ЛЕКЦІЯ 2. АРХІТЕКТУРА ТА РЕСУРСИ ОПЕРАЦІЙНИХ СИСТЕМ

Операційну систему можна розглядати як сукупність компонентів, кожен з яких відповідає за певні функції. Набір таких компонентів і порядок їхньої взаємодії один з одним та із зовнішнім середовищем визначається *архітектурою операційної системи.*

У цьому розділі ми ознайомимося з основними поняттями архітектури операційних систем, підходами до її реалізації, особливостями взаємодії ОС із зовнішнім середовищем. Реалізацію архітектури буде розглянуто на прикладах UNIX, Linux і Windows.

2.1. Базові поняття архітектури операційних систем

2.1.1. Механізми і політика

В ОС насамперед необхідно виділити набір фундаментальних можливостей, які надають її компоненти; ці базові можливості становлять *механізм* (mechanism). З іншого боку, необхідно приймати рішення щодо використання зазначених можливостей; такі рішення визначають *політику* (policy). Отже, механізм показує, що реалізовано компонентом, а політика – як це можна використати.

Коли за реалізацію механізму і політики відповідають різні компоненти (механізм відокремлений від політики), спрощується розробка системи і підвищується її гнучкість. Компонентам, що реалізують механізм, не повинна бути доступна інформація про причини та цілі його застосування; усе, що потрібно від них, - це виконувати призначену їм роботу. Для таких компонентів використовують термін «вільні від політики» (policy-free). Компоненти, відповідальні за політику, мають оперувати вільними від неї компонентами як будівельними блоками, для них недоступна інформація про деталі реалізації механізму.

Прикладом відокремлення механізму від політики є керування введенням-виведенням. Базові механізми доступу до периферійних пристроїв реалізують драйвери. Політику використання цих механізмів задає програмне забезпечення, що здійснює введення-виведення. Докладніше це питання буде розглянуто у наступних лекціях.

2.1.2. Ядро системи. Привілейований режим і режим користувача

Базові компоненти ОС, які відповідають за найважливіші її функції, зазвичай перебувають у пам'яті постійно і виконуються у привілейованому режимі, називають *ядром операційної системи* (operating system kernel).

Існуючі на сьогодні підходи до проектування архітектури ОС по-різному визначають функціональність ядра. До найважливіших функцій ОС, виконання яких звичайно покладають на ядро, належать обробка переривань, керування пам'яттю, керування введенням-виведенням. До надійності та продуктивності ядра висувають підвищені вимоги.

Основною характерною ознакою ядра є те, що воно виконується у привілейованому режимі. Розглянемо особливості цього режиму.

Для забезпечення ефективного керування ресурсами комп'ютера ОС повинна мати певні привілеї щодо прикладних програм. Треба, щоб прикладні програми не втручалися в роботу ОС, і водночас ОС повинна мати можливість втрутитися в роботу будь-якої програми, наприклад для перемикання процесора або розв'язання конфлікту в боротьбі за ресурси. Для реалізації таких привілеїв потрібна апаратна підтримка: процесор має підтримувати принаймні два режими роботи – *привілейований* (захищений режим, режим ядра, kernel mode) і *режим користувача* (user mode). У режимі користувача недопустимі команди, які є критичними для роботи системи (перемикання задач, звертання до пам'яті за заданими межами, доступ до пристроїв введення-виведення тощо).

Розглянемо, яким чином використовуються різні режими процесора під час взаємодії між ядром і застосуваннями.

Після завантаження ядро перемикає процесор у привілейований режим і отримує цілковитий контроль над комп'ютером. Кожне застосування запускається і виконується в режимі користувача, де воно не має доступу до ресурсів ядра й інших програм. Коли потрібно виконати дію, реалізовану в ядрі, застосування робить *системний виклик* (system call). Ядро перехоплює його, перемикає процесор у привілейований режим, виконує дію, перемикає процесор назад у режим користувача і повертає результат застосування.

Системний виклик виконується повільніше за виклик функції, реалізованої в режимі користувача, через те що процесор двічі перемикається між режимами. Для підвищення продуктивності в деяких ОС частина функціональності реалізована в режимі користувача, тому для доступу до неї системні виклики використовувати не потрібно.

2.1.3. Системне програмне забезпечення

Окрім ядра, важливими складниками роботи ОС є також застосування режиму користувача, які виконують системні функції. До такого *системного програмного забезпечення* належать:

- системні програми (утиліти), наприклад: командний інтерпретатор, програми резервного копіювання та відновлення даних, засоби діагностики й адміністрування;
- системні бібліотеки, у яких реалізовані функції, що використовуються у застосуваннях користувача.

Системне програмне забезпечення може розроблятися й постачатися окремо від ОС. Наприклад, може бути кілька реалізацій командного інтерпретатора, засобів резервного копіювання тощо. Системні програми і бібліотеки взаємодіють з ядром у такий самий спосіб, як і прикладні програми.

2.2. Реалізація архітектури операційних систем

У цьому розділі розглянуто кілька підходів до реалізації архітектури операційних систем. У реальних ОС звичайно використовують деяку комбінацію цих підходів.

2.2.1. Монолітні системи

ОС, у яких усі базові функції сконцентровані в ядрі, називають *монолітними системами.* У разі реалізації монолітного ядра ОС стає продуктивнішою (процесор не перемикається між режимами під час взаємодії між її компонентами), але менш надійною (весь її код виконується у привілейованому режимі, і помилка в кожному з компонентів є критичною).

Монолітність ядра не означає, що всі його компоненти мають постійно перебувати у пам'яті. Сучасні ОС дають можливість динамічно розміщувати в адресному просторі ядра фрагменти коду *{модулі ядра).* Реалізація модулів ядра дає можливість також досягти його розширюваності (для додання нової функціональності досить розробити і завантажити у пам'ять відповідний модуль).

2.2.2. Багаторівневі системи

Компоненти *багаторівневих ОС* утворюють ієрархію рівнів (шарів, layers), кожен, з яких спирається на функції попереднього рівня. Найнижчий рівень безпосередньо взаємодіє з апаратним забезпеченням, на найвищому рівні реалізуються системні виклики.

У традиційних багаторівневих ОС передача керування з верхнього рівня на нижній реалізується як системний виклик. Верхній рівень повинен мати права на виконання цього виклику, перевірка цих прав виконується за підтримки апаратного забезпечення. Прикладом такої системи є ОС Multics, розроблена в 60-ті роки. Практичне застосування цього підходу сьогодні обмежене через низьку продуктивність.

Рівні можуть виділятися й у монолітному ядрі; у такому разі вони підтримуються програмне і спричиняють спрощення реалізації системи. У монолітному_ ядрі визначають рівні, перелічені нижче.

