Лекція 11. КЕРУВАННЯ ПРИСТРОЯМИ ВВЕДЕННЯ-ВИВЕДЕННЯ

- Завдання і організація підсистеми введення-виведення
- Способи виконання операцій введення-виведення
- Введення-виведення у режимі користувача
- Таймери та системний час
- Керування введенням-виведенням в Linux, Unix та Windows XP

11.1. Завдання підсистеми введення-виведення

Основним завданням підсистеми введення-виведення є реалізація доступу до зовнішніх пристроїв із прикладних програм, яка повинна забезпечити:

- ефективність (можливість використання ОС всіх засобів оптимізації, які надає апаратне забезпечення), спільне використання і захист зовнішніх пристроїв за умов багатозадачності;
- універсальність для прикладних програм (ОС має приховувати від прикладних програм відмінності в інтерфейсі апаратного забезпечення, надаючи стандартний інтерфейс доступу до різних пристроїв), при цьому потрібно завжди залишати можливість прямого доступу до пристрою, оминаючи стандартний інтерфейс;
- універсальність для розробників системного програмного забезпечення (драйверів пристроїв), щоб під час розробки драйвера для нового пристрою можна було скористатися наявними напрацюваннями і легко забезпечити інтеграцію цього драйвера у підсистему введення-виведення.

Забезпечення ефективності вимагає розв'язання кількох важливих задач.

- Передусім−це коректна взаємодія процесора із контролерами пристроїв. Відомо, що кожен зовнішній пристрій має контролер, який забезпечує керування пристроєм на найнижчому рівні і є фактично спеціалізованим процесором. Після отримання команди від ОС контролер забезпечує її виконання, при цьому пристрій якийсь час не взаємодіє із процесором комп'ютера, тому той може виконувати інші задачі. Виконавши команду, контролер повідомляє системі про завершення операції введення-виведення, генеруючи відповідну подію. Операційній системі в цьому разі потрібно спланувати процесорний час таким чином, щоб драйвери пристроїв могли ефективно реагувати на події контролера та було забезпечене виконання коду процесів користувача.
- Керування пам'яттю під час введення-виведення. Оперативна пам'ять є швидшим ресурсом, ніж зовнішні пристрої, тому ОС може підвищувати ефективність доступу до пристроїв проміжним зберіганням даних у пам'яті (із використанням таких технологій, як кешування і буферизація).

Під час ***спільного використання*** зовнішніх пристроїв мають виконуватися певні умови.

- ОС повинна мати можливість забезпечувати одночасний доступ кількох процесів до зовнішнього пристрою і розв'язувати можливі конфлікти (тобто необхідна підтримка синхронізації доступу до пристроїв). Деякі пристрої (наприклад, модем або сканер) можна використати тільки одним процесом у конкретний момент часу, тоді як жорсткий диск завжди використовують спільно;
- Слід забезпечити захист пристроїв від несанкціонованого доступу. Такий захист можна організувати або для пристрою як цілого (наприклад, можна відкрити модем

для доступу тільки певній групі користувачів), або для деякої підмножини даних пристрою (наприклад, різні файли на жорсткому диску можуть мати різні права доступу).

- У разі спільного використання пристрою треба розподілити операції введення-виведення різних процесів, для того щоб уникнути «накладок» даних одних процесів на дані інших (наприклад, під час спільного використання принтера важливо відрізняти одні задачі від інших і не переходити до друкування результатів наступної задачі до того, як завершилося виведення попередньої).

Оскільки пристрої введення-виведення доволі різноманітні, дуже важливо ***уніфікувати доступ*** до них із прикладних програм. Для реалізації цієї ідеї підсистема введення-виведення має використовувати набір базових абстракцій, під час застосування яких можна надати доступ до різних зовнішніх пристроїв узагальненим способом. Для більшості сучасних ОС такою абстракцією є абстракція файла, що відображається як набір байтів, з яким можна працювати за допомогою спеціальних операцій файлового введення-виведення. До таких операцій належать, наприклад, системні виклики відкриття файла `open()`, файлового читання `read()` і записування `write()`. Файл, що відповідає пристрою (його називають файлом пристрою), не відповідає набору даних на диску, а є засобом організації універсального доступу різних компонентів ОС і прикладних програм до деякого пристрою введення-виведення.

Зазначимо, що не всі пристрої добре «вписуються» у модель файлового доступу (до подібних пристроїв належить, наприклад, системний таймер). У цьому разі, з одного боку, ОС може надавати унікальний, нестандартний інтерфейс до таких пристроїв, з іншого − стандартного набору файлових операцій може бути недостатньо для використання всіх можливостей пристрою. Для вирішення цієї проблеми можна запропонувати два підходи.

1. Розширити допустимий набір операцій, створивши інтерфейс, що відображає особливості конкретного пристрою. Його будують на основі стандартного файлового інтерфейсу, створюючи операції, характерні для конкретного пристрою. Ці операції, в свою чергу, використовують стандартні виклики, подібні до `read()` і `write()`.
2. Надати прикладним програмам можливість взаємодіяти із драйвером пристрою безпосередньо. Для цього звичайно пропонують універсальний системний виклик (в UNIX його називають `ioctl()`, у Windows XP − `DeviceIoControl()`), параметри якого задають необхідний драйвер, команду, яку потрібно виконати, і дані для неї.

Як відомо, драйвер пристрою − це програмний модуль, що керує взаємодією ОС із конкретним зовнішнім пристроєм.

Драйвер можна розглядати як транслятор, що отримує на свій вхід команди високого рівня, зумовлені його інтерфейсом із операційною системою, а на виході генерує низькорівневі інструкції, специфічні для апаратного забезпечення, яке він обслуговує. Звичайно на вхід драйвера команди надходять від підсистеми введення-виведення, у більшості ОС їх можна задавати і у прикладних програмах. На практиці драйвери зазвичай записують спеціальні набори бітів у пам'ять контролерів, повідомляючи їм, які дії потрібно виконати. ***Драйвери практично завжди виконують у режимі ядра***. Вони можуть бути завантажені у пам'ять і вивантажені з неї під час виконання.

Набір драйверів, доступних для операційної системи, визначає набір апаратного забезпечення, із яким ця система може працювати. Якщо драйвер для потрібного пристрою відсутній, користувачу залишається лише чекати, поки він не буде створений

(альтернативою є самостійна розробка, однак більшість користувачів до розв'язання такого завдання не готові). Часто в цій ситуації користувач змушений перейти до використання іншої ОС. Для того щоб знизити ймовірність такого небажаного розвитку подій, підсистемі введення-виведення потрібно забезпечити зручний універсальний і добре документований інтерфейс між драйверами й іншими компонентами операційної системи. Наявність такого інтерфейсу і зручного середовища розробки драйверів дає змогу спростити їхнє створення і розширити коло осіб, які можуть цим займатися (розробка драйверів має бути до снаги виробникам відповідного апаратного забезпечення).

Зручність середовища розробки драйверів визначає набір функцій і шаблонів, наданих програмістові. Часто набір засобів, призначений для розробки драйверів під конкретну операційну систему, постачають розробники цієї ОС у вигляді окремого продукту, який називають *DDK* (Driver Development Kit). У нього входять заголовні файли, бібліотеки, можливо, спеціальні версії компіляторів і налагоджувачів, а також документація.

11.2. Організація підсистеми введення-виведення

Загальна структура підсистеми введення-виведення показана на рис. 11.1.

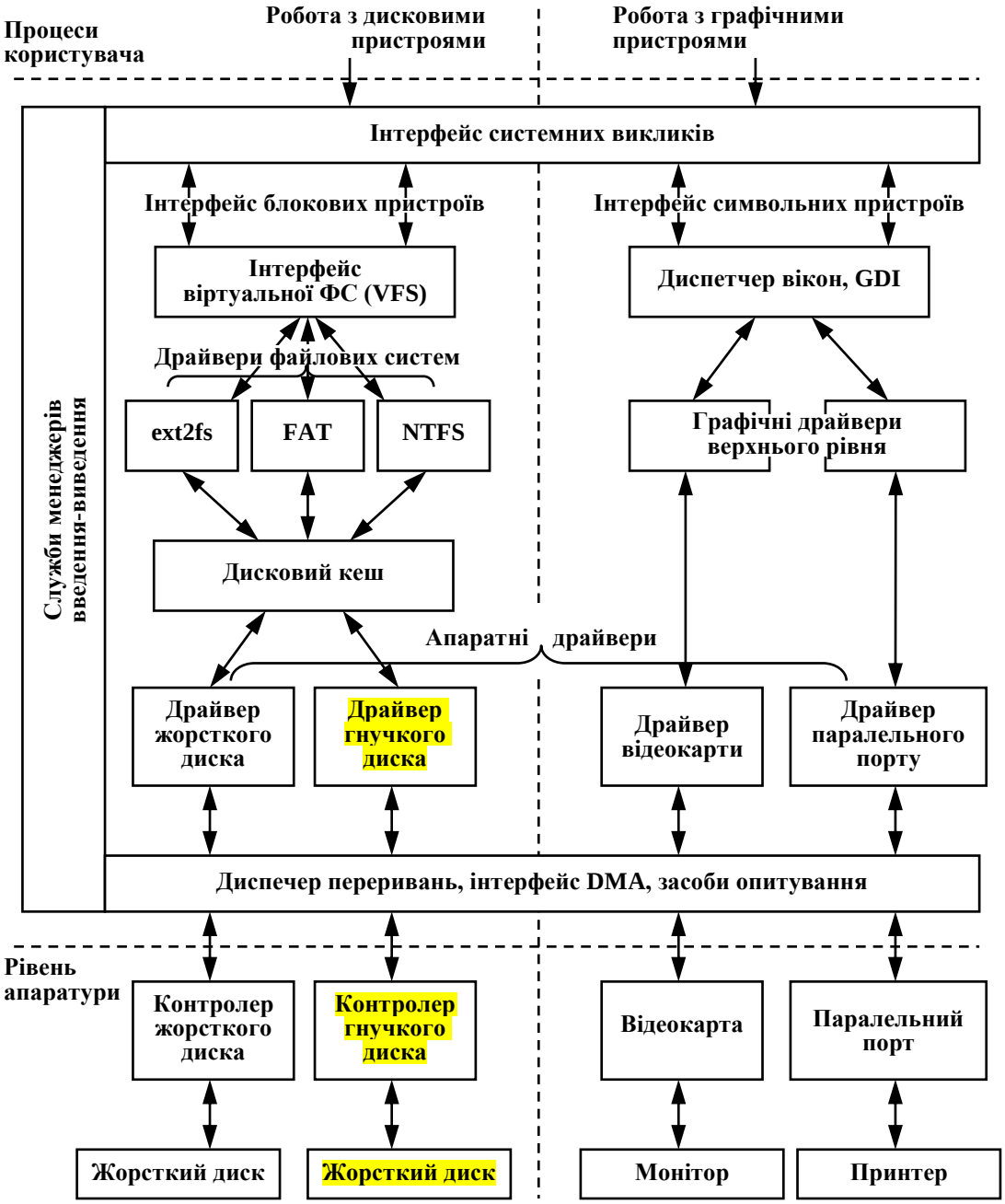

Рис.11.1. Багаторівнева структура підсистеми введення-виведення

Розглянемо особливості організації такої підсистеми.

