуску системи). Дана функція виконує деякі дії, потрібні для нормальної роботи драйвера (наприклад, реєструє в спеціальному масиві адреси решти всіх функцій драйвера, щоб диспетчер введення – виведення міг викликати їх по цих адресах). Якщо це не WDM драйвер, то в цій функції відбувається локалізація устаткування, що обслуговується, виділення і/або підтвердження використовуваних апаратних ресурсів, видача видимих для системи імен всім знайденим обслуговуваним пристроям і так далі. WDM драйвера цю роботу перекладають на функцію AddDevice. Функція DriverEntry може викликатися з рівня IRQL == PASSIVE_LEVEL. Функція повертає значення типу NTSTATUS, і приймає два параметри: адреса об’єкту драйвера (PDRIVER_OBJECT) і шлях в реєстрі до підрозділу драйвера (PUNICODE_STRING). Отримавши від диспетчера введення/виведення

покажчик на структуру DRIVER_OBJECT, драйвер повинен заповнити в ній деякі поля:

1. Поле DriverUnload – для реєстрації власної функції Unload, що викликається при вивантаженні драйвера. Ця функція викликається тільки при динамічному вивантаженні драйвера (тобто системи, що відбулася не в результаті завершення роботи). Legacy драйвера в цій функції виконують повне звільнення всіх зайнятих драйвером системних ресурсів. WDM драйвера виконують таке звільнення у функції RemoveDevice при видаленні кожного пристрою (якщо драйвер обслуговує декілька пристроїв). Функція Unload викликається з рівня IRQL PASSIVE_LEVEL, приймає єдиний параметр (PDRIVER_OBJECT) – покажчик на об'єкт драйвера, і повертає void.

2. Поле DriverStartIo – для реєстрації власної функції StartIo. Реєстрація функції StartIo потрібна для участі в System Queuing – створенні черг необроблених запитів системними засобами, на відміну від DriverQueuing – коли те ж саме реалізується засобами самого драйвера.

3. Поле AddDevice в підструктурі DriverExtension використовується для реєстрації WDM драйвером своєї процедури AddDevice.

4. Поле MajorFunction використовується для реєстрації драйвером точок входу в своїй робочій процедурі.

Бувають ситуації, коли при первинному завантаженні драйвер не може до кінця закінчити процедуру ініціалізації (наприклад, якщо необхідні будь-які системні об'єкти або інші драйвера, ще не завантажені). В цьому випадку драйвер реєструє свою процедуру для завершення ініціалізації пізніше. Реєстрація цієї процедури виконується викликом IoRegisterDriverReinitialization з рівня IRQL PASSIVE_LEVEL, що приймає наступні параметри: покажчик на об'єкт драйвера (PDRIVER_OBJECT), покажчик на процедуру реініціалізації, що надається драйвером (PDRIVER_REINITIALIZE) і контекстний покажчик, який отриманий за допомогою реєстрованої функції при виклику, і повертає значення void.

Якщо вивантаження драйвера відбувається в результаті завершення роботи системи, то функція Unload не викликається. Зрозуміло, що при виключенні системи можна не піклуватися про звільнення пам'яті і так далі. Тому викликається функція Shutdown, яка просто надає драйверу можливість залишити пристрій в прийнятному стані спокою.

Інколи виникають ситуації, коли драйверу необхідно буде отримати управління при краху системи. В цьому випадку йому потрібно зареєструвати callback-процедуру Bugcheck. Якщо драйвер правильно виконав реєстрацію цієї функції, то він буде викликаний під час виконання crash-процесу.

Вище перераховані основні функції драйвера для його завантаження і вивантаження. Тепер необхідно розглянути робочі процедури драйвера такі як: обслуговування введення/виведення, що включає процедури передачі даних і обслуговування переривань, callback-процедури для синхронізації доступу до об'єктів і деякі інші.

Якщо драйверу потрібно обробляти призначений для користувача виклик CloseHandle, то він повинен мати обробник CloseDispatch.

Розглянемо процедури передачі даних – це обробник призначених для користувача викликів ReadFile, WriteFile і DeviceIoControl.

Процедура StartIo розглядалася вище, тому перейдемо до процедури обслуговування переривань (ISR – Interrupt Service Routine). Дана процедура викликається диспетчером переривань ядра (Kernel's Interrupt Dispatcher) при кожній генерації переривання пристроєм, і вона зобов'язана повністю обслужити це переривання.

Тепер розглянемо callback-процедури синхронізації доступу до об'єктів. Спочатку необхідно розібратися в чому відмінності принципів синхронізації доступу до об'єктів в режимі користувача та режимі ядра. Наприклад, в режимі користувача, якщо будь-який потік звертається до об'єкту, вже зайнятого іншим потоком, то він (перший потік) запросто може бути заблокований до кращих часів. В режимі ядра такий підхід "заморожування" потоків неприйнятний, тому існує особлива техніка синхронізації, яка полягає в наступному: коли будь-який потік звертається до об'єкту, вже зайнятого іншим потоком, то він залишає свій запит в черзі. Якщо драйвер заздалегідь зареєстрував особливу callback-функцію, то диспетчер введення/виведення при звільненні необхідного ресурсу повідомить про це драйвер, викликавши callback-функцію. Таким чином, забезпечується гарантія, що будь-який запит до ресурсу, навіть якщо він (драйвер) повинен чекати в тому, щоб повідомити про затримку в обробці і розміщення запиту в черзі. До функцій, які це реалізують, відносять:

- IoAllocateController, яка використовується для синхронізації доступу до контролера;
- AdapterControl, яка використовується для синхронізації доступу до DMA каналів;
- SynchCritSection, яка використовується для коректного звернення до ресурсів, точніше – ця функція дозволяє коду з низьким рівнем IRQL зробити роботу при рівні DIRQL пристрою без побоювання виникнення конфлікту з ISR.

Також, можна зазначити таймерні процедури, які потрібні для драйверів, що відслідковують точний відлік часу в системі:

- IoTimer – процедура, яка найчастіше використовується;
- IoCompletion – це процедура, що дозволяє WDM драйверу, який працює усередині багатошарової драйверної структури, отримувати повідомлення про завершення обробки IRP запиту, направленого до драйвера нижнього рівня;
- CancelRoutine. Якщо драйвер зареєструє callback-процедуру за допомогою виклику IoSetCancelRoutine, то диспетчер введення/виведення зможе повідомити його про видалення IRP запитів, які знаходяться в очікуванні обробки, що може знадобитися диспетчеру, якщо додаток користувача,

що ініціював ці IRP запити, несподівано завершить свою роботу після зняття завдання диспетчером завдань.

5.6. Найпростіший драйвер

Перейдемо до розгляду початкового тексту простих драйверів. Всі початкові тексти драйверів будуть оформлятися у вигляді *.bat файлу, який, насправді, є комбінацією *.bat і *.asm файлів, але має розширення .bat:

```
;@echo off
;goto make

.386                        ; початок початкового тексту драйвера

  ; решта коду драйвера

end DriverEntry             ; кінець початкового тексту драйвера

:make
\masm32\bin\ml /nologo /c /coff driver.bat
\masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:driver.sys /subsystem:native driver.obj

del driver.obj

echo.
pause
```

Якщо такий файл, що “самокомпілюється”, запустити, то відбудеться наступне. Перші дві команди закомментовані, тому, вони ігноруються компілятором masm, але приймаються командним процесором, який, у свою чергу, ігнорує символ “крапка з комою”. Управління передається на мітку :make, за якою знаходяться інструкції для компілятора і компонувальника. Все, що знаходиться за директивою асемблера end, ігнорується компілятором masm. Таким чином, весь текст між командою goto make і міткою :make, ігнорується командним процесором, але приймається компілятором masm. А все, що зовні (включаючи команду goto make і мітку :make), ігнорується компілятором masm, але приймається командним процесором. Цей метод надзвичайно зручний, оскільки початковий текст “пам’ятає” з якими параметрами його потрібно компілювати. Тому будемо застосовуватиму таку техніку в початкових текстах драйверів, а в початкових текстах програм управління, користуватимуся звичайним методом.

Параметри компоновки мають наступне значення:

`/driver` – вказує компонувальнику, що потрібно сформувати файл драйвера режиму ядра Windows NT;

`/base:0x10000` – встановлює зумовлену адресу завантаження образу

драйвера рівною 10000h;

`/align:32` – пам’ять режиму ядра – дорогоцінний ресурс. Тому, файли драйверів мають “дрібніше” вирівнювання секцій;

`/out:driver.sys` – за умовчанням компонувальник проводить файли з розширенням .exe. За наявності ключа /dll файл матиме розширення .dll. Нам потрібно отримати файл з розширенням .sys;

`/subsystem:native` – у PE-заголовку є поле, що вказує завантажувачу образу виконуваного файлу, для якої підсистеми цей файл призначений: Win32, POSIX або OS/2. Підсистема Win32 автоматично запускається при завантаженні системи. Якщо ж запускається файл, призначений для функціонування, наприклад, в підсистемі POSIX, то спочатку ОС запускає саму підсистему POSIX. Таким чином, за допомогою цього ключа можна вказати компонувальнику, яка підсистема необхідна. Коли компілюємо *.exe або *.dll, то указуємо під цим ключем значення windows, яке означає, що файлу потрібна підсистема Win32. Драйверу взагалі не потрібна жодна з підсистем, оскільки він працює в природному (native) для самої ОС середовищі.

Найпростіший драйвер режиму ядра може мати вигляд:

```
 ;@echo off
 ;goto make

 ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 ;simplest - найпростіший драйвер режиму ядра
 ;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

 .386
 .model flat, stdcall
 option casemap:none

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;              Ф А Й Л И, Щ О  В К Л Ю Ч А Ю Т Ь С Я
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

 include \masm32\include\w2k\ntstatus.inc
 include \masm32\include\w2k\ntddk.inc
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
;                             К О Д
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

 .code

;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
```

```
;                    DriverEntry
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

 DriverEntry proc pDriverObject:PDRIVER_OBJECT,
pusRegistryPath:PUNICODE_STRING

     mov eax, STATUS_DEVICE_CONFIGURATION_ERROR
     ret
 DriverEntry endp
;::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 end DriverEntry

 :make
 \masm32\bin\ml /nologo /c /coff simplest.bat
 \masm32\bin\link /nologo /driver /base:0x10000 /align:32
/out:simplest.sys /subsystem:native simplest.obj

 del simplest.obj

 echo.
     pause
```

Як і у будь-якого іншого здійснюваного модуля, у драйвера повинна бути точка входу, на яку система передасть управління після завантаження драйвера в пам’ять. Як і вважається в програмі на асемблері, точкою входу є перша інструкція, позначена міткою вказаною в директиві end. Тому тут, як і в текстах на мові С, це DriverEntry, який оформлений у вигляді процедури, що приймає два параметри. Ім’я процедури, природно, може бути будь-яким. Прототип DriverEntry виглядає так:

```
DriverEntry proto DriverObject:PDRIVER_OBJECT,
RegistryPath:PUNICODE_STRING
```

Типи даних PDRIVER_OBJECT і PUNICODE_STRING визначені у файлах \include\w2k\ntddk.inc і \include\w2k\ntdef.inc відповідно:

```
 PDRIVER_OBJECT   typedef PTR DRIVER_OBJECT
 PUNICODE_STRING  typedef PTR UNICODE_STRING
```

де pDriverObject – покажчик на об’єкт тільки створеного драйвера.

Windows є об’єктно-орієнтованою системою. Тому, поняття об’єкт розповсюджується на все, що тільки можна, і що не можна теж. І об’єкт “драйвер” не є виключенням. Завантажуючи драйвер, система створює об’єкт “драйвер” (driver object), що представляє для неї образ драйвера в пам’яті. Через цей об’єкт система управляє драйвером. Об’єкт драйвера є звичайною структурою даних типу DRIVER_OBJECT (визначена в \include\w2k\ntddk.inc). Деякі поля цієї структури заповнює система, деякі доведеться заповнювати нам самим. Звертаючись до цієї структури, система і управляє драйвером. Отже, як стало зрозуміло, першим параметром, що передається у функцію DriverEntry,

якраз і є покажчик на цю саму структуру (або користуючись об'єктно-орієнтованою термінологією – об'єкт "драйвер").

5.7. Поради щодо програмування в режимі ядра

Для гарного програмування в режимі ядра слід притримуватись декількох основних рекомендацій:

1. *Виділення пам'яті*. Зробіть драйвер здатним обробляти ситуації браку пам'яті. Критично важливо уникнути збоїв драйверів Windows в ситуаціях, коли вони не можуть виділити пам'ять. Якщо для деяких аспектів роботи драйвера абсолютно необхідно мати пам'ять, що виділяється при його ініціалізації і зберігає покажчик на нього для подальшого використання. Виділяйте тегову пам'ять, щоб полегшити відладку витоку пам'яті.

2. *Використання спін-блокувань*. Не дозволяйте процедурам, що мають спін-блокування, звертатися до коду або даних в сторінковій пам'яті. Навіть процедури, які зазвичай виконуються на рівні IRQL = PASSIVELEVEL, виконуються на рівні IRQL = DISPATCH LEVEL, коли вони утримують спін-блокування. Для виявлення випадків входу в сторінковий код з неприпустимо високими пріоритетами використовуйте в цьому коді макрос paged_code ().

Не утримуйте спін-блокування довше, ніж це необхідно. Якщо процедура А утримує заблокований спін-блокований ресурс в той час, як процедура В також намагається отримати спін-блокування, то процедура В "обертатиметься" в циклі очікування доти, доки спін-блокування не стане доступним. Доки процедура В "обертається" в очікуванні, на даному процесорі не може виконуватися жоден код з рівнем IRQL = PASSIVE_LEVEL.

Дотримуйтеся обережності з отриманням і звільненням спін-блокувань. Якщо процедура має спін-блокування і, не звільнивши її, намагається отримати її знову, виникає взаємоблокування.

Звільняйте множинні спін-блокування в порядку, зворотному тому, в якому вони були отримані. Розглянемо наступний сценарій. Процедура драйвера отримує спін-блокування А, отримує спін-блокування В, звільняє спін-блокування А і звільняє спін-блокування В. Цей порядок отримання і звільнення спін-блокувань створює інтервал, в якому один потік може мати спін-блокування А, але не може отримати спін-блокування В, тоді як інший потік має спін-блокування В, але не може отримати спін-блокування А, внаслідок чого виникає стан взаємоблокування.

При використанні Driver Verifier включайте опцію виявлення взаємоблокувань. Функція виявлення взаємоблокувань інструменту Driver Verifier допоможе у верифікації та відладці використання спін-блокувань в коді, що розробляється.

3. *Управління помилками сторінок*. Пам'ять для певного типу завдань необхідно виділяти з відповідного пулу. Визначите, з якого пулу потрібно виділити пам'ять: із сторінкового або несторінкового.

При використанні Driver Verifier включайте опцію Force IRQL Checking. Це допоможе вам виявити багато проблем, пов'язаних із сторінковою пам'яттю.

4. *Звернення до пам'яті в режимі користувача.* Не довіряйте нічому, що поступає з режиму користувача. Завжди перевіряйте розмір всіх отриманих буферів та даних, що містяться в них, на цілісність і безпеку. Для прямого введення/виведення або методу введення/виведення NEITHER перевіряйте достовірність будь-яких параметрів, заздалегідь скопіювавши їх в пам'ять ядра, щоб після перевірки додаток не міг змінити їх.

У процедурах режиму ядра не обмежуйтеся простим розйменуванням покажчиків призначеного для користувача режиму. В кращому разі результатом цього буде просто незрозумілість, а в гіршому – порушення безпеки системи або ж виникнення її збою. Щоб розйменувати покажчик режиму користувача, необхідно, щоб процедура виконувалася в належному контексті процесу. Також потрібно прозондувати і заблокувати пам'ять за допомогою процедур DD1, які викликають виключення у разі декількох невдалих перевірок достовірності адреси. Будь-яку спробу доступу до покажчика режиму користувача необхідно інкапсулювати в блок структурної обробки виключень (наприклад, таких як блок try/_except). Таким чином, можна уловити виключення доступу до пам'яті, які можуть виникнути, якщо в процесі звернення до адреси додаток робить його недійсним.

5. *Блокування потоків.* Дотримуйтеся обережності з блокуванням потоків. Процедура, що виконується з підвищеним рівнем IRQL, не дозволяє виконуватися на даному процесорі жодному іншому коду, за винятком коду з ще вищим рівнем IRQL.

6. *Верифікація драйверів.* Користуйтеся інструментами Driver Verifier і KMDF Verifier. Починайте використовувати Driver Verifier і KMDF Verifier одразу ж після успішного завантаження драйвера. За допомогою Driver Verifier можна уловити помилки, які важко виявити при нормальній роботі, такі як, наприклад, використання коду або даних в сторінковій пам'яті на рівні IRQL > DISPATCHLEVEL.

Після компіляції перевіряйте код за допомогою інструментів PRE/ast і SDV. За допомогою цих інструментів статичного аналізу помилки в коді можна уловити на ранніх стадіях розробки, до того, як їх стане важко і складно усунути.

7. *Використання макросів.* Застосовуйте макрос verify_is_irqlj?assive_level() на початку всіх процедур, які повинні виконуватися на рівні IRQL = PASSIVELEVEL. У драйверах KMDF за допомогою макросу verify_is_irql_passive_levelo можна упевнитися в тому, що процедура виконується на рівні IRQL = PASSIVELEVEL. Якщо рівень IRQL вище значення PASSIVELEVEL, макрос генерує налагоджувальне повідомлення.

Застосовуйте макрос unreferenced_parameter для всіх параметрів, що не використовуються процедурою. Він відключає повідомлення компілятора про параметри, які не використовувались. Макрос unreferenced_parameter

визначений в стандартному заголовному файлі WDK Ntdef.h, який підключається у файлі Ntddk.h.

Висновки

- Драйвери повинні надавати зв'язок та контролювати взаємодію між застосуваннями і пристроєм, проте їх робота досить схожі із роботою сервісів. До основних завдань драйверів відносять: досить виконуються у фоновому режимі, окремо від процесів та доступні багатьом користувачам, мають довгий час життя, відповідають на зовнішні запити введення/виведення тощо.
- Програмування в режимі ядра має масу особливостей, для прикладників дуже незвичних, а для новачків досить складних. Тому для розробника драйвера обов'язково необхідно знати, що для коду, який виконується в режимі ядра, дуже велике значення має його рівень IRQL, та додаткові описувачі типів, що є в режимі ядра.
- Архітектура драйверів Windows включає такі компоненти: драйвер шини і об'єкт фізичного пристрою, драйвер функції і об'єкт функціонального пристрою, драйвери фільтрів і об'єкти пристрою фільтру.

Контрольні запитання та завдання

1. Структура драйвера.
2. Функції підтримки ядра.
3. Операції з пам'яттю в режимі ядра.
4. Керування розміщенням коду драйвера в пам'яті.
5. Робота з рядками, файлами та системний реєстром в режимі ядра.
6. Огляд функцій для роботи з посиланнями на об'єкти Windows.

Тести для самоконтролю

1. PDO – це…
    a. процес створення об'єктів драйвера
    b. фізичний об'єкт пристрою
    c. ідентифікатор пристрою
    d. функція драйвера
2. FDO – це…
    a. процес створення об'єктів драйвера
    b. фізичний об'єкт пристрою
    c. функція драйвера
    d. функціональний об'єкт пристрою

3. Як називається колекція об’єктів пристроїв і зв’язаних драйверів, які обробляють взаємодію з певним пристроєм?
    a. стек пристроїв
    b. адресний простір
    c. контекст пристрою
    d. регістр пристрою

4. Працюючи на рівні PASSIVE_LEVEL (0), потік може отримати від операційної системи Windows XP згоду на рівень IRQL. Якому числу буде дорівнювати цей рівень IRQL?
    a. 2
    b. 6
    c. 10

Теми для самостійного опрацювання

1. Об’єкти та структури даних ядра.
2. Стек ядра.
3. Застосування допоміжних процедур режиму ядра в драйверах KMDF.
4. Синхронізаційні об’єкти ядра.
5. Функції для роботи із Системним Реєстром.

Список використаної літератури

1. *Комиссарова В.* Программирование драйверов для Windows / В. Комиссарова. – СПб.: БХВ-Петербург, 2007. – 256 с.
2. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
3. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.
4. *Пирогов В. Ю.* Ассемблер для Windows / В. Ю. Пирогов. – М.: Издатель Молгачева С. В., 2002. – 552 с.
5. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО “Бином-Пресс”, 2004. – 480 с.
6. *Сорокина С. И.* Программирование драйверов и систем безопасности : учебное пособие / С. И. Сорокина, А. Ю. Тихонов, А. Ю. Щербаков. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2002. – 256 с.
7. *Хоглунд Г.* Рукиты: внедрение в ядро Windows / Г. Хоглунд, Дж. Баталер. – СПб.: Питер, 2007. – 285 с.
8. *Шрайбер С.* Недокументированный возможности Windows 2000 : Библиотека программиста / Свен Шрайбер. – СПб.: Питер, 2002. – 544 с.

РОЗДІЛ 6. СТРУКТУРА LEGACY-ДРАЙВЕРА ТА ЙОГО ОСНОВНІ ПРОЦЕДУРИ

План

- ✦ Призначення головних функцій legacy-драйвера
- ✦ Структура та призначення IRP-пакетів
- ✦ Функціонування робочих процедур драйвера

В даному розділі для розуміння того, як драйвер взаємодіє з іншими компонентами ОС, будуть детально розглядатися будова та призначення функцій legacy-драйвера та структура й призначення IRP-пакетів.

6.1. Призначення головних функцій legacy-драйвера

Драйвера під Windows діляться на два типи: legacy (застарілий) і WDM (PNP). Legacy драйвери (інакше називаються як “драйвери в стилі NT”) надзвичайно криво працюють (якщо працюють взагалі) під Windows 98, не працюють з PNP пристроями, та зате можуть користуватися старими функціями HalGetBusData, HalGetInterruptVector тощо, але при цьому не мають підтримки зі сторони шинних драйверів. В деяких випадках краще за написання legacy драйвера нічого не придумати. Тому приділимо особливу увагу даному драйверу.

Для того щоб створити драйвер, який можна запустити і зупинити достатньо реалізувати дві функції DriverEntry, DriverUnload. Для того щоб драйвер приймав участь в процедурах введення/виведення, потрібно організувати обробку IRP пакетів. Розглянемо данні функції більш детально.

6.1.1. Процедура DriverEntry

Процедура DriverEntry присутня в будь-якому драйвері і має дане стандартне ім’я. Legacy драйвери виконують в DriverEntry більшу роботу, ніж WDM драйвера – останні відкладають частину роботи по ініціалізації пристрою до моменту виявлення пристрою системою, коли буде викликана процедура AddDevice. Для подальшого розуміння параметри виклику функції DriverEntry подамо в таблиці 6.1.

Отримавши від диспетчера введення/виведення покажчик на структуру DRIVER_OBJECT, драйвер повинен заповнити в ній певні поля, а саме:

- поле pDriverObject->DriverUnload – для реєстрації власної функції Unload, яка викликається при вивантаженні драйвера;
- поле pDriverObject->DriverStartIo – для реєстрації власної функції StartIo, яка необхідна для організації обробки черги необроблених запитів System Queuing;
- поле pDriverObject->DriverExtension->AddDevice – в структурі розширення об'єкту драйвера DRIVER_EXTENSION, в якому WDM драйвер реєструє свою процедуру AddDevice;
- у масиві pDriverObject->MajorFunction[IRP_MJ_Xxx] драйвер реєструє точки входу у власні робочі процедури

Таблиця 6.1

Параметри виклику функції DriverEntry

Параметри	Опис
IN PDRIVER_OBJECT pDriverObject	Адреса об'єкту драйвера
IN PUNICODE_STRING pRegistryPath	Шлях в реєстрі до підрозділу драйвера
Значення, що повертається	STATUS_SUCCESS STATUS_XXX код помилки

Реєстрація робочих процедур буде виконана у вигляді:

```
DriverObject->MajorFunction[IRP_MJ_READ]= ReadWrite_IRPhandler;
DriverObject->MajorFunction[IRP_MJ_WRITE]=ReadWrite_IRPhandler;
DriverObject-
>MajorFunction[IRP_MJ_DEVICE_CONTROL]=DeviceControlRoutine;
```

Тут всі запити приводимуть до виклику функції myPassIrpDown, яка займається лише передачею запитів нижнім драйверам в стеку WDM-драйверів. Виняток становитиме функція-обробник IOCTL запитів, які потраплятимуть у функцію DeviceControlRoutine фільтр-драйвера.

Крім реєстрації функцій, процедура DriverEntry драйвера "в стилі NT" може виконувати наступну роботу:

1. DriverEntry виявляє апаратне забезпечення, яке драйвер контролюватиме. Це апаратне забезпечення виділяється драйверу, тобто позначається як те, що знаходиться під управлінням даного драйвера.

2. Якщо драйвер управляє багатокомпонентним (multiunit) або багатофункціональним контролером, використовується IoCreateController для створення об'єкту контролера, після чого ініціалізується структура розширення контролера.

- виконує виклик IoCreateDevice для створення об'єкту пристрою для кожного фізичного або логічного пристрою під управлінням даного драйвера, в процесі ініціалізується структура розширення пристрою для кожного створеного об'єкту пристрою. Рекомендується після цього виклику явно встановити прапори (поле Flags в об'єкті пристрою), що описують способи буферизації, який використовується даним пристроєм;

- створені пристрої повинні стають видимими для додатків режиму користувача шляхом виконання виклику IoCreateSymbolicLink;
- пристрій підключається до об'єкту переривань. У випадку, якщо ISR процедура потребує використання об'єкту DPC, то він створюється й ініціалізується на цьому етапі;
- кроки з 3 по 5 повторюються для кожного фізичного або логічного пристрою, що опрацьовується драйвером даного драйвера.

3. У випадку успішного завершення, функція DriverEntry повинна повернути диспетчеру вводу/виводу значення STATUS_SUCCESS.

У випадку, якщо процедура DriverEntry виявила, що деякі дії, які повинна виконати як частина підготовки до запуску драйвера, не можуть бути виконані зараз, а лише в майбутньому (після завантаження деяких інших драйверів, наприклад), це стає можливим, якщо DriverEntry повинна зареєструвати власну процедуру, яка завершить ініціалізацію трохи пізніше. Реєстрація такої процедури виконується системним викликом IoRegisterDriverReinitialization (див. табл. 6.2).

Таблиця 6.2

Параметри системного виклику IoRegisterDriverReinitialization

Параметри	Реєструє функцію драйвера для відкладеної ініціалізації
IN PDRIVER_OBJECT pDriverObject	Покажчик на об'єкт драйвера
IN PDRIVER_REINITIALIZE DriverReinitializationRoutine	Покажчик на процедуру реініціалізації, що надається драйвером (див. табл. 6.3)
IN PVOID Context	Контекстний покажчик, який отримає реєстрована функція при виклику
Значення, що повертається	Void

6.1.2. Процедура Unload

Як правило, одного разу завантажений драйвер залишається в системі до перезавантаження. Для того, щоб зробити драйвер вивантажуваним, необхідно написати і зареєструвати процедуру вивантаження Unload (див. табл. 6.3). Диспетчер вводу/виводу потім проводить виклик цієї процедури у момент ручного або автоматичного вивантаження драйвера – якраз перед видаленням драйвера з пам'яті.

Таблиця 6.3

Опис прототипу функції Unload

VOID Unload	IRQL == PASSIVE_LEVEL
Параметри	**Виконує завершуючі дії**
IN PDRIVER_OBJECT pDriverObject	Покажчик на об’єкт драйвера
Значення, що повертається	Void

Хоча дії процедури Unload можуть змінюватися від драйвера до драйвера, загальними є наступні кроки, характерні більш для legacy драйверів:

1. Для деяких типів апаратури необхідно зберегти її розташування в системному реєстрі. При подальшому завантаженні драйвера ці дані можуть бути використані в процедурі DriverEntry. Скажімо, драйвер принтера може зберегти останнє значення дозволу друку.

2. Якщо переривання дозволені для пристрою, що обслуговується, то процедура вивантаження повинна заборонити їх і провести відключення від об’єкту переривань. Ситуація, коли пристрій породжуватиме запити на переривання в той час, як об’єкт переривань не існує, неминуче приведе до краху системи.

3. Символьне посилання повинне бути видалене з простору імен, видимого додатками користувача. Це виконується за допомогою виклику IoDeleteSymbolicLink.

4. Об’єкт пристрою повинен бути видалений викликом IoDeleteDevice.

5. У випадку, якщо драйвер управляє багатокомпонентним контролером, необхідно повторити кроки 3 і 4 для кожного пристрою, підключеного до контролера, а потім необхідно видалити сам об’єкт контролера за допомогою виклику IoDeleteController.

6. Слід виконати звільнення пам’яті, виділеної драйверу, у всіх типах оперативної пам’яті.

Оскільки процедура Unload виконується на рівні PASSIVE_LEVEL IRQL, то це означає можливість безпечного доступу до ресурсів сторінкової пам’яті.
До речі, процедура вивантаження драйвера Unload не викликається у момент вимкнення системи, і якщо існує необхідність виконувати яку-небудь роботу при вимкненні системи, то це слід зробити в спеціально призначеній на те процедурі драйвера, зареєстрованій для обробки IRP пакетів з кодом IRP_MJ_SHUTDOWN. Об’єкт пристрою повинен бути за допомогою виклику IoRegisterShutdownNotification занесений в чергу об’єктів, що одержують повідомлення про перезавантаження, – тільки за цієї умови буде викликана процедура, зареєстрована для обробки пакетів з кодом IRP_MJ_SHUTDOWN.