- *Засоби абстрагування від устаткування,* які взаємодіють із апаратним забезпеченням безпосередньо, звільняючи від реалізації такої взаємодії інші компоненти системи.
- *Базові засоби ядра,* які відповідають за найфундаментальніші, найпростіші дії ядра, такі як запис блоку даних на диск. За допомогою цих засобів виконуються вказівки верхніх рівнів, пов'язані з керуванням ресурсами.
- *Засоби керування ресурсами* (або менеджери ресурсів), що реалізують основні функції ОС (керування процесами, пам'яттю, введенням-виведенням тощо). На цьому рівні приймаються найважливіші рішення з керування ресурсами, які виконуються з використанням базових засобів ядра.
- *Інтерфейс системних викликів,* який служить для реалізації зв'язку із системним і прикладним програмним забезпеченням.

Розмежування базових засобів ядра і менеджерів ресурсів відповідає відокремленню механізмів від політики в архітектурі системи. Базові засоби ядра визначають механізми функціонування системи, менеджери ресурсів реалізують політику.

2.2.3. Системи з мікроядром

Один із напрямів розвитку сучасних ОС полягає в тому, що у привілейованому режимі реалізована невелика частка функцій ядра, які є *мікроядром* (microkernel). Інші функції ОС виконуються процесами режиму користувача (серверними процесами, серверами). Сервери можуть відповідати за підтримку файлової системи, за роботу із процесами, пам'яттю тощо.

Мікроядро здійснює зв'язок між компонентами системи і виконує базовий розподіл ресурсів. Щоб виконати системний виклик, процес (клієнтський процес, клієнт) звертається до мікроядра. Мікроядро посилає серверу запит, сервер виконує роботу і пересилає відповідь назад, а мікроядро переправляє його клієнтові (рис. 2.1). Клієнтами можуть бути не лише процеси користувача, а й інші модулі ОС.

Перевагами мікроядрового підходу є:

- невеликі розміри мікроядра, що спрощує його розробку й налагодження;
- висока надійність системи, внаслідок того що сервери працюють у режимі користувача й у них немає прямого доступу до апаратного забезпечення;
- більша гнучкість і розширюваність системи (непотрібні компоненти не займають місця в пам'яті, розширення функціональності системи зводиться до додавання в неї нового сервера);
- можливість адаптації до умов мережі (спосіб обміну даними між клієнтом і сервером не залежить від того, зв'язані вони мережею чи перебувають на одному комп'ютері).

Головним недоліком мікроядрового підходу є зниження продуктивності. Замість двох перемикань режиму процесора у разі системного виклику відбувається чотири (два – під час обміну між клієнтом і мікроядром, два – між сервером та мікроядром).

Зазначений недолік є, швидше, теоретичним, на практиці продуктивність і надійність мікроядра залежать насамперед від якості його реалізації. Так, в ОС QNX мікроядро займає кілька кілобайтів пам'яті й забезпечує мінімальний набір функцій, при цьому система за продуктивністю відповідає ОС реального часу.

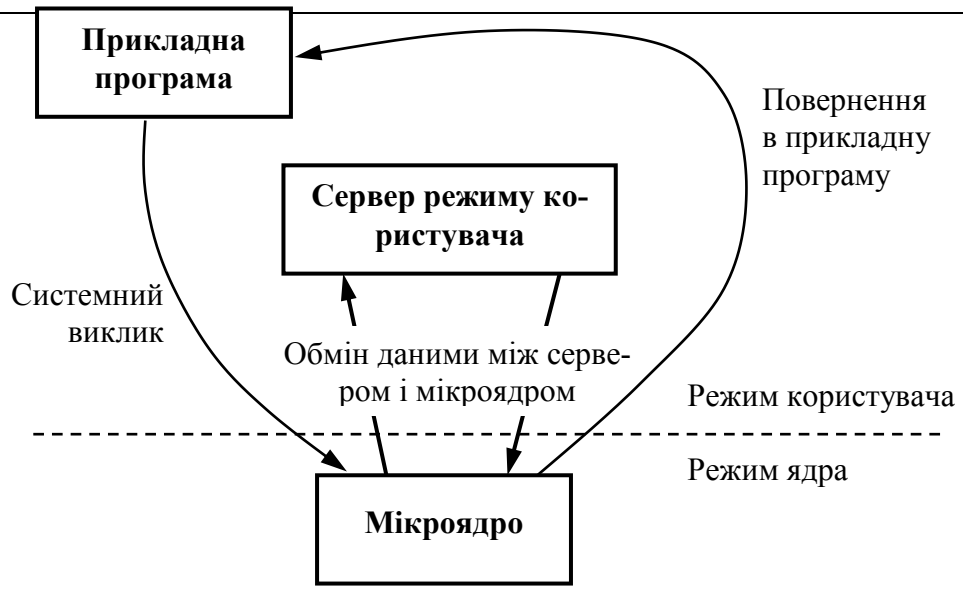

Рис. 2.1. Виконання системного виклику в архітектурі з мікроядром

2.2.4. Концепція віртуальних машин

У *системах віртуальних машин* програмним шляхом створюють копії апаратного забезпечення (відбувається його емуляція). Ці копії *(віртуальні машини)* працюють паралельно, на кожній із них функціонує програмне забезпечення, з яким взаємодіють прикладні програми і користувачі.

Уперше концепція віртуальних машин була реалізована в 70-ті роки в операційній системі VM фірми IBM. У СРСР варіант цієї системи (VM/370) був широко розповсюджений у 80-ті роки і мав назву Система віртуальних машин ЄС ЕОМ (СВМ ЄС). Розглянемо архітектуру цієї ОС, що показана на рис. 2.2.

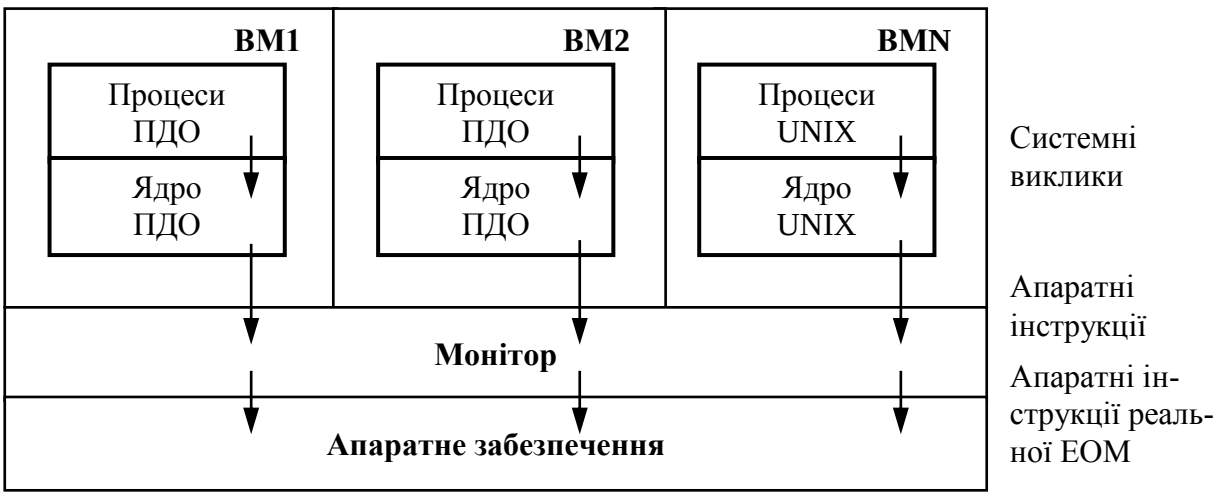

Рис. 2.2. Архітектура VM/370

Ядро системи, яке називалося монітором віртуальних машин (VM Monitor, МВМ), виконувалося на фізичній машині, безпосередньо взаємодіючи з її апаратним забезпеченням. Монітор реалізовував набір віртуальних машин (ВМ). Кожна ВМ була точною копією апаратного забезпечення, на ній могла бути запущена будь-яка ОС, розроблена для цієї архітектури. Найчастіше на ВМ встановлювали спеціальну однокористувацьку ОС CMS (підсистема діалогової обробки, ПДО). На різних ВМ могли одночасно функціонувати різні ОС.

