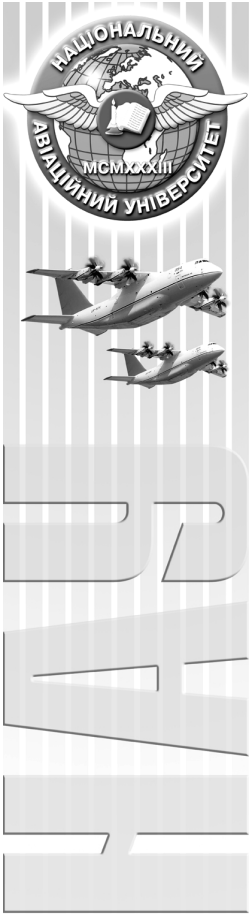


VIVERE!
VINCERE!
CREARE!

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ УКРАЇНИ
Національний авіаційний університет

В. П. Харченко, Є. А. Знаковська, В. А. Бородін

ОПЕРАЦІЙНІ СИСТЕМИ ТА СИСТЕМИ ПРОГРАМУВАННЯ

Навчальний посібник

Київ 2012

УДК
ББК

Рецензенти: *К.С. Сундучков* – д-р техн. наук, професор, заступник директора інституту телекомунікаційних систем, Національний технічний університет України «Київський політехнічний інститут»
В.Г. Самойленко – д-р фіз.-мат. наук, професор, завідувач кафедри математичної фізики, Київський національний університет імені Тараса Шевченка
О.М. Алєксєєв – канд тех. наук, головний спеціаліст-інспектор управління незалежного розслідування авіаційних випадків Державної авіаційної адміністрації України – секретар Державної комісії з безпеки польотів

Затверджено методично-редакційною радою Національного авіаційного університету (протокол № 8/10 від 16.12.2010 р.).

Харченко В. П., Знаковська Є. А., Бородін В. А.

Операційні системи та системи програмування: навч. посіб /В. П. Харченко, Є. А. Знаковська, В. А. Бородін – К.: Вид-во Нац. авіац. ун-ту «НАУ-друк», 2012.– 360с.
ISBN

Гриф надано Міністерством освіти і науки України (лист за №1/11-4722 від 09.06.2011 року)

Присвячено вивченню призначення, функцій і загальних структурних рішень побудови операційних систем, дослідженню операційних систем та систем програмування.

Для студентів напряму підготовки «Аеронавігація», Інституту заочного та дистанційного навчання.

УДК
ББК

ISBN

ЗМІСТ

УМОВНІ ПОЗНАЧЕННЯ І СКОРОЧЕННЯ

АСК ТП – автоматизовані системи керування технологічними процесами
АЦД – алфавітно-цифровий дисплей
АЦДП – алфавітно-цифровий друкувальний пристрій
ВАП – віртуальний адресний простір
ДПД – діаграма потоків данних
ЕОМ – електронно-обчислювальна машина
ЕСТ – електронна струминна трубка
ІС – інформаційна система
НГМД – нагромаджувач на гнучких магнітних дисках
ОЗП – оперативний запам'ятовувальний пристрій
ОС – операційна система
ОСРЧ – операційна система реального часу
ПВВ – пристрій візуального відображення
ПЗ – програмне забезпечення
ПЗП – постійний запам'ятовувальний пристрій
ПК – персональний комп'ютер
САПР – система автоматизованого проектування
СКБД – система керування базами даних
ФС – файлова система
ADO (Active Data Objects) – активні об'єкти даних
APC (Asynchronous Procedure Call) – виклик асинхронної процедури
API (Application Programming Interface) – інтерфейс прикладного програмування
BASIC (Beginner's All-purpose Symbolic Instruction Code) – універсальний код символічних інструкцій для початківців
BDE (Borland Database Engine) – двигун баз даних Борланд
BIOS (Basic Input/Output System) – базова система введення-виведення
CASE (Computer Software Engineering) – програмна інженерія з комп'ютерною підтримкою
CCD (Charge Coupled Device) – пристрій із зарядовим зв'язком
CE (Consumer Electronics) – побутова техніка
CGI (Common Gateway Interface) – загальний інтерфейс шлюза
CLI (Command Line Interface) – інтерфейс командного рядка
CPS (Characters Per Second) – символів за секунду

DARPA (Defense Advanced Research Projects Agency) – агентство передових оборонних дослідницьких проектів
DFD (Data Flow Diagrams) – діаграми потоків даних
DMA (Direct Memory Access) – прямий доступ до пам'яті
DPC (Deffered Procedure Call) – виклик відкладеної процедури
DPI (Dot Per Inch) – точок на дюйм
EDF (Earliest Deadline First) – «найближчий строк завершения – в першу чергу»
EFS (Encrypted File System) – шифрована файлова система
ERD (Entity-Relationship Diagrams) – діаграми «сутність-зв`язок»
FAT (File Allocation Table) – таблиця розміщення файлів
FIFO (First In Fist Out) – «Перший прийшов – першим обслужений»
GUI (Graphical User Interface) – графічний користувацький інтерфейс
HTML (HyperText Markup Language) – протокол передавання гіпертексту
ICAM (Integrated Computer Aided Manufacturing) – комплексно-автоматизоване виробництво
IDEF0 (ICAM DEFinition language 0) – Function Modeling — методология функционального моделирования
IIS (Internet Information Services) – інформаційні служби Інтернета
IPC (Inter Process Communications) – засоби міжпроцесної взаємодії
IRQL (Interrupt Request Level) – рівень запиту переривання
JVM (Java Virtual Machine) – інтерпретатор віртуальної машини Java
LCN (Logical Cluster Number) – логічний номер кластера
LOM (Lifecycle Objective Milestone) – об'єктивна контрольна точка життєвого циклу
LPS (Lines Per Second) – рядків за секунду
MBR (Master Boot Record) – програма головного завантажувального запису
MFT (Master File Table) – головна таблиця файлів
NTFS (New Technology File System) – файлова система нової технології
OCL (Object Constraint Language) – формальна мова
OSF (Open Software Foundation) – фонд відкритого програмного забезпечення
PCB (Process Control Block) – керуючий блок процесу
PDA (Personal Digital Assistant) – персональний цифровий помічник

PMT (PhotoMultiplier Tube) – фотомножник
POSIX – (Portable Operating System interface for unIX) – визначає портативний інтерфейс ОС на рівні вихідних текстів
RAD (Rapid Application Development) – модель швидкого розроблення додатків
RAID (Redundant Array of Inexpensive Disks) – надмірний масив недорогих дисків
RMS (Rate Monotonic Scheduling) – статичний алгоритм планування
ROM (Read Only Memory) – пам'ять тільки для читання
RPC (Remote Procedure Call) – виклик віддалених процедур
RPW (Rational Process Workbench) – спеціальний набір інструментів і шаблонів для настроювання й публікації Web-сайтів на основі RUP
RUP (Rational Unified Process) – методологія розроблення програмного забезпечення, створена компанією Rational Software
SADT (Structured Analysis and Design Technique) – методологія структурного аналізу і проектування
SIDM (Serial Impact Dot Matrix) – послідовні ударно-матричні принтери
SQL (Structured Query Language) – мова структурованих запитів
TPU (Turbo Pascal Unit) – модуль Турбо Паскаля
UML (Unified Modeling Language) – уніфікована мова моделювання
VCN (Virtual Cluster Number) – віртуальний номер кластера
VFS (Virtual File System) – віртуальна файлова система
WWW (World Wide Web) – Всесвітня Інформаційна Павутина

ПЕРЕДМОВА

Сучасна авіація використовує велику кількість складної техніки, зокрема, комп'ютерної. Це зумовлено збільшенням кількості повітряних об'єктів, їх швидкостей та підвищенням складності експлуатації повітряних кораблів. Відповідно зростають вимоги до точності та швидкодії технічних засобів, які використовуються в авіації, та підвищується навантаження на диспетчерів. Полегшити працю пілотів літальних апаратів, диспетчерів та інших працівників авіації покликане широке та системне використання інформаційних технологій та компютерів.

Дійсно, використання новітніх інформаційних технологій майже в будь-якій галузі, зокрема в авіації, дозволяє пришвидшити оброблення динамічних та статичних даних, надійно зберігати великі обсяги інформації, виконувати швидкі обчислення, покращити візуалізацію необхідної для прийняття рішень інформації, допомогти при аналізі ситуації й т.ін.

Серед чисельних шляхів застосування комп'ютерних технологій в авіації можна зокрема виділити такі: обрахунок польотної інформації, зберігання та оброблення авіаційних баз даних, прийняття рішень в експертних системах в авіації, геоінформаційні технології, відображення інформації в системах візуалізації інформації, віртуальне імітування роботи авіаційних систем у тренажерних системах. Різноманітність відповідних систем вимагає не лише від інженерів, але й від діспетчерів та пілотів не тільки практичних навичок роботи з цими системами, але й глибокого розуміння того, як ці системи розроблені та взаємодіють одна з одною, які концепції втілені в тому чи іншому програмному продукті.

Одним з перших кроків на шляху до вивчення комп'ютерної техніки є вивчення операційних систем (ОС). Операційні системи – це комплекс керувальних програм, який виконує завдання керування ресурсами системи та надає прикладним програмам середовище для їх виконання. Від швидкодії, зручності та надійності ОС залежить надійність та ефективність роботи кожної з програм, що мають працювати на комп'ютері. Тому для користувачів та розробників авіаційної техніки потрібно розуміти структуру та склад ОС, механізми їх роботи та керування процесами, що відбуваються в ОС.

Від розуміння роботи ОС можна перейти до розуміння роботи систем програмування. Системи програмування – це комплекс програмних засобів, призначених для кодування, тестування й налагодження програмного забезпечення. Розуміння сутності проектування програмного забезпечення є необхідною ланкою підготовки спеціаліста з інформаційного забезпечення, без знання основ про методи та системи проектування ускладнюється робота з налагодження існуючих програмних систем та створення нових.

Стрімкий розвиток інформаційних технологій, сфер їх використання, розвиток ОС, систем та методик програмування вимагає від сучасного спеціаліста не лише ґрунтовних знань конкретних програмних систем, а й розуміння загальних принципів їх побудови, уміння користуватись системами програмування для створення власних застосувань і просто більш ефективного використання вже існуючих.

Здобути відповідні знання студенти, які вивчають авіаційні дисципліни, зможуть опанувавши дисципліну «Операційні системи та системи програмування».

Навчальний посібник складається з чотирьох розділів, що відповідають модулям цієї дисципліни.

У першому розділі розглянуто сучасні ОС, які використовуються в авіації, класифікації та відмінність між ними; поняття ядра системи; концепції архітектури ОС, інтерфейсів, файлової системи; взаємодію з периферійними пристроями.

Другий розділ присвячено вивченню процесів і потоків, зокрема розглянуто багатозадачні та багатопотокові системи, алгоритми планування потоків та процесів, керування ресурсами, оперативною пам'яттю, файлами та зовнішними пристроями.

У третьому розділі розглянуто принципи систем програмування. Основну увагу приділено сучасним технологіям побудови програмного забезпечення – RUP, SADT, мові UML. Приділено увагу класифікації мов програмування, концепції архітектури програмного забезпечення та відповідно до кожної категорії програмним мовам системи програмування, їх структурі та засобам.

У четвертій частині описано прикладні програмні середовища, способи їх реалізації в ОС та шляхи їх використання для проектування програмних засобів оброблення інформаційно-технологічних процесів.

У процесі підготовки навчального посібника використано офіційні документи України, стандарти та рекомендовану практику Міжнародної організації цивільної авіації, відкриті відомості про ОС та технології побудови програмного забезпечення, а також результати авторських досліджень.

Кожний розділ навчального посібника закінчується запитаннями для самоперевірки.

Автори сподіваються, що вивчення та використання викладеного матеріалу буде сприяти підвищенню рівня підготовки бакалаврів з аеронавігації.

ВСТУП

Операційну систему можна розглядати як частину програмного забезпечення (ПЗ) персонального комп'ютера (ПК) або іншого пристрою, що керує взаємодією між технічними вузлами, пакетами прикладних програм та користувачем. Це програмне середовище, що керує ресурсами комп'ютера чи автоматизованої системи для виконання покладених на них завдань.

Більшість сучасних ПК для звичайних користувачів використовують обмежений набір найбільш популярних ОС. Водночас багато спеціалізованих технічних комплексів та систем, зокрема тих, які використовуються в авіації, можуть працювати під більш специфічними і менш відомими та популярними ОС, правила роботи з якими менш зрозумілі користувачам.

Сучасні комп'ютеризовані та автоматизовані системи підтримання функціонування аеронавігаційних систем разом з іншими автоматизованими системами, які ґрунтуються на комп'ютерних технологіях, використовують різні ОС, зокрема Windows-подібні та UNIX-подібні, однак досить часто застосовуються і системи, що ґрунтуються на DOS-командах або системах, подібних до Solaris, Xenix, OS/2 та ін.

Таким чином, можна дійти до висновку, що сучасному інженеру та користувачу авіаційних систем необхідно мати уявлення про принципи функціонування різних ОС для продуктивної роботи з ними. Метою цього посібника і є систематичне вивчення відповідних принципів функціонування ОС, ПЗ взагалі та навчання студентів навиків роботи з різними ОС і програмними середовищами. Відповідно до цієї мети посібник поділено на чотири розділи, що відповідають змістовним модулям дисципліни «Операційні системи та системи програмування».

У першому розділі вивчаються загальні теоретичні основи побудови та архітектури ОС, а також основи взаємодії ОС з апаратними засобами та периферією. Розглядається робота з файлами, механізми розподілу часу, поняття процесів і потоків. Вивчаються основи побудови інтерфейсу взаємодії ОС та користувачів.

У другому розділі вивчення ОС поглиблюється детальним описом механізмів функціонування ОС. Розглядаються поняття процесів та потоків, механізми їх створення, засоби планування та

диспетчеризація потоків, їх стани, програмування багатопотокових додатків, алгоритми планування потоків. Особливу увагу приділено методам планування процесів у системах реального часу. Вивчаються призначення й типи переривань, їх диспетчеризація в ОС, процедури оброблення переривань та системні виклики, а також конкретні принципи роботи з перериваннями у Windows та UNIX.

Розглядається також поняття ресурсу, класифікація ресурсів системи за різними ознаками, розподілення ресурсів і керування оперативною пам'яттю, файлами та зовнішніми пристроями.

Роботою з ОС не вичерпується вся робота з програмним забезпеченням – потрібно також застосовувати ОС для написання прикладних програм, керування ресурсами системи та розв'язання певних програмних задач. Для коректної та професійної роботи в цьому напрямі спеціалісту потрібно знати основи системного програмування. Цьому присвячено третій розділ, в якому надано класифікації мов програмування як об'єктно-орієнтованих так і інших. Розглянуто також концепції архітектури систем ПЗ, відповідні до кожної категорії програмних мов системи програмування, їх структуру та засоби.

Важливі розділи також приділені сучасним технологіям побудови ПЗ – Rational Unified Process (RUP), Structured Analysis and Design Technique (SADT). Знання цих систем та стандартів побудови ПЗ є необхідною складовою сучасного програміста та користувача ПЗ, оскільки осмислення та документальне оформлення розробки прикладних програм у сучасних засобах ускладнюється без відповідних знань.

В останньому четвертому розділі розглянуто прикладні програмні середовища, способи їх реалізації в ОС та їх використання для проектування програмних засобів оброблення інформаційно-технологічних процесів.

ТЕРМІНИ ТА ВИЗНАЧЕННЯ

Адресний простір – сукупність усіх ділянок оперативної пам'яті, виділених операційною системою для процесу.

Алгоритм планування – використовуваний алгоритм для планування.

Алгоритм планування без переключень (непріоритетний) – не потребує переривання за апаратним таймером, процес зупиняється тільки коли блокується або завершує роботу.

Алгоритм планування з переключеннями (пріоритетний) – вимагає переривання за апаратним таймером, процес виконується тільки у відведений період часу, після цього він зупиняється за таймером, щоб передати керування планувальнику.

Асиметрична операційна система – цілком виконується тільки на одному з процесорів системи, розподіляючи прикладні завдання по інших процесорах.

Атрибут – елемент даних класу, що інкапсулюється, тобто елемент даних, який міститься в об'єкті, що належить описуваному класу.

Багатозадачні операційні системи – підтримують одночасне існування декількох процесів, кожен з яких може мати тільки один потік.

Багатозадачність – здатність операційної системи виконувати декілька програм одночасно.

Багатопоточність – можливість програми бути багатозадачною.

Блокові пристрої – інформація зчитується і записується блоками, блоки мають свою адресу (диски).

Виклик – спеціальна стрілка, що вказує на іншу модель роботи. Стрілка виклику виходить з нижньої межі роботи.

Виконуючий код – готовий програмний продукт, який можна запустити; має розширення *.exe*, *.com*.

Вихід – матеріал або інформація, що отримується роботою. Кожна робота повинна мати хоча б одну стрілку виходу. Робота без результату не має сенсу і не повинна моделюватися.

Відкладений запис – принцип кешування, за якого дані, призначені для записування на диск, якийсь час зберігаються в кеші й лише у вільний від інших занять час зберігаються фізично.

Віртуальний номер кластера – номер кластера всередині певного файлу.

Вхід – матеріал або інформація, що використовується або перетворюється роботою для отримання результату (виходу).

Графопобудовник – пристрій, призначений для виведення даних в графічній формі на папір.

Група – це користувачі, об'єднані за якою-небудь ознакою, наприклад, за належністю до однієї розробки.

Дані – елементи інформації, що передаються між компонентами.

Дескриптор процесу – додаткова інформація, використовувана операційною системою для планування процесів: ідентифікатор процесу, стан процесу, дані про ступінь привілейованості процесу, місце перебування кодового сегмента й інша інформація.

Діаграма – графічне подання множини елементів. Найчастіше її зображують у вигляді зв'язного графу з вершинами (сутностями) і ребрами (відносинами); являє собою певну проекцію системи.

Діаграма послідовності – діаграма, на якій показано взаємодії об'єктів, упорядковані за часом їх прояву.

Драйвер – 1) сукупність програм, призначена для керування передаванням даних між зовнішнім пристроєм і оперативною пам'яттю; 2) програма, яка відповідає за функціонування певного пристрою, містить набір команд для цього пристрою і забезпечує зв'язок між комп'ютером та пристроєм.

Елементи керування – візуальний засіб для створення об'єктів на формі.

Іменоване значення – пара рядків «тег = значення», або «ім'я = вміст», у яких зберігається додаткова інформація про який-небудь елемент системи.

Ім'я змінної – це послідовність символів – літер англійського алфавіту і цифр – що починається з літери.

Інструментальна система – комплекс програмних або програмних і технічних засобів, який використовується фахівцями з програмування як інструмент для розроблення програмного забезпечення.

Інтерпретатор – транслятор, що забезпечує послідовний синхронний переклад і виконання кожного рядка програми, причому з

кожним запуском програми на виконання вся процедура повністю повторюється.

Інтерпретатор компілювального типу – система з компілятора, який переводить вихідний код програми в проміжне подання, наприклад, в байт-код або *p*-код, і власне інтерпретатора, який виконує отриманий проміжний код (віртуальна машина).

Інтерфейс в уніфікованій мові моделювання – повністю абстрактний клас (це означає, що не можуть бути створені екземпляри цього класу), що не має власних даних.

Інформаційна модель описує інформацію, яку за задумом має обробляти програмне забезпечення.

Зв’**язок** – абстрактний механізм, що забезпечує передавання даних, керування та взаємодію компонентів.

Каталог – файл, що містить службову інформацію файлової системи про групу файлів, що входять у цей каталог.

Керування – правила, стратегії, процедури або стандарти, якими керується робота.

Клас – суть, що описує множину об’єктів зі схожою структурою, поведінкою та зв’язками з іншими об’єктами.

Кластер – ціле число дискових секторів як мінімальний одиночний блок даних.

Кодування – переведення результатів проектування на текст мовою програмування.

Компілятор – комп’ютерна програма (або набір комп’ютерних програм), що перетворює (компілює) програмний код, написаний певною мовою програмування, у семантично еквівалентний код іншою мовою програмування.

Компонент – абстрактна одиниця програмного забезпечення, яка забезпечує оброблення інформації в системі з певним інтерфейсом.

Комп’ютерний принтер – пристрій для друкування інформації на папір.

Контекст процесу – інформація про стан операційного середовища, що відображається станом регістрів і програмного лічильника, режимом роботи процесора, вказівниками на відкриті файли, інформацією про незавершені операції введення/виведення, кодами помилок виконуваних даним процесом системних викликів і т.ін.

Контролер переривань – обслуговує переривання, що надходять від пристроїв.

Контрольна точка – запис у лог про успішне завершення попередніх транзакцій.

Критична область – частина програми, в якій є звернення до спільно використовуваних даних.

Лінія життя об'єкта – вертикальна лінія на діаграмі послідовності, яка вказує на існування об'єкта протягом певного періоду часу.

Логічний номер кластера – номер, починаючи з нуля, призначений файловою системою кожному кластеру.

Машинна мова – єдина мова, яку розуміє ЕОМ.

Механізм – ресурси, які виконують роботу (наприклад, персонал підприємства, верстати, пристрої і т.ін.).

Міжпрограмний інтерфейс – розподіл програмного забезпечення на декілька пов'язаних між собою рівнів.

Мініплотер – принтер з можливістю графічного дампа екрана.

Мова – набір правил, які визначають систему записів, що складають програму, синтаксис і семантику використовуваних граматичних конструкцій.

Мова програмування – формальна знакова система, призначена для записування програм, що задають алгоритм у формі, зрозу-мілій для виконавця (наприклад, комп'ютера). Мова програмування визначає набір лексичних, синтаксичних і семантичних правил, що використовуються для складання комп'ютерної програми.

Мовна машина – пристрій, який виконує послідовність пропозицій деякої нормалізованої мови.

Модель поведінки – фіксує бажану динаміку системи (режими її роботи).

Модем – пристрій, призначений для приєднання комп'ютера до звичайної телефонної лінії.

М'ютекс – одномісний семафор, що служить в програмуванні для синхронізації процесів і потоків, які виконуються одночасно; керує доступом до ресурсу.

Непривілейовані програмні модулі – звичайні програмні модулі, що можуть бути перервані під час своєї роботи.

Нерезидентний атрибут – атрибут, обсяг даних якого завеликий для зберігання в головній таблиці файлів, і там зберігається інформація про розміщення цих даних (номери кластерів, у яких розміщені дані).

Об'єкт – сутність, яка використовується під час виконання певної функції або операції (перетворення, оброблення, формування і т.ін.).

Об'єктний модуль (об'єктний файл) – файл з проміжним поданням окремого модуля програми, отриманий внаслідок обробленя вихідного коду компілятором.

Однозадачні операційні системи – виконують передусім функцію надання користувачу віртуальної машини, роблячи більш простим і зручним процес взаємодії користувача з комп'ютером.

Однократно використовувані програмні модулі – такі програмні модулі, що можуть бути правильно виконані тільки один раз; це означає, що в процесі виконання вони можуть пошкодити частину коду або вихідні дані, від яких залежить хід обчислень. Однократно використовувані програмні модулі є неподільним ресурсом.

Операція – суть, що визначає деяку дію, яка може бути виконана представником класу або з класом.

Операційне середовище – середовище виконання прикладних програм.

Операційна система – комплекс керувальних і оброблювальних програм, що виконує завдання керування ресурсами системи і надає прикладним програмам операційне середовище для їх виконання.

Переривання – основна рушійна сила будь-якої операційної системи.

Периферійні або зовнішні пристрої – пристрої, розміщені поза системним блоком і задіяні на певному етапі оброблення інформації.

Підсистема керування процесами – планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно працюючими в системі процесами, а також створює і знищує процеси, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами.

Планувальник – відповідальна за планування частина операційної системи.

Планування – забезпечення почергового доступу процесів до одного процесора.

Плотер – широкоформатний, найчастіше струминний принтер, зорієнтований на друкування аркушів формату A0, A1, A2, A3, A4 і т. ін. різної товщини.

Посилання (ярлик) – невеликий за обсягом файл, в якому міститься шлях до певного об'єкта; Під час зчитування з посилання або записування в нього створюється ілюзія роботи з файлом. Робота виконується з тим файлом, на який указує посилання.

Потік виконання або просто **потік** – абстракція, що являє собою виконання програми, яка розгортається в часі.

Привілейовані програмні модулі – працюють у привілейованому режимі, тобто коли вимкнено систему переривань (переривання закриті) таким чином, що ніякі зовнішні події не можуть порушити природний порядок обчислень. Програмний модуль виконується до кінця, після чого може бути викликаний на інше виконання.

Пріоритет – це числове значення, що характеризує ступінь привілейованості потоку з використанням ресурсів обчислювальної машини, зокрема процесорного часу: чим вищий пріоритет, тим вищий привілей, тим менше часу проводитиме потік у чергах.

Процедурне розроблення – описує послідовність дій в структурних компонентах, тобто визначає їх уміст.

Процес – абстракція, що являє собою програму під час її виконання.

Процесор – будь-який пристрій у складі ЕОМ, здатен автоматично виконувати прийнятні для нього дії (процесори, канали та пристрої, що працюють з каналами).

Процесор робочої станції (інколи називають графічними процесором) – комп'ютер (переважно 16-бітовий з ємністю пам'яті 256-768 кбайт) всередині кожної робочої станції, який допомагає головному комп'ютеру підвищити швидкість формування графічних зображень, використовуючи при цьому пам'ять головного комп'ютера.

Реалізація мови – системна програма, яка переводить (перетворює) запис мовою високого рівня в послідовність машинних команд.

Редактор зв'язків (збирач) – програма, що поєднує об'єктні модулі окремих частин програми й додає до них стандартні модулі підпрограм стандартних функцій (файли з розширенням *.lib*), які містяться в бібліотеках, що поставляють разом з компілятором у єдину програму готову до виконання, тобто створює *.exe* файл.

Резидентний атрибут – атрибут, дані якого збережені безпосередньо в записі головної таблиці файлів.

Рентабельні програмні модулі – допускають повторне багаторазове переривання свого виконання і повторний їх запуск зі звертанням з інших обчислювальних процесів.

Ресурси – 1) повторно використовувані, відносно стабільні й часто відсутні об'єкти, які запитуються, використовуються й звільняються процесами в період їх активності; 2) засоби обчислювальної системи, які виділяються для процесу на певний інтервал часу.

Розроблення даних – перетворення інформаційної моделі аналізу в структури даних для реалізації програмної системи.

Сегментний розподіл – розбиття адресного простору на «осмислені» частини.

Семафори – змінні для підрахунку сигналів запуску, збережених для подальшого використання.

Символьний зв'язок – файл даних, що містить ім'я файлу, з яким передбачається встановити зв'язок.

Символьні пристрої – пристрої, які зчитують і записують інформацію посимвольно.

Симетрична операційна система – повністю децентралізована операційна система, що використовує весь пул процесорів, розділяючи їх між системними й прикладними завданнями.

Система переривань – система, яка перемикає процесор на виконання потоку команд, відмінного від того, який виконувався до цього, з подальшим поверненням до вихідного коду.

Система програмування – система автоматичного програмування, що складається з мови програмування, компілятора або інтерпретатора програм, написаних цією мовою, та допоміжних засобів для підготовки програм до виконання.

Сканер – пристрій, який дає змогу вводити в комп'ютер чорно-біле або кольорове зображення, зчитувати графічну та текстову інформацію.

Сокет – підтримуваний ядром механізм, що приховує особливості середовища і дозволяє процесам одноманітно взаємодіяти як на одному комп'ютері, так і в мережі.

Спеціальний файл – фіктивний файл, що асоціюється з будь-яким пристроєм уведення-виведення, використовується для уніфікації механізму доступу до файлів і зовнішніх пристроїв.

Стан дії – спеціальний стан з деякою вхідною дією та принаймні одним переходом, що є виходом зі стану.

Стан змагання – ситуація, коли декілька процесів зчитують або записують дані (у пам'ять або файл) одночасно.

Стереотип – новий тип елемента моделі, який визначається на основі вже існуючого елемента.

Ступінь багатозадачності – кількість процесів, що перебувають у пам'яті.

Сумісність – можливість операційної системи виконувати додатки, написані для інших систем.

Супроводження – внесення змін в експлуатоване програмне забезпечення.

Текстовий редактор – редактор, що дозволяє набрати текст програми мовою програмування.

Термосублімація (**сублімація**) – швидке нагрівання барвника, коли пропускається рідка фаза.

Тестування – виконання програми з метою виявлення дефектів у функціях, логіці та формі реалізації програмного продукту.

Технологія розроблення програмного забезпечення – система інженерних принципів для створення економічного програмного забезпечення, яке надійно та ефективно працює в реальних коп'ютерах.

Транслятори – програми, що забезпечують переклад вихідного тексту програми машинною мовою (об'єктний код); бувають двох типів: інтерпретатори й компілятори.

Файлова система – частина операційної системи, призначення якої полягає в забезпеченні зручного інтерфейсу для роботи з даними, що зберігаються на диску, та сумісного використання файлів декількома користувачами і процесами.

Фокус керування – спеціальний символ на діаграмі послідовності, який вказує період часу, протягом якого об'єкт виконує деяку дію, перебуваючи в активному стані.

Функціональна модель – визначає перелік функцій оброблення.

Portable Operating System interface for unIX – визначає портативний інтерфейс операційної системи на рівні вихідних текстів. Основну специфікацію розроблено як *IEEE* 1003.1 та погоджено як міжнародний стандарт *ISO/IEC* 9945-1:1990. Найбільш поширені три стандарти: 1003.1a (*OS Definition*), 1003.1b (*Realtime Extensions*) та 1003.1c (*Threads*).

1. ДОСЛІДЖЕННЯ ОПЕРАЦІЙНИХ СИСТЕМ ТА ЇХ АРХІТЕКТУРА

1.1. Поняття, функції та історія операційних систем. Класифікація операційних систем за надаваними можливостями

1.1.1. Поняття операційної системи

Операційна система найбільшою мірою визначає вигляд обчислювальної системи в цілому. Незважаючи на це, користувачі, що активно використовують обчислювальну техніку, найчастіше зазнають труднощів при спробі дати визначення ОС. Частково це зумовлено тим, що ОС виконує дві по суті мало пов'язані функції: забезпечує користувачу-програмісту зручності за допомогою надання для нього розширеної машини й підвищує ефективність використання комп'ютера, раціонально керуючи його ресурсами.

Операційна система – комплекс керувальних і оброблювальних програм, що виконує завдання керування ресурсами системи, і надає прикладним програмам операційне середовище для їх виконання.

Дві основні функції ОС:

- розширення можливостей ЕОМ;
- керування її ресурсами.

Операційне середовище – середовище виконання прикладних програм.

Операційне середовище визначає для прикладних програм безліч команд процесора, які вони можуть використовувати, модель адресації й логічні структури адресного простору процесу, безліч системних викликів, доступних процесу і т.ін. Операційна система може підтримувати декілька різних операційних середовищ.

Ресурси – повторно використовувані, відносно стабільні й часто відсутні об'єкти, які запитуються, використовуються й звільняються процесами в період їх активності.

Ресурс може бути *поділюваним*, коли кілька процесів можуть його використовувати одночасно (у той самий момент часу) або паралельно (протягом деякого інтервалу часу процеси використовують ресурс поперемінно) і *неподільним*.

Існують *апаратні ресурси*, такі як процесорний час, оперативна пам'ять і дисковий простір; *програмні ресурси*, наприклад, бі-

бліотеки функцій; *інформаційні ресурси* – вміст файлів і баз даних; *ресурси операційного середовища* – структури, використовувані для виконання системних викликів наприклад (структура повідомлення); *інші типи ресурсів*. З погляду системи керування ресурсами, поняття ресурсу нівелювалося до рівня абстрактної структури з набором атрибутів, що характеризують методи доступу до цієї структури і її фізичне подання в системі.

Одним з основних понять, пов'язаних з ОС, є поняття *процесу*.

Процес – абстракція, що являє собою програму під час її виконання. Процес є споживачем різних ресурсів ОС, наприклад:

– адресний простір процесу містить його програмний код, дані й стек (або стеки);

– файли використовуються процесом для зчитування вхідних даних і запису вихідних;

– обладнання введення-виведення використовується відповідно до його призначення.

Безліч доступних процесу ресурсів і порядок їх використання визначаються архітектурою ОС. Зокрема, адресний простір процесу створюється в момент запуску програми на підставі інформації, отриманої ОС із вмісту програми й параметрів задання/запуску (якщо вони є). Залежно від архітектури обчислювального обладнання й ОС надаваний процесу адресний простір буде мати різні параметри (кількість адресних просторів, їх розмір, початкові й кінцеві адреси, способи адресації команд та даних і т.ін.).

Потік виконання або просто *потік* – абстракція, що являє собою виконання програми, яка розгортається в часі. Кожний процес має, принаймні, один потік. Потоки процесу розділяють його програмний код, глобальні змінні й системні ресурси, але кожний потік має власний програмний лічильник, власний вміст регістрів і власний стек. *Процес* являє собою сукупність взаємодійних потоків і виділених для нього ресурсів.

1.1.2. Операційна система як розширена машина

Використовувати більшість комп'ютерів на рівні машинної мови важко, особливо це стосується введення-виведення. Наприклад, для організації зчитування блока даних із гнучкого диска програміст може використовувати 16 різних команд, кожна з яких

вимагає 13 параметрів, таких як номер блока на диску, номер сектора на доріжці і т.ін. Коли виконання операції з диском завершується, контролер повертає 23 значення, що відображають наявність і типи помилок, які, мабуть, треба аналізувати. Навіть якщо не займатися реальними проблемами програмування введення-виведення, то серед програмістів найшлося б не багато бажаючих безпосередньо програмувати ці операції. Для роботи з диском програмісту-користувачу досить подавати його у вигляді деякого набору файлів, кожний з яких має ім'я. Робота з файлом полягає в його відкритті, зчитуванні або записуванні і закритті файлу. Питання чи використовувати у процесі записування вдосконалену частотну модуляцію або в якому стані перебуває двигун механізму переміщення зчитувальних головок, не повинні хвилювати користувача. Програма, яка приховує від програміста реальний стан апаратури й надає можливість простого, зручного перегляду файлів, зчитування або записування – це *ОС*. Так само, як ОС відділяє програмістів від апаратури дискового нагромаджувача і надає йому простий файловий інтерфейс, ОС оброблює переривання, керує таймерами й оперативною пам'яттю, а також виконує інші низькорівневі завдання. У кожному випадку та абстрактна, уявна машина, з якою завдяки ОС тепер може працювати користувач, набагато простіша й зручніша, ніж реальна апаратура, покладена в основу цієї абстрактної машини.

Отже функцією ОС є надання користувачу деякої розширеної або віртуальної машини, яку легше програмувати і з якою простіше працювати, ніж безпосередньо з апаратурою, що становить реальну машину.

1.1.3. Операційна система як система керування ресурсами

Ідея про те, що ОС насамперед система, яка забезпечує зручний інтерфейс користувачам, відповідає підходу «згори вниз». Підхід «знизу вгору» дає уявлення про ОС як про деякий механізм, що керує всіма частинами складної системи. Сучасні обчислювальні системи складаються із процесорів, пам'яті, таймерів, дисків, мереж комунікаційної апаратури, принтерів і іншого обладнання. Відповідно до другого підходу функцією ОС є розподіл процесорів,

пам'яті, обладнань і даних між процесами, що конкурують за ці ресурси.