6.2. Структура та призначення IRP-пакетів

6.2.1. Адресація і доступ до даних в IRP пакетах читання/запису

Після створення об'єкту пристрою (чи буде це зроблено в процедурі DriverEntry, як це було вказано вище для legacy, або в процедурі AddDevice для WDM драйверів) рекомендується явно описати спосіб, яким новий об'єкт пристрою готовий сприймати поля пакету IRP, що описують адреси областей пам'яті, через які відбуватиметься обмін між драйвером і його клієнтами. Наприклад:

```
PDEVICE_OBJECT pNewDeviceObject;
IoCreateDevice(. . . , &pNewDeviceObject);
pNewDeviceObject->Flags = DO_BUFFERED_IO;
```

або:

```
pNewDeviceObject->Flags = DO_DIRECT_IO;
```

За умовчанням, pNewDeviceObject->Flags = 0 – метод NEITHER.

В тому випадку, якщо новий об'єкт пристрою орієнтований на роботу з нижнім драйвером в стеку драйверів, слід скопіювати дані прапори з об'єкту пристрою, до якого підключений (одним з викликів IoAttachXxx), наприклад:

```
pNewDeviceObject->Flags =
(pUnderlyingDevObject->Flags) & (DO_BUFFERED_IO
 DO_DIRECT_IO);
```

За традицією, головними вважаються запити у формі пакетів IRP основним кодом IRP_MJ_READ (в результаті виклику ReadFile) або IRP_MJ_WRITE (в результаті виклику WriteFile).

Диспетчер введення/виведення, якщо він помічає в описі пристрою встановлений прапор DO_DIRECT_IO, неодмінно перевіряє можливість доступу до буфера, який клієнт драйвера в своєму запиті як буфер із даними (WRITE) або для даних (READ), готує MDL список для нього і фіксує сторінки в оперативній пам'яті. Адреса підготовленого таким чином MDL списку вноситься до поля IRP пакету під назвою MdlAddress. Якщо спробувати отримати віртуальну адресу від даного списку MDL викликом MmGetMdlVirtualAddress, то вийде саме віртуальна адреса, яку надав додаток користувача як адресу буфера з даними, що виводилися. Адреса в термінах системного адресного простору для тієї ж області даних можна отримати, якщо викликати MmGetSystemAddressForMdl. Коли обробка запиту введення/виведення повністю завершена, клієнтський буфер віддаляється схеми розподілу системної області пам'яті.

В тому випадку, якщо об'єкт пристрою, якому адресований IRP пакет, описаний прапором DO_BUFFERED_IO, то драйвер повинен узяти адресу буферної області з поля IRP пакету AssociatedIrp.SystemBuffer. Дана адреса

буде адресою в системному адресному просторі. Диспетчер введення/виведення виділяє цей буфер в несторінковій пам'яті після перевірки на доступність наданого клієнтом буфера. При запиті введення даних (READ-запит) диспетчер введення/виведення після закінчення операції переносить дані з системного буфера в клієнтський, а адреса клієнтського буфера запам'ятовується в полі IRP пакету UserBuffer. При запиті виведення даних (WRITE-запит) в системний буфер переносяться дані з клієнтського буфера (поле UserBuffer рівне NULL). В обох описаних вище випадках, надані в IRP пакеті адреси AssociatedIrp.SystemBuffer можна використовувати в довільному контексті в потоках режиму ядра.

В тому випадку, якщо пристрій не оголосив ознак DO_DIRECT_IO або DO_BUFFERED_IO, то диспетчер введення/виведення не робить особливих дій, а просто поміщає адресу буферної області, переданої йому ініціатором запиту в полі IRP пакету UserBuffer. Обидва поля Associate.SystemBuffer і MdlAddress встановлюються рівними NULL.

У останньому випадку поміщений в поле UserBuffer віртуальна адреса є "нехорошою" адресою. В більшості випадків ініціатором звернення до драйвера є програмний потік у режимі користувача. Відповідно, наданий їм віртуальна адреса вимагає для своєї коректної інтерпретації контекст відповідного призначеного для користувача потоку. Що, зрозуміло, виключає можливість використання такої адреси в довільних контекстах режиму ядра без попередньої підготовки (підготовки MDL списку і т.п.). Тому метод NEITHER можна рекомендувати тільки для драйверів самих верхніх драйверних шарів.

Довжина буфера у всіх випадках передається в полях Parameters.Write.Length (при WRITE-запитах) або Parameters.Read.Length (при READ-запитах).

Всі функції, опубліковані в процедурі DriverEntry шляхом заповнення масиву DriverObject->MajorFunction [...], викликаються диспетчером введення/виведення для обробки відповідних запитів від клієнтів драйвера (додатків режиму користувача або від модулів режиму ядра). Запити ці завжди оформлені у вигляді спеціальних структур даних – пакетів IRP.

6.2.2. Пакети IRP

Практично весь процес введення/виведення, що має місце в Windows, є пакетно-керованим. За допомогою IRP простежується також обробка запиту в підсистемі введення/виведення. Цей робочий рецепт має форму структури даних IRP.

При кожному запиті з програмного коду клієнта драйвера на виконання операції введення/, включаючи IOCTL запити, диспетчер введення/ виділяє під IRP область несторінкової пам'яті. Визначивши по дескриптору відкритог файлу, до якого драйвера і об'єкту пристрою адресовано звернення, і за запитаним кодом операції введення/виведення (IRP_MJ_Xxx), диспетчер передаєсформований пакет IRP у відповідну робочу процедуру драйвера. Слід

зазначити, що для доступу з програмного коду клієнта застосовується “файлова абстракція” процесу взаємодії з драйвером – відкриття, читання, запис, дескриптори і т.п.

Пакети IRP є структурами даних змінної довжини, і складаються із стандартного заголовка, що містить загальну облікову інформацію, і одного або декількох блоків параметрів, що називаються I/O stack location – стека введення/. Структура IRP пакету представлена на рис. 6.1

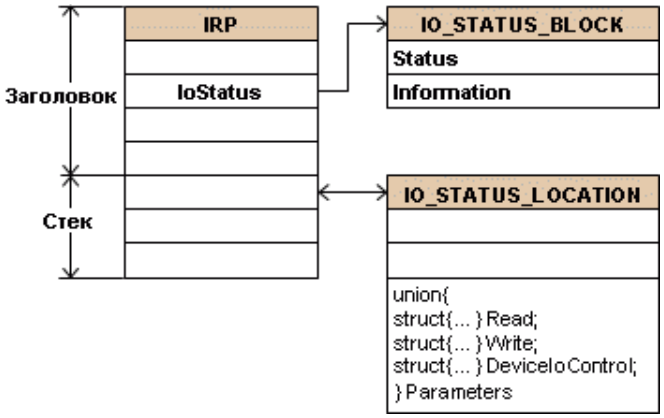

Рис. 6.1. Структура пакету IRP

6.2.3. Заголовок IRP

Нижче перераховані поля заголовка, до яких можна звертатися з програмного коду драйвера (див. табл. 6.4). До інших полів заголовка звертатися не рекомендується.

Таблиця 6.4

Заголовок пакету IRP

Поля	Опис
`IO_STATUS_BLOCK IoStatus`	Код стану (статус) запиту
`PVOID AssociatedIrp.SystemBuffer`	Покажчик на системний буфер для випадку, якщо пристрій підтримує буферизоване введення/в
`PMDL MdlAddress`	Покажчик на MDL список у випадку, якщо пристрій підтримує пряме введення/в
`PVOID UserBuffer`	Адреса призначеного для користувача буфера для введення/виведення
`BOOLEAN Cancel`	Індикатор того, що пакет IRP повинен бути анульований

Фрагмент структури IRP під назвою IoStatus фіксує остаточний стан даної операції введення/виведення. Коли драйвер готовий завершити обробку пакету

IRP, він встановлює в полі IoStatus.Status значення STATUS_XXX. У полі IoStatus.Information цього блоку записується 0 (якщо відбулася помилка) або інше визначене операцією введення/ значення, найчастіше – кількість переданих/отриманих байт даних.

6.2.4. Комірка стека введення/виведення

Основне призначення комірок стека введення/виведення (I/O stack location) полягає в тому, щоб зберігати функціональний код і параметри запиту на введення/ви (останні можуть зазнавати зміни при подорожі пакету по стеку драйверів). Нижче, у табл. 6.5 приводяться поля комірок стека введення/виведення, до яких драйвер може звертатися безпосередньо по покажчику (чого не рекомендується робити для решти полів).

Таблиця 6.5

Деякі елементи стека введення/виведення
(`IO_STACK_LOCATION *PIO_STACK_LOCATION`)

Поля	**Опис**
`UCHAR MajorFunction`	Код IRP_MJ_XXX, що описує призначення операції
`UCHAR MinorFunction`	Суб-код операції
`PDEVICE_OBJECT DeviceObject`	Покажчик на об’єкт пристрою, якому був адресований даний запит IRP
`PFILE_OBJECT FileObject`	Файловий об’єкт для даного запиту, якщо він заданий
union Parameters (трактування визначається значенням MajorFunction):	

`struct Read`	Параметри для IRP типу IRP_MJ_READ:
	`ULONG Length`
	`ULONG Key`
	`LARGE_INTEGER ByteOffset`

`struct Write`	Параметри для IRP типу IRP_MJ_WRITE:
	`ULONG Length`
	`ULONG Key`
	`LARGE_INTEGER ByteOffset`

`struct DeviceIoControl`	Параметри для IRP типу IRP_MJ_DEVICE_CONTROL:
	`ULONG OutputBufferLenght`
	`ULONG InputBufferLenght`
	`ULONG IoControlCode`
	`PVOID Type3InputBuffer`

Для запиту, який адресований драйверу самого нижнього рівня, відповідний пакет IRP має тільки один стек. Для запиту, який н драйверу верхнього рівня, диспетчер введення/виведення створює пакет IRP декількома стеками – по одній на кожен драйверний шар. Іншими словами, розмір стека в пакеті IRP чисельно дорівнює кількості драйверних шарів, що беруть участь в обробці запиту на дану операцію. Будь-якому драйверу в ієрархії дозволений доступ тільки до його власного, що "числиться" за ним, стеку пакету IRP (хоча ніхто не контролює й інші "протиправні" дії).

У випадку, якщо поточний драйвер вирішить звернутися до драйвера нижчого рівня з новим пакетом IRP власного виробництва, то він повинен гарантувати, що в новому пакеті IRP є достатня кількість стекових комірок.

Коли драйвер передає IRP пакет нижньому драйверному рівню, диспетчер введення/виведення автоматично змінює покажчик стека введення/виведення IRP пакету таким чином, що він указує на стек, призначений для драйвера чергового нижнього рівня. Коли обробка пакету драйвером нижнього рівня завершена, і він "відпускає" пакет IRP, покажчик стека знову повертається в початкове положення і вказує на стек для вищележачого драйвера. Зрозуміло, для отримання покажчика на поточний стек існує спеціальний системний виклик IoGetCurrentStackLocation.

6.3. Функціонування робочих процедур драйвера

6.3.1. Робочі процедури драйвера

Робочі процедури драйвера (у англомовній літературі по драйверах вони завжди називаються dispatch routines) реєструються драйвером під час роботи DriverEntry. Фактично, таким чином драйвер повідомляє диспетчера введення/виведення про свої наміри, якого типу запити (тобто IRP_MJ_Xxx коди) він збирається підтримувати.

В той момент, коли ініціалізувався запит введення/виведення, диспетчер введення/виведення в першу чергу створює пакет IRP, щоб зафіксувати факт проходження запиту. Разом іншою інформацією, в цьому IRP пакеті в полі MajorFunction зберігається код IRP_MJ_Xxx для того, щоб однозначно ідентифікувати тип цього запиту.

Код IRP_MJ_Xxx використовується диспетчером введення/виведення для того, щоб витягнути з масиву MajorFunction об'єкту драйвера покажчик на потрібну для обробки запиту процедуру драйвера. В тому випадку, якщо драйвер не підтримує операцію, то відповідний елемент MajorFunction указує на код усередині диспетчера введення/виведення (оскільки драйвер надав у такому разіиспетчерові введення/виведення право розпорядитися за власним розсудом), а саме – на функцію _IoInvalidDeviceRequest, який повертає клієнтові драйвера повідомлення про помилку. Таким чином, ініціатива

забезпечення кожного потрібногої коду IRP_MJ_Xxx власною оброблювальною процедурою належить авторові драйвера.

Метод оголошення процедур введення/виведення дозволяє також оголосити одну процедуру (функцію драйвера) для обслуговування декількох запитів введення/виведення. Для цього потрібно лише помістити адресу такої універсальної (якщо вона дійсно для цього призначена) функції в декількох к масиву MajorFunction, як це було показано вище для DriverEntry фільтр-драйвера. Оскільки пакет IRP містить код IRP_MJ_Xxx, то з його допомогою завжди можна відновити точний код потрібної дії.

Не слід виконувати усередині DriverEntry операції над позиціями в масиві MajorFunction, відповідних функціям введення/виведення, що не підтримуютьься драйвером. Диспетчер введення/виведення заповнює пропущені місця в масиві MajorFunction значенням покажчика на функцію _IoInvalidDeviceRequest перш, ніж звертається до процедури DriverEntry.

Всі функції драйвера, що беруть участь в обробці запитів і підлягають занесенню в таблицю 6.6 (масив) MajorFunction використовують один і той же протокол викликів, що включає число і тип параметрів, що передаються ним. Робочі процедури виконуються на PASSIVE_LEVEL рівні IRQL, що означає, що вони можуть звертатися до ресурсів сторінкової пам’яті.

6.3.2. Набір робочих процедур

Всі драйвери зобов’язані підтримувати функцію, яка відповідає за обробку запиту з кодом IRP_MJ_CREATE, оскількицей код генерується у відповідь на виклик в режимі користувача функцією CreateFile. Без підтримки цього запиту додаток користувача не матиме жодної можливості отримати дескриптор (handle) для доступу до драйвера пристрою, а значить, і до самого пристрою. Звісно, повинна існувати функція, яка оброблює запит із кодом IRP_MJ_CLOSE, що генерується по зверненню додатку до драйвера за допомогою функції API CloseHandle в режимі користувача. Необхідно відмітити, виклик CloseHandle виконується системою автоматично у момент завершення додатку у режимі користувача для всіх дескрипторів ресурсів, що залишили ці додатки відкритими.

Набір решти функцій залежить від особливостей пристрою, який знаходиться “під заступництвом” драйвера. У табл. 6.6 показано взаємозв’язок між кодами запитів введення/вив і викликами API режиму користувача, які приводять до їх генерації. При написанні багатошарових драйверів слід пам’ятати, що вищестоячий драйвер зобов’язаний підтримувати підмножину запитів нижче стоячого драйвера (драйверів), оскількизапит додатку користувача проникає в нижні шари тільки через нього.

Диспетчер введення/виведення викликає процедури диспетчеризації у відповідь на запити, що поступають, від додатків користувача або запитів від клієнтів, що працюють в режимі ядра. Перед викликом диспетчер введення/виведення будує пакет IRP і заповнює його необхідними даними,

включаючи покажчик на призначений для користувача буфер. Цей призначений для користувача буфер перевіряється диспетчером введення/виведення з тією метою, щоб забезпечити доступність для читання або запису сторінкових адрес буфера, якщо такі є в області сторінкової пам'яті, в контексті запиту, що поступає. В тому випадку, якщо поступив запит на буферизацію операції введення/виведення, диспетчер спочатку виділяє буфер в несторінковій пам'яті і, у запиту на запис, копіює дані з призначеного для користувача буфера в створений. У запиту на пряме введення/в, диспетчер "викликає" цілком призначений для користувача буфер у фізичну пам'ять і фіксує його (не допускаючи його переміщення на жорсткий диск, як це може бути з областями сторінкової пам'яті).

Таблиця 6.6

Коди запитів IRP і функції API призначеного для користувача режиму

IRP коди	Виклики API режиму користувача
IRP_MJ_CREATE	CreateFile
IRP_MJ_CLEANUP	Очищення чекаючих обробки пакетів IRP при закритті дескриптора при відробітку виклику CloseHandle
IRP_MJ_CLOSE	CloseHandle
IRP_MJ_READ	ReadFile
IRP_MJ_WRITE	WriteFile
IRP_MJ_DEVICE_CONTROL	DeviceIoControl
IRP_MJ_INTERNAL_DEVICE_CONTROL	Дії з управління пристроєм, доступні тільки для клієнтів, що працюють в режимі ядра (недоступно для викликів у режимі користувача)
IRP_MJ_QUERY_INFORMATION	Передача довжини файлу у відповідь на виклик GetFileSize
IRP_MJ_SET_INFORMATION	Установка довжини файлу по виклику SetFileSize
IRP_MJ_FLUSH_BUFFERS	Запис або очищення службових буферів при відпрацюванні, наприклад: FlushFileBuffers, FlushConsolelnputBuffer, PurgeComm
IRP_MJ_SHUTDOWN	Дії, які потрібно виконати драйверу в процесі підготовки системи до завершення роботи

Робоча процедура зазвичай може відстежувати стан обробки запиту тільки з використанням пакету IRP. В тому випадку, якщо робоча процедура використовує будь-які дані за межами пакету IRP, драйвер повинен забезпечити достатньо надійні заходи по синхронізації доступу до цих даних. Це означає,

що можна було б використовувати блокування” для координації дій з іншими процедурами, що виконуються на рівні IRQL не вище DISPATCH_LEVEL. Для синхронізації доступу з процедур, що виконуються на рівні коди обслуговування переривань, слід використовувати KeSyncronizeExecution.

Пакети IRP є даними колективного користування, хоч і з послідовним доступом. Зокрема, диспетчер введення/виведення використовує поля об’єднання Parameters для того, щоб завершити обробку запиту. Наприклад, після закінчення обробки буферизованого введення/виведення, йому необхідно виконати копіювання даних з області пам’яті несторінкового пулу в буфер, заданий додатком користувача. Після закінчення копіювання диспетчер введення/виведення повинен звільнити буфер в несторінковому пулі. Одне з полів об’єднання (union) Parameters на цей буфер, і зміна коду драйвера даного покажчика приведе до порушення роботи системи.

6.3.3. Послідовність дій робочих процедур

Конкретна поведінка кожної з робочих процедур драйвера залежатиме від функцій, які їй буде доручено підтримувати. Проте загальні обов’язки цих процедур включають наступні моменти:

1. Виклик IoGetCurrentIrpStackLocation для того, щоб отримати покажчик на стека IRP пакету, що відноситься до ведення даного драйвера.
2. Додаткову перевірку параметрів, специфічну для даного типу запиту і пристрою.
3. Продовження обробки IRP до моменту успішного завершення або виникнення помилкової ситуації, що перешкоджає подальшій обробці.

Коли робоча процедура драйвера обробляє пакет IRP, існує тільки три можливі варіанти закінчення її роботи:

- параметри запиту не проходять перевірку на повноту і правильність, і запит відхиляється;
- запит може бути оброблений в межах даної робочої процедури драйвера, без залучення фізичного пристрою, наприклад, читання/запис даних нульової довжини;
- відбувається звернення до фізичного пристрою з метою отримати дані або виконати дії над пристроєм, необхідні для завершення запиту.

Розглянемо на прикладі ці три випадки.

Випадок 1: Помилкова ситуація.

У випадку, якщо процедура диспетчеризації не може вирішити проблем, що виникли при обробці запиту, їй необхідно відхилити запит і повідомити про це іншій стороні. Наступні кроки необхідно зробити при віддзеркаленні “запиту”:

1. Відповідний код помилки зберігається в полі Status в блоці IoStatus пакету IRP і проводиться обнулення поля Information.
2. Проводиться виклик IoCompleteRequest для того, щоб завершити обробку пакету IRP (без підвищення пріоритету).

3. Робоча процедура, повертаючи управління, повинна повернути той же код помилки, що був розміщений в поле IoStatus.Status пакету IRP.

Наприклад:

```
NTSTATUS
WriteRequestHandler ( IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp)
{
   :
   // Запит не підтримується даним пристроєм (наприклад):
   pIrp->IoStatus.Status = STATUS_NOT_SUPPORTED;
   pIrp->IoStatus.Information = 0;      // Жодного байту не
передано
   IoCompleteRequest(pIrp,  IO_NO_INCREMENT);     //  без  змін
пріоритету
return STATUS_NOT_SUPPORTED;
}
```

Виклик IoCompleteRequest – слід зазначити, що після нього область пам'яті, зайнята під власне пакет IRP може виявитися вільним. Тому категорично не можна економити і писати операторів типу "return pIrp->IoStatus.Status;", проте, як і звертатисяза адресою pIrp в яких би то не було цілях після виклику IoCompleteRequest.

Випадок 2: Завершення роботи над IRP запитом.

Деякі запити можуть бути повністю оброблені без звернення до фізичного пристрою, за який відповідає драйвер, наприклад, отримання дескриптора пристрою або настройки режиму роботи самого драйвера. В цьому випадку робоча процедура повинна виконати наступні дії:

1. Розмістити код успішного завершення в поле Status в блоці IoStatus пакету IRP і вказати прийнятне значення в полі Information.
2. Виконати виклик IoCompleteRequest, щоб звільнити пакет IRP без підвищення пріоритету.
3. Повернути управління з кодом STATUS_SUCCESS.

Наприклад:

```
NTSTATUS
CloseRequestHandler( IN PDEVICE_OBJECT pDevObj, IN PIRP pIrp)
{
   :
   pIrp->IoStatus.Status = STATUS_SUCCESS;
   pIrp->IoStatus.Information = 0;
   IoCompleteRequest ( pIrp, IO_NO_INCREMENT );
   return STATUS SUCCESS;
}
```

Випадок 3: Робота через черги IRP пакетів.

Зрозуміло, простий драйвер може ініціалізувати свій пристрій в процедурі DriverEntry, в обробнику запитів IRP_MJ_READ одразу ж прочитувати дані з пристрою (наприклад, LPT порту). Проте, повномасштабний ритуал роботи з

підсистемою введення/виведення Windows, тобто диспетчером введення/виведення, диктує іншу послідовність дії. Робоча процедура повинна помістити пакет IRP в чергу для подальшої обробки процедурою STARTIO і повернути диспетчерові введення/виведення повідомлення про те, що обробка IRP не завершена, а саме:

1. Виконати виклик IoMarkIrpPending – щоб інформувати диспетчера введення/виведення про те, що пакет IRP поставлений в чергу на обробку.

2. Виконати виклик IoStartPacket, щоб помістити пакет IRP в системну чергу для подальшої його обробки процедурою STARTIO. Драйвер може реалізовувати і свої черги IRP пакетів.

3. Повернути управління з робочої процедури з кодом завершення STATUS_PENDING.

Фрагмент коду, приведений нижче, демонструє, як робоча процедура розміщує IRP запит в черзі на обробку:

```
NTSTATUS ReadRequestHandler( IN PDEVICE_OBJECT  pDeviceObject,
                             IN PIRP            pIrp )
{
   :
   // IRP "в роботі", але робота з ним буде відбуватися
   // через чергу пакетів (в даному випадку - системну):
   IoMarkIrpPending( pIrp );

   // Четвертий параметр дозволяє вказати процедуру видалення
   // CancelRoutine
   IoStartPacket( pDeviceObject, pIrp, 0, NULL );

   return STATUS_PENDING;
}
```

Деякі джерела указують, що диспетчер введення/виведення автоматично завершує запити, не помічені кодом STATUS_PENDING, відразу ж після отримання управління з робочої процедури. Можливо, даний автоматично механізм, що запускається, колись і працював саме так. Проте в доступних на сьогодні версіях Windows це не підтверджується, тобто для нормального завершення обробки IRP пакету необхідно, щоб поточний драйвер проводив виклик IoCompleteRequest в кінці обробки IRP пакету в своїй робочій процедурі (якщо тільки він не помічений як відкладений в системну чергу, STATUS_PENDING). До того ж, за цієї умови будуть запущені всі процедури завершення у вище стоячих драйверах, якщо такі, звичайно, є.

Висновки

- Legacy Driver, NT Style Driver – успадкований драйвер (застарілий драйвер), “драйвер в силі NT”. При компіляції використовує тільки

визначення з файлу ntddk.h, тому може повною мірою користуватися застарілими функціями HalGetBusData, HalGetInterruptVector тощо. “Правильний” драйвер “в стилі NT” може бути запущений і зупинений за допомогою програми Monitor (зі складу пакету Numega Kernel Driver) або процедурами SCM Менеджера.

- Для створення драйверу, який можна запустити і зупинити достатньо реалізувати дві функції DriverEntry, DriverUnload. Для того щоб драйвер приймав участь в процедурах введення/виведення, потрібно організувати обробку IRP пакетів.

Контрольні запитання та завдання

1. Назвіть основні параметри виклику процедури DriverEntry.
2. Призначення процедури DriverEntry.
3. Призначення та основні характеристики процедури вивантаження Unload.
4. Адресація і доступ до даних в IRP пакетах читання/запису.
5. Назвіть та охарактеризуйте основні складові структура пакету IRP.
6. Назвіть характерні особливості заголовку пакету IRP.
7. Поняття комірки стека введення/виведення.
8. Охарактеризуйте основні робочі процедури драйвера.
9. Назвіть послідовність дій робочих процедур.

Тести для самоконтролю

1. Legacy-драйвер – це:
   a. драйвер орієнтований на пристрої, що підтримують специфікацію PNP
   b. драйвер “в стилі NT”
   c. драйвер, що бере участь в обміні даними з приводу змін в енергопостачанні
2. IRQL – це:
   a. файловий об’єкт з особливими властивостями
   b. розширення об’єкту пристрою
   c. код управління введенням/виведенням
   d. рівень пріоритету виконання
3. IOCTL – це:
   a. код управління введенням/виведенням
   b. пакет запиту на введенням/виведенням
   c. рівень пріоритету виконання
   d. файловий об’єкт з особливими властивостями
4. IRP – це:
   a. розширення об’єкту пристрою
   b. код управління введенням/виведенням

c. пакет запиту на введенням/виведенням
d. рівень пріоритету виконання

5. Для того щоб створити драйвер, який можна запустити і зупинити достатньо:
   a. організувати обробку IRP пакетів
   b. необхідно написати і зареєструвати процедуру AddDevice
   c. реалізувати функції DriverEntry, DriverUnload
   d. необхідно написати і зареєструвати процедуру Unload
6. Для того щоб драйвер приймав участь в процедурах введення/виведення, потрібно:
   a. організувати обробку IRP пакетів
   b. реалізувати функцію DriverEntry
   c. необхідно написати і зареєструвати процедуру AddDevice
   d. реалізувати функцію DriverUnload
7. Процедура Unload необхідна для:
   a. для того, щоб завантажити драйвер
   b. для створення об'єкту пристрою для кожного фізичного або логічного пристрою під управлінням даного драйвера
   c. для того, щоб зробити драйвер вивантажуваним
   d. для визначення апаратного забезпечення, яке драйвер контролюватиме
8. DriverEntry – це:
   a. функція драйвера, яка буде викликана першою при його завантаженні
   b. функція драйвера, яка буде викликана першою при його вивантаженні
   c. функція драйвера, в якій організовано обробку IRP пакетів
   d. функція драйвера, що звільняє пам'ять, виділену драйверу
9. Реєстрація основних робочих процедур (Dispatch Routines) відбувається в:
   a. IoCreateDevice
   b. AddDevice
   c. IoCreateSymbolicLink
   d. DriverEntry
10. Для legacy-драйвера об'єкт пристрою створюється в процедурі:
    a. DriverEntry
    b. AddDevice
    c. DriverUnload
    d. DispatchRoutines
11. У разі успішного завершення, функція DriverEntry повинна повернути
    a. покажчик на об'єкт драйвера
    b. PASSIVE_LEVEL
    c. STATUS_SUCCESS
    d. Void
12. Процедура Unload повертає:
    a. покажчик на об'єкт драйвера
    b. PASSIVE_LEVEL
    c. STATUS_SUCCESS
    d. Void

13. Яким чином драйвер повідомляє диспетчер введення/виведення про те, якого типу запити, тобто IRP_MJ_Xxx коди, він збирається підтримувати?
    a. шляхом виведення відладочних повідомлень
    b. шляхом реєстрації робочих процедур драйверу в DriverEntry
    c. шляхом створення об'єкту пристою
    d. шляхом реєстрації робочих процедур драйверу в процедурі Unload
14. Яке основне призначення комірок стека введення/?
    a. необхідні для підключення пристрою до об'єкту переривань
    b. необхідні для реєстрації робочих процедур драйверу
    c. необхідні для збереження функціонального коду і параметрів запиту на введення/виведення
    d. необхідні для обробки відповідних запитів від клієнтів драйвера
15. Де обробляються пакети IRP?
    a. в процедурі DriverEntry
    b. в процедурі Unload
    c. в робочих процедурах драйвера
    d. в процедурі AddDevice
16. Навіщо необхідний виклик IoCompleteRequest?
    a. для завершення обробки пакету IRP (без підвищення пріоритету)
    b. для підключення пристрою до об'єкту переривань
    c. для реєстрації робочих процедур драйверу
    d. для обробки відповідних запитів від клієнтів драйвера

Теми для самостійного опрацювання

1. Уніфікована модель драйвера.
2. Диспетчерські точки входу драйвера.
3. Обробка запитів IRP стеком драйверів.
4. Поняття серіалізації.
5. Робочі процедури обслуговування IOCTL запитів.