- Структура підсистеми розділена на рівні, що дає змогу забезпечити, з одного боку, підтримку всього спектра зовнішніх пристроїв (на нижньому рівні, де розташовані апаратні драйвери), а з іншого − уніфікацію доступу до різних пристроїв (на верхніх рівнях). Зазначимо, що досягти повної ідентичності доступу до всіх можливих пристроїв майже неможливо, тому, крім горизонтальних рівнів, виділяються вертикальні групи пристроїв, взаємодія з якими ґрунтується на загальних принципах. На рис. 11.1 у такі групи об'єднані графічні пристрої та дискові накопичувачі.
- Деякі функції цієї підсистеми не можуть бути реалізовані у драйверах. Це, насамперед, засоби, що забезпечують координацію роботи всіх модулів підтримки введення-виведення у системі, які об'єднуються у *менеджер введення-виведення.* Він, з одного боку, забезпечує доступ прикладних програм до відповідних засобів введення-виведення (наприклад, через інтерфейс файлової системи, реалізований як набір системних викликів), з іншого − середовище функціонування драйверів пристроїв, реалізує загальні служби, такі як буферизація, обмін даними між драйверами тощо.
- Драйвери пристроїв теж розглядають на кількох рівнях. Поряд із низькорівневими апаратними драйверами для керування конкретними пристроями є драйвери вищого рівня. Вони взаємодіють із апаратними драйверами і на підставі отриманих від них даних виконують складніші завдання. Так, над драйвером відеокарти можна розмістити драйвери, які реалізовуватимуть складні графічні примітиви (наприклад, аналогічні рівню GDI у Windows-системах), а над апаратними драйверами дискових пристроїв − драйвери файлових систем. Як наслідок, код драйверів кожного рівня спрощується, кожен із них тепер розв'язує меншу кількість завдань.

Ще одна важлива класифікація драйверів уперше з'явилася в UNIX і відтоді її широко використовують, оскільки вона добре відображає специфіку різних пристроїв. Пристрої та драйвери відповідно до цієї класифікації розділяються на три категорії: *блокові або блок-орієнтовані* (block-oriented, block), *символьні* або *байт-орієнтовані* (character-oriented, character) і *мережні* (network).

- Для блокових пристроїв дані зберігають блоками однакового розміру, при цьому кожен блок має свою адресу, і за допомогою відповідного драйвера до нього можна отримати прямий доступ. Основним блоковим пристроєм є диск.
- Символьні пристрої розглядають дані як потік байтів, при цьому окремий байт адресований бути не може. Прикладами таких пристроїв є модем, клавіатура, миша, принтер тощо. Базовими системними викликами для символьних пристроїв є виклики читання і записування одного байта.
- Окремою категорією є мережні пристрої, які надаються прикладним програмам у вигляді *мережних інтерфейсів* зі своїм набором допустимих операцій, які відображають специфіку мережного введення-виведення (наприклад, ненадійність зв'язку). Деякі ОС реалізують у вигляді мережних драйверів не тільки засоби доступу до пристроїв, але й мережні протоколи.

Концепція драйверів є типовим прикладом розділення механізму і політики в операційних системах. Засобом реалізації механізму доступу до конкретного пристрою є драйвер. Він завжди виконує базові дії із доступу до цього пристрою, не цікавлячись, чому ці дії потрібно виконувати (він вільний від політики). Політику необхідно реалізовувати у програмному забезпеченні вищого рівня, що виконує конкретні операції введення-виведення.

11.3. Способи виконання операцій введення-виведення

Зовнішній пристрій взаємодіє із комп'ютерною системою через точку зв'язку, яку називають *портом* (port). Якщо декілька пристроїв з'єднані між собою і можуть обмінюватися повідомленнями відповідно до заздалегідь визначеного протоколу, то кажуть, що вони використовують *шину* (bus).

Як відомо, пристрої зв'язуються із комп'ютером через контролери. Є два базові способи зв'язку із контролером: через *порт введення-виведення* (I/O port) і *відображувану пам'ять* (memory-mapped I/O). У першому випадку дані пересилають за допомогою спеціальних інструкцій, у другому − робота із певною ділянкою пам'яті спричиняє взаємодію із контролером.

Деякі пристрої застосовують обидві технології відразу. Наприклад, графічний контролер використовує набір портів для організації керування і регіон відображуваної пам'яті для зберігання вмісту екрана.

Спілкування із контролером через порт звичайно зводиться до використання чотирьох регістрів. Команди записують у *керуючий регістр* (control), дані − у *регістр виведення* (data-out), інформація про стан контролера може бути зчитана із *регістра статусу* (status), дані від контролера − із *регістра введення* (data-in). Ядро ОС має реєструвати всі порти введення-виведення і діапазони відображуваної пам'яті, а також інформацію про використання пристроєм порту і діапазону.

Припустимо, що контролер може повідомити, що він зайнятий, увімкнувши біт `busy` регістра статусу. Застосування повідомляє про команду записування вмиканням біта `write` командного регістра, а про те, що є команда − за допомогою біта `cready` того самого регістра. Послідовність кроків базового протоколу взаємодії з контролером наведено нижче.

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

На першому етапі застосування займається *опитуванням пристрою* (polling), фактично воно перебуває в циклі активного очікування. Одноразове опитування здійснюється дуже швидко, для нього досить трьох інструкцій процесора − читання регістра, виділення біта статусу і переходу за умовою, пов'язаною з цим бітом. Проблеми з'являються, коли опитування потрібно повторювати багаторазово, у цьому разі буде зайвим завантаження процесора. Для деяких пристроїв прийнятним є опитування через фіксований інтервал часу. Наприклад, так можна працювати із CD (цей пристрій є досить повільним, що дає можливість між звертаннями до нього виконувати інші дії).

У більшості інших випадків потрібно організовувати введення-виведення, кероване перериваннями.

Базовий механізм переривань дає змогу процесору відповідати на асинхронні події. Така подія може бути згенерована контролером після закінчення введення-виведення або у разі помилки, після чого процесор зберігає стан і переходить до виконання оброблювача переривання, встановленого ОС. Цим знімають необхідність опитування пристрою − система може продовжувати звичайне виконання після початку операції введення-виведення.

Розглянемо деякі ***додаткові дії***, що виникають під час організації обробки переривань у сучасних ОС.

Насамперед необхідно мати можливість скасовувати або відкладати обробку переривань під час виконання важливих дій.

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

Зазначимо, що за деяких умов переривання вищого рівня можуть переривати виконання оброблювачів нижчого рівня.

Контролер переривань здійснює роботу з перериваннями на апаратному рівні. Він є спеціальною мікросхемою, що дає змогу відсилати сигнал процесору різними лініями. Процесор вибирає оброблювач переривання, на який потрібно перейти, на підставі номера лінії, що нею прийшов сигнал, її називають лінією запиту переривання або просто *лінією переривання* (IRQ line). У старих архітектурах застосовувалися контролери переривань, розраховані на 15-16 ліній переривання і на один процесор, сучасні системи мають спеціальні *розширені програмовані контролери переривань* (Advanced Programmable Interrupt Controllers, APIC), які реалізують багато ліній переривання (наприклад, для стандартного APIC фірми Intel таких ліній 255) і коректно розподіляють переривання між процесорами за умов багатопроцесорних систем. Ядро ОС зберігає інформацію про всі лінії переривань, доступні у системі. Драйвер пристрою дає запит каналу переривання (IRQ) перед використанням (встановленням оброблювача) і вивільняє після використання. Крім того, різні драйвери можуть спільно використовувати лінії переривань.

Оброблювач переривання може встановлюватися під час ініціалізації драйвера або першого доступу до пристрою (його відкриття). Через обмеженість набору ліній переривання частіше застосовують другий спосіб, у цьому разі, якщо пристрій не використовують, відповідна лінія може бути зайнята іншим драйвером. Перед тим як встановити оброблювач переривання, драйвер визначає, яку лінію переривання використовуватиме пристрій, який він обслуговує (інакше кажучи, чому дорівнює *номер переривання* для цього пристрою). Є кілька підходів до розв'язання цього завдання.

- Розробник може надати користувачу право самому задати номер цієї лінії. Це − найпростіше вирішення, однак його слід визнати неприйнятним, тому що користувач не зобов'язаний знати номер лінії (для багатьох пристроїв його визначення пов'язане із дослідженням конфігурації перемичок на платі). Усі драйвери у сучасних ОС автоматично визначають номер переривання.
- Драйвер може використати ці номери прямо, оскільки для деяких стандартних пристроїв номер лінії переривання документований і незмінний (або, як для паралельного порту, однозначно залежить від базової адреси відображуваної пам'яті). Але так можна робити далеко не для всіх пристроїв.
- Драйвер може виконати *зондування* (probing) пристрою. Під час зондування драйвер посилає контролеру пристрою запити на генерацію переривань, а потім перевіряє, яка з ліній переривань була активізована. Проте цей підхід є досить незручним, і його вважають застарілим.
- І, нарешті, найкращим є підхід, за якого пристрій сам «повідомляє», який номер переривання він використає. У цьому разі завданням драйвера є просте визначення цього номера, наприклад, читанням регістра статусу одного із портів введення-

виведення пристрою або спеціальної ділянки відображуваної пам'яті. Більшість сучасних пристроїв допускають таке визначення конфігурації. До них належать, наприклад, усі пристрої шини PCI (для них це визначено специфікацією, інформацію зберігають у спеціальному просторі конфігурації), а також ISA-пристрої, що підтримують специфікацію Plug and Play.