Операційна система повинна керувати всіма ресурсами обчислювальної машини таким чином, щоб забезпечити максимальну ефективність її функціонування. Критерієм ефективності може бути, наприклад, пропускна здатність або реактивність системи. Керування ресурсами включає розв'язання двох загальних, незалежних від типу ресурсу, завдань:

– *планування ресурсу* – визначення кому, коли, а для поділюваних ресурсів – в якій кількості необхідно виділити цей ресурс;

– *відстеження стану ресурсу* – підтримання оперативної інформації про те, зайнятий чи не зайнятий ресурс, а для поділюваних ресурсів – яка кількість ресурсу вже розподілена, а яка вільна.

Для розв'язання цих загальних завдань керування ресурсами різні ОС використовують різні алгоритми, що і визначає їх вигляд в цілому, включаючи галузь застосування й користувацький інтерфейс. Так, наприклад, алгоритм керування процесором значною мірою визначає, чи є ОС системою поділу часу, системою пакетного оброблення чи системою реального часу.

1.1.4. Класифікація операційних систем

Операційні системи розрізняються особливостями реалізації внутрішніх алгоритмів керування основними ресурсами комп'ютера (процесорами, пам'яттю, обладнанням), особливостями використаних методів проектування, типами апаратних платформ, галузями використання й багатьма іншими властивостями.

Особливості алгоритмів керування ресурсами. Залежно від особливостей використаного алгоритму керування процесором ОС поділяють на багатозадачні й однозадачні, багатокористувацькі й однокористувацькі, на системи, що підтримують багатониткове оброблення і що не підтримують його, на багатопроцесорні й однопроцесорні системи.

Підтримка багатозадачності. За кількістю одночасно виконуваних завдань ОС можна поділити на два класи: *однозадачні* (наприклад, MS-DOS) і *багатозадачні* (OS/2, UNIX, Windows xx та ін.).

Однозадачні ОС виконують здебільшого функцію надання користувачу віртуальної машини, роблячи більш простим і зручним процес взаємодії користувача з комп'ютером.

Рис. 1.1. Однозадачна ОС

Однозадачні ОС (рис. 1.1) розраховані на підтримку тільки одного процесу в кожний момент часу. Цей єдиний процес може мати тільки один потік. Програми можна запускати тільки послідовно – до завершення виконання процесу не можна створювати ще один процес.

Однозадачні ОС включають в себе засоби керування периферійним обладнанням, засоби керування файлами, засоби спілкування з користувачем. У ході виконання процесу завдання однозадачної ОС зводиться до підтримки системних викликів.

Багатозадачні ОС підтримують одночасне існування декількох процесів, кожний з яких може мати тільки один потік (рис. 1.2).

Рис. 1.2. Декілька процесів (у кожного – один потік)

м спільно використовуваивна пам'ять, файли й зовється перехід виконання між потоками різних процесів, для чого потрібно перемикати контекст процесу й контекст потоку. Виконуваний потік змінює ядро, перед зміною активного потоку відбувається перемикання в контекст ядра (змінюється при цьому контекст процесу або контекст потоку залежно від ОС).

Багатозадачні операційні системи з підтримкою багатопоточності. У таких ОС до одного процесу можуть належати декілька потоків виконання команд (рис. 1.3). Усі потоки одного процесу розділяють його ресурси, наприклад, адресний простір або відкриті файли, проте характеризуються власним апаратним контекстом.

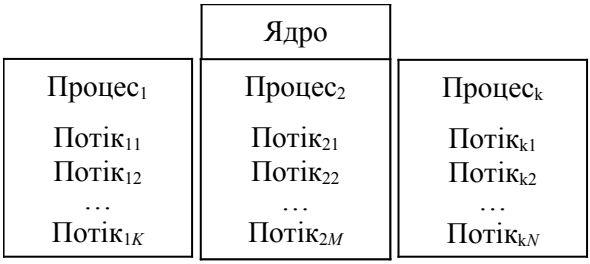


Рис. 1.3. Декілька процессів (у кожного – декілька потоків)

У такій системі може відбуватися перехід виконання від одного потоку процесу до іншого потоку того ж процесу. У цьому випадку не потрібно перемикати контекст процесу, відповідно таке перемикання проводиться швидше, ніж у випадку перемикання між процесами.

Залежно від керування ресурсами власником або користувачем кожного типу ресурсів може бути або процес, або потік.

Підтримка багатокористувацького режиму. За кількістю одночасно працюючих користувачів ОС поділяють на *однокористувацькі* (MS-DOS, Windows 3.x) та *багатокористувацькі* (UNIX, Windows NT та ін.).

Головною відмінністю багатокористувацьких систем від однокористувацьких є наявність засобів захисту інформації кожного користувача від несанкціонованого доступу інших користувачів. Слід зазначити, що не кожна багатозадачна система є багатокористувацькою, і не кожна однокористувацька ОС є однозадачною.

Витісняльна і невитісняльна багатозадачність. Найважливішим поділюваним ресурсом є процесорний час. Спосіб розподілу процесорного часу між декількома одночасно наявними в системі процесами (або нитками) багато в чому визначає специфіку ОС. Серед варіантів реалізації багатозадачності можна виокремити дві групи алгоритмів:

– *невитісняльна багатозадачність* (Netware, Windows 3.x);

– *витісняльна багатозадачність* (Windows NT, OS/2, UNIX).

Основною відмінністю між варіантами витісняльної та невитісняльної багатозадачності є ступінь централізації механізму планування процесів. У першому випадку механізм планування процесів цілком зосереджений в ОС, а в другому – розподілений між ОС і

прикладними програмами. За алгоритму невитісняльної багатозадачності активний процес виконується доти, доки він сам, за власною ініціативою, не віддасть керування ОС для того, щоб та вибрала з черги інший готовий до виконання процес. За алгоритму витіснянльної багатозадачності рішення про перемикання процесора з одного процесу на інший, приймається ОС, а не активним процесом.

Підтримка багатонитковості. Важливою властивістю ОС є можливість розпаралелювання обчислень у межах одного завдання. *Багатониткова ОС* розподіляє процесорний час не між завданнями, а і між їхніми окремими галузями (нитками).

Багатопроцесорне оброблення. Загальноприйнятим є введення в ОС функцій підтримки багатопроцесорного оброблення даних. Такі функції є в ОС Solaris 2.x фірми Sun, Open Server 3.x компанії Santa Crus Operations, OS/2 фірми IBM, Windows NT фірми Microsoft і Netware 4.1 фірми Novell.

Багатопроцесорні ОС можна класифікувати за способом організації обчислювального процесу в системі з багатопроцесорною архітектурою: асиметричні ОС і симетричні ОС.

Асиметрична ОС виконується тільки на одному із процесорів системи, розподіляючи прикладні завдання по інших процесорах.

Симетрична ОС повністю децентралізована й використовує весь пул процесорів, розподіляючи їх між системними й прикладними завданнями.

Вище були розглянуті характеристики ОС, пов'язані з керуванням тільки одним типом ресурсів – *процесором*. На вигляд ОС в цілому та на можливості її використання в тій або іншій галузі впливають особливості й інших підсистем керування локальними ресурсами – підсистем керування пам'яттю, файлами, обладнанням введення-виведення.

Специфіка ОС проявляється й у тому, яким чином вона реалізовує мережеві функції: розпізнавання й перенапрямлення у мережу запитів до вилучених ресурсів, передавання повідомлень по мережі, виконання вилучених запитів. У ході реалізації мережевих функцій виникає комплекс завдань, пов'язаних з розподіленим характером зберігання й оброблення даних у мережі: ведення довідкової інформації про всі доступні в мережі ресурси й сервери, адресація взаємодійних процесів, забезпечення прозорості доступу, тиражування даних, узгодження копій, підтримка безпеки даних.

Особливості апаратних платформ. На властивості ОС безпосередньо впливають апаратні засоби, на які вона орієнтована. За типом апаратури розрізняють ОС ПК, мінікомп'ютерів, мейнфреймів, кластерів та мереж комп'ютерів. Серед цих типів комп'ютерів можуть бути як однопроцесорні варіанти, так і багатопроцесорні. У кожному разі специфіка апаратних засобів зазвичай відображається на специфіці ОС.

Очевидно, що ОС великої машини є більш складною й функціональною, ніж ОС ПК. Так, в ОС великих машин функції планування потоку виконуваних завдань, мабуть, реалізуються шляхом використання складних пріоритетних дисциплін і потребують більшої обчислювальної потужності, ніж в ОС ПК. Аналогічно виконуються й інші функції.

Мережева ОС має у своєму складі засоби передавання повідомлень між комп'ютерами по лініях зв'язку, які не потрібні в автономній ОС. На основі цих повідомлень мережева ОС підтримує розділення ресурсів комп'ютера між віддаленими користувачами, підключеними до мережі. Для підтримки функцій передавання повідомлень мережеві ОС містять спеціальні програмні компоненти, що реалізують популярні комунікаційні протоколи, такі як IP, IPX, Ethernet та ін.

Багатопроцесорні системи вимагають від ОС особливої організації, за допомогою якої сама ОС, а також підтримувані нею додатки, могли б виконуватися паралельно окремими процесорами системи. Паралельна робота окремих частин ОС створює додаткові проблеми для розробників ОС, оскільки в цьому випадку набагато складніше забезпечити погоджений доступ окремих процесів до загальних системних таблиць, виключити ефект гонок та інші небажані наслідки асинхронного виконання робіт.

Інші вимоги ставляться до ОС кластерів.

Кластер – слабко поєднана сукупність декількох обчислювальних систем, що працюють спільно для виконання спільних додатків, що подаються користувачу єдиною системою.

Поряд зі спеціальною апаратурою для функціонування кластерних систем необхідна й програмна підтримка з боку ОС, яка зводиться в основному до синхронізації доступу до поділюваних ресурсів, виявлення відмов і динамічної реконфігурації системи.

Однією з перших розробок у галузі кластерних технологій були рішення компанії Digital Equipment на базі комп'ютерів VAX. Ця компанія уклала угоду з корпорацією Microsoft про розроблення кластерної технології, що використовує Windows NT. Кілька компаній пропонують кластери на основі Unix-Машин.

Поряд з ОС, орієнтованими на певний тип апаратної платформи, існують ОС, спеціально розроблені таким чином, щоб їх можна було легко переносити з комп'ютера одного типу на комп'ютер іншого типу – *мобільні* ОС. Найбільш яскравим прикладом такої ОС є популярна система UNIX. У цих системах апаратнозалежні місця ретельно локалізовані, тому під час перенесення системи на нову платформу переписуються тільки вони. Засобом, що полегшує перенесення іншої частини ОС, є написання її машиннонезалежною мовою, наприклад, С, яку і було розроблено для програмування ОС.

Особливості галузей використання. Багатозадачні ОС підрозділяють на три типи відповідно до використовуваних для їх розроблення критеріїв ефективності:

– *системи пакетної обробки* (ОС ЕС);
– *системи розділення часу* (UNIX, VMS);
– *системи реального часу* (QNX, RT/11).

Системи пакетного оброблення призначались насамперед для розв'язання обчислювальних задач, що не потребують швидкого отримання результатів. Головною метою й критерієм ефективності систем пакетного оброблення є максимальна пропускна здатність, тобто вирішення максимальної кількості завдань за одиницю часу. Для досягнення цієї мети в системах пакетного оброблення використовується така схема функціонування: на початку роботи формується пакет завдань, кожне завдання містить вимогу до системних ресурсів; із цього пакета завдань формується мультипрограмна суміш, тобто множина одночасно виконуваних завдань. Для одночасного виконання вибираються завдання, що потребують різних ресурсів для забезпечення збалансованого завантаження всіх блоків обчислювальної машини; так, наприклад, у мультипрограмній суміші бажана одночасна наявність обчислювальних завдань і завдань із інтенсивним уведенням-виведенням. Таким чином, вибір нового завдання з пакета завдань залежить від внутрішньої ситуації, що складається в системі, тобто вибирається «вигідне» завдання. Отже, у таких ОС не-

можливо гарантувати виконання того або іншого завдання протягом певного періоду часу. У системах пакетного оброблення процесор з виконання одного завдання перемикається на виконання іншого тільки в тому випадку, якщо активне завдання саме відмовляється від процесора, наприклад, через необхідність виконання операції введення-виведення. Тому одне завдання може надовго зайняти процесор, що робить неможливим виконання інтерактивних завдань. Таким чином, взаємодія користувача з обчислювальною машиною, на якій встановлено систему пакетного оброблення, зводиться до того, що він завдання подає диспетчеру-оператору, а після виконання всього пакета завдань отримує результат. Очевидно, що такий порядок знижує ефективність роботи користувача.

Системи розділення часу покликано усунути основний недолік систем пакетного оброблення – ізоляцію користувача-програміста від процесу виконання його завдань. Кожному користувачу системи розділення часу надається термінал, з якого він може вести діалог зі своєю програмою. Оскільки в системах розділення часу для кожного завдання виділяється тільки квант процесорного часу, жодне завдання не займає процесор надовго, і час відповіді виявляється прийнятним. Якщо квант обирається досить малим, то всі користувачі, що одночасно працюють на одній і тій же машині, вважають, що кожен з них одноосібно використовує машину. Зрозуміло, що системи розділення часу мають меншу пропускну здатність, ніж системи пакетного оброблення, тому що до виконання приймається кожне запущене користувачем завдання, а не те, яке «вигідне» системі, і, крім того, є накладні витрати обчислювальної потужності на більш часте перемикання процесора із завдання на завдання. Критерієм ефективності систем розділення часу є не максимальна пропускна спроможність, а зручність і ефективність роботи користувача.

Системи реального часу застосовуються для керування різними технічними об’єктами, такими як, наприклад, супутник, наукова експериментальна установка, або технологічними процесами, такими, як гальванічна лінія, доменний процес і т.ін. Для цих випадків існує граничнодопустимий час, протягом якого має бути виконана та або інша програма, що керує об’єктом, а якщо ні, то може статися аварія: супутник вийде із зони видимості, експериментальні дані, що надходять з датчиків, будуть загублені, товщина гальванічного покриття не буде відповідати нормі. Таким чином, критерієм ефектив-

ності для систем реального часу є їх здатність витримувати заздалегідь задані інтервали часу між запуском програми й отриманням результату (керувальний вплив). Цей час називають *часом реакції системи*, а відповідна властивість системи – *реактивністю*. Для цих систем мультипрограмна суміш являє собою фіксований набір заздалегідь розроблених програм, а програма до виконання вибирається виходячи з поточного стану об'єкта або відповідно до розкладу планових робіт.

Деякі ОС можуть поєднувати в собі властивості систем різних типів, наприклад, частину завдань можна виконувати в режимі пакетного оброблення, а частину – у режимі реального часу або в режимі розділення часу. В таких випадках режим пакетного оброблення часто називають фоновим режимом.

Особливості методів побудови. Описуючи ОС, часто вказують особливості її структурної організації й основні концепції, покладені в її основу. Розглянемо такі базові концепції.

Способи побудови ядра системи – монолітне ядро або мікроядерний підхід. Більшість ОС використовує *монолітне ядро*, яке компонується як одна програма, що працює в привілейованому режимі, що використовує швидкі переходи з однієї процедури на іншу, які не вимагають перемикання з привілейованого режиму в користувацький і навпаки. Альтернативою є побудова ОС на базі *мікроядра*, що працює також у привілейованому режимі й виконує тільки мінімум функцій з керування апаратурою, в той час, як функції ОС більш високого рівня виконують спеціалізовані компоненти ОС – *сервери*, що працюють у користувацькому режимі. За такої побудові ОС працює більш повільно, оскільки часто виконуються переходи між привілейованим режимом і користувацьким, зате система виходить більш гнучкою – її функції можна нарощувати, модифікувати або звужувати, крім серверів користувацького режиму. Крім того, сервери добре захищені один від одного, як і будь-які користувацькі процеси.

Побудова ОС на базі об'єктно-орієнтованого підходу дає можливість використовувати всі його переваги, що добре зарекомендували себе на рівні додатків, усередині ОС, а саме: акумуляцію вдалих рішень у формі стандартних об'єктів, можливість створення нових об'єктів на базі наявних за допомогою механізму спад-

кування, захист даних за рахунок їх інкапсуляції у внутрішні структури об'єкта, що робить дані недоступними для несанкціонованого використання ззовні, структурованість системи, що складається з набору добре визначених об'єктів.

Наявність декількох прикладних середовищ дає змогу у межах однієї ОС одночасно виконувати додатки, розроблені для кількох ОС. Багато сучасних ОС підтримують одночасно прикладні середовища MS-DOS, Windows, UNIX (POSIX), OS/2 або хоча б деякої підмножини із цього широковживаного набору. Концепція множинних прикладних середовищ просто реалізується в ОС на базі мікроядра, з яким працюють різні сервери, частина яких реалізовує прикладне середовище тієї або іншої ОС.

Розподілена організація ОС дозволяє спростити роботу користувачів і програмістів у мережевих середовищах. У розподіленій ОС реалізовано механізми, які дають можливість користувачу уявляти й сприймати мережу як традиційний однопроцесорний комп'ютер. Характерними ознаками розподіленої організації ОС є: наявність єдиної довідкової служби поділюваних ресурсів, єдиної служби часу, використання механізму виклику віддалених процедур (RPC) для прозорого розподілу програмних процедур по машинах, багатониткового оброблення, що дозволяє розпаралелювати обчислення в межах одного завдання і виконувати це завдання відразу на декількох комп'ютерах мережі, а також наявність інших розподілених служб.

1.2. Огляд сучасних операційних систем. Головні особливості цільової операційної системи

1.2.1. Огляд сучасних операційних систем

Операційні системи мейнфреймів. На верхньому рівні міститься ОС для *мейнфреймів*. Ці комп'ютери розміром з кімнату все ще можна знайти в центрах даних великих корпорацій. Мейнфрейми відрізняються від ПК можливостями введення-виведення. Досить часто трапляються мейнфрейми з тисяччю дисків і терабайтами даних, а ПК з такими параметрами здався б дійсно незвичайним. Мейнфрейми немовби повертаються у вигляді потужних

web-серверів, серверів для великомасштабних електронно-комерційних сайтів і серверів для транзакцій в бізнесі.

Операційні системи для мейнфреймів насамперед орієнтовані на оброблення одночасно безлічі завдань, більшість з яких потребує численних операцій введення-виведення. Зазвичай вони пропонують три види обслуговування: пакетне оброблення, оброблення транзакцій (групові операції) й розділення часу. *Пакетне оброблення* є системою, що виконує стандартні завдання без присутності користувачів, які працюють в інтерактивному режимі. Оброблення позовів в страхових компаніях або складання звітів про продажі для мережі магазинів – це типові завдання, що обробляються в пакетному режимі. *Системи оброблення транзакцій* керують дуже великою кількістю дрібних запитів, наприклад контролюють процес роботи в банку або бронювання авіаквитків. Кожен окремий запит невеликий, але система повинна відповідати на сотні або тисячі запитів за секунду. *Системи розділення часу*, дозволяють безлічі видалених користувачів одночасно виконувати завдання на одній машині. Наочним прикладом є робота з великою базою даних. Всі ці функції тісно пов'язані між собою, і часто ОС мейнфрейму виконує їх. Прикладом ОС для мейнфрейму є OS/390 на базі OS/360.

Серверні операційні системи. Рівнем нижче розміщено *серверні ОС*. Вони працюють на серверах, які є або дуже великими ПК, або робочими станціями, або навіть мейнфреймами. Вони одночасно обслуговують безліч користувачів і дозволяють їм ділити між собою програмні та апаратні ресурси. Сервери дають змогу працювати з друкувальними пристроями, файлами або з Інтернетом. Інтернет-провайдери зазвичай запускають у роботу декілька серверів для підтримання одночасного доступу до мережі безлічі клієнтів. На серверах зберігаються сторінки web-сайтів і обробляються вхідні запити. UNIX, Linux і Windows 2000 і вище є типовими серверними ОС.

Багатопроцесорні операційні системи. Найбільшого поширення набув спосіб збільшення потужності комп'ютерів, що полягає в з'єднанні декількох центральних процесорів в одній системі. Залежно від виду з'єднання процесорів і розділення роботи такі системи називаються паралельними комп'ютерами, мультикомп'ютерами або багатопроцесорними системами. Вони потребують

спеціальних ОС, але часто такі ОС є варіантами серверних ОС із спеціальними можливостями зв'язку.

Операційні системи для персональних комп'ютерів. Операційні системи для ПК надають зручний інтерфейс для одного користувача. Такі системи широко використовують для роботи з текстом, електронними таблицями і доступу до Інтернету. Найбільш поширені – це Windows 98, Windows 2000, Windows 7, ОС комп'ютера Macintosh і Linux. Насправді багато людей навіть не знають про існування інших видів ОС, окрім тієї, якою вони користуються.

Операційні системи реального часу. Ще один вид ОС – це *системи реального часу*. Головним параметром таких систем є час. Часто такі процеси мають задовольняти жорсткі часові вимоги. Якщо деяка дія має відбутися в конкретний момент часу (або в заданому діапазоні часу), це буде *жорстка система реального часу*.

Існує також *гнучка система реального часу*, для якої допустимі пропуски термінів виконання операції, що трапляються час від часу. Це зокрема цифрові аудіо- і мультимедійні системи. Системи Vx Works і QNX є добре відомими ОС реального часу.

Убудовані операційні системи. Наступним кроком від величезних систем до менших є кишенькові комп'ютери і вбудовані системи. Кишеньковий комп'ютер або PDA (Personal Digital Assistant – персональний цифровий помічник) – це малий за розміром комп'ютер, що виконує невеликий набір функцій (телефонного записника та блокнота). *Убудовані системи*, призначені для керування діями пристроїв, працюють на машинах, що зазвичай не вважаються комп'ютерами, наприклад в телевізорах, мікрохвильових печах і мобільних телефонах. Їх характеристики часто такі самі, що й систем реального часу, але при цьому вони мають особливий розмір, пам'ять і обмеження потужності, що виділяє їх в окремий клас. Прикладами таких ОС є PALM OS і Windows CE (Consumer Electronics – побутова техніка).

Операційні системи для смарт-карт. Найменші ОС працюють на *смарт-картах*, які є пристроями розміром з кредитну карту, що містять центральний процесор. На такі ОС накладаються жорсткі обмеження потужності процесора і пам'яті. Деякі з них можуть керувати тільки однією операцією, наприклад електронним платежем, але інші ОС на тих же самих смарт-картах виконують складні функції. Часто вони є патентованими системами.

Деякі смарт-карти є Java-орієнтованими. Це означає, що постійний запом'ятовувальний пристрій (ПЗП, або ROM – Read Only

Memory – пам'ять тільки для читання) смарт-карт містить інтерпретатор віртуальної машини Java (JVM – Java Virtual Machine). Аплети Java (маленькі програми) завантажуються на карту і виконуються JVM-інтерпретатором. Деякі з таких карт можуть одночасно керувати декількома аплетами Java, що приводить до багатозадачності й необхідності планування. За одночасної роботи двох і більше програм виникає потреба в керуванні ресурсами і захистом. Відповідно всі ці завдання виконує примітивна ОС, розміщена у смарт-карті.

1.2.2. Головні особливості цільової операційної системи

Операційна система Unix. Системи Unix розроблялися різними виробниками, тому доцільно розглянути історію створення сім'ї цих ОС.

У 1968 р. консорціум дослідників фірм General Electric, AT&T Bell Labs і Массачусетського технологічного інституту завершив роботу над науково-дослідним проектом Multics, результатом якого стала операційна система, яка увібрала до свого складу останні досягнення у вирішенні проблем багатозадачності, керування файлами та взаємодії з користувачем. У 1969 р. Кен Томпсон розробив ОС Unix, у якій використано багато результатів проекту Multics. Він пристосував цю систему, призначену для роботи на міні-ЕОМ, до потреб дослідників. І з самого початку Unix стала зручною, ефективною, розрахованою на багато користувачів і багатозадачною ОС.

З часом популярність Unix в Bell Labs зростала, і в 1970 р. Денніс Рітчі і Кен Томпсон переписали код системи мовою програмування С. Денніс Рітчі, колега Томпсона з Bell Labs, створив цю мову для забезпечення гнучкості під час розроблення програм. Одна з переваг мови С полягає в тому, що вона дозволяє звертатися безпосередньо до апаратних засобів комп'ютера за допомогою узагальненого набору команд. До цього текст програми ОС потрібно було спеціально переписувати апаратно-залежною мовою Assembler для кожного типу комп'ютера. Мова С дозволила Рітчі та Томпсону написати всього одну версію ОС Unix, яку потім можна було компілювати С-компіляторами на різних машинах. Операційна си-

стема Unix стала *мобільною*, тобто здатною працювати на різних типах машин без перепрограмування.

Поступово Unix стала стандартним програмним продуктом, який поширюється багатьма фірмами, включаючи Novell та IBM. Спочатку цю ОС вважали дослідним продуктом, тому перші версії розповсюджувалися безкоштовно по факультетах обчислювальної техніки багатьох відомих університетів. У 1972 р. Bell Labs почала випускати офіційні версії Unix і продавати ліцензії на неї різним користувачам. Одним з таких користувачів був факультет обчислювальної техніки Каліфорнійського університету в Берклі. Його фахівці ввели в систему багато нових особливостей, які згодом стали стандартними. У 1975 р. у Берклі була випущена власна версія системи, відома як Berkeley Software Distribution (BSD). Ця версія Unix стала основним суперником версії AT. За нею послідувала System V, яка стала важливим підтримуваним програмним продуктом.

Паралельно випускалися версії BSD. Наприкінці 70-х років BSD Unix стала основою дослідницького проекту, що виконувався в Агентстві перспективних досліджень і розробок (DARPA) міністерства оборони США. У результаті в 1983 р. Каліфорнійський університет випустив потужну версію системи під назвою BSD 4.2 Вона включала в себе досить досконалу систему керування файлами і мережеві засоби, засновані на використанні протоколів TCP/IP, що застосовуються в Інтернеті. Версія BSD 4.2 набула поширення і була обрана багатьма фірмами-виробниками, зокрема Sun Microsystems.

Поширення різних версій Unix зумовило потребу у розробленні стандарту на цю ОС. Іншого способу дізнаватися про те, в яких версіях будуть працювати призначені для використання в цьому середовищі програми, у розробників ПЗ не було. В середині 80-х років з'явилися два конкуруючі стандарти: один був створений на основі версії AT. У 1991 р. Unix System Laboratories розробила System V версії 4, в якій реалізовано майже всі можливості варіантів попередньої версії BSD версії 4.3, SunOS і Xenix. У відповідь кілька компаній, зокрема, IBM і Hewlett-Packard, створили фонд відкритого програмного забезпечення (Open Software Foundation, OSF), метою якого стало розроблення власної стандартної версії Unix. У результаті з'явилися два конкуруючі комерційні стандартні

варіанти: версія OSF і System V версії 4. У 1993 р. компанія AT&T продала свою частку прав на Unix фірмі Novell, і деякий час Unix Systems Laboratories належала до Novell. За цей час фірма випустила власні версії Unix на базі System V версії 4 під загальною назвою UnixWare, призначені для взаємодії із системою NetWare розробки Novell.

Протягом свого розвитку Unix залишалася великою і вимогливою до апаратних засобів ОС, для ефективної роботи якої необхідна робоча станція або міні-ЕОМ. Деякі версії ОС були розраховані в основному на робочі станції. Те, що ця ОС установлюється на комп'ютерах всіх типів (робочих станціях, міні-ЕОМ і навіть супер-ЕОМ), є свідченням її мобільності, що забезпечила можливість ефективної версії Unix для ПК.

Операційна система Linux. Найпоширенішим проектом системи Unix кінця XX ст. стала альтернатива дорогим рішенням – ОС Linux. Натепер темпи освоєння ринку цією системою найбільш інтенсивні порівняно з іншими відомими ОС.

Створення цієї системи починалося з розроблення проекту Лінуса Торвальда – студента факультету обчислювальної техніки Гельсінкського університету. У той час студенти користувалися програмою Minix, яка демонструвала різні можливості Unix. Ця програма, розроблена професором Ендрю Таннебаумом, поширилася по мережі Інтернет серед студентів усього світу.

Лінус поставив за мету створити ефективну ПК-версію Unix для користувачів Minix. Він назвав її Linux і в 1991 р. випустив версію 0.11. Система широко розповсюдилася по Інтернету і в наступні роки була допрацьована іншими програмістами, які ввели до неї можливості та особливості, притаманні стандартним системам Unix. Зокрема, було перенесено всі основні програми-менеджери вікон. У цій ОС використовуються утиліти Інтернет, є і повний набір засобів розроблення програм, включаючи компілятори і налагодник С. Незважаючи на такі широкі можливості, ОС Linux залишається невеликою, стабільною й швидкодійною. У мінімальній конфігурації вона може ефективно працювати навіть на 386 комп'ютерах за ємності оперативної пам'яті 4 Мбайт.

Сильною стороною Linux є її універсальність. Система покриває весь діапазон застосувань: від настільного ПК до надпотужних багатопроцесорних серверів і кластерів.

Linux виконує ті ж функції, що й DOS і Windows, однак відрізняється від них особливою потужністю й гнучкістю. Більшість ОС ПК створювалися для невеликих ПК, що мали обмежені можливості і лише нещодавно перетворилися на універсальні машини. Такі ОС постійно модернізуються, щоб відповідати можливостям апаратних засобів ПК, які безперервно розвиваються. Linux же розроблялася в зовсім іншому контексті.

Розроблення вихідної для Linux системи Unix полягало в створенні продукту, який міг би задовольняти співробітників, що займаються різними дослідженнями. Операційна система розглядалася як механізм, що надає користувачу набір високоефективних інструментів. Така орієнтація на користувача означала можливість конфігурації і програмування системи відповідно до конкретних потреб. У випадку з Linux ОС дійсно стала операційним середовищем.

З фінансового погляду Linux має одну істотну перевагу: вона є не комерційною, і на відміну від ОС Unix поширюється за генеральною відкритою ліцензією GNU в межах фонду безкоштовного ПЗ, тому ця ОС доступна для всіх. GNU складена таким чином, що Linux залишається безкоштовною і водночас стандартизованою системою – існує лише один офіційний її варіант.

Апаратні потреби для Linux – мінімальні (рекомендовані):

- пам’ять: 4 Мбайт (32Мбайт);
- процесор: 80386 (IP-166МГц) або сумісний;
- вінчестер: 100 Мбайт (600 Мбайт).

Операційні системи сім’ї Windows. *Перші версії Windows.* Перша версія Windows вийшла в світ наприкінці 80-х років і залишилася абсолютно непоміченою. Аналогічна доля спіткала і наступну версію – лише версія Windows 3.0 (1992) зуміла прокласти собі дорогу і стати «продуктом року». А ще через два роки були випущені версії 3.1 і 3.11, які остаточно укріпили динамічні позиції Windows. Остання включала повну підтримку мультимедіа і роботу в локальній мережі – тому й отримала уточнюючу назву Windows For Workgroups.

Апаратні потреби для Windows 3.1 – мінімальні (рекомендовані):

- пам’ять: 1 Мбайт (4 Мбайт);
- процесор: 80286 (80386) або сумісний;

– вінчестер: 20 Мбайт (80 Мбайт).

Апаратні потреби для Windows 3.11 – мінімальні (рекомендовані):

– пам'ять: 2 Мбайт (8 Мбайт);
– процесор: 80386 (80486) або сумісний;
– вінчестер: 40 Мбайт (100 Мбайт).

Покоління 9X. Windows 95. Нова ОС, мала була вийти ще в 1994 р. – саме тоді з'явилися офіційні повідомлення про завершення розроблення нової ОС, що отримала назву Chicago. Однак термін представлення «Чикаго» постійно відкладався, корпорація Microsoft робила щоразу обнадійливі заяви. Зрештою у серпні 1995 р. Windows 95 вийшла в світ.

Більше того – нова ОС стала 32-розрядною. Усі попередні версії DOS і Windows були 16-розрядними і, отже, не могли повною мірою використовувати можливості навіть процесорів сім'ї 386 і, тим паче, нових процесорів Pentium. Звичайно в цьому полягали й деякі незручності – спеціально під Windows 95 користувачам довелося замінювати Windows-програми на нові 32-розрядні версії. Однак на практиці перехід виявився порівняно легким – уже за рік з'явилися нові версії програмних продуктів.

Windows 95 отримала абсолютно новий графічний інтерфейс – більш елегантний, зручний для користувача і зовні привабливий порівняно з попередніми ОС.

Windows 98 і Windows 98SE. До роботи над новою версією Windows корпорація Microsoft приступила відразу ж після виходу Windows 95. Нова ОС очікувалася наприкінці 1996 р. і мала називатися Memphis. Але цього не сталося ні в 1996 р., ні в 1997 р. Тільки 25 червня 1998 р. нова ОС Microsoft надійшла до магазинів. А приблизно через місяць вийшла у світ і російськомовна версія Windows 98.

Основні зміни торкнулися інтерфейсу – тепер «Робочий стіл» Windows 98 став ще красивішим, а головне – він повністю інтегрований із середовищем Інтернет. У новій ОС остаточно була стерта відмінність між файлами і папками на комп'ютері та об'єктами Всесвітньої інформаційної павутини (WorldWideWeb). Основний засіб роботи з файлами та папками в обох випадках – програма Internet Explorer.

Інша важлива відмінність Windows 98 від Windows 95 полягає в розширених можливостях керування інтерфейсом. Але є і більш важ-

ливі зміни – у внутрішній будові ОС. Хоча основна «начинка» ОС залишилася колишньою, Windows 98 вигравала у попередньої ОС за рахунок коректної роботи з новими комплектуючими – процесором Pentium II, графічним портом AGP, шиною USB, новими моделями відеокарт, материнських плат, модемів і т.ін. Нарешті Windows 98 містила велику кількість нових програм і утиліт – в першу чергу повний комплект ПЗ для роботи в Інтернеті та утиліту конвертації файлової системи FAT16 у більш нову версію FAT32.

Наприкінці 1999 р. у продажу з'явилася російськомовна версія нового комплекту Windows 98 – Windows 98 SE. Від попередньої версії нова Windows відрізнялась тим, що до її складу включено п'яту версію браузера Internet Explorer, оновлено систему з'єднання з Інтернетом, а також зроблено численні виправлення помилок і є нова бібліотека драйверів пристроїв.

Windows ME (Microsoft Windows Millennium Edition) – остання еволюція ОС класу Windows 95 – Windows 98, запущена в серійне виробництво в 2000 р. Російська локалізація Windows ME з'явилася на ринку ПЗ у листопаді 2000 р.

Windows ME значно відрізнялась від сім'ї системних платформ Windows 9X, передусім тим, що в цій реалізації Windows зовсім не підтримується MS DOS – коректно запустити на комп'ютері, що працює під керуванням цієї системи, деякі програми DOS – досить складне завдання. Windows ME тісно інтегрована з Internet Explorer 5.0, що зробило її ще більше ресурсомісткою, в комплект поставки за замовчуванням включено більшу частину елементів Microsoft Plus для Windows 98, базовий набір ігор розширено новими програмами, що дозволяють користувачу «грати» в мережі Інтернет з живими суперниками, додано Windows Media Player 7.0, що підтримує відтворення файлів безлічі нових аудіо- та відеоформатів. Інтерфейс Windows ME майже повністю збігається із зовнішнім оформленням Windows 2000 Professional, включаючи системні іконки й оновлене діалогове вікно вимкнення/перезавантаження комп'ютера, але майже всі базові елементи налаштування Windows 98 збереглися на своїх колишніх місцях.