Список використаної літератури

1. *Комиссарова В.* Программирование драйверов для Windows / В. Комиссарова. – СПб.: БХВ-Петербург, 2007. – 256 с.
2. *Сорокина С. И.* Программирование драйверов и систем безопасности : учебное пособие / С. И. Сорокина, А. Ю. Тихонов, А. Ю. Щербаков. – СПб.: БХВ-Петербург, М.: Издатель Молгачева С. В., 2002. – 256 с.
3. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО "Бином-Пресс", 2004. – 480 с.

РОЗДІЛ 7. ВИКОРИСТАННЯ ДИСПЕТЧЕРА КЕРУВАННЯ СЛУЖБАМИ ДЛЯ ЗАВАНТАЖЕННЯ/ВИВАНТАЖЕННЯ ДРАЙВЕРІВ

План

- ✦ Призначення та основні задачі Windows Service Control Manager
- ✦ Програмний інтерфейс для роботи з SCM
- ✦ Програмний інтерфейс для керування службами і драйверами

В даному розділі будуть розглядатися призначення та основні задачі диспетчера керування службами, який забезпечує програмний інтерфейс для ведення бази даних встановлених служб; запуску/зупинки служб та драйверів при завантаженні системи або на вимогу; підтримки інформації про запущені служби; передачі запитів керування від користувальницьких додатків служби.

7.1. Призначення та основні функції Windows Service Control Manager

Windows Service Control Manager (SCM) являє собою сервер видаленого виклику процедур, який управляє створенням, видаленням, запуском і зупинкою системних служб Windows. SCM запускається під час завантаження системи і побудований так, що програми налаштування та управління службами можуть маніпулювати їм навіть з віддаленої машини.

SCM забезпечує інтерфейс для вирішення наступних задач:

- ведення бази даних встановлених служб;
- запуск/зупинка служб та драйверів при завантаженні системи або на вимогу;
- підтримка інформації про запущені служби;
- передача запитів керування від користувальницьких додатків службам.

Service Control Manager підтримує базу даних встановлених служб в реєстрі. Ця база даних використовується як самим SCM, так і іншими програмами для додавання, змінення або налаштувати служб. Для зберігання необхідної інформації використовується гілка системного реєстру HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services. Початкова копія бази даних створюється при встановленні системи і містить записи для драйверів і служб, необхідних під час завантаження системи.

База даних включає в себе такі параметри:

1. Тип служби вказує чи служба виконується у власному процесі, чи у спільному з іншими службами процесі. Для служб драйверів, вказується, чи це драйвер ядра, чи драйвер файлової системи.

2. Тип запуску визначає, чи служба запускається автоматично при завантаженні системи (auto-start service), чи SCM запускає її на вимогу програми управління службою (demand-start service). Тип запуску може також вказувати, що служба відключена (в цьому випадку вона не може бути запущена).

3. Рівень контролю помилок вказує на серйозність згенерованої помилки, якщо служба або драйвер не може запуститися при старті системи, і визначає заходи, які будуть вжиті.

4. Повний шлях до виконуваного файлу. Служби повинні мати розширення .exe, а драйвери – .sys.

5. Додаткова інформація про залежності, що використовується для визначення належного порядку запуску служб.

6. Для служб – ім'я облікового запису користувача та пароль. Сервісна програма виконується в контексті цього облікового запису. Якщо ця інформація не зазначена, то служба виконується в контексті LocalSystem.

7. Для драйверів – ім'я об'єкта драйвера (наприклад, \FileSystem\Rdr або \Driver\XNS), що використовується системою введення/виведення для завантаження драйвера пристрою. Якщо назва не задано, система створює ім'я за замовчуванням на основі імені драйвера.

7.2. Програмний інтерфейс для роботи з Service Control Manager

Програмний інтерфейс SCM підтримує роботу з такими об'єктами:

- база даних встановлених служб (SCManager);
- служба (service);
- блокування бази даних.

Об'єкт SCManager являє собою контейнер, який містить об'єкти служб. Функція OpenSCManager повертає дескриптор об'єкту SCManager на певному комп'ютері. Цей дескриптор використовується, для установки, видалення, відкриття та перерахуванням служб і при блокуванні бази даних.

Функція має такі параметри:

```
SC_HANDLE WINAPI OpenSCManager(
  __in_opt  LPCTSTR lpMachineName,
  __in_opt  LPCTSTR lpDatabaseName,
  __in      DWORD dwDesiredAccess
);
```

де:

- lpMachineName – ім'я цільового комп'ютера. Якщо цей параметр рівний NULL або вказує на пустий рядок, функція підключається до менеджера служб на локальному комп'ютері;

- lpDatabaseName – ім’я бази даних SCM. Цей параметр повинен бути встановлений на SERVICES_ACTIVE_DATABASE. Якщо це NULL, база даних SERVICES_ACTIVE_DATABASE відкривається за замовчуванням;
- dwDesiredAccess – бажаний доступ до менеджера служб. Право SC_MANAGER_CONNECT зазначається неявно при виклику цієї функції.

У разі успіху функція повертає дескриптор потрібного менеджера служб, інакше NULL. Цей дескриптор може використовуватися лише в рамках даного процесу і може бути закритий за допомогою функції CloseServiceHandle.

7.3. Програмний інтерфейс для керування службами і драйверами

Для роботи зі службою необхідно отримати її дескриптор. Для цього використовують функції CreateService і OpenService для створення нової чи відкриття існуючої служби відповідно.

Функція CreateService:

```
SC_HANDLE WINAPI CreateService(
  __in        SC_HANDLE hSCManager,
  __in        LPCTSTR lpServiceName,
  __in_opt    LPCTSTR lpDisplayName,
  __in        DWORD dwDesiredAccess,
  __in        DWORD dwServiceType,
  __in        DWORD dwStartType,
  __in        DWORD dwErrorControl,
  __in_opt    LPCTSTR lpBinaryPathName,
  __in_opt    LPCTSTR lpLoadOrderGroup,
  __out_opt   LPDWORD lpdwTagId,
  __in_opt    LPCTSTR lpDependencies,
  __in_opt    LPCTSTR lpServiceStartName,
  __in_opt    LPCTSTR lpPassword
);
```

де:

- hSCManager – дескриптор менеджера служб. Він повертається функцією OpenSCManager і повинен мати право доступу SC_MANAGER_CREATE_SERVICE;
- lpServiceName – назва служби для встановлення. Максимальна довжина рядка становить 256 символів. Слеш (/) і зворотній слеш (\) не є допустимими символами в імені служби;
- lpDisplayName – ім’я служби, яке буде використовуватися програмами для відображення у інтерфейсі користувача. Цей рядок має максимальну довжину 256 символів;
- dwDesiredAccess – бажаний доступ до служби;
- dwServiceType – тип служби. Цей параметр може бути одним з наступних значень, що наведені в таблиці 7.1.

Таблиця 7.1

Параметри служби dwServiceType

Значення	Опис
SERVICE_ADAPTER 0x00000004	Зарезервовано
SERVICE_FILE_SYSTEM_DRIVER 0x00000002	Драйвер файлової системи
SERVICE_KERNEL_DRIVER 0x00000001	Драйвер пристрою
SERVICE_RECOGNIZER_DRIVER 0x00000008	Зарезервовано
SERVICE_WIN32_OWN_PROCESS 0x00000010	Служба, яка працює у своєму власному процесі
SERVICE_WIN32_SHARE_PROCESS 0x00000020	Служба, яка розділяє процес з однією або кількома іншими

– dwStartType – тип запуску служби. Цей параметр може бути одним з наступних значень, що представлені в табл. 7.2.

Таблиця 7.2

Параметри служби dwStartType

Значення	Опис
SERVICE_AUTO_START 0x00000002	Служба запускається автоматично при запуску системи
SERVICE_BOOT_START 0x00000000	Драйвер пристрою запускається системним завантажувачем. Це значення є дійсним тільки для драйверів
SERVICE_DEMAND_START 0x00000003	Служба запускається коли певний процес викликає функцію StartService
SERVICE_DISABLED 0x00000004	Служба, яка не може бути запущена. Спроба запустити таку службу поверне код помилки ERROR_SERVICE_DISABLED
SERVICE_SYSTEM_START 0x00000001	Драйвер пристрою запускається функцією IoInitSystem. Це значення є дійсним тільки для драйверів

– dwErrorControl – серйозність згенерованої помилки, при невдалому запуску цієї служби. Цей параметр може приймати одне з наступних значень, що подано в табл. 7.3.

Таблиця 7.3

Параметри служби dwErrorControl

Значення	Опис

Значення	Опис
SERVICE_ERROR_CRITICAL 0x00000003	При можливості реєструється помилка в журналі подій. Якщо використовувалась остання вдала конфігурація, запуск не відбувається. У іншому випадку, система перезавантажується з використанням останньої вдалої конфігурації
SERVICE_ERROR_IGNORE 0x00000000	Помилка ігнорується, запуск продовжується
SERVICE_ERROR_NORMAL 0x00000001	Помилка реєструється в журналі подій, запуск продовжується
SERVICE_ERROR_SEVERE 0x00000002	Реєструється помилка в журналі подій. Якщо використовувалась остання вдала конфігурація, запуск продовжується. У іншому випадку, система перезавантажується з використанням останньої вдалої конфігурації

– lpBinaryPathName – повний шлях до виконуваного файлу служби. Якщо шлях містить пробіли, він повинен бути в лапках. Наприклад, шлях "d:\\my share\\myservice.exe" повинен бути вказаний як "\"d:\\my share\\myservice.exe\".

Шлях може також включати аргументи для автоматичного запуску служби. Наприклад, "d:\\myshare\\myservice.exe arg1 arg2". Ці аргументи передаються до служби у точці входу (як правило, функція main).

Якщо вказується шлях на інший комп'ютер, він повинен бути доступним для облікового запису локального комп'ютера. Однак ця вимога дозволяє будь-якій потенційній вразливості на віддаленому комп'ютері впливати на локальний комп'ютер. Тому, кращим способом є використання локальних файлів, а саме:

– lpLoadOrderGroup – ім'я групи, у яку входить ця служба. NULL або порожній рядок означає, що служба не належать до жодної групи. Групи використовується для визначення порядку запуску служб і перераховані у такому розділі реєстру: HKEY_LOCAL_MACHINE\SYSTEM\ CurrentControlSet\ Control\ ServiceGroupOrder;
– lpdwTagId – вказівник на змінну, яка отримує значення тега, що є унікальним в групі, зазначеній у параметрі lpLoadOrderGroup. Цей параметр можу приймати значення NULL;
– lpDependencies – вказівник на NULL -NULL -завершений масив імен служб або груп, які система повинна запустити перед цією службою. NULL або порожній рядок означають, що служба не має залежностей. Залежність від групи означає, що ця служба може працювати, якщо хоча б один член групи працює після спроби запустити всю групу. Імена груп повинні позначатися префіксом SC_GROUP_IDENTIFIER з тим щоб їх можна було відрізнити від імен служб, так як вони знаходяться в одному й тому ж просторі імен;

- lpServiceStartName – ім’я облікового запису, під яким служба повинна працювати. Якщо служба типу SERVICE_WIN32_OWN_PROCESS, використовується ім’я облікового запису у вигляді [ім’я домену]\[ім’я користувача]. Якщо обліковий запис належить до вбудованого домену, можна вказати .\[ім’я користувача]. Якщо цей параметр дорівнює NULL, CreateService використовує обліковий запис LocalSystem. Якщо служба має тип SERVICE_INTERACTIVE_PROCESS, то вона повинна працювати в контексті LocalSystem. Якщо служба типу SERVICE_KERNEL_DRIVER або SERVICE_FILE_SYSTEM_DRIVER, цей параметр повинен містити ім’я об’єкта драйверу, яке система використовує для завантаження драйвера пристрою. При значенні NULL буде використане ім’я за замовчуванням;
- lpPassword – пароль для облікового запису, ім’я якого задане параметром lpServiceStartName. Вкажіть порожню рядок, якщо обліковий запис не має паролю, або якщо служба працює в контексті LocalService, NetworkService, або LocalSystem. Для драйверів цей параметр ігнорується.

У разі успіху функція повертає дескриптор нової служби, інакше значення NULL. Цей дескриптор може використовуватися лише в рамках даного процесу і може бути закритий за допомогою функції CloseServiceHandle.

Наприклад, для того, щоб встановити просту службу, можна використовувати таку процедуру:

```
VOID SvcInstall(TCHAR szPath, TCHAR svcName)
{
    SC_HANDLE schSCManager;
    SC_HANDLE schService;

    // Get a handle to the SCM database.

    schSCManager = OpenSCManager(
        NULL,                    // local computer
        NULL,                    // ServicesActive database
        SC_MANAGER_ALL_ACCESS);  // full access rights

    if (NULL == schSCManager)
    {
        printf("OpenSCManager failed (%d)\n", GetLastError());
        return;
    }

    // Create the service

    schService = CreateService(
        schSCManager,              // SCM database
        svcName,                   // name of service
        svcName,                   // service name to display
        SERVICE_ALL_ACCESS,        // desired access
        SERVICE_WIN32_OWN_PROCESS, // service type
```

```
        SERVICE_DEMAND_START,       // start type
        SERVICE_ERROR_NORMAL,       // error control type
        szPath,                     // path to service's binary
        NULL,                       // no load ordering group
        NULL,                       // no tag identifier
        NULL,                       // no dependencies
        NULL,                       // LocalSystem account
        NULL);                      // no password

    if (schService == NULL)
    {
        printf("CreateService failed (%d)\n", GetLastError());
        CloseServiceHandle(schSCManager);
        return;
    }
    else printf("Service installed successfully\n");

    CloseServiceHandle(schService);
    CloseServiceHandle(schSCManager);
}
```

Функція OpenService має вигляд, як:

```
SC_HANDLE WINAPI OpenService(
  __in  SC_HANDLE hSCManager,
  __in  LPCTSTR lpServiceName,
  __in  DWORD dwDesiredAccess
);
```

де:

- hSCManager – дескриптор менеджера служб. Цей дескриптор повертається функцією OpenSCManager і повинні мати право доступу SC_MANAGER_CREATE_SERVICE;
- lpServiceName – назва служби для встановлення. Максимальна довжина рядка становить 256 символів. Слеш (/) і зворотній слеш (\) не є допустимими символами в імені служби;
- dwDesiredAccess – бажаний доступ до служби.

У разі успіху функція повертає дескриптор існуючої служби, інакше значення NULL. Цей дескриптор може використовуватися лише в рамках даного процесу і може бути закритий за допомогою функції CloseServiceHandle.

Маючи дескриптор служби драйвера, над ним можна виконати такі операції:

- запуск служби за допомогою функції StartService;
- управління службою (наприклад, зупинка) за допомогою функції ControlService;
- деінсталяція служби (DeleteService).

Функція StartService представляється як:

```
BOOL WINAPI StartService(
  __in      SC_HANDLE hService,
  __in      DWORD dwNumServiceArgs,
  __in_opt  LPCTSTR *lpServiceArgVectors
);
```

де:

- hService – дескриптор служби. Цей дескриптор повертається функціями OpenService або CreateService, і повинен мати право доступу SERVICE_START;
- dwNumServiceArgs – число рядків у масиві lpServiceArgVectors. Якщо lpServiceArgVectors не задано, то цей параметр може бути NULL.
- lpServiceArgVectors – масив рядків, які передаються функції ServiceMain в якості аргументів. Якщо немає аргументів, то цей параметр може бути нульовим. В іншому випадку, перший аргумент (lpServiceArgVectors[0]) це ім'я служби, після чого будь-які додаткові аргументи. Цей параметр не використовується для драйверів.

При запуску драйверів дана функція не повертає контролю до користувацького процесу, доки не буде завантажено драйвер.

Функція ControlService:

```
BOOL WINAPI ControlService(
  __in   SC_HANDLE hService,
  __in   DWORD dwControl,
  __out  LPSERVICE_STATUS lpServiceStatus
);
```

де:

- hService – дескриптор служби. Цей дескриптор повертається функціями OpenService або CreateService;
- dwControl – один з наступних контролюючих кодів, що представлений в табл. 7.4.

Таблиця 7.4

Параметри служби dwControl

Код	Опис
SERVICE_CONTROL_CONTINUE 0x00000003	Повідомляє призупинену службу, що слід відновити роботу. Дескриптор повинен мати право доступу SERVICE_PAUSE_CONTINUE
SERVICE_CONTROL_INTERROGATE 0x00000004	Повідомляє службу, що вона повинна представити свої поточні відомості про стан. Дескриптор повинен мати право доступу SERVICE_ INTERROGATE. Зазвичай не використовується
SERVICE_CONTROL_NETBINDADD	

Код	Опис
0x00000007	Коди для підтримки роботи мережевих служб. Застарілі і зараз не використовуються
SERVICE_CONTROL_NETBINDDISABLE 0x0000000A	
SERVICE_CONTROL_NETBINDENABLE 0x00000009	
SERVICE_CONTROL_NETBINDREMOVE 0x00000008	
SERVICE_CONTROL_PARAMCHANGE 0x00000006	Повідомляє службу про зміну стартових параметрів. Дескриптор повинен мати право доступу SERVICE_CONTROL_PARAMCHANGE
SERVICE_CONTROL_PAUSE 0x00000002	Надсилає запит на призупинення служби. Дескриптор повинен мати право доступу SERVICE_PAUSE_CONTINUE
SERVICE_CONTROL_STOP 0x00000001	Повідомляє службу про те, що вона повинна зупинитися. Дескриптор повинен мати право доступу SERVICE_STOP. Після відправлення запиту на зупинку служби, не можна надсилати інші коди контролю

– lpServiceStatus – вказівник на структуру SERVICE_STATUS, яка отримує найсвіжішу інформацію про стан служби. Вона відображає останній статус, що служба повідомила менеджеру служб.

Функція DeleteService:

```
BOOL WINAPI DeleteService(
  __in  SC_HANDLE hService
);
```

– hService – дескриптор служби. Цей дескриптор повертається функціями OpenService або CreateService, і повинен мати право доступу DELETE.

Варто відмітити, що дана функція не видаляє службу з бази даних SCM, а лише помічає її до видалення. Службу буде видалено лише після того, як всі її дескриптори будуть закриті, а сама вона – зупинена. Якщо це неможливо, то видалення відбудеться при перезавантаженні системи.

Приклад деінсталяції служби чи драйвера:

```
VOID _stdcall DoDeleteSvc(TCHAR szSvcName)
{
    SC_HANDLE schSCManager;
    SC_HANDLE schService;
    SERVICE_STATUS ssStatus;

    // Get a handle to the SCM database.

    schSCManager = OpenSCManager(
```

```
        NULL,                    // local computer
        NULL,                    // ServicesActive database
        SC_MANAGER_ALL_ACCESS);  // full access rights

    if (NULL == schSCManager)
    {
        printf("OpenSCManager failed (%d)\n", GetLastError());
        return;
    }

    // Get a handle to the service.

    schService = OpenService(
        schSCManager,       // SCM database
        szSvcName,          // name of service
        DELETE);            // need delete access

    if (schService == NULL)
    {
        printf("OpenService failed (%d)\n", GetLastError());
        CloseServiceHandle(schSCManager);
        return;
    }

    // Delete the service.

    if (! DeleteService(schService) )
    {
        printf("DeleteService failed (%d)\n", GetLastError());
    }
    else printf("Service deleted successfully\n");

    CloseServiceHandle(schService);
    CloseServiceHandle(schSCManager);
}
```

Висновки

SCM – сервер видаленого виклику процедур, який управляє створенням, видаленням, запуском і зупинкою системних служб Windows та забезпечує програмний інтерфейс ведення бази даних встановлених служб, запуску/зупинки служб та драйверів та передачу запитів керування.

Контрольні запитання та завдання

1. Назвіть основні параметри виклику DriverEntry.
2. Призначення процедури DriverEntry.
3. Призначення та основні характеристики процедури вивантаження Unload.
4. Адресація і доступ до даних в IRP пакетах читання/запису.

5. Назвіть та охарактеризуйте основні складові структура пакету IRP.
6. Назвіть характерні особливості заголовку пакету IRP.
7. Поняття комірки стека введення/виведення.
8. Охарактеризуйте основні робочі процедури драйвера.
9. Назвіть послідовність дій робочих процедур.

Теми для самостійного опрацювання

1. Служби. Диспетчер керування системними службами.
2. Структура програми керування системною службою.
3. Встановлення каналу зв'язку з SCM-менеджером, запуск та реєстрація драйвера.

Список використаної літератури

1. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО "Бином-Пресс", 2004. – 480 с.
2. *Харт Джонсон М.* Системное программирование в среде Windows / М. Джонсон Харт; пер. с англ. – [3-е изд.]. – М.: Издательский дом "Вільямс", 2005. – 592 с.
3. *Хоглунд Г.* Рукиты: внедрение в ядро Windows / Г. Хоглунд, Дж. Баталер. – СПб.: Питер, 2007. – 285 с.

РОЗДІЛ 8. МЕТОДИКИ ТЕСТУВАННЯ ТА ВІДЛАДКИ ДРАЙВЕРІВ

План

- ✦ Програмні засоби від Microsoft
- ✦ Особливості тестування та налагодження драйверів
- ✦ Усунення несправностей
- ✦ Запобігання збоїв під час роботи драйверів
- ✦ Загальні прийоми відладки
- ✦ Приватні прийоми відновлення системи
- ✦ Процес завантаження операційної системи

В даному розділі детально розглянуті програмні засоби тестування та налагодження драйверів, особливості даного процесу. Багато уваги буде приділено загальним прийомама відладки та тестування.

8.1. Програмні засоби від Microsoft

Основним засобом розробки є Microsoft DDK (Driver Development Kit). Windows DDK, Device Driver Kit – пакет розробки драйверів, що включає компілятор, редактор зв'язків, заголовні файли, бібліотеки, великий набір прикладів, частина з яких є драйверами, що реально працюють в операційній системі, і документацію. До складу пакету входить також відладчик WinDbg, що дозволяє проводити інтерактивну відладку драйвера на двохкомп'ютерній конфігурації і за наявності файлів налагоджувальних ідентифікаторів операційної системи WinDbg крім того, дозволяє проглядати файли дампу (образу) пам'яті, отриманого при фатальних збоях ОС (crash dump file). До цього пакету також додається і загальна документація.

У безкоштовно поширюваному пакеті DDK було завжди відсутнє інтегроване середовище розробки. Тому програмісти драйверів завжди були вимушені підбирати для себе і засіб редагування початкового коду. Вибір був, практично, безальтернативний – пакет Visual C++ (тепер це Visual Studio 7 Net). При належній настройці цього середовища процес виявлення синтаксичних помилок істотно полегшується – це є невід'ємною перевагою інтегрованих середовищ програмування. Компілятор і редактор зв'язків Visual Studio C++ створюють нормальний бінарний код, цілком працездатний при вказівці відповідних опцій (настройок) компіляції, проте еталоном слід вважати бінарний код, що виходить при компіляції коду драйвера з використанням

утиліти Build з складу пакету DDK. Зрозуміло, вбудований інтерактивний відладчик Visual Studio і документація, що додається, стають для розробки драйвера абсолютно даремними, оскільки не призначені для роботи з програмним забезпеченням в режимі ядра.

Є пакети розробки драйверів і від третіх фірм, наприклад: WinDriver або NuMega Driver Studio. Але у них є відмінності від базису функцій Microsoft і багато інших дрібних незручностей. Отже DDK – найкращий варіант. Якщо ж є бажання писати драйвера виключно на асемблері, тобі підійде KmdKit (Kernel Mode Driver Development Kit) для MASM32. Правда, цей варіант підходить тільки для Win2k/XP.

Що стосується сторонніх утиліт, то деякі вже включені в стандартне постачання Windows: редактор реєстру. Але їх у будь-якому випадку не вистачить. Треба будемо проводити інсталяцію окремо. Безліч найкорисніших утиліт створили патріархи системного кодинга в Windows: Марк Руссинович, Гарі Неббет, Свен Шрайбер та інші. Наприклад, Марк Руссинович створив багато корисних утиліт: RegMon і FileMon (монітори звернень до реєстру і файлів відповідно), WinObj (засіб перегляду директорій імен об'єктів), DebugView,DebugPrint (програми перегляду, збереження тощо).

Свена Шрайбера розробив утиліти w2k_svc -_sym, -_mem, що дозволяють переглядати встановлені драйвера, додатки і служби, які працюють в режимі ядра, робити дамп пам'яті тощо.

Досить добре себе зарекомендували такі програми, як PE Explorer, PE Browse Professional Explorer, і такі незамінні, як дизасемблер IDA і відладчик SOFTICE.

Тому, найбільш поширені інструменти розробки драйверів розглянемо детальніше.

8.1.1. Установки проекту в Visual Studio 7 Net

Налаштування проекту для компіляції і збірки драйвера режиму ядра суттєво відрізняються від параметрів, які використовуються для роботи з додатками і динамічними бібліотеками у режимі користувача. Нижче наводиться точний текст файлу Example.sln ("sln" є скороченням від "solution"), який описує проект драйвера Example:

```
Microsoft Visual Studio Solution File, Format Version 7.00
Project ( "(8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942)") = "Example",
"Example.vcproj", "(E524BA09-7993-4528-91A9-7E27FAA3565F)"
EndProject
Global
GlobalSection (SolutionConfiguration) = preSolution
ConfigName.0 = Checked
EndGlobalSection
GlobalSection (ProjectDependencies) = postSolution
EndGlobalSection
GlobalSection (ProjectConfiguration) = postSolution
(E524BA09-7993-4528-91A9-7E27FAA3565F). Checked.ActiveCfg =
```

```
Checked  Win32
(E524BA09-7993-4528-91A9-7E27FAA3565F). Checked.Build.0 =
Checked  Win32
EndGlobalSection
GlobalSection (ExtensibilityGlobals) = postSolution
EndGlobalSection
GlobalSection (ExtensibilityAddIns) = postSolution
EndGlobalSection
EndGlobal
```

Значно важливішим в проекті Example є файл Example.vcproj, який містить конкретні значення настройок і опису файлів, які використовуються. Точний текст Example.vcproj (файлу налаштувань для компіляції і збірки найпростішого не-WDM драйвера checked-версії в середовищі Visual Studio 7 Net) наводиться нижче:

```
<? xml version = "1.0" encoding = "windows-1251"?>
<VisualStudioProject ProjectType = "Visual C + +"
Version = "7.00"
Name = "Example"
SccProjectName = ""
SccLocalPath = "">
<Platforms> <Platform Name="Win32"/> </ Platforms>
<Configurations> <Configuration Name = "Release  Win32"
OutputDirectory = ". \ Checked"
IntermediateDirectory = ". \ Checked"
ConfigurationType = "2"
UseOfMFC = "0"
ATLMinimizesCRunTimeLibraryUsage = "FALSE"
CharacterSet = "1">
<Tool Name = "VCCLCompilerTool"
AdditionalOptions = "/ Zel-cbstring / QIfdiv-/ QIf / Gi-/ Gm-/ GX"
Optimization = "0"
EnableIntrinsicFunctions = "FALSE"
OmitFramePointers = "TRUE"
OptimizeForProcessor = "2"
AdditionalIncludeDirectories = "C: \ WinDDK \ 2600 \ inc \ ddk \ w2k;
C: \ WinDDK \ 2600 \ inc \ w2k; C: \ WinDDK \ 2 600 \ inc \ crt "
PreprocessorDefinitions = "_X86_ = 1; i386 = 1;
CONDITION_HANDLING = 1; NT_UP = 1;
NT_INST = 0; WIN32 = 100; _NT1X_ = 100; WINNT = 1;
_WIN32_WINNT = 0x0400; WIN32_LEAN_AND_MEAN = 1;
DEVL = 1; DBG = 1; FPO = 0 "
IgnoreStandardIncludePath = "TRUE"
StringPooling = "TRUE"
ExceptionHandling = "TRUE"
RuntimeLibrary = "O"
StructMemberAlignment = "4"
BufferSecurityCheck = "FALSE"
EnableFunctionLevelLinking = "TRUE"
PrecompiledHeaderFile = ". \ Checked / Example.pch"
AssemblerListingLocation = ". \ Checked /"
ObjectFile = ". \ Checked /"
ProgramDataBaseFileName = ". \ Checked \ Example.pdb"
WarningLevel = "3"
SuppressStartupBanner = "TRUE"
DebugInformationFormat = "1"
CallingConvention = "2"
```

```
CompileAs = "0"
ForcedIncludeFiles = "warning.h" />
<Tool Name="VCCustomBuildTool"/>
<Tool Name = "VCLinkerTool" AdditionalOptions = ""
AdditionalDependencies = "hal.lib ntoskrnl.lib int64.1ib
msvcrt.lib "
OutputFile = ". \ Checked \ Example.sys"
Version = "5.0"
LinkIncremental = "1"
SuppressStartupBanner = "TRUE"
AdditionalLibraryDirectories = "C: \ WinDDK \ 2 600 \ lib \ w2k \ i386"
IgnoreAllDefaultLibraries = "TRUE"
ProgramDatabaseFile = ". \ Checked / Example.pdb"
GenerateMapFile = "TRUE"
MapFileName = "Example.map"
StackReserveSize = "262144"
StackCommitSize = "4096"
OptimizeReferences = "2"
EnableCOMDATFolding = "2"
EntryPointSymbol = "DriverEntry"
SetChecksum = "TRUE"
BaseAddress = "0x10000"
ImportLibrary = ""
MergeSections = ". Rdata =. text"
TargetMachine = "1" />
<Tool Name = "VCMIDLTool"
MkTypLibCompatible = "TRUE"
SuppressStartupBanner = "TRUE"
TargetEnvironment = "1"
TypeLibraryName = ". \ Checked / Example.tlb" />
<Tool Name="VCPostBuildEventTool"/>
<Tool Name="VCPreBuildEventTool">
<Tool Name="VCPreLinkEventTool"/>
<Tool Name="VCResourceCompilerTool"/>
<Tool Name="VCWebServiceProxyGeneratorTool"/>
<Tool Name="VCWebDeploymentTool"/>
</ Configuration>
</ Configurations>
<Files> <Filter Name="Header Files" Filter=".h">
<File RelativePath=".\Driver.h"> </ File>
</ Filter>
<Filter Name="Source Files" Filter=".c;.cpp">
<File RelativePath="Init.cpp"> </ File>
</ Filter>
</ Files>
<Globals> </ Globals>
</ VisualStudioProject>
```

Наведений вище текст переформатований, оскільки розташування слів дещо відрізняється від їх розміщення в оригіналі .vcproj-файлах, що генеруються в середовищі Visual Studio 7 Net. Значення рядкових параметрів AdditionalIncludeDirectories і PreprocessorDefinitions обов'язково повинні бути записані в один рядок.