Сьогодні концепція віртуальних машин застосовується і в прикладному програмному забезпеченні.

2.3. Операційна система та її оточення

Із означення ОС випливає, що вона реалізує зв'язок між апаратним забезпеченням комп'ютера (через інтерфейс апаратного забезпечення) і програмами користувача (через інтерфейс прикладного програмування). У цьому розділі розглянуто особливості реалізації та використання цих інтерфейсів.

2.3.1. Взаємодія ОС і апаратного забезпечення

Взаємодію ОС і апаратного забезпечення слід розглядати з двох боків. З одного боку, ОС повинна реалізовувати засоби взаємодії з апаратним забезпеченням, з іншого – архітектуру комп'ютера треба проектувати з урахуванням того, що на комп'ютері функціонуватиме ОС.

Інтерфейс апаратного забезпечення має бути повністю схований від прикладних програм і користувачів, усю роботу з ним виконує ОС. Розглянемо особливості апаратної підтримки цього інтерфейсу і його використання в сучасних ОС.

Засоби апаратної підтримки операційних систем

Сучасні апаратні архітектури комп'ютерів реалізують базові засоби підтримки операційних систем. До них належать: система переривань, привілейований режим процесора, засоби перемикання задач, підтримка керування пам'яттю (механізми трансляції адрес, захист пам'яті), системний таймер, захист пристроїв введення-виведення, базова система введення-виведення (BIOS). Розглянемо ці засоби докладніше.

Система переривань є основним механізмом, що забезпечує функціонування ОС. За допомогою переривань процесор отримує інформацію про події, не пов'язані з основним циклом його роботи (отриманням інструкцій з пам'яті та їхнім виконанням). Переривання бувають двох типів: *апаратні* і *програмні.*

Апаратне переривання – це спеціальний сигнал (запит переривання, IRQ), що передається процесору від апаратного пристрою.

До апаратних переривань належать:

- переривання введення-виведення, що надходять від контролера периферійного пристрою; наприклад, таке переривання генерує контролер клавіатури при натисканні на клавішу;
- переривання, пов'язані з апаратними або програмними помилками (такі переривання виникають, наприклад, у разі збою контролера диска, доступу до забороненої області пам'яті або ділення на нуль).

Програмні переривання генерує прикладна програма, виконуючи спеціальну *інструкцію переривання.* Така інструкція є в системі команд більшості процесорів. Обробка програмних переривань процесором не відрізняється від обробки апаратних переривань.

Якщо переривання відбулося, то процесор негайно передає керування спеціальній процедурі – *оброблювачеві переривання.* Після виходу з оброблювача процесор продовжує виконання інструкцій перерваної програми. Розрізняють два типи переривань залежно від того, яка інструкція буде виконана після виходу з оброблювача: для *відмов* (faults) повторюється інструкція, що спричинила переривання, для *пасток* (traps) – виконується наступна інструкція. Усі переривання введення-виведення і програмні переривання належать до категорії пасток, більшість переривань через помилки є відмовами.

За встановлення оброблювачів переривань зазвичай відповідає ОС. Можна сказати, що сучасні ОС керовані перериваннями (interrupt-driven), бо, якщо ОС не зайнята виконанням якої-небудь задачі, вона очікує на переривання, яке й залучає її до роботи.

Для реалізації *привілейованого режиму процесора* в одному з його регістрів передбачено спеціальний біт (біт режиму), котрий показує, у якому режимі перебуває процесор. У разі програмного або апаратного переривання процесор автоматично перемикається у привілейований режим, і саме тому ядро ОС (яке складається з оброблювачів переривань) завжди отримує керування в цьому режимі. За будь-якої спроби безпосередньо виконати привілейовану інструкцію в режимі користувача відбувається апаратне переривання.

Засоби перемикання задач дають змогу зберігати вміст регістрів процесора (контекст задачі) у разі припинення задачі та відновлювати дані перед її подальшим виконанням.

Механізм трансляції адрес забезпечує перетворення адрес пам'яті, з якими працює програма, в адреси фізичної пам'яті комп'ютера. Апаратне забезпечення генерує фізичну адресу, використовуючи спеціальні таблиці трансляції.

Захист пам'яті забезпечує перевірку прав доступу до пам'яті під час кожної спроби його отримати. Засоби захисту пам'яті інтегровані з механізмами трансляції адрес: у таблицях трансляції утримується інформація про права, необхідні для їхнього використання, і про *ліміт* (розміри ділянки пам'яті, до якої можна отримати доступ з їхньою допомогою). Неможливо одержати доступ до пам'яті поверх ліміту або за відсутності прав на використання таблиці трансляції.

Системний таймер є апаратним пристроєм, який генерує *переривання таймера* через певні проміжки часу. Такі переривання обробляє ОС; інформацію від таймера найчастіше використовують для визначення часу перемикання задач.

Захист пристроїв введення-виведення ґрунтується на тому, що всі інструкції введення-виведення визначені як привілейовані. Прикладні програми здійснюють введення-виведення не прямо, а за посередництвом ОС.

Базова система введення-виведення (BIOS) – службовий програмний код, що зберігається в постійному запам'ятовувальному пристрої і призначений для ізоляції ОС від конкретного апаратного забезпечення. Зазначимо, що засоби BIOS не завжди дають змогу використати всі можливості архітектури: наприклад, процедури BIOS для архітектури IA-32 не працюють у захищеному режимі. Тому сучасні ОС використовують їх тільки для початкового завантаження системи.

Апаратна незалежність і здатність до перенесення ОС

Компоненти ядра, які відповідають за безпосередній доступ до апаратного забезпечення, виділено в окремий рівень абстрагування від устаткування, що взаємодіє з іншою частиною системи через стандартні інтерфейси. Тим самим спрощується досягнення апаратної незалежності ОС.

Рівень абстрагування від устаткування відображає такі особливості архітектури, як число процесорів, типи їхніх регістрів, розрядність і організація пам'яті тощо. Що більше відмінностей між апаратними архітектурами, для яких призначена ОС, то складніша розробка коду цього рівня.

Крім рівня абстрагування від устаткування, від апаратного забезпечення залежать драйвери зовнішніх пристроїв. Такі драйвери проектують заздалегідь як апаратнозалежні, їх можна додавати та вилучати за потребою; для доступу до них зазвичай використовують універсальний інтерфейс.