Windows ME дійсно стала останньою ОС сім'ї Windows 9X, оскільки всі наступні ОС Windows як для домашніх комп'ютерів, так і для робочих станцій, створювалися на платформі NT.

Апаратні потреби для Windows ME – мінімальні (рекомендовані):

– пам'ять: 32 Мбайт (64 Мбайт);
– вінчестер: 500 Мбайт.

Покоління NT. Windows NT (New Technology). 32-розрядна Windows NT, перша версія якої з'явилася на ринку в 1993-му, а остання – у 1998 р., із самого початку створювалася як надстабільна, надійна система, розрахована передусім на роботу. І в цьому сенсі Windows 98/ME може їй тільки позаздрити: випадки помилок, крахів і «зависання» під час роботи у Windows NT траплялися вкрай рідко. Відбувалося це тому, що у Windows NT розроблено надійне розділення програм, які працюють під її керуванням, що не дає їм «змагатися» за ресурси. У Windows 3.1/95/98/ME кожна із завантажених програм відчувала себе в оперативній пам'яті повновладним господарем. Нерідко програми перезавантажували процесор запитами на ресурси, у результаті чого ОС «зависала».

На відміну від Windows 98/ME Windows NT забороняє беззаперечний доступ до ресурсів комп'ютера будь-яким програмам. Це дозволяє системі уникнути конфліктів, проте в результаті під NT відмовляються працювати будь-які програми, написані для DOS, і багато створених для Windows 95.

Слід враховувати і той факт, що велика частина роботи з NT є лише в мережевому режимі роботи, тобто разом з іншими комп'ютерами.

Windows 2000. Вона з'явилася на ринку на початку 2000 р. Операційна система Microsoft Windows 2000 являє собою друге покоління ОС, побудованих за архітектурою Windows NT. Вона випускається в трьох модифікаціях: Windows 2000 Professional для ноутбуків, настільних систем і робочих станцій, Windows Server 2000 для серверних комп'ютерів і Windows 2000 Datacenter Server для великих серверних систем, робочих станцій великих корпоративних мереж та спеціалізованих банківських і файлових серверів.

Завдяки використанню удосконаленої технології NT, що поєднується з об'єктивною простотою інтерфейсу Windows 9X, Windows 2000 має високу надійність і стабільність, також вона значно легше піддається налаштуванню та конфігурації, ніж попередні версії Windows. Розмежування доступу до системи реалізовано на високому рівні, що дозволяє забезпечити безпеку збереження да-

них на дисках, якщо за комп'ютером працює більше ніж один користувач. Windows 2000 була визнана однією з найкращих, і досі використовується на багатьох комп'ютерах, незважаючи на вихід більш нових версій ОС Windows.

Windows XP. Операційна система Microsoft Windows XP (від англ. EXPerience – досвід) відома також під кодовим найменуванням Microsoft Codename Whistler. Спочатку корпорація Microsoft планувала розробити дві незалежні ОС нового покоління. Перший проект отримав робочу назву Neptune, ця ОС мала б стати черговим оновленням Windows ME, новою системою лінійки Windows 9X. Другий проект, що мав назву Odyssey, передбачав створення ОС на платформі Windows NT, яка повинна змінити Windows 2000. Проте керівництво Microsoft визнало недоцільним розосереджувати ресурси на просування двох різних ОС, унаслідок чого обидва напрями розробок були об'єднані в один проект – Microsoft Whistler. Можливо, саме завдяки цьому Windows XP поєднує в собі переваги ОС попередніх поколінь: зручність, простоту в інсталяції та експлуа-тації ОС сім'ї Windows 98 і Windows ME, а також надійність і багатофункціональність Windows 2000. Натепер Windows XP для настільних ПК і робочих станцій випускається в трьох модифікаціях: Home Edition для домашніх ПК, Professional Edition – для офісних ПК і, нарешті, Microsoft Windows XP 64bit Edition – це версія Windows XP Professional для ПК, складених на базі 64-бітного процесора Intel Itanium з тактовою частотою понад 1 ГГц.

Апаратні потреби для Windows XP, мінімальні:

- пам'ять: 64 Мбайт;
- процесор: Pentium-сумісний, тактова частота від 233 МГц;
- вільний дисковий простір: 1,5 Гбайт.

Windows.NET. Операційна система MS Windows.NET – це сім'я серверних ОС, розроблених корпорацією Microsoft на основі Windows XP, які змінили Windows 2000 Server, Advanced Server і Datacenter Server. Windows.NET поставляється у варіантах Windows.NET Server, Windows.NET Advanced Server і Windows.NET Datacenter Server. Відповідно технічні можливості цих версій ОС розрізняються: наприклад, Windows.NET Server може адресувати чотирипроцесорні системи, Windows.NET Advanced Server працює з восьмипроцесорними комп'ютерами, а Windows.NET Datacenter Server підтримує машини,

апаратна конфігурація яких включає до 32 синхронно працюючих процесорів.

Windows Vista. Ця версія Windows вийшла восени 2006 р., хоча бета- і піратські версії стали з'являтися ще з кінця 2005 р. Усього випущено сім варіантів Windows Vista, які можна розбити на дві групи – Home і Business.

Windows 7. Компанія Microsoft випустила нову ОС Windows 7. У Windows 7 є можливість вимкнення або ввімкнення браузера Internet Explorer і програвача Windows Media Player. Також ОС має підтримку multitouch-моніторів.

Функція Branch Cache дозволяє зменшити затримки у користувачів, що працюють з комп'ютером віддалено. Наприклад, файл доступний в мережі, кешується локально, тому він скачується вже не з віддаленого сервера, а з локального комп'ютера. Ця функція може працювати в двох режимах – Hosted Cache і Distributed Cache. У першому випадку файл зберігається на виділеному локальному сервері під керуванням Windows Server 2008 R2, у другому – на комп'ютері у клієнта.

Функція ReadyBoost дозволяє використовувати флеш-нагромаджувач як додаткову кеш-пам'ять для прискорення роботи системи.

Операційна система Windows CE. Ця ОС відрізняється від інших хоча б тому, що вона призначена винятково для встановлення на кишенькові комп'ютери (palm-top). Такі міні комп'ютери, що з'явилися наприкінці 90-х років, усього за кілька років зуміли набути поширення. Сьогодні «електронними органайзерами» користуються і ділові люди, які постійно перебувають у роз'їздах, і студенти.

У невеликій ОС інтегровані всі необхідні програми для роботи з міні комп'ютером – простий текстовий редактор, записна книжка, електронна таблиця і система електронної пошти. Власники ПК навряд чи зіткнуться з цією ОС, а от власники різноманітних побутових пристроїв – цілком можливо. За задумом Microsoft, Windows CE незабаром буде встановлюватися навіть на бортові комп'ютери деяких моделей автомобілів. Нині на ринку наладонних комп'ютерів Windows CE не є лідером, поступаючись PalmOS та іншим конкуруючим продуктам.

Огляд деяких комерційних операційних систем реального часу. Основними відмінностями операційних систем реального часу (ОСРЧ) від ОС загального призначення є:

– орієнтація на оброблення зовнішніх подій;

– детермінований час реакції на зовнішню подію;

– модульна організація;

– невеликий розмір системи.

Операційна система OS-9. Операційна система OS-9 фірми Microware належить до класу UNIX – подібних ОСРЧ. По суті OS-9 є багатозадачною ОС із пріоритетною витісняльною диспетчеризацією. Ця ОС допускає можливість багатокористувацької роботи. Об'єктно-орієнтовний модульний дизайн системи дозволяє конфігурувати систему в дуже широкому діапазоні від убудованих систем, до великих мережевих додатків. Відповідно до цієї концепції всі функціональні компоненти OS-9 (ядро, ієрархічні файлові менеджери, драйвера пристроїв і т.ін.), реалізовані у вигляді незалежних модулів. Усі модулі ОС позиційно-незалежні й можуть бути розміщені в ПЗП, а також видалятися із системи в процесі її функціонування без якої-небудь повторної інсталяції або перекомпонування. Ядро забезпечує основний системний сервіс, включаючи керування процесами й розподіл ресурсів.

Основні характеристики:

1. Архітектура: на основі мікроядра.
2. Стандарт: власний, виклики схожі на UNIX.

Властивості як ОСРЧ:

– багатозадачність: багатопроцесність;

– багатопроцесорність;

– рівні пріоритетів: 65535;

– час реакції: 3 мкс;

– планування: пріоритетне, FIFO, спеціальний механізм планування; preemptive ядро.

3. Операційна система розроблення (host): UNIX/Windows.
4. Процесори (target): Motorola 68xxx, Intel 80×86, ARM, MIPS, PowerPC.
5. Лінії зв'язку (host-target): послідовний канал і ethernet.
6. Мінімальний розмір: 16 кбайт.
7. Засоби синхронізації й взаємодії: поділювана пам'ять, сигнали, семафори, події.

Операційна система VxWorks. VxWorks належить до ОС «твердого» реального часу. Характерною особливістю цієї ОС є те, що завдяки її розвиненим мережевим можливостям усі розроблен-

ня ПЗ виконуються на інструментальному комп'ютері (хост-системі) з використанням крос-засобів для наступного виконання на цільовій машині під керуванням VxWorks.

Відмітна ознака системи − можливість керувати роботою складних комплексів реального часу й бортових пристроїв, що використовують процесорні елементи різних постачальників. Три основні компоненти цієї ОСРЧ утворюють єдине інтегроване середовище: власне ядро системи, що керує процесором; набір засобів міжпроцесорної взаємодії; комплект комунікаційних програм для роботи з Ethernet або послідовними каналами зв'язку.

Основні характеристики:

1. Архітектура: монолітна.
2. Стандарт: власний і POSIX 1003.
3. Властивості як ОСРЧ:

– багатозадачність: багатопроцесність;
– багатопроцесорність;
– рівні пріоритетів: 256;
– час реакції: 4 мкс;
– час перемикання контексту: 15 мкс;
– планування: пріоритетне; preemptive ядро.

4. Операційна система розроблення (host): UNIX/Windows.
5. Процесори (target): Motorola 68xxx, Intel 80×86, Intel 80960, PowerPC, SPARC, Alpha, MIPS, ARM.
6. Лінії зв'язку host-target: послідовний канал, ethernet, шина VME.
7. Мінімальний розмір: 22 кбайт.
8. Засоби синхронізації й взаємодії: семафори POSIX 1003, черги, сигнали.

Операційна система QNX. Операційну систему QNX канадської компанії QNX Software System Ltd. побудовано на основі ієрархічної мікроядерної архітектури.

Мікроядро QNX виконує такі функції:

– міжпроцесорний обмін;
– низькорівневий мережевий обмін;
– диспетчеризація завдань;
– низкорівневе оброблення переривань.

Основні характеристики:

1. Архітектура: на основі мікроядра.
2. Стандарт: POSIX 1003.

3. Властивості як ОСРЧ:

– багатозадачність: POSIX 1003 (багатопроцесність і багатозадачність);

– багатопроцесорність;

– рівні пріоритетів: 32;

– час реакції: 4,3 мкс;

– час перемикання контексту: 13 мкс;

– планування: FIFO, round robin, адаптивне; preemptive ядро.

4. Процесори (target): Intel 80×86.

5. Мінімальний розмір: 60 кбайт.

6. Засоби синхронізації й взаємодії: POSIX 1003 (семафори, mutex, condvar).

Операційна система LynxOS. Система LynxOS випускається фірмою Lynx Real Time Systems (Los Gatos, USA). ОСРЧ із клону UNIX-систем, що забезпечує детермінований час відгуку за запитами.

Основні характеристики:

1. Архітектура: на основі мікроядра.

2. Стандарт: POSIX 1003.

3. Властивості як ОСРЧ:

– багатозадачність POSIX 1003 (багатопроцесність та багатозадачність);

– багатопроцесорність;

– рівні пріоритетів: 255;

– час реакції: 7 мкс;

– час перемикання контексту: 17 мкс;

– планування: FIFO, round robin, Quantum, preemptive ядро.

4. Процесори (target): Intel 80×86, Motorola 68xxx, SPARC, PowerPC.

5. Мінімальний розмір:

– повної системи: 256 кбайт.

– зрізаної системи: 124 кбайт.

– тільки ядра: 33 кбайт.

Систему можна записати в ROM.

6. Засоби синхронізації і взаємодії: POSIX 1003 (семафори, mutex, condvar).

Операційна система pSOS. Система pSOS випускається Integrated Systems (Santa Clara, USA).

Основні характеристики:

1. Архітектура: на основі мікроядра.
2. Стандарт: власний.
3. Властивості як ОСРЧ:
– багатозадачність: багатопроцесність;
– багатопроцесорність;
– рівні пріоритетів: 255;
– час реакції: 4 мкс;
– час перемикання контексту: 12мкс;
– планування: пріоритетне; preemptive ядро.
4. Операційна система розроблення (host): UNIX/Windows.
5. Процесори (target): Motorola 68xxx, Intel 80×86, Intel 80960, ARM, MIPS, PowerPC.
6. Мінімальний розмір: 15 кбайт.
7. Засоби синхронізації й взаємодії: семафори, mutex, події і т.ін.

1.3. Архітектура та структура операційної системи

1.3.1. Архітектура операційної системи

Монолітна система. Структуру системи показано на рис. 1.4.

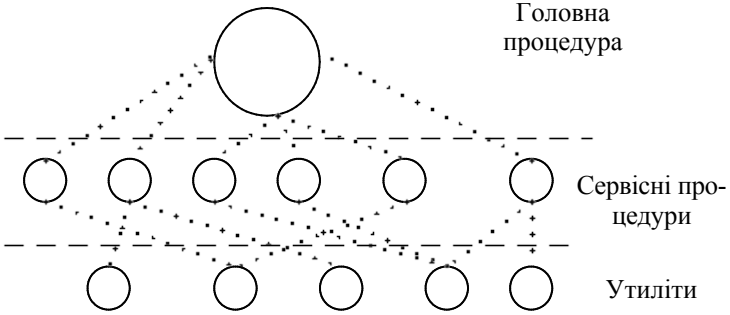


Рис. 1.4. Проста модель монолітної системи

Монолітна система має такі рівні:
1) головна програма, яка викликає необхідні сервісні процедури;
2) набір сервісних процедур, що реалізують системні виклики;
3) набір утиліт, які обслуговують сервісні процедури.

У цій моделі для кожного системного виклику є одна сервісна процедура (наприклад, читати з файлу). Утиліти виконують функції,

які потрібні декільком сервісним процедурам (наприклад, для зчитування й записування файлу необхідна утиліта роботи з диском).

Етапи оброблення виклику:

– приймається виклик;

– виконується перехід з режиму користувача в режим ядра;

– параметри виклику перевіряються ОС для визначення системного виклику;

– після цього ОС звертається до таблиці, яка містить посилання на процедури, та викликає відповідну процедуру.

Багаторівнева структура операційної системи. Узагальненням монолітної системи є організація ОС як ієрархії рівнів (рис. 1.5). Рівні утворюються групами функцій ОС, такими, як файлова система, керування процесами та пристроями і т.ін. Кожний рівень може взаємодіяти тільки з безпосереднім сусіднім рівнем – вищим або нижчим. Прикладні програми або модулі самої ОС передають запити вгору і вниз за цими рівнями.

Рівні	Функції				
7	Оброблювач системних викликів				
6	Файлова система 1		Файлова система *n*		
5	Віртуальна пам’ять				
4	Драйвер 1	Драйвер 2	…	…	Драйвер *n*
3	Керування потоками				
2	Оброблення переривань, керування пам’яттю				
1	Приховування апаратури низького рівня				

Рис. 1.5. Приклад структури багаторівневої системи

Переваги: Висока продуктивність.

Недоліки:

– великий код ядра і, як наслідок, великий вміст помилок;

– ненадійний захист ядра від допоміжних процесів.

Приклад реалізації багаторівневої моделі UNIX показано на рис. 1.6 і 1.7.

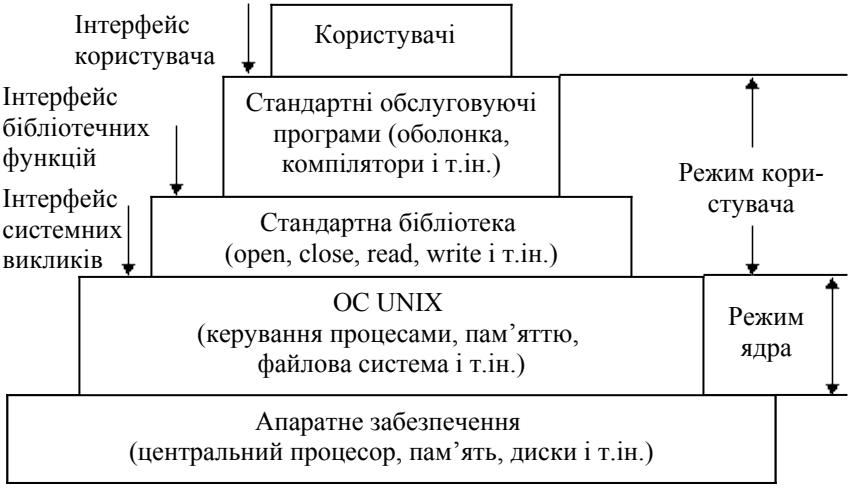


Рис 1.6. Структура ОС UNIX

<table>
<tr><td colspan="3">Системні виклики</td><td colspan="4">Апаратні та емульовані переривання</td></tr>
<tr><td colspan="2">Керування терміналом</td><td>Сокети</td><td>Найменування файлу</td><td>Відображення адрес</td><td>Сторінкові переривання</td><td rowspan="2">Оброблення сигналів</td><td rowspan="2">Створення та</td></tr>
<tr><td rowspan="2">Необроблений телетайп</td><td>Оброблений телетайп</td><td>Мережеві протоколи</td><td>Файлові системи</td><td colspan="2">Віртуальна пам'ять</td></tr>
<tr><td>Дисципліни лінії зв'язку</td><td>Маршрутизація</td><td>Буферний кеш</td><td colspan="2">Драйвери мережевих пристроїв</td><td colspan="2">Планування процесів</td></tr>
<tr><td colspan="2">Символьні пристрої</td><td>Драйвери мережевих пристроїв</td><td colspan="3">Драйвери дискових пристроїв</td><td>Диспетчеризація процесів</td></tr>
<tr><td colspan="7">Апаратура</td></tr>
</table>

Рис 1.7. Ядро ОС UNIX

Приклад реалізації багаторівневої моделі Windows показано на рис. 1.8.

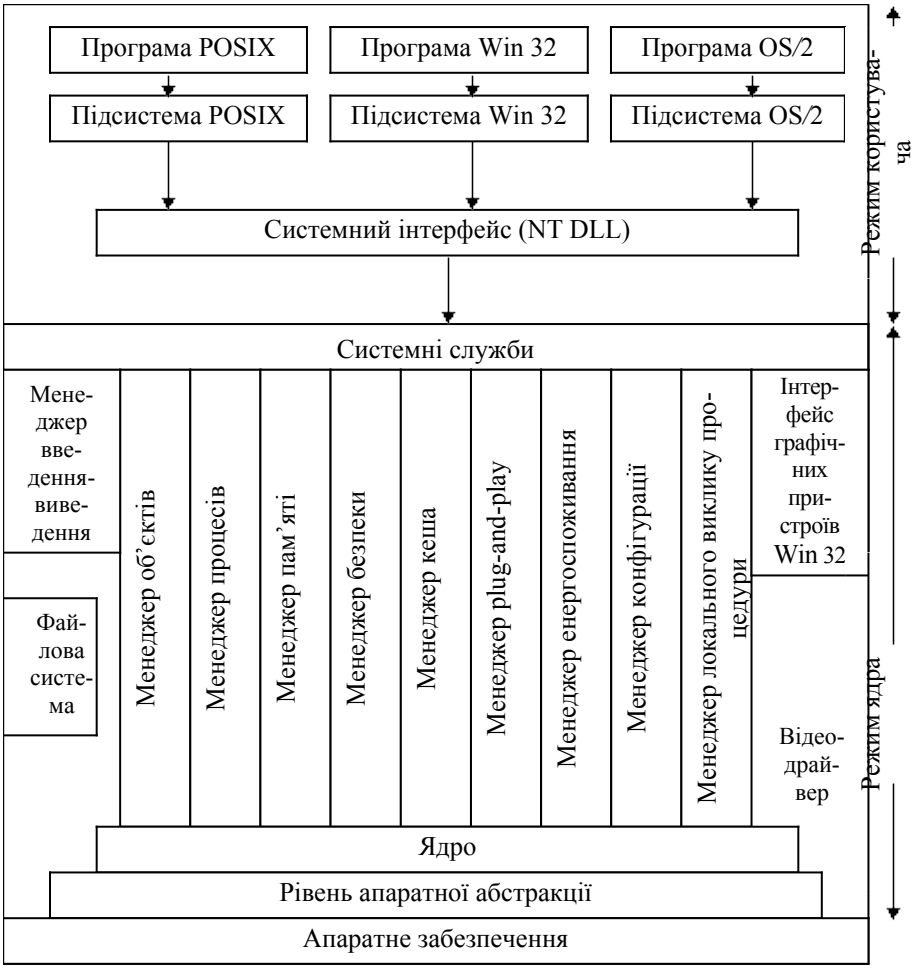


Рис 1.8. Структура ОС Windows 2000

1.3.2. Модель екзоядра

Якщо попередні моделі виконували максимум функцій, то принцип екзоядра – усі функції передати користувачу програм. Тобто кожна призначена для користувача програма зможе мати свою файлову систему. Така ОС повинна забезпечити безпечний розподіл ресурсів серед тих користувачів, що змагаються за них.

1.3.3. Мікроядерна архітектура

Концепція. Мікроядерна архітектура є альтернативою класичному способу побудови ОС. Під класичною архітектурою в цьому випадку розумють структурну організацію ОС, відповідно до якої основні функції ОС, що складають багатошарове ядро, виконуються в привілейованому режимі. При цьому деякі допоміжні функції ОС оформлюються у вигляді додатків і виконуються в користувацькому режимі поряд зі звичайними користувацькими програмами (стаючи системними утилітами або обробними програмами). Кожний додаток користувацького режиму працює у власному адресному просторі й захищений тим самим від втручання інших додатків. Код ядра, виконуваний у привілейованому режимі, має доступ до ділянок пам'яті всіх додатків, але сам повністю від них захищений. Додатки звертаються до ядра із запитами на виконання системних функцій.

Суть мікроядерної архітектури полягає в такому. У привілейованому режимі залишається працювати тільки дуже невелика частина ОС, названа мікроядром (рис. 1.9). Мікроядро захищене від інших частин ОС і додатків. До складу мікроядра входять машиннозалежні модулі, а також модулі, що виконують базові функції ядра з керування процесами, оброблення переривань, керування віртуальною пам'яттю, пересилання повідомлень і керування пристроями введення-виведення, пов'язані із завантаженням або зчитуванням регістрів пристроїв. Набір функцій мікроядра зазвичай відповідає функціям шару базових механізмів звичайного ядра. Такі функції ОС важко, або навіть неможливо, виконати в просторі користувача.

Усі інші більш високорівневі функції ядра оформлюються у вигляді додатків, що працюють у користувацькому режимі. Однозначного рішення про те, які із системних функцій потрібно залишити в привілейованому режимі, а які перенести в користувацький, немає. У загальному випадку багато менеджерів ресурсів, що є невід'ємними частинами звичайного ядра − файлова система, підсистеми керування віртуальною пам'яттю й процесами, менеджер безпеки − стають «периферійними» модулями, що працюють у користувацькому режимі.

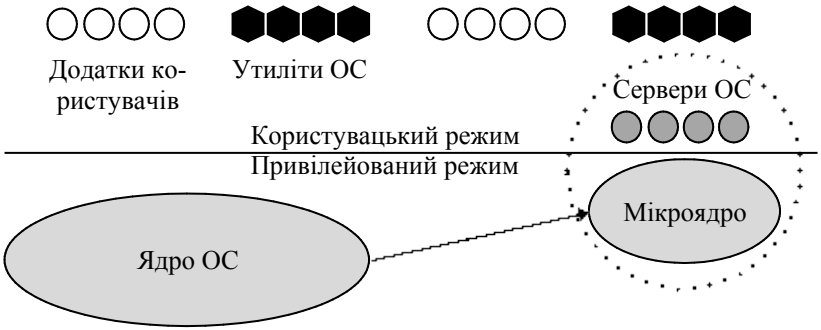


Рис. 1.9. Перенесення основного обсягу функцій ядра в користувацький простір

Працюючі в користувацькому режимі менеджери ресурсів мають принципові відмінності від традиційних утиліт і обробних програм ОС, хоча за мікроядерної архітектури всі ці програмні компоненти також оформлені у вигляді додатків. Утиліти й обробні програми викликаються загалом користувачами. Ситуації, коли один додаток потребує виконання функції (процедури) іншого додатка, виникають украй рідко. Тому в ОС із класичною архітектурою немає механізму, за допомогою якого один додаток міг би викликати функції іншого. Коли ж у формі додатка оформлюється частина ОС, то за визначенням основним призначенням такого додатка є обслуговування запитів інших додатків, наприклад створення процесу, виділення пам'яті, перевірка прав доступу до ресурсу і т.ін. Саме тому менеджери ресурсів, винесені в користувацький режим, називаються серверами ОС, тобто модулями, основним призначенням яких є обслуговування запитів локальних додатків та інших модулів ОС. Очевидно, що для реалізації мікроядерної архітектури необхідною умовою є наявність в ОС зручного й ефективного способу виклику процедур одного процесу з іншого. Підтримка такого механізму і є однією з головних функцій мікроядра. Схематично механізм звертання до функцій ОС, що оформлені у вигляді серверів, показано на рис. 1.10.

Клієнт, яким може бути або прикладна програма, або інший компонент ОС, запитує виконання якоїсь функції у відповідного сервера, посилаючи йому повідомлення. Безпосереднє передавання повідомлень між додатками неможливе, оскільки їхні адресні про-

стори ізольовані один від одного. Мікроядро, що виконується в привілейованому режимі, має доступ до адресних просторів кожного з цих додатків і тому може працювати як посередник. Мікроядро спочатку передає повідомлення, яке містить ім'я й параметри викликуваної процедури, потрібне серверу; потім сервер виконує запитану операцію, після чого ядро повертає результати клієнту за допомогою іншого повідомлення. Таким чином, робота мікроядерної ОС відповідає відомій моделі клієнт−сервер, у якій роль транспортних засобів виконує мікроядро.

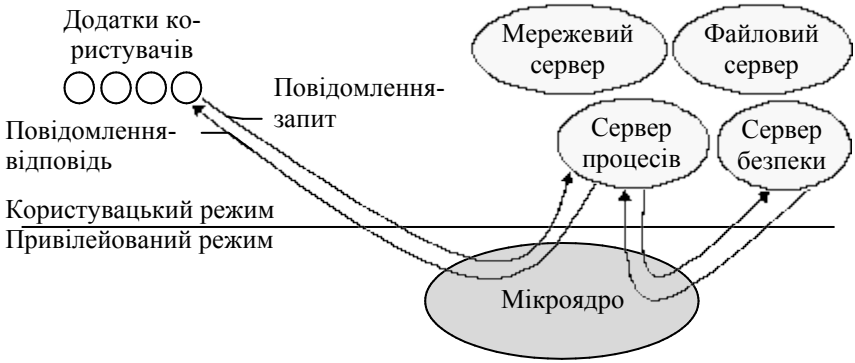


Рис. 1.10. Реалізація системного виклику в мікроядерній архітектурі

Переваги й недоліки мікроядерної архітектури. Операційні системи, засновані на концепції мікроядра, великою мірою задовольняють більшість вимог, поставлених до сучасних ОС, маючи переносимість, розширюваність, надійність і створюючи належні передумови для підтримання розподілених додатків. За ці переваги доводиться поступатися зниженням продуктивності, і це є основним недоліком мікроядерної архітектури.

Високий ступінь переносимості зумовлюється тим, що весь машиннозалежний код ізольований у мікроядрі, тому для перенесення системи на новий процесор потрібно менше змін, і всі вони логічно згруповані.

Надвисока розширюваність властива мікроядерній ОС. У традиційних системах навіть за наявності багатошарової структури нелегко вилучити один шар і поміняти його на інший через множинність і розмитість інтерфейсів між шарами. Додавання нових функцій і зміна існуючих потребують відповідного знання ОС та вели-

ких витрат часу. Водночас обмежений набір чітко визначених інтерфейсів мікроядра дає змогу впорядковано збільшуватись і розвиватися ОС. Додавання нової підсистеми потребує розроблення нового додатка, що ніяк не стосується цілісності мікроядра. Мікроядерна структура дозволяє не тільки додавати, але й скорочувати кількість компонентів ОС, що також буває дуже корисно. Наприклад, не всім користувачам потрібні засоби безпеки або підтримки розподілених обчислень, а видаляти їх із традиційного ядра найчастіше неможливо. Зазвичай традиційні ОС дозволяють динамічно додавати в ядро або видаляти з ядра тільки драйвери зовнішніх пристроїв − через часті зміни в конфігурації підключених до комп'ютера зовнішніх пристроїв підсистема введення-виведення ядра допускає завантаження й вивантаження драйверів «на ходу», але для цього вона розробляється особливим способом (наприклад, середовище STREAMS в UNIX або менеджер уведення-виведення в Windows NT). За мікроядерного підходу конфігурованість ОС не створює проблем і не потребує особливих заходів – достатньо змінити файл із налагодженнями початкової конфігурації системи або ж зупинити не потрібні більше сервери в ході роботи звичайними для зупинення додатків засобами.

Використання мікроядерної моделі підвищує надійність ОС. Кожний сервер виконується у вигляді окремого процесу у власній ділянці пам'яті й у такий спосіб захищений від інших серверів ОС, що не спостерігається в традиційній ОС, де всі модулі ядра можуть впливати один на одного. І якщо окремий сервер пошкоджується, то він може бути перезапущений без зупинення або пошкодження інших серверів ОС. Більше того, оскільки сервери виконуються в користувацькому режимі, вони не мають безпосереднього доступу до апаратури й не можуть модифікувати пам'ять, у якій зберігається й працює мікроядро. Іншим потенційним джерелом підвищення надійності ОС є зменшений об'єм коду мікроядра порівняно з традиційним ядром – це знижує ймовірність появи помилок програмування.

Модель із мікроядром добре підходить для підтримання розподілених обчислень, оскільки в ній використовуються механізми, аналогічні мережевим: взаємодія клієнтів і серверів через обмін повідомленнями. Сервери мікроядерної ОС можуть працювати як на одному, так і на різних комп'ютерах. Отримавши повідомлення від додатка, мікроядро може обробити його самостійно й передати локально-

му серверу або ж переслати по мережі мікроядру, що працює на іншому комп'ютері. Перехід до розподіленого оброблення потребує мінімальних змін у роботі ОС – локальний транспорт заміняється на мережевий.

Продуктивність. За класичної організації ОС (рис. 1.11, а) виконання системного виклику супроводжується двома переключеннями режимів, а в разі мікроядерної організації (рис. 1.11, б) – чотирма. Таким чином, ОС на основі мікроядра за інших рівних умов завжди буде менш продуктивною, ніж ОС із класичним ядром. Саме з цієї причини мікроядерний підхід не дістав очікуваного поширення.

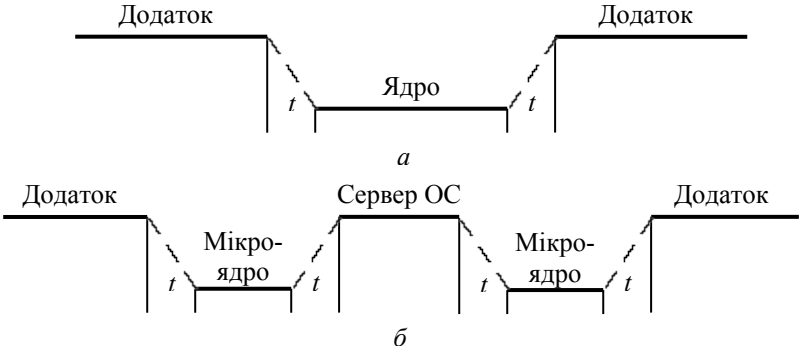


Рис. 1.11. Зміна режимів для виконання системного виклику

Недолік мікроядерного підходу добре ілюструє історія розвитку Windows NT. У версіях 3.1 і 3.5 диспетчер вікон, графічна бібліотека й високорівневі драйвери графічних пристроїв входили до складу сервера користувацького режиму, і виклик функцій цих модулів здійснювався відповідно до мікроядерної схеми. Однак незабаром розробники Windows NT зрозуміли, що такий механізм звертань до часто використовуваних функцій графічного інтерфейсу істотно сповільнює роботу додатків і робить ОС вразливою в умовах гострої конкуренції. В результаті версію Windows NT 4.0 істотно змінено – всі наведені вище модулі перенесено в ядро, що віддалило цю ОС від ідеальної мікроядерної архітектури, та зате різко підвищило її продуктивність.

Цей приклад ілюструє головну проблему, з якою стикаються розробники ОС, які вирішили застосувати мікроядерний підхід, – що включати в мікроядро, а що виносити в користувацький простір. В

ідеальному випадку мікроядро може складатися тільки із засобів передавання повідомлень, засобів взаємодії з апаратурою, зокрема засобів доступу до механізмів привілейованого захисту. Однак багато розробників не завжди жорстко дотримуються принципу мінімізації функцій ядра, часто жертвуючи цим заради підвищення продуктивності. В результаті реалізації ОС утворюють деякий спектр, на одному краю якого розміщені системи з мінімально можливим мікроядром, а на іншому – системи, подібні до Windows NT, у яких мікроядро виконує досить великий обсяг функцій.

1.4. Ядро та допоміжні модулі операційної системи. Багатошарова структура операційної системи

1.4.1. Ядро та допоміжні модулі операційної системи

Найбільш загальним підходом до структуризації ОС є поділ усіх її модулів на дві групи:

1. ядро – модулі, які виконують основні функції ОС;
2. модулі, що виконують допоміжні функції ОС.

Модулі ядра виконують такі базові функції ОС, як керування процесами, пам'яттю, пристроями введення-виведення і т.ін. Ядро є серцевиною ОС, без нього ОС є повністю непрацездатною і не зможе виконати жодної зі своїх функцій.

До складу ядра входять функції, спрямовані на виконання внутрішньосистемних завдань з організації обчислювального процесу, а саме: перемикання контекстів, завантаження-вивантаження сторінок, оброблення переривань. Ці функції недоступні для додатків. Інший клас функцій ядра служить для підтримки додатків, створюючи для них прикладне програмне середовище. Додатки можуть звертатися до ядра із запитами – системними викликами – для виконання тих або інших дій, наприклад для відкриття і зчитування файлу, виведення графічної інформації на дисплей, отримання системного часу і т.ін. Функції ядра, які можуть викликатися додатками, утворюють інтерфейс прикладного програмування – API.