Слід відзначити, що особливу важливість для компіляції мають значення параметрів: IgnoreStandardIncludePath скасовує стандартні шляхи для виявлення

заголовків файлів, які явно задані у параметрі AdditionalIncludeDirectories, значення AdditionalOptions і PreprocessOrDefinitions рекомендується повторити в точності, CallingConvention визначає _stdcall.

З параметрів збирання слід зазначити параметри IgnoreAllDefaultLibraries, який скасовує використання бібліотек, що призначаються у Visual Studio за замовчуванням, AdditionalLibraryDirectories і AdditionalDependencies визначають використовувані бібліотеки – в даному випадку для збірки не-WDM драйвера під Windows 2000, обов'язково слід вказати BaseAddress = 0x10000 та параметр SetChecksum повинен бути "TRUE".

Всі ці параметри можна налаштувати інтерактивно і в самому інтегрованому середовищі Visual Studio, однак, потім рекомендується порівняти вміст файлу .vcproj з наведеним текстом, намагаючись отримати повний збіг.

Компіляція та збирання драйвера утилітою Build пакету DDK

В тому варіанті, як поставляється пакет DDK, дуже просто використовувати компілятор і редактор зв'язків цього пакету. Для цього слід вибрати в меню запуску програм Пуск / Програми / ... запуск відповідного середовища, в результаті чого з'явиться консольне вікно, для якого вже автоматично будуть належним чином встановлені змінні оточення. У тому випадку, якщо у розробника є файли makefile, sources, які описують процес складання даного конкретного драйвера, а пакет DDK встановлено вірно, необхідно лише перейти в робочу директорію проекту командою cd і ввести команду build (див. рис. 8.1). Зрозуміло, що в разі помилок компіляції або збирання висновок буде містити й їх діагностику.

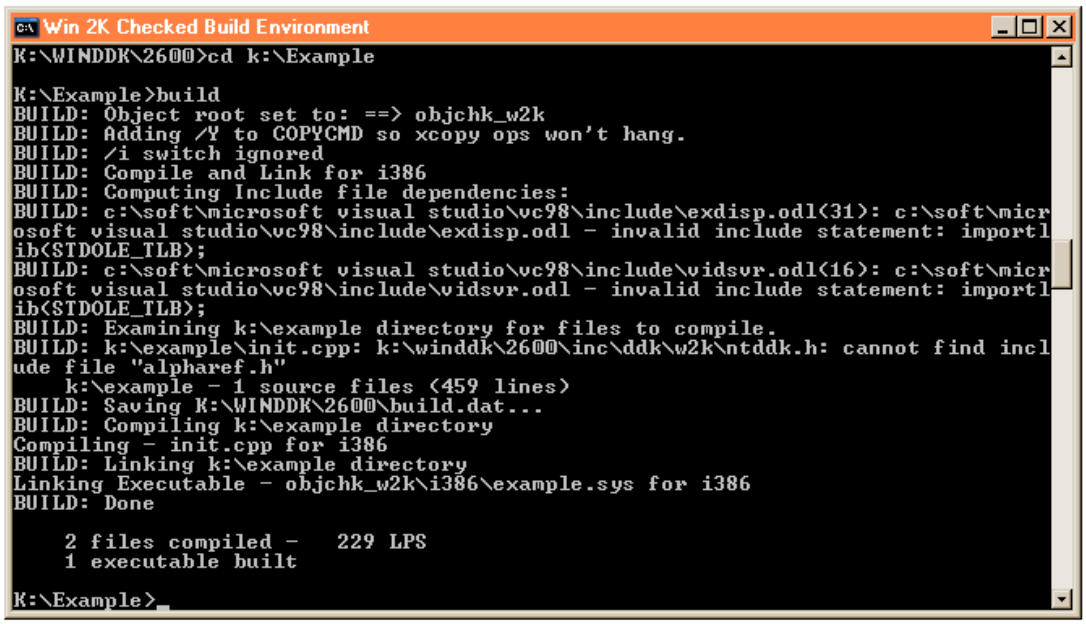

Рис. 8.1. Робоче вікно збирання драйвера під Windows 2000 DDK версії checked

8.1.2. Програма Depends

Програма Depends призначена для перегляду викликів додаткових бібліотек. Програма Depends була створена в 1996 році і раніше постачалася у складі Visual Studio 6. Тепер вона є частиною Platform SDK.

Скріншот (знімок екрану), представлений на рис. 8.2, виконаний для перегляду дерева викликів відомого драйвера GiveIo.sys. Цей драйвер розблокує доступ до портів введення/виведення з додатків режиму користувача при роботі під Windows NT, використовуючи при цьому недокументовані можливості Windows.

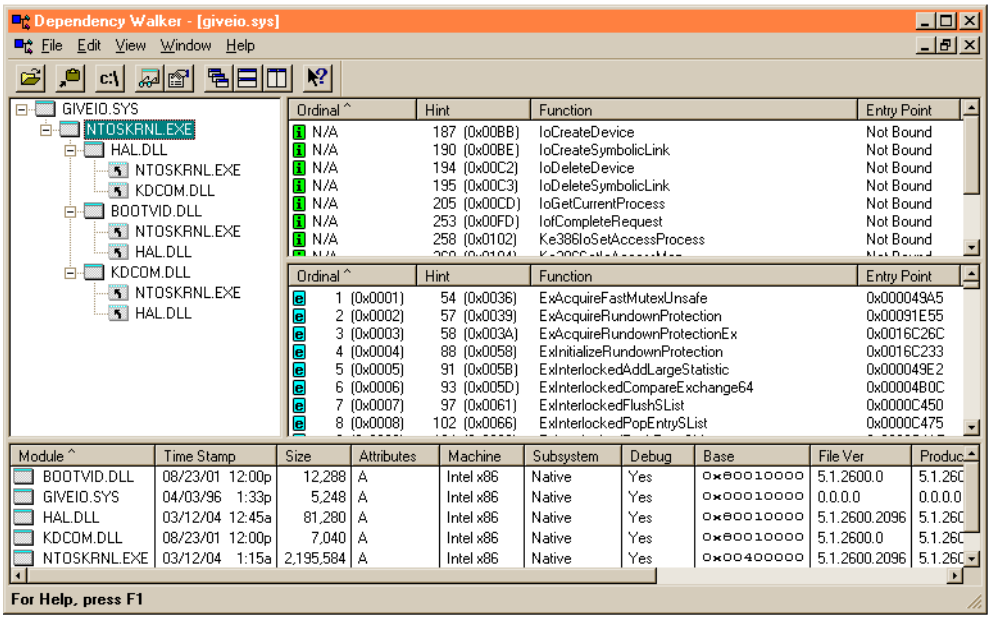

Рис. 8.2. Програма Depends для драйвера GiveIo

З рис. 8.2 видно, що драйвер звертається до функції NTOSKRNL.EXE, де в алфавітному порядку першими є виклики IoCreateDevice, IoCreateSymbolicLink, IoDeleteDevice, IoDeleteSymbolicLink та ін.

Програма може бути використана для перегляду викликів, які виконуються із драйверів, виконуваних файлів (exe-файлів) і динамічних бібліотек. Програма працює і під Windows 98.

8.1.3. Програми від SmidgeonSoft

Якщо програми Свена Шрайбера цікаві для дослідження Windows і дозволяють використання свого коду в нових розробках, будучи лише консольними додатками, то програми фірми SmidgeonSoft є завершеними програмними продуктами зі зручним графічним інтерфейсом і реалізовують практично всі можливості з числа згаданих раніше.

8.1.4. Програма PEBrowseProfessional Interactive

Являє собою програму перегляду “начинки” виконуваних exe, dll, sys модулів (файлів PE-формату, Portable Executable File Format) і дизасемблер, включаючи навіть інтерактивний відладчик програм у режимі користувача. Дана програма істотно поступається програмі PE Explorer продукту SmidgeonSoft та програмі IDA Pro. Проте, вона є потужнішою та зручнішою за програму Depends і представляє інтерес при вивченні особливостей бінарних драйверних модулів (готових драйверів). Крім того, спрощений варіант PEBrowse Professional дозволяє переглядати вміст lib-файлів.

8.1.5. Програма NTDevices

Програма NTDevices, представлена на рис. 8.3, являє собою істотно більш зручний у використанні аналог згаданої раніше програми DeviceTree, призначеної для дослідження стека драйверів в ОС. Дозволяє також переглядати всі символьні посилання, що зареєстровані в системі, і список об’єктів пристроїв, які мають підключення (attached) об’єкти інших пристроїв.

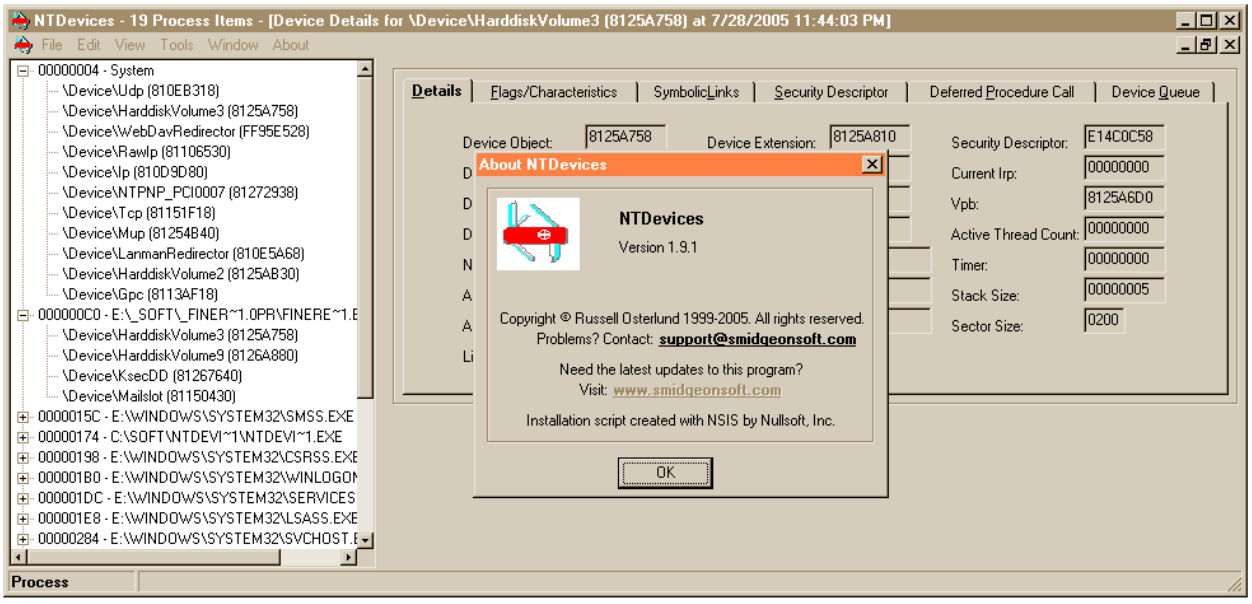

Рис. 8.3. Програма NTDevices

8.2. Особливості тестування та налагодження драйверів

8.2.1. Тестування та налагодження драйверів

Жоден складний фрагмент програмного коду не може вважатися безпомилковим. Існує твердження, що якщо в програмі немає помилок, то помилка в компіляторі, оскільки він не виявив всіх помилок в програмі.

Помилки не можна знищити, їх можна лише обмежити. Все це стає найбільш явним, коли програмне забезпечення починає взаємодіяти з програмами та апаратурою інших постачальників, часто представляють чорний ящик.

Тестування можна розділити на два етапи. Перший етап первісного емпіричного накопичення даних про поведінку сформованої конфігурації апаратури і програмного забезпечення. У досвідчених розробників цей етап коротше і проходить раціональніше.

Другий етап зобов'язаний узагальнити отримані на першому етапі відомості, побудувати прийнятну модель виникнення проблем і провести їх поетапну ліквідацію. Якщо другий етап не справляється з такою постановкою питання і перетворюється в перший, то вся робота перетворюється на марні пошуки.

Зрозуміло, поетапне тестування значно ефективніше, ніж тестування після закінчення розробки системи в цілому. Хоча такий підхід потребує більшого набору фрагментарних тестів, дієвість цього підходу незаперечна. Хоча б з тієї простої причини, що дата завершення тестування при такому підході більш передбачувана, ніж для продукту, який жодного разу не тестувався по частинах.

Зазвичай тестування поділяють на два види: "приростає" тестування, з поступовим ускладненням та більш формальне регресивне тестування, коли відбувається відкат на попередній крок у випадку невдачі.

У міру розширення проекту, ці збережені попередні тести дозволять упевнитися, що зміни не внесли помилок в раніше розроблені фрагменти.

Найчастіше апаратура розробляється паралельно із програмним забезпеченням. При ранньому частковому тестуванні дефекти в апаратурі виявляються раніше і, відповідно, залишається більше часу на виправлення помилок в проектуванні.

Небагато фірми можуть дозволити собі мати окремі команди розробників і випробувачів. Таким чином, автор драйвера повинен часто паралельно розробляти і драйвер, і поетапні тести для нього. Перевагою такого "унітарного" підходу є те, що автор у курсі граничних параметрів драйвера та пристрої. Недолік полягає в тому, що оптимально влаштоване мислення розробника не планує тестів на деякі помилкові ситуації, які рідко зустрічаються, що згодом може виявитися серйозною проблемою конструкції та програмного забезпечення.

Зрозуміло, повинна бути встановлена гарна дисципліна для забезпечення достатнього часу і для розробки, і для тестування. Скорочення стадії тестування для прискорення загального графіка виконання – погана послуга будь-яким проектам.

Загалом, тести для драйвера, якщо вважати, що апаратне забезпечення тестується додатково, можна поділити на такі категорії:

1. Тести на нормальну реакцію повинні підтверджувати повноту і точність функцій драйвера.

2. Тести на помилкові вхідні значення, чи правильно реагує драйвер на дії, які, наприклад, не мають жодного сенсу. “Хибний” вплив може також виникати у поганому наборі даних, що надійшов в запит користувача.

3. Тести граничних умов визначають абсолютні межі функціонування драйвера і пристрою. Не виключено, що в силу роботи драйвера в системі кількісні параметри виявляться іншими, ніж заплановані.

4. Тести на граничне навантаження перевіряють драйвер і пристрій при високих рівнях активності.

5. Тести на функціонування в умовах обмежених ресурсів, тобто робота при обмеженій доступності центрального процесора, обмеженість обсягу доступної оперативної пам’яті.

Фірма Microsoft пропонує тести на апаратну сумісність (Hardware Compatibility Tests, HCT), які є офіційними тестами для апаратури з приводу можливої роботи під Windows 2000/XP/2003. Набір включає наступні тести:

– загальні системні тести, які випробовують центральний процесор, послідовні і паралельні порти, материнську плату, клавіатуру і засоби підтримки широкого ряду апаратних абстракцій;
– тести з перевірки специфічних типів апаратного забезпечення, а саме – дисплеїв, мультимедійних пристроїв, мережевих інтерфейсів, накопичувачів, амоніфікації (?), SCSI пристроїв тощо;
– загальні тести з оцінки роботи системи під час високих навантажень на систему, ресурси і процесів введення/виведення.

Тестування проходить під управлінням тест-менеджера з графічним інтерфейсом, який автоматизує проходження тестів і звіт про результати. Навіть якщо клас апаратури, для якої розробляється драйвер, не пройшов його випробування HCT, то все одно цей набір тестів може послужити засобом для з’ясування того, як буде працювати новий драйвер в умовах дії високого навантаження на систему.

8.2.2. Цифрове підписання драйверів

Велика кількість драйверів сторонніх постачальників (третіх сторін, third-party companies) поставляється в складі CD дистрибутиву Microsoft Windows. Для того, щоб бути включеним у цей програмний продукт, постачальник драйвера повинен виконати деякі вимоги, сформульовані Microsoft. Вимоги Microsoft забезпечити досягнення двох основних цілей:

– забезпечити роботу під управлінням Windows максимально різноманітної і численної апаратури за рахунок єдиного способу розповсюдження дистрибутива;
– забезпечити стабільність роботи нових драйверів без порушення будь-яких важливої області системи.

Оскільки драйвери працюють в режимі ядра, у них є можливість привести систему до краху. Також нестабільність системи, швидше за все, буде сприйнята

користувачам на необхідність огляду, очевидно, що в інтересах Microsoft збільшувати список сертифікованих пристроїв і сертифікованих драйверів для своєї ОС.

З розумінням, сумісність ОС Windows 2000/XP/2003 із великою кількістю апаратури, більшою ніж інші ОС, є сильним фактором стимулювання продажів. Отже, Microsoft безпосередньо контактує з виробниками апаратного забезпечення на предмет отримання якісних і протестованих нових версій драйверів. Аж до того, що конфігурації Windows Server 2003 Datacenter Edition постачаються виробниками ЕОМ повністю разом з комп'ютером.

Для досягнення вищезазначених цілей, в Microsoft була створена спеціальна група – the Windows Hardware Quality Labs (WHQL), яка проводить сертифікацію апаратури і драйверів програм. Вигоди, які дає участь в програмі сертифікації для виробників апаратного і програмного забезпечення, полягають у:

– використанні логотипу Windows як знака відповідності сертифікованого пристрою і апаратури і програмного забезпечення до нової ОС, що поставляється;
– включенні в офіційний перелік рекомендованого і сертифікованого обладнання для версій ОС розробки Microsoft;
– автоматичному поширенні драйверного забезпечення разом з майбутніми поставками нових версій Windows;
– отриманні цифрового підпису.

Як частина результату участі в програмі WHQL, драйвер отримує цифровий підпис (digital signature), захищений криптографічними методами, який дозволяє Windows 2000/XP/2003 встановлювати нормально драйвер без видачі відповідного попередження про "несертифікованість" небезпеки. Цифровий підпис є, на думку Microsoft, засобом підтвердження непорушності коду та гарантії походження і того, що встановлений код є оригінальним драйверним кодом від постачальника. З використанням вашого цифрового підпису сусідують два моменти:

1. Наявність файлу-каталогу (з розширенням .cat), який включається до поширюваного пакету драйверного програмного забезпечення. Він містить цифровий підпис, наданий Microsoft.
2. Запис у inf-файлі у секції [Version], де вказується посилання на cat-файл.

Сам цифровий підпис жодною мірою не виробляє модифікацію коду драйверного програмного забезпечення.

8.3. Усунення несправностей

8.3.1. Усунення несправностей WinDbg

Одним з відладчиків, який може використовувати розробник драйверів, є відладчик WinDbg, що поставляється в складі пакету DDK. Усунення

несправностей WinDbg є гібридом відладчика рівня ядра і призначеного для користувача режиму. За допомогою WinDbg можна проводити аналіз файлів “crash dump files”, а саме відображення фізичної пам’яті у момент збою, налагоджувати драйверний код шляхом трасування (виконання), аналізувати “dump file” додатків у режимі користувача.

Крім того, що WinDbg виконує свої завдання і в режимі ядра, і в режимі користувача, він також надає робочий інтерфейс і в графічній формі, і у формі командного рядка, характерної для старих відладчиком.

Однією з найбільш значних можливостей, якою володіє відладчик WinDbg, є його здатність підтримувати налагодження компонентів рівня ядра у вихідних текстах налагоджувати модулів. При цьому відладчику повинні бути доступні і вихідні тексти, файли та налагоджування символами (далі вони будуть називатися “файли ідентифікаторів”).

Однак, налагодження у вихідних текстах представляє досить істотну організаційну проблему.

По-перше, для налагодження драйвера необхідно два комп’ютери, на одному з яких працює власне відладчик, а на другому – налагоджувальний агент, відносно невеликий обсяг коду, який отримує керуючі команди з першого комп’ютера.

По-друге, для проведення налагодження буде потрібно відлагоджувальна версія ОС, для якої є відповідні файли налагоджувальних символів.

По-третє, для проведення відладки буде потрібно налагоджувальну версію ОС, для якої є відповідні файли налагоджувальних символів.

Директорії ідентифікаторів

Файли ідентифікаторів (symbol files) створюються по додатково встановлених прапорах компіляції при компіляції і збірці проекту. У них знаходяться імена локальних і глобальних змінних, адреси функцій і інформація про визначення типів даних. Приводиться також інформація про номери рядків, тому машинні інструкції можуть бути легко співвіднесені з початковим текстом. Програмні засоби Microsoft можуть надавати ці файли в декількох форматах: старішому, але типовішому, COFF (Common Object File Format) і форматі PDB (Program Database), що визначається настроювальними прапорами компіляції і збірки.

Сама ОС Windows теж поставляється з файлами ідентифікаторів, які можуть бути і на додатковому диску дистрибутива, і у складі DDK. Для ОС Windows 2000 таких файлів можна завантажити з Інтернет сервера Microsoft. Для решти версій вони розповсюджуються Microsoft в рамках підписки MSDN або у складі набору CD дисків Driver Suite. Оскільки файли ідентифікаторів змінюються при кожній новій збірці (build), то оновлення системи (service pack) потребує також і оновлення файлів ідентифікаторів, що відносяться до системних модулів.

Доступ до файлів ідентифікаторів драйвера і системних модулів виключно важливе, оскільки вони забезпечують інформацію про стек викликів і прив’язку до початкового тексту драйвера.

Коли встановлені необхідні файли ідентифікаторів, відладчику WinDbg необхідно повідомити про їх місцезнаходження. Це виконується з використанням в пункті Symbol File Path пункту меню File. Ідентифікатори ОС зазвичай встановлюються в директорії %SystemDir%\Symbols. Файли ідентифікаторів драйвера із розширенням .dbd або .pdb зазвичай зберігаються разом з бінарним файлом драйвера із розширенням .sys, .wdm.

Директорії початкових текстів

Крім файлів ідентифікаторів відладчику знадобляться файли початкових текстів драйвера (.c, .cpp, .h). Шлях до них необхідно вказати в пункті Source File Path пункту меню File.

Запуск і закінчення налагоджувальної сесії

Основне призначення відладчика WinDbg – інтерактивна відладка. Для цього необхідна наявність двох комп'ютерів: цільового, на якому буде запущений тестовий код, і хост-комп'ютер, на якому буде проводитися розробка і працює сам WinDbg. Істотно тут те, що для управління і моніторингу активності на цільовому комп'ютері використовується інтерфейс послідовних портів. На рис. 8.4 показано з'єднання цільового і хост-комп'ютерів із розміщення файлів на них. Для успішної відладки необхідно, щоб всі потрібні файли опинилися на своїх місцях.

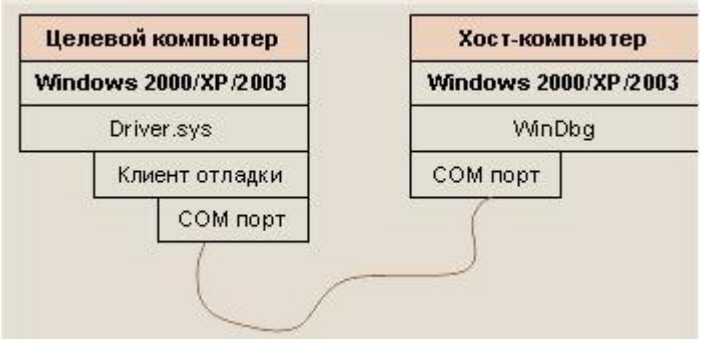

Рис. 8.4. Інтерактивна відладка з використанням WinDbg

Як видно з рис. 8.4, послідовність дій така:

1. Інсталювати бінарний файл драйвера на цільовій машині, тут не потрібно встановлювати файли ідентифікаторів або початкового тексту.
2. На хост-комп'ютері необхідно мати файли ідентифікаторів драйвера і файли ідентифікаторів ОС, встановленої на цільовому комп'ютері. Непогано було б, щоб на двох комп'ютерах була б встановлена одна і та ж версія системи, і однакові оновлення Service pack, проте це не є жорсткою умовою.
3. На хост-комп'ютері запускається WinDbg. Відбувається установка шляхів до початкового коду і до файлів ідентифікаторів. Ці файли повинні відповідати бінарному файлу відладжуваного драйвера, який знаходиться на цільовому комп'ютері.
4. Необхідно вибрати відладку в режимі ядра в меню View закладки Options Kernel Debugger у відладчику WinDbg.

5. Необхідно встановити прийнятні значення номера COM-порту і швидкості передачі із згаданого діалогового вікна.

6. Слід вибрати кнопку GO або ввести команду g у вікні командного рядка відладчика для того, щоб перевести WinDbg в режим очікування з'єднання з цільовим комп'ютером.

7. Необхідно виконати перезавантаження цільового комп'ютера, вирішивши використання клієнта відладки. Коли завершиться перезавантаження системи на цільовому комп'ютері, з'єднання буде встановлено. У командному вікні WinDbg буде виведене відповідне повідомлення.

Хост-комп'ютер тепер має повний контроль над цільовим комп'ютером. Можна встановлювати точки переривання, навіть в тому коді, який ще не був завантажений в пам'ять ядра.

Щоб розірвати налагоджувальну сесію, необхідно виконати наступне:

1. Зробити паузу, натиснувши CTRL-C в командному вікні WinDbg.
2. Слід вибрати Edit / Breakpoints у відладчику WinDbg, потім виконати Clear All. Важливо виконати очищення точок останову до припинення налагоджувальної сесії.
3. Слід вибрати Run / Go для того, щоб вирішити подальшу роботу на цільовому комп'ютері.
4. Вийти з WinDbg (Alt-F4).

Цільовий комп'ютер може відреагувати деякою затримкою на розрив з'єднання, можливо через те, що процедури KdPrint і DbgPrint більш не мають одержувача своїх повідомлень і їм потрібен певний час для відробітку цієї ситуації.

8.3.2. Усунення несправностей SoftIce

Налагодження драйвера режиму ядра на одному комп'ютері забезпечує потужний відладчик SoftIce, розроблений фірмою CompuWare Corporation, який входить до складу пакету DriverStudio. Зовнішній вигляд інтерфейсу відладчика зберігся ще з часів MS DOS, проте, при деякому тренуванні з використанням мнемонічних команд, це не приносить незручностей, тим більше що SoftIce управляється мишкою, включаючи мишку з колесом.

Параметри запуску встановлюються конфігуратор DSConfig, який визначає спосіб запуску SoftIce (при завантаженні, по COM, за IP-адресою, запуск на вимогу на даному комп'ютері), настройки розміру вікна, настройки мишки.

Під час налаштування завантаження на вимогу слід явно запустити з меню команд Пуск / ... / Start SoftIce. Ця команда завантажує відладчик, а власне його активація здійснюється комбінацією Ctrl-D.

У тому випадку, якщо потрібно виконати налагодження драйвера, після завантаження SoftIce слід завантажити файл драйвера (для налагодження у вихідних текстах – обов'язково налагоджувальну версію) програмою loader32 командою Пуск / ... / Symbol Loader. Лише після цього слід активувати відладчик натисканням Ctrl-D.

За допомогою команди h у вікні активованого відладчика можна ознайомитися з повним набором і форматом команд. Це ж можна дізнатися і з доданої документації.

За допомогою певних команд нескладно відшукати драйвер в пам'яті, якщо відоме ім'я драйвера або об'єкта пристрою, після чого можна встановлювати точки переривання в коді драйвера. Потім слід деактивувати відладчик повторним натисканням Ctrl-D. У момент, коли програма для режиму користувача викличе драйвер і, відповідно, досягне цієї точки переривання, відладчик активізується самостійно. У вікні відладчика буде видно потрібний фрагмент драйвера.