Здатність до перенесення ОС визначається обсягом робіт, необхідних для того, щоб система могла працювати на новій апаратній платформі. ОС з такими властивостями має відповідати певним вимогам.

- Більша частина коду операційної системи має бути написана мовою високого рівня (звичайно для цього використовують мови С і С++, компілятори яких розроблені для більшості архітектур). Використання мови асемблера допустиме лише тоді, коли продуктивність компонента є критичною для системи.
- Код, що залежить від апаратного забезпечення (рівень абстрагування від устаткування) має бути відокремлений від іншої частини системи так, щоб у разі переходу на іншу архітектуру потрібно було переписувати тільки цей рівень.

2.3.2. Взаємодія ОС і виконуваного програмного коду

У роботі в режимі користувача часто необхідне виконання дій, реалізованих у ядрі ОС (наприклад, під час запису на диск із прикладної програми). Для цього треба забезпечити взаємодію коду режиму користувача та ОС. Розглянемо особливості такої взаємодії.

Системні виклики та інтерфейс програмування застосувань

Системний виклик – це засіб доступу до певної функції ядра операційної системи із прикладних програм. Набір системних викликів визначає дії, які ядро може виконати за запитом процесів користувача. Системні виклики задають інтерфейс між прикладною програмою і ядром ОС.

Розглянемо послідовність виконання системного виклику.

1. Припустимо, що для процесу, який виконується в режимі користувача, потрібна функція, реалізована в ядрі системи.
2. Для того щоб звернутися до цієї функції, процес має передати керування ядру ОС, для чого необхідно задати параметри виклику і виконати програмне переривання *(інструкцію системного виклику)*. Відбувається перехід у привілейований режим.
3. Після отримання керування ядро зчитує параметри виклику і визначає, що потрібно зробити.
4. Після цього ядро виконує потрібні дії, зберігає в пам'яті значення, які слід повернути, і передає керування програмі, що його викликала. Відбувається перехід назад у режим користувача.
5. Програма зчитує з пам'яті повернені значення і продовжує свою роботу.

Як бачимо, кожний системний виклик спричиняє перехід у привілейований режим і назад (у мікроядровій архітектурі, як було зазначено вище, таких переходів може бути і більше).

Розглянемо способи передачі параметрів у системний виклик. До них належать:

- передача параметрів у регістрах процесора;
- занесення параметрів у певну ділянку пам'яті й передача покажчика на неї в регістрі процесора.

Системні виклики призначені для безпосереднього доступу до служб ядра ОС. На практиці вони не вичерпують (а іноді й не визначають) ті функції, які можна використати у прикладних програмах для доступу до служб ОС або засобів системних бібліотек. Для позначення цього набору функцій використовують термін *інтерфейс програмування застосувань* (Application Programming Interface, API).

Взаємозв'язок між функціями API і системними викликами неоднаковий у різних ОС.

По-перше, кожному системному виклику може бути поставлена у відповідність бібліотечна функція, єдиним завданням якої є виконання цього виклику. Таку функцію називають *пакувальником системного виклику* (system call wrapper). Для програміста в цьому разі набір функцій API виглядає як сукупність таких пакувальників і додаткових функцій, реалізованих бібліотеками повністю або частково в режимі користувача. Це рішен-

ня прийняте за основу в UNIX; у такому разі прийнято говорити про використання системних викликів у прикладних програмах (насправді у програмах викликають пакувальники системних викликів).

По-друге, можна надати для використання у прикладних програмах універсальний інтерфейс програмування застосувань (API режиму користувача) і повністю сховати за ним набір системних викликів. Для програміста кожна функція такого API є бібліотечною функцією режиму користувача, пакувальника в цьому разі немає, відомості про системні виклики є деталями реалізації ОС. Це властиве Windows-системам, де подібний універсальний набір функцій називають *Win32 API* .

Програмна сумісність

Дотепер ми розглядали виконання в ОС програм, розроблених спеціально для неї. Іноді буває необхідно виконати в середовищі ОС програми, розроблені для інших ОС і, можливо, для іншої апаратної архітектури. У цьому разі виникає проблема *програмної сумісності.*

Програмна сумісність означає можливість виконувати в середовищі однієї операційної системи програми, розроблені для іншої ОС. Розрізняють *сумісність на рівні вихідних текстів* (можливість перенесення вихідних текстів) та *бінарну сумісність* (можливість перенесення виконуваного коду).

Для сумісності на рівні вихідних текстів необхідно, щоб для всіх ОС існувала реалізація компілятора мови і API, що його використовує програма.

Сьогодні таку сумісність забезпечує стандартизація розробки програмного забезпечення, а саме:

- наявність стандарту на мови програмування (насамперед на C і C++) і стандартних компіляторів;
- наявність стандарту на інтерфейс операційної системи (API).

Робота зі стандартизації інтерфейсу операційних систем відбувається у рамках проекту POSIX (Portable Operating System Interface). Найбільш важливим стандартом є POSIX 1003.1, який описує набір бібліотечних процедур (таких, як відкриття файла, створення нового процесу тощо), котрі мають бути реалізовані в системі. Цей процес стандартизації триває дотепер, останньою редакцією стандарту є базова специфікація Open Group/IEEE. Зазначені стандарти відображають традиційний набір засобів, реалізованих в UNIX-сумісних системах.

Завдання забезпечення бінарної сумісності виникає тоді, коли потрібно запустити на виконання файл прикладної програми у середовищі іншої операційної системи. Таке завдання значно складніше в реалізації, найпоширеніший підхід до його розв'язання - *емуляція середовища виконання.* У цьому разі програма запускається під керуванням іншої програми – *емулятора,* який забезпечує динамічне перетворення інструкцій програми в інструкції потрібної архітектури. Прикладом такого емулятора є програма wine, яка дає можливість запускати програми, розроблені для Win32 API, у середовищі Linux через емуляцію функцій Win32 API системними викликами Linux.

2.4. Особливості архітектури: UNIX і Linux 2.4.1. Базова архітектура UNIX

UNIX є прикладом досить простої архітектури ОС. Більша частина функціональності цієї системи міститься в ядрі, ядро спілкується із прикладними програмами за допомогою системних викликів. Базова структура класичного ядра UNIX зображена на рис. 2.3.