Функції, що виконуються модулями ядра, є найбільш часто використовуваними функціями ОС, тому швидкість їх виконання визначає продуктивність системи в цілому. Для забезпечення високої швидкості роботи ОС усі модулі ядра або більша їх частина постійно перебувають в оперативній пам'яті, тобто є *резидентними*.

Ядро є рушійною силою всіх обчислювальних процесів у комп'ютерній системі, і руйнування ядра рівносильний руйнуванню всієї системи. Тому розробники ОС приділяють особливу увагу надійності кодів ядра, у результаті процес їх налагодження може тривати на багато місяців.

Ядру зазвичай надають вигляду програмного модуля деякого спеціального формату, що відрізняється від формату користувацьких додатків.

Інші модулі ОС виконують дуже корисні, але менш обов'язкові функції. Наприклад, такими допоміжними модулями можуть бути програми архівації даних, дефрагментації диска, текстові редактори. Допоміжні модулі ОС оформляють у вигляді або додатків, або бібліотек процедур.

Оскільки деякі компоненти ОС мають вигляд звичайних програм, тобто вигляд виконуваних модулів стандартного для цієї ОС формату, то часто дуже складно чітко розмежувати ОС і додатки.

Рішення про те, чи є яка-небудь програма частиною ОС чи не є нею, приймає виробник ОС. Серед багатьох факторів, здатних вплинути на це рішення, важливими є перспективи того, чи буде програма мати масовий попит у потенційних користувачів цієї ОС.

Деяка програма може існувати певний час як користувацький додаток, а потім стати частиною ОС, або навпаки. Яскравим прикладом такої зміни статусу програми є Web-браузер компанії Microsoft, який спочатку поставлявся як окремий додаток, потім став частиною ОС Windows.

Допоміжні модулі ОС зазвичай поділяють на такі групи:

– *утиліти* – програми, які вирішують окремі завдання керування і супроводу комп'ютерної системи (наприклад, програми стиску дисків, архівації даних);

– *системні обробні програми* – текстові або графічні редактори, компілятори, компонувальники, відладники;

– *програми надання користувачу додаткових послуг* – спеціальний варіант інтерфейсу користувача, калькулятор і навіть ігри;

– *бібліотеки процедур різного призначення*, що спрощують розроблення додатків (наприклад, бібліотека математичних функцій, функцій введення-виведення).

Як і звичайні програми, для виконання своїх завдань утиліти, системні обробні програми та бібліотеки ОС, звертаються до функцій ядра за допомогою системних викликів (рис. 1.12).

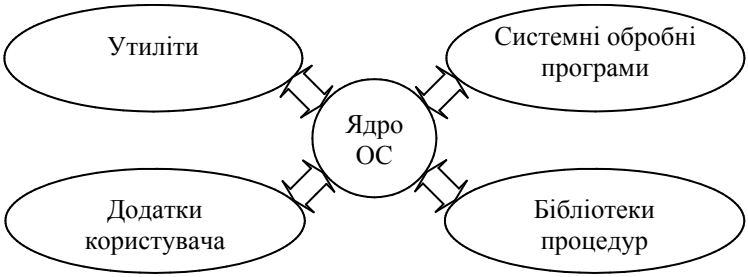


Рис. 1.12. Взаємодія між ядром і допоміжними модулями ОС

Поділ ОС на ядро і модулі-додатки забезпечує легку розширюваність ОС. Щоб додати нову високорівневу функцію, достатньо розробити новий додаток, і при цьому не потрібні модифіковані відповідальні функції, що утворюють ядро системи. Однак унесення змін у функції ядра може виявитися набагато складніше, і складність ця залежить від структурної організації самого ядра. У деяких випадках кожне виправлення ядра може вимагати його повної перекомпіляції. Модулі ОС, оформлені у вигляді утиліт, системних обробних програм і бібліотек, зазвичай завантажуються в опертивну пам'ять лише на час виконання своїх функцій, тобто є *транзитними*. Постійно в оперативній пам'яті розміщуються тільки найнеобхідніші коди ОС, які становлять її ядро, що економить оперативну пам'ять комп'ютера.

Важливою властивістю архітектури ОС, заснованої на ядрі, є можливість захисту кодів і даних ОС за рахунок виконання функцій ядра в привілейованому режимі.

1.4.2. Ядро в привілейованому режимі

Для надійного керування ходом виконання додатків ОС повинна мати стосовно додатків певні привілеї. Інакше програма, що некоректно працює, може втрутитися в роботу ОС і, наприклад, зруйнувати частину її кодів. Усі зусилля розробників ОС виявляться марними, якщо їх рішення втілені в незахищені від додатків модулі системи. Операційна система повинна мати виняткові повноваження для того, щоб відігравати роль арбітра в суперечності додатків за ресурси комп'ютера в мультипрограмному режимі. Жодна програма не повинна мати можливості без відома ОС отримувати додаткову ділянку пам'яті, займати процесор довше дозво-

леного ОС періоду часу, безпосередньо керувати спільно використовуваними зовнішніми пристроями.

Забезпечити привілеї ОС неможливо без спеціальних засобів апаратної підтримки. Апаратура комп'ютера повинна підтримувати як мінімум два режими роботи – режим користувача (user mode) і привілейований режим, який також називають режимом ядра (kernel mode), або режимом супервізора (supervisor mode). Мається на увазі, що ОС або деякі її частини працюють в привілейованому режимі, а додатки – у режимі користувача.

Оскільки ядро виконує всі основні функції ОС, то найчастіше саме ядро стає тією частиною ОС, що працює в привілейованому режимі (рис. 1.13). Іноді ця властивість – робота в привілейова-ному режимі – є основним визначенням поняття «ядро».

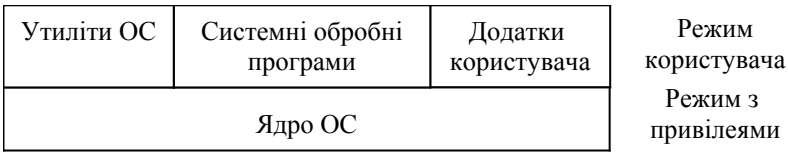


Рис. 1.13. Архітектура ОС з ядром у привілейованому режимі

Програми ставляться в залежність через заборону виконання в режимі користувача деяких критичних команд, пов'язаних з переключенням процесора із завдання на завдання, керуванням пристроями введення-виведення, доступом до механізмів розподілу та захисту пам'яті. Виконання деяких інструкцій в режимі користувача забороняється безумовно (очевидно, що до таких інструкцій належить інструкція переходу в привілейований режим), тоді як інші інструкції забороняється виконувати тільки за певних умов. Наприклад, інструкції введення-виведення можуть бути заборонені програмам при доступі до контролера жорсткого диска, який зберігає дані, загальні для ОС і всіх додатків, але дозволені для доступу до послідовного порту, який виділений у монопольне володіння для певної програми. Важливо, що умови дозволу виконання критичних інструкцій перебувають під повним контролем ОС і цей контроль забезпечується за рахунок набору інструкцій, безумовно заборонених для режиму користувача.

Аналогічним чином забезпечуються привілеї при доступі до пам'яті. Наприклад, виконання інструкції доступу до пам'яті для про-

грами дозволяється, якщо інструкція звертається до ділянки пам'яті, відведеної для додатка ОС, і забороняється під час звернення до ділянок пам'яті, зайнятих ОС або іншими додатками. Повний контроль ОС над доступом до пам'яті досягається за рахунок того, що інструкцію або інструкції конфігурування механізмів захисту пам'яті (наприклад, зміни ключів захисту пам'яті в мейнфреймах IBM або вказівник таблиці дескрипторів пам'яті в процесорах Pentium) дозволяється виконувати тільки в привілейованому режимі.

Дуже важливо, що механізми захисту пам'яті використовуються ОС не тільки для захисту власних ділянок пам'яті від додатків, але і для захисту ділянок пам'яті, виділених ОС будь-якому додатку, від інших додатків. Кожен додаток працює у власному адресному просторі. Ця властивість дозволяє локалізувати додаток, що некоректно працює, у власній ділянці пам'яті, тому його помилки не впливають на інші програми і ОС.

Між кількістю рівнів привілеїв, що реалізуються апаратно, і кількістю рівнів привілеїв, підтримуваних ОС, немає прямої відповідності. Так, на базі чотирьох рівнів, що забезпечуються процесорами компанії Intel, ОС OS/2 будує трирівневу систему привілеїв, а ОС Windows NT, UNIX і деякі інші обмежуються дворівневою системою.

Однак, якщо апаратура підтримує хоча б два рівні привілеїв, то ОС може на цій основі створити програмним способом як завгодно розвинену систему захисту.

Ця система може, наприклад, підтримувати декілька рівнів привілеїв, що утворюють ієрархію. Наявність декількох рівнів привілеїв дозволяє більш тонко розподіляти повноваження як між модулями ОС, так і між самими програмами. Поява всередині ОС більш привілейованих і менш привілейованих частин дозволяє підвищити стійкість ОС до внутрішніх помилок програмних кодів, оскільки такі помилки будуть поширюватися тільки всередині модулів з певним рівнем привілеїв. Диференціація привілеїв у середовищі прикладних модулів дозволяє будувати складні прикладні комплекси, у яких частина більш привілейованих модулів може, наприклад, отримувати доступ до даних менш привілейованих модулів і керувати їх виконанням.

На основі двох режимів привілеїв процесора ОС може побудувати складну систему індивідуального захисту ресурсів, прикладом якої є типова система захисту файлів і каталогів. Така система

дозволяє задати для будь-якого користувача певні права доступу до кожного з файлів і каталогів.

Підвищення стійкості ОС, що забезпечується переходом ядра в привілейований режим, досягається деяким уповільненням виконання системних викликів. Системний виклик привілейованого ядра ініціює перемикання процесора з режиму користувача в привілейований, а в разі повернення до додатка – перемикання з привілейованого режиму в режим користувача (рис. 1.14).

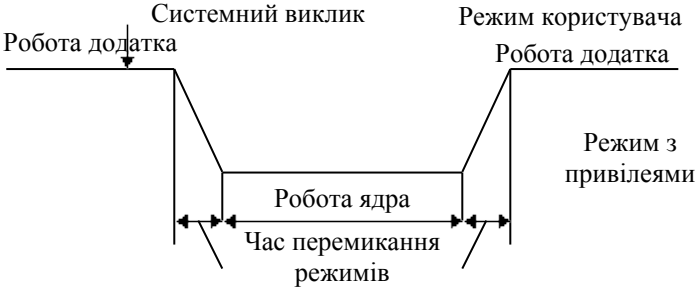


Рис. 1.14. Зміна режимів під час виконання системного виклику до привілейованого ядра

У всіх типах процесорів, через додаткове дворазове затримання перемикання, перехід на процедуру зі зміною режиму виконуєть-ся повільніше, ніж виклик процедури без зміни режиму.

Архітектура ОС, побудована на привілейованому ядрі та додат-ках режиму користувача, стала, по суті, класичною. Її використовують багато популярних ОС, зокрема численні версії UNIX, VAX VMS, OS/2 і з певними модифікаціями – Windows NT та ін.

У деяких випадках розробники ОС відступають від цього класичного варіанта архітектури, організовуючи роботу ядра та програм в одному і тому ж режимі. За такої побудови ОС звер-нення додатків до ядра виконуються швидше, оскільки немає перемикання режимів, однак немає надійного апаратного захисту пам'яті, займаної модулями ОС, від некоректно працюючого додатка. Розробники спеціалізованої ОС NetWare компанії Novell вдалися до такого потенційного зниження надійності власної ОС, оскільки обмежений набір її спеціалізованих додатків дозволяє компенсувати цей архітектурний недолік ретельним налагодженням кожної програми.

В одному режимі працюють також ядро і програми тих ОС, що розроблені для процесорів, які взагалі не підтримують

привілейований режим роботи. Найпоширенішим процесором такого типу був процесор Intel 8088/86, що став основою для ПК компанії IBM. Операційна система MS-DOS (дод. 1), розроблена компанією Microsoft для цих комп'ютерів, складалася з двох модулів msdos.sys і io.sys, що становили ядро системи (хоча назва «ядро» для цих модулів не вживалася, за своїм змістом вони ним були), до яких із системними викликами зверталися командний інтерпретатор command.com, системні утиліти та додатки. Архітектура MS-DOS відповідає архітектурі ОС. Некоректно написані програми цілком могли зруйнувати основні модулі MS-DOS, що іноді й відбувалося, але до використання MS-DOS (і багатьох подібних їй ранніх ОС для ПК, таких як MSX, CP/M) і не ставилися високі вимоги до надійності ОС.

1.4.3. Багатошарова структура операційної системи

Обчислювальну систему, якою керує ОС на основі ядра, можна розглядати як систему, що складається з трьох ієрархічно розміщених шарів. На нижньому шарі міститься апаратура, на проміжному – ядро, а на верхньому – утиліти, системні обробні програми й додатки (рис. 1.15). Шарову структуру обчислювальної системи зображають у вигляді системи концентричних кіл, ілюструючи той факт, що кожен шар може взаємодіяти тільки із суміжними шарами. Дійсно, за такої організації ОС програми не можуть безпосередньо взаємодіяти з апаратурою, а тільки через шар ядра.

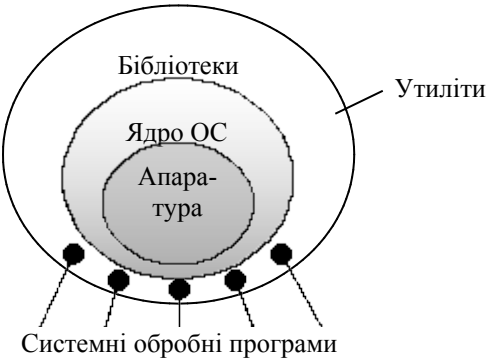


Рис. 1.15. Тришарова схема обчислювальної системи

Багатошаровий підхід є універсальним і ефективним способом декомпозиції складних систем будь-якого типу, в тому числі і програмних. Згідно з цим підходом система складається з ієрархії шарів. Кожен шар обслуговує розміщений вище шар, виконуючи для нього деякий набір функцій, які утворюють міжшаровий інтерфейс. На основі функцій розміщеного нижче шару наступний (вгору за ієрархією) шар будує свої функції – більш складні й потужніші, які, в свою чергу, виявляються примітивами для створення ще більш потужних функцій розміщеного вище шару. Жорсткі правила стосуються тільки взаємодії між шарами системи, а між модулями всередині шару зв'язки можуть бути довільними. Окремий модуль може виконати роботу або самостійно, або звернутися до іншого модулю свого шару, або звернутися за допомогою до нижчого рівня шару через міжшаровий інтерфейс.

Така організація системи має багато переваг. Вона суттєво спрощує розроблення системи, оскільки дозволяє спочатку визначити функції шарів і міжшарові інтерфейси «згори вниз», а потім за детальної реалізації поступово нарощувати потужність функцій шарів, рухаючись «знизу вгору». Крім того, у процесі модернізації системи можна змінювати модулі всередині шару без необхідності робити будь-які зміни в інших шарах, якщо ці внутрішні зміни не впливають на міжшарові інтерфейси. Оскільки ядро являє собою складний багатофункціональний комплекс, то багатошаровий підхід зазвичай поширюється і на структуру ядра.

Ядро може складатися з таких шарів: засобів апаратної підтримки, машиннозалежних модулів, базових механізмів ядра, менеджерів ресурсів, інтерфейсу системних викликів.

Засоби апаратної підтримки ОС. Операційна система є комплексом програм, але частину функцій ОС можуть виконувати і апаратні засоби. Тому іноді ОС визначають як сукупність програмних і апаратних засобів (рис. 1.16). До ОС належать, не всі апаратні пристрої комп'ютера, а лише засоби апаратної підтримки ОС, тобто ті, які безпосередньо беруть участь в організації обчислювальних процесів: засоби підтримки привілейованого режиму, система переривань, засоби перемикання контекстів процесів, засоби захисту ділянок пам'яті і т.ін.

Машиннозалежні модулі ОС. Цей шар утворюють програмні модулі, в яких відображається специфіка апаратної платформи

комп'ютера. В ідеалі цей шар повністю екранує розміщені вище шари ядра від особливостей апаратури. Це дозволяє розробляти розміщені вище шари на основі машиннонезалежних модулів, що існують у єдиному екземплярі для всіх типів апаратних платформ, які підтримує ця ОС, як приклад екрануючого шару можна навести шар HAL ОС Windows NT.

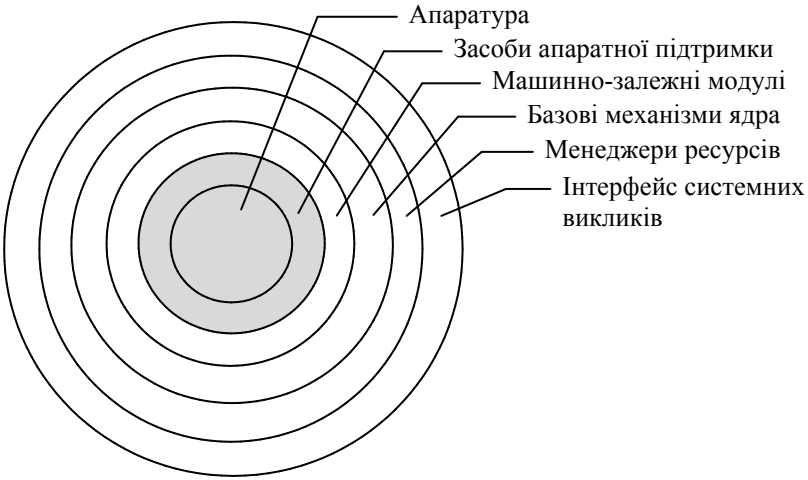


Рис. 1.16. Багатошарова структура ядра ОС

Базові механізми ядра. Цей шар виконує найбільш примітивні операції ядра, такі як програмне перемикання контекстів процесів, диспетчеризацію переривань, переміщення сторінок з пам'яті на диск і назад і т. ін. Модулі цього шару не приймають рішень про розподіл ресурсів – вони тільки відпрацьовують прийняті «вгорі» рішення, що стало приводом називати їх виконавчими механізмами для модулів верхніх шарів. Наприклад, рішення про те, що в заданий момент часу потрібно перервати виконання поточного процесу *A* і почати виконання процесу *B*, приймається менеджером процесів на шарі, розміщеному вище, а шару базових механізмів передається тільки директива про потребу виконати перемикання з контексту поточного процесу на контекст процесу *B*.

Менеджери ресурсів. Цей шар складається з потужних функціональних модулів, що реалізують стратегічні завдання з керування основними ресурсами обчислювальної системи. Зазвичай на

цьому шарі працюють менеджери (диспетчери) процесів, введення-виведення, файлової системи й оперативної пам'яті. Розбиття на менеджери може бути і трохи іншим, наприклад менеджер файлової системи іноді поєднують з менеджером введення-виведення, а функції керування доступом користувачів до системи в цілому та до її окремих об'єктів доручають окремому менеджеру безпеки. Кожен з менеджерів веде облік вільних і використовуваних ресурсів певного типу і планує їх розподіл відповідно до запитів додатків. Наприклад, менеджер віртуальної пам'яті керує переміщенням сторінок з оперативної пам'яті на диск і назад. Менеджер має відстежувати інтенсивність запитів сторінок, час перебування їх у пам'яті, стани процесів, що використовують дані, й багато інших параметрів, на підставі яких він час від часу приймає рішення про те, які сторінки необхідно вивантажити і які завантажити. Для виконання прийнятих рішень менеджер звертається до нижчого рівня шару базових механізмів із запитами про завантаження (вивантаження) конкретних сторінок. Усередині шару менеджерів існують тісні взаємні зв'язки, що відображають той факт, що для виконання процесу потрібен доступ одночасно до декількох ресурсів – процесору, ділянки пам'яті, можливо, до певного файлу або пристрою введення-виведення. Наприклад, під час створення процесу менеджер процесів звертається до менеджера пам'яті, який повинен виділити процесу певну ділянку пам'яті для його кодів і даних.

Інтерфейс системних викликів. Цей шар є найбільш верхнім шаром ядра й взаємодіє безпосередньо з додатками та системними утилітами, утворюючи прикладний програмний інтерфейс ОС (API). Функції API, що обслуговують системні виклики, надають доступ до ресурсів системи в зручній і компактній формі без вказівки деталей їх фізичного розміщення. Наприклад, в ОС UNIX за допомогою системного виклику

```
fd = open ( "/ doc / a.txt", 0_RDONLY)
```

програма відкриває файл *a.txt*, що зберігається в каталозі */ doc*, а за допомогою системного виклику

```
read (fd, buffer, count)
```

зчитує з цього файлу в ділянку власного адресного простору, що має ім'я *buffer*, деяку кількість байтів. Для здійснення таких комплекс-

них дій системні виклики зазвичай звертаються за допомогою до функцій шару менеджерів ресурсів, причому для виконання одного системного виклику може знадобитися декілька таких звернень.

Наведене розбиття ядра ОС на шари є досить умовним. У реальній системі кількість шарів і розподіл функцій між ними можуть бути й іншими. У системах, призначених для апаратних платформ одного типу, наприклад ОС NetWare, шар машиннозалежних модулів зазвичай не виділяється, зливаючись із шаром базових механізмів і, частково, з шаром менеджерів ресурсів. Не завжди оформляються в окремий шар базові механізми – в цьому випадку менеджери ресурсів не тільки планують використання ресурсів, а й самостійно реалізують свої плани.

Можлива й протилежна картина, коли ядро складається з більшої кількості шарів. Наприклад, менеджери ресурсів, складаючи певний шар ядра, в свою чергу, можуть мати багатошарову структуру. Перш за все це стосується менеджера введення-виведення, нижній шар якого складають драйвери пристроїв, наприклад драйвер жорсткого диска або драйвер мережевого адаптера, а верхні шари – драйвери файлових систем або протоколів мережевих служб, які відповідають за логічну організацію інформації.

Спосіб взаємодії шарів у реальній ОС також може відхилятися від описаної вище схеми. Для прискорення роботи ядра в деяких випадках відбувається безпосереднє звернення з верхнього шару до функцій нижніх шарів, оминаючи проміжні. Типовим прикладом такої «неправильної» взаємодії є початкова стадія оброблення системного виклику. На багатьох апаратних платформах для реалізації системного виклику використовується інструкція програмного переривання. Цим додаток фактично викликає модуль первинного оброблення переривань, який міститься в шарі базових механізмів, а вже цей модуль викликає потрібну функцію з шару системних викликів. Самі функції системних викликів також іноді порушують субординацію ієрархічних шарів, звертаючись прямо до базових механізмів ядра.

Вибір кількості шарів ядра є відповідальним і складним завданням: збільшення кількості шарів веде до деякого уповільнення роботи ядра за рахунок додаткових накладних витрат на міжшарову взаємодію, а зменшення кількості шарів погіршує розширюваність і логічність системи. Зазвичай ОС, що пройшли довгий шлях

еволюційного розвитку, мають невпорядковане ядро з невеликою кількістю чітко виділених шарів, а у порівняно «молодих» ОС ядро розділене на більшу кількість шарів і їх взаємодія формалізована набагато більше.

1.5. Апаратна залежність і переносимість операційних систем

1.5.1. Апаратна залежність операційних систем

Багато ОС успішно працюють на різних апаратних платформах без істотних змін у своєму складі. Багато в чому це пояснюється тим, що, незважаючи на відмінності в деталях, засоби апаратної підтримки ОС більшості комп'ютерів набули тепер багато типових ознак, зокрема ці засоби впливають передусім на роботу компонентів ОС. У результаті в ОС можна виділити досить компактний шар машиннозалежних компонентів ядра й зробити інші шари ОС загальними для різних апаратних платформ.

1.5.2. Типові засоби апаратної підтримки операційних систем

Чіткої межі між програмною й апаратною реалізацією функцій ОС немає– рішення про те, які функції ОС будуть виконуватися програмно, а які апаратно, приймають розробники апаратного й програмного забезпечення комп'ютера. Проте всі сучасні апаратні платформи мають деякий типовий набір засобів апаратної підтримки ОС, зокрема таких:

– засобів підтримки привілейованого режиму;
– засобів трансляції адрес;
– засобів перемикання процесів;
– системи переривань;
– системного таймера;
– засобів захисту ділянок пам'яті.

Засоби підтримки привілейованого режиму зазвичай ґрунтуються на системному регістрі процесора, який часто називають «словом стану» машини або процесора. Цей регістр має деякі ознаки, що обумовлюють режими роботи процесора, в тому числі й ознаку по-

точного режиму привілеїв. Зміна режиму привілеїв виконується за рахунок зміни «слова стану» машини внаслідок переривання або виконання привілейованої команди. Кількість градацій привілейованості може бути різною у різних типів процесорів, найбільш часто використовуються два рівні (ядро – користувач) або чотири (наприклад, ядро – супервізор – виконання – користувач у платформи VAX або 0–1–2–3 у процесорів Intel x86/Pentium). Засобами підтримки привілейованого режиму є виконання перевірки допустимості виконання активною програмою інструкцій процесора за поточного рівня привілейованості.

Засоби трансляції адрес виконують операції перетворення віртуальних адрес, які містяться в кодах процесу, в адреси фізичної пам'яті. Таблиці, призначені для трансляції адрес, зазвичай мають великий обсяг, тому для їх зберігання використовуються ділянки оперативної пам'яті, а апаратура процесора містить тільки вказівники на ці ділянки. Засоби трансляції адрес використовують дані вказівники для доступу до елементів таблиць і апаратного виконання алгоритму перетворення адреси, що значно прискорює процедуру трансляції порівняно з її чисто програмною реалізацією.

Засоби перемикання процесів призначені для швидкого збереження контексту процесу, що припиняється, й відновлення контексту процесу, який стає активним. Уміст контексту містить усі регістри загального призначення процесора, регістр прапорів операцій (тобто прапорів нуля, перенесення, переповнення і т.ін.), а також ті системні регістри і вказівники, які пов'язані з окремим процесом, а не з ОС, наприклад, вказівника на таблицю трансляції адрес процесу. Для зберігання контекстів припинених процесів використовуються ділянки оперативної пам'яті, які підтримуються вказівниками процесора.

Перемикання контексту виконується за певними командами процесора, наприклад за командою переходу до виконання нового завдання. Така команда викликає автоматичне завантаження даних зі збереженого контексту в регістри процесора, після чого процес триває з перерваного раніше місця.

Система переривань дозволяє комп'ютеру реагувати на зовнішні події, синхронізувати виконання процесів і роботу пристроїв введення-виведення, швидко переходити з однієї програми на іншу. Механізм переривань потрібен для того, щоб сповістити проце-

сор про виникнення в обчислювальній системі якоїсь непередбаченої події або події, яка не синхронізована з циклом роботи процесора. Як приклади таких подій можна навести завершення операції введення-виведення зовнішнім пристроєм (наприклад, запис блока даних контролером диска), некоректне завершення арифметичної операції (наприклад, переповнення регістра), закінчення інтервалу астрономічного часу. За наявності умов переривання його джерело (контролер зовнішнього пристрою, таймер, арифметичний блок процесора тощо) виставляє певний електричний сигнал. Цей сигнал перериває виконання процесором послідовності команд, що задається виконуваним кодом, і спричиняє автоматичний перехід на заздалегідь визначену процедуру, названу процедурою оброблення переривань. У більшості моделей процесорів виконуваний апаратурою перехід на процедуру оброблення переривань супроводжується заміною «слова стану» машини (або навіть усього контексту процесу), що дозволяє одночасно з переходом за потрібною адресою виконати перехід у привілейований режим. Після завершення процесу оброблення переривань відбувається повернення до виконання перерваного коду.

Переривання є найважливішою функцією будь-якої ОС, будучи її рушійною силою. Дійсно, більша частина дій ОС ініціюється перериваннями різного типу. Навіть системні виклики від додатків виконуються на багатьох апаратних платформах за допомогою спеціальної інструкції переривання, що викликає перехід до виконання відповідних процедур ядра (наприклад, інструкція int у процесорах Intel або SVC у мейнфреймах IBM).

Системний таймер, часто реалізований у вигляді швидкодійного регістра-лічильника, необхідний ОС для витримування інтервалів часу. Для цього в регістр таймера програмно завантажується значення необхідного інтервалу в умовних одиницях, з якого потім автоматично з певною частотою починає відраховуватися по одиниці. Частота «тиків» таймера, як правило, тісно пов’язана з частотою тактового генератора процесора. (Не слід плутати таймер ні з тактовим генератором, який виробляє сигнали, що синхронізують усі операції в комп’ютері, ні з системним годинником – працюючою на батареях електронною схемою, – які ведуть незалежний відлік часу й календарної дати.) Із досягненням нульового значення лічильника таймер ініціює переривання, яке обробляється процеду-

рою ОС. Переривання від системного таймера використовуються ОС у першу чергу для спостереження за тим, як окремі процеси витрачають час процесора. Наприклад, у системі розділення часу підчас оброблення чергового переривання від таймера планувальник процесів може примусово передати керування іншому процесу, якщо цей процес вичерпав виділений йому квант часу.

Засоби захисту ділянок пам'яті забезпечують на апаратному рівні перевірку можливості програмного коду здійснювати з даними певної ділянки пам'яті такі операції, як зчитування, запис або виконання (у разі передавання керування). Якщо апаратура комп'ютера підтримує механізм трансляції адрес, то засоби захисту ділянок пам'яті вмонтовуються в цей механізм. Функції апаратури із захисту пам'яті полягають у порівнянні рівнів привілеїв поточного коду процесора й сегмента пам'яті, до якого робиться звернення.

1.5.3. Машиннозалежні компоненти операційної системи

Одна й та сама ОС не може без яких-небудь змін установлюватися на комп'ютерах, що відрізняються типом процесора і/або способом організації всієї апаратури. У модулях ядра ОС не можуть не відобразитися такі особливості апаратної платформи, як кількість типів переривань і формат таблиці посилань на процедури оброблення переривань, склад регістрів загального призначення й системних регістрів, стан яких потрібно зберігати в контексті процесу, особливості підключення зовнішніх пристроїв і т. ін.

Однак досвід розроблення ОС показує: ядро можна спроектувати таким чином, що тільки деякі модулі будуть машиннозалежними, а інші не будуть залежати від особливостей апаратної платформи. У добре структурованому ядрі машиннозалежні модулі локалізовані й утворюють програмний шар, що природно примикає до шару апаратури, як це й показано на рис. 1.16. Така локалізація машиннозалежних модулів суттєво спрощує перенесення ОС на іншу апаратну платформу.

Обсяг машиннозалежних модулів ОС залежить від того, наскільки великі відмінності в апаратних платформах, для яких розробляється ОС. Наприклад, ОС, побудована на 64-бітових адресах, для перенесення на машину з 32-бітовими адресами має бути переписана наново. Одна з найбільш очевидних відмінностей – незбіж-

ність системи команд процесорів – долається достатньо просто. Операційна система програмується мовою високого рівня, а потім відповідним компілятором виробляється код для конкретного типу процесора. Однак у багатьох випадках відмінності в організації апаратури комп'ютера лежать набагато глибше й подолати їх у такий спосіб не вдається. Наприклад, однопроцесорний і двопроцесорний комп'ютери потребують застосування в ОС зовсім різних алгоритмів розподілу процесорного часу. Аналогічна відсутність апаратної підтримки віртуальної пам'яті приводить до принципової різниці в реалізації підсистеми керування пам'яттю. У таких випадках не обійтися без внесення в код ОС специфіки апаратної платформи, для якої ця ОС призначається.

Для зменшення кількості машиннозалежних модулів виробники ОС зазвичай обмежують універсальність машиннонезалежних модулів. Це означає, що їх незалежність має умовний характер і поширюється тільки на кілька типів процесорів і створених на основі цих процесорів апаратних платформ. Цим шляхом пішли, наприклад, розробники ОС Windows NT, обмеживши кількість типів процесорів для своєї системи чотирма й поставляючи різні варіанти кодів ядра для однопроцесорних та багатопроцесорних комп'ютерів.

Особливе місце серед модулів ядра займають низькорівневі драйвери зовнішніх пристроїв. З одного боку, ці драйвери, як і високорівневі драйвери, входять до складу менеджера введення-виведення, тобто належать до шару ядра, що займає досить високе місце в ієрархії шарів. З другого боку, низькорівневі драйвери відображають усі особливості керованих зовнішніх пристроїв, тому їх можна віднести й до шару машиннозалежних модулів. Така подвійність низькорівневих драйверів ще раз підтверджує схематичність моделі ядра із суворою ієрархією шарів.

Для комп'ютерів на основі процесорів Intel x86/Pentium розроблення екрануючого машиннозалежного шару ОС дещо спрощується за рахунок убудованої в постійну пам'ять комп'ютера базової системи введення-виведення – BIOS, що містить драйвери для всіх пристроїв, що входять у базову конфігурацію комп'ютера: жорстких і гнучких дисків, клавіатури, дисплея і т. ін. Ці драйвери виконують досить примітивні операції з керованими пристроями, наприклад зчитування групи секторів даних з певної доріжки диска, але за рахунок цих операцій екрануються відмінності апаратних

платформ ПК і серверів на процесорах Intel різних виробників. Розробники ОС можуть користуватися шаром драйверів BIOS як частиною машиннозалежного шару ОС, а можуть і замінити всі або частину драйверів BIOS компонентами ОС.

1.5.4. Переносимість операційних систем

Якщо код ОС може бути порівняно легко перенесений з процесора одного типу на процесор іншого типу й з апаратної платформи одного типу на апаратну платформу іншого типу, то таку ОС називають *переносимою* (*portable*), або *мобільною*.

Хоча ОС часто описуються або як переносимі, або як непереносимі, мобільність – це не бінарний стан, а поняття ступеня переносимості. Річ не в тім, чи може бути система перенесена, а в тім, наскільки легко можна це зробити. Для того щоб забезпечити властивість мобільності ОС, розробники повинні додержуватися таких правил.

1. Більша частина коду повинна бути написана мовою, транслятори якої є на всіх машинах, куди передбачається переносити систему. Такими мовами є стандартизовані мови високого рівня. Більшість переносимих ОС написано мовою С, яка має багато особливостей, корисних для розроблення кодів ОС, і компілятори якої широко доступні. Програма, написана мовою асемблеру, є переносимою тільки в тих випадках, коли перенесення ОС планується на комп'ютер, що має ту ж систему команд. В інших випадках асемблер використовується тільки для тих непереносимих частин системи, які повинні безпосередньо взаємодіяти з апаратурою (наприклад, обробник переривань), або для частин, які потребують максимальної швидкості (наприклад, цілочислова арифметика підвищеної точності).

2. Обсяг машиннозалежних частин коду, які безпосередньо взаємодіють з апаратними засобами, має бути по можливості мінімізований. Так, наприклад, слід усіляко уникати прямого маніпулювання регістрами й іншими апаратними засобами процесора. Для зменшення апаратної залежності розробники ОС повинні також унеможливити використання за замовчуванням стандартних конфігурацій апаратури або їх характеристик. Апаратно залежні параметри можна «сховати» у програмно-задавані дані абстрактного

типу. Для виконання необхідних дій з керування апаратурою за цими параметрами повинен бути написаний набір апаратно залежних функцій. Щоразу, коли якому-небудь модулю ОС потрібно виконати якусь дію, пов'язану з апаратурою, він маніпулює абстрактними даними, використовуючи відповідну функцію з наявного набору. Коли ОС переноситься, то змінюються тільки ці дані й функції, які ними маніпулюють. Наприклад, в ОС Windows NT диспетчер переривань перетворює апаратні рівні переривань конкретного типу процесора в стандартний набір рівнів переривань IRQL, з якими працюють інші модулі ОС. Тому для перенесення Windows NT на нову платформу потрібно переписати, зокрема, ті коди диспетчера переривань, які займаються відображенням рівнів переривання на абстрактні рівні IRQL, а ті модулі ОС, які користуються цими абстрактними рівнями, змін не зажадають.