У той час, коли активно вікно відладчика SoftIce, активність ОС завмирає: не оновлюються свідчення годин, не працює обмін через clipboard і т.п.

Слід зазначити, що при роботі з SoftIce не слід вдаватися до складних мишок, наприклад, з чотирма коліщатками та навіть з двома, і мишки USB. Пояснюється це тим, що, будучи справжнім відладчиком режиму ядра, SoftIce намагається працювати з пристроями введення напряму, не завжди розуміючи нові екзотичні пристрої.

8.4. Запобігання збоїв під час роботи драйверів

Якщо тестування сигналізує про наявність помилок, то складнішою проблемою є локалізація їх джерела. Зрозуміло, драйвери можуть відмовлятися працювати багатьма цікавими способами. Навряд чи можливо або слід було б приводити загальний список причин збоїв, але перерахувати деякі загальні типи драйверних патологій все-таки має сенс.

8.4.1. Апаратні проблеми

Розробник програмного забезпечення переконані в тому, що апаратура є джерелом проблем. Вірогідність того, що це дійсно так, тим вище, чим новіша апаратура. Симптомами апаратних проблем є:

- помилки при передачі даних;
- коди стану пристрою сигналізують про помилку;
- пристрій не реагує належним чином на команди;
- сигнали переривання не поступають, або вони помилкові.

Причина згаданих відхилень може бути просто в недостатній документованості поведінки пристрою. Розробник апаратури після деяких роздумів змінив конструкцію, але не виправив документацію і не повідомив про зміни розробника драйвера, можливо, порахувавши їх незначними. Можуть існувати маловідомі або недосліджені обмеження на порядок проходження команд – як в сенсі послідовності, так і в сенсі їх тимчасових діаграм. Апаратні прошивки (програми, що завантажуються в обслуговувані драйвером пристрою) також можуть містити помилки. Можуть виникати збої в шинних

протоколах, причому із-за неперіодичних збоїв інших пристроїв, підключених до даної шини.

8.4.2. Програмні проблеми

Оскільки драйвер працює в режимі ядра, для нього вельми нескладним завданням є “обвалення” всієї ОС. Найбільш складними для трасування сценаріями є операції DMA, в яких некоректно встановлені регістри відображення (mapping registers). Дані записуються пристроєм у випадкові області пам’яті, і відбувається збій, винними здаються абсолютно інші підсистеми.

До основних програмних проблем можна віднести:

1. *Витік ресурсів*. ОС ніколи не контролює, як драйвер використовує ресурси і як він їх повертає після закінчення роботи. Коли драйвер завершує роботу і вивантажується, саме на ньому лежить вся відповідальність за звільнення всіх будь-коли задіяних ним ресурсів. Витік пам’яті може відбуватися і в той час, коли драйвер ще продовжує працювати. Наприклад, якщо він періодично проводить тимчасове виділення пам’яті під свої потреби і “забуває” їх звільнити. Драйвери, які займаються генерацією додаткових пакетів IRP, можуть “забувати” виконати очищення цих областей пам’яті. Витоки ресурсів приводять до падіння продуктивності системи і, кінець кінцем, до її фатального збою.

Пошук витоку ресурсів і в режимі користувача є непростим завданням, а в режимі ядра він граничить з магією. Можливо, тут може виявитися корисною методика виділення блоків пам’яті, помічених дескрипторами, за допомогою виклику ExAllocatePoolWithTag. При аналізі системної пам’яті після фатального збою, по збереженому на диску стану оперативної пам’яті, ці ідентифікатори дозволяють визначити, де блоки були виділені, який їх розмір і, що найважливіше, яка підсистема виконала їх виділення.

2. *Гальмування програмних потоків*. Одна з помилок, що приводить до зупинки програмних потоків, полягає в тому, що драйвер не завершує обробку IRP пакетів. В результаті, потік додатку користувача, що зробив запит, так і залишається в стані очікування. Це може бути обумовлено декількома причинами.

Найпростіша помилка полягає в тому, що драйвер з якоїсь причини не виконує виклик IoCompleteRequest. В результаті, надісланий пакет IRP ніколи не повертається диспетчерові введення/виведення. Іноді не така очевидна необхідність зробити виклик IoStartNextPacket (при використанні черг IRP пакетів). Проте, навіть якщо не існує запитів, які чекають обробки, драйвер повинен виконати цей виклик, оскільки тільки таким чином об’єкт пристрою буде “помічений” як той, що простоює. Без цього виклику нові пакети IRP розміщуються в чергу очікування обробки, так і не поступаючи в процедуру StartIo драйвера.

Інша логічна помилка, що полягає в спробі рекурсивно отримати об’єкт синхронізації або інші ресурси виконавчої підсистеми, може зупинити програмний потік драйвера в одній з його робочих процедур. Можливо, що інший фрагмент коду повинен був звільнити об’єкт синхронізації, але так і не зміг “дістатися” до потрібного місця або “пішов по іншій дорозі”. Подальші запити тільки посилюють ситуацію.

Аналогічно, DMA драйвери можуть зупинитися в точці, де вони зробили запит на володіння об’єктом адаптера. Драйвери, які управляють складними контролерами, можуть створити схожі проблеми у випадку, якщо вони не звільняють об’єкт контролера.

На жаль, не існує безумовно хорошого способу вирішення таких проблем. Іноді корисно визначити якусь структуру даних, в якій всі власники об’єктів синхронізації “наголошуватимуться”, коли вони щось використовують, і видаляти ці відмітки, коли використання відповідного об’єкту синхронізації припиняється. Зрозуміло, такому прийому необхідні будуть додаткові зусилля, але він вельми ефективний у виявленні джерела проблем.

Деколи помилки драйвера можуть викликати блокування всієї системи. Наприклад, фатальні взаємоблокування можуть викликати некоректне перехресне використання декількох об’єктів спін-блокувань або спроби повторно отримати спін-блокування на однопроцесорній платформі. Нескінченні цикли з коду процедур обслуговування переривання або процедур DpcForIsr можуть привести до аналогічного результату. Кращим рішенням в даній ситуації було б здійснення інтерактивної відладки драйвера.

3. *Проблема пріоритетів часу виконання*. Опис всіх системних викликів приводиться з обов’язковою вказівкою, на якому рівні IRQL (пріоритетів рівня ядра) може працювати програмний код у момент звернення до них. Оскільки пріоритет зухвалого коду за умовчанням переходить на процедури (які, можливо, не повинні ніколи працювати на цьому рівні), що викликаються, то ОС ретельно відстежує, щоб рівень пріоритету, який “передався” її компонентам, відповідав би встановленим правилам. Найчастіше жертвою такого дотримання правил стає високопріоритетний код, що звертається до сторінкової пам’яті, яка виявилася скинутою на диск. Система не дозволяє вступати в роботу своїм функціям, оскільки відступ від даного правила вважається шкідливим для системи.

Розробник драйвера повинен завжди стежити за пріоритетом поточного програмного коду і документованим рівнем IRQL системних функцій, що викликаються. Інакше можливий (а іноді – просто неминучий) крах ОС, а відповідь доведеться шукати вже в crach dump файлі.

Показовим випадком неправильного використання пріоритетів є виклик іншого драйвера за допомогою IoCallDriver, який формально може бути виконаний на рівнях IRQL APC_LEVEL і DISPATCH_LEVEL. Підвищивши поточний пріоритет потоку перед викликом IoCallDriver до значення, наприклад, DISPATCH_LEVEL, драйвер, швидше за все, організовує блокування: якщо драйвер, що викликається, повинен працювати на рівні

PASSIVE_LEVEL, то він не може зробити потрібні йому виклики, поки працює потік, що викликав його, який чекає повернення управління з IoCallDriver. Система приходить в непрацездатний стан, інакше кажучи “замерзає”.

4. *Відстежування помилок*. Дослідження показують, що помилки не розподіляються рівномірно за всім програмним кодом, а мають тенденцію концентруватися в декількох специфічних процедурах, зазвичай, майже пропорційно їх функціональній складності. Ретельне протоколювання дозволяє ідентифікувати ці процедури для того, щоб тримати їх згодом “на замітці”. Найважливішим моментом відладки є відтворення помилкових ситуацій, де важливе значення має протоколювання. Добре виконане протоколювання дозволяє зрозуміти проблеми, що виникають в конфігурації проекту, впорядковувавши першу фазу тестування – етап накопичення спостережень. Крім того, виявляються проломи у власне стратегії тестування.

8.5. Загальні прийоми відладки

Часто проблеми виправлення помилок можна легко усунути за наявності достатнього об’єму діагностичних повідомлень. Нижче приводяться прийоми, засновані якраз на цьому.

8.5.1. Установка фіксованих точок переривання

При використанні інтерактивних відладчиків не знайдеться багато причин встановлювати фіксовані точки переривання усередині коду драйвера (hard breakpoints). Проте, це можна зробити, якщо скористатися двома функціями:

```
VOID DbgBreakPoint();
VOID KdBreakPoint();
```

Виклик KdBreakPoint представляє собою макровизначення, яке визначає умовну компіляцію з метою виконати виклик DbgBreakPoint. Це макровизначення не виконує даного виклику, якщо виконана релізна (без налагоджувальних інструкцій) збірка драйвера (free build).

Часто Windows дає фатальний збій з повідомленням KMODE_EXCEPTION_NOT_HANDLED в тому випадку, якщо драйвер застосував фіксовану точку переривання (через згадані виклики), але у цей момент клієнт відладки був недоступний. Якщо драйвер дістався до такої точки переривання, і не опинилося відладчика, підключеного до послідовного порту, то драйвер “висне”. В деяких випадках, ситуація може бути виправлена запуском відладчика на хост-комп’ютері.

8.5.2. Проміжне виведення на екран

Робити повідомлення з надр відладжуваного коду за допомогою функції printf – це метод з давніми і славними традиціями. Його іноді називають “генерацією проміжного виведення”. Фактично, немає таких помилок, які не можна було б знайти, якщо застосувати такий метод в достатньому об’ємі.

І хоча даний метод сьогодні не досить поширений, як відладка з використанням точного переривання в інтерактивних відладчиках, проте, він може бути дуже корисним при пошуках збоїв, пов’язаних з тимчасовими труднощами драйвера (наприклад, помилках або збоях в послідовності подій, пов’язаних з пристроєм). Для генерації проміжних повідомлень використовуються дві функції DbgPrint і KdPrint. Обидві функції посилають форматовані рядки, створені на цільовому комп’ютері, відладчику WinDbg, що працює на хост-комп’ютері.

Як було сказано раніше, збирати налагоджувальні повідомлення можна і за допомогою DebugView.

Виклик KdPrint насправді є макровизначенням і перетворюється на невиконувану ділянку в релізній збірці драйвера.

8.5.3. Збереження налагоджувального коду в початковому тексті драйвера

Хорошим прийомом в справі розробки драйвера є збереження налагоджувальних фрагментів коду на їх місці, навіть якщо для споживача була зібрана релізна версія. Все це нескладно зробити за допомогою директив умовної компіляції, і при поверненні до виявлення глибших помилок і повторного тестування цей прийом подасть відмінну підтримку.

Утиліта BUILD використовує символ часу компіляції DBG, який може бути використаний при складанні умовно компільованих фрагментів. У налагоджувальній версії (checked build) цьому символу привласнено значення 1, у версії free значення DBG рівне 0. При внесенні налагоджувального коду до драйвера слід обмежувати його рамками директив умовної компіляції типу #if DGB і #endif.

8.5.4. Перехоплення некоректних умов

Розробник драйвера, як і будь-який розробник, створює свій продукт, відштовхуючись від деяких початкових допущень. Важливо лише, щоб вони не були безпідставними або надмірно оптимістичними. В тому випадку, якщо виникають деякі сумніви в умовах роботи деяких важливих ділянок коду драйвера, краще підстраховатися. Наприклад, припускаючи, що якийсь фрагмент коду, буде викликаний тільки на певному рівні IRQL, автор закладає потенційну можливість майбутніх збоїв (якщо дані очікування не будуть виправдані).

Для того, щоб перехопити ці непередбачені відхилення, слід застосовувати нескладний прийом: такі допущення повинні перевірятися під час виконання, хоч би в налагоджувальній збірці драйвера. Макровизначення ASSERT і ASSERTMSG допоможуть вирішити цю задачу:

```
ASSERT( Expression );
ASSERTMSG( Message, Expression );
```

У випадку, якщо вираз Expression, розглянутий як логічний, то він буде рівний FALSE, макровизначення ASSERT проведе запис повідомлення в командне вікно WinDbg, підключеного на хост-комп'ютері для проведення відладки. Це повідомлення містить початковий код виразу, значення якого опинилося рівним FALSE, ім'я файлу (у якому був початковий текст даного фрагмента) і номер рядка, де було викликано макровизначення ASSERT. Потім пропонується зробити вибір, чи зробити переривання в місці даного макровизначення ASSERT, ігноруючи причину зупинки, або перервати процес або потік, в якому "спрацював" ASSERT (що абсолютно аналогічно додатку ASSERT в Visual Studio).

Макровизначення ASSERTMSG демонструє таку ж саму поведінка, з тією лише різницею, що в повідомлення включається вміст Message, що є рядком. На відміну від debug print функцій, описаних раніше, ASSERTMSG не допускає виведення формату даних в стилі printf.

Слід також звернути увагу на те, що обидва макровизначення використовують умовну компіляцію, унаслідок чого у версії "free" воно абсолютно зникає. Отже, абсолютно неприпустимо розміщати будь-який виконуваний код всередину цих макровизначень – у версії "free" вони зникнуть зовсім.

Крім того, в основі згаданих вище макровизначень лежить використання функції RtlAssert, яка в "free" версіях Windows 2000/XP/2003 перетворюється на не "виконувану ділянку". Отже, щоб спостерігати наслідки помилок в макровизначеннях ASSERT і ASSERTMSG слід тестувати драйвер під налагоджувальною (checked) версією Windows.

8.5.5. Використання діагностичних callback-функцій

Діагностичні зворотні виклики є процедурами драйвера, який той пропонує для виклику в моменти початку фатальних збоїв системи. Ці процедури пропонують зручний спосіб захоплення налагоджувальної інформації під час збою. Крім того, вони можуть бути використані для того, щоб перевести апаратуру в певний стан перед тим, як система остаточно "звалиться". Для використання діагностичних зворотних викликів потрібно виконати наступне:

1. У процедурі DriverEntry необхідно виконати виклик функції KeInitializeCallbackRecord для виконання настройки структури KBUGCHECK_CALLBACK_RECORD. Місце для зберігання цієї закритої

структури повинне бути виділене в несторінковому пулі (причому повинно міститися у недоторканості до моменту її дерегистрації, що виконується за допомогою функції KeDeregisterBugCheckCallback в процедурі Unload даного драйвера).

2. У DriverEntry необхідно виконати виклик KeRegisterBugCheckCallback для підключення драйвера до механізму повідомлення про помилкові ситуації. Аргументами цього виклику буде покажчик на структуру типу KBUGCHECK_CALLBACK_RECORD, адреса callback-функції, що надається драйвером, адреса і розмір буфера, що надається драйвером, і рядок, який буде використаний для ідентифікації буфера. Місце для буфера повинно бути виділене в несторінковому пулі, де також повинно міститися в недоторканності до моменту виклику функції KeDeregisterBugCheckCallback.

3. В тому випадку, якщо була виявлена помилка, система виконує виклик зареєстрований в п. 2 callback-функції, якою буде передана адреса буфера і його розмір. Робота викликаної функції полягає в тому, що вона повинна заповнити наданий буфер інформацією, яку визнає необхідною, і яка не змогла б іншим чином знайти своє віддзеркалення в crash dump файлі, наприклад, внутрішній вміст регістрів обслуговуваного пристрою.

4. При аналізі crash dump файлу за допомогою WinDbg цю інформацію можна вивести на екран, якщо скористатися командою !bugdump.

Існують деякі обмеження на дії, дозволені реєстрованій callback-функції. Під час роботи вона не може отримувати які-небудь системні ресурси (наприклад, пам’ять). Вона також не повинна використовувати об’єкти синхронізації. Проте дозволено робити виклики процедур ядра, які не порушують вказаних обмежень, і виклики процедур HAL рівня, наприклад, для того, щоб дістати доступ до регістрів пристрою.

8.5.6. Виявлення витоків пам’яті

Витоки пам’яті є найжорстокішою патологією програмних кодів. Драйвери, які отримали для себе простір в одному з пулів пам’яті, а потім забули його звільнити, можуть привести до серйозної деградації системи і, через деякий час, до її фатального збою. Використання тегового механізму виділення пам’яті може надати істотну допомогу у виявленні витоків:

1. Необхідно замінити виклики ExAllocatePool на виклики ExAllocatePoolWithTag. Додатковий аргумент, що є чотирьохбайтною величиною (4 символи), використовується для того, щоб помітити знов виділений блок цим значенням (тегом).

2. Необхідно запустити драйвер під налагоджувальною версією (checked build) Windows. Підтримка трасування сторінок пулу є дорогим “задоволенням”, тому доступно воно тільки в налагоджувальних версіях Windows.

3. Коли виконується аналіз crash dump файлу або за ситуації, коли досягнута точка переривання при відладці “живого” драйвера, слід скористатися

командами !poolused або !poolfind для того, щоб ознайомитися із станом пулів пам'яті. Ці команди сортують області пулів за значенням тегів і висвічують різні статистичні дані про використання пам'яті. Слід пам'ятати, що у відсутності файлів налагоджувальних символів вказані команди WinDbg не працюють.

Легкий спосіб повсюдної заміни викликів ExAllocatePool на виклики ExAllocatePoolEx полягає в тому, щоб спочатку використовувати фрагменти умовної компіляції, наприклад:

```
#if DBG==1
  #define ALLOCATE_POOL(type,size)\
              ExAllocatePoolWithTag((type),(size),'1234')
#else
      #define ALLOCATE_POOL(type,size)
ExAllocatePool((type),(size))
#endif
```

Аргумент тега у виклику ExAllocatePoolWithTag складається з чотирьох букв (у верхньому регістрі), які на екрані відладчика з'являться в зворотному порядку, тобто "4321".

У даному прикладі для всіх операцій виділення пам'яті за допомогою ExAllocatePoolWithTag використаний один і той же тег. У деяких ситуаціях може виявитися доцільним використання різних значень тега для виділення пам'яті під структури даних різного типу, або для виділення пам'яті в різних фрагментах драйвера. Ці прийоми допоможуть ефективніше ідентифікувати витік пам'яті.

Що поставляється у складі DDK утиліта PoolTag дозволяє спостерігати теговое виділення пам'яті і без залучення відладчика WinDbg. Ця програма безперервно виводить на екран оновлювані дані про сторінкові тегах.

8.6. Приватні прийоми відновлення систем

При поетапному тестуванні драйвера (спочатку – процедур завантаження і вивантаження, потім скелета робочих процедур, потім алгоритму функціонування і тому подібне) неполадки в коді навряд чи викличуть проблеми загального функціонування системи. Проте повністю виключати вірогідність руйнування системи не можна. Тому розглянемо деякі методи приведення її до працездатного стану після збоїв.

Перш за все, після фатального збою системи слід дозволити їй виконати перевірку цілісності файлової системи. Не дивлячись на тривалість цієї процедури, слід довести її до завершення (при відповідному запиті системи), оскільки накопичення дефектів неприпустимо – це небезпечніше, ніж кожен із збоїв окремо.

В деяких випадках установка нового драйвера може приводити до неможливості завантажити систему звичайним способом. В цьому випадку можливі декілька варіантів розвитку подій.

По-перше, по натисненню клавіші F8 можна провести вибір на початку завантаження комп'ютера різних варіантів, з яких найчастіше використовуються для відновлення працездатності два, а саме:

- завантаження останньої вдалої конфігурації;
- завантаження в безпечному режимі.

У останньому випадку після закінчення завантаження можна видалити драйвер, що став джерелом неприємностей, з системи засобами диспетчера пристроїв, або просто стерти бінарний файл нового "поганого" драйвера в тому місці, де його повинна виявляти ОС – з метою не допустити його завантаження наступного разу.

У тих випадках, коли завантаження системи більш менш успішно доходить до кінця, адміністратор може виконати відновлення стану системи в одній з раніше збережених резервних точок.

При серйозніших проблемах можна скористатися завантаженням із зовнішнього носія. Завантажувальна дискета дозволить дістати доступ до логічних дисків зі встановленою системою Windows. Логічно користуватися цим способом на комп'ютерах, що не підтримують завантаження з CD ROM, оскільки дискета забезпечує доступ до завантажувального диска CD, який також необхідний в цьому процесі.

В тому випадку, якщо в розпорядженні є компакт-диск для установки Windows і комп'ютер може бути завантажений з CD-ROM (практично всі сучасні комп'ютери на платформі Intel дозволяють це зробити), то слід виконати саме такий спосіб завантаження і увійти до Консолі відновлення системи (Recovery Console). В цьому режимі будуть доступні системні консольні команди, список яких можна отримати по команді help. Найбільш актуальною є команда примусової перевірки диска, наприклад, логічного диска D: CHKDSK D: /p. В даному випадку ключ /p забезпечує примусову перевірку.

В тому випадку, коли з'явився зіпсований завантажувальний запис (MBR) або слід позбавитися від завантажувального запису нестандартних завантажувачів LILO, MILO і т. ін., слід скористатися програмою fixmbr. У цьому ж режимі можна виконати перевірку диска. Перезаписати завантажувальний (boot) сектор на логічному диску можна за допомогою команди fixboot.

При непорушеній файловій системі утруднення в завантаженні можуть бути обумовлені поганим розміщенням інформації в Системному Реєстрі. Деякі джерела пропонують, окрім спеціальних програм, такий метод відновлення як просте копіювання раніше збережених файлів Системного Реєстру на їх місце в директорії %systemroot%\config. У нормальному робочому стані ОС не вирішує питання доступу до них (для перегляду, копіювання або перезапису), проте, при завантаженні з настановного диска CD, це стає допустимою операцією. Файли

Реєстру, які завжди мають солідні розміри, для цього випадку можна зберігати на інших логічних дисках комп'ютера.

Всі утруднення, пов'язані із специфікою доступу до розділів файлової системи у форматі NTFS, можна обійти, якщо встановити ОС на логічний диск з форматом FAT32. Втративши деякі переваги NTFS, такі, як запис файлів більш 4 Гбайт і висока надійність, можна буде діставати доступ до таких розділів з дискети MS-DOS або при завантаженні Windows 98 з іншого логічного диска. Абсолютно аналогічна і офіційна рада дає Microsoft, пропонуючи встановити на комп'ютері ще один екземпляр Windows NT на іншому логічному диску, що, безумовно, надасть можливість доступу до всіх NTFS розділам, зокрема – до того, на якому випробовується драйвер і, можливо, відбувся збій.

Нарешті, ще одним засобом доступу до файлів ОС Windows NT 5, розміщених на диску з файловою системою NTFS, є програма NTFSDOS, розроблена фірмою SysInternals. При завантаженні MS DOS або при завантаженні із завантажувальної дискети, зробленої під Windows 98, ця програма (якщо вона записана в не-NTFS розділах) може бути запущена і дозволяє дістати доступ до NTFS розділам жорсткого диска, включаючи запуск програми CHKDSK. В свій завантажувальний набір програма NTFSDOS при установці включає ще файли Ntoskrnl.exe, Ntfs.sys і Ntdll.dll зі встановленої на комп'ютері версії Windows NT. Відповідно, це дозволяє модифікувати дані в NTFS розділах – за наявності повної версії NTFSDOS Professional – або копіювати їх в інші не-NTFS розділи (демо-версія), якщо вони присутні в системі. Можливий варіант, при якому готується комплект з 2-х дискет, з яких і виконується завантаження.

Приступаючи до тестування драйвера, розробник повинен в достатній мірі підготуватися до потенційно можливих неприємностей, які можуть виникнути із-за порушення цілісності ОС, зводячи своє втручання до мінімуму. Тим паче, що порушення, які дійсно потребують відновлення заново всієї ОС виникають не так вже часто.

8.7. Процес завантаження операційної системи

При завантаженні 32-розрядних версій ОС Windows NT 5 (2000, XP або Server 2003) використовуються наступні файли:

- C:\NTLDR – стадії підготовки до завантаження і завантаження;
- C:\BOOT.INI – стадія завантаження ядра;
- C:\BOOTSECT.DOS – стадія завантаження ядра (опційно);
- C:\NTDETECT.COM – стадія завантаження ядра;
- *%systemroot%*\System32\NTOSKRNL.EXE – стадія завантаження ядра (або NTKRNLPA.EXE);
- *%systemroot%*\System32\HAL.DLL – стадія завантаження ядра;
- Розділ реєстру HKLM\SYSTEM – стадія ініціалізації ядра;

– *%systemroot%*\System32*.sys – стадія ініціалізації ядра.

де *%systemroot%* – це директорія, в якій розміщені основні файли ОС, наприклад, D:\Windows – для Windows XP, встановленої на логічному диску D. Для системи Windows 2000, наприклад, встановленої на логічному диску C, це буде директорія C:\WINNT.

Підготовка до завантаження

1. Комп'ютер тестує себе (стадія POST, Power On Self Test), оперативну пам'ять, фізичні пристрої. Якщо BIOS підтримує специфікацію PNP, то відбувається визначення і настройка такого типу пристроїв.
2. BIOS виявляє завантажувальний пристрій (жорсткий диск, привід CD ROM), завантажує і запускає на виконання основний завантажувальний запис.
3. MBR проглядає таблицю розділів (partition table), щоб знайти активний, завантажує завантажувальний вектор активного розділу в пам'ять і запускає його на виконання.
4. Завантажує та ініціалізує файл NTLDR, який є завантажувач ОС.

Початкова стадія завантаження

Завантажувач NTLDR перемикає процесор з реального режиму в 32-розрядний захищений, що необхідно для надання можливості виконати будь-яку додаткову функцію. Файлова міні-система, вбудована в NTLDR дозволяє завантажувати Windows NT 5 з носіїв FAT, FAT32 і NTFS.

Стадія завантаження

Завантажувач NTLDR прочитує файл BOOT.INI, використовує його для відображення екрану завантажувача, надає користувачеві можливість вибору ОС, вибирає профіль устаткування і завантажує ядро, але не ініціалізує його. Файл BOOT.INI містить 2 розділи: [Boot Loader] і [Operating System].

Розпізнавання устаткування

NTDETECT.COM запускається після напису "Виберіть операційну систему для завантаження", складає список встановленого на даний момент устаткування, і повертає цей список в NTLDR для подальшого включення його в розділ системного реєстру HKLM\HARDWARE. NTDETECT.COM виконує визначення характеристик пристроїв:

- тип шини;
- послідовні порти;
- математичний співпроцесор;
- гнучкі диски;
- клавіатура і вказуючи пристрої;
- паралельні порти;
- адаптери SCSI;
- відеоадаптери.

Слід зазначити, що пристрої, які підключаються до шин і повинні бути виявлені шинними драйверами, наприклад, зовнішні пристрої на шинах USB,

тут не виявляються – по тій простій причині, що шинні драйвери, які повинні виконати цю роботу, на даний момент ще не завантажені.

Вибір конфігурації

Після того, як NTLDR починає завантаження і отримає інформацію про устаткування, буде відображений екран з діалогом “Вибір профілю устаткування / Відновлення системи” (Hardware profile/Configuration recovery). У випадку, якщо профіль устаткування є єдиним, то діалогу представлено не буде.

Завантаження ядра

На етапі завантаження ядра NTLDR виконує такі дії:

- завантажує код ядра з файлу NTOSKRNL.EXE (NTKRNLPA.EXE за наявності опції /PAE у файлі boot.ini), але не ініціалізує його;
- завантажує код шару апаратних абстракцій з файлу HAL.DLL;
- завантажує розділ HKLM\SYSTEM з %systemroot%\System32\Config\System;
- вибирає набір параметрів для конфігурації (список драйверів, пристроїв, пристроїв і служб, які необхідно запустити);
- завантажує драйвери (звичайно це низькорівневі драйвера, як, наприклад, драйвера дисків) із значенням параметра Start рівним 0x0.

Значення параметра List в HKLM\SYSTEM\CurrentContorlSet \Control\ServiceGroupOrder визначає порядок завантаження їх завантажувачем NTLDR. Драйвери, регулюючі своє завантаження у такий спосіб, повинні мати відповідні значення параметра Group в своїх підрозділах розділу Системного Реєстру HKLM\System\CurrentControlSet\Services.

Ініціалізація ядра

Після закінчення завантаження, ядро ініціалізувалося і йому передається управління від завантажувача NTLDR:

1. Створюється розділ HKLM\Hardware за наслідками розпізнавання апаратури, куди заноситься інформація про системну плату, пристрої і переривання.
2. Створюється набір параметрів Clone шляхом копіювання параметрів, що управляють; інформація про яких міститься в параметрі Current в розділі HKLM\System\Select. Набір Clone ніколи не модифікується.
3. Завантажуються драйвери, вказані в розділі системного реєстру HKLM\System\CurrentControlSet\Services, в параметрах яких присутнє значення Start рівне 0x01, порядок завантаження яких так само, як і було вказано вище, визначається в параметрі Group. Драйвери ініціалізувалися відразу ж після їх завантаження. Значення параметра ErrorControl в описі драйвера, тобто в його параметрі, вказаному в Системному Реєстрі, визначає реакцію системи в тому випадку, якщо при завантаженні і ініціалізації даного драйвера відбулася помилка.
4. Запускаються сервіси (наприклад, Служба журналу подій) і драйвери.