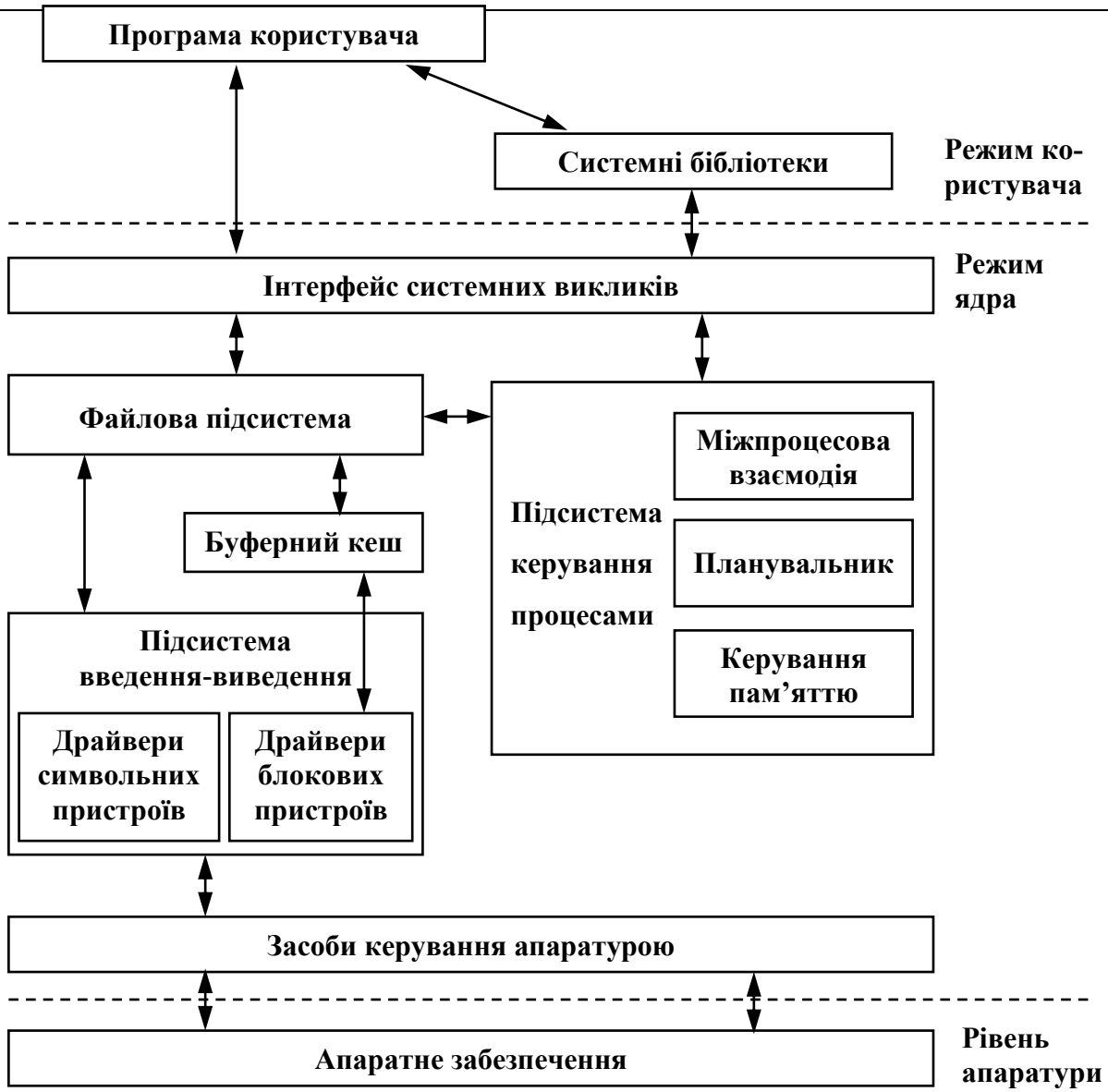

Рис. 2.3. Архітектура UNIX

Система складається із трьох основних компонентів: підсистеми керування процесами, файлової підсистеми та підсистеми введення-виведення.

Підсистема керування процесами контролює створення та вилучення процесів, розподілення системних ресурсів між ними, міжпроцесову взаємодію, керування пам'яттю.

Файлова підсистема забезпечує єдиний інтерфейс доступу до даних, розташованих на дискових накопичувачах, і до периферійних пристроїв. Такий інтерфейс є однією з найважливіших особливостей UNIX. Одні й ті самі системні виклики використовують як для обміну даними із диском, так і для виведення на термінал або принтер (програма працює із принтером так само, як із файлом). При цьому файлова система переадресовує запити відповідним модулям підсистеми введення-виведення, а ті – безпосередньо периферійним пристроям. Крім того, файлова підсистема контролює права доступу до файлів, які значною мірою визначають привілеї користувача в системі.

Підсистема введення-виведення виконує запити файлової підсистеми, взаємодіючи з драйверами пристроїв. В UNIX розрізняють два типи пристроїв: символьні (наприклад, принтер) і блокові (наприклад, жорсткий диск). Основна відмінність між ними полягає в тому, що блоковий пристрій допускає прямий доступ. Для підвищення продуктивності роботи із блоковими пристроями використовують буферний кеш – ділянку пам'яті, у якій зберігаються дані, зчитані з диска останніми. Під час наступних звертань до цих даних вони можуть бути отримані з кеша.

Сучасні UNIX-системи дещо відрізняються за своєю архітектурою.

- У них виділено окремий менеджер пам'яті, відповідальний за підтримку віртуальної пам'яті.
- Стандартом для реалізації інтерфейсу файлової системи є *віртуальна файлова система,* що абстрагує цей інтерфейс і дає змогу організувати підтримку різних типів файлових систем.
- У цих системах підтримується багатопроцесорна обробка, а також багатопотоковість.

Базові архітектурні рішення, такі як доступ до всіх пристроїв введення-виведення через інтерфейс файлової системи або організація системних викликів, залишаються незмінними в усіх реалізаціях UNIX.

2.4.2. Архітектура Linux

В ОС Linux можна виділити три основні частини:

- *ядро,* яке реалізує основні функції ОС (керування процесами, пам'яттю, введенням-виведенням тощо);
- *системні бібліотеки,* що визначають стандартний набір функцій для використання у застосуваннях (виконання таких функцій не потребує переходу в привілейований режим);
- *системні утиліти* (прикладні програми, які виконують спеціалізовані задачі).

Призначення ядра Linux і його особливості

Linux реалізує технологію монолітного ядра. Весь код і структури даних ядра перебувають в одному адресному просторі. У ядрі можна виділити кілька функціональних компонентів.

- *Планувальник процесів* – відповідає за реалізацію багатозадачності в системі (обробка переривань, робота з таймером, створення і завершення процесів, перемикання контексту).
- *Менеджер пам'яті* – виділяє окремий адресний простір для кожного процесу і реалізує підтримку віртуальної пам'яті.
- *Віртуальна файлова система* – надає універсальний інтерфейс взаємодії з різними файловими системами та пристроями введення-виведення.
- *Драйвери пристроїв* – забезпечують безпосередню роботу з периферійними пристроями. Доступ до них здійснюється через інтерфейс віртуальної файлової системи.
- *Мережний інтерфейс* – забезпечує доступ до реалізації мережних протоколів і драйверів мережних пристроїв.
- *Підсистема міжпроцесової взаємодії* – пропонує механізми, які дають змогу різним процесам у системі обмінюватися даними між собою.

Деякі із цих підсистем є логічними компонентами системи, вони завантажуються у пам'ять разом із ядром і залишаються там постійно. Компоненти інших підсистем (наприклад, драйвери пристроїв) вигідно реалізовувати так, щоб їхній код міг завантажуватися у пам'ять на вимогу. Для розв'язання цього завдання Linux підтримує концепцію модулів ядра.