Оброблювачі переривань − це звичайні послідовності інструкцій процесора; їх можна розробляти як на асемблері, так і мовами програмування високого рівня. В оброблювачах дозволено виконувати більшість операцій за деякими винятками:

- не можна обмінюватися даними із адресним простором режиму користувача, оскільки він не виконується у контексті процесу;
- не можна виконувати жодних дій, здатних спричинити очікування (явно викликати `sleep()`, звертатися до синхронізаційних об'єктів із викликами, які можна заблокувати, резервувати пам'ять за допомогою операцій, що призводять до сторінкового переривання).

Фактично оброблювачі не можуть взаємодіяти із планувальником потоків. Основні дії оброблювача:

- повідомлення пристрою про те, що переривання оброблене (щоб той міг почати приймати нові переривання);
- читання або записування даних відповідно до специфікації оброблюваного переривання;
- поновлення потоку або кількох потоків, що очікують у черзі, пов'язаній із цим пристроєм (якщо переривання позначає подію, настання якої вони очікували, наприклад прихід нових даних).

Зауважимо, що в деяких випадках для запобігання порушенню послідовності отримання даних оброблювач має на певний час забороняти переривання – або всі, або тільки для його лінії.

Основною вимогою до оброблювачів є ефективність їхньої реалізації. Оброблювач переривання повинен завершувати свою роботу швидко, щоб переривання не залишалися заблокованими надто довго. З іншого боку, часто у відповідь на переривання необхідно виконати досить великий обсяг роботи. Два критерії (швидкість і обсяг роботи) у цьому разі конфліктують один із одним.

Сучасні ОС вирішують цю проблему через поділ коду оброблювача переривання навпіл.

Верхня половина (top half) − це безпосередньо оброблювач переривання, що виконується у відповідь на прихід сигналу відповідною лінією. Зазвичай у верхній половині здійснюють мінімально необхідну обробку (наприклад, повідомляють пристрій про те, що переривання оброблене), після чого вона планує до виконання другу частину.

Нижня половина (bottom half) не виконується негайно у відповідь на переривання, ядро планує її до виконання пізніше, у безпечніший час. Основна відмінність між виконанням обох частин полягає в тому, що під час виконання нижньої половини переривання дозволені, тому вона не впливає на обслуговування інших переривань − ті з них, які виникли після завершення верхньої половини, будуть успішно оброблені. В усьому іншому до коду нижньої половини ставлять ті самі вимоги, що й до коду оброблювачів.

Таку технологію називають *відкладеною обробкою переривань,* вона реалізована в усіх сучасних ОС. У Linux механізми реалізації коду нижньої половини, починаючи із версії 2.4, називають *тасклетами* (tasklets), у Windows XP − *відкладеними викликами процедур* (deferred procedure calls, DPC).

Обидва наведені підходи до організації введення-виведення не позбавлені недоліку: вони надто завантажують процесор. Як зазначалося, це є головною проблемою для опитування пристроїв, але введення-виведення на основі переривань теж може мати проблеми, якщо переривання виникатимуть надто часто.

Проблема полягає в тому, що процесор бере участь у кожній операції читання і записування, просто пересилаючи дані від пристрою у пам'ять і назад. Із цією задачею упорався б і простіший пристрій; зазначимо також, що розмір даних, які пересилають за одну операцію, обмежений розрядністю процесора (32 біти, для пересилання наступних 32 біт потрібна нова інструкція процесора). Якщо переривання йдуть часто (наприклад, від жорсткого диска), то час їхньої обробки може бути порівняний із часом, відпущеним для решти робіт. Бажано пересилати дані між пристроєм і пам'яттю більшими блоками і без участі процесора, а його у цей час зайняти більш продуктивними операціями.

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

Завдання ОС під час взаємодії із DMA-контролером досить складні, оскільки такі контролери відчутно відрізняються для різних архітектур і не завжди легко інтегровані із поширеними методами керування пам'яттю. Особливо багато проблем виникає у зв'язку з тим, що буфер для приймання даних від контролера має перебувати у неперервному блоці невивантажуваної фізичної пам'яті. Зазвичай драйвери розміщують буфер для пересилання даних від DMA-пристрою під час ініціалізації і вивільняють після завершення роботи системи; усі операції введення-виведення звертаються до цього буфера. Для драйвера важливо також задавання оброблювача переривання від контролера.

11.4. Підсистема введення-виведення ядра

Планування введення-виведення звичайно реалізоване як середньотермінове планування. Як відомо, з кожним пристроєм пов'язують чергу очікування, під час виконання блокувального виклику (такого як `read()` або `fcntl()`) потік поміщають у чергу для відповідного пристрою, з якої його звичайно вивільняє оброблювач переривання. Різним пристроям можуть присвоювати різні пріоритети.

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

Відмітимо причини, які викликають необхідність буферизації.

- Різниця у пропускній здатності різних пристроїв. Наприклад, якщо дані зчитують із модему, а потім зберігають на жорсткому диску, без буферизації процес переходитиме у стан очікування перед кожною операцією отримання даних від модему. Власне кажучи, буферизація потрібна у будь-якій ситуації, коли для читання даних потік має у циклі виконати блокувальну операцію введення-виведення для кожного отримуваного символу; без неї така робота буде вкрай неефективною.
- Різниця в обсязі даних, переданих пристроями або рівнями підсистеми введення-виведення за одну операцію. Типовим прикладом у цьому разі є мережний обмін даними, коли відправник розбиває велике повідомлення на фрагменти, а одержувач

у міру отримання поміщає ці фрагменти в буфер (його ще називають *буфером повторного збирання* − reassembly buffer) для того щоб зібрати з них первісне повідомлення. Такий буфер ще більш необхідний, оскільки фрагменти можуть приходити не в порядку відсилання.

- Необхідність підтримки для застосування семантики копіювання (copy semantics). Вона полягає в тому, що інформація, записана на диск процесом, має зберігатися в тому вигляді, у якому вона перебувала у пам'яті в момент записування, незалежно від змін, зроблених після цього.

Розглянемо різні ***способи реалізації буферизації*** (рис. 11.2).

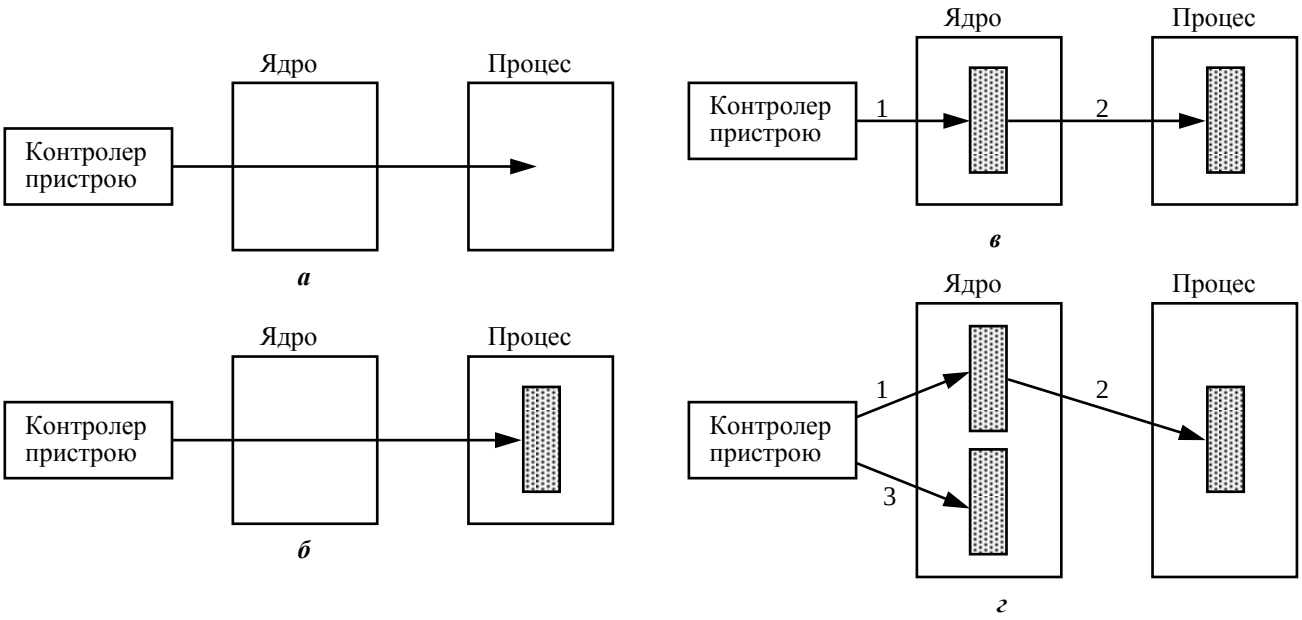

Рис.11.2. Різні способи буферизація: а – введення-виведення без буферизації; б – буферизація в просторі користувача; в – одинарна буферизація в ядрі; г – подвійна буферизація

Буфер, у який копіюються дані від пристрою, можна організувати в адресному просторі процесу користувача. Хоча такий підхід і дає виграш у продуктивності, його не можна вважати прийнятним, оскільки сторінка із таким буфером може у будь-який момент бути заміщена у пам'яті та скинута на диск. Можна дозволити процесам фіксувати свої сторінки у пам'яті під час кожної операції введення-виведення, але це призводить до невиправданих витрат основної пам'яті.

Першим підходом, який реально використовують на практиці, є одинарна буферизація в ядрі. У цьому разі у ядрі створюють буфер, куди копіюють дані в міру їхнього надходження від пристрою. Коли цей буфер заповнюється, весь його вміст за одну операцію копіюють у буфер, що перебуває у просторі користувача. Аналогічно під час виведення даних їх спочатку копіюють у буфер ядра, після чого вже ядро відповідатиме за їхнє виведення на пристрій. Це дає змогу реалізувати семантику копіювання, оскільки після копіювання даних у буфер ядра інформація із буфера користувача у підсистему введення-виведення гарантовано більше не потрапить − процес може продовжувати свою роботу і використовувати цей буфер для своїх потреб.

Використання одинарного буфера не позбавлене недоліків, головний з яких пов'язаний з тим, що в момент, коли буфер переповнений, нові дані нікуди помістити (а буфер може бути переповнений деякий час, наприклад, поки його зберігають на диску або поки йде завантаження з диска сторінки з буфером користувача). Для вирішення цієї проблеми запропонована технологія подвійної буферизації, за якої у пам'яті ядра створюють два

буфери. Коли перший з них заповнений, дані починають надходити в другий, і до моменту, коли заповниться другий, перший уже буде готовий прийняти нові дані і т. д.

Узагальнення цієї схеми на *n* буферів називають *циклічною буферизацією,* її можна використовувати тоді, коли час збереження буфера перевищує час його заповнення.