Виведення на екран інформації про процес завантаження

Вказавши параметр /sos у відповідному рядку файлу boot.ini, наприклад: multi(0) disk(0) rdisk(0) partition(2)\Windows="Комментарий для користувача" /sos можна забезпечити виведення на екран інформації про завантажені програмні модулі ОС.

У випадках, коли збій системи відбувається на одному з етапів завантаження, така діагностика може бути вельми корисна.

Примітка 1. В тому випадку, якщо в рядку опису параметрів завантаження після опції /sos ввести /PAE, то перший рядок прийме вигляд:

multi(0) disk(0) rdisk(0) partition(1)\Windows\System32\ntkrnlpa.exe

Примітка 2. Переконатися програмним способом в тому, чи підтримує операційна система зараз роботу з PAE, тобто з яким ключем вона завантажена), нескладно. Достатньо в текст драйвера включити наступний фрагмент:

```
if (*Mm64bitPhysicalAddress==TRUE)
{
 #if DBG
 DbgPrint("System supports IO operations over 4GB");
 #endif
}
else
{
 #if DBG
 DbgPrint("System doesn't support IO ioerations over 4GB");
 #endif
```

Змінна Mm64BitPhysicalAddress оголошується у файлах wdm.h і ntddk.h. Інше підтвердження роботи ОС з підтримкою PAE можна виявити в Системному Реєстрі: у такому разі параметр PhysicalAddressExtension у розділі HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management прийме значення 1, при відсутності підтримки PAE цей параметр прийме значення 0.

Висновки

В даному розділі були детально розглянуті програмні засоби тестування та налагодження драйверів, особливості даного процесу, загальні прийомт відладки та існуючі методики відладки та тестування. Також звернута увага на приватні прийоми відновлення системи та розглянутий процес завантаження операційної системи.

Контрольні запитання та завдання

1. Що таке драйвер?
2. Що таке підписаний драйвер?
3. Яким чином перевірити працездатність Windows XP/2003 DDK?
4. Для чого потрібен файл SOURCES?
5. Для чого потрібен файл MAKEFILE?
6. Для чого потрібен файл DDK Build?
7. Основне призначення відладчика WinDbg?
8. Для чого необхідний I/O ConTroL code (IOCTL)?
9. Для чого необхідна процедура DriverEntry?

Теми для самостійного опрацювання

1. Обмеження, що накладаються на драйвер.
2. Відладчики.
3. Діагностичне виведення інформації із драйверу.
4. Активізації відладки режиму ядра на тестовому комп'ютері.
5. Активізації відладки режиму ядра для Windows XP.
6. Активізації відладки режиму ядра на тестовому комп'ютері Windows NT.

Список використаної літератури

1. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО "Бином-Пресс", 2004. – 480 с.
2. *Рудаков П. И.* Язык ассемблера: уроки программирования / П. И. Рудаков, К. Г. Финогенов. – М.: ДИАЛОГ-МИФИ, 2001. – 640 с.
3. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
4. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.

РОЗДІЛ 9. ВИКОРИСТАННЯ INF-ФАЙЛІВ ДЛЯ ІНСТАЛЯЦІЇ ДРАЙВЕРІВ

План

- ✦ Призначення інтерпретаторів
- ✦ Ідентифікаційний заголовок
- ✦ Скрипти загального призначення
- ✦ Копіювання файлів
- ✦ Видалення файлів і директорій
- ✦ Перейменування об’єктів та зміна атрибутів
- ✦ Деінсталяція програми
- ✦ Секція strings (оголошення рядкових змінних)

Сьогодні всі драйвери до всіх пристроїв для MS Windows 4 і вище написані на технології INF. На діалектах INF базується установка MS Internet Explorer всіх версій, MS Media Player та таке інше. Тому в даному розділі розглядаються призначення та методи використання INF-файлів для інсталяції драйверів.

9.1. Інтерпретатори

В inf-скриптах чассто відсутні основні смислові конструкції, як то: if ... then ... else, for, while, case і т.д. Проте, в цьому не завжди є потреба, вони займають свою нішу, а смислові конструкції містять замість них бінарний код. Скрипти inf нічим не відрізняються від простих BAT-файлів за принципом – задана послідовність дій, інтерпретатор виконує, або виводить помилку. Вони вміють працювати з реєстром, файлами-папками, ini-, lnk-файлами, виводити простенькі діалоги, і послідовно запускати PE-файли на виконання. У принципі, для встановлення програм цього вистачає. Знання їх структури, роботи та синтаксису буде просто необхідно системним адміністраторам Windows, а також корисно інсталювальникам та початківцям програмістам.

У MS Windows присутні за замовчуванням два інтерпретатора скриптів INF: SETUPAPI і ADVANCEDINF. Обидва інтерпретатора представляють два DLL-файли в системній директорії і деяку кількість ключів у реєстрі. Інтерпретатор SETUPAPI знаходиться в бібліотечному файлі setupapi.dll, інтерпретатор ADVANCEDINF – у бібліотечному файлі advpack.dll. В ОС MS Windows 95, 98 інтерпретатор SETUPAPI знаходиться в 16-розрядної бібліотеці setupx.dll. Бібліотека setupapi.dll є в MS Windows 98 і включена в пакети латок для MS Windows 95, проте основним інтерпретатором залишається setupx.dll.

Тобто, слід враховувати, що setupapi.dll не завжди присутній в системі. Виходячи з того, що бібліотеки інтерпретаторів не є виконуваними файлами, потрібний зовнішній ініціатор запуску функції інтерпретації скрипта. Ним є системна утиліта RunDLL32.exe.

Формат запуску будь-якої бібліотеки за допомогою RunDLL32 має вигляд:

rundll32.exe libraryname, EntryPoint parameters

де:

- libraryname – назва файлу бібліотеки (у даному випадку бібліотеки інтерпретатора), припустимо вказувати без розширення, якщо бібліотека зареєстрована в списку SharedDLLs в системному реєстрі;
- EntryPoint – регістрочутливе ім'я точки входу в бібліотеку (ім'я викликається функцією), вказується відразу після коми, без пробілів;
- parameters – параметри, що передаються функції.

Синтаксис командного рядка для запуску інтерпретаторів скриптів INF побудований на цих же принципах. Формат запуску обумовлений високим ступенем дозволеності дій у системі файлу простого текстового формату. Microsoft не звикла довіряти скриптам.

9.1.1. SETUPAPI

Основний інтерпретатор. Виконує основну масу дій:

- запис і видалення ключів, параметрів і значень реєстру;
- додавання та видалення рядків, заміна значень ini-файлів;
- розпакування файлів з CAB-архівів, копіювання та видалення файлів;
- перейменування, зміна атрибутів файлів і папок;
- установка драйверів;
- створення системних сервісів та пристроїв (Windows NT-based);
- перевірка користувацьких повноважень (перевірка на адміністратора);

Типовий приклад запуску інтерпретатора SETUPAPI для виконання скрипта:

rundll32.exe setupapi, InstallHinfSection DefaultInstall 132 C: \ Script.inf

де:

- InstallHinfSection – ім'я функції, що викликається (точка входу);
- DefaultInstall – перший параметр для функції, що викликається, означає ім'я виконуваної с ції в INF-скрипті;
- 132 – другий параметр для функції, що викликається, прапорець для обробки скрипта;
- C: \ Script.inf – третій параметр для функції, що викликається, повний шлях до файлу скрипта. Зверніть увагу, потрібно саме повний шлях, тому що просте зазначення імені файлу передбачає розташування файлу скрипта в

системній директорії Windows. Ця ж примітка в рівній мірі відноситься і до інтерпретатора AdvancedINF.

У випадку з MS Windows 95 рядок запуску буде таким:

```
rundll.exe setupx.dll, InstallHinfSection DefaultInstall 132
C:\Short_~1\Script.inf
```

де:

- rundll.exe – 16-розрядний бінарник для запуску 16-розрядної бібліотеки setupx.dll;
- C:\ Short_~1\Script.inf – коротке, DOS повне ім'я до файлу скрипта, де кожне ім'я об'єкта не повинно перевищувати 8-ми символів. Точка і три символи розширення не підпадають під це правило. Якщо ім'я папки або файлу довше 8-ми символів, беруться перші 6, а інші замінюються двома символами: ~1. Інший об'єкт довший 8-ми символів з однаковими першими 6-ма символами в DOS-інтерпретації буде закінчуватися на ~2 і так далі.

9.1.2. ADVANCEDINF

AdvancedINF – це надбудовний над SETUPAPI інтерпретатор, що дозволяє виконувати додаткові функції. Стандартні функції передає на виконання інтерпретатору SETUPAPI. До основних функцій, які підтримуються інтерпретатором AdvancedINF відносять:

- попередній запис змінюваних ключів реєстру в бінарний файл (функція відкату);
- одноразове виконання дій під кожним користувачем (доустановка) при інсталяції та деінсталяції під час входу в систему (Active Setup);
- запуск виконуваних файлів з параметрами в прихованому і нормальному режимах;
- висновок простих діалогових вікон;
- читання директорій призначення операцій з файлами з реєстру.

Виходячи з вищезазначеного, буде розумним віддавати перевагу інтерпретатору AdvancedINF. Виняток становить установка драйверів – тут з цим може справлятися тільки SETUPAPI.

Типовий приклад запуску інтерпретатора AdvancedINF для виконання скрипта:

```
rundll32.exe advpack, LaunchINFSection C: \ Script.inf,
DefaultInstall, 4
```

де:

- LaunchINFSection – точка входу;
- C: \ Script.inf – перший параметр для функції, що викликається, повний шлях до файлу скрипта;

– DefaultInstall – другий параметр, ім'я виконуваної секції в INF-скрипті, де ім'я секції непомітно для регістру на відміну від точки входу;
– 4 – прапорець реакції інтерпретатора при обробці команд скрипта.

9.2. Ідентифікаційний заголовок

Для впізнання текстового файлу як файлу з inf-структурою інтерпретатор шукає в непустих рядках секційний заголовок [Version]. Заголовок традиційно розташовується на початку файлу і містить у своїй секції кілька рядків, що визначають тип скрипта. Заголовки бувають наступних типів: стандартні, драйверні і розширені. Тип скрипта вказує, яким саме інтерпретатором слід виконувати скрипт. В залежності від інтерпретатора можна виконувати різний набір дій. Тому розглянемо дані заголовки більш детально.

9.2.1. Стандартний заголовок

Скрипт зі стандартним заголовком призначений для виконання операцій загального призначення. Набір функцій, які підтримуються скриптом зі стандартним заголовком, визначається інтерпретатором, якому було передано на виконання цей скрипт. Ось секція стандартного заголовка скрипта:

```
[Version]
Signature = "$ CHICAGO $"
SetupClass = BASE
ClassGUID = (00000000-0000-0000-0000-000000000000)
```

Вказані тут параметри SetupClass і ClassGUID рідко використовуються разом, звичайно досить будь-якого з них. З драйверними заголовками ситуація дещо інша. Обмеження для використання стандартного заголовка: вкрай не рекомендується використовувати при написанні скрипта установки драйверів. При побудові листа драйверів, що відповідають типу встановленого пристрою, інтерпретатор відбирає скрипти драйверів саме за заголовком і драйверні скрипти зі стандартним заголовком ніколи не будуть включені в список.

Інші обмеження на даний момент не відомі.

9.2.2. Драйверний заголовок

Драйверний заголовок вміє обробляти тільки інтерпретатор SETUPAPI. Приклад драйверного заголовка:

```
[Version]
Signature = "$ CHICAGO $"
Class = System
ClassGuid = (4d36e97d-e325-11ce-bfc1-08002be10318)
```

```
CatalogFile = sample.cat
Provider =% MSFT%
LayoutFile = layout.inf
DriverVer = 03/16/2005, 6.00.9830.1
```

Визначенням драйверного заголовка є параметри Class і ClassGuid. Вони вказують на один і той же тип драйвера, Class вказує ім'я типу, ClassGuid – його GIUD. Список відомих поточній ОС типів драйверів можна знайти в системному реєстрі по шляху: HKEY_LOCAL_MACHINE \ SYSTEM \ CurrentControlSet \ Control \ Class.

Класи драйверних типів:

```
(4D36E966-E325-11CE-BFC1-08002BE10318) Computer
(4D36E968-E325-11CE-BFC1-08002BE10318) Display
(4D36E96B-E325-11CE-BFC1-08002BE10318) Keyboard
(4D36E96A-E325-11CE-BFC1-08002BE10318) HDC
(745A17A0-74D3-11D0-B6FE-00A0C90F57DA) HID
(6BDD1FC6-810F-11D0-BEC7-08002BE2092F) Image
(4D36E96C-E325-11CE-BFC1-08002BE10318) Media
(4D36E96D-E325-11CE-BFC1-08002BE10318) Modem
(4D36E96E-E325-11CE-BFC1-08002BE10318) Monitor
(4D36E96F-E325-11CE-BFC1-08002BE10318) Mouse
(50906CB8-BA12-11D1-BF5D-0000F805F530) MultiPortSerial
(4D36E972-E325-11CE-BFC1-08002BE10318) Net
(4D36E979-E325-11CE-BFC1-08002BE10318) Printer
(4D36E97B-E325-11CE-BFC1-08002BE10318) SCSI Adapter
(50DD5230-BA8A-11D1-BF5D-0000F805F530) Smart Cart Reader
```

Не рекомендується вказувати тільки Class, тому що не всі системи цим задовольняться. При визначенні скрипта як драйверного, в системі, починаючи з версії MS Windows 2000 – NT 5.0, ініціюється "вакцина" – реакція на зміну системної частини реєстру. При цьому, в залежності від поточних налаштувань, може бути виведений запит на підтвердження установки драйвера. Починаючи з MS Windows XP (NT 5.1) при налаштуваннях за умовчанням створюється точка відкату системної частини реєстру і змінених в ході установки системних файлів. Крім того, за умов згадування драйверного заголовка скрипт буде позбавлений можливостей, які підтримуються в AdvancedINF.

9.2.3. Розширений заголовок

Розширений заголовок скрипта вказує, яка версія інтерпретатора AdvancedINF необхідна для виконання скрипта. Нові версії бібліотеки advpack.dll (AdvancedINF) звичайно поставляються з дистрибутивами Microsoft Internet Explorer, тому що технологія встановлення останніх базується саме на AdvancedINF. Старі версії інтерпретатора (нижче 2.0) можна виявити на Windows NT 4.0 з сервіс пакетом 3 і нижче і на Windows 95 OSR 2 (без нового MSIE) і нижче. Якщо поточна версія інтерпретатора менше вказаного у скрипті,

обробка скрипта буде перервана і виведеться помилка, у нашому випадку – “Error message”:

```
[Version]
Signature = "$ CHICAGO $"
AdvancedINF = 2.0, "Error message"
```

Якщо файл з розширеним заголовком, який використовується для обробки інтерпретатором AdvancedINF, буде в командному рядку переданий на виконання для SETUPAPI, виконаний він буде як файл зі стандартним заголовком.

9.2.4. Сигнатура системи

Як видно, в кожному типі заголовка незмінним рядком є параметр Signature. Його значення визначає, для якої ОС призначений скрипт. Відомі два значення: $CHICAGO$ і $WINDOWS NT$. Перший призначений для всіх ОС MS Windows; другий тільки для MS Windows NT 5.0 (2000), 5.1 (XP), 5.2 (2003). Виняток становлять драйверні скрипти – тут необхідно якомога точніше визначити тип системи, інакше драйвер (в залежності від типу) не буде розпізнаний, як відповідний.

9.3. Скрипти загального призначення

У кожному inf-файлі присутня виконувана секція, яка визначається вже в рядку виклику inf-файлу. Саме ця секція розглядається як керівництво до виконання команд, незалежно від наявності інших виконуваних с цій в цьому скрипті. Подібний підхід (на відміну від bat-файлів) дозволяє зберігати безліч пакетів процедур в одному і тому ж файлі скрипта. Як правило, в одному скрипті установки зберігаються пакети процедур установки і деінсталяції програми. Виконувана секція містить ключі, прапори для перевірки можливості виконання скрипта і, найголовніше, параметри типів дій, чиї значення містять імена дочірніх с цій, у яких прописані об’єкти для виконання дій над ними.

Наявність параметрів у виконуючій секції багато в чому залежить від типу скрипта (стандартний, драйверний, розширений). В одній головній секції не може бути двох і більше однойменних параметрів. Імена параметрів, за винятком спеціальних прапорів, визначають тип дій, наприклад, запис до реєстру, копіювання файлів. Значення цих параметрів складають імена інших, дочірніх секцій, що містять адреси та імена об’єктів для виконання операцій. Імена дочірніх секцій повинні даватися через кому і знак пробілу. Якщо імен дочірніх секцій декілька, вони виконуються по черзі, в порядку написання. Деякі параметри повинні містити тільки одну секцію. Приклад виконуваної секції розширеного скрипта:

```
[DefaultInstall]
CopyFiles = cpf.test, cpf.testAA
AddReg = adr.tester, adr.tester2
DelReg = dlr.testing
UpdateInis = ini.test_section
```

Правило: імена дочірніх секцій (у цьому прикладі вони: cpf.test, cpf.tester, cpf.testing ...) не повинні починатися з цифри (0-9) і не повинні бути рівні зарезервованим іменам параметрів, як наприклад: CopyFiles, DelFiles, AddReg, DelReg, UpdateInis, Reboot, CheckAdminRights, RequiredEngine, CustomDestination, BeginPrompt, EndPrompt, ComponentName, ComponentVersion, PreRollBack, PerUserInstall, RunPreSetupCommands, RunPostSetupCommands. Крім того, імена секцій не повинні містити пропусків і знаків: “” , ‘ / \ ? * ; : ^ () [].

Основні описи імен параметрів виконуваної секції подані в табл. 9.2.

Таблиця 9.2

Опис імен параметрів виконуваної функції

Параметр	Можливі значення	Визначення
Інтерпретатор SETUPAPI:		
CopyFiles	CopyFiles = Імена дочірніх секцій	Копіювання файлів, розташованих відносно файлу скрипта в поточній директорії, в підтеках або в cabinet-архівах в назначені директорії копіювання
DelFiles	DelFiles = Імена дочірніх секцій	Видалення файлів у вказаних директоріях
RenFiles	RenFiles = Імена дочірніх секцій	Перейменування файлів у вказаних директоріях
AddReg	AddReg = Імена дочірніх секцій	Додавання або заміна ключів, параметрів і значень реєстру на зазначені в дочірніх секціях
DelReg	DelReg = Імена дочірніх секцій	Видалення з системного реєстру ключів або параметрів (не значень), зазначених у дочірніх секціях
UpdateInis	UpdateInis = Імена дочірніх секцій	Видалення або запис параметрів і значень в ini-файлах, а також створення користувальницьких ярликів

9.4. Копіювання файлів

Можна сказати, основне завдання інсталяційного inf-скрипта – це копіювання файлів і зміна ключів реєстру. Ось саме ці секції продумані в реалізації inf досконало. Копіювання файлів включає в себе можливе розпакування з стиснених архівів формату cabinet (розширення файлів .cab), просте копіювання, перейменування в процесі копіювання, створення директорії призначення, прив'язку реального шляху по системним змінним, видалення файлів, підтримку багатодисковим і багатоархівних дистрибутивів, перевірку версії, локалізації та наявності файлу, який замінюють.

9.4.1. Секції копіювання

Для успішного копіювання файлу необхідно знати три речі: куди, що і звідки. Куди – це вміст секції [DestinationDirs] і, можливо, ще її помічника параметра CustomDestination. Що – це вміст дочірніх секцій, що містять імена файлів. Звідки – це вміст секцій [SourceDisksFiles] і [SourceDisksNames]. Об'єкти "куди" і "що" пов'язані між собою однаковими іменами дочірніх с цій. Тобто, виходячи з того, що вся дочірня секція може бути скопійована тільки в один каталог, логічним є те, що у дочірньої секції всього один пункт призначення і, отже, їй належить один параметр в секції [DestinationDirs]. Далі, об'єкти "що" і "звідки" пов'язані між собою іменами файлів, значить, файл, присутній в операції переміщення (дочірньої секції), зобов'язаний бути зазначеним у джерелі [SourceDisksFiles]. Для полегшення розуміння цих зв'язків розглянемо приклад:

```
[DefaultInstall]
CopyFiles = cpf.test
[cpf.test]
filename.ext
[DestinationDirs]
cpf.test =- 1, C: \ Temp
[SourceDisksFiles]
filename.ext = 1
[SourceDisksNames]
1 = "TEST CD ","", 0
```

У цьому простому прикладі копіюється файл filename.ext в каталог C: \ Temp. Якщо необхідно скопіювати кілька файлів в директорію C: \ Temp, їх потрібно дописати в двох секціях: дочірній секції [cpf.test] і в секції "Звідки" – [SourceDisksFiles]. Якщо наступний файл необхідно буде копіювати в іншу директорію, для нього доведеться створювати нову дочірню секцію, додавати її до першого в параметрі CopyFiles і вказувати для неї новий каталог призначення [DestinationDirs]. У місці "Звідки" (дистрибутив) файли можуть перебувати в різних підкаталогах і/або в cab-архівах. За це відповідає секція

[SourceDisksNames]. Вона містить пронумеровані дистрибутивні шляхи, причому кожен шлях може вказувати одночасно і на окремий диск, і на окрему директорію і на окремий cab-архів. Формат запису дистрибутивних секцій має вигляд:

```
[SourceDisksNames]
1 = "Drive_Name", "data1.cab", subdir
2 = "Drive_Name 2 ","", 0
```

де:

- 1 – змінна дистрибутивного шляху;
- “Drive_Name” – ім’я диска (label);
- “data1.cab” – cabinet-архів на диску “Drive_Name”, що містить файли, призначені для копіювання (допускається вказувати ім’я файлу cab-архіву без розширення);
- subdir – Піддиректорія на диску “Drive_Name”, що містить файл “data1.cab”. У другому рядку вказано лише ім’я диску, ім’я файлу cab-архіву залишено порожнім, а піддиректорія дорівнює нулю. Більш традиційним вважається ставити для поточної директорії знак точки – “.”, Однак, швидше за все, у Microsoft проблеми з синтаксисом і, крім того, постійні порушення неписаних стандартів.

Якщо припустити, що всі три зазначені файли вказані в одній дочірній секції копіювання, однак, як зазначено вище, знаходяться на різних змінних дисках – перші два на диску з міткою “Drive_Name, а третій – на диску “Drive_Name2”, то в процесі копіювання буде показаний діалог з вимогою змінити диск для продовження операції копіювання або вказати інший шлях до диска, як це показано на рис. 9.1.

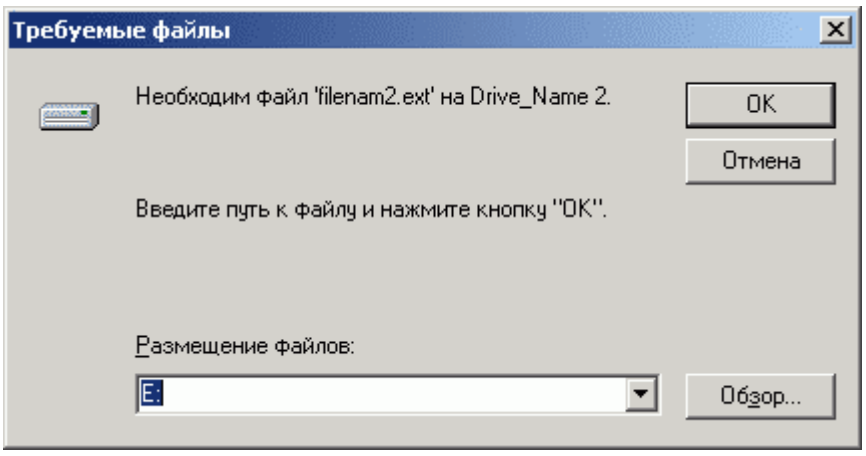

Рис. 9.1.

9.4.2. Директорії призначення

Змінні шляхів – це результат використання дискової системи в MS Windows, де реальний шлях до додатка може не збігатися з передбачуваним

через різницю імен розділів. Напприклад, не завжди системна директорія розташована як C: \ Windows. Для того, щоб файли потрапили в потрібну директорію призначення, в системі є декілька системних змінних, що визначають конкретні шляхи в системі, наприклад, у будь-якій MS Windows 16-бітна змінна %WinDir% (формат 8.3) містить повний шлях до каталогу з Windows. Крім нього, в NT-системах є аналогічна змінна з ім'ям% SystemRoot%, 32-х розрядна. Ці змінні чомусь не можна використовувати в inf-скриптах, однак замість цього inf підтримує цілу купу власних змінних для системних директорій, на 90% непотрібних, тому що на всіх існуючих системах MS Windows ці шляхи ведуть в одні і ті ж піддиректорії головного каталогу з MS Windows. Виняток становить лише системний підкаталог, в MS Windows 95, 98, ME він виглядає як %WinDir%\System, а в Windows NT 4.0, Windows 2000 і вище – %SystemRoot%\System32. Деякі змінні не підтримуються в усіх системах (і в 9x і в NT). Це відноситься до DOS-утиліт і директорії завантажувача. Змінні в тілі inf-скрипта позначаються так само, як і в bat-файлах – обрамляються знаками відсотка з боків, наприклад, %11%. Їх неможливо змінити в секції [strings] для використання як директорій призначення і файли все одно будуть скопійовані в директорію, яка відповідає змінній. Однак можна використовувати цю змінну для значень, які підставляються в інші види с цій (не секція [DestinationDirs]). Хоча при цьому весь сенс підміни змінної втрачається. Основні значення параметрі в директорії призначення описані в табл.. 9.3.

Таблиця 9.3

Опис значень параметрів директорії призначення

ID	Значення	Опис
DIRID_ABSOLUTE	-1	Реальний шлях, вказаний після коми (cpf.test = -1, C: \ Temp)
DIRID_ABSOLUTE_16BIT	0xffff	16-бітний реальний шлях, формату 8.3 (для 16-розрядної setupx.dll в MS Windows 95)
DIRID_NULL	0	Заглушка, порожній шлях. Застосовується для налагодження
DIRID_SRCPATH	1	Директорія, де знаходиться сам inf-скрипт. Корисна для запуску інших с цій в цьому ж скрипті допомогою RunP * SetupCommands
DIRID_WINDOWS	10	Власне, сама MS Windows. Типово, але не закон, що для Windows NT 4.0, Windows 2000 – це C: \ Winnt, для інших – C: \ Windows. У командному рядку ця директорія також міститься в змінних % Windir% (усі MS Windows) і % SystemRoot% (NT 4 і вище)
DIRID_SYSTEM	11	Системна піддиректоріях MS Windows. Для NT 4, 5 і вище – це % SystemRoot% \ system32, для Windows 95, 98, ME – це % Windir% \ SYSTEM.
DIRID_DRIVERS	12	Директорія розміщення системних драйверів. Для MS Windows NT 4 і вище – це % SystemRoot% \ system32 \ drivers, для MS Windows 95 – це % Windir% \ SYSTEM \

ID	Значення	Опис
		IOSUBSYS, для MS Windows 98, ME – % Windrir% \ SYSTEM \ DRIVERS
DIRID_COMMANDS	13	Директорія з консольними DOS-утилітами, тільки для MS Windows 95, 98, ME. Розміщення: % Windir% \ Command. Для MS Windows NT 4 і вище ця змінна невідома і файли, спрямовані туди, потраплять у сміттєзбірник % SystemRoot% \ System32 \ unknown
DIRID_INF	17	Типово – % Windir% \ INF. Зазвичай цей каталог має атрибут "прихований". У ньому зберігаються майже всі inf-скрипти самої системи та їх бінарні кешовані копії, потрібні для прискорення побудови списку драйверів
DIRID_HELP	18	Директорія % Windir% \ Help. У ній знаходяться майже всі файли довідок (розширення. Chm,. Hlp,. Cnt,. Gid).
DIRID_FONTS	20	Системні шрифти. Розміщення %Windir% \ Fonts. Просте копіювання файлів шрифтів у цей каталог не зробить доступним цей шрифт для ваших додатків
DIRID_VIEWERS	21	Директорія модулів програми QuickView. Розташування: MS Windows 95, 98, ME – % WinDir% \ SYSTEM \ VIEWERS; MS Windows NT 4.0 і вище – % SystemRoot% \ System32 \ viewers
DIRID_COLOR	23	Директорія, яка містить профілі кольорових налаштувань моніторів. Типово – % Windir% \ system (32) \ COLOR
DIRID_APPS	24	Невідомо, але для Windows NT 4.0 і вище, встановленої на диск C: \ - це директорія C: \. Швидше за все, це аналог системної змінної % SystemDrive%
DIRID_SHARED	25	Та ж% SystemRoot% або% WinDir%
DIRID_BOOT	30	Коренева директорія завантажувального диска. Зазвичай C: \
DIRID_COMPBOOT	31	Коренева директорія головного диска стисненого завантажувального диска. Тільки для MS Windows 95, 98
DIRID_SYSTEM16	50	Системна директорія для 16-бітових програм , тільки для MS Windows NT 4.0 і вище,% SystemRoot% \ system
DIRID_SPOOL	51	Директорія принтерного спула (кешу), тільки для MS Windows NT 4.0 і вище – % SystemRoot% \ system32 \ spool
DIRID_SPOOLDRIVERS	52	Директорія драйверів принтерів, тільки для MS Windows NT 4.x і вище – % SystemRoot% \ system32 \ spool \ drivers \ w32x86
DIRID_USERPROFILE	53	Директорія з профілем поточного користувача. У MS Windows NT 4.0 – це % SystemRoot% \ Profiles \% USERNAME%, в MS Windows 2000 і

ID	Значення	Опис
		вище – це % SystemDrive% \ Documents and Settings \% USERNAME%. У MS Windows 95, 98, ME – це, швидше за все, сама% WinDir%
DIRID_LOADER	54	Тільки для систем NT. Шлях до завантажувального каталогу, що містить файли ntldr або osloader.exe. У 99% випадків це C: \
DIRID_PRINTPROCESSOR	55	

Якщо в секції "куди" [DestinationDirs] не визначено ні однієї директорії, всі файли будуть копіюватися в каталог за замовчуванням, яким є DIRID_SYSTEM, описаний вище в таблиці 9.3. Каталог за замовчуванням можна змінити (тільки у сесії виконання поточного скрипта), вказавши в секції [DestinationDirs] параметр DefaultDestDir. Його значення буде директорією за замовчуванням. Це корисно для деякої економії в розмірі файлу скрипта, наприклад, при встановленні безлічі пакетів в один програмний каталог. Наприклад:

```
[DestinationDirs]
DefaultDestDir = 24, Program Files \ Program Name
```

Змінні шляхів не можна використовувати в секції [strings], в діалогах BeginPrompt і EndPrompt, в іменах будь-яких секцій і як позначення типів параметрів реєстру. У всіх місцях, що залишилися, крім секції [DestinationDirs], змінні директорій необхідно обрамляти знаками відсотка. У секції [DestinationDirs] цього робити не можна. Якщо потрібно копіювати файли не в існуючу директорію, а в її підтеку, необхідно вказати шлях, що залишився після знаку коми і пробілу відносно імені змінної.