3. Апаратно-залежний код повинен бути надійно ізольований у декількох модулях, а не бути розподіленим по всій системі. Ізоляції підлягають усі частини ОС, які відображають специфіку як процесору, так і апаратну платформу в цілому. Низькорівневі компоненти ОС, що мають доступ до процесорно залежних структур даних і регістрів, мають бути оформлені у вигляді компактних модулів, які можна замінити аналогічними модулями для інших процесорів. Для усунення платформної залежності, що виникає через відмінності між комп'ютерами різних виробників, побудованими на тому самому процесорі (наприклад, MIPS R4000), повинен бути введений добре локалізований програмний шар машиннозалежних функцій.

В ідеалі шар машиннозалежних компонентів ядра повністю екранує іншу частину ОС від конкретних деталей апаратної платформи (кеш, контролери переривань введення-виведення і т. ін.), принаймні для того набору платформ, який підтримує ця ОС. У результаті відбувається підміна реальної апаратури якоюсь уніфікованою віртуальною машиною, однаковою для всіх варіантів апаратної платформи. Усі шари ОС, розміщені вище від шару машиннозалежних компонентів, можуть бути написані для керування саме цією віртуальною апаратурою. Таким чином, у розробників з'являється можливість створювати один варіант машиннонезалежної частини ОС (включаючи компоненти ядра, утиліти, системні обробні програми) для всього набору підтримуваних платформ.

1.6. Файлові системи

1.6.1. Основні визначення

Файлова система (ФС) – це частина ОС, призначення якої полягає в тому, щоб забезпечити користувачу зручний інтерфейс для роботи з даними, що зберігаються на диску, а також сумісне використання файлів декількома користувачами і процесами.

У широкому сенсі поняття «файлова система» охоплює:

– сукупність всіх файлів на диску;

– набори структур даних, використовуваних для керування файлами, такі, наприклад, як каталоги файлів, дескриптори файлів, таблиці розподілу вільного і зайнятого простору на диску;

– комплекс системних програмних засобів, що реалізовують керування файлами, зокрема: створення, знищення, зчитування, запис, іменування, пошук та інші операції над файлами.

1.6.2. Іменування файлів

Правила іменування файлів залежать від ОС:

– у багатьох ОС підтримуються імена з двох частин (імені та розширення), наприклад progr.c (файл, що містить текст програми мовою С) або autoexec.bat (файл, що містить команди інтерпретатора командної мови);

– тип розширення файлу дозволяє ОС організувати роботу з ним для різних прикладних програм за наперед обумовленими узгодженнями;

– зазвичай ОС накладають деякі обмеження як на використовувані в імені символи, так і на довжину імені файла;

– відповідно до стандарту POSIX популярні ОС оперують зручними для користувача довгими іменами (до 255 символів).

1.6.3. Типи файлів

Файли розрізняють за типами:

– звичайний;

- текстовий;

• двійковий;
– спеціальний;
– каталог.

1.6.4. Логічна організація файлової системи

Види логічної організації ФС показано на рис. 1.17.

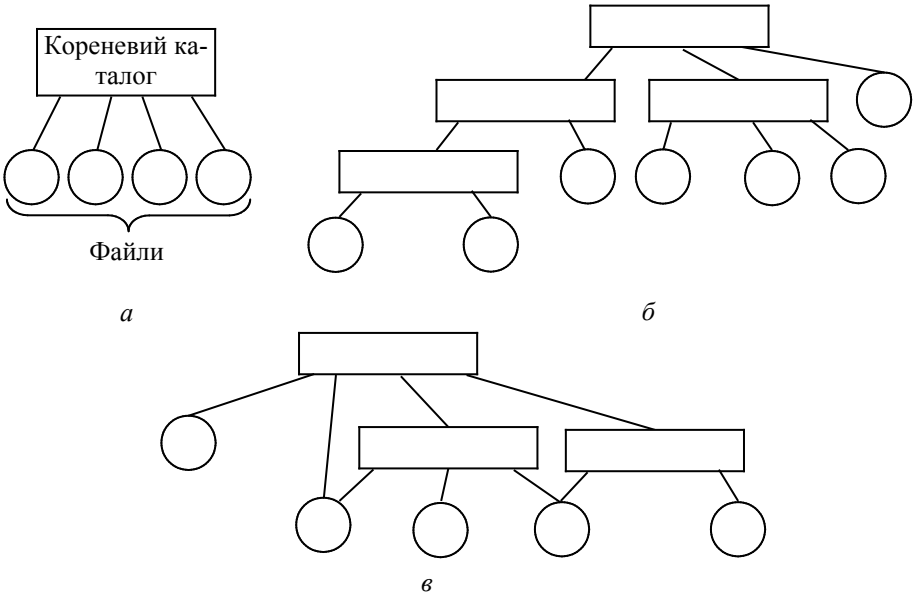


Рис. 1.17. Логічна організація ФС:
а – однорівнева; б – ієрархічна; *в* – ієрархічна мережа

1.6.5. Реалізація файлової системи

Для організації зберігання інформації на диску користувач спочатку зазвичай його форматує, виділяючи на ньому місце для структур даних, які описують ФС у цілому, потім створює потрібну йому структуру каталогів, які, по суті, є списками вкладених каталогів і, власне, файлів. Нарешті, він заповнює дисковий простір файлами, приписуючи їх до того або іншого каталога. Операційна

система повинна надати в розпорядження користувача сукупність системних викликів, які забезпечують його необхідними сервісами.

1.6.6. Можлива структура та модель файлової системи

Можливу структуру ФС показано на рис. 1.18.

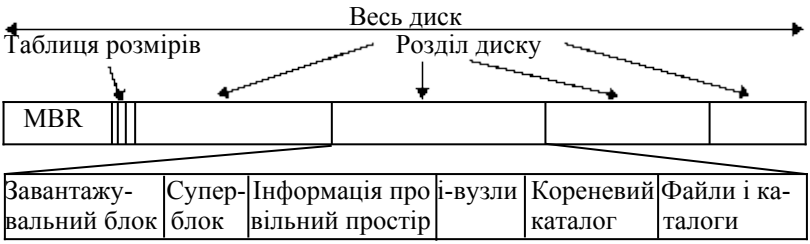


Рис. 1.18. Структура ФС

Багаторівневу модель ФС зображено на рис. 1.19.

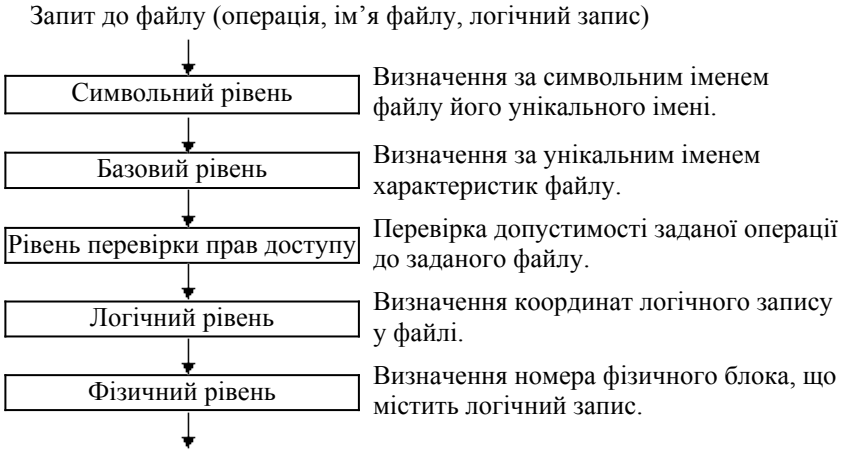


Рис. 1.19. Багаторівнева модель ФС

1.6.7. Фізичні й логічні диски. Розділи диска

Перші ПК IBM PC були укомплектовані тільки нагромаджувачами на гнучких магнітних дисках, або дискетах. Дискети дозво-

ляють зберігати відносно невеликі обсяги інформації, тому ділити флоппі-диск на частини не має сенсу. Наступна модель комп'ютера – IBM XT – мала жорсткий диск ємністю 10 або 20 Мбайт. Диск ємністю 20 Мбайт мали й деякі екземпляри IBM AT. Використання таких дисків і ОС MS-DOS версій до 3.20 для користувачів не становило жодних труднощів і не викликало бажання розбити диск такої малої ємності на ще менші частини.

Проблеми виникли, коли виробники жорстких дисків освоїли випуск дисків ємністю 40 Мбайт і більше. Виявилось, що використовуваний DOS механізм 16-розрядної адресації секторів не дозволяє використовувати диски з ємністю понад 32 Мбайт.

Причини для розділення диска на розділи. У разі пошкодження логічного диска зникає тільки та інформація, яка міститься на цьому логічному диску.

Реорганізація та вивантаження диска малого розміру простіші й швидші, ніж великого.

Можливе розділення дискового простору між окремими користувачами ПК. Така практика «колективної» роботи на ПК дуже поширена.

У разі використання спеціальних утиліт для розбиття диска на частини (диск-менеджерів) можливе встановлення для окремих логічних дисків захисту від запису. Можна записувати на такі диски інформацію, що не змінюється. Шкода від програм-вірусів також буде менша – вірус не зможе записати себе на захищений диск.

Один диск може містити декілька різних ОС, розміщених у різних розділах диска. В ході початкового завантаження можна вказати розділ диска, з якого повинна завантажуватися ОС.

Декілька варіантів розбиття фізичного диска на розділи показано на рис. 1.20.

Головний завантажувальний запис і таблиця розділів диска. Найперший сектор жорсткого диска (сектор 1, доріжка 0, головка 0) містить головний завантажувальний запис (Master Boot Record – MBR). Цей запис займає не весь сектор, а тільки його початкову частину. Сам по собі головний завантажувальний запис є програмою. Ця програма під час початкового завантаження ОС з жорсткого диска поміщається за адресою 7C00:0000, після чого їй

передається керування. Завантажувальний запис продовжує процес завантаження ОС. Формат MBR наведено в табл. 1.1.

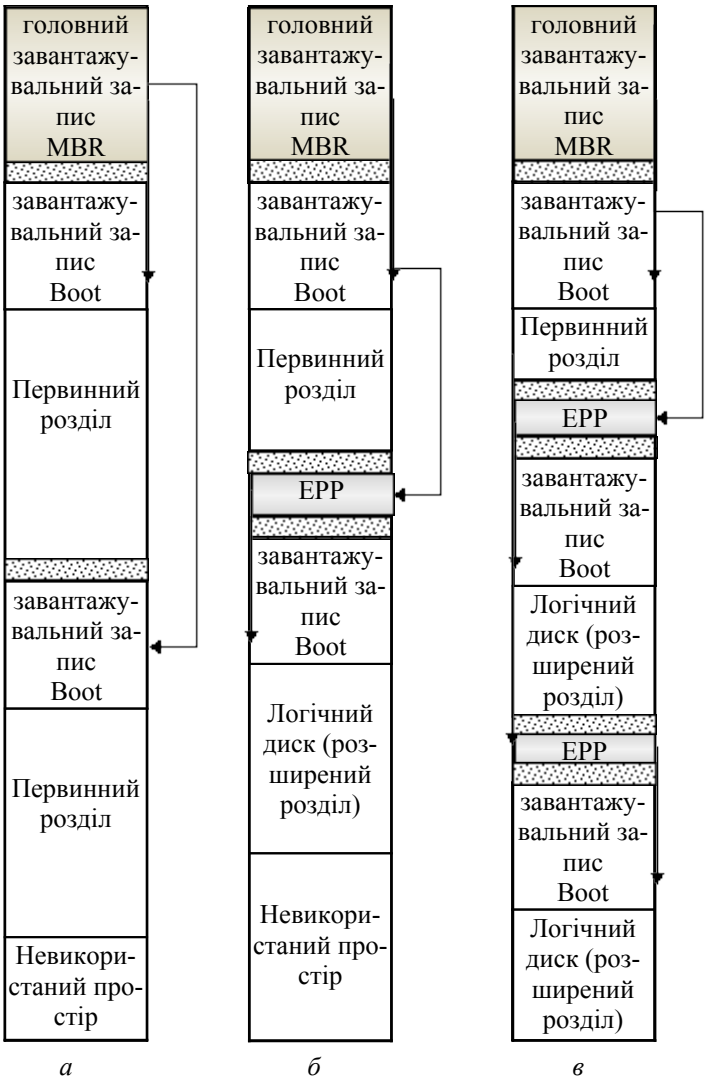


Рис. 1.20. Розділи фізичного диска:

а – первинні розділи (розширених розділів немає); *б* – один первинний розділ і один логічний диск у розширеному розділі; *в* – один первинний розділ і два логічні розділи (відповідні записи розширених розділів сполучені в ланцюжок)

Таблиця 1.1

Формат *MBR*

Зсув	Розмір	Уміст
(+0)	1BEh	Завантажувальний запис – програма, яка завантажується і виконується під час початкового завантаження ОС
(+1BEh)	10h	Елемент таблиці розділів диска
(+1CEh)	10h	Елемент таблиці розділів диска
(+1DEh)	10h	Елемент таблиці розділів диска
(+1EEh)	10h	Елемент таблиці розділів диска
(+1FEh)	2	Ознака таблиці розділів – 55AAh

У кінці першого сектора жорсткого диска розміщується таблиця розділів диска (*partition table*). Ця таблиця містить чотири елементи, що описують максимальні чотири розділи диска. В останніх двох байтах сектора міститься число 55AA. Це ознака таблиці розділів, формат елемента якої подано в табл. 1.2.

Таблиця 1.2

Формат елемента таблиці розділів

Зсув	Розмір	Уміст
(+0)	1	Ознака активного розділу: 0 – розділ не активний; 80h – розділ активний
(+1)	1	Номер головки для початкового сектора розділу
(+2)	2	Номер сектора і циліндра для початкового сектора розділу у форматі функції читання сектора *INT* 13h
(+4)	1	Код системи: 0 – невідома система; 1, 4 – DOS; 5 – розширений розділ DOS
(+5)	1	Номер головки для останнього сектора розділу

(+6)	2	Номер сектора і циліндра для останнього сектора розділу у форматі функції читання сектора *INT* 13h
(+8)	4	Відносний номер сектора початку розділу
(+12)	4	Розмір розділу в секторах

Завантажувальний запис BOOT. Найперший сектор логічного диска займає завантажувальний запис (*Boot Record*). Цей запис зчитується з активного розділу диска програмою головного завантажувального запису (*Master Boot Record*) і запускається на виконання. Завдання завантажувального запису – виконати завантаження ОС. Кожен тип ОС має свій завантажувальний запис. Навіть для різних версій однієї і тієї ж ОС програма завантаження може виконувати різні дії. Окрім програми початкового завантаження ОС, завантажувальний запис містить параметри, що описують характеристики логічного диску.

Формат *BOOT* наведено в табл. 1.3.

Таблиця 1.3

Формат завантажувального запису *BOOT*

Зсув	Розмір	Уміст
(+0)	3	Команда *JMP xxxx* – перехід типа *NEAR* на програму початкового завантаження
(+3)	8	Назва фірми-виробника ОС і версія (наприклад IBM 4.0)
(+11)	25	*Extended BPB* – розширений блок параметрів BIOS
(+36)	1	Фізичний номер дисковода (0 – гнучкий диск, 80*h* – жорсткий диск)
(+37)	1	Зарезервовано
(+38)	1	Символ ‘)’ – ознака розширеного завантажувального запису DOS 4.0
(+39)	4	Серійний номер диска (*volume serial number*), створюється під час форматування диска
(+43)	11	Мітка диска (*volume label*)
(+54)	8	Зарезервовано, зазвичай містить запис типу ‘FAT12’, яка ідентифікує формат таблиці розміщення файлів FAT

Таблиця розміщення файлів FAT. Таблиця розміщення файлів FAT є базою даних, що зв'язує кластери дискового простору з файлами. У цій базі для кожного кластера передбачається тільки один елемент. Перші два елементи містять інформацію про саму систему FAT. Третій і подальші елементи ставляться у відповідність до кластерів дискового простору, починаючи з першого кластера, відведеного для файлів. Елементи FAT можуть містити декілька спеціальних значень, які вказують, що кластер вільний, тобто не використаний жодним файлом (для FAT16 це значення становить 0000H); кластер містить один або декілька секторів з фізичними дефектами і не повинен використовуватися (для FAT16 це значення становить FFF7H); цей кластер – останній кластер файлу (для FAT16 це значення становить FFF8 FFFFH). Для будь-якого використовуваного файлом, але не останнього кластера елемент FAT містить номер наступного кластера, зайнятого файлом. Кожен каталог – незалежно кореневий або підкаталог – також є базою даних.

Розрахунки розмірів кластерів і секторів. У ФС FAT32 як елементи FAT, так і номери секторів – 32-розрядні. Ось що це означає: помножимо 4 294 967 296 різних 32-розрядних значень на 512 байт у секторі і отримаємо 2 Тбайт (2 199 023 255 552 байт), що є максимально можливою ємністю диска при використанні FAT32. Залежність розміру кластера від ємності диска наведено в табл. 1.4.

Таблиця 1.4

Залежність розміру кластера від ємності диска

Ємність диска, Гбайт	Розмір кластера, кбайт
Менше 8	4
Менше 16	8
Менше 32	16
32 и більше	32

Операційна система зберігає дві копії FAT, тому під елемент кожного кластера у FAT потрібно 8 байт. На двогігабайтному диску FAT займе 32 Мбайт його простору за розміру кластера 512 байт. А якщо розмір кластера становить 4 кбайт, для зберігання двох та-

блиць FAT буде потрібно всього 4 Мбайт, тобто буде заощаджено 28 Мбайт.

Інші зміни в FAT32:

1. У записі каталога для кожного файлу виділяється 4 байт для початкового кластера файлу (замість 2 байт у системі FAT16).

2. Операційна система завжди передбачала наявність на диску двох екземплярів FAT, але використовувався тільки один з них. З переходом до FAT32 ОС може працювати з будь-якою з цих копій.

3. Кореневий каталог, що мав раніше фіксований розмір, і строго визначене місце на диску, тепер можна вільно нарощувати в міру потреби подібно до підкаталогу.

4. Поєднання переміщуваного кореневого каталогу і можливості використання обох копій FAT – належні передумови для безперешкодної динамічної зміни розмірів розділів диска, наприклад зменшення розділу для вивільнення місця для іншої ОС. Цей новий підхід менш небезпечний, ніж ті, що застосовувалися в програмах незалежних постачальників для зміни розділів диска для роботи з FAT16.

Файлова система NTFS.Відмінності ФС NTFS:

– спроектована спеціально для Windows;

– підтримання транзакцій;

– всі дані зберігаються у файлах;

– підтримання 64-бітових вказівників для структур даних;

– підтримання імен файлів до 255 символів (повний шлях до файлу до 32767 символів) і кодувань Unicode;

– підтримання стиснення;

– підтримання шифрування (EFS);

– стійкість до відмов;

– підтримання декількох потоків даних для одного файлу.

Логічні та віртуальні номери кластерів NTFS. Файлова система NTFS працює з цілим числом дискових секторів як з мінімальним одиничним блоком даних. Такий блок називають *кластером*. Розмір кластера визначають під час форматування тому. Різні томи можуть мати різні розміри кластерів. Стосовно UNIX, можна відзначити, що термін «кластер» у Windows аналогічний терміну «розмір блока» ФС в UNIX. Файлова система обчислює розмір кластера, зважаючи на розмір диска та тип використовуваної ФС. Кластер може мати розмір 1–64 кбайт.

До кластерів належать декілька важливих параметрів NTFS. Перший параметр називають *логічним номером кластера* (Logical Cluster Number – LCN). Файлова система NTFS ділить диск на кластери й призначає кожному кластеру номер, починаючи з нуля. Цей номер називають LCN.

Іншим важливим параметром є *віртуальний номер кластера* (Virtual Cluster Number – VCN), який вказує номер кластера всередині певного файлу.

Віртуальний номер кластера дозволяє обчислити місцеположення атрибута, наприклад зсув даних усередині файлу, а логічний номер кластера дає змогу обчислити зсув відповідно тому або розділу для певного блока даних.

Максимальний розмір розділу NTFS обмежений лише розмірами жорстких дисків.

Диск NTFS умовно поділяють на дві частини. Перші 12% диска відводяться під зону MFT– ділянку, в якій зростає метафайл MFT. Запис будь-яких даних в цю ділянку неможливий. Зона MFT завжди утримується порожньою – це робиться для того, щоб найголовніший, службовий файл (MFT) не фрагментувався у разі свого зростання. Інші 88% диска є звичайним простором для зберігання файлів.

Фізична структура NTFS. Фізичну структуру NTFS показано на рис. 1.21.

Рис. 1.21. Фізична структура NTFS

Вільне місце на диску включає все фізично вільне місце й незаповнені зони MFT туди також включаються. Механізм використання зони MFT такий: коли файли вже не можна записувати в звичайну ділянку, зона MFT просто скорочується, звільняючи таким чином місце для запису файлів. У разі звільнення місця в зви-

чайній ділянці зона MFT може знову розширитись. При цьому не виключена ситуація, що в цій зоні залишилися й звичайні файли.

MFT та її структура. Файлова система NTFS є значним досягненням структуризації: кожен елемент системи є файл – навіть службова інформація.

Найголовніший файл на NTFS називається MFT, або Master File Table – головна таблиця файлів. Саме він розміщується в зоні MFT і є централізованим каталогом решти всіх файлів диска, та себе самого.

Файл MFT поділено на записи фіксованої ємності (зазвичай 1 кбайт), і кожен запис відповідає якому-небудь файлу (в загальному сенсі цього слова).

Перші 16 файлів мають службовий характер і недоступні ОС – вони називаються метафайлами (табл. 1.5), причому найперший метафайл – сама MFT. Ці перші 16 елементів MFT – єдина частина диска, що має фіксоване положення. Решта MFT файлу може розміщуватися, як і будь-який інший файл, у довільних місцях диска – відновити його положення можна за допомогою його самого, «зачепившись» за саму основу – за перший елемент MFT.

Таблиця 1.5

Метафайли

Файл	Номер запису	Опис
$*Mft*	0	Головна файлова таблиця
$*MftMirr*	1	Дзеркало MFT, що містить копію перших 16 файлів MFT
$*LogFile*	2	Файл журналу (для відновлення після збоїв і підтримання цілісності ФС)
$*Volume*	3	Опис тому, включаючи серійний номер тому, дату та час створення, а також прапор тому
$*AttrDef*	4	Визначення атрибута
. (точка)	5	Кореневий каталог
$*Bitmap*	6	Бітова карта розміщення кластерів
$*Boot*	7	Завантажувальний запис диска
$*BadClus*	8	Список пошкоджених кластерів

$Quota $*Secure*	9	У Windows NT 4 визначений як файл призначених для користувача квот, проте ніколи не використовувався. У Windows 2000 перевизначений як дескриптор безпеки
$*UpCase*	10	Таблиця верхнього реєстра
$*Extend*	11	Каталог, який містить файли $*Objid*, $*Quota* і $*UsrJrnl*. Використовується в Windows 2000 і пізніших версіях
---	12–23	Зарезервовані

Файли і потоки. Файлова система NTFS – це файл-об'єкт, що містить файли-об'єкти.

Файл-об'єкт щонайменше має запис в MFT. У цьому місці зберігається вся інформація про файл, за винятком власних даних. Ім'я файлу, розмір, положення на диску окремих фрагментів і т. ін. Якщо для інформації не вистачає одного запису MFT, то використовуються декілька, причому не обов'язково підряд.

У NTFS підтримується декілька потоків даних для одного файлу. Потік можна відкрити за допомогою функції Win32 API *CreateFile*, а ім'я потоку у вигляді :Ім'яПотоку може бути додано до імені файлу, наприклад *File*1:*Stream*25. Потоки підтримують запис, зчитування та незалежне від інших відкритих потоків блокування.

Попри те, що хоча NTFS і підтримує декілька потоків, множині утиліт і програм про це нічого не відомо.

Атрибути файлів NTFS. Як атрибут можна вказати ім'я файлу, список керування доступом до файлу й дані файлу (рис. 1.22). Якщо дані атрибуту мають невеликий розмір, вони будуть збережені безпосередньо в записі MFT, такі атрибути називають *резидентними*.

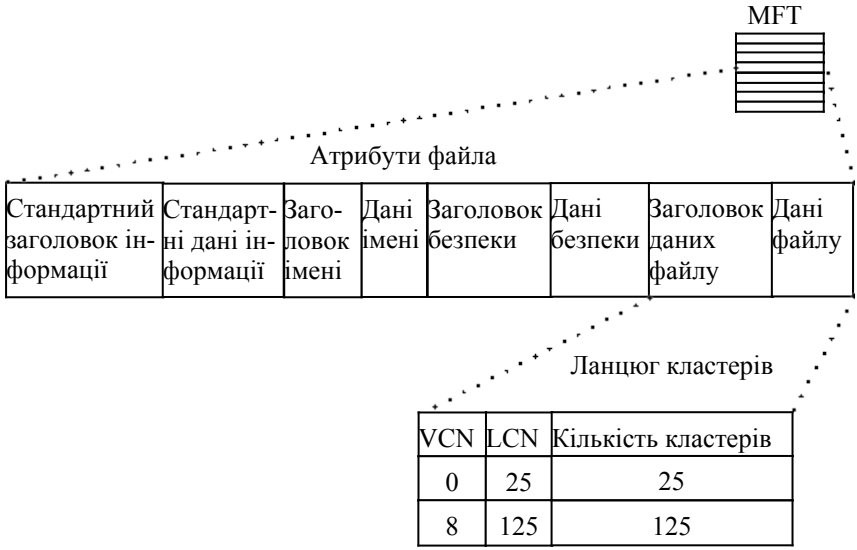


Рис. 1.22. Атрибути файлів

Якщо дані завеликі для зберігання в MFT, зберігається інформація про розміщення цих даних (номери кластерів, у яких розміщені необхідні дані). Такі атрибути називають *нерезидентними*.

Деякі атрибути файлів та їх опис наведено в табл. 1.6.

Таблиця 1.6

Атрибути файлів

Тип атрибута	Опис
Standard Information (стандартна інформація)	Включає бюджет зв'язку
Attribute List (список атрибутів)	Перераховує всі інші атрибути (тільки у великих файлах)
Filename (ім'я файлу)	Атрибут, що повторюється для довгих і коротких імен файлів. Довге ім'я файлу може містити до 255 символів Unicode. Коротке ім'я, – доступне для MS-DOS, вісім плюс три символи, без урахування регістра. Додаткові імена, або жорсткі зв'язки (hard

	links), використовуються POSIX і можуть бути також включені як додаткові атрибути імені файлу
Security Descriptor (дескриптор безпеки)	Фіксує інформацію про того, хто може звертатися до файлу, хто є його власником і т. ін.
Data (дані)	Містить дані файлу
Index Root (корінь индексів)	Використовується під час роботи з каталогами
Index Allocation (індексне розміщення)	Використовується під час роботи з каталогами
Volume Information (информація тому)	Використовується тільки в системному файлі тому і включає зокрема версію та ім'я тому
Bitmap (бітовий масив)	Надає інформацію про використання записів в MFT або каталозі
Extended Attribute Information (інформація розширеного атрибуту)	Використовується файловими серверами, які пов'язані із системами OS/2. Цей тип атрибуту не використовується Windows NT
Extended Attributes (розширені атрибути)	Використовується файловими серверами, які пов'язані із системами OS/2. Цей тип атрибуту не використовується Windows NT

Каталоги. Каталог у NTFS є специфічним файлом, що зберігає посилання на інші файли та каталоги, створюючи ієрархічну будову даних на диску.

Файл каталога поділений на блоки, кожний з яких містить ім'я файлу, базові атрибути та посилання на елемент MFT, який вже надає повну інформацію про елемент каталога. Внутрішня структура каталога є бінарним деревом.

Ведення журналу. Відмовостійка система NTFS цілком може набути коректного стану у разі будь-яких реальних збоїв. Будь-яка сучасна ФС ґрунтується на такому понятті, як *транзакція* – дія, що виконується цілком і коректно або не виконується взагалі. У NTFS просто не буває проміжних (помилкових або некоректних) станів – квант зміни даних не може бути поділений на до і після збоїв – він або здійснений, або відмінений.

Приклад 1.1. Виконується запис даних на диск. У тому місці, куди виконується запис даних, є фізичне пошкодження поверхні. Поведінка *NTFS*: транзакція запису відкатується цілком – система усвідомлює, що запис не проведений. Місце позначається як збійне, а дані записуються в інше місце – починається нова транзакція.

Приклад 1.2. Вимкнення живлення в момент запису файлу. На якій фазі зупинився запис, де є дані, а де немає? Рішення приймається на підставі журналу транзакцій. У метафайлі *$LogFile* система відзначає намір виконати запис. У разі перезавантаження цей файл вивчається на предмет наявності незавершених транзакцій, які були перервані аварією й результат яких непередбачуваний, – усі ці транзакції скасовуються: місце, у яке здійснювався запис, позначається знову як вільне, індекси й елементи MFT зводяться до стану, у якому вони були до збою, й система в цілому залишається стабільною.

Ведення журналу – лише засіб істотно скоротити кількість помилок і збоїв системи. Система відновлення NTFS гарантує коректність ФС, а не даних.

Операції, які журналюються системою, – це операції зі структурами самої системи, тобто з файлами та каталогами: додавання файлів, перейменування, перенесення, створення та видалення. Записуються до журналу також і операції дефрагментації – тобто переміщення фрагментів файлів.

Відкладений запис і контрольні точки ведення журналу. Відкладений запис – принцип кешування, за якого дані, призначені для запису на диск, якийсь час зберігаються в кеші й лише у вільний від інших занять час зберігаються фізично.

Відкладений запис істотно підвищує ефективність дискових операцій, оскільки таке кешування групує безліч операцій в одну. Ще одна перевага відкладеного запису – не заважати потрібним операціям зчитування і виконувати запис тільки тоді, коли система вільна і їй не потрібен доступ до диска для інших потреб.

Проблема відкладеного запису: відбувається розузгодження часу запису: деякі службові ділянки можуть бути оновлені, а деякі суміжні по сенсу – ще не оновлені, оскільки їх оновлення можливо відкладеться ще на декілька секунд і не відбудеться через збій.

За спроби почати операцію ведення журналу в лог тут же записується намір – наприклад, стерти файл. Це трапляється без затримок – на цьому етапі відкладений запис не виконується. Решта всіх операцій виконуються в затриманому режимі, тобто вони можуть бути частковими (і навіть не в тому порядку) або не виконуватися взагалі.

Єдина затримана операція, яка дещо відрізняється від простого запису – запис у лог про вдале завершення попередніх транзакцій – *контрольна точка*. Через певні проміжки часу – зазвичай через кожні декілька секунд – система в обов'язковому порядку скидає всі затримані операції на диск.

Після виконання цієї операції в журнал записується простий запис – контрольна точка – яка свідчить про те, що всі попередні операції виконано коректно на всіх рівнях – як на логічному, так і на фізичному.

Концепція дублювання інформації.

Приклад 1.3. Журнал містить запис – «файл *N* стирається». Звільняється місце, займане файлом, а потім видаляється інформація з фізичних структур MFT і каталога. Припустімо, що диск перебуває в активній роботі, і на місце, що звільнилося, миттєво записується інший файл. У цей момент часу відбувається збій. Система, завантажуючись, досліджує журнал і бачить незавершену операцію «файл *N* стирається», контрольна точка після якої зникає. Наступна фаза була б «відкат операції», тобто відновлення файлу. Проте місце, фізично займане файлом, містить вже інші дані.

Для недопущення таких ситуацій система вимушена застосовувати принцип «тимчасово зайнятого місця». Місце, звільнене яким-небудь об'єктом або записом про нього, не оголошується вільним доти, доки фізично не завершаться всі операції з логічними структурами.

Розріджені та стислі файли NTFS. Термін «розріджений» стосується файлів, що містять дані, після яких розміщується велика ділянка без даних, за нею – невеликий фрагмент даних і т.д. Файлова система NTFS не виділяє дискового простору для зберігання порожніх ділянок файлів.

Файлова система NTFS підтримує *стиснення файлів*, якщо файли розміщені в томі з ємністю кластера меншою за 4 кбайт. Дані стискаються і розархівуються «на льоту» в той момент часу, ко-

ли додаток викликає функції API для зчитування й запису. Стиснення може «вимикатися» або «вмикатися» для всього тому, каталога або окремого файлу. Стислі файли зберігаються в ланцюжках завдовжки по 16 кластерів кожний. Файлова система NTFS бере перші 16 кластерів і намагається їх стиснути. Для зчитування файлу NTFS потрібно визначити, чи стиснений він. Один зі способів зробити це – перевірити кінцевий логічний номер кластера файлу. Нульове значення цього параметра вказує на стисливість файлу.

Додаткові функції NTFS. Hard Links – це коли один і той самий файл має два імені (декілька вказівників файлу-каталога або різних каталогів указують на один і той же запис MFT). Припустімо, один і той же файл має імена 1.*txt* і 2.*txt*: якщо користувач зітре файл 1, залишиться файл 2. Якщо зітре файл 2, залишиться файл 1, тобто обидва імені з моменту створення абсолютно рівноправні. Файл фізично стирається лише тоді, коли буде видалено його останнє ім'я.

Symbolic Links (NT5) (аналог ярлика) дозволяє створювати віртуальні каталоги – так само, як і віртуальні диски командою subst у DOS.

У шифрованій ФС (EFS) використовується симетрична та асиметрична криптографія.

Фрагментація файлів у NTFS. Фрагменти даних можуть бути в різних кластерах жорсткого диска. У результаті після видалення файлів звільнений дисковий простір також стає фрагментованим. Чим вищий ступінь фрагментації жорсткого диска, тим нижча продуктивність ФС. Припустімо, що є два файли, необхідні для роботи певної програми: один на початку диска, другий, через порожній простір, в іншому місці. Операційній системі доводиться звертатися до обох цих файлів одночасно, що істотно гальмує систему, тим більше псує головку жорсткого диска.

Стверджувалося, що NTFS не схильна до фрагментації файлів. Це виявилось не зовсім так, і твердження змінили на таке: NTFS перешкоджає фрагментації. Виявилося, що і це не зовсім так. Тобто вона, звичайно, перешкоджає, але зиск від цього близький до нуля.

Стало зрозумілим, що NTFS – система, яка як ніяка інша схильна до фрагментації, що б не стверджувалося офіційно. Але всі внутрішні структури побудовані таким чином, що фрагментація не заважає швидко знаходити фрагменти даних.

Диск NTFS поділений на дві зони. На початку диска міститься зона MFT– зона, куди зростає MFT. Зона займає мінімум 12% диска, і запис даних в цю зону неможливий. Це зроблено для того, щоб MFT не фрагментувалася. Але коли решта диска заповнюється – зона скорочується рівно в два рази. В результаті, якщо NTFS працює при диску, заповненому близько на 90%, – фрагментація зростає.