Цікавим підходом до оптимізації операцій введення-виведення, пов'язаним із записуванням великих обсягів даних протягом однієї операції, є введення-виведення з розподілом та об'єднанням (scatter-gather I/O). У даному випадку в системі дозволяється використовувати під час введення-виведення набір непов'язаних ділянок пам'яті. При цьому можливі дві дії:

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

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

В UNIX-системах (відповідно до стандарту POSIX) введення із розподілом реалізують системним викликом `readv()`, виведення з об'єднанням – `writev()`.

```
#include <sys/uio.h>
ssize_t readv(int fdl, const struct iovec *iov, int count);
ssize_t writev(int fdl, const struct iovec *iov, Int count);
```

де: `fdl` – дескриптор відкритого файла; `iov` – масив структур, які задають набір ділянок пам'яті для введення і виведення; `count` – кількість структур у масиві `iov`.

Кожний елемент масиву містить два поля: `iov_base`, що задає адресу ділянки пам'яті, та `iov_len`, що задає її довжину. Елемент масиву має бути повністю оброблений (наприклад, заповнений до довжини iov_len за виконання `readv()`) перед тим як система перейде до обробки наступного.

Виклик `readv()` повертає загальну кількість зчитаних байтів, а виклик `writev()` – записаних.

```
int bytes_read;
char buf0[1024], buf1[512];
struct iovec iov[2];
iov[0].iov_base = buf0;
iov[1].iov_base = buf1;
iov[0].iov_len = sizeof(buf1);
iov[1].iov_len = sizeof(buf2);
// infile.outfile – дескриптори відкритих файлів
do {
// читання у два буфери
bytes_read = readv(infile, iov, 2);
// записування із двох буферів
if (bytes_read > 0) writev(outfile, iov, 2);
} while (bytes_read > 0);
```

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

Спулінг (spooling) – технологія виведення даних із використанням буфера, що працює за принципом FIFO. Такий буфер називають *спулом* (spool) або ділянкою спула (spool area).

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

Роботи надходять у спул і в ньому вишиковуються у FIFO-чергу (нові роботи додаються у її хвіст). Як тільки пристрій вивільняється, роботу із голови черги передають пристрою для виведення. Звичайно спулінг пов'язаний із повільними пристроями, тому найчастіше ділянку спула організовують на жорсткому диску, а роботи відображають файлами. Для керування спулом (підтримки черги, пересилання робіт на пристрій) зазвичай використовують фоновий процес або потік ядра. Має бути доступний інтерфейс керування спулом, за допомогою якого можна переглядати вміст черги, вилучати роботи з неї, міняти їхній порядок, тимчасово призупиняти виведення (наприклад, на час обслуговування пристрою).

Зазначимо, що загалом спулінг не можна назвати технологією керування введенням-виведенням режиму ядра − його часто реалізують процеси користувача (такі як демон друкування `lpr` для UNIX).

Менш гнучкою альтернативою спулінгу є надання можливості монопольного захоплення пристрою потоками. Такий монопольний режим може, наприклад, задаватися під час відкриття пристрою, після чого інші потоки не зможуть отримати доступу до нього, поки цей режим не буде знято. Такий підхід вимагає реалізації синхронізації доступу до пристрою. У більшості випадків спулінг є гнучкішим і надійнішим вирішенням, але для таких пристроїв, як записувальний CD-дисковід або сканер, монопольне захоплення є прийнятним варіантом організації доступу.

У підсистемі введення-виведення під час роботи виникають різні ***помилки***, які можна віднести до кількох категорій.

- Помилки в програмному коді введення-виведення (доступ до відсутнього пристрою, недопустимі дії із пристроєм тощо). Реакцією на такі помилки звичайно є повернення коду помилки в застосування.
- Помилки, викликані апаратними проблемами. Серед них розрізняють:
  - викликані тимчасовими причинами (високе навантаження на мережу, сигнал «зайнято» для модему); для цих помилок звичайною реакцією є повторна спроба виконання введення-виведення;
  - що вимагають втручання користувача (відсутність паперу у принтері); за такої помилки зазвичай потрібно попросити користувача виконати певні дії;
  - викликані некоректною роботою апаратного забезпечення (збій контролера, дефектні сектори на диску); у цьому разі важливим є надання користувачу (який може виявитися представником служби технічної підтримки фірми-розробника пристрою) якомога більше повної інформації про помилку (зазначимо, що після деяких таких помилок продовження роботи системи може стати неможливим).

У програмному забезпеченні передача докладних відомостей про помилку є головним завданням підсистеми введення-виведення у разі виникнення проблеми. Справді, для більшості системних викликів інтерфейс повідомлень про помилки доволі скромний (цілочисловий код повернення, глобальна змінна `errno`), тому розробникам драйверів необхідно передбачати додаткові способи отримання інформації про помилку.

15.5. Введення-виведення у режимі користувача

У більшості випадків введення-виведення на рівні апаратного забезпечення кероване перериваннями, а отже є асинхронним. Однак використати асинхронну обробку даних завжди складніше, ніж синхронну, тому найчастіше введення-виведення в ОС реалізоване у вигляді набору *блокувальних* або *синхронних* системних викликів, подібних до `read()`, `write()` або `fcntl()`. Під час виконання такого виклику поточний потік призупиняють, переміщуючи в чергу очікування для цього пристрою. Після завершення операції введення-виведення і отримання всіх даних від пристрою потік переходить у стан готовності та може продовжити своє виконання.

Однак синхронне введення-виведення підходить не для всіх застосувань. Зокрема, воно ***не підходить*** для таких категорій програм:

- серверів, що обслуговують багатьох клієнтів (отримавши з'єднання від одного клієнта, потрібно мати можливість відразу обслуговувати й інших);
- застосувань, що працюють із журналом (після виклику функції записування в журнал потрібно продовжити виконання негайно, не очікуючи завершення виведення);
- мультимедійних застосувань (відіславши запит на читання одного кадру, потрібно одночасно показувати інші).

Для вирішення цієї проблеми запропоновано кілька підходів.

Принципи, що лежать в основі ***першого із можливих підходів*** до розв'язання проблем синхронного введення-виведення полягають в тому, що за необхідності виконання асинхронного введення-виведення у застосуванні створюють новий потік, у якому виконуватиметься звичайне, синхронне введення-виведення. При блокуванні цього потоку вихідний потік продовжуватиме своє виконання.

Такий підхід має багато переваг і може бути рекомендований для використання у багатьох видах застосувань. Наведемо приклад розробки багатопотокового сервера за принципом «потік для запиту» (thread per request).

У таких серверах є головний потік, який очікує безпосередніх запитів клієнта на отримання даних (виконуючи синхронну операцію `read()` або `recvfrom()` для сокетів). Після отримання кожного запиту головний потік створює новий робочий потік для обробки його запиту, після чого продовжує очікувати подальших запитів. Робочий потік обробляє запит і завершується. Такий підхід застосовують для зв'язку без збереження стану, коли обробка одного запиту не залежить від обробки іншого. Ось псевдокод сервера, що працює за таким принципом:

```
void concurrent_server() {
for (; ;) {
   read (fd, &request); // синхронно очікувати запит
   // створити потік для обробки запиту
   create_thread (worker_thread, request);
   }
}
// функція потоку обробки запиту
void worker_thread(request_t request) {
   process_request(request); // обробити запит
}
```

Переваги такого підходу полягають у простоті реалізації та низьких вимогах до ресурсів, недоліки – у недостатній масштабованості (за великої кількості одночасних

запитів витрати на створення потоків для кожного із них можуть спричиняти зменшення продуктивності).

У зв'язку з використанням багатопотоковості для організації асинхронного введення-виведення виникають ще деякі проблеми.

- Не завжди варто додавати багатопотоковість в однопотокове застосування тільки тому, що в ньому знадобилося виконати асинхронне введення-виведення.
- Потрібно реалізовувати синхронізацію потоків.
- Може знизитися надійність застосування.
- Кваліфікація програмістів може виявитися недостатньою для реалізації багатопотоковості.

Усе це призводить до значного поширення технологій, альтернативних до цього підходу:

- введення-виведення із повідомленням;
- введення-виведення із повідомленням про події;
- введенням-виведенням із перекриттям;
- порт завершення введення-виведення.

Першою технологією, яку можна використати для організації введення-виведення без блокування і яка не вимагає організації багатопотоковості, є ***введення-виведення із повідомленням*** (notification-driven I/O). Ця технологія має й інші назви, наприклад мультиплексування введення-виведення (I/O multiplexing).

Якщо потрібно в циклі виконати блокувальний виклик (наприклад, `read()`) для кількох файлових дескрипторів, може трапитися так, що один із викликів заблокує поточний потік у той момент, коли на дескрипторі, який використовується в іншому виклику, з'являться дані. Доцільно організувати одночасне очікування отримання даних із кількох дескрипторів. Це і є основним мотивом розробки даної категорії засобів введення-виведення.

У цьому разі виконання введення-виведення поділяють на кілька етапів.

1. Спеціальний системний виклик (виклик повідомлення) визначає, чи можна виконати синхронне введення-виведення хоча б для одного дескриптора із заданого набору без блокування потоку. У POSIX визначено виклики повідомлення `poll()` і `select()`.
2. Як тільки хоча б один дескриптор із набору стає готовий до введення-виведення без блокування, виклик повідомлення повертає керування; при цьому поточний потік може визначити, для яких саме дескрипторів може бути виконане введення-виведення або які з них змінили свій стан (тобто отримати повідомлення про стан дескрипторів).
3. Потік, що викликає, може тепер у циклі обійти всі дескриптори, визначені внаслідок повідомлення на етапі 2, і виконати введення-виведення для кожного з них, блокування поточного потоку ця операція в загальному випадку не спричинить.

Проілюструємо застосування цієї технології на прикладі реалізації *реактивного сервера*. Такий сервер відповідає на запити клієнтів в одному потоці виконанням у циклі опитування стану набору дескрипторів, визначення тих із них, на які прийшли запити, та подальшого виконання цих запитів.

Спочатку реалізуємо реактивний сервер на основі технології *введення-виведення із повідомленням про стан дескрипторів* (state-based notification). Це традиційний підхід, який використовується багато років. Необхідно виконати такі кроки.

1. Підготувати структуру даних (назвемо її `fdarr`) з описом усіх дескрипторів, стан яких потрібно відстежувати.