9.4.3. Прапори копіювання

Під час копіювання файлів за допомогою inf-скрипта можна перейменовувати файли, тобто вказувати не тільки каталог призначення, а й нове ім'я файлу в цьому каталозі призначення, а також визначати поведінку інтерпретатора прапорами копіювання під час виконання процедури копіювання при зустрічі помилок. В дочірній секції копіювання в кожному рядку через кому потрібно вказувати всі умови, що відносяться до копіювання конкретного файлу. Приклад опису цих можливостей наводиться нижче:

```
[DefaultInstall]
CopyFiles = cpf.files
[cpf.files]
filenam2.exe, filename.ext, 2
```

де:

– filenam2.exe – ім'я файлу, яке необхідно отримати в місці призначенні;

– filename.ext – старе ім’я файлу, яким він володіє в директорії (архіві) дистрибутиву;
– 2 – прапор обробки події, що настане, якщо в місці призначення вже є файл з таким же ім’ям або ж у разі збою при копіюванні.

Всі відомі значення прапорів подано в табл. 9.4.

Таблиця 9.4

Прапори копіювання

ID	Значення	Опис
COPYFLG_WARN_IF_SKIP	0x00000001	Виводити попередження, якщо користувач намагається пропустити файл після невдалої операції копіювання
COPYFLG_NOSKIP	0x00000002	Сховати кнопку пропуску файлу в разі невдалого копіювання. Корисно при заміні важливих системних файлів
COPYFLG_NOVERSIONCHECK	0x00000004	Не звертати увагу на версію файлу, який вже присутній у місці призначення і завжди перезаписувати файлом з дистрибутива
COPYFLG_FORCE_FILE_IN_USE	0x00000008	Замінити файл в місці призначення, навіть якщо він використовується. Теоретично при цьому інтерпретатор спробує перейменувати на випадкове ім’я, а потім скопіює файл з дистрибутива
COPYFLG_NO_OVERWRITE	0x00000010	Не копіювати файл, якщо він вже існує. Наприклад, особисті налаштування в ini-файлі некультурно замінювати налаштуваннями за замовчуванням при кожному оновленні програми
COPYFLG_NO_VERSION_DIALOG	0x00000020	Не копіювати і не виводити питань, якщо файл у місці призначення вже існує і його версія новіший за версію файлу в дистрибутиві
COPYFLG_OVERWRITE_OLDER_ONLY	0x00000040	Не копіювати, якщо версії збігаються. Тобто замінювати тільки старі версії файлів
COPYFLG_REPLACEONLY	0x00000400	Здійснювати операцію копіювання тільки якщо файл вже існує. Корисно при випуску оновлень до численних пакетів

9.5. Видалення файлів і директорій

Видалення об’єктів можна виконати кількома способами. Перерахуємо їх:

1. Видалення файлів за допомогою вказівки дочірніх с цій копіювання у параметрі DelFiles. Тобто у файлі скрипта вже є секція [DestinationDirs], яка вказує місце розташування файлів і дочірня секція копіювання, що містить список файлів, скопійованих в каталог призначення. Залишається лише вказати цю ж секцію в параметрі DelFiles. При цьому секції розташування файлів в дистрибутиві ([SourceDisksFiles], [SourceDisksFiles]) не використовуються. Цей метод найчастіше застосовується в секції деінсталяції пакета. Директорії цим способом видалити не можна, тільки файли.

Наприклад:

```
[DefaultInstall]
CopyFiles = files
[files]
filename.ext
[DestinationDirs]
files = 11
[Uninstall]
DelFiles = files
```

2. Видалення директорій параметром DelDirs із секції. Краще ставити нижче всіх параметрів DelFiles, якщо застосовується і такий параметр – внаслідок спочатку видаляться файли, а потім і директорії. В якості значення до параметру DelDirs вказується дочірня секція, яка містить просто повні шляхи до каталогу, що видаляється. Можна застосовувати змінні шляхів. Також можна вказувати параметр CleanUp, теоретично призначений для попереднього очищення непустих директорій від файлів перед видаленням. Це не врятує від файлів, що використовуються системою або будь-яким додатком – не можливо буде видалити такий каталог. Наприклад:

```
[DefaultInstall]
DelDirs = dirs
Cleanup = 1
[dirs]
"% 24% \ Program Files \ Program Name"
```

3. Видалення пустих директорій за допомогою виклику функції з параметрами з бібліотеки інтерпретатора AdvancedINF. Цей метод – Windows-команда, яку можна викликати з будь-яких скриптів. У inf-скрипті її можна викликати з параметрів RunPreSetupCommands і RunPostSetupCommands виконуваної секції. Цей метод видаляє непусті директорії. Формат запису такий:

```
[DefaultInstall]
```

```
    RunPostSetupCommands = cmd.deldirs
    [cmd.deldirs]
    rundll32.exe advpack, DelNodeRunDLL32 "% 24% \ Program Files \
Program Name"
```

9.6. Перейменування об’єктів та зміна атрибутів

Параметр RenFiles містить імена секцій, які в свою чергу містять імена файлів, призначених для перейменування. Папка, де буде відбуватися перейменування повинна бути визначена в секції [DestinationDirs]. Вимоги до імен секцій копіювання файлів: назва секції не повинна містити зарезервованих слів, таких як RenFiles, CopyFiles і т.д.

Нижче наведено приклад дочірньої секції перейменування файлів:

```
[Section.Ren]
NewFileName, OldFileName
```

де:

- NewFileName – ім’я файлу з розширенням. Вказує нове ім’я файлу, яке буде присвоєно файлу OldFileName в папці, заданій в секції [DestinationDirs];
- OldFileName – ім’я файлу з розширенням. Вказує старе ім’я файлу, яке має бути позначене в секції [SourceDisksFiles];
- секція [DestinationDirs] в INF-скрипті управляє призначенням всіх операцій перейменування файлів.

Якщо секція, зазначена значенням до параметра RenFiles, описана в секції [DestinationDirs], то всі операції з перейменування файлів будуть відбуватися в директорії, яка зазначена в [DestinationDirs]. В іншому випадку, для вказівки директорії призначення всіх операцій перейменування файлів, inf-скрипт буде використовувати параметр DefaultDestDir.

9.7. Деінсталяція програми

При створенні коректного інсталяційного пакету доводиться замислюватися про його деінсталяцію, видалення з системи. Зазвичай в системі існує стандартна утиліта видалення встановлених програм, що знаходиться в панелі керування. Для того, щоб зареєструвати в її списку свій додаток, необхідно додати до реєстру новий ключ з необхідними параметрами, а також створити функцію видалення пакету із системи. Логічно буде створити в inf-скрипті нову виконувану секцію, що виконує цю дію і скопіювати при інсталяції сам скрипт в систему. Місце розміщення ключа деінсталяції програми Program_Name в реєстрі: HKEY_LOCAL_MACHINE \ Software \ Microsoft \ Windows \ CurrentVersion \ Uninstall \ Program_Name. Представимо

опис параметрів типового ключа видалення програми у MS Windows 95, 98, NT і вище (табл. 9.5) та у MS Windows 2000 (табл.. 9.6).

Таблиця 9.5

Опис параметрів типового ключа видалення програми (підтримується MS Windows 95, 98, NT і вище)

Назва	Тип рядка	Inf-дані для запису в реєстр	Опис
DisplayName	REG_SZ	HKLM, “%Key%”, “DisplayName”, “Program_Name”	Ім’я програми, що відображається в утиліті деінсталяції
UninstallString	REG_SZ	HKLM, “%Key%”, “UninstallString”, “rundll32.exe advpack, LaunchINFSectionEx C: \ Windows \ Inf \ program.inf, Uninstall, 64, A”	Рядок, який виконується при деінсталяції пакету
DsiplayIcon	REG_SZ	HKLM, “%Key%”, “DsiplayIcon”, “C: \ program.exe, 0”	Відображення ICO-піктограми, вказується шлях до файлу з піктограмою і її номер у списку, починаючи з нуля

Таблиця 9.6

Опис параметрів типового ключа видалення програми (підтримується MS Windows 2000 і вище)

Назва	Тип рядка	Inf-дані для запису в реєстр	Опис
DisplayVersion	REG_SZ	HKLM, “% Key%”, “DisplayVersion”, “10.241.8.44”	Версія програми
HelpLink	REG_SZ	HKLM, “% Key%”, “HelpLink”, “http://company.com/support.php”	URL технічної підтримки програми
ModifyPath	REG_SZ	HKLM, “% Key%”, “ModifyPath”, “rundll32.exe params”	Команда, що виконується при натисканні кнопки “Замінити”
EstimatedSize	REG_DWORD	HKLM, “% Key%”, “EstimatedSize”, 0x10001, “1024”	Розмір встановленої програми в Кбайта
NoRemove	REG_DWORD	HKLM, “% Key%”, “NoRemove”, 0x10001, “0”	Прапори блокування видалення, оновлення або відновлення встановленої програми. Корисно для різного роду хотфіксів і апдейтів
NoModify	REG_DWORD	HKLM, “% Key%”, “NoModify”, 0x10001, “0”	
NoRepair	REG_DWORD	HKLM, “% Key%”, “NoRepair”, 0x10001, “0”	

Крім того, існують додаткові малозначимі ключі, в основному для формування вікна довідки про програму, побачити приклади яких можна, дослідивши існуючі ключі деінсталяції програм в реєстрі. Крім того, для коректної деінсталяції Microsoft реалізувала технологію точок відкату реєстру. Залишається додати у файл скрипта нову виконувану с цію Uninstall, додати до

інсталяційної секції завдання на запис ключа деінсталяції і скопіювати свій inf-скрипт (setup.inf) в систему під час інсталяції програми:

```
    [Version]
    Signature = "$ CHICAGO $"
    AdvancedINF = 2.0, "Error message"
    [DefaultInstall.NT]
    CopyFiles = cpf.inf
    AddReg = adr.uninstall
    [cpf.inf]
    program.inf, setup.inf
    [DestinationDirs]
    cpf.inf = 17
    [SourceDisksFiles]
    setup.inf = 1
    [SourceDisksNames]
    1 = "Drive of program_name ","", 0,
    [adr.uninstall]
    HKLM, "%Key%", "DisplayName",, "Program_Name"
    HKLM, "%Key%", "UninstallString",, "rundll32.exe advpack,
LaunchINFSectionEx %17% \ program.inf, Uninstall,, 64, A"
    HKLM, "%Key%", "DsiplayIcon",, "C: \ program.exe, 0"
    HKLM, "%Key%", "DisplayVersion",, "10.241.8.44"
    HKLM, "%Key%", "EstimatedSize", 0x10001, "1024"
    [Uninstall]
    DelFiles = cpf.inf
    DelReg = dlr.uninstall
    [dlr.uninstall]
    HKLM, "%Key%"
    [strings]
    Key = "Software \ Microsoft \ Windows \ CurrentVersion \ Uninstall \
Program_Name"
```

Після виконання подібного скрипту в утиліті деінсталяції з’явиться ось пункт, що представлений на рис. 9.2.

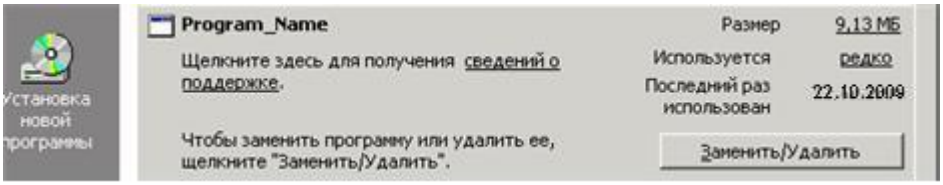

Рис. 9.2.

9.8. Оголошення рядкових змінних – секція strings

Для зменшення об’єму файлу inf-скрипта і для полегшення мовної локалізації пакетів рекомендується вживати змінні, визначення яких знаходиться в окремій секції. Назва секції [strings]. Вона зустрічається майже в кожному INF-файлі і її використання вважається правилом хорошого тону в написанні INF-скриптів. У тілі скрипта змінні позначаються знаками відсотка (%) по краях, як і змінні директорій. Приклад використання секції змінних:

```
[section]
parameter =%message1%
[strings]
regpath = "Software \ Firm_name \ Program_name \ Options"
message1 = "Довільне повідомлення"
```

За традицією секція strings розташовується нижче всіх інших секцій, вона являється завершальною.

Місця, де забороняється вживання змінних: в іменах будь-яких функцій, замість назв параметрів і замість керуючих знаків ;=[],. Під розділ змінних, які можна використовувати в inf-скриптах не потрапляють ніякі інші змінні, крім тих, які визначені безпосередньо в цьому файлі скрипта і змінних директорій, частину яких можна визначити динамічно. У значеннях змінних не можна використовувати ніякі інші змінні, включаючи змінні шляхів.

Висновки

В розділі були розглянуті призначення та методи використання INF-файлів для інсталяції драйверів. Розглянуті призначення інтерпретаров, види ідентифікаційного заголовку, скрипти загального призначення, процес копіювання файлів, видалення файлів і директорій тощо.

Контрольні запитання та завдання

1. Структура типового INF-файлу.
2. Секції INF-файлу. Загальні правила введення записів в секції.
3. Перевірка синтаксису INF-файлу.
4. Використання INF-файлів.

Тести для самоконтролю

1. Що використовується при установці пакетів і драйверів в Microsoft?
    a. бінарники
    b. скрипти
    c. bat-файли
    d. нічого, все встановлюється вручну
2. Яка конструкція відсутня в діалекті inf-файлу?

a. open
b. action
c. case
d. label

3. Яка конструкція присутня в діалекті inf-файлу?
   a. description
   b. case
   c. for
   d. if…then…else
4. Що таке інтерпретатор?
   a. програма або технічний засіб, який виконує трансляцію програми
   b. програма, що перетворює програмний код, написаний певною мовою програмування, на семантично еквівалентний код в іншій мові програмування
   c. програма, що розпізнає синтаксис скрипта і послідовно виконує команди, вказані в скрипті
   d. програма, яка виконує компонування – приймає на вхід один або кілька об’єктних модулів і збирає у виконуваний модуль
5. Що таке скрипт?
   a. текст комп’ютерної програми на якій-небудь мові програмування; в узагальненому сенсі – будь-які вхідні дані для транслятора
   b. текстовий файл, що містить секції, параметри секцій і значення параметрів секцій, що описують дії, які необхідно виконати інтерпретатору скрипта
   c. незалежний модульний програмний модуль, що динамічно підключається до основної програми, призначений для розширення та/або використання її можливостей
6. Яким символом позначаються коментарі при написанні inf-файлів?
   a. %
   b. //
   c. ;
   d. /*…*/
7. Як позначається секційний заголовок при написанні inf-файлів?
   a. [ім’я]
   b. title = ім’я
   c. class ім’я: title
8. Що таке параметр в контексті теми inf-файли?
   a. величина, значення якої служить для розрізнення елементів деякої множини між собою
   b. вираз (зазвичай одне слово), що знаходиться в полі секції, розташоване ліворуч від першого символу “=”
   c. об’єкти, що володіють принципово більш простою структурою та змістом у порівнянні з іншими об’єктами у своєму класі, тобто такі,

які, навіть будучи взятими разом, не дають повного уявлення про всі класи

9. Що таке значення в контексті теми inf-файли?
    a. асоціативний зв'язок між знаком і предметом позначення
    b. призначення певного класу, закладене конкретною особливістю.
    c. вираз, що стоїть праворуч від параметра
10. Який інтерпретатори скриптів INF за замовчуванням присутній в MS Windows?
    a. SETUPAPI
    b. Object REXX
    c. GNASH
    d. SWFDEC
11. Вкажіть в якому пункті правильно наведений формат запуску бібліотеки за допомогою RunDLL32
    a. rundll32.exe libraryname*EntryPoint, parameters
    b. rundll32.exe libraryname parameters*EntryPoint
    c. rundll32.exe libraryname, EntryPoint parameters
    d. libraryname rundll32.exe
12. Який надбудований над SETUPAPI інтерпретатор дозволяє виконувати додаткові функції?
    a. ADVANCEDINF
    b. Object REXX
    c. GNASH
    d. SWFDEC
13. Який із перерахованих заголовків не належить до ідентифікаційного заголовка?
    a. стандартний
    b. драйверний
    c. розширений
    d. змішаний
14. Які параметри визначають драйверний заголовок?
    a. Signature і Class
    b. Class і ClassGUID
    c. Class і DriverVer
    d. Class і Provider
15. Що відбувається при визначенні скрипта як драйверного?
    a. система визначає точки відкату, щоб в разі помилки вернутися на те місце до якого скрипт був викликаний
    b. система готується до перезавантаження для того, щоб драйвер почав коректно працювати
    c. в системі ініціюється реакція на зміну системної частини реєстру
16. Яку інформацію варто вказувати, щоб показати, що це є розширений заголовок?
    a. версія інтерпретатора

b. назва інтерпретатора
c. повний шлях до інтерпретатора

17. Якщо в секції [DestinationDirs] не визначено ні однієї директорії, то куди будуть копіюватися файли?
    a. в каталог за замовчуванням, яким є DIRID_WINDOWS
    b. в каталог за замовчуванням, яким є DIRID_SYSTEM
    c. в каталог за замовчуванням, яким є DIRID_INF
    d. в каталог за замовчуванням, яким є DIRID_BOOT
18. Що станеться у разі виконання наступного скрипта:
    [DefaultInstall]
    CopyFiles = cpf.files
    [cpf.files]
    filenam2.exe, filename.ext, 2
    a. *cpf.files* збирається з файлів *filenam2.exe* і *filename.ext*
    b. *filenam2.exe* перейменовується на *filename.ext*
    c. *filename.exe* перейменовується на *filenam2.ext*
    d. замість файлів *filename.exe* і *filenam2.ext* вставляється файл *cpf.files*

Теми для самостійного опрацювання

1. Структурні блоки INF-файлу.
2. Копіювання файлів за допомогою INF-скриптів.
3. Робота з INI-файлами.

Список використаної літератури

1. *Агуров П. В.* Интерфейсы USB. Практика использования и программирования / П. В. Агуров. – СПб.: БХВ-Петербург, 2004. – 576 с.
2. *Они У.* Использование Microsoft Windows Driver Model. Для профессионалов / Уолтер Они. – [2-е изд.]. – СПб.: Питер, 2007. – 768 с.
3. *Орвик П.* Windows Driver Foundation. Разработка драйверов / Пенни Орвик, Гай Смит. – М.: Русская редакция, 2008. – 880 с.
4. Павлов Станислав Введение в разработку компонентов драйверов и приложений для Windows XP Embedded / Станислав Павлов. – М.: Quarta technologies, 2004. – 27 с.
5. *Солдатов В. П.* Программирование драйверов Windows / В. П. Солдатов. – [2-е изд., перераб. и доп.]. – М.: ООО “Бином-Пресс”, 2004. – 480 с.

ДОДАТОК А. ГЛОСАРІЙ

Аварійний останов bugcheck – помилка, що генерується, коли цілісність структур ядра Windows була безповоротно порушена. Іноді також називається системним збоєм. Аварійні останови bugcheck генеруються тільки помилками при виконання процесів режиму ядра. Коли відбувається аварійний останов bugcheck, система намагається завершити роботу найбільш організованим чином і, в деяких версіях Windows, виводить повідомлення про помилку на блакитному екрані. Систему можна також набудувати для створення дампу файлу останову, який можна потім проаналізувати за допомогою відладчика ядра.

Адресний простір ядра (kernel address space) – блок віртуальної пам'яті, виділений для використання кодом режиму ядра.

Витиснена сторінка (paged out) – сторінка, тимчасово видалена з пам'яті і записана на диск, доки вона знову не знадобиться.

Віддалений код – код, який виконується поза контекстом, що спочатку його містить.

Диспетчерські (Dispatch**) точки входу** – точки входу Dispatch драйвера викликаються диспетчером введення/виведення, щоб запросити драйвер ініціювати деяку операцію введення/виведення.

Довільний потік (arbitrary thread) – потік, що виконується процесором, коли система "позичає" його для виконання процедури драйвера, наприклад, такий як процедура ISR. Процедура драйвера не знає, який процес є власником потоку.

Драйвер (англ. driver) – це комп'ютерна програма, за допомогою якої інша програма (звичайно операційна система) одержує доступ до апаратного забезпечення деякого пристрою. У загальному випадку, для використання будь-якого пристрою (як зовнішнього, так і внутрішнього) необхідний драйвер. Але звичайно з операційними системами поставляються драйвери для ключових компонентів апаратного забезпечення, без яких система не зможе працювати. Однак для деяких пристроїв (таких, як графічна плата або принтер) можуть знадобитися спеціальні драйвери, що надаються виробником пристрою.

Драйвер BSD (boot-start driver) – драйвер, який встановлюється під час завантаження операційної системи.

Драйвер класу (class driver) – драйвер, який зазвичай надає незалежну від апаратури підтримку для класу фізичних пристроїв і поставляється корпорацією Microsoft.

Драйвер фільтру (filter driver) – драйвер, який модифікує або відстежує запити введення/виведення при їх проходженні через стек пристроїв.

Драйвер шини (bus driver) – драйвер, який перераховує пристрої, підключені до шини.

Драйверна концепція – невід'ємна частина сучасних операційних систем. Ця концепція – основа взаємодії системи (користувача) з будь-якими пристроями (системними/периферійними, реальними/віртуальними і т.д.).

Драйверна модель Windows (Windows Driver Model, WDM) – драйверна модель, що передує моделі VVDF, що підтримує розробку драйверів для сімейства операційних систем Microsoft Windows NT, починаючи з Windows 2000.

Драйверний пакет (driver package) – інсталяційний пакет, що включає драйвер і допоміжні файли для нього.

Запит IOCTL (I/O control) – тип запиту введення/виведення, що застосовується для взаємодії з драйвером для інших цілей, ніж читання або запис в пристрій. Наприклад, додаток може використовувати запит IOCTL для зміни конфігурації пристрою або отримати номер версії драйвера.

Значення – вираз, що стоїть праворуч від параметра. Наприклад, Parameter = value.

Ідентифікатор екземпляра пристрою (device instance ID) – ідентифікаційний рядок, що надається системою, який однозначно ідентифікує екземпляр пристрою в системі.

Інтерпретатор – програма, що розпізнає синтаксис скрипта і послідовно виконує команди, вказані в скрипті.

Інтерфейс драйвера пристрою (device driver interface, DDI) – колекція системних процедур, що викликаються драйвером для взаємодії з сервісами ядра. По суті, інтерфейс DDI – це еквівалент інтерфейсу API для драйверів. Імена процедур інтерфейсу DDI зазвичай починаються префіксами, що вказують на їх призначення.

Клас інтерфейсів пристрою (device interface class) – група інтерфейсів, які зазвичай застосовні до певного типу пристрою і є засобом, за допомогою якого драйвери надають пристрій додаткам і іншим драйверам.

Команда відладчика (debugger command) – утиліта командного рядка відладчика WinDbg, за допомогою якої можна виконувати різні базові налагоджувальні операції.

Коментар, закоментований рядок – усі символи праворуч від символу;. Коментар може бути розташований у робочому, діючому рядку за умови, що він буде праворуч від будь-яких діючих символів.

Менеджер PNP (PNP manager) – підсистема ядра, що управляє операціями Plug and Play.

Менеджер драйвера (driver manager) – компонент UMDF, який створює і видаляє хост-процеси драйвера, містить інформацію про їх статус, а також відповідає на повідомлення від відбивача.

Менеджер об'єктів (object manager) – підсистема ядра, що управляє об'єктами ядра.

Об'єкт драйвера (driver object) – об'єкт, що представляє драйвер.

Об'єкт зворотного виклику (callback object) – створений драйвером об'єкт, в якому драйвер реалізує інтерфейси зворотного виклику, які потрібні для обробки подій, що викликаються одним або декількома інфраструктурними об'єктами.

Об'єкт інфраструктури (framework object) – об'єкт, керований WDF.

Об'єкт колекції (collection object) – об'єкт KMDF, що містить ланцюговий список (linked list) інших об'єктів KMDF будь-яких типів.

Об'єкт переривання (interrupt object) – об'єкт KMDF, що представляє підключення джерела апаратного переривання і драйверної процедури обробки переривання до системної таблиці векторів переривань.

Об'єкт пристрою (device object) – об'єкт пристрою, що представляє участь драйвера в обробці запитів введення/виведення певного пристрою.

Об'єкт таймера (timer object) – об'єкт KMDF, за допомогою якого драйвер може запрошувати функцію зворотного виклику періодично через рівні проміжки часу або один раз після закінчення вказаного періоду часу.

Об'єкт фізичного пристрою (physical device object, PDO) – об'єкт пристрою, що представляє об'єкт для свого драйвера шини.

Об'єкт функціонального пристрою (functional device object, FDO) – об'єкт пристрою, що представляє об'єкт для функціонального драйвера.

Обробник переривання (interrupt service routine, ISR) – процедура, що реалізовується драйвером пристрою для обробки апаратних переривань.

Параметр – вираз (зазвичай одне слово), що знаходиться в полі секції, розташоване ліворуч від першого символу =, наприклад Parameter =... . Не всі рядки поля секції обов'язково містять параметри.

Переривання (interrupt) – сповіщення, що посилається системі, що відбулося щось поза рамками звичайної обробки потоку, наприклад, апаратна подія, яку необхідно обробити якнайскоріше.

Перехоплювачі переривань – сервіси системи для установки власних процедур-обробників системних функцій.

Підписаний драйвер – це драйвер пристрою, який містить цифровий підпис. Цифровий підпис – це позначка електронного захисту, що може вказувати на видавця програмного забезпечення та повідомляти, чи було змінено вихідний вміст пакета драйвера. Якщо драйвер підписано видавцем, який засвідчив його справжність у центрі сертифікації, – можна бути впевненим, що драйвер справді надійшов від цього постачальника і його не було змінено. Windows сповіщує одним із нижченаведених повідомлень, якщо драйвер не підписано, підписано видавцем, підписано непідтвердженим видавцем чи змінено після випуску.

Підсистеми ядра (kernel subsystems) – компоненти ядра Windows, що підтримують базові сервіси, такі як Plug and Play, і надаючі процедури інтерфейсу DDI, які дозволяють драйверам режиму ядра взаємодіяти з системою.

Політика енергоспоживання (power policy) – набір правил, що визначають, як і коли система або пристрій переходить з одного стану енергоспоживання в інше.

Режим користувача (user mode) – обмежений режим роботи, в якому виконуються додатки і драйвери UMDF, в якому заборонений прямий доступ до процедур і структур даних ядра Windows.

Прикладний програмний інтерфейс (англ. Application Programming Interface, API) – набір визначень взаємодії різнотипного програмного забезпечення. API – це зазвичай (але не обов’язково) метод абстракції між низькорівневим та високорівневим програмним забезпеченням.

Пріоритети диспетчеризації – це пріоритети IRQL, де лише усередині найнижчого пріоритету IRQL, рівного PASSIVE_LEVEL є градація потоків по пріоритетах планування (scheduling), частина з яких може мати потоки додатків режиму користувача.

Процедура DPC (Deferred Procedure Call) – процедура, яку код, що виконується на рівні DIRQL, може відкласти і виконати пізніше на рівні DISPATCH_LEVEL.