Модулі ядра

Ядро Linux дає можливість на вимогу завантажувати у пам'ять і вивантажувати з неї окремі секції коду. Такі секції називають *модулями ядра* (kernel modules) і виконують у привілейованому режимі.

Модулі ядра надають низку переваг.

- Код модулів може завантажуватися в пам'ять у процесі роботи системи, що спрощує налагодження компонентів ядра, насамперед драйверів.
- З'являється можливість змінювати набір компонентів ядра під час виконання: ті з них, які в цей момент не використовуються, можна не завантажувати у пам'ять.
- Модулі є винятком із правила, за яким код, що розширює функції ядра, відповідно до ліцензії Linux має бути відкритим. Це дає змогу виробникам апаратного забезпечення розробляти драйвери під Linux, навіть якщо не заплановано надавати доступ до їхнього вихідного коду.

Підтримка модулів у Linux складається із трьох компонентів.

- Засоби керування модулями дають можливість завантажувати модулі у пам'ять і здійснювати обмін даними між модулями та іншою частиною ядра.
- Засоби реєстрації драйверів дозволяють модулям повідомляти іншу частину ядра про те, що новий драйвер став доступним.
- Засоби розв'язання конфліктів дають змогу драйверам пристроїв резервувати апаратні ресурси і захищати їх від випадкового використання іншими драйверами.

Один модуль може зареєструвати кілька драйверів, якщо це потрібно (наприклад, для двох різних механізмів доступу до пристрою).

Модулі можуть бути завантажені заздалегідь – під час старту системи (завантажувальні модулі) або у процесі виконання програми, яка викликає їхні функції. Після завантаження код модуля перебуває в тому ж самому адресному просторі, що й інший код ядра. Помилка в модулі є критичною для системи.

Особливості системних бібліотек

Системні бібліотеки Linux є *динамічними бібліотеками,* котрі завантажуються у пам'ять тільки тоді, коли у них виникає потреба. Вони виконують ряд функцій:

- реалізацію пакувальників системних викликів;
- розширення функціональності системних викликів (до таких бібліотек належить бібліотека введення-виведення мови С, яка реалізує на основі системних викликів такі функції, як `printf()`);
- реалізацію службових функцій режиму користувача (сортування, функції обробки рядків тощо).

Застосування користувача

Застосування користувача в Linux використовують функції із системних бібліотек і через них взаємодіють із ядром за допомогою системних викликів.

2.5. Особливості архітектури Windows

У цьому розділі ми розглянемо основні компоненти Windows, які зображені на рис. 2.4. Деякі компоненти Windows виконуються у привілейованому режимі, інші компоненти – у режимі користувача. Ми почнемо розгляд системи з компонентів режиму ядра.

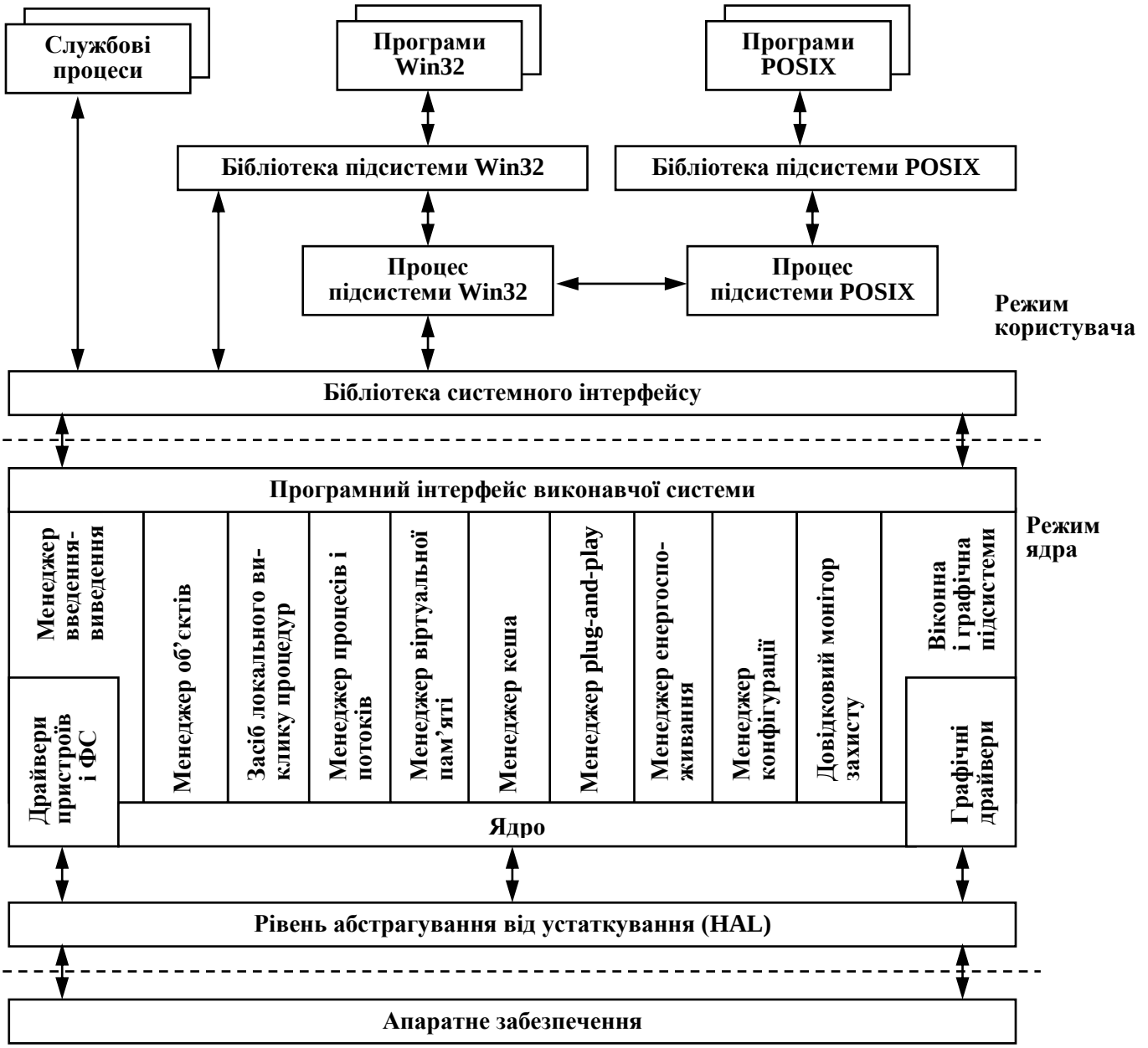

Рис. 2.4. Базові компоненти Windows

2.5.1. Компоненти режиму ядра

У традиційному розумінні ядро ОС містить усі компоненти привілейованого режиму, однак у Windows поняття ядра закріплене тільки за одним із цих компонентів.

Рівень абстрагування від устаткування

У Windows реалізовано рівень абстрагування від устаткування (у цій системі його називають HAL, hardware abstraction layer). Для різних апаратних конфігурацій фірма Microsoft або сторонні розробники можуть постачати різні реалізації HAL.