2. Передати `fdarr` у системний виклик повідомлення (у POSIX до таких викликів належать уже згадані `select()` і `poll()`). Після виходу із виклику повідомлення `fdarr` міститиме інформацію про стан усіх відстежуваних дескрипторів (які з них готові до виконання введення-виведення без блокування, а які − ні).
3. Для дослідження результатів повідомлення обійти в циклі всі елементи `fdarr` і для кожного із них визначити готовність відповідного дескриптора; якщо він готовий − виконати для нього введення-виведення.

Ось псевдокод реактивного сервера із використанням повідомлення про стан дескрипторів:

```
void reactive_server () {
// цикл опитування набору дескрипторів fdarr, підготовленого раніше
for ( ; ; ) {
select (&fdarr);          // прослуховування набору
  // цикл визначення активних дескрипторів
  for (i=0; i <= count(fdarr); i++) {
    if ( request_is_ready (fdarr[i]) ) { // якщо був запит
    read (fdarr[i], &request); // одержати дані запиту
    process_request (request); // обслужити клієнта
    }
  }
}
```

Незважаючи на те що такий сервер використовує всього один потік, продуктивність його роботи може бути високою, якщо запити обслуговують достатньо швидко. Сервер працює так:

- `select()` повертає інформацію про ті запити, які потрібно обслужити;
- усі ці запити обслуговують один за одним без затримок на введення-виведення; поки їх обслуговують, надходять нові;
- коли всі запити, про які сповістив минулий виклик `select()`, обслужені, починають нову ітерацію зовнішнього циклу, `select()` викликають знову, і він негайно повертає відомості про всі запити, які надійшли із часу минулого виклику.

Отже, запити потрапляють на обробку групами тим більшими, чим більше приходить запитів. Немає очікування ні під час виклику `select()`, ні під час читання даних із дескриптора. Усе це значно підвищує продуктивність.

Зазначимо, що в циклі обходу `fdarr` було обстежено готовність всіх його елементів до введення-виведення. Виникає запитання: чи завжди обов'язково проводити вичерпне обстеження, чи є можливість отримувати тільки інформацію про готові дескриптори? На жаль, для цього підходу такої можливості немає.

Основною особливістю введення-виведення із повідомленням про стан дескрипторів є те, що в разі його використання не зберігається стан. Кожен виклик повідомлення вимагає передавання всього набору дескрипторів і повертає «миттєвий знімок» стану цих дескрипторів. Це потребує повного обходу цього списку як всередині виклику повідомлення, так і в коді, що його викликав. Такий обхід може серйозно позначитися на продуктивності у разі великої кількості дескрипторів. Таким чином, інформація про неактивні дескриптори даремно копіюватиметься у ядро і назад під час кожного виклику повідомлення.

Для того щоб підвищити ефективність цієї схеми, запропоновано інший підхід – ***введення-виведення із повідомленням про події*** (event-based notification).

Основною відмінністю цього підходу є збереження у ядрі інформації про набір

дескрипторів, зміна стану яких становить інтерес. Унаслідок цього з'являється можливість повертати інформацію про стан не всіх дескрипторів, а тільки про ті з них, які перейшли у стан готовності з моменту останнього виклику функції повідомлення (тобто, про всі події зміни стану).

Найвідомішими є приклади реалізації такого підходу для FreeBSD (`kqueue`) і для Linux 2.6 (`epoll`).

Виконання введення-виведення в цьому разі зводиться до таких кроків.

1. Спеціальний системний виклик (у Linux – `epoll_create()`) створює структуру даних у ядрі; зазвичай як параметр у такий виклик передають максимальну кількість дескрипторів, які потрібно контролювати. Таку структуру даних називають прослуховувальним об'єктом.
2. Після створення такого об'єкта для нього потрібно сформувати набір контрольованих дескрипторів, для кожного з них вказують події, які цікавлять потік, що виконував виклик. У Linux це робиться окремим викликом `epoll_ctl()`, один із можливих варіантів виконання якого додає дескриптор у набір.
3. Виклик повідомлення (у Linux – `epoll_wait()`) при цьому повертає інформацію тільки про ті дескриптори, які змінили стан із моменту останнього виклику (а не про всі дескриптори, як для `select()` або `poll()`).

Зазначимо також, що використання внутрішньої структури даних позбавляє необхідності обходу всіх дескрипторів усередині `epoll_wait()`.

Асинхронне введення-виведення реалізоване у деяких UNIX-системах (є стандарт POSIX для таких операцій). Одна з найповніших реалізацій цієї технології доступна також в системах лінії Windows XP, де її називають *введенням-виведенням із перекриттям* (overlapped I/O). Основна ідея тут полягає в тому, що потік, який почав виконувати введення-виведення, не блокують до його завершення. Асинхронне введення-виведення зводиться до виконання таких дій.

- Потік виконує системний виклик асинхронного введення або виведення, який ставить операцію введення-виведення в чергу і негайно повертає керування.
- Потік продовжує виконання паралельно з операцією введення-виведення.
- Коли операція введення-виведення завершується, потік отримує про це повідомлення.

Операція може бути перервана до свого завершення.

Основним підходом до отримання повідомлення про завершення асинхронного введення-виведення є виконання операції очікування завершення введення-виведення. При цьому потік призупиняють до завершення асинхронної операції.

Після продовження виконання потоку можна отримати інформацію про результат виконання операції. Зазначимо, що тільки на цей час буфер, покажчик на який було передано в операцію асинхронного введення, заповниться зчитаними даними.

Можливий ще один підхід, коли в операцію асинхронного введення-виведення передають адресу процедури зворотного виклику, яка автоматично викликається після завершення виконання операції. Цей підхід складніший у використанні, тому його застосовують рідше.

У разі асинхронного введення-виведення не гарантовано порядок виконання операцій. Якщо, наприклад, потік виконає підряд асинхронні операції читання і записування, немає гарантії, що ОС виконає їх саме в цьому порядку; цілком можливо, що спочатку виконається записування, а потім − читання.

Стандарт POSIX передбачає для виконання асинхронного введення і виведення

відповідно виклики `aio_read()` і `aio_write()`, для очікування – `aio_suspend()`, для переривання – `aio_cancel()`, а для отримання результату – `aio_return()`. Ці функції (крім `aio_suspend()`) параметром приймають покажчик на структуру `aiocb` із полями:

- `aio_fildes` – дескриптор файла, для якого здійснено введення-виведення;
- `aio_buf` – покажчик на буфер, у який зчитає дані `aio_read()` і з якого запише дані `aio_write`();
- `aio_nbytes` – розмір буфера.

Функція `aio_suspend()` має такий вигляд:

```
int aio_suspend(struct aiocb *list[], int cnt, struct timespec *tout);
```

де: `list` – масив покажчиків на структури aiocb, можливо, пов'язані із різними операціями введення-виведення (так здійснюють очікування завершення відразу кількох асинхронних операцій); `cnt` – кількість елементів у масиві `list`; `tout` – структура, що задає максимальний час очікування (NULL – очікувати нескінченно довго).

Проміжний результат виконання операції можна дістати за допомогою функції `aio_error()`, що повертає значення EINPROGRESS, якщо операція ще не була завершена, і нуль – якщо вона завершилась успішно. У цьому разі результат операції (обсяг зчитаних даних для `aio_read()` і записаних – для `aio_write()`) може бути визначений за допомогою виклику функції `aio_return()`.

У Win32 API застосовується трохи інший підхід: для асинхронного введення-виведення можна використати стандартні функції файлового введення і виведення `ReadFile()` і `WriteFile()`, якщо в них передається останнім параметром покажчик на спеціальну структуру типу OVERLAPPED. При цьому файл має бути відкритий із увімкнутим прапорцем, що дозволяє асинхронні операції. Для очікування завершення введення-виведення використовують універсальну функцію очікування, наприклад `WaitForSingleObject()`. У найпростішому випадку вона має очікувати на файловому дескрипторі, для якого було виконане введення або виведення. Для отримання результату треба використати функцію `GetOverlappedResult()`, для переривання введення-виведення – `CancelIo()`.

Остання із розглянутих нами технологій – ***порт завершення введення-виведення*** (I/O completion port) – поєднує багатопотоковість із асинхронним введенням-виведенням для вирішення проблем розробки серверів, що обслуговують велику кількість одночасних запитів.

Для кращого розуміння передумов появи цієї технології повернімося до моделі багатопотокового сервера «потік для запиту». Як відомо, це вирішення є не досить добре масштабованим через витрати на організацію паралельного виконання великої кількості потоків. Загалом небажано допускати, щоб кількість потоків у системі, які перебувають у стані виконання, набагато перевищувала кількість процесорів.

З іншого боку, запропоновані дотепер альтернативні однопотокові підходи (введення-виведення із повідомленням і асинхронне введення-виведення) не можуть використати переваги багатопроцесорних архітектур. Хотілося б досягти деякого компромісу між великою кількістю потоків і необхідністю використовувати багатопроцесорність.

Одним із варіантів такого компромісу є організація *пула потоків* (thread pool). При цьому в момент запуску серверного застосування заздалегідь створюють набір потоків (пул), кожен із яких готовий обслуговувати запити. Коли приходить запит, перевіряють,

чи є в пулі вільні потоки, якщо є, з нього вибирають потік, який починає обслуговувати запит. Після виконання запиту потік повертають у пул. Коли з появою нового запиту вільних потоків у пулі немає, запит поміщають у чергу, і він там очікує, поки не вивільниться потік, що може його обслужити. При цьому можна керувати розміром пула, досягаючи того, щоб кількість активних потоків збігалася із кількістю процесорів у системі.

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

Спочатку створюють новий об'єкт порту завершення викликом `CreateIoCompletionPort()`. При цьому задають максимальну кількість потоків, пов'язаних із цим портом, які можуть одночасно виконуватися у системі. Позначимо цю величину R_{max}. Рекомендують задавати R_{max} рівним кількості процесорів у системі (якщо задати його рівним нулю, система сама надасть йому цього значення).

```
ph=CreateIoCompletionPort (INVALID_HANDLE_VALUE, 0, 0, Rmax);
```

Після цього із портом пов'язують файлові дескриптори, відкриті для асинхронного введення-виведення. Потоки очікуватимуть його завершення на цих дескрипторах. Для додавання дескриптора в порт також використовують виклик `CreateIoCompletionPort()`, при цьому як параметри в нього повинні бути передані наявний дескриптор порту, файловий дескриптор і унікальний ключ для його однозначного визначення.

```
// додаємо дескриптор з масиву fdarr у порт
CreateIoCompletionPort (fdarr[key], ph, key, Rmax);
```

Після створення порту формують набір (пул) робочих потоків, які обслуговуватимуть запити. Кількість таких потоків має перевищувати R_{max}. Усі потоки пула повинні виконувати один і той самий код, що має приблизно такий вигляд:

```
for ( ; ; ) {
// очікування на об'єкті порту, заданому дескриптором ph
GetQueuedCompletionStatus (ph, nbytes, &key, &ov, INFINITE);
// тут потік є активним
ReadFile (fdarr[key], request, ...):  // прочитати запит
process_request (request);          // обслужити клієнта
}
```

Тут виклик `GetQueuedCompletionStatus()` означає опитування порту завершення; саме на ньому очікуватимуть всі потоки пула. Потік стане активним, коли ця функція поверне керування. При цьому вона поверне ключ дескриптора key, заданий під час його додавання. Цей ключ ідентифікує дескриптор, у який прийшов пакет завершення (у прикладі видно використання ключа як індексу у масиві дескрипторів). Активний потік виконуватиме код обробки запиту клієнта.