Попутний наслідок – диск, заповнений більш ніж на 88%, дефрагментувати майже неможливо – навіть API дефрагментація не може переміщувати дані в зону MFT.

Алгоритм дій за будь-якого запису такий: береться якась певна ємність диска і заповнюється файлом до упору. Причому за дуже цікавим алгоритмом: спочатку заповнюються великі дірки, потім малі. Тобто типовий розподіл фрагментів файлу за розміром на фрагментованій NTFS виглядає так (розміри фрагментів): 16–16–16–16–16– [скачок назад] –15–15–15– [назад] –14–14–14 .... 1–1–1–1–1...

Так процес рухається до найдрібніших дірок в 1 кластер, попри те, що на диску напевно є і набагато більші ділянки вільного місця.

Засоби дефрагментації. У NTFS існує стандартна API дефрагментація, що має цікаве обмеження для переміщення блоків файлів: за один раз можна переміщати не менше 16 кластерів, причому починатися ці кластери повинні з позиції, кратної 16 кластерам у файлі. Загалом операція виконується винятково по 16 кластерів. Наслідки:

1) у дірку вільного місця менше 16 кластерів не можна нічого перемістити (окрім стислих файлів);

2) файл, будучи переміщеним в інше місце, залишає після себе (на новому місці) «тимчасово зайняте місце», доповнюючи його до кратності 16 кластерам;

3) у разі спроби неправильно («не кратно 16») перемістити файл результат часто непередбачуваний. Щось округляється, щось просто не переміщається. Проте все місце дії сильно розсипається «тимчасово зайнятим місцем».

Процес стандартної дефрагментації складається з таких фаз:

1) виймання файлів з зони MFT. Не спеціально – просто назад туди їх покласти неможливо;

2) дефрагментація файлів. Ускладнюється обмеженнями кратності переміщень;

3) дефрагментація MFT, файлу підкачування (*pagefile.sys*) та каталогів;

4) складання файлів ближче до початку –дефрагментація вільного місця.

Підтримання програмного RAID. Ведення журналу NTFS не гарантує від збоїв із втратою призначеної для користувача інформації. Тим часом, NTFS пропонує декілька варіантів створення систем, де в належних умовах гарантується абсолютно все. Можна також використовувати більшу кількість дисків для забезпечення не підвищеної надійності, а, навпаки, підвищеної швидкості – або того й того одночасно.

RAID (Redundant Array of Inexpensive Disks) – надмірний масив недорогих дисків. Ця технологія полягає в одночасному використанні декількох дискових пристроїв для забезпечення характеристик надійності або швидкості, яких немає у нагромаджувачах окремо.

Порівняння файлових систем NTFS і FAT. *Пошук даних файлу* (швидкість доступу до довільного фрагмента файлу). Цей параметр показує, наскільки сильно сама ФС потерпає від фрагментації файлів.

Абсолютний лідер – система FAT16, яка ніколи не змусить систему виконувати зайві дискові операції. Наступна система – NTFS – система, що також не вимагає зчитування зайвої інформації, принаймні доти, доки файл має прийнятну кількість фрагментів. Файлова система FAT32 зазнає величезних труднощів, аж до зчитування зайвих сотень кілобайтів з ділянки FAT, якщо файл розкиданий по різних ділянках диска. Якщо файл фрагментований, але містить компактно укладені фрагменти, то FAT32 все ж не зазнає великих труднощів, оскільки фізичний доступ до FAT буде також компактний і буферизуватиме.

Пошук вільного місця. Ця операція виконується в тому випадку, коли файл потрібно створити з нуля або скопіювати на диск. Пошук місця під фізичні дані файлу залежить від того, як зберігається інформація про зайняті ділянки диска. Цей параметр показує, наскільки швидко система зможе знайти місце для записування на диск нових даних і які операції їй доведеться для цього виконати.

Файлова система NTFS – найбільш ефективна система знаходження вільного місця. Варто відзначити, що діяти «в лоб» на FAT16 або FAT32 дуже повільно, тому для знаходження вільного місця в цих системах застосовуються різні методи оптимізації, унаслідок чого і там досягається прийнятна швидкість.

Робота з каталогами і файлами. Впливає на швидкість виконання будь-яких операцій з файлом, зокрема – на швидкість будь-якої операції доступу до файлу, особливо в каталогах з великою кількістю файлів (тисячі).

Структура каталогів у NTFS теоретично набагато ефективніша, але якщо розмір каталога становить декілька сотень файлів, це не має значення. Для малих і середніх каталогів NTFS на практиці має меншу швидкодію.

Переваги каталогів NTFS стають реальними та незаперечними тільки в тому випадку, якщо в одному каталозі містяться тисячі файлів – у цьому випадку швидкодія компенсує фрагментацію самого каталога й труднощі фізичного звернення до даних (уперше – далі каталог кешується). Напружена робота з каталогами, що містять близько тисячі й більше файлів, відбувається у NTFS у декілька разів швидше, а іноді виграш у швидкості порівняно з FAT і FAT32 досягає десятків разів.

Переваги FAT. Ефективна робота потребує мало оперативної пам'яті. Швидкою є робота з малими і середніми каталогами. Диск здійснює в середньому меншу кількість рухів головок (порівняно з NTFS). Ефективною є робота на повільних дисках.

Недоліки FAT. Катастрофічна втрата швидкодії зі збільшенням фрагментації, особливо для великих дисків (тільки FAT32); складності з довільним доступом до великих файлів; дуже повільна робота з каталогами, що містять велику кількість файлів.

Переваги NTFS. Фрагментація файлів не має ніяких наслідків для самої ФС. Складність структури каталогів і кількість файлів в одному каталозі також не чинить особливих перешкод швидкодії. Швидкий доступ до довільного фрагмента файлу. Дуже швидкий доступ до маленьких файлів.

Недоліки NTFS. Істотні вимоги до пам'яті системи (від 64 Мбайт). Повільні диски і контролери без *Bus Mastering* дуже знижують швидкодію NTFS. Робота з каталогами середніх розмірів ускладнюється тим, що вони майже завжди фрагментовані. Диск, що

довго працює в заповненому на 80 – 90% стані, показуватиме украй низьку швидкодію.

Файлові системи UNIX System V Release 4. У UNIX System V Release 4 реалізовується механізм віртуальної ФС VFS (Virtual File System), який дозволяє ядру системи одночасно підтримувати декілька різних типів ФС.

Типи файлових систем, підтримуваних в UNIX System V Release 4:

s5 – традиційна ФС UNIX System V, підтримувана в ранніх версіях UNIX System V від AT&T;

ufs – ФС, використовувана за замовчуванням в UNIX System V Release 4, яка походить від ФС SUNOS, що у свою чергу, походить від ФС Berkeley Fast File System (FFS);

nfs – адаптація відомої ФС NFS фірми Sun Microsystems, яка дозволяє розділяти файли та каталоги в гетерогенних мережах;

rfs – ФС Remote File Sharing з UNIX System V Release 3. За функціональними можливостями близька до NFS, але вимагає на кожному комп'ютері встановлення UNIX System V Release 3 або пізніших версій цієї ОС;

veritas – відмовостійка ФС з транзакційним механізмом операцій;

specfs – новий тип ФС, що забезпечує єдиний інтерфейс до всіх спеціальних файлів, що описуються в каталозі /dev;

fifofs – нова ФС, що використовує механізм VFS для реалізації файлів FIFO, відомих також як конвеєри (*pipes*), у середовищі STREAMS;

bfs – завантажувальна ФС. Призначена для швидкого і простого завантаження і тому є дуже простою плоскою ФС, що складається з одного каталога;

/proc – ФС цього типу забезпечує доступ до образу адресного простору кожного активного процесу системи, зазвичай використовується для відладжування і трасування;

/dev/fd – цей тип ФС забезпечує зручний метод посилань на дескриптори відкритих файлів.

Традиційна файлова система S5. Типи файлів. Файлова система UNIX s5 підтримує логічну організацію файлу у вигляді послідовності байтів. За функціональним призначенням розрізняються звичайні файли, каталоги та спеціальні файли.

Звичайні файли містять ту інформацію, яку заносить у них користувач або яка утворюється в результаті роботи системних і призначених для користувача програм, тобто ОС не накладає ніяких обмежень на структуру та характер інформації, що зберігається в звичайних файлах.

Каталог – файл, що містить службову інформацію ФС про групу файлів, що входять у цей каталог. У каталог можуть входити звичайні, спеціальні файли і каталоги нижчого рівня.

Спеціальний файл – фіктивний файл, що асоціюється з будь-яким пристроєм введення-виведення, використовується для уніфікації механізму доступу до файлів і зовнішніх пристроїв.

Структура ФС. Файлова система s5 має ієрархічну структуру, в якій рівні створюються за рахунок каталогів, що містять інформацію про файли нижчого рівня.

Кореневий каталог ФС завжди розміщується на системному пристрої (диск, що має таку ознаку). Проте це не означає, що й решта файлів може міститися тільки на ньому. Для зв'язку ієрархій файлів, розміщених на різних носіях, застосовується монтування ФС, що виконується системним викликом mount.

Операція монтування полягає в такому: у кореневій ФС вибирається деякий існуючий каталог, що містить один порожній файл. Після виконання монтування вибраний каталог стає кореневим каталогом іншої ФС. Через цей каталог змонтована ФС приєднується як гілка до загального дерева.

Привілеї доступу. В UNIX s5 всі користувачі за доступом до файлу діляться на три категорії: власник, член групи власника та всі інші.

Група – це користувачі, які об'єднані за якою-небудь ознакою, наприклад, за належністю до однієї розробки. Окрім цього, в системі існує суперкористувач, що має абсолютні права доступу до всіх файлів системи.

Визначено три види доступу до файлу – зчитування, запис і виконання. Привілеї доступу до кожного файлу визначені для кожної з трьох категорій користувачів і для кожної з трьох операцій доступу. Початкові значення прав доступу до файлу встановлюються під час його створення ОС і можуть змінюватися його власником або суперкористувачем.

Фізична організація файлу. У загальному випадку файл може розміщуватися в несуміжних блоках дискової пам'яті. Логічна послі-

довність блоків у файлі задається набором з 13 елементів. Перші 10 елементів призначаються для безпосередньої вказівки номерів перших 10 блоків файлу. Якщо розмір файлу перевищує 10 блоків, то в 11-му елементі указується номер блока, в якому міститься список наступних 128 блоків файлу. Якщо файл має розмір більший, ніж 10+128 блоків, то використовується 12-й елемент, що містить номер блока, у якому вказуються номери 128 блоків, кожен з яких може містити ще по 128 номерів блоків файлу. Таким чином, 12-й елемент використовується для дворівневої непрямої адресації. У випадку, якщо файл більший, ніж 10+128+1282 блоки, то використовується 13-й елемент для трирівневої непрямої адресації. За такого способу адресації граничний розмір файлу становить 2 113 674 блоки. Традиційна ФС s5 підтримує ємності блоків 512, 1024 або 2048 байт.

Інформація про файл. Індексні дескриптори. Уся необхідна ОС інформація про файл, окрім його символьного імені, зберігається в спеціальній системній таблиці – індексному дескрипторі (*inode*) файлу. Індексні дескриптори всіх файлів мають однакову ємність – 64 байт і містять дані про тип файлу, фізичне розміщення файлу на диску (описані вище 13 елементів), ємність у байтах, дату створення останньої модифікації та останнього звернення до файлу, привілеї доступу та деяку іншу інформацію. Індексні дескриптори пронумеровані й зберігаються в спеціальній ділянці ФС. Номер індексного дескриптора є унікальним іменем файлу. Відповідність між повними символьними іменами файлів і їх унікальними іменами встановлюється за допомогою ієрархії каталогів.

Структура каталога. Каталог є сукупністю записів про всі файли та каталоги, що входять до нього. Кожен запис складається з 16 байт; 14 байт відводиться під коротке символьне ім’я файлу або каталога, а 2 байт – під номер індексного дескриптора цього файлу. У каталозі файлової системи s5 безпосередньо не вказуються характеристики файлів. Така організація ФС дозволяє з найменшими витратами перебудовувати систему каталогів. Наприклад, у разі включення або виключення файлу з каталога відбувається маніпулювання меншими обсягами інформації. Крім того, для включення одного й того ж файлу в різні каталоги не потрібно мати декількох копій як характеристик, так і самих файлів. Для цього в індексному дескрипторі ведеться облік посилань на цей файл зі всіх каталогів.

Як тільки кількість посилань дорівнюватиме нулю, індексний дескриптор цього файлу знищується.

Структура диска s5. Весь дисковий простір, відведений під ФС, поділяють на чотири блоки (рис. 1.23):

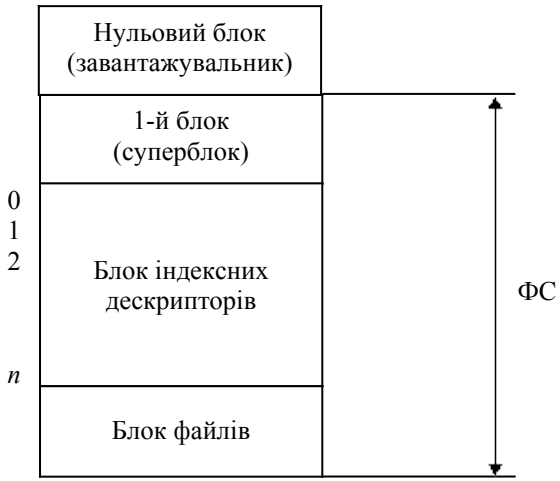


Рис. 1.23. Структура диска

– *завантажувальний блок* (*boot*), у якому зберігається завантажувач ОС;

– *суперблок* (*superblock*) – містить найзагальнішу інформацію про ФС: розмір ФС, розмір блоку індексних дескрипторів, кількість індексних дескрипторів, список вільних блоків і список вільних індексних дескрипторів, а також іншу адміністративну інформацію;

– *блок індексних дескрипторів*, порядок розміщення індексних дескрипторів у якій відповідає їх номерам;

– *блок даних*, у якій розміщені як звичайні файли, так і файли-каталоги. Спеціальні файли подані у ФС тільки записами у відповідних каталогах та індексними дескрипторами спеціального формату, але місця в блоці даних не займають.

Доступ до файлу. Доступ до файлу здійснюється послідовним переглядом усього ланцюга каталогів, що входять у повне ім’я файлу, та відповідних їм індексних дескрипторів. Пошук завершується після отримання всіх характеристик з індексного дескрип-

тора заданого файлу. Ця процедура вимагає в загальному випадку декількох звернень до диска пропорційно до кількості складових у повному імені файлу. Для зменшення середнього часу доступу до файлу його дескриптор копіюється в спеціальну системну ділянку пам'яті. Копіювання індексного дескриптора входить до процедури відкриття файлу.

Відкриття файлу. Для відкриття файлу ядро виконує такі дії:

– перевіряє існування файлу; якщо файлу немає, то чи можна його створити. Якщо він є, то чи дозволений до нього доступ необхідного вигляду;

– копіює індексний дескриптор з диска в оперативну пам'ять; якщо з вказаним файлом уже ведеться робота, то нова копія індексного дескриптора не створюється;

– створює в ділянці ядра структуру, призначену для відображення поточного стану операції обміну даними з указаним файлом. Ця структура, названа *file*, містить дані про тип операції (зчитування, запис або зчитування й запис), про кількість лічених або записаних байтів, вказівник на байт файлу, з яким виконується операція;

– робить відмітку в контексті процесу, що видав системний виклик на операцію з файлом.

Віртуальна файлова система VFS. Ідеологія VFS. Система VFS не орієнтується на яку-небудь конкретну ФС, механізми реалізації ФС повністю приховані як від користувача, так і від додатків. В ОС немає системних викликів, призначених для роботи зі специфічними типами ФС, а є абстрактні виклики типу *open* (відкриття), *read* (зчитування), *write* (запис) та інші, які мають змістовний опис, узагальнювальний деяким чином зміст цих операцій в найбільш популярних типах ФС (наприклад, s5, ufs, nfs і т.п.). Система VFS також надає ядру можливість операції з ФС як з єдиним цілим: операції монтування та демонтажу, а також операції отримання загальних характеристик конкретної ФС (розміру блока, кількості вільних і зайнятих блоків і т. ін.) у єдиній формі. Якщо конкретний тип ФС не підтримує якоїсь абстрактної операції VFS, то ФС повинна повернути ядру код повернення, що сповіщає про цей факт.

Інформація про файли та типи файлів VFS. У VFS вся інформація про файли розділена на дві частини – незалежну від типу ФС, яка зберігається в спеціальній структурі ядра – структурі *vnode*, і за-

лежну від типу ФС – структура *inode*, формат якої на рівні VFS не визначений, а використовується тільки посилання на неї в структурі *vnode*. Ім'я inode не означає, що ця структура збігається зі структурою індексного дескриптора inode ФС s5.

Віртуальна ФС VFS підтримує такі типи файлів:

– звичайні файли;

– каталоги;

– спеціальні файли;

– іменовані конвеєри;

– символьні зв'язки.

Змістовний опис звичайних файлів, каталогів і спеціальних файлів та зв'язків не відрізняється від їх опису у ФС s5.

Символьні зв'язки. М'який зв'язок, названий символьним зв'язком і реалізується за допомогою системного виклику *symlink*.

Символьний зв'язок – це файл даних, що містить ім'я файлу, з яким передбачається встановити зв'язок.

Символьний зв'язок може бути створений навіть з неіснуючим файлом. Під час створення символьного зв'язку утворюється як новий вхід у каталозі, так і новий індексний дескриптор *inode*. Окрім цього, резервується окремий блок даних для зберігання повного імені файлу, на який він посилається.

Є три системні виклики, які стосуються символьних зв'язків:

– *readlink* – зчитування повного імені файлу або каталога, на який посилається символьний зв'язок. Ця інформація зберігається в блоці, пов'язаному із символьним зв'язком;

– *lstat* – аналогічний системному виклику *stat*, але використовується для отримання інформації про сам зв'язок;

– *lchown* – аналогічний системному виклику *chown*, але використовується для зміни власника самого символьного зв'язку.

Реалізація файлової системи VFS. UNIX System V Release 4 має масив структур vfssw, кожна з яких описує ФС конкретного типу, яка може бути встановлена в системі. Структура vfssw складається з чотирьох полів:

– символьного імені ФС;

– вказівника на функцію ініціалізації ФС;

– вказівника на структуру, що описує функції, які реалізують абстрактні операції VFS у цій конкретній ФС;

– прапорів, які не використовуються в описуваній версії UNIX.

Операції над файловою системою. Операції, виконувані над ФС VFS, наведено в табл. 1.7.

Таблиця 1.7

Операції із ФС

VFS_MOUNT	Монтування
VFS_UNMOUNT	Розмонтування
VFS_ROOT	Отримання vnode для кореня
VFS_STATVFS	Отримання статистики
VFS_SYNC	Виштовхування буферів на диск
VFS_VGET	отримання vnode за номером дескриптора файлу
VFS_MOUNTROOT	Монтування кореневої ФС

Абстрактні операції із файлами. Абстрактні операції, виконувані із файлами, наведено в табл. 1.8.

Таблиця 1.8

Абстрактні операції із файлами

VOP_OPEN	Відкрити файл
VOP_CLOSE	Закрити файл
VOP_READ	Зчитувати з файлу
VOP_WRITE	Записати в файл
VOP_IOCTL	Керування введенням/виведенням
VOP_SETFL	Встановити прапори статусу
VOP_GETATTR	Отримати атрибути файлу
VOP_SETATTR	Встановити атрибути файлу

VOP_LOOKUP	Знайти vnode за іменем файлу
VOP_CREATE	Створити файл
VOP_REMOVE	Видалити файл
VOP_LINK	Зв'язати файл
VOP_MAP	Відобразити файл у пам'ять

Структура vnodeops. Окрім операцій із ФС, для кожного типу ФС (s5, ufs), установленої в ОС, необхідно описати спосіб реалізації абстрактних операцій із файлами, які допускаються у VFS. Цей спосіб описується для кожного типу ФС у структурі *vnodeops*. Як видно зі складу списку абстрактних операцій, вони утворені об'єднанням операцій, характерних для найбільш поширених ФС UNIX. Для того щоб звернення до специфічних функцій не залежало від типу ФС, для кожної операції у *vnodeops* визначається макрос із загальним для всіх типів ФС іменем, наприклад, *VOP_OPEN*, *VOP_CLOSE*, *VOP_READ* і т. ін. Ці макроси визначаються у файлі й відповідають системним викликам. Таким чином, у структурі vnodeops приховані залежні від типу ФС реалізації стандартного набору операцій над файлами. Навіть якщо ФС якого-небудь конкретного типу не підтримує певну операцію над своїми файлами, вона повинна створити відповідну функцію, яка виконує деякий мінімум дій: або відразу повертає успішний код завершення, або повертає код помилки. Для аналізу та оброблення повного імені файлу в VFS використовується операція *VOP_LOOKUP*, яка дозволяє за іменем файлу знайти посилання на його структуру *vnode*.

Структура vnode. Структура vnode (рис. 1.24) використовується ядром для зв'язку файлу з певним типом ФС через поле *v_vfsp* і конкретними реалізаціями файлових операцій через поле *v_op*. Поле *v_pages* використовується для вказівки на таблицю фізичних сторінок пам'яті у разі, коли файл відображається у фізичну пам'ять. У *vnode* також міститься тип файлу та вказівник на залежну від типу ФС частину опису характеристик файлу – структуру *inode*, що зазвичай містить адресну інформацію про розміщення

файлу на носії та про права доступу до файлу. Окрім цього, *vnode* використовується ядром для зберігання інформації про блокування (*locks*), застосовані процесами до окремих ділянок файлу.

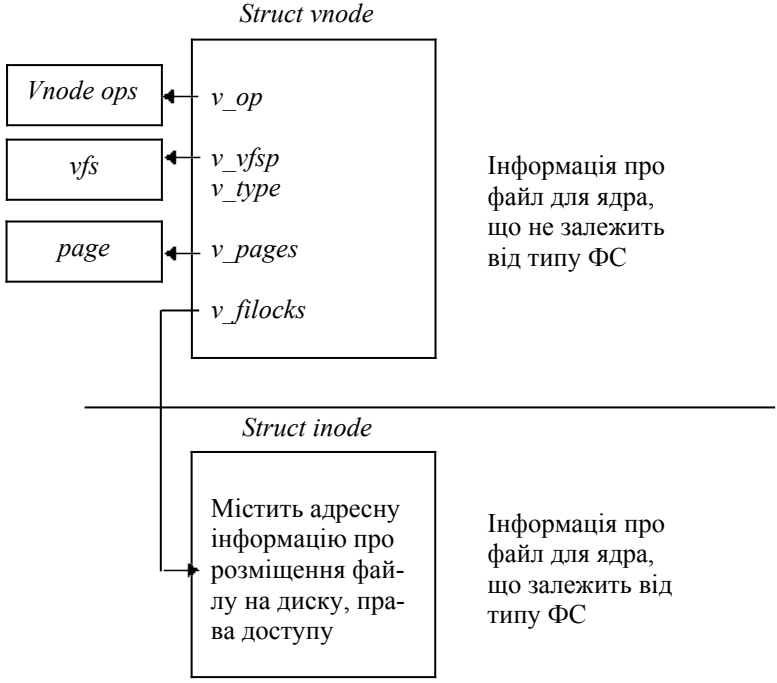


Рис. 1.24. Структура *vnode*

Структура FILE. Із кожним відкриттям процесом файлу ядро створює в системній ділянці пам'яті нову структуру типу *file*, яка, як і в разі традиційної ФС s5, описує як відкритий файл, так і операції, які процес збирається проводити з файлом (наприклад, зчитування). Структура *file* містить такі поля:

– *flag* – визначення режиму відкриття (тільки для зчитування, для зчитування й запису і т. ін.);

– *S truct vnode * f_vnode* – вказівник на структуру *vnode* (що замінив порівняно з s5 вказівник на *inode*);

– *offset* – зсув у файлі під час операцій зчитування/записування;

– *struct cred * f_cred* – вказівник на структуру, що містить права процесу, який відкрив файл (структура міститься у дескрипторі процесу);

– вказівники на попередню та подальшу структуру типу *file*, що зв'язують усі такі структури в список.

File і vnode. На відміну від структур типу *file* структури типу *vnode* заводяться ОС для кожного активного (відкритого) файлу в єдиному екземплярі, тому структури *file* можуть посилатися на одну й ту ж структуру *vnode*.

Структури *vnode* не зв'язані в який-небудь список. Вони з'являються на вимогу в системному пулі пам'яті й приєднуються до структури даних, яка ініціювала появу цього *vnode*, за допомогою відповідного вказівника. Наприклад, у разі структури *file* в ній використовується вказівник *f_vnode* на відповідну структуру *vnode*, що описує потрібний файл. Аналогічно, якщо файл пов'язаний з образом процесу (тобто це файл, що містить виконуваний модуль), то сегмент пам'яті, що містить частини цього файлу, відображається за допомогою вказівника *vp* (у структурі *segvn_data*) на *vnode* цього файлу.

Усі операції з файлами в UNIX System V Release 4 виконуються за допомогою зв'язаної з файлом структури *vnode*. Коли процес запрошує операцію з файлом (наприклад, операцію *open*), то незалежна від типу ФС частина ОС передає керування залежної від типу ФС частини ОС для виконання операції. Якщо залежна частина виявляє, що структури *vnode*, яка описує потрібний файл, немає в оперативній пам'яті, то залежна частина заводить для нього нову структуру *vnode*.

Система введення-виведення в операційній системі Unix System V. Підсистема буферизації. Основу системи введення-виведення ОС UNIX складають драйвери зовнішніх пристроїв і засобу буферизації даних. Операційна система UNIX використовує два різні інтерфейси із зовнішніми пристроями: *байт-орієнтований* і *блок-орієнтований.*

Будь-який запит на введення-виведення до блок-орієнтованого пристрою перетвориться в запит до підсистеми буферизації, яка є буферним пулом і комплексом програм керування цим пулом. Буферний пул складається з буферів, що містяться в режимі ядра. Розмір окремого буфера дорівнює розміру блока даних на диску. З

кожним буфером пов'язана спеціальна структура – заголовок буфера, в якому міститься така інформація:

– дані про стан буфера:
  • зайнятий/вільний,
  • зчитування/записування,
  • ознака відкладеного запису,
  • помилка введення-виведення;

– дані про пристрій – джерело інформації, що міститься в цьому буфері:
  • тип пристрою,
  • номер пристрою,
  • номер блоку на пристрої,
  • адреса буфера.

Драйвери. Драйвер – це сукупність програм (секцій), призначена для керування передаванням даних між зовнішнім пристроєм і оперативною пам'яттю.

Зв'язок ядра системи з драйверами забезпечується за допомогою двох системних таблиць:

– *bdevsw* – таблиця блок-орієнтованих пристроїв;

– *cdevsw* – таблиця байт-орієнтованих пристроїв.

Для зв'язку використовується така інформація з індексних дескрипторів спеціальних файлів:

– клас пристрою (байт-орієнтований або блок-орієнтований);

– тип пристрою (стрічка, гнучкий диск, жорсткий диск, друкувальний пристрій, дисплей, канал зв'язку і т. ін.);

– номер пристрою.

1.7. Користувацький інтерфейс

Як випливає з визначення ОС вона повинна надавати призначений для користувача інтерфейс. Щонайменше ОС повинна надавати командну оболонку (*shell*), яка дає користувачу можливість тим або іншим способом запустити його прикладну програму. Проте в деяких випадках, наприклад, у вбудовуваних контролерах і інших спеціалізованих застосуваннях, такої оболонки не може бути. При цьому або система взагалі функціонує без втручання людини, або користувач працює лише з однією прикладною програмою. Крім того, ОС часто надають засоби для реалізації графічного, призначеного

для користувача, інтерфейсу прикладними програмами. Часто виявляється складно розмежувати ядро ОС і ці засоби, особливо якщо стандартна оболонка ОС реалізована з їх використанням. У деяких системах, наприклад у MS Windows 3.x і MAC OS, майже все ядро складається із засобів реалізації графічного інтерфейсу.

Натепер оформилося два принципово різні підходи до організації користувацького інтерфейсу. Перший, історично раніший підхід, полягає в наданні користувачу командної мови, у якій програми запускаються окремими командами. Цей підхід відомий як інтерфейс командного рядка (Command Line Interface – CLI). Другий, альтернативний підхід полягає в символічному зображенні доступних дій у вигляді картинок – ікон icons на екрані та наданні користувачу можливості вибирати дії за допомогою мишки або іншого координатного пристрою введення. Цей підхід відомий як графічний користувацький інтерфейс (Graphical User Interface – GUI).

Розробники сучасних ОС зазвичай надають засоби для реалізації обох підходів і оболонок, що використовують обидва типи інтерфейсів. Проте серед користувачів щодо переваг різних підходів точаться гострі дискусії. Спробуємо далі викласти аргументи на користь кожного з підходів.

1.7.1. Користувацький інтерфейс CLI

Мова є одним з найбільших досягнень людства. Багато учених і філософів обґрунтовано вважають, що саме мова визначає межу між людиною і твариною. За сучасними уявленнями людський мозок містить природжені структури, що полегшують розуміння текстів природними і штучними мовами та генерацію таких текстів.

У свою чергу, комп'ютер, зокрема універсальний нейманівський комп'ютер, є *мовною машиною* – пристрій, який виконує послідовність пропозицій деякої нормалізованої мови.

Мова комп'ютера може бути лише функціональною підмножиною природної мови, оскільки обчислювальних можливостей сучасних комп'ютерів явно не досить для відображення всього багатства й складності реального світу. У зв'язку з цим видається доцільним використовувати повністю синтетичні мови, а не підмножини природних, адже людині простіше вивчити нову

мову, ніж привчатися користуватися обмеженою підмножиною рідної мови.

Удало спроектовані штучні мови зручні й для комп'ютера: інтерпретатори повнофункціональних командних мов становлять десятки, у гіршому разі сотні кілобайтів оперативної пам'яті й забезпечують дуже високу швидкість і ефективність використання ресурсів системи. Тому синтетична мова як засіб людино-машинного спілкування є найкращим і найбільш природним вибором для обох сторін.

Командні мови дозволяють природним чином перейти до написання командних файлів або скриптів (*scripts*), що дозволяють автоматизувати часто виконувані завдання. Важко визначити межу між написанням скриптів і програмуванням, оскільки написання скриптів і навіть інтерактивне використання командної мови є окремим випадком програмування.

Сучасні інтерактивні командні процесори вирішують майже всі проблеми командних мов попередніх поколінь:

– виправлення друкарських помилок у командах і набір послідовностей однакових або схожих команд здійснюються з використанням засобів згадування раніше набраних рядків – «історії». Сучасні командні процесори забезпечують гнучкі засоби пошуку команд у історичному списку, їх редагування, повторного використання окремих частин цих команд і т. ін.;

– набір довгих імен файлів, каталогів та інших об'єктів полегшується автоматичним розширенням імен. Цей засіб, реалізований в багатьох командних процесорах ОС сім'ї Unix, дозволяє, набравши перші декілька символів імені, розширити це ім'я до повного або отримати список усіх імен, що починаються із заданої послідовності букв;

– незручні команди, такі, що важко запам'ятовуються або чимось не влаштовують користувача, можуть бути перейменовані з використанням синонімів (*aliases*). Цей же механізм можна використовувати для скорочення часто виконуваних складних команд.

Важливою перевагою хороших командних мов порівняно з GUI є їх алгоритмічна повнота: у GUI користувач обмежений тими можливостями, для яких розробник програми зобразив іконки або вигадав пункти в меню. Командні ж мови можуть використовуватися для вирішення будь-яких алгоритмізованих завдань. Цікаво,

що останнє досягнення в галузі користувацького інтерфейсу – розпізнавання мови й мовне керування – означає, по суті, повернення до командної мови, з тією лише різницею, що команди промовляються, а не вводяться з клавіатури. Імовірно, слід очікувати появу нового покоління синтетичних командних мов, що відповідають вимогам мовного введення команд.

1.7.2. Користувацький інтерфейс GUI

Командні мови потребують для їх вивчення часу й зусиль і не лише інтелектуальних, але й емоційних. У цьому сенсі перші хвилини спілкування з невідомою системою для користувача є найскладнішими, коли він відчуває розгубленість і не може сформулювати бажану дію не лише незнайомою синтетичною мовою, але навіть і рідною природною.

Перше негативне враження може створити стійкий страх перед комп'ютером, утрудняючи його ефективне використання та вивчення. Навпаки, графічний інтерфейс дає змогу новому користувачу швидко переглянути доступні можливості та вибрати бажану. У багатьох випадках наочність варіантів виявляється важливішою за безліч можливостей.

Проте в деяких випадках зайве різномаїття варіантів може просто заплутати користувача. Не потрібно забувати, що людина здатна одночасно оперувати лише досить обмеженою кількістю об'єктів і параметрів; для людини і більшості теплокровних тварин ця кількість обмежена 6 – 7 об'єктами.

Навіть після освоєння базових можливостей системи людина може забути команду для виконання якої-небудь операції; в цьому розумінні графічні інтерфейси, де всі можливості перед очима, виявляються переважними.

Твердження про те, що GUI обмежує користувача заздалегідь обумовленими можливостями не відповідає дійсності: добре продумані інтерфейси забезпечують майже таку ж гнучкість в комбінації операцій, як і командні мови. Можливість же записувати та знову програвати послідовності дій в багатьох ситуаціях може замінити командні файли.

Багато сучасних настільних систем використовуються для додатків, які самі по собі потребують високоякісної графіки та вели-

кої обчислювальної потужності (поліграфічні роботи, синтез та оброблення відеоданих і т. ін.). Добре продуманий графічний інтерфейс з правильно підібраними кольорами, естетично зображеними елементами вікон сам по собі приємний для очей.

1.8. Периферійні пристрої

Периферійні або зовнішні пристрої – це пристрої, розміщені поза системним блоком і задіяні на певному етапі оброблення інформації. Передусім це пристрої фіксації вихідних результатів: принтери, плотери, модеми, сканери тощо. Поняття «периферійні пристрої» досить умовне. Таким пристроєм є, наприклад, нагромаджувач на компакт-дисках, якщо він виконаний у вигляді самостійного блока й приєднується спеціальним кабелем до зовнішнього рознімного з'єднання системного блока. І навпаки, модем може бути вбудованим, тобто конструктивно виконаним як плата розширення, і тоді немає підстав вважати його периферійним пристроєм.

1.8.1. Драйвери зовнішніх пристроїв, призначення, приклади

Драйвер – це програма, яка відповідає за роботу пристрою, містить набір команд для цього пристрою і забезпечує зв'язок між комп'ютером і пристроєм.

Спілкування користувача із зовнішніми пристроями (монітором, принтером, каналами зв'язку з іншими ПК тощо) здійснюється через спеціальні програми ОС – *драйвери* (від англ. *driver* – водій). Так, якщо потрібно вивести на екран монітора вміст певного файлу, досить точно вказати номер диска та ім'я файлу й дати команду його виведення на монітор. Відповідно до цієї команди ФС за каталогом визначить, де саме на диску міститься потрібний файл, і надасть інформацію драйверу, який запустить потрібний дисковод, переведе зчитувальні головки на потрібну доріжку, прочитає файл в оперативну пам'ять і виведе інформацію на екран монітора.

Таким чином, драйвери забезпечують взаємодію комп'ютера з його зовнішніми пристроями та відображають їх специфіку. Драйвери пристроїв постачаються виробниками разом з новими пристроями на компакт-дисках або дискетах.

1.8.2. Принтер

Комп'ютерний принтер (від англ. *printer* – друкар) – пристрій для друку інформації на папір.

Процес друку називають *виходом на друк*, а отриманий документ, – *роздруківкою* або *твердою копією*. Принтери мають перетворювач цифрової інформації (тексти, фото, графіки), що зберігається в запам'ятовувальних пристроях комп'ютера, фотоапарата та цифрової пам'яті, у спеціальну машинну мову.

За технологією друку принтери поділяють на *матричні*, *струминні*, *лазерні* й *сублімаційні*, а за кольором друку – *кольорові* й *монохромні*.

Монохромні принтери мають кілька градацій, зазвичай 2 – 5, наприклад: чорний – білий, одноколірний (або червоний, або синій, або зелений) – білий, багатоколірний (чорний, червоний, синій, зелений) – білий.

З розвитком комп'ютерних технологій монохромні принтери поступаються повноколірним або скорочено колірним, які друкують «весь спектр кольорів», видимий людським оком.

Інші принтери (наприклад, матричні) використовуються як спеціалізовані для друку на безперервний рулон паперу в лабораторіях, банках, бухгалтеріях, для друку на багатошарові бланки (наприклад, паспорти, авіаквитки), а також, коли важливий сам факт друку ударом. Вважається, що факт удару утрудняє внесення несанкціонованих змін до фінансового документа.

Набули поширення багатофункціональні пристрої, в яких об'єднані принтер, сканер, ксерокс і факс. Таке об'єднання раціональне технічно і зручне в роботі. Широкоформатні (А3, А2) принтери іноді неправильно називають плотерами.

Сучасні технічні вимоги для принтерів початкового рівня. Цифровий фотоапарат початкового рівня з матрицею 5,25 мегапікселів забезпечує світність з роздільною здатністю 2560×1920. Якщо друкувати фото розміром 10×15 см, то принтер повинен мати роздільну здатність не менше 2560 : 10 см · 2,54 см = 650 dpi (dots per inch – точок на дюйм). Стосовно градацій чорного і кольорів, то – матриці nf просвітлені об'єктиви цифрових фотоапаратів мають фотографічну широту (динамічний діапазон), «близьку» до можли-

востей людського ока. Людина неозброєним оком може оцінити поліпшення зображення фотографії з роздільною здатністю до 750 dpi і фотографічною широтою до 24 біт за 3 базовими кольорами, True Color, і 8 біт сірого, тобто фотографії, що містить до 16777216 кольорів з плавними, реальними півтонами і до 256 градацій чорного. До цих характеристик «близькі» результати друку на аналоговий кольоровий фотопапір, друк сублімації «схожий» на якісний, струминний і лазерний друк – лише наближаються до цих можливостей.

Способи з'єднання принтера з носієм цифрової інформації. З'єднання принтера з комп'ютером через паралельний порт або послідовний порт зі швидкістю до 50 кбайт/с не задовольняє сучасні вимоги на відміну від з'єднання принтера з комп'ютером через USB-інтерфейс. Швидкість передавання даних через USB-2.0 – до 480 Мбіт/с, а також легше відбувається процес підключення, принтер може значно більше повідомляти про свій стан, готовність, наявність паперу тощо.

Поширення набули бездротові підключення принтерів: інфрачервоний порт (ІЧ-порт), Bluetooth, Wi-Fi.

Інфрачервоний порт забезпечує друк з комп'ютерів, кишенькових персональних комп'ютерів, телефонів та інших пристроїв, оснащених ІЧ-портом,і мають можливість друку. Зона застосування обмежена декількома сантиметрами прямої видимості між портами пристроїв. Натепер ІЧ-порт поступився радіоінтерфейсам Bluetooth і Wi-Fi, які дозволяють встановити принтер у будь-якому зручному місці на відстані до 10-100 м.

Сучасні принтери читають флеш-пам'ять, мають відеоекран і можуть друкувати фотографії без комп'ютера. Принтери, які мають мережевий інтерфейс, підключаються до локальної мережі, що дозволяє користуватися принтером автономно з декількох комп'ютерів.

Технічний аналіз сучасних технологій цифрового друку. За поширеністю першим лідирує струминний друк, другим – лазерний, третім – термосублімаційний, четвертим – матричний. У разі струминного, лазерного і матричного способів друку лінеатура становить 300-80-30 lpi і залежить від роздільної здатності пристрою. При друці сублімації лінеатура отримуваних півтонів перевищує 300 lpi, тому наймасовішого застосування монохромні лазерна і

матрична технології набули в друці текстів і графіки, а повнокольорова термосублімаційна технологія використовується у фотопринтерах. Кольоровий струминний друк показує задовільні результати при друці текстів, графіки і фотографій.

За кольороутворенням до повнокольорових (від англ. *continuous tone* – безперервний тон кольору) належить тільки термосублімаційна технологія. Струминна, лазерна і матрична технології – растрові (від англ. *bi-level* – два рівні), тобто для отримання однієї повнокольорової точки растра (2-й рівень) потрібен мікрорастр – по 16 × 16 = 256 «службових» мікропікселів кожного кольору (1-й рівень). Головний конструктивний недолік лазерних технологій – труднощі досягнення допуску понад 1200 dpi, точок на дюйм. Межа для лазерного друку кожного кольору за растрування 2400 dpi /16 = 150 lpi, що на порядок менше за характеристики аналогового кольорового фотопаперу.

Нові модифікації лазерних, струминних і термосублімаційних технологій друку дозволяють досягати задовільних результатів і належать до комбінованих (від англ. *contone* – півтоновий колір). *Contone = bi-level + continuous tone*. Таке півтонове зображення місцями друкується точками, а місцями безперервною заливкою, барвником. Струминна і лазерна технології друкують точки з «різкими» межами, без перекриття, що добре за високого допуску, а якщо допуск менший від 4800 dpi, то на кінцевому зображенні видно растр, в аналоговій фотографії це називають зернистістю зображення. На аналоговому кольоровому фотопапері зображення створюється теж точками (зерном) з «різкими» межами, але роздільна здатність фотопаперу висока і зображення виходить дрібнозернистим і відмінної якості. За термосублімаційної технології сусідні пікселі частково перекриваються. Це, на жаль, знижує допуск до 300 lpi (300 lpi для растра – 300 × 16 = 4800 dpi), але створює ефект безперервності зображення, як на аналоговому кольоровому фотопапері. Візуально фото, видруковане на термосублімаційному принтері, виглядає відмінно.

Перевагою лазерного друку є постійна готовність до роботи. Порошок тонера лазерного принтера не сохне, вали не засмічуються. Правда, на простих моделях принтерів тонер і папір дряпають світлочутливий шар на барабані, що обмежує термін експлуатації барабана 4 – 5 заправленнями картриджа. Ресурс фотобарабана ро-

зрахований на 10.000 – 15.000 сторінок. Ресурс картриджа розрахований на 2.000 – 5.000 сторінок за п'ятивідсоткового заповнення.

Світлодіодні принтери. Принцип дії світлодіодних принтерів багато в чому подібний до принципу роботи лазерних. Робота принтера ґрунтується на принципі сухого електростатичного перенесення.

Принципова відмінність світлодіодного від лазерного принтера полягає в механізмі освітлення світлочутливого вала. За лазерною технологією це виконується одним джерелом світла (лазером), який за допомогою сканувальної системи призм та дзеркал пробігає всією поверхнею вала. У світлодіодних же принтерах замість одного лазера використовується лінійка світлодіодів, розташована вздовж усієї поверхні вала. Кількість світлодіодів у лінійці становить від 2,5 до 10 тисяч штук.

Струминні принтери. Принцип дії струминних принтерів схожий на принцип дії матричних принтерів тим, що зображення на носієві формується з точок. Але замість головок з голками в струминних принтерах використовується матриця, що друкує рідкими барвниками. Картриджі з барвниками бувають із вбудованою друкувальною головкою – в основному такий підхід використовується компаніями Hewlett-Packard, Lexmark. Фірми Epson, Canon виробляють струминні принтери, у яких друкувальна матриця є деталлю принтера, а змінні картриджі містять тільки барвник. У разі тривалого простою принтера (тиждень і більше) залишки барвника на соплах друкувальної головки висихають. Принтер уміє сам автоматично чистити друкувальну головку. Можна також примусово очищувати сопла із відповідного розділу налаштувань драйвера принтера. Під час прочищання сопел друкувальної головки відбувається інтенсивна витрата барвника. Особливо критичне засмічення сопел друкувальної матриці принтерів Epson і Canon. Якщо штатними засобами принтера не вдається очистити сопла друкувальної головки, то подальше очищення і/або заміна друкувальної головки проводиться в ремонтних майстернях. Заміна картриджа, що містить друкувальну матрицю, на новий труднощів не викликає. Друкувальні головки струминних принтерів використовують різні типи подачі барвника.

Безперервна подача (*Continuous Ink Jet*) – барвник під час друку подається безперервно, факт потрапляння барвника на запечату-

вану поверхню визначається модулятором потоку барвника. Стверджується, що патент на цей спосіб друку виданий Вільяму Томпсону в 1867 р. Технічна реалізація такої друкувальної головки: у сопло під тиском подається барвник, який на виході із сопла розбивається на послідовність мікрокрапель (об'ємом декілька десятків піколітрів), яким додатково повідомляється електричний заряд. Потік барвника розбивається на краплі розміщеним на соплі п'єзокристалом, на якому формується акустична хвиля (частотою десятки кілогерців). Відхиляється потік крапель за допомогою електростатичної відхильної системи (дефлектора). Ті краплі барвника, які не повинні потрапити на запечатувану поверхню, збираються у збирач барвника і, як правило, повертаються назад в основний резервуар з барвником. Перший струминний принтер виготовлений з використанням даного способу подачі барвника випустила Siemens в 1951 р.

Подача на вимогу (*Drop-on-demand*) – барвник із сопла друкувальної головки подається тільки тоді, коли барвник дійсно треба нанести на відповідну соплу ділянку запечатуваної поверхні. Саме цей спосіб подачі барвника й набув найбільшого поширення в сучасних струминних принтерах.

Натепер є дві технічні реалізації цього способу подачі барвника: п'єзоелектрична і термічна.

П'єзоелектрична реалізація (*Piezoelectric Ink Jet*). Над соплом розміщений п'єзокристал з діафрагмою. Коли на п'єзоелемент подається електричний струм, він згинається й тягне за собою діафрагму – формується крапля, яка згодом виштовхується на папір. Поширення набула в принтерах компанії Epson. Технологія дозволяє змінювати розмір краплі.

Термічна (*Thermal Ink Jet*), або *BubbleJet*. Розробник – компанія Canon. Принцип розроблений наприкінці 70-х років. У соплі розміщений мікроскопічний нагрівальний елемент, який від проходження електричного струму миттєво нагрівається до температури близько 500 °С. Під час нагрівання в чорнилі утворюються газові бульбашки (*bubbles* – звідси й назва технології), які виштовхують краплі рідини із сопла на носій. У 1981 р. технологія була представлена на виставці *Canon Grand Fair*. У 1985-му з'явилася перша комерційна модель монохромного принтера *Canon BJ*-80, а в 1988 р. –перший кольоровий принтер *BJC*-440 формату *A*2, з роздільною здатністю 400 dpi.

Сублімаційні принтери. *Термосублімація* (*сублімація*) – це швидке нагрівання барвника, коли пропускається рідка фаза. З твердого барвника відразу утворюється пара. Чим менша порція, тим більша фотографічна широта (динамічний діапазон) перенесення кольорів. Пігмент кожного з основних кольорів, а їх може бути три або чотири, міститься на окремій (або на загальній багатошаровій) тонкій лавсановій стрічці (термосублімаційні принтери фірми *Mitsubishi Electric*). Остаточний колір друкується за декілька проходів: кожна стрічка послідовно протягується під щільно притиснутою термоголовкою, що складається з безлічі термоелементів. Ці останні, нагріваючись, переганяють барвник. Точки, завдяки малій відстані між головкою та носієм, стабільно позиціонуються й виходять дуже малого розміру. Як одну з проблем друку сублімації можна відзначити чутливість вживаного чорнила до ультрафіолету. Якщо зображення не покрити спеціальним шаром, який блокує ультрафіолет, то фарби незабаром поблякнуть. Якщо застосовують тверді барвники і додатковий ламінуючий шар з ультрафіолетовим фільтром для зберігання зображення, отримувані відбитки не жолобляться і добре переносять вологість, сонячне світло і навіть агресивні середовища, але зростає ціна фотографій. Найвідоміші виробники термосублімаційних принтерів – фірми *Mitsubishi*, *Sony* і *Toshiba.* Фірми-виробники відзначають, що фотографічна широта кольору 24 біт, ще більш бажана, ніж дійсна. Реально фотографічна широта кольору становить не більше 17 біт.

Матричні принтери. *Матричні принтери* – найдавніші з нині вживаних типів принтерів; його механізм був винайдений в 1964 р. корпорацією Seiko Epson. Матричні принтери стали першими пристроями, що забезпечили графічне виведення твердої копії. Зображення формується друкувальною головкою, яка складається з набору голок (голкова матриця), що приводяться в дію електромагнітами. Головка пересувається порядково вздовж аркуша, при цьому голки вдаряють по паперу через фарбувальну стрічку, формуючи точкове зображення. Цей тип принтерів називається SIDM (Serial Impact Dot Matrix – послідовні ударно-матричні принтери). Випускалися принтери з 9, 12, 14, 18 і 24 голками в головці. Основного поширення набули 9- і 24-голкові прин-

тери. Якість друку і швидкість графічного друку залежить від кількості голок: більше голок – більше точок.

Принтери з 24 голками називають LQ (Letter Quality – якість друкувальної машинки). Існують монохромні та п'ятиколірні матричні принтери, в яких використовується 4-колірна стрічка CMYK. Зміна кольору відбувається зсувом стрічки вгору-вниз відносно друкувальної головки. Швидкість друку матричних принтерів вимірюється у символах за секунду (CPS – Characters Per Second).

Типовий результат роботи матричного принтера в режимі *draft* подано на рис. 1.25. Цей рисунок показує фрагмент друку розміром 4,5×1,5 см.

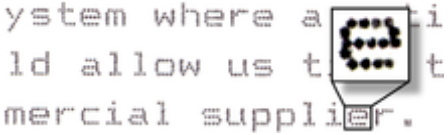


Рис. 1.25. Результат роботи матричного принтера в режимі *draft*

Основними недоліками матричних принтерів є монохромність, низька швидкість роботи і високий рівень шуму. Матричні принтери застосовують дотепер завдяки дешевизні копії (витратним матеріалом, по суті, є тільки фарбувальна стрічка) і можливості роботи з безперервним (рулонним, фальцованим) і копіювальним папером. Випускаються і швидкісні лінійно-матричні принтери, в яких велика кількість голок, рівномірно розміщених на човниковому механізмі (фреті) по всій ширині листа. Швидкість таких принтерів вимірюється в рядках за секунду (LPS –Lines Per Second).

Барабанні принтери (*drum printer*). Перший принтер, що отримав назву UNIPRINTER, був створений в 1953 р. компанією Remington Rand для комп'ютера UNIVAC. За принципом дії він нагадував друкарську машинку. Основним елементом такого принтера був обертовий барабан, на поверхні якого нанесені рельєфні зображення літер і цифр. Ширина барабана відповідала ширині паперу, а кількість кілець з алфавітом дорівнювала максимальній кількості символів у рядку. За папером розміщувалася лінійка молоточків, що приводяться в дію електромагнітами. У момент проходження потрібного сим-

вола на обертовому барабані молоточок ударяв по паперу, притискуючи його через фарбувальну стрічку до барабана. Таким чином, за один оберт барабана можна було надрукувати весь рядок. Далі папір зміщався на один рядок і машина друкувала далі. У СРСР такі машини називалися алфавітно-цифровим друкувальним пристроєм (АЦДП). Їх роздруківки можна впізнати за шрифтом, схожим на шрифт друкарської машинки і літерами, що «стрибають» у рядку.

Ромашкові (*пелюсткові*) *принтери* (*daisywheel printer*). За принципом дії вони були схожі на барабанні, однак мали один набір літер, розміщених на гнучких пелюстках пластмасового диска. Диск обертався, і спеціальний електромагніт притискував потрібну пелюстку до фарбувальної стрічки та паперу. Оскільки набір символів був один, то треба було, щоб друкувальна головка переміщувалася уздовж рядка і швидкість друку була нижчою, ніж у випадку барабанних принтерів. Замінивши диск із символами, можна отримати інший шрифт, а вставивши стрічку не чорного кольору – «кольоровий» відбиток.

Гусеничні принтери (*train printer*) – набір букв закріплений на гусеничному ланцюжку.

Ланцюжкові друкувальні пристрої (*chain printer*) характеризувалися розміщенням друкувальних елементів на з'єднаних у ланцюг пластинах.

Термічні принтери фірми Xerox. Характеризуються витратним матеріалом – речовиною на основі парафіну, що плавиться за температури 60°С.

Використання принтерів не за призначенням. Останнім часом дедалі частіше принтери стали використовуватися не тільки для друку на папері. Наприклад, радіоаматори застосовують лазерні принтери в «лазерно-прасувальній» технології виготовлення плат, наносячи витравлювальну маску використовуючи лазерний принтер. Перспективна технологія друку електронних схем за допомогою принтера, коли в картрідж замість чорнила заливають спеціальні хімічні речовини.

1.8.3. Плотери

Графопобудівник – пристрій, призначений для виведення даних у графічній формі на папір.

Плотер – широкоформатний, найчастіше струминний принтер, зорієнтований на друк аркушів формату А0, А1, А2, А3, А4 і т.д. різної товщини (від 80 г/м^2, ватманів, півватманів тощо). Використовується для друку як у чорно-білому, так і в кольоровому варіантах, креслень, схем, карт та ін.

Особливості: великі місткості чорнильниць, можливість підключення системи неперервної подачі чорнил для ще більших масштабів друку, «гаряча» заміна чорнильниць та картриджа.

Плотер може бути обладнаний головкою з лезом, або спеціальним пером, а виведення інформації здійснюється шляхом нарізки матеріалу по кривих лініях (на оракалі, плівках для термоперенесення тощо). Дані на плотер подаються з програм, призначених для роботи з векторною графікою (Corel Draw, Adobe Illustrator тощо).

Плотери автоматично виготовляють на аркуші паперу точну тверду копію обрахованого на комп'ютері креслення. Переважно в системах автоматизованого проектування (САПР) використовуються такі типи плотерів:

– рулонний,
– планшетний,
– електростатичний,
– струминний.

Рулонні та планшетні плотери є векторними пристроями, які будуться з окремих геометричних елементів або тушшю, або кульковим пером. Такі плотери переважно мають декілька пер, які рисують лінії різних товщин і кольору. Пера вибираються автоматично відповідно до атрибутів «шару», установлених в процесі створення креслення в САПР.

1.8.4. Пристрій візуального відображення

Усі комп'ютерні креслення відображаються на пристроях візуального відображення (ПВВ), або алфавітно-цифрових дисплеях (АЦД). На АЦД можуть виводитися деякі текстові додатки. Алфавітно-цифрові дисплеї переважно застосовуються для введення ко-

манд у комп'ютер, виведення службових повідомлень та роботи з меню функцій креслення.

Більшість ПВВ систем САПР відображають графічну інформацію на пристрій, називаний електронною струминною трубкою (ЕСТ). Вона являє собою вакуумну скляну трубку, в якій є електронна пушка, яка випускає пучок електронів в екран з люмінофорним покриттям, у результаті чого на екрані з'являється світний слід.

За останні 20 років було розроблено декілька типів ЕСТ. Найбільш відомі з них такі:

– запам'ятовувальна трубка з безпосереднім відтворенням інформації;

– ЕСТ (дисплей) з векторною регенерацією;

– ЕСТ (растровий дисплей) з растровою регенерацією.

Кольорова ЕСТ. Особливістю кольорового растрового дисплею є те, що кольорова ЕСТ має три електронні пушки, тоді як у монохромній ЕСТ – тільки одна. Кожна з трьох пушок відповідає відтінку (червоному, синьому і зеленому). Екран кольорової ЕСТ складається з тисяч люмінофорних плям, згрупованих по три (кожній з них відповідає основний колір). Коли промінь від певної електронної пушки потрапляє на кожну таку пляму, вона світить відповідним йому відтінком і тим самим створює кольоровий піксел. Три промені розділяються дуже малими отворами в металевій пластині, яка називається тіньовою маскою, розміщеною на зворотному боці екрану.

Процесор робочої станції. У ранніх розробках систем САПР наголос робився на центральний (головний) комп'ютер, який надавав величезну пам'ять і забезпечував можливості графічним командам. Робочі ж станції тоді фактично були просто «мовчазними» (неінтелектуальними) терміналами з невеликими (якщо вони були) засобами процесування. З появою растрових ЕСТ і завдяки безперервному розвитку графічних засобів почали використовуватися «інтелектуальні» робочі станції з власними локальними процесорами.

Процесор робочої станції (інколи його називають графічним процесором) являє собою комп'ютер (переважно 16-бітовий з ємністю пам'яті 256–768 кбайт) усередині кожної робочої станції, який допомагає головному комп'ютеру підвищити швидкість формування графічних зображень, використовуючи пам'ять головного комп'ютера.

Такі складні засоби, як тривимірне моделювання суттєво потребують пам'яті головного комп'ютера. Ці засоби стають дедалі більш популярними, що сприяє підвищенню інтелекту робочих станцій.

Меню САПР являє собою список доступних графічних команд, наприклад *LIST* (список), *CIRCLE* (коло), *ARC* (дуга), *ZOOM* (наїзд) і т.ін.

Один з типів меню, яке видається на екран дисплею, показано на рис. 1.26. У разі використання такого меню графічні елементи вибираються за допомогою набору відповідних команд на алфавітно-цифровій клавіатурі (або інколи натисканням спеціальних функціональних клавіш, розміщених у функціональному блоці).

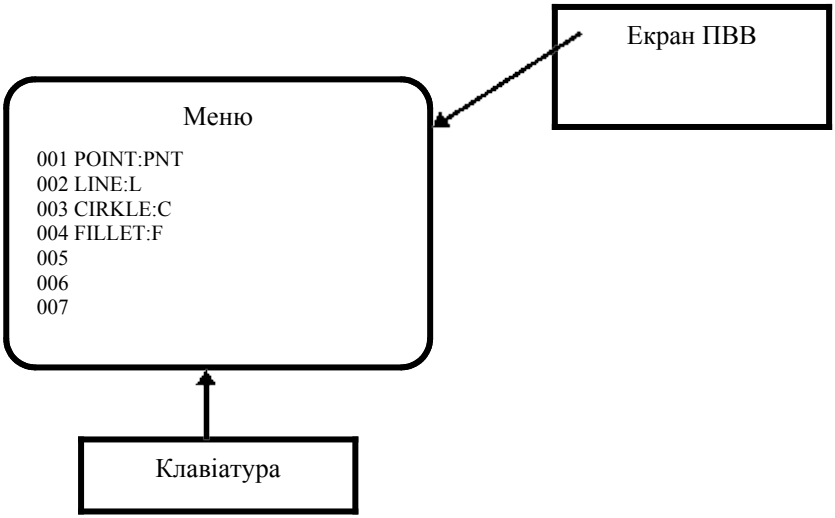


Рис 1.26. Меню

1.8.5. Електронний командний планшет

Більшість електронних командних планшетів (інколи їх називають бітовою пам'яттю) мають двоцільове призначення, оскільки об'єднують в собі меню і оцифрування.

Більш потужні системи САПР дозволяють розширити вибір за рахунок кількох бланків, які мають різні набори команд. У про-

цесі роботи ці бланки можна почергово накладати на меню планшета. Майже всі великі системи також надають користувачу засоби самому підготовлювати бланки меню для спеціалізованих команд. Графічні команди вибирають дотиканням відповідного місця на бланку електронним пером або візиркою.

Якщо перо або візирка потрапляють на ділянку оцифрування планшета, то вони стають пристроями керування курсором. Сам курсор являє собою дві прямі, що перетинаються (перехрещення), які з'являються на екрані після вибору графічної команди. За його допомогою можна ефективно розміщувати в потрібному місці екрана графічні елементи. Ці елементи (наприклад, крапка, пряма, коло і т.ін.) в результаті виконання відповідної команди будуть зображатися на екрані так, що перехрещення курсора розміститься на останньому побудованому елементі. Керуючи курсором, можна вказати на екрані відповідні елементи для їх наступної модифікації.

Рух курсора по екрану відповідає переміщенню електронного пера або візирки, якими керує людина.

1.8.6. Сканери

Сканер – це пристрій, який дає змогу вводити в комп'ютер чорно-біле або кольорове зображення, зчитувати графічну і текстову інформацію. Сканер використовують у випадку, коли виникає потреба ввести в комп'ютер з оригіналу текст і/або графічне зображення для його наступного оброблення. Уведення такої інформації за допомогою стандартних пристроїв потребує багато часу і зусиль. Сканована інформація потім обробляється за допомогою спеціального ПЗ (наприклад, програмою *FineReader*) і зберігається у вигляді текстового або графічного файлу.

Основним елементом сканера є *CCD*-матриця (*Charge Coupled Device* – пристрій із зарядовим зв'язком) або *PMT* (*Photo Multiplier Tube* – фотомножник). Колби-фотомножники використовуються лише у складних і дорогих барабанних професійних сканерах, тому доцільніше розглядати принцип дії сканерів із *CCD*-матрицею. *CCD*-матриця – це набір діодів, що реагують на світло при дії зовнішньої напруги. Від якості матриці залежить якість розпізнавання зображення.

Дешеві моделі розпізнають наявність/відсутність кольору, складні моделі – відтінки сірого кольору, ще складніші – всі кольори. Аркуш, що сканується, освітлюється ксеноновою лампою або набором світлодіодів. Відбитий промінь за допомогою системи дзеркал або лінз проеціюється на *CCD*-матрицю. Під дією світла та зовнішньої напруги матриця генерує аналоговий сигнал, що змінюється у ході переміщення відносно неї аркуша та інтенсивності відображення різних елементарних фрагментів. Сигнал подається на аналогово-цифровий перетворювач, де він оцифровується (надається у вигляді набору нулів та одиниць) і передається у пам'ять комп'ютера. Є два способи сканування: переміщення аркуша відносно нерухомої *CCD*-матриці або переміщення світлочутливого елемента відносно нерухомого аркуша.

Існує чимало моделей сканерів, що різняться методом сканування, допустимим розміром оригіналу та якістю оптичної системи. За способом організації переміщення зчитувального вузла відносно оригіналу сканери поділяють на *планшетні*, *барабанні* та *ручні*.

У *планшетних сканерах* оригінал кладуть на скло, під яким рухається оптико-електронний зчитувальний пристрій.

У *барабанних сканерах* оригінал через вхідну щілину втягується барабаном у транспортний тракт і пропускається через нерухомий зчитувальний пристрій. Барабанні сканери не дають змоги сканувати книги, переплетені брошури тощо.

Ручний сканер необхідно плавно переміщувати вручну по поверхні оригіналу, що не дуже зручно. Для систематичного користування краще мати, хоча і дорожчий, настільний планшетний сканер.

Основні технічні характеристики сканерів. *Роздільна здатність.* Сканер розглядає будь-який об'єкт як набір окремих точок (пікселів). Щільність пікселів (кількість на одиницю площі) називається *роздільною здатністю сканера* і вимірюється у dpi. Пікселі розташовуються рядками, утворюючи зображення. Процес сканування відбувається по рядках, весь рядок сканується одночасно. Звичайна роздільна здатність сканера становить 200–720 dpi. Більше значення (понад 1000 dpi) відображає інтерполяційну роздільну здатність, досягнуту програмним способом із використанням математичного оброблення параметрів розміщених поруч точок зображення. Якість відсканованого матеріалу залежить також від оптичної роздільної здатності (визначається кількістю світлочутливих діодів

CCD-матриці на дюйм) та механічної роздільної здатності (визначається дискретністю руху світлочутливого елемента або системи дзеркал відносно аркуша). Роздільну здатність визначають залежно від застосування результатів сканування: для художніх зображень, які потрібно друкувати на фотонабірних машинах, – 1000–1200 dpi, для друкування зображення на лазерному або струминному принтері – 300–600 dpi, для перегляду зображення на екрані монітора – 100–200 dpi, для розпізнавання тексту – 200–400 dpi.

Глибина кольорів. У разі перетворення оригіналу у цифрову форму зберігаються дані про кожний піксел зображення. Прості сканери визначають наявність або відсутність кольору, результуюче зображення буде чорно-білим. Для пікселів достатньо одного розряду (0 або 1). Для передачі відтінків сірого між чорним та білим кольором необхідно як мінімум 4 розряди (16 відтінків) і 8 розрядів (256 відтінків). Чим більше розрядів, тим якісніше передаються кольори. Більшість сучасних кольорових сканерів підтримує глибину кольору 24 розряди. Відповідно сканер дозволяє розпізнавати близько 16 млн кольорів і якісно сканувати фотографії. На ринку сканерів є моделі, що мають глибину кольору 30 та 34 розряди.

Динамічний діапазон. За діапазоном оптичної щільності визначається спектр півтонів. Оптична щільність визначається як відношення падаючого світла до відображеного і коливається від 0,0 (абсолютно біле тіло) до 4,0 (абсолютно чорне тіло). Значення діапазону доповнюється літерою *D* і визначає ступінь його чутливості. Більшість планшетних сканерів мають стандартний діапазон 2,4 *D*, важко розрізняють близькі відтінки одного кольору, але цього достатньо для непрофесійного користувача.

Метод сканування. Якість сканованого кольорового зображення залежить від методу нагромадження даних сканером. Розрізняють два основні методи, що відрізняються кількістю проходів *CCD*-матриці над оригіналом. Перші сканери використовували трипрохідне сканування. Із кожним проходом сканувався один із кольорів палітри *RGB*. Сучасні сканери використовують однопрохідну методику, яка розділяє світловий промінь на складові за допомогою призми.

Ділянка сканування. Це максимальний розмір сканованого зображення (для ручних сканерів до 105 мм, для барабанних, планшетних – від А4 до *Full Legar* (8.5′×14′)).

Швидкість сканування. Немає стандартної методики, що визначає продуктивність сканера. Виробники вказують кількість мілісекунд сканування одного рядка. Але потрібно враховувати також спосіб підімкнення сканера до комп'ютера, драйвер, схему передавання кольорів, роздільну здатність. Тому швидкість сканування визначається експериментальним шляхом.

1.8.7. Модеми

Модем – це пристрій, призначений для підімкнення комп'ютера до звичайної телефонної лінії. Назва походить від скорочення двох слів – модуляція та демодуляція.

Комп'ютер виробляє дискретні електричні сигнали (послідовності двійкових нулів та одиниць), а по телефонних лініях інформація передається в аналоговій формі (тобто у вигляді сигналу, рівень якого змінюється безперервно, а не дискретно). Модеми виконують цифро-аналогове й обернене перетворення. У процесі передавання даних модеми накладають цифрові сигнали комп'ютера на безперервну несучу частоту телефонної лінії (модулюють її), а під час їх приймання демодулюють інформацію і передають її в цифровій формі в комп'ютер. Модеми передають дані по звичайних, тобто комутованих, телефонних каналах зі швидкістю від 300 до 56 000 біт/с, а по орендованих (виділених) каналах ця швидкість може бути і вищою. Окрім того, сучасні модеми стискають дані перед відправленням, і відповідно реальна швидкість може перевищувати максимальну швидкість модема.

За конструктивним виконанням модеми бувають *убудованими* (вставляються в системний блок комп'ютера в один із слотів розширення) і *зовнішніми* (підключаються через один із комунікаційних портів, маючи окремий корпус і власний блок живлення). Однак без відповідного комунікаційного ПЗ, найважливішою складовою якого є протокол, модеми не можуть працювати. Найбільш поширеними протоколами модемів є *v.*32 *bis*, *v.*34, *v.*42 *bis* та інші.

Сучасні модеми для широкого кола користувачів мають вбудовані засоби відправлення і отримання факсимільних повідомлень. Такі пристрої називаються факс-модемами. Вони можуть підтримувати мовні функції за допомогою звукового адаптера.

Запитання для самоперевірки

1. Наведіть визначення терміна *операційна система*.
2. Які бувають ОС?
3. Яке призначення систем пакетної обробки?
4. Яке призначення систем розділення часу?
5. Яке призначення систем реального часу?
6. Наведіть визначення терміна *однозадачна операційна система*.
7. Наведіть визначення терміна *багатозадачна операційна система*.
8. Наведіть визначення терміна *операційне середовище.*
9. Наведіть визначення терміна *ресурс.*
10. Наведіть визначення терміна *кластер*.
11. Наведіть визначення терміна *архітектура операційної системи*.
12. Які бувають архітектури ОС?
13. Наведіть визначення терміна *структура операційної системи*.
14. Які бувають структури ОС?
15. Що таке системні оброблювальні програми?
16. Що таке програми надання користувачу додаткових послуг?
17. Що таке бібліотеки процедур різного призначення?
18. Що таке утиліти?
19. Назвіть типові засоби апаратної підтримки ОС та їх призначення.
20. Наведіть визначення терміна *файлова система*.
21. Які бувають файлові системи?
22. Що таке користувацький інтерфейс?
23. Назвіть види користувацьких інтерфейсів.
24. Наведіть визначення терміна *периферійні* або *зовнішні пристрої*.
25. Які бувають периферійні або зовнішні пристрої?

2. МЕХАНІЗМИ ОПЕРАЦІЙНИХ СИСТЕМ

2.1. Поняття *процес* та *потік*. Створення, планування та диспетчеризація процесів та потоків

2.1.1. Поняття *процес* і *потік*

Під *процесом* розуміють програму в стадії виконання. Процес можна розглядати також як одиницю роботи для процесора. Для сучасних типів процесорів використовують також дрібну одиницю роботи – потік або нитку. Інакше кажучи процес може породити один і більше потоків.

Принципова відмінність між поняттями *процес* і *потік* полягає в тому, що *процес* ОС розглядає як заявку на всі види ресурсів (пам'ять, файли тощо), крім одного, – процесорного часу, а *потік* – як заявку на процесорний час.

Надалі як одиницю роботи ОС використовуватимуть поняття *процес* і *потік*. У випадках, коли це не відіграє суттєвої ролі, вони будуть називатися завданнями.

2.1.2. Планування процесів та потоків

Планування процесів та потоків включає:

– створення–знищення процесів;

– взаємодію між процесами;

– розподіл процесорного часу;

– забезпечення процесів необхідними ресурсами (одноосібно, спільно);

– синхронізацію (контроль за виникненням «перегонів», блокувань);

– після завершення процесу – «зачищення», тобто видалення слідів перебування в системі.

Кожен процес ізолюється від інших своїм віртуальним адресним простором, що є сукупністю адрес, якими може маніпулювати програмний модуль процесу. Операційна система відображає віртуальний адресний простір на відведену для процесу фізичну пам'ять. Для взаємодії процеси звертаються до ОС, що надає засо-

би спілкування (конвеєри, поштові скриньки, колективні секції пам'яті та ін.).

Можливість розпаралелювання обчислень у межах процесу на потоки підвищує ефективність ОС. Механізм розпаралелювання обчислень для однієї програми називається *багатопотоковим обробленням* (*multithreading*). Потоки процесу мають один адресний віртуальний простір. Розпаралелювання пришвидшує виконання процесу, оскільки ОС не перемикається з одного адресного простору на інший, як це відбувається під час виконання процесів. Програми стають більш логічними. Особливий ефект при цьому досягається в мультипроцесорних системах. Прикладом багатопотокового оброблення є виконання запитів MS SQL Server.

2.1.3. Створення процесів

Створити процес – це створити описувач процесу (інформаційну структуру, що містить відомості, необхідні для керування цим процесом).

Зміст описувача: ідентифікатор, адреса виконуваного модуля, пріоритет, права доступу та ін.

Приклади описувачів для:

– Windows NT/2000/XP – об'єкт-процес (*object-process*);

– UNIX – дескриптор процесу;

– OS/2 – керуючий блок процесу (*PCB – Process Control Block*).

Крім того, створення процесу означає виконання таких дій:

– знайти програму на диску;

– перерозподілити оперативну пам'ять;

– виділити пам'ять новому процесу;

– переписати програму у виділену пам'ять;

– змінити деякі параметри програми.

У деяких системах дані можуть відразу не вміститися в пам'ять, а переписуються в спеціальну ділянку диска – *ділянку підкачування*.

2.1.4. Створення потоків

У багатопотоковій системі під час створення процесу виникає хоча б один потік. Для потоку ОС генерує описувач потоку

(ідентифікатор потоку, дані про права, пріоритет, стан потоку тощо). Вихідний стан потоку – припинений.

Потік може породити інший потік – нащадок. По завершенні батьківського потоку використовуються різні алгоритми. Асинхронне завершення передбачає продовження виконання потоків-нащадків після завершення батьківського потоку. Синхронне завершення батьківського потоку призводить до завершення всіх його нащадків.

Приклад створення потоків у *Windows* (*object Pascal*):

T = TThread; Create (false).

Приклад видалення потоку:

T. Suspend; T. Terminate; T. Free.

2.1.5. Планування і диспетчеризація потоків

Протягом існування процесу виконання його потоків може бути багаторазово перериватися та продовжуватися. Перехід від виконання одного потоку до іншого здійснюється в результаті планування та диспетчеризації.

Планування означає визначення моменту часу: коли перервати виконання активного потоку і який потік активізувати. Планування виконується на основі описувачів потоків. Планування потоків, передбачає вирішення двох завдань:

– визначення моменту часу для зміни поточного активного потоку;

– вибір для виконання потоку з черги готових потоків.

Застосовуються різні алгоритми планування.

У більшості ОС планування *динамічне*, тобто рішення приймаються під час функціонування ОС на основі аналізу поточної ситуації.

Інший тип планування *статичний*, в якому весь набір одночасно виконуваних завдань визначають заздалегідь (*off-line*), наприклад, за розкладом. Такий тип планування застосовується в спеціалізованих ОС, у системах реального часу.

Диспетчеризація – це реалізація результатів планування, зокрема:

– перемикання потоків;

– збереження контексту поточного потоку;

– завантаження контексту нового потоку;

– запуск нового потоку.

Оскільки операція перемикання контекстів істотно впливає на продуктивність обчислювальної системи, програмні модулі ОС виконують диспетчеризацію потоків разом з апаратними засобами процесора.

У контексті потоку можна виділити частину, загальну для всіх потоків даного процесу (посилання на відкриті файли), і частину, яка стосується тільки цього потоку (вміст регістрів, лічильник команд, режим процесора). Наприклад, у середовищі NetWare 4.x розрізняються три види контекстів: глобальний контекст (контекст процесу), контекст групи потоків і контекст окремого потоку. Співвідношення між даними цих контекстів подібне до співвідношення глобальних і локальних змінних у програмі, написаній мовою С. Змінні глобального контексту доступні для всіх потоків, створених в межах одного процесу. Змінні локального контексту доступні тільки для кодів певного потоку аналогічно локальним змінним функції. У NetWare можна створювати кілька груп потоків усередині одного процесу й ці групи будуть мати свій груповий контекст. Змінні, що належать до групового контексту, доступні для всіх потоків, що входять до групи, але недоступні для інших.

Така ієрархічна організація контекстів пришвидшує перемикання потоків, оскільки в межах однієї групи немає потреби змінювати контексти груп або глобальні контексти, достатньо лише змінити контексти потоків, які менші. Аналогічно при перемиканні з потоку однієї групи на потік іншої групи в межах одного процесу глобальний контекст не змінюється, а змінюється лише контекст групи. Глобальні контексти перемикаються лише під час переходу з потоку одного процесу на потік іншого процесу.

2.1.6. Стани потоку

Потік може перебувати в одному з трьох основних станів:

– *виконання* – активний стан потоку, під час якого потік має всі необхідні ресурси і безпосередньо виконується процесором;

– *очікування* – пасивний стан потоку, перебуваючи в якому, потік заблокований через внутрішні причини (чекає здійснення деякої події, наприклад, завершення операції введення-виведення,

отримання повідомлення від іншого потоку або звільнення необхідного йому ресурсу);

– *готовність* – теж пасивний стан потоку, але в цьому випадку потік заблокований у зв'язку із зовнішніми відносно нього обставинами (має всі потрібні ресурси для нього, готовий виконуватися, проте процесор зайнятий виконанням іншого потоку).

Протягом життєвого циклу кожен потік переходить з одного стану в інший відповідно до алгоритму планування потоків, прийнятим в цій ОС.

Розглянемо типовий граф стану потоку (рис. 2.1). Тільки що створений потік перебуває в стані готовності, він готовий до виконання і очікує звільнення процесора. Коли в результаті планування підсистема керування потоками приймає рішення про активізацію потоку, він переходить у стан виконання і перебуває в ньому доти, доки або він сам звільнить процесор, перейшовши в стан очікування якоїсь події, або буде примусово «витіснений» з процесора, наприклад, унаслідок вичерпання відведеного потоку кванта процесорного часу. В останньому випадку потік повертається в стан готовності. В цей же стан потік переходить зі стану очікування, після того, як очікувана подія відбудеться.

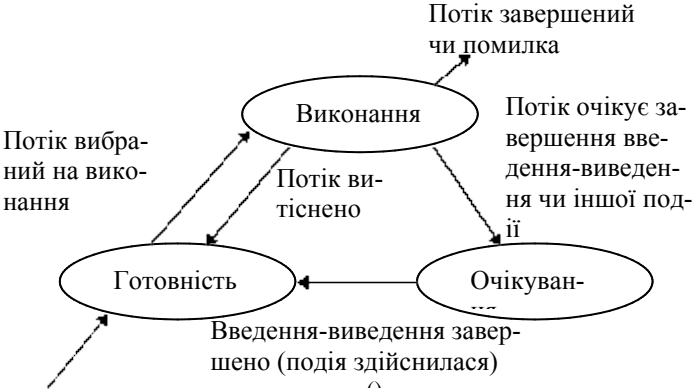


Рис. 2.1. Граф станів потоку в багатозадачному середовищі

У стані виконання в однопроцесорній системі може перебувати не більше одного потоку, а в кожному зі станів очікування і готовності – кілька потоків. Ці потоки утворюють черги відповідно до очікуючих і готових потоків. Черги потоків організуються через

об’єднання в списки описувачів окремих потоків. Таким чином, кожен описувач потоку, крім усього іншого, містить принаймні один вказівник на інший описувач, сусідній з ним у черзі. Така організація черг дозволяє легко їх перевпорядковувати, вмикати та вимикати потоки, переводити потоки з одного стану в інший. Якщо припустити, що на рис. 2.2 показано чергу готових потоків, то запланований порядок виконання виглядає так: *A*, *B*, *E*, *D*, *C*.

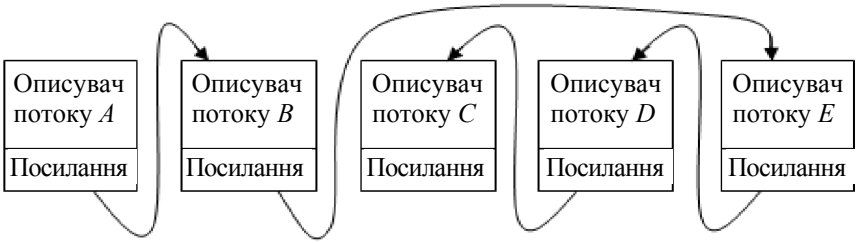


Рис. 2.2. Черга потоків

2.2. Алгоритми планування потоків та планування у системах реального часу.

2.2.1. Витісняльні і невитісняльні алгоритми планування

Із найзагальніших позицій всі алгоритми планування можна розділити на два класи: витісняльні та невитісняльні алгоритми планування.

Невитісняльні (*non-preemptive*) алгоритми – активному потоку дозволяється виконуватися, поки він сам за власною ініціативою не віддасть керування ОС, щоб та вибрала з черги інший, готовий до виконання, потік.

Витісняльні (*preemptive*) алгоритми – це такі способи планування потоків, за яких рішення про перемикання процесора з виконання одного потоку на виконання іншого потоку приймається ОС, а не активним завданням.

Основною відмінністю між витісняльними і невитісняльними алгоритмами є ступінь централізації механізму планування потоків. За витісняльного мультипрограмування функції планування потоків цілком зосереджені в ОС, і програміст створює свій додаток,

не зважаючи на те, що він виконуватиметься одночасно з іншими завданнями. При цьому ОС виконує такі функції:

– визначає момент зняття з виконання активного потоку;

– запам'ятовує його контекст;

– вибирає з черги готових потоків наступний;

– запускає новий потік на виконання, завантажуючи його контекст.

За невитісняльного мультипрограмування механізм планування розподілений між ОС і прикладними програмами. Прикладна програма, отримавши керування від ОС, сама визначає момент завершення чергового циклу виконання і тільки тоді передає керування ОС за допомогою якого-небудь системного виклику. Операційна система формує черги потоків і вибирає відповідно до деякого правила (наприклад, з урахуванням пріоритетів) наступний потік для виконання. Такий механізм створює проблеми як для користувачів, так і для розробників додатків.

Для користувачів це означає, що керування системою втрачається на довільний період часу, який визначається додатком (а не користувачем). Якщо додаток витрачає дуже багато часу на виконання якої-небудь роботи, наприклад на форматування диска, користувач не може переключитися з цього завдання на інше, наприклад на текстовий редактор, тоді як форматування продовжувалося б у фоновому режимі.

Тому розробники додатків для операційного середовища з невитісняльною багатозадачністю вимушені, покладаючи на себе частину функцій планувальника, створювати додатки так, щоб вони виконували завдання невеликими частинами. Наприклад, програма форматування може відформатувати одну доріжку дискети і повернути керування системі. Після виконання інших завдань система повертає керування програмі форматування, щоб та відформатувала наступну доріжку. Подібний метод розділення часу між завданнями працює, але він істотно ускладнює розроблення програм і ставить підвищені вимоги до кваліфікації програміста. Програміст повинен забезпечити «дружнє» ставлення своєї програми до інших виконуваних одночасно з нею програм. Для цього в програмі мають бути передбачені частини передачі керування ОС. Крайнім проявом «недружності» додатка є його зависання, яке призводить до загального руйнування системи. У системах з витісняльною ба-

гатозадачністю такі ситуації, як правило, виключаються, оскільки центральний планувальний механізм має можливість зняти зависле завдання з виконання.

Проте розподіл функцій планування потоків між системою і додатками не завжди є недоліком, а за певних умов може бути і перевагою, оскільки дає змогу розробнику додатків самому проектувати алгоритм планування, що найбільше відповідає фіксованому набору завдань. Оскільки розробник сам визначає в програмі момент повернення керування, то при цьому виключаються нераціональні переривання програм у «незручні» для них моменти часу. Крім того, легко вирішуються проблеми сумісного використання даних: завдання під час кожного циклу виконання використовує їх монопольно і впевнене, що впродовж цього періоду ніхто інший не змінить дані. Істотною перевагою невитісняльного планування є вища швидкість перемикання з потоку на потік.

Примітка. *Поняття витісняльних і невитісняльних алгоритмів планування іноді ототожнюють з поняттями пріоритетних і непріоритетних дисциплін, що, ймовірно, зумовлено співзвучністю відповідних англомовних термінів preemptive і non-preemptive. Проте це неправильно, оскільки пріоритети в тому й іншому випадках можуть як використовуватися, так і не використовуватися.*

Майже у всіх сучасних ОС, орієнтованих на високопродуктивне виконання додатків (UNIX, Windows NT/2000/7, OS/2, VAX/VMS), реалізовані витісняльні алгоритми планування потоків (процесів).

Прикладом ефективного використання невитісняльного планування є файл-сервери NetWare 3.x і 4.x, у яких завдяки цьому досягається висока швидкість виконання файлових операцій. Відповідно до концепції невитісняльного планування, щоб не займати процесор дуже довго, потік у NetWare сам віддає керування планувальнику ОС, використовуючи такі системні виклики:

– *ThreadSwitch* – потік, що викликав цю функцію, вважається готовим до негайного виконання, але віддає керування для того, щоб могли виконуватися й інші потоки;

– *ThreadSwitchWithDelay* – функція аналогічна до попередньої, але потік вважає, що буде готовий до виконання тільки через певну кількість перемикань з потоку на потік;

– *Delay* – функція аналогічна до попередньої, але затримка вимірюється в мілісекундах;

– *ThreadSwitchLowPriority* – функція передавання керування, відрізняється від *ThreadSwitch* тим, що потік «просить» помістити його в чергу готових до виконання, але низькопріоритетних потоків.

Планувальник NetWare використовує декілька черг готових потоків (рис. 2.3). Тільки що створений потік потрапляє в кінець черги *RunList*, яка містить готові до виконання потоки. Після відмови від процесора потік потрапляє в ту або іншу чергу залежно від того, який із системних викликів був використаний для передачі керування. Потік надходить у кінець черги *RunList* при виклику *ThreadSwitch*, у кінець черги *DelayedWorkToDoList* при викликах *ThreadSwitchWithDelay* або *Delay*, або ж у кінець черги *LowPriority-RunList* при виклику *ThreadSwitchLowPriority*.

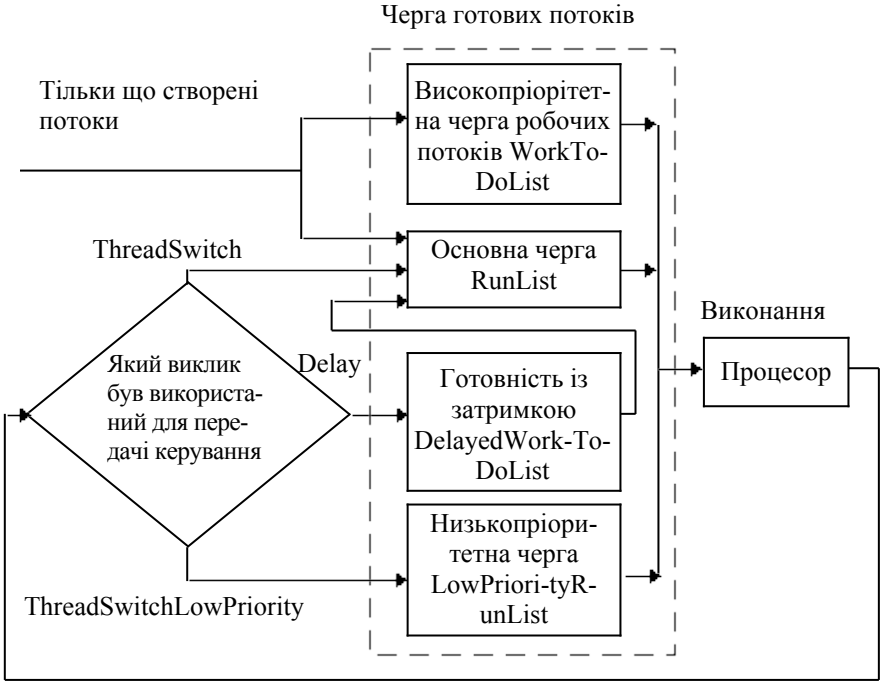


Рис. 2.3. Схема планування потоків у NetWare

Після того, як потік, що виконувався процесором, завершить свій черговий цикл виконання, віддавши керування за допомогою одного з викликів передачі керування (або виклику очікування на

семафорі), планувальник вибирає для виконання потік, що стоїть першим у черзі *RunList*, і запускає його. Потоки, що перебувають у черзі *DelayedWorkToDoList*, після завершення умови очікування переміщаються в кінець черги *RunList*.

Потоки, що перебувають у черзі *LowPriorityRunList*, запускаються на виконання тільки в тому випадку, якщо черга *RunList* порожня. Зазвичай в цю чергу призначаються потоки, що виконують нетермінову фонову роботу.

Черга *WorkToDoList* є в системі найпріоритетнішою. В цю чергу потрапляють робочі потоки. У NetWare, як і в деяких інших ОС, замість створення нового потоку для виконання певної роботи може бути використаний вже існуючий системний потік. Пул робочих потоків створюється при старті системи для системних цілей і виконання термінових робіт. Робочі потоки ОС мають найвищий пріоритет, тобто потрапляють на виконання перед потоками з черги *RunList*. Планувальник дозволяє виконатися підряд тільки певній кількості потоків з черги *WorkToDoList*, а потім запускає потік з черги *RunList*.

Описаний невитісняльний механізм організації багатопотокової роботи в ОС NetWare v3.x і NetWare 4.x потенційно дуже продуктивний, оскільки відрізняється невеликими накладними витратами ОС на диспетчеризацію потоків за рахунок простих алгоритмів планування та ієрархії контекстів. Але для досягнення високої продуктивності до розробників додатків для ОС NetWare ставляться високі вимоги, оскільки розподіл процесорного часу між різними додатками залежить зрештою від мистецтва програміста.

2.2.2. Алгоритми планування, основані на квантуванні

В основу багатьох витісняльних алгоритмів планування покладено концепцію квантування. Відповідно до цієї концепції кожному потоку по черзі для виконання надається обмежений безперервний період процесорного часу – квант. Зміна активного потоку відбувається, якщо:

– потік завершився та покинув систему;
– сталася помилка;
– потік перейшов у стан очікування;
– вичерпано квант процесорного часу, відведений потоку.

Потік, який вичерпав свій квант, переводиться в стан готовності й чекає, коли йому буде наданий новий квант процесорного часу, а на виконання, відповідно до певного правила, вибирається новий потік з черги готових. Граф станів потоку, зображений на рис. 2.4, відповідає алгоритму планування, що ґрунтується на квантуванні.

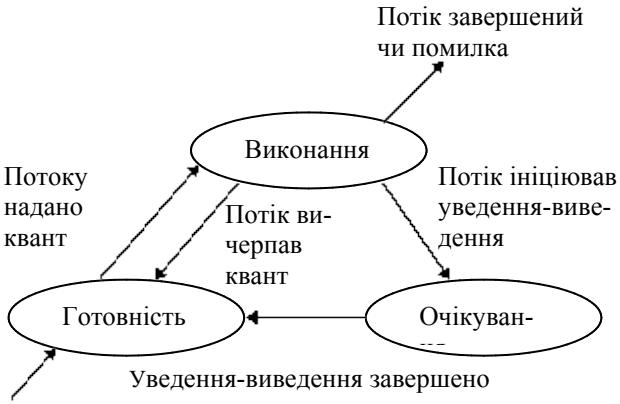


Рис. 2.4. Граф станів потоку в системі з квантуванням

Кванти, що виділяються потокам, можуть бути однаковими для всіх потоків або різними. Розглянемо, наприклад, випадок, коли всім потокам надаються кванти однакової довжини q (рис. 2.5). Якщо в системі є n потоків, то час, протягом якого потік очікує наступного кванта, можна грубо оцінити як $q(n-1)$. Чим більше потоків у системі, тим більш тривалий час очікування, тим менше можливості вести одночасну інтерактивну роботу декільком користувачам. Але якщо величину кванта вибрано не дуже великою, то значення цієї величини – $q(n-1)$ все одно буде досить малим для того, щоб користувач не відчував дискомфорту від присутності в системі інших користувачів. Типове значення кванта у системах розділення часу становить десятки мілісекунд.

Якщо квант короткий, то сумарний час, який проводить потік в очікуванні процесора, прямо пропорційний часу, потрібному для його виконання (тобто часу для виконання цього потоку в разі монопольного використання обчислювальної системи). Дійсно, оскільки час очікування між двома циклами виконання дорівнює $q(n-1)$, а

кількість циклів B/q, де B – необхідний час виконання, то $W \cdot B(n - 1)$. Проте це співвідношення є грубою оцінкою, що ґрунтується на припущенні, що B значно перевищує q. При цьому не враховується, що потоки можуть використовувати кванти не повністю, що частину часу вони можуть витрачати на введення-виведення, що кількість потоків у системі може динамічно змінюватися.

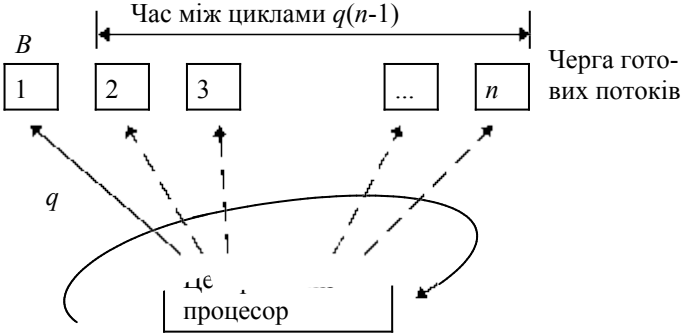


Рис. 2.5. Ілюстрація розрахунку часу очікування в черзі

Чим більший квант, тим вища ймовірність того, що потоки завершаться внаслідок першого ж циклу виконання, і тим менш помітною стає залежність часу очікування потоків від їх часу виконання. За великого кванта алгоритм квантування перероджується в алгоритм послідовного оброблення, властивий однопрограмним системам, за якого час очікування завдання в черзі взагалі ніяк не залежить від її тривалості.

Кванти, що виділяються одному потоку, можуть бути фіксованої величини, а можуть і змінюватися в різні періоди життєвого циклу потоку. Нехай, наприклад, спочатку кожному потоку призначається великий квант, а величина кожного наступного кванта зменшується до деякої наперед заданої величини. У такому разі перевагу отримують невеликі завдання, які встигають виконуватися протягом першого кванта, а тривалі обчислення виконуватимуться у фоновому режимі. Можна уявити алгоритм планування, в якому кожен наступний квант, що виділяється для певного потоку, більший від попереднього. Такий підхід дозволяє зменшити значні затрати на перемикання завдань у тому випадку, коли відразу декілька завдань виконують тривалі обчислення.

Потоки отримують для виконання квант часу, але деякі з них використовують його не повністю, наприклад через потребу виконати введення або виведення даних. У результаті виникає ситуація, коли потоки з інтенсивними зверненнями до введення-виведення використовують тільки невелику частину виділеного їм процесорного часу. Алгоритм планування може виправити цю «несправедливість». Як компенсація за невикористані повністю кванти потоки отримують привілеї для подальшого обслуговування. Для цього планувальник створює дві черги готових потоків (рис. 2.6). Черга 1 утворена потоками, які набули стану готовності в результаті вичерпання кванта часу, а черга 2 – потоками, в яких завершилася операція введення-виведення. Під час вибору потоку для виконання перш за все є видимою друга черга, і лише якщо вона порожня, квант виділяється для потоку з першої черги.

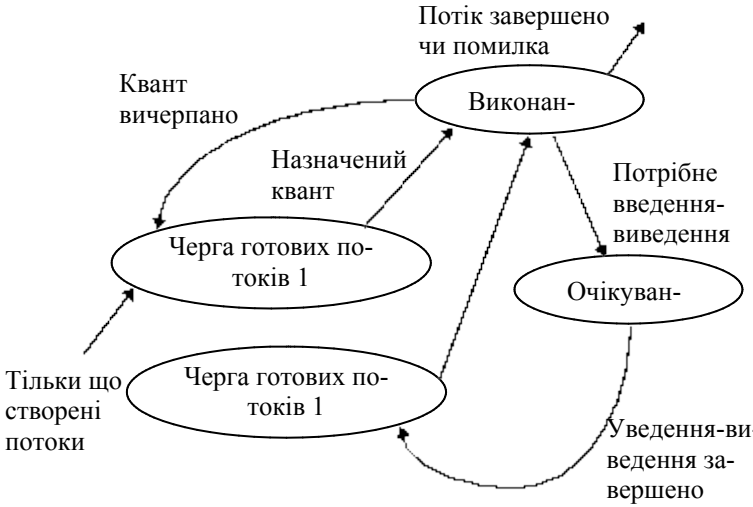


Рис. 2.6. Квантування з перевагою потоків, що інтенсивно звертаються до введення-виведення

Багатозадачні ОС втрачають деяку кількість процесорного часу для виконання допоміжних робіт під час перемикання контекстів завдань. При цьому запам’ятовуються і відновлюються регістри, прапори та вказівники стека, а також перевіряється статус завдань для передачі керування. Витрати на ці допоміжні дії не за-

лежать від величини кванта часу, тому чим більший квант, тим менші сумарні накладні витрати, пов'язані з перемиканням потоків.

Примітка. *В алгоритмах, заснованих на квантуванні, якою б не була їх мета (перевага коротких або довгих завдань, компенсація недовикористаного кванта або мінімізація накладних витрат, пов'язаних з перемиканнями), не використовується ніякої попередньої інформації про завдання. Під час надходження завдання на оброблення ОС не має ніяких відомостей про те, чи є воно коротким або довгим, наскільки інтенсивними будуть його запити до пристроїв уведення-виведення, наскільки важливе його швидке виконання і т. ін. Диференціація обслуговування під час квантування базується на «історії існування» потоку в системі.*

2.2.3. Алгоритми планування, основані на пріоритетах

Іншою важливою концепцією, покладеною в основу багатьох витісняльних алгоритмів планування, є пріоритетне обслуговування. Пріоритетне обслуговування припускає наявність у потоків деякої спочатку відомої характеристики – пріоритету, на підставі якої визначається порядок їх виконання.

Пріоритет – це числове значення, що характеризує ступінь привілейова-ності потоку для використання ресурсів обчислювальної машини, зокрема процесорного часу: чим вищий пріоритет, тим вищий привілей, тим менше часу проводитиме потік у чергах.

Пріоритет може виражатися цілим або дробовим, додатним або від'ємним значенням. У деяких ОС пріоритет потоку тим вищий, чим більше (у арифметичному сенсі) його значення. В інших системах, навпаки, чим менше значення, тим вищий пріоритет.

У більшості ОС, що підтримують потоки, пріоритет потоку безпосередньо пов'язаний з пріоритетом процесу, у межах якого виконується цей потік. Пріоритет процесу призначається ОС під час його створення. Значення пріоритету включається в описувач процесу і використовується під час призначення пріоритету потокам цього процесу. Для призначення пріоритету знову створеному процесу ОС враховує: чи є цей процес системним чи прикладним, який статус користувача, що запустив процес, чи була явна вказівка користувача на привласнення процесу певного рівня пріоритету. Потік може бути ініційований не тільки за командою користувача,

але і внаслідок виконання системного виклику іншим потоком. У цьому випадку під час призначення пріоритету новому потоку ОС має брати до уваги значення параметрів системного виклику.

У багатьох ОС передбачається можливість зміни пріоритетів протягом життя потоку. Зміна пріоритету може відбуватися за ініціативою самого потоку, коли він звернеться з відповідним викликом до ОС, або за ініціативою користувача, коли він виконує відповідну команду. Крім того, ОС сама може змінювати пріоритети потоків залежно від ситуації, що складається в системі. В останньому випадку пріоритети називаються *динамічними* на відміну від незмінних фіксованих пріоритетів.

Від того, які пріоритети призначені потокам, істотно залежить ефективність роботи всієї обчислювальної системи. У сучасних ОС, щоб уникнути розбалансування системи, яке може виникнути у разі неправильного призначення пріоритетів, можливості користувачів впливати на пріоритети процесів і потоків прагнуть обмежувати. При цьому звичайні користувачі, як правило, не мають права підвищувати пріоритети своїм потокам, це дозволено робити (ще й з обмеженнями) тільки адміністраторам. У більшості ж випадків ОС привласнює пріоритети потокам за замовчуванням.

Як приклад розглянемо схему призначення пріоритетів потокам, прийняту в ОС Windows NT (рис. 2.7). У системі визначено 32 рівні пріоритетів і два класи потоків – потоки реального часу і потоки зі змінними пріоритетами. Діапазон від 1 до 15 включно відведено для потоків зі змінними пріоритетами, а від 16 до 31 – для більш критичних до часу потоків реального часу (пріоритет 0 зарезервований для системних цілей).

Під час створення процесу він, залежно від класу, отримує за замовчуванням базовий пріоритет у верхній або нижній частині діапазону. Базовий пріоритет процесу надалі може бути підвищений або знижений ОС. Спочатку потік отримує значення базового пріоритету з діапазону базового пріоритету процесу, в якому він був створений. Якщо, наприклад, значення базового пріоритету деякого процесу дорівнює K, тоді всі потоки цього процесу отримають базові пріоритети з діапазону $[K\text{-}2, K\text{+}2]$. Звідси видно, що, змінюючи базовий пріоритет процесу, ОС може впливати на базові пріоритети його потоків.

Рис. 2.7. Схема призначення пріоритетів у Windows NT

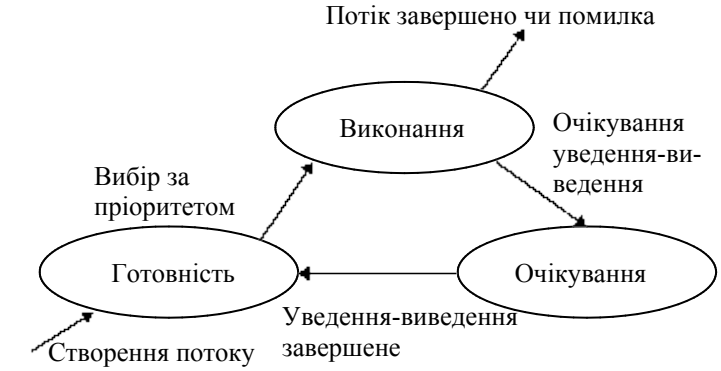
У Windows NT з часом пріоритет потоку, що належить до класу потоків зі змінними пріоритетами, може відхилятися від ба-зового пріоритету потоку, причому ці зміни можуть бути не пов'я-зані зі змінами базового пріоритету процесу. Операційна система може підвищувати пріоритет потоку (який в цьому випадку називають *динамічним*) у тих випадках, коли потік не повністю використовував відведений йому квант, або знижувати пріоритет, якщо квант був використаний повністю. Операційна система нарощує пріоритет диференціювання залежно від того, якого типу подія завадила потоку повністю використовувати квант. Зокрема, ОС підвищує пріоритет більшою мірою потокам, які чекають уведення з клавіатури (інтерактивним додаткам) і менше – потокам, що виконують дискові операції. Саме на основі динамічних пріоритетів плануються потоки. Початковою точкою відліку для динамічного пріоритету є значення базового пріоритету потоку. Значення динамічного пріоритету потоку обмежене знизу його базовим пріоритетом, верхньою ж межею є нижня межа діапазону пріоритетів реального часу.

Є два різновиди пріоритетного планування: обслугову-вання з відносними пріоритетами і обслуговування з абсолютними пріоритетами.

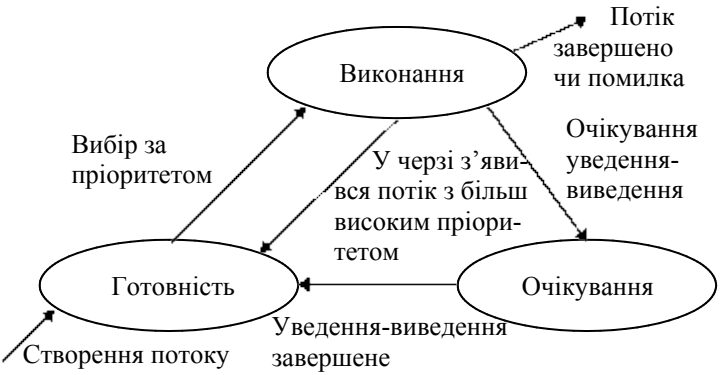
В обох випадках потік на виконання з черги готових вибирається однаково, тобто вибирається потік, що має найвищий пріоритет. Проте проблема визначення моменту зміни активного потоку вирішується по-різному. У системах з відносними пріоритетами активний потік виконується доти, доки він сам не покине процесор,

перейшовши в стан очікування (або ж станеться помилка, або потік завершиться). Граф станів потоку в системі з відносними пріоритетами показано на рис. 2.8.

У системах з абсолютними пріоритетами виконання актив-ного потоку переривається, крім указаних вище причин, ще за од-нієї умови: якщо в черзі готових потоків з'явився потік, пріоритет якого вищий за пріоритет активного потоку. В цьому випадку перерваний потік переходить у стан готовності (рис. 2.8).



а



б

Рис. 2.8. Графи станів потоків у системах з відносними і абсолютними пріоритетами

У системах, в яких планування здійснюється на основі відносних пріоритетів, мінімізуються витрати на перемикання про-цесора з виконання одного завдання на інше. З іншого боку, тут можуть виникати ситуації, коли одне завдання займає процесор тривалий час. Для систем розділення часу і реального часу така дисципліна обслуговування не підходить: інтерактивний додаток може чекати своєї черги годинами, поки обчислювальному завданню не буде потрібне введення-виведення. А в системах пакетного оброблення (зокрема відомою ОС OS/360) відносні пріоритети широко використовуються.

У системах з абсолютними пріоритетами час очікування потоку в чергах може бути зведений до мінімуму, якщо йому призначити найвищий пріоритет. Такий потік витіснятиме з процесора решту потоків (окрім потоків, що мають такий самий найвищий пріоритет). Це робить планування на основі абсолютних пріоритетів відповідним для систем керування об'єктами, для яких важлива швидка реакція на подію.

2.2.4. Змішані алгоритми планування

У багатьох ОС алгоритми планування побудовані з використанням як концепції квантування, так і пріоритетів. Наприклад, в основі планування лежить квантування, але величина кванта і/або порядок вибору потоку з черги готових визначається пріоритетами потоків. Саме так реалізується планування в системі Windows NT, у якій квантування поєднується з динамічними абсолютними пріоритетами. Для виконання вибирається готовий потік з найвищим пріоритетом. Йому виділяється квант часу. Якщо під час виконання в черзі готових потоків з'являється потік з вищим пріоритетом, то він витісняє виконуваний потік. Витіснений потік повертається в чергу готових, причому він стає попереду решти тих потоків, що мають такий самий пріоритет.

Розглянемо детальніше алгоритм планування в ОС UNIX System V Release 4. У цій ОС поняття потоку немає, і планування виконується на рівні процесів. У системі UNIX System V Release 4 реалізується витісняльна багатозадачність, що ґрунтується на використанні пріоритетів і квантування.

Кожен процес