Хоча код HAL є дуже ефективним, його використання може знижувати продуктивність застосувань мультимедіа. У такому разів використовують спеціальний пакет DirectX, який дає змогу прикладним прогарам звертатися безпосередньо до апаратного забезпечення, обминаючи HAL та інші рівні системи.

Ядро

Ядро Windows відповідає за базові операції системи. До його основних функцій належать:

- перемикання контексту, збереження і відновлення стану потоків; + планування виконання потоків;
- реалізація засобів підтримки апаратного забезпечення, складніших за засоби HAL (наприклад, передача керування оброблювачам переривань).

Ядро Windows відповідає базовим службам ОС і надає набір механізмів для реалізації політики керування ресурсами.

Основним завданням ядра є якомога ефективніше завантаження процесорів системи. Ядро постійно перебуває в пам'яті, послідовність виконання його інструкцій може порушити тільки переривання (під час виконання коду ядра багатозадачність не підтримується). Для прискорення роботи ядро ніколи не перевіряє правильність параметрів, переданих під час виклику його функцій.

Windows не можна віднести до якогось певного класу ОС. Наприклад, хоча за функціональністю ядро системи відповідає поняттю мікроядра, для самої ОС не характерна класична мікроядрова архітектура, оскільки у привілейованому режимі виконуються й інші її компоненти.

Виконавча система

Виконавча система (ВС) Windows (Windows Executive) - це набір компонентів, відповідальних за найважливіші служби ОС (керування пам'яттю, процесами і потоками, введенням-виведенням тощо).

Компонентами ВС є передусім базові засоби підтримки. Ці засоби використовують у всій системі.

- *Менеджер об'єктів* – відповідає за розподіл ресурсів у системі, підтримуючи їхнє універсальне подання через об'єкти.
- *Засіб локального виклику процедур* (LPC) - забезпечує механізм зв'язку між процесами і підсистемами на одному комп'ютері.

Інші компоненти ВС реалізують найважливіші служби Windows. Зупинимося на деяких із них.

- *Менеджер процесів і потоків* – створює та завершує процеси і потоки, а також розподіляє для них ресурси.
- *Менеджер віртуальної пам'яті* – реалізує керування пам'яттю в системі, насамперед підтримку віртуальної пам'яті.
- *Менеджер введення-виведення* – керує периферійними пристроями, надаючи іншим компонентам апаратно-незалежні засоби введення-виведення. Цей менеджер реалізує єдиний інтерфейс для драйверів пристроїв.
- *Менеджер кеша* – керує кешуванням для системи введення-виведення. Часто використовувані блоки диска тимчасово зберігаються в пам'яті, наступні операції введення-виведення звертаються до цієї пам'яті, внаслідок чого підвищується продуктивність.
- *Менеджер конфігурації* – відповідає за підтримку роботи із *системним реєстром* (registry) – ієрархічно організованим сховищем інформації про налаштування системи і прикладних програм.
- *Довідковий монітор захисту* – забезпечує політику безпеки на ізольованому комп'ютері, тобто захищає системні ресурси.

Драйвери пристроїв

У Windows драйвери не обов'язково пов'язані з апаратними пристроями. Застосування, якому потрібні засоби, доступні в режимі ядра, завжди варто оформляти як драйвер. Це пов'язане з тим, що для зовнішніх розробників режим ядра доступний тільки з коду драйверів. Докладніше реалізацію драйверів Windows буде розглянуто в подальших лекціях.

Віконна і графічна підсистеми

Віконна і графічна підсистеми відповідають за інтерфейс користувача – роботу Звікнами, елементами керування і графічним виведенням.

- *Менеджер вікон* – реалізує високорівневі функції. Він керує віконним виведенням, обробляє введення з клавіатури або миші й передає застосуванням повідомлення користувача.
- *Інтерфейс графічних пристроїв* (Graphical Device Interface, GDI) – складається з набору базових операцій графічного виведення, які не залежать від конкретного пристрою (креслення ліній, відображення тексту тощо).
- *Драйвери графічних пристроїв* (відеокарт, принтерів тощо) - відповідають за взаємодію з контролерами цих пристроїв.

Під час створення вікон або елементів керування запит надходить до менеджера вікон, який для виконання базових графічних операцій звертається до GDI. Потім запит передається драйверу пристрою, затим - апаратному забезпеченню через HAL.

2.5.2. Компоненти режиму користувача

Компоненти режиму користувача не мають прямого доступу до апаратного забезпечення, їхній код виконується в ізольованому адресному просторі. Більша частина коду режиму користувача перебуває в динамічних бібліотеках, які у Windows називають DLL (dynamic-link libraries).

Бібліотека системного інтерфейсу

Для доступу до засобів режиму ядра в режимі користувача необхідно звертатися до функцій бібліотеки системного інтерфейсу (ntdll.dll). Ця бібліотека надає набір функцій-перехідників, кожній з яких відповідає функція режиму ядра (системний виклик). Застосування зазвичай не викликають такі функції безпосередньо, за це відповідають підсистеми середовища.

Підсистеми середовища

Підсистеми середовища надають застосуванням користувача доступ до служб операційної системи, реалізуючи відповідний API. Ми зупинимося на двох підсистемах середовища: Win32 і POSIX.

Підсистема Win32, яка реалізує Win32 API, є обов'язковим компонентом Windows XP. До неї входять такі компоненти:

- процес підсистеми Win32 (csrss.exe), що відповідає, зокрема, за реалізацію текстового (консольного) введення-виведення, створення і знищення процесів та потоків;
- бібліотеки підсистеми Win32, які надають прикладним програмам функції Win32 API. Найчастіше використовують бібліотеки gdi32.dll (низькорівневі графічні функції, незалежні від пристрою), user32.dll (функції інтерфейсу користувача) і kernel32.dll (функції, реалізовані у ВС і ядрі).

Після того, як застосування звернеться до функції Win32 API, спочатку буде викликана відповідна функція з бібліотеки підсистеми Win32. Розглянемо варіанти виконання такого виклику.

1. Якщо функції потрібні тільки ресурси її бібліотеки, виклик повністю виконується в адресному просторі застосування без переходу в режим ядра.

2. Якщо потрібен перехід у режим ядра, з коду бібліотеки підсистеми виконується системний виклик. Так відбувається у більшості випадків, наприклад під час створення вікон або елементів керування.

3. Функція бібліотеки підсистеми може звернутися до процесу підсистеми Win32, при цьому:

- коли потрібна тільки функціональність, реалізована даним процесом, переходу в режим ядра не відбувається;
- коли потрібна функціональність режиму ядра, процес підсистеми Win32 виконує системний виклик аналогічно до варіанта 2.

Зазначимо, що до виходу Windows NT 4.0 (1996 рік) віконна і графічна підсистеми працювали в режимі користувача як частина процесу підсистеми Win32 (тобто виклики базових графічних функцій Win32 API оброблялися відповідно до варіанту 3). Надалі для підвищення продуктивності реалізацію цих підсистем було перенесено в режим ядра [14].