Клієнти працюють із файловими дескрипторами, пов'язаними із портом, виконуючи для них асинхронне введення-виведення. Завершення його не очікують − за це відповідає порт.

Розглянемо основні ***етапи роботи порту завершення***.

1. Якщо на жоден дескриптор не прийшло повідомлення про завершення асинхронного введення-виведення, усі потоки очікують на виклик `GetQueuedCompletionStatus()`, перебуваючи у черзі заблокованих потоків. Кажуть, що ці потоки заблоковані на порту завершення. Черга заблокованих потоків фактично і є пулом потоків.
2. Після того як асинхронне введення-виведення для одного із дескрипторів, пов'язаних із портом, завершується (іншими словами, у порт приходить пакет завершення), система перевіряє, скільки активних потоків зараз пов'язані з цим

портом (назвемо таку величину R_{cur}).

- за $R_{cur} < R_{max}$ один із потоків пула, що очікують на порту завершення, поновлюється і починає обслуговувати запит, стаючи активним потоком; при цьому *RCU*, збільшують на одиницю;
- за $R_{cur} > R_{max}$ пакет завершення стає у чергу, де й перебуватиме, поки R_{cur} не стане меншим за R_{max} (що може статися на кроці 3).

3. Коли потік завершує обслуговування запиту, його повертають у пул, блокуючи на порту. Значення R_{cur} зменшують на одиницю і за наявності пакетів, що очікують завершення, один із них починають обробляти. Зазначимо, що пул потоків побудований за принципом LIFO (стека), тобто потік, що починає обробляти пакет завершення, буде потоком, призупиненим останнім (якщо в пул не прийшли нові потоки із початку виконання цього кроку, це буде потік, який щойно повернувся у пул). Це зроблено для оптимізації за малої кількості запитів, оскільки тоді частина потоків взагалі не отримуватиме керування, і їхні ресурси система може вивантажити на диск.
4. У разі заблокування потоку на об'єкті синхронізації під час обслуговування запиту або під час виконання синхронної операції введення-виведення його переміщують у чергу заблокованих потоків, R_{cur} зменшують на одиницю, і пакет, що очікує завершення, починають обробляти аналогічно до кроку 3. Таким чином R_{cur} для порту увесь час підтримують на рівні R_{max}.

Є можливість явно поміщати пакети завершення у порт за допомогою виклику `PostQueuedCompletionStatus()`. У такий спосіб основний потік застосування може надсилати різні повідомлення робочим потокам.

11.6. Таймери і системний час

Таймери керують пристроями, які передають у систему інформацію про час. Вони відстежують поточний час доби, здійснюють облік витрат процесорного часу, повідомляють процеси про події, що відбуваються через певний проміжок часу тощо. Робота із такими пристроями відрізняється від традиційної моделі введення-виведення, для них використовують окремий набір системних викликів.

Апаратний таймер − це пристрій, що генерує переривання таймера через певний проміжок часу. Розглянемо, як такий пристрій можна використати для відстеження поточного системного часу.

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

- 32-бітне значення не може зберігати достатньо великий проміжок часу (переповнення такого лічильника за частоти переривання таймера 60 Гц настане упродовж двох років);
- 64-бітне значення на 32-бітному процесорі (наприклад, в архітектурі IA-32) оброблятиметься неефективно.

Для реалізації 32-бітного лічильника звичайно використовують такі підходи.

- Зберігають лише інформацію про секунди, а про долі поточної секунди (мілісекунди, мікросекунди) − окремо. У цьому разі лічильника секунд вистачить для зберігання інформації про 2^{32} с (більш як на 135 років).
- Зберігають інформацію про кількість переривань із моменту останнього завантаження системи, а час останнього завантаження зберігають окремо (як 64-бітне значення). У разі запиту поточного часу значення лічильника і збережений час

завантаження додають.

Якщо поряд із таймером у системі є годинник, значення цього лічильника може час від часу звірятися із показаннями годинника.

Визначення системного часу в Linux

Для визначення системного часу в Linux використовують системний виклик `gettimeofday()`. Зазначимо, що формат часу, який повертає цей виклик, не дуже зручний для використання у застосуваннях (структура `time-val` з полями `tv_sec` – секунди, що пройшли з 1.01.1970, і `tv_usec` – мікросекунди в поточній секунді).

Для перетворення часу у зручний формат використовують функції стандартної бібліотеки мови С, зокрема функцію `localtime()`. Вона приймає покажчик на поле tv_sec структури `timeval` і повертає покажчик на структуру `tm`, поля якої відповідають елементам системного часу: `tm_year` (рік з 1900), `tm_mon` (місяць від нуля) і т. д. до `tm_sec` (секунди).

Розглянемо приклад відображення поточного системного часу у зручному для сприйняття форматі:

```
#include <sys/time.h>
#include <time.h>
struct timeval tv; struct tm *ptm;
gettimeofday(&tv, NULL);
ptm = localtime(&tv, tv_sec);
printf("Зараз %02d/%02d/%d %02d:%02d\n",
          ptm->tm_mday, ptm->tm_mon+1,
          ptm->tm_year+1900, ptm->tm_hour, ptm->tm_min);
```

Для того щоб дізнатися про ***поточний системний час*** у Windows XP, можна використати функцію `GetSystemTime()`:

```
VOID GetSystemTime(LPSYSTEMTIME time);
```

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

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

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

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

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

Аналогічні таймери використовують у ядрі для керування деякими апаратними

пристроями. Наприклад, дисковід CD дисків не можна використати відразу після ввімкнення двигуна, йому потрібен час для розгону. Для розв'язання цього завдання драйвер диска встановлює таймер після включення двигуна так, щоб він спрацював через час, необхідний для розгону. Після спрацювання такого сторожового таймера (watchdog timer) вважають, що дисковід готовий до роботи.

Для забезпечення відкладеного і періодичного виконання коду в Linux використовують інтервальні таймери. Вони керовані системним викликом `setitimer()` і дають змогу спланувати доставлення сигналу процесу на заданий час у майбутньому.

```
#include <sys/time.h>
int setitimer(int type, const struct itimerval *ptimer,
     struct itimerval *old);
```

де: `type` – визначає вид таймера (ITIMER_REAL – відлічує системний час і відсилає сигнал SIGALRM, ITIMER_VIRTUAL – відлічує час виконання процесу в режимі користувача і відсилає сигнал SIGVTALRM, ITIMER_PROF – відлічує час виконання процесу в усіх режимах і відсилає сигнал SIGPROF);

`ptimer` – покажчик на структуру `itimerval`, що задає параметри таймера (із полями `it_value` – час, через який буде відіслано сигнал, і `it_interval` – період відсилання сигналу; обидва ці поля є структурами `timeval`).

Наведемо приклад використання інтервального таймеру в Linux.

```
struct itimerval mytimer = { 0 };
mytimer.it_value.tv_usec = 1000000; //початок роботи – через 1 сек.
mytimer.it_interval.tv_usec = 500000; // період – 0.5 сек.
// початок виконання таймера
setitimer(ITIMER_REAL, &mytimer, NULL);
// тут процес буде одержувати сигнал SIGALRM через заданий час
pause();
```

Є спрощений варіант інтерфейсу інтервального таймера – системний виклик `alarm()`, що змушує систему надіслати процесу сигнал SIGALRM через задану кількість секунд.

```
#include <unistd.h>
alarm(600);     // сигнал надійде через 10 хвилин
```

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

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

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

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

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

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

```
BOOL SetWaitableTimer(HANDLE ht, const LARGE_INTEGER *endtime,
```

```
        LONG period, PTIMERAPCROUTINE pfun, LPVOID pfun_arg,
        BOOL resume);
```

де: `ht` – дескриптор таймера;

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

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

`pfun` – функція користувача, яку викликатимуть у разі спрацювання таймера.

Наведемо приклад використання таймеру очікування у Windows XP.

```
LARGE_INTEGER endtime;
endtime.QuadPart = -10000000;   // сигналізація через 1 сек.
HANDLE ht=CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(ht, Sendtime, 0, NULL, NULL, TRUE); // запуск
WaitForSingleObject(ht, INFINITE); // очікування сигналізації
// ... код. виконуваний у разі спрацювання таймера
```

11.7. Керування введенням-виведенням: UNIX і Linux

Для організації уніфікованого доступу до пристроїв введення-виведення важливо вибрати спосіб звертання до них. Розглянемо прийнятий в UNIX-системах підхід *інтерфейсу файлової системи,* за якого пристрої відображаються спеціальними файлами.

У такому разі кожному драйверу пристрою відповідає один або кілька спеціальних файлів пристроїв. Такі файли за традицією поміщаються в каталог `/dev`, хоча ця вимога не є обов'язковою.

Кожний спеціальний файл пристрою характеризується чотирма атрибутами.

1. *Ім'я файла* використовують для доступу до пристрою із процесів користувача за допомогою файлових операцій. Прикладами імен файлів пристроїв можуть бути `/dev/tty0` (перший термінал), `/dev/null` (спеціальний «порожній» пристрій, все виведення на який зникає), `/dev/hda` (перший жорсткий диск), `/dev/hda1` (перший розділ на цьому диску), `/dev/cdO` (дисковід гнучкого диску), `/dev/mouse` (миша) тощо.
2. *Тип пристрою* дає змогу розрізняти блокові та символьні пристрої. Для символьних тип позначають як 'c', для блокових – як ' b'.
3. *Номер драйвера* (major number) – це ціле число (зазвичай займає 1 байт, хоча може й 2 байти), що разом із типом пристрою однозначно визначає драйвер, який обслуговує цей пристрій. Ядро системи використовує номер драйвера для визначення того, якому драйверу передати керування в разі доступу до відповідного файла пристрою. Зазначимо, що драйвери блокових і символьних пристроїв нумеруються окремо.
4. *Номер пристрою* (minor number) – ціле число, що характеризує конкретний пристрій, для доступу до якого використовують файл. Цей номер передають безпосередньо драйверу під час виконання кожної операції доступу до файла, на його підставі драйвер визначає, який код йому потрібно виконувати.

Розглянемо, як відбувається звертання до драйверів через спеціальні файли. Насамперед, кожен драйвер під час своєї реєстрації у ядрі вказує, який номер драйвера він використовуватиме. Крім того, у коді драйвера мають бути реалізовані файлові операції драйвера. Кожна з них – це реакція на виконання із файлом пристрою стандартних файлових операцій (системних викликів `open()`, `read()`, `write()`,

`lseek()` тощо). У коді кожної операції можна виконати відповідні дії над пристроєм (туди передають номер пристрою, на підставі якого й відбувається вибір пристрою в коді драйвера).

У системі зберігають дві таблиці драйверів: одна − для символьних, інша − для блокових пристроїв. Кожна така таблиця − це масив елементів, проіндексований за номером драйвера. Елементами таблиць драйверів є структури даних, полями кожної з них є покажчики на реалізації файлових операцій відповідного драйвера.

Тип пристрою, номер драйвера і номер пристрою зберігають в індексному дескрипторі відповідного файла. Під час виконання системного виклику для файла пристрою ядро ОС виконує такі дії:

- звертається до індексного дескриптора файла пристрою;
- отримує звідти тип пристрою, номер драйвера і номер пристрою;
- за типом пристрою вибирає потрібну таблицю драйверів;
- за номером драйвера знаходить відповідний елемент таблиці;
- викликає реалізацію файлової операції для драйвера, що відповідає цьому системному виклику, і передає в неї номер пристрою.

Драйвер визначає пристрій за його номером та виконує із ним відповідні дії.

Для створення файлів пристроїв у UNIX використовують утиліту `mknod`, у виклику якої потрібно задати всі чотири характеристики файла:

```
$ mknod /dev/mydevice c 150 1
```

Так створюють файл символьного пристрою, який обслуговуватиме драйвер із номером 150 і передаватиме у його функції номер пристрою 1.

Зазначимо, що файли пристроїв зберігають на диску як звичайні файли, які в будь-який момент можуть бути створені та вилучені. У разі вилучення файла пристрою вилучають лише засіб доступу до драйвера, на сам драйвер це жодним чином не впливає. Якщо згодом файл пристрою створити заново, через нього можна буде знову працювати із пристроєм, звертаючись до драйвера.

Використання для доступу до драйверів інтерфейсу файлової системи дає змогу легко забезпечити захист пристроїв від несанкціонованого доступу − для цього потрібно просто задати для файлів пристроїв відповідні права.

Стандартні файлові операції часто не вичерпують усіх дій, які можна робити із пристроями. Для того щоб не доводилося вводити нові системні виклики для додаткових дій, більшість ОС реалізують «*таємний хід*» − спеціальний системний виклик, що дає змогу передавати драйверу будь-які команди та обмінюватися інформацією в довільному форматі.

Драйвер має реалізувати функцію реакції на такий виклик так само, як реакцію на стандартні файлові виклики. Елемент таблиці драйверів містить поле, призначене для зберігання покажчика на цю функцію.

В UNIX-системах такий виклик називають `ioctl()`.

```
#include <sys/ioctl.h>
int ioctl(int d, int request[, char *argp]);
```

де: `d` − файловий дескриптор, що відповідає відкритому файлові пристрою; `request` − ціле число, що задає команду драйвера; `argp` − покажчик на довільну пам'ять, за допомогою якого застосування і драйвер можуть обмінюватися даними будь-якої природи.

Наведемо приклад алгоритму обробки запиту введення-виведення в UNIX (не

враховуючи особливості функціонування пристрою).

1. Процес користувача виконує системний виклик `read()` для спеціального файла пристрою, перед цим підготувавши буфер у своєму адресному просторі та передавши його адресу в цей системний виклик.
2. Відбувається перехід у привілейований режим для виконання коду драйвера пристрою. Як відомо, кожному процесові у привілейованому режимі доступна вся область даних ядра, тому цей перехід не спричиняє перемикання контексту − система продовжує виконувати той самий процес.
3. На підставі інформації, що зберігається в індексному дескрипторі спеціального файла, викликається функція, зареєстрована як реалізація файлової операції `read()` для відповідного драйвера.
4. Ця функція виконує необхідні підготовчі операції (наприклад, розміщує буфер у пам'яті ядра для збереження даних, отриманих від пристрою), відсилає контролеру пристрою запит на виконання операції читання та переходить у режим очікування (призупиняючи цим процес, що виконав операцію читання); для цього зазвичай використовують функцію `sleep_on()`.
5. Контролер читає дані із пристрою, можливо, заповнюючи буфер, наданий реалізацією реакції на `read()`. Коли читання завершене, він генерує переривання.
6. Апаратне забезпечення активізує верхню половину оброблювача переривання. Код верхньої половини ставить у чергу на виконання код нижньої половини.
7. Код нижньої половини заповнює даними буфер, якщо він не був заповнений контролером, виконує інші необхідні дії для завершення операції введення і поновлює виконання процесу, що очікує.
8. Після поновлення керування знову потрапляє в код реакції на `read()` − цього разу у фрагмент, що слідує за викликом функції `sleep_on()`. Цей код копіює дані із буфера ядра у буфер режиму користувача, після чого виконання системного виклику завершується.
9. Керування повертають у процес користувача.

15.8. Керування введенням-виведенням: Windows XP

Базовим компонентом підсистеми введення-виведення Windows XP є менеджер введення-виведення (I/O Manager). Зупинимося на тому, як він розв'язує одну із найважливіших задач − передає дані між рівнями підсистеми.

Обмін даними між рівнями підсистеми введення-виведення є асинхронним. Більшу частину таких даних подано у вигляді пакетів, які передають від одного компонента підсистеми до іншого, можливо, змінюючи на ходу. Кажуть, що ця підсистема Windows XP є *керованою пакетами* (packet driven). Такі пакети називають *пакетами запитів введення-виведення* (I/O Request Packet, IRP), для стислості називатимемо їх *пакетами IRP*.

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

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

Такі драйвери мають відповідати вимогам стандарту, який називають *Windows*

Driver Model (WDM). Його розроблено для драйверів, використовуваних у лінії Windows XP та останніх версіях Consumer Windows (Windows 98/Me). Звичайно для переносу таких драйверів між системами достатньо їх перекомпілювати, а деякі з них сумісні на рівні двійкового коду. Розрізняють три типи WDM-драйверів.

- *Драйвери шини* (bus drivers) керують логічною або фізичною шиною (наприклад, PCI, USB, ISA). Такий драйвер відповідає за виявлення пристроїв, з'єднаних із певною шиною.
- *Функціональні драйвери* (function drivers) керують пристроєм конкретного типу. Драйвери шини надають пристрої функціональним драйверам. Звичайно тільки функціональний драйвер працює з апаратним забезпеченням пристрою, саме він дає змогу системі використати пристрій.
- *Драйвери-фільтри* (filter drivers) доповнюють або змінюють поведінку інших драйверів.

Насправді жоден драйвер, відповідно до стандарту WDM, не може цілковито відповідати за керування пристроєм, усі вони доповнюють один одного.

Крім WDM-драйверів, у Windows XP підтримують такі категорії драйверів ядра: *файлових систем,* відповідальні за перетворення запитів введення-виведення, що використовують файли, у запити до низькорівневих драйверів пристроїв (наприклад, драйвера жорсткого диска); *відображення* (display drivers) підсистеми Win32, які перетворюють незалежні від пристрою запити GDI-підсистеми в команди графічного адаптера або у прямі операції записування у відеопам'ять; *успадковані,* розроблені для Windows NT.

На доповнення до драйверів ядра Windows XP підтримує драйвери режиму користувача. До них, зокрема, належать драйвери принтерів, які перетворюють незалежні від пристрою запити GDI-підсистеми в команди відповідного принтера і передають ці команди WDM-драйверу (наприклад, драйверу паралельного порту або універсальному драйверу USB-принтера).

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

- *Клас-драйвери* (class drivers). Реалізують інтерфейс обробки запитів введення-виведення, специфічних для конкретного класу пристроїв, наприклад драйвери дисків або пристроїв CD-ROM.
- *Порт-драйвери* (port drivers). Реалізують інтерфейс обробки запитів введення-виведення, специфічних для певного класу портів введення-виведення; зокрема до цієї категорії належить драйвер підтримки SCSI.
- *Мініпорт-драйвери* (miniport drivers). Керують реальними пристроями (наприклад, SCSI-адаптерами конкретного типу) і реалізують інтерфейс, наданий клас-драйверами і порт-драйверами.

Розглянемо структуру драйвера пристрою. Вона багато в чому подібна до структури, прийнятої в Linux. Можна виділити основні процедури драйвера.

- *Процедура ініціалізації.* Звичайно називається DriverEntry, її виконує менеджер введення-виведення під час завантаження драйвера у систему, і зазвичай вона здійснює глобальну ініціалізацію структур даних драйвера.
- *Процедура додавання пристрою* (add-device routine). Вона має бути реалізована будь-яким драйвером, що підтримує специфікацію Plug and Play. Менеджер Plug and Play викликає цю процедуру, якщо знаходить пристрій, за який відповідає драйвер. У ній звичайно створюють структуру даних, відображувану пристроєм (об'єкт

пристрою).

- Набір *процедур диспетчеризації* (dispatch routines), аналогічних функціям файлових операцій у Linux. Ці процедури реалізують дії, допустимі для пристрою (відкриття, закриття, читання, записування тощо). Саме їх викликає менеджер введення-виведення під час виконання запиту.
- *Процедура обробки переривання* (interrupt service routine, ISR) аналогічна коду верхньої половини оброблювача переривання для Linux. Вона є оброблювачем переривання від пристрою, виконується із високим пріоритетом; основне її завдання − запланувати для виконання нижню половину оброблювача (DPC-процедуру).
- *Процедура відкладеної обробки переривання,* DPC-процедура (DPC routine), відповідає коду нижньої половини оброблювача переривання в Linux. Вона виконує більшу частину роботи, пов'язаної з обробкою переривання, після чого сигналізує про необхідність переходу до коду завершення введення-виведення. Особливості виклику цих процедур під час виконання операції введення-виведення наведено нижче.

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

Зупинимося на обробці запиту синхронного введення-виведення до однорівневого драйвера. Цей процес зводиться до такого.

1. Запит введення-виведення перехоплює динамічна бібліотека підсистеми (наприклад, підсистема Win32 перехоплює виклик функції `WriteFile()`).
2. Динамічна бібліотека підсистеми викликає внутрішню функцію `NtWriteFile()`, що звертається до менеджера введення-виведення.
3. Менеджер введення-виведення розміщує у пам'яті пакет IRP, що описує запит, і відсилає його відповідному драйверу пристрою викликом функції `IoCallDriver()`.

Подальші кроки аналогічні до описаних для Linux.

4. Драйвер витягає дані із пакета IRP, передає їх контролеру пристрою і дає йому команду почати введення-виведення.
5. Драйвер викликає функцію очікування, поточний потік при цьому призупиняють. Для асинхронного введення-виведення цей етап не виконують.
6. Коли пристрій завершує операцію, контролер генерує переривання, яке обслуговує драйвер.
7. Драйвер викликає функцію `IoCompleteRequest()` для того щоб повідомити менеджерові введення-виведення про завершення ним обробки запиту, заданого пакетом IRP, після чого виконують код завершення операції.

На двох останніх етапах зупинимося окремо.

Принципи обробки переривань введення-виведення у Windows XP майже не відрізняються від розглянутих для Linux. У разі виникнення переривання апаратура викликає оброблювач переривання для даного пристрою. При цьому безпосередній оброблювач (верхня половина) звичайно залишається на рівні переривань пристрою тільки для того щоб поставити на виконання нижню половину (DPC) і завершитися. Основну роботу здійснює, як і в Linux, нижня половина, що виконується із меншим пріоритетом (на рівні переривань DPC/dispatch). Після завершення обробки драйвер просить менеджера введення-виведення завершити обробку запиту і вилучити із системи пакет IRP.

Після завершення виконання функції DPC починається останній етап обробки

запиту − завершення введення-виведення (I/O completion).

Таке завершення розрізняють для різних операцій. Звичайно воно зводиться, як і в Linux, до копіювання даних в адресний простір процесу користувача (це може бути буфер введення-виведення або блок статусу операції − структура, задана потоком, що робив виклик).

У разі синхронного введення-виведення адресний простір належить до процесу, що робив виклик, і дані можуть бути записані в нього безпосередньо. Якщо запит був асинхронним, активний потік швидше за все належить до іншого процесу, і потрібно дочекатися, поки адресний простір потрібного процесу не стане доступним (тобто поки не почне виконуватися потік, що викликав операцію). Для цього менеджер введення-виведення планує до виконання спеціальну процедуру, яку називають *APC-процедурою* (від Asynchronous Procedure Call − асинхронний виклик процедури). APC-процедура виконується лише в контексті конкретного потоку, тому очікуватиме, поки цей потік не продовжить своє виконання. Далі вона отримує керування, копіює потрібні дані в адресний простір процесу, що робив виклик, вивільняє пам'ять із-під пакета IRP і переводить файловий дескриптор, для якого виконувалась операція (або інший об'єкт, наприклад, порт завершення введення-виведення) у сигналізований стан, для того щоб потік, який викликав операцію (або будь-який потік, що очікував на цих об'єктах) відновив своє виконання. Після цього введення-виведення вважають завершеним.

Підхід із використанням пакетів IRP найзручніший для роботи із багаторівневими драйверами. Особливості обробки в цьому разі опишемо на прикладі виконання запиту до файлової системи, драйвер якої розташований поверх драйвера диска.

Після створення пакета IRP менеджер введення-виведення передає його драйверу верхнього рівня (у нашому випадку − файлової системи). Подальші дії залежать від запиту і реалізації його обробки в цьому драйвері − він може відіслати драйверу нижнього рівня той самий пакет, а може згенерувати і відіслати набір нових пакетів. Ці два підходи розглянемо докладніше.

Повторне використання пакета IRP найчастіше застосовують, коли один запит до драйвера верхнього рівня однозначно транслюється в один запит до драйвера нижнього рівня (наприклад, запит до файлової системи − у запит до драйвера диска на читання одного сектора). Структура пакета IRP розрахована на те, що він буде використаний різними драйверами, розташованими один під одним. Фактично відбувається обробка за принципом стека (пакет із верхнього рівня передають на нижній, обробляють, а потім знову повертають на верхній рівень, як у разі вкладених викликів функцій), тому дані для різних драйверів усередині пакета IRP організовані у вигляді стека. Окремі позиції в цьому стеку можуть бути заповнені відповідними драйверами на шляху пакета від одного драйвера до іншого. Зазначимо, однак, що розмір пакета під час його переміщень не змінюють − пам'ять для нього відразу виділяють з урахуванням кількості драйверів, через які він має пройти.

З іншого боку, драйвер верхнього рівня може розбити пакет IRP на кілька пов'язаних пакетів, які задають паралельну обробку одного запиту введення-виведення. Наприклад, коли дані для читання розкидані по диску, файлова система може створити набір пакетів, кожен із яких відповідатиме за читання окремого сектора або групи секторів. Усі ці пакети доставляють драйверу пристрою (диска), що обробляє їх по одному, при цьому файлова система відслідковує процес. Після того як усі дії відповідно до пакетів набору виконані, підсистема введення-виведення відновлює первісний пакет і повертає керування процесу або драйверу верхнього рівня.

Аналогічно до `ioctl()` в UNIX/Linux, у Win32 API є функція, що безпосередньо

викликає команду драйвера пристрою і передає йому необхідні параметри. Це функція DeviceIoControl().

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

де: hd – дескриптор пристрою, відкритого CreateFile(); це може бути том, каталог тощо, імена довільних пристроїв утворюються як \\.\ім'я_пристрою;

ioc_code – код команди драйвера;

in_buf – буфер із вхідними даними для виклику, in_bufsize – його довжина;

out_buf – буфер з вихідними даними виклику, out_bufsize – його довжина;

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

Наведемо приклад використання DeviceloControl() для реалізації стискання файла на файловій системі NTFS. Щоб стиснути файл, потрібно надіслати драйверу файлової системи команду із кодом FSCTL_SET_COMPRESSION. Як вхідні дані при цьому передають формат стискання (заданий цілочисловою змінною зі значенням COMPRESSION_FORMAT_DEFAULT). Файл має бути відкритий для читання і записування.

```
unsigned short ctype = COMPRESSION_FORMAT_DEFAULT;
DWORD ret_bytes = 0;
HANDLE fh=CreateFile("myfile.txt",GENERIC_READ|GENERIC_WRITE,...);
DeviceIoControl(fh, FSCTL_SET_COMPRESSION,
  (LPVOID) &ctype, sizeof(ctype), NULL, 0, &ret_bytes, NULL);
CloseHandle(fh);
```

Зазначимо, що багато функцій керування файловою системою NTFS (зокрема, робота з точками з'єднання) реалізовані через інтерфейс цієї функції.

Висновки

- Однією із найважливіших функцій ОС є керування пристроями введення-виведення. Під час його реалізації насамперед важливо безпосередньо реалізувати виконання операцій введення-виведення. Найпоширенішими підходами до розв'язання цього завдання є опитування пристроїв введення-виведення на основі переривань і використання контролерів доступу до пам'яті (DMA).
- Другим важливим завданням є реалізація операцій з організації виконання введення-виведення у ядрі. Основними підходами тут є планування операцій введення-виведення, буферизація і спулінг. Необхідно завжди враховувати можливість виникнення помилок введення-виведення.
- Третім завданням є організація різних засобів введення-виведення для використання в режимі користувача. Сучасні ОС надають різні високоефективні підходи до реалізації таких засобів: синхронне й асинхронне введення-виведення, введення-виведення із повідомленням, порти завершення введення-виведення. Більшість цих засобів розраховані на використання у поєднанні з багатопотоковістю.
- Для реалізації всіх цих можливостей ОС повинна мати драйвери пристроїв, які реалізують базовий набір операцій доступу до пристроїв і надають для використання цих операцій простий у застосуванні універсальний інтерфейс (подібний до інтерфейсу файлової системи в UNIX-сумісних ОС).

Контрольні запитання та завдання

1.Коли краще використовувати опитування завершення введення-виведення, а коли − введення-виведення, кероване перериваннями? Опишіть гібридну стратегію введення-виведення, яка об'єднувала б переваги обох підходів.

2.Для яких із приведених пристроїв є сенс використовувати буферизацію, для яких − спулінг, а для яких − кешування:

а) миша;
б) накопичувач на магнітній стрічці (стример);
в) накопичувач на жорстких магнітних дисках;
г) графічний адаптер?

3.У сучасних ОС введення-виведення з повідомленням часто дає змогу досягти підтримки більшого числа клієнтів порівняно з іншими підходами. Поясніть, у яких випадках це відбувається і чому.

4.Видозмініть клієнт-серверну систему, розроблену під час виконання завдання 11 з розділу 11. Сервер має реалізовувати архітектуру «потік для запиту». Клієнт для Windows XP повинен використовувати введення-виведення з перекриттям, клієнт для Linux − асинхронне введення-виведення відповідно до POSIX, якщо в системі є його реалізація.

5.Що потрібно робити, якщо у разі використання багатопотокового введення-виведення, введення-виведення з повідомленням або порту завершення введення-виведення надійшов запит, що містить тільки частину даних, необхідних для обслуговування клієнта, і передбачено, що інші дані надійдуть слідом? Пам'ятайте про можливості використання буферизації.

6.Чи можливо, щоб кількість активних потоків для порту завершення введення-виведення перевищило *Rmax?* Якщо так, то в якому випадку це трапляється? За яких умов можна сказати, що порт завершення введення-виведення реалізує спулінг?

7.Реалізуйте сервер із завдання 4 цього розділу з використанням порту завершення введення-виведення.

8.Модифікуйте клієнтське застосування з завдання 11 розділу 11, зазначивши максимально допустимий час встановлення з'єднання T_{max}. Виконання застосування має припинятися, якщо з'єднання не було встановлене протягом T_{max}. Використовуйте таймер відкладеного виконання.

9.Назвіть основні відмінності в реалізації та використанні драйверів блокових і символьних пристроїв у Linux.

10.Перелічіть послідовність дій ОС під час виконання операції write() для асинхронного пристрою.

11.У Windows XP функціональність підсистеми введення-виведення надана у вигляді об'єктів виконавчої підсистеми. Оцініть переваги і недоліки цього підходу.