Процедура ISR (Interrupt Service Routine) – процедура, що реалізовується драйвером пристрою для обробки апаратних переривань.

Процедура завершення введення/виведення (I/O completion routine) – Процедура драйвера, що виконується по завершенню запиту введення/виведення розташованим нижче драйвером в стеку.

Процедура обслуговування (service routine) – процедура, що виконує обробку для переривання.

Прямий доступ до пам’яті (direct memory access, DMA) – метод обміну даними між пристроєм і системною пам’яттю без участі центрального процесора.

Pn Manager (PNP-менеджер) – один з головних компонентів операційної системи. Складається із двох частин: PNP-менеджера режиму користувача та режиму ядра. Перший в основному взаємодіє з користувачем, коли тому потрібно, наприклад, установити нові драйвера й т.д. А другий управляє роботою драйверів, їх завантаженням тощо.

Реєстр Windows – база даних, що зберігає параметри і налаштування для операційних систем Microsoft Windows 32-бітних версій, 64-бітних версій та Windows Mobile. Він містить інформацію і параметри настроювання для всіх апаратних засобів, програмного забезпечення, користувачів тощо. Кожен раз, коли користувач змінює будь-які параметри в "Панелі керування", зміни відбиваються у реєстрі. Реєстр Windows було введено, щоб відмовитись від використання файлів ini, що використовувалися для збереження параметрів конфігурації програм Windows раніше (тобто кожна програма зберігала свої настроювання в окремому файлі). Тому ці файли мали тенденцію бути розкиданими по всій системі, що робило важким спостереження і контроль за ними.

Реєстрація – процес передачі адреси функції, що викликається.

Режим ядра (kernel mode) – привілейований режим, в якому дозволено виконувати відповідальні інструкції (команди процесора).

Рівень IRQL (Interrupt Request Level) – чисельне значення, яке Windows призначає кожному перериванню. У разі конфлікту переривання з вищим рівнем IRQL має пріоритет, і його процедура обслуговування виконується першою.

Рівень виконання DISPATCHLEVEL – рівень IRQL, на якому виконується код диспетчеризації потоків Windows, код для роботи із сторінками, а також код багатьох функцій драйверів режиму ядра.

Рівень запиту переривання пристрою (DIRQL) – діапазон значень рівнів IRQL, що асоціюється з перериваннями пристроїв. Точний діапазон рівнів DIRQL залежить від певної процесорної архітектури.

Робочий елемент (work item) – механізм, що виконується процедурами рівня HIGH-IRQL для часткового виконання на рівні IRQL = PASSIVE LEVEL.

Робочий стан системи (system working state) – повністю робочий стан енергоспоживання.

Ручна диспетчеризація (manual dispatch) – спосіб відправки запитів з черги, при якому драйвер витягує запити з черги самостійно, коли йому це потрібно, викликаючи метод черги.

Секція, секційний заголовок – рядок, що знаходиться в квадратних дужках, наприклад: [section.name]. Секція містить в собі всі рядки, розташовані нижче рядки секційного заголовка аж до кінця файлу або ж до наступного

секційного заголовка, що позначає наступну секцію. Рядки, що відносяться до секції, називаються полем секції.

Служби (Services) – це процеси режиму користувача, для запуску та функціонування яких реєстрація інтерактивного користувача в системі не потрібна. І тому, переважна більшість служб не мають призначеного для користувача інтерфейсу. Це єдина категорія користувальницьких програм, які можуть працювати в такому режимі. Служби використовуються, наприклад, для реалізації серверів. Вони можуть запускатися, як при завантаженні операційної системи, так і після неї.

Спін-блокування (spin lock) – об'єкт синхронізації, який можна використовувати при виконання на рівні IRQL = DISPATCH_LEVEL.

Стек драйвера (driver stack) – ланцюжок драйверів, що асоціюються з одним або декількома стеками пристроїв для надання підтримки роботи пристроїв.

Стек пристроїв (device stack) – колекція об'єктів пристроїв і зв'язаних драйверів, які обробляють взаємодію з певним пристроєм.

Файл inf (inf) – текстовий файл, що містить дані по установці драйвера.

Файл мови опису інтерфейсів (interface definition language file) – файл, в якому за допомогою структурованої мови опису інтерфейсів визначаються інтерфейси і об'єкти COM.

Файловий об'єкт (file object) – об'єкт WDF, що представляє один випадок використання окремого файлу і що містить інформацію для даного використання.

Фатальна помилка (fatal error) – помилка, після якої драйвер або система не може відновитися.

Функціональний драйвер (function driver) – основний драйвер пристрою.

Чисто програмний драйвер (software driver) – драйвер, який не управляє ніяким апаратним устаткуванням, ні прямим, ні непрямим (наприклад, за допомогою протоколу USB) чином.

Ядро (англ. kernel) – базова компонента операційної системи, що реалізує інтерфейс між прикладними процесами та обладнанням комп'ютера. Завантажується в оперативну пам'ять комп'ютера і безпосередньо взаємодіє з апаратурою, забезпечуючи керування апаратними засобами (при цьому використовуються драйвери (модулі ядра) підключеного в систему обладнання), підтримку одночасної роботи багатьох користувачів (багатокористувацький режим), підтримку паралельного виконання багатьох процесів в системі (багатозадачність).

ДОДАТОК Б. БАЗОВІ ТЕРМІНИ РОЗРОБКИ ДРАЙВЕРІВ ПІД ОПЕРАЦІЙНОЮ СИСТЕМОЮ WINDOWS NT

Abstraction – абстракція, представлення складного предмету штучно створеним формальним описом. Абстракція дозволяє відійти від розгляду деяких надмірно конкретних питань реалізації свого прототипу. Абстракції, Microsoft DDK, що вводяться, виникли, головним чином з прагнення полегшити переносність коду на інші апаратні платформи (забезпечення HAL) і прагнення уберегти розробника драйвера від необхідності вникати в тонкощі постійно змінних версій апаратного забезпечення для кожної конкретної платформи (наприклад, об'єкт адаптера дозволяє абстрагуватися від реалізацій контролерів DMA).

Access Violation – порушення доступу. Спроба доступу до області пам'яті, яка викликала виключення унаслідок захисту доступу до сторінок пам'яті (унаслідок того, що дана сторінка відноситься до набору для іншого процесу, а не з міркувань безпеки).

ACPI (Advanced Configuration and Power Interface) – абстрактний інтерфейс, що визначає механізми управління енергоспоживанням і конфігурації апаратного забезпечення і компонентів операційної системи.

ACPI Driver. У системах з ACPI BIOS програмне забезпечення HAL забезпечує завантаження ACPI драйвера під час старту операційної системи в корені дерева пристроїв.

AddDevice – функція, яку обов'язково повинні підтримувати WDM драйвери, для того, щоб в своєму складі мати обробник, який буде викликаний у момент виявлення пристрою. Реєстрація AddDevice виконується у момент роботи DriverEntry.

Affinity – спорідненість до процесора. У багатопроцесорному середовищі цей параметр визначає, на якому процесорі слід виконувати даний код. На симетричних багатопроцесорних системах (SMP) – за умовчанням – потоки можуть виконуватися на будь-якому. Відповідно, потоки, що належать одному процесу, можуть виконуватися на різних процесорах одночасно.

Callback, callback function – функція зворотного виклику, callback функція. Прийом програмування, коли один програмний потік повідомляє адресу відомої йому функції іншому програмному потоку з тією метою, щоб другою виконав виклик цієї вказаної функції в певний момент. Ннаприклад, якщо поступив якийсь сигнал від апаратного забезпечення або закінчився певний інтервал очікування.

Class Driver – класовий драйвер. Високорівневий драйвер, який представляє підтримку цілого класу пристроїв (наприклад, клавіатури і миші).

При цьому класовий драйвер абстрагується від їх апаратних особливостей, підтримка яких покладається на драйвери нижчого рівня, спілкування з якими відбувається за допомогою IOCTL запитів, callback функцій або функцій, що експортуються цими драйверами нижнього рівня.

CurrentControlSet – поточна працююча конфігурація. Підрозділ Системного Реєстру, що описує поточну конфігурацію системи: HKLM\System\CurrentControlSet00X, що називаються, де X – число від 1 до 3. Інформація якого конкретно фрагмента реєстру використовується як поточна (тобто значення X), можна визначити за значенням параметра Current в розділі HKLM\System\Select, який є покажчиком на один з підрозділів HKLM\System\CurrentControlSet00X. Насправді, якщо щось додати або видалити в підрозділі CurrentControlSet, то така сама зміна відбудеться і в тому підрозділі CurrentControlSet00X, на який "указує" параметр Current.

Deferred Procedure Call – процедура відкладеного виклику, тобто функція, яка буде викликана. З таких функцій складається черга, яка підпорядкована менеджеру (диспетчеру) введення/виведення. Для обліку DPC процедур операційна система підтримує DPC об'єкти.

Devnode – елемент в дереві пристроїв PNP-менеджера. Вузол devnode стека пристрою використовується PNP-менеджером для зберігання інформації конфігурації і для відстежування пристрою.

Device Extension – розширення об'єкту пристрою. Структура, кінцевий вид якої визначається автором. Дана структура створюється в сам момент створення об'єкту пристрою за допомогою системного виклику IoCreateDevice, одним з аргументів якого є її розмір – у момент створення об'єкту пристрою система виділяє місце (у несторінковому пулі пам'яті) і під цю "авторську" структуру.

Device ID – ідентифікатор пристрою.

Device Instance – фізичний апаратний компонент. Екземпляр пристрою, можливо, один з декількох інших однотипних пристроїв, що обслуговуються даним драйвером. Це може бути як геометрично великий і автономний фрагмент (монітор або привід CD-ROM), так і дрібний, фізично невіддільний від інших, поза сумнівом солідних пристроїв усередині сучасного комп'ютера, компонентів, наприклад, набір мікросхем на материнській платі для обслуговування PCI шини.

Device Stack, Driver Stack – стек пристроїв, стек драйверів – лише об'ємне дерево пристроїв. Його можна розглянути, якщо звернутися в програмі DeviceTree, що поставляється у складі пакету Microsoft DDK.

DEVICEID – ідентифікатор пристрою. Інформація, що ідентифікує пристрій. У inf-файлі, що використовується при інсталяції, а пізніше – в Системному Реєстрі інформація про пристрій зберігається у вигляді рядка, формат якого може бути визначений, наприклад, як

VEN_XXXX&DEV_YYYY&SUBSYS_ZZZZZZZZ&REV_VV, де XXXX – ідентифікатор виробника (VENDOR – постачальник), YYYY – ідентифікатор пристрою, ZZZZZZZZ і W уточнюючі параметри (ці числа зберігаються у внутрішніх регістрах PNP пристрою і стають доступними при його підключенні до шини).

DIRQL – рівні апаратних переривань. Обробка сигналів переривань (IRQ), що поступають від реальних пристроїв, вважається важливою справою і повинна проходити при підвищених рівнях пріоритету режиму ядра (IRQL). Тому даним рівням дана особлива назва Device IRQL, тобто DIRQL. Рівні пріоритетів DIRQL в системах на базі Intel займають верхні 16 IRQL рівнів в операційній системі.

Dispatch Routines – робочі процедури. Функції, які реєструється в найпершій процедурі драйвера, що викликається (DriverEntry). Реєстрація проводиться шляхом заповнення елементів масиву MajorFunction (покажчик на початок цього масиву DriverEntry отримує побічно через аргументи свого виклику). Індексом в цьому масиві є коди IRP_MJ_Xxx, тобто описані числами типи пакетів IRP. Якщо драйвер вважає за необхідне обробляти IRP запити якого-небудь типу, то у відповідному елементі масиву MajorFunction він реєструє відповідну функцію (записує її адресу). Диспетчер введення/виведення, орієнтуючись на заповнення цього масиву, викликає потрібні функції драйвера – dispatch routines. Сенс даного словосполучення - процедури драйвера (а не диспетчерські!), що "диспетчеризуються", що правильніше буде замінити на "робочі процедури" (функції) драйвера. Оскільки поза драйвером важливі тільки адреси робочих процедур (які і реєструє функція DriverEntry), то всі робочі процедури драйвера можуть мати абсолютно довільні імена. Рідкісні винятки становлять обов'язкові імена функцій в деяких спеціальних драйверах, наприклад, відео.

DMA (Direct Memory Access) – метод обміну даними між пристроєм і оперативною пам'яттю без участі центрального процесора. Процес DMA перенесення даних може протікати або під управлінням самого пристрою (bus-mastering DMA або first-party DMA) або під управлінням системного контролера DMA (slave DMA або third-party DMA).

DriverEntry – функція драйвера, яка буде викликана першою при його завантаженні. Дана функція виконується реєстрацію основних процедур, зокрема робочих (Dispatch Routines). Для WDM і не-WDM драйверів завдання, які повинна вирішити ця функція, трохи розрізняються. Функція DriverEntry виконується на рівні IRQL рівному PASSIVE_LEVEL і навіть може бути розміщена в сторінковій організованій пам'яті (шляхом спеціальних директив для редактора зв'язків, лінкера).

Enumeration – процес перерахування (нумерації). Позначає послідовність дій з боку системних компонентів по виявленню пристроїв, реалізованих відповідно до специфікації PNP. Можливий унаслідок того, що пристрої

надають про себе інформацію у вигляді ідентифікаційних кодів по протоколах, специфічних для кожного типу шини (PCI, USB і так далі).

Enumerator – системний компонент, що здійснює процес перерахування (enumeration). Як правило, в цій ролі виступають шинні драйвери, які повідомляють про виявлені пристрої PNP Менеджера, після чого той або робить кроки по завантаженню і старту відповідного драйвера (який він визначає по записах в Системному Реєстрі), або починає процес установки драйвера (якщо відповідних записів не виявлено). Нумерація деяких пристроїв виконується шинними фільтр-драйверами, наприклад, ACPI драйвером.

Filter Device Object (FDO) – об'єкт пристрою, який створюється фільтр-драйвером. Має наступну формальну відмінність – цей пристрій залишається безіменним (драйвер не указує його імені при створенні об'єкту) і не має символьного посилання. Цей об'єкт пристрою, як правило, не призначений для безпосереднього доступу до нього, але після належного підключення його під або над пристроєм основного драйвера, це фільтр-пристрій пропускає через себе все IRP пакети, насправді призначені основному драйверу.

Filter Driver (фільтр-драйвер) – драйвери, що підключаються до основного драйвера або зверху (Upper Filter), або знизу (Lower Filter). Фільтр-драйвери (їх може бути декілька) виконують фільтрацію IRP пакетів. У одного основного драйвера може бути декілька фільтр-драйверів, але вони для основного драйвера як би не існують. Розбиття на основні і фільтр-драйвера робить сам розробник основного драйвера, визначаючи порядок їх установки і завантаження, що забезпечує належну конфігурацію стека пристроїв в цьому місці дерева пристроїв. Як правило, для основного драйвера Filter Drivers невидимі або працюють прозоро.

HAL (**Hardware Abstraction Layer**) – шар апаратних абстракцій в Windows NT, який покликаний приховати специфіку апаратної платформи (Intel32, Intel64, Alpha) від решти компонентів операційної системи, забезпечуючи малі витрати при перенесенні системи або елементів програмного забезпечення. Рівень HAL надає процедури, які дозволяють абстрагуватися від апаратних тонкощів, як, наприклад, деталі реалізації шин введення/виведення, переривань, DMA операцій тощо. Для не-WDM драйверів, наприклад, рівень HAL надає функції для отримання інформації про підключені до шин пристрої.

Hardware branch – підрозділ Системного Реєстру HKLM\Hardware. Розширена множина, що описує всю апаратуру, будь-коли встановлену на даному комп'ютері. Підмножина цього списку, що називається hardware tree, яка є резидентною, знаходяться в оперативній пам'яті, містить тільки реально присутні в системі пристрої.

HID (human interface device) – пристрій інтерфейсу з користувачем, наприклад: миша, клавіатура.

ID класу (class ID, CLSID) – ідентифікатор GUID, який унікально ідентифікує певний об'єкт COM. Ідентифікатори CLSID потрібні для об'єктів COM, створюваних фабрикою класу, але є необов'язковими для об'єктів, що створюються по-іншому.

IO stack location – стек введення-виведення в пакеті IRP.

IOCTL (I/O ConTroL code) – код управління введенням/виведенням. Дозволяє звертатися до драйвера із запитами, що відрізняються від операцій читання і запису в пристрій (хоча і вони легко реалізуються через IOCTL запити). Розробник драйвера має можливість створювати свої власні коди IOCTL. Даний код є одним з аргументів функції режиму користувача DeviceIoControl. Що поступає в драйвер в результаті роботи цієї призначеної для користувача функції і диспетчера введення/виведення пакет IRP матиме код IRP_MJ_DEVICE_CONTROL, а одним з внутрішніх параметрів даного пакету IRP буде вказаний у виклику функції DeviceIoControl код IOCTL.

IOManager – менеджер (диспетчер) введення/виведення, ДВВ, працює в режимі ядра і відноситься до основних компонентів операційної системи. Відає питаннями введення/виведення, зокрема, пов'язуючи додатки в режимі користувача з власне драйверами. Все спілкування ДВВ з драйверами відбувається виключно за допомогою виклику його процедур і передачі ним (через заголовок) стандартизованій структури даних IRP, в якій сформульована вся суть його звернення до драйвера.

IRP (Input/Output Request Packet, IRP request, IRP packet) – пакет запиту на введення/виведення. IRP запит, IRP пакет. Рівноцінні переклади, оскільки звернення ДВВ до драйвера є запит за допомогою IRP пакету, а IRP пакет не має сенсу, окрім як в контексті поводження з якимсь запитом до драйвера (або від ДВВ, або від іншого драйвера, але, знову-таки, за посередництва ДВВ).

IRQ (Interrupt Request Line) – апаратна лінія (провідник) від периферійного пристрою, контролера шини, іншого процесора (у багатопроцесорній системі), по якій вони можуть подавати сигнали про те, що їм потрібне обслуговування від даного процесора (мікропроцесора).

IRQL (Interrupt ReQuest Level) – рівень пріоритету виконання. Термін, що історично дістався від завдань по обслуговуванню переривань (IRQ), але тепер пов'язаний не тільки з ними. Пріоритети, прийняті для програмного коду, що працює в режимі ядра. Планування потоків з пріоритетами IRQL хоч би на 1 вище мінімального (PASSIVE_LEVEL) сильно відрізняється від планування потоків в режимі користувача. У режимі ядра потік, що працює при деякому пріоритеті IRQL може бути перерваний тільки для виконання роботи потоком з вищим IRQL. Навіть потік з рівним пріоритетом IRQL повинен чекати природного закінчення роботи свого "рівноправного колеги". Потік може самостійно підвищити свій IRQL, проте, величина підвищення в деяких версіях Windows не довільна. Наприклад, працюючи на рівні PASSIVE_LEVEL (0),

потік може отримати від операційної системи Windows XP згоду тільки на рівень IRQL рівний 10 (для порівняння, DISPATCH_LEVEL має чисельне значення 3, а INTERRUPT_LEVEL чисельно рівний 13). На відміну від дій з підвищення власного IRQL, знижувати пріоритет потоку в режимі ядра категорично не рекомендується, оскільки наслідки можуть бути непередбачуваними.

ISR (Interrupt Service Routine) – процедура обслуговування переривань. Функція, яку драйвер реєструє для того, щоб вона отримувала управління в мить, коли апаратура, що обслуговується драйвером, передала сигнал переривання. Завдання цієї функції виконати деяку наймінімальнішу роботу і зареєструвати callback-функцію, що називається процедурою відкладеного виклику для обслуговування переривання (часто позначається ім'ям DpcForISR, проте автор драйвера може дати їй будь-яке ім'я).

Kernel mode – режим ядра, привілейований режим, в якому дозволено виконувати відповідальні інструкції (команди процесора). У режимі ядра можна достовірно визначати реально присутні в системі апаратні ресурси, безпосередньо звертатися до них, викликати потужні системні функції, впливати на проходження даних (підраховувати, кодувати/декодувати) і так далі. Програмування в режимі ядра має істотні особливості, перш за все, це стосується відповідальності і необхідності звертати увагу на ті питання, які в режимі користувача не виникають.

KMDF (Kernel Mode Driver Framework) – інфраструктура для драйверів режиму ядра.

Layering – багатошаровість, що підтримується моделлю WDM, можливість реалізовувати стекове з'єднання між драйверами. Знаходячись в стеку, верхній драйвер, що підключився до стека пізніше, має можливість адресувати/переадресовувати IRP запити нижнім драйверам, що знаходилися в стеку до нього. Абстракціями, які виступають дійовими особами в обмінах запитами, насправді є не драйвери, а об'єкти пристроїв – саме вони з'єднуються в стек і є адресатами в отриманні і передачі пакетів IRP.

Legacy Driver, NT Style Driver – успадкований драйвер (застарілий драйвер), драйвер "в стилі NT".

Major IRP Code – старший код IRP пакету. Число, яке позначає призначення пакету IRP, а значить і основний сенс даного звернення до драйвера. У пакеті Microsoft DDK кожне таке число має ще й символьне позначення, яке встановлене через #define директиву.

Microsoft Windows DDK, Device Driver Kit – пакет розробки драйверів, що включає компілятор, редактор зв'язків (лінкер), заголовки, бібліотеки, великий набір прикладів (частина з яких є драйверами, реально працюють в операційній системі) і документації. До складу пакету входить також відладчик WinDbg, що дозволяє проводити інтерактивну налагодження драйвера на двокомп'ютерної конфігурації і за наявності файлів налагоджувальних

ідентифікаторів операційної системи WinDbg крім того, дозволяє проглядати файли дампа (образу) пам'яті, отриманого при фатальних аварійних завершеннях роботи операційної системи (crash dump file).

Mini Driver (міні-драйвер) – драйвер, який за розміром менший "повного" драйвера. Звичайно реалізується у вигляді динамічно завантажуваної бібліотеки DLL й має оболонку у вигляді "повного" драйвера. Ймовірно, найвідомішим прикладом міні-драйвера є SCSI міні-порт-драйвер (міні-драйвер для SCSI порт-драйвера). У даному терміні в глосарії MS Windows DDK спостерігається відвертий збій, оскільки він трактується як DLL, яка використовує класовий драйвер для завершення своїх запитів.

Nonpaged Memory, Nonpaged Pool – несторінково організована пам'ять, несторінковий пул, несторінкова пам'ять. Віртуальна пам'ять, яка ніколи не може бути переміщена системою на жорсткий диск і завжди залишається у фізичній оперативній пам'яті. Цінний ресурс операційної системи, яким слід розпоряджатися вельми обачно. До пам'яті даного типу можна безпечно звертатися з потоків, що працюють на будь-якому рівні IRQL.

NTSTATUS – тип, що використовується як значення багатьма процедурами режиму ядра, яке повертається.

Object – об'єкт. У програмуванні драйверів об'єкт завжди є структурою або об'єднанням, з яким пов'язана одна з абстракцій. У режимі ядра є труднощі з реалізацією трюків C++ (оператора new, ідентифікації типів під час виконання і віртуальних методів) і основних прийомів об'єктно-орієнтованого програмування. До кожного об'єкту в режимі ядра додається набір функцій, і фірма Microsoft рекомендує працювати з об'єктами ядра тільки за допомогою цих спеціалізованих функцій.

Paged Memory, Paged Pool – сторінково організована пам'ять, сторінковий пул, сторінкова пам'ять. Віртуальна пам'ять, яка може бути переміщена системою на жорсткий диск, у будь-який момент, коли вона визнає цю операція доцільна (наприклад, при низькій активності застосування). Ця переміщена область має назву "paged", тобто "посторінково скинута на диск".

Plug and Play (скор. PNP) – комбінація системного програмного забезпечення, апаратного забезпечення пристрою і драйверної підтримки пристрою, за допомогою якої комп'ютерна система може з мінімальним або взагалі відсутнім втручанням з боку кінцевого користувача розпізнавати зміни в конфігурації апаратного забезпечення і настроюватися для роботи з цими змінами. Розроблена фірмою Microsoft при сприянні інших компаній.

PNP Manager, один з ключових компонентів операційної системи, що конструктивно складається з двох частин, PNP-менеджера, що працює в режимі ядра, і PNP-менеджера призначеного для користувача режиму. Перший взаємодіє з рештою компонентів системи і драйверів в процесі завантаження програмного забезпечення, необхідного для обслуговування наявних в системі

пристроїв. Його частина, що працює в режимі користувача, відповідає за взаємодію з користувачем в ситуаціях, що вимагають установки нових

Polling – це особливий метод програмування, при якому не пристрій посилає сигнали переривання драйверу, а сам драйвер періодично опитує пристрій, що обслуговується ним. Зокрема, пристрої USB конструктивно не можуть генерувати сигнали переривання, але можуть повідомляти про подібні бажання у момент їх опиту хост-контролером.

Pool Memory – пам'ять в пулах (сторінковому або несторінковому). Області в просторі пам'яті ядра (адреси вище 0x80000000 для стандартної конфігурації системи – оскільки для сервера можна визначити інакше), в яких можна динамічно виділяти, отримувати і звільняти області пам'яті.

Registry – Системний Реєстр. База даних різноманітних настроювальних та інформаційних параметрів (аж до ключа, використаного при інсталяції даного екземпляра Windows), як для додатків режиму користувача, так і конфігураційних параметрів для всієї системи. Файли, складові цієї бази, в 32-розрядних версіях операційної системи зберігаються в директорії %systemroot%\System32\Config\ і захищені системою від змін. Не дивлячись на те, що програмісти вважають за краще обходити стороною це "страшне" місце, проте, запис конфігураційних параметрів в Системний Реєстр є нормальною і рядовою практикою.Робота з Системним Реєстром через системні функції в Windows NT потребує використання кодування Unicode.

Routine – процедура, розуміється виключно як функція мови С. То ж відноситься і до всієї документації англійською мовою (щодо слова "routine").

Scatter/Gather Problem Проблема збірки/розбирання адрес. Виникла у момент визначення методів роботи апаратури DMA в системах з віртуальною адресацією.

Structure – структура, тип даних мови С. Складається з простих типів даних (char, int і тому подібне) і вкладених структур або об'єднань (union).

Symbolic Link – символьне посилання, символьний зв'язок. В області розробки драйверів – файловий об'єкт з особливими властивостями.

SYS – розширення імені-файлу двійкових файлів драйверів режиму ядра. Хоча вони схожі на файли DLL, файли SYS не мають прямого експортування.

System Paging File – системний файл для зберігання тимчасово не використовуваних областей сторінково організованої пам'яті (вивантажених з фізичної пам'яті).

Thread, Thread Object – програмний потік ("нитка"), об'єкт потоку. Потік є мінімальною одиницею виконання і планування в багатозадачній операційній системі. Для обліку потоків Windows використовує об'єкти потоків. У режимі ядра слід планувати роботу з такими потоками, щоб забезпечувати їх природне завершення, наприклад, сигналізуючи їм за допомогою об'єктів синхронізації.

UMDF (User Mode Driver Framework) – інфраструктура для драйверів призначеного для користувача режиму.

Unicode – двобайтове кодування символів алфавіту. Робить можливою підтримку всіх мов, що мають буквений або складовий алфавіт. Давно і широко застосовується в Windows NT. В режимі ядра існує достатній набір функцій для роботи з рядками Unicode.

Union – об'єднання, тип даних мови С, складається з простих типів даних (char, int і т. ін.) і вкладених структур або об'єднань. Реалізує доступ до однієї і тієї ж області пам'яті, як до даним різних типів.

User mode – режим користувача. Непривілейований режим, в якому виконуються звичайні додатки, що не мають можливості діставати доступ до системних даних інакше, як звертаючись до викликів функцій підсистем (наприклад, Win32, GDI, Posix), які, у свою чергу, вдаються до допомоги системних викликів. У режимі користувача всі потоки, що навіть мають найнижчий пріоритет, рано чи пізно отримують управління (можливість роботи) на відміну від потоків режиму ядра підвищених пріоритетів, які можуть бути перервані тільки потоками, що отримали ще вищий пріоритет.

User Space – простір пам'яті, який призначений для користувача. Область віртуальної пам'яті, виділена для роботи додатків користувача.

Virtual Memory – віртуальна пам'ять.

WDF (Windows Driver Foundation) – драйверна модель WDF.

WDM (Windows Driver Model) – це драйверна модель від Microsoft для ОС Windows, що прийшла на зміну попередньому середовищу написання драйверів для ОС Windows – VxD (virtual device driver). WDM – це спеціальний інтерфейс між програмним і апаратним забезпеченням. WDM драйвер влаштований таким чином, що будь-яка програма, що вміє працювати із цим стандартним інтерфейсом, зможе управляти пристроєм – більшість функцій уніфікована. У системах Windows 2000/XP/2003 використовуються тільки WDM-драйвери, які з'явилися ще в Windows 98. Подальшим розвитком WDM драйверів є драйвери WDF, значно спрощують і полегшують розробку драйверів WDM.

Windows API (application programming interfaces) – загальне найменування для цілого набору базових функцій інтерфейсів програмування додатків операційних систем родини Windows корпорації Майкрософт. Є найпрямішим способом взаємодії додатків Windows. Для створення програм, що використовують Windows API, Майкрософт випускає SDK, який називається Platform SDK і містить документацію, набір бібліотек, утиліт і інших інструментальних засобів.