Підсистема POSIX працює в режимі користувача й реалізує набір функцій, визначених стандартом POSIX 1003.1. Оскільки застосування, або прикладні програми (applications), написані для однієї підсистеми, не можуть використати функції інших, у POSIX-програмах не можна користуватися засобами Win32 API (зокрема, графічними та мережними функціями), що знижує важливість цієї підсистеми. Підсистема POSIX не є обов'язковим компонентом Windows.

Наперед визначені системні процеси

Ряд важливих процесів користувача система запускає автоматично до закінчення завантаження. Розглянемо деякі з них.

- Менеджер сесій (Session Manager, smss.exe) створюється в системі першим. Він запускає інші важливі процеси (процес підсистеми Win32, процес реєстрації в системі тощо), а також відповідає за їхнє повторне виконання під час аварійного завершення.
- Процес реєстрації в системі (winlogon.exe) відповідає за допуск користувача в систему. Він відображає діалогове вікно для введення пароля, після введення передає пароль у підсистему безпеки і в разі успішної його верифікації запускає засоби створення сесії користувача.
- Менеджер керування службами (Service Control Manager, services.exe) відповідає за автоматичне виконання певних застосувань під час завантаження системи. Застосування, які будуть виконані при цьому, називають *службами* (services). Такі служби, як журнал подій, планувальник задач, менеджер друкування, постачають разом із системою. Крім того, є багато служб сторонніх розробників; так зазвичай реалізовують серверні застосування (сервери баз даних, веб-сервери тощо).

Застосування користувача

Застосування користувача можуть бути створені для різних підсистем середовища. Такі застосування використовують тільки функції відповідного API. Виклики цих функцій перетворюються в системні виклики за допомогою динамічних бібліотек підсистем середовища.

2.5.3. Об'єктна архітектура Windows

Керування ресурсами у Windows реалізується із застосуванням концепції об'єктів. Об'єкти надають універсальний інтерфейс для доступу до системних ресурсів, для яких передбачено спільне використання, зокрема таких, як процеси, потоки, файли і розподілювана пам'ять. Концепція об'єктів забезпечує важливі переваги.

- Імена об'єктів організовані в єдиний простір імен, де їх легко знаходити.
- Доступ до всіх об'єктів здійснюється однаково. Після створення нового об'єкта або після отримання доступу до наявного менеджер об'єктів повертає у застосування *дескриптор об'єкта* (object handle).
- Забезпечено захист ресурсів. Кожну спробу доступу до об'єкта розглядає підсистема захисту – без неї доступ до об'єкта, а отже і до ресурсу, отримати неможливо.

Менеджер об'єктів відповідає за створення, підтримку та ліквідацію об'єктів, задає єдині правила для їхнього іменування, збереження й забезпечення захисту. Підсистеми середовища звертаються до менеджера об'єктів безпосередньо або через інші сервіси ВС. Наприклад, під час запуску застосування підсистема Win32 викликає менеджер процесів для створення нового процесу. В свою чергу менеджер процесів звертається до менеджера об'єктів для створення об'єкта, що представляє процес.

Об'єкти реалізовано як структури даних в адресному просторі ядра. При перезавантаженні системи вміст усіх об'єктів губиться.

Особливості отримання доступу до об'єктів із процесів режиму користувача буде розглянуто в наступній лекції.

Структура заголовка об'єкта

Об'єкти складаються з двох частин: заголовка і тіла об'єкта. У заголовку міститься інформація, загальна для всіх об'єктів, у тілі - специфічна для об'єктів конкретного типу.

До атрибутів заголовка об'єкта належать:

- ім'я об'єкта і його місце у просторі імен;
- дескриптор захисту (визначає права, необхідні для використання об'єкта);
- витрата квоти (ціна відкриття дескриптора об'єкта, дає змогу регулювати кількість об'єктів, які дозволено створювати);
- список процесів, що дістали доступ до дескрипторів об'єкта.

Менеджер об'єктів здійснює керування об'єктами на підставі інформації з їхніх заголовків.

Об'єкти типу

Формат і вміст тіла об'єкта визначається його типом. Новий тип об'єктів може бути визначений будь-яким компонентом ВС. Існує визначений набір типів об'єктів, які створюються під час завантаження системи (такі об'єкти, наприклад, відповідають процесам, відкритим файлам, пристроям введення-виведення).

Частина характеристик об'єктів є загальними для всіх об'єктів цього типу. Для зберігання відомостей про такі характеристики використовують спеціальні *об'єкти типу* (type objects). У такому об'єкті, зокрема, зберігають:

- ім'я типу об'єкта («процес», «потік», «відкритий файл» тощо);
- режими доступу (залежать від типу об'єкта: наприклад, для файла такими режимами можуть бути «читання» і «запис»).

Об'єкти типу недоступні в режимі користувача.

Методи об'єктів

Коли компонент ВС створює новий тип об'єкта, він може зареєструвати у диспетчері об'єктів один або кілька *методів.* Після цього диспетчер об'єктів викликає ці методи на певних етапах життєвого циклу об'єкта. Наведемо деякі з методів об'єктів:

- open – викликається при відкритті дескриптора об'єкта;
- close – викликається при закритті дескриптора об'єкта;
- delete – викликається перед вилученням об'єкта з пам'яті.

Покажчики на код реалізації методів також зберігаються в об'єктах типу.

Простір імен об'єктів

Усі імена об'єктів у ВС розташовані в глобальному просторі імен, тому будь-який процес може відкрити дескриптор об'єкта, вказавши його ім'я. Простір імен об'єктів має ієрархічну структуру, подібно до файлової системи. Аналогом каталогу файлової системи в такому просторі імен є *каталог об'єктів.* Він містить імена об'єктів (зокрема й інших каталогів). Перелічимо деякі наперед визначені імена каталогів:

- Device – імена пристроїв введення-виведення;
- Driver – завантажені драйвери пристроїв;
- ObjectTypes – об'єкти типів.

Простір імен об'єктів, як і окремі об'єкти, не зберігається після перезавантаження системи.

Висновки

- Архітектура ОС визначає набір її компонентів, а також порядок їхньої взаємодії один з одним та із зовнішнім середовищем.
- Найважливішим для вивчення архітектури ОС є поняття ядра системи. Основною характеристикою ядра є те, що воно виконується у привілейованому режимі.
- Основними типами архітектури ОС є монолітна архітектура й архітектура на базі мікроядра. Монолітна архітектура вимагає, щоб головні функції системи були сконцентровані в ядрі, найважливішою її перевагою є продуктивність. У системах на базі мікроядра в привілейованому режимі виконуються тільки базові функції, основними перевагами таких систем є надійність і гнучкість.
- Операційна система безпосередньо взаємодіє з апаратним забезпеченням комп'ютера. Сучасні комп'ютерні архітектури пропонують багато засобів підтримки роботи операційних систем. Для зв'язку з апаратним забезпеченням в ОС виділяється рівень абстрагування від устаткування.
- Операційна система взаємодіє із прикладними програмами. Вона надає набір системних викликів для доступу до функцій, реалізованих у ядрі. Для прикладних програм системні виклики разом із засобами системних бібліотек доступні через інтерфейс програмування застосувань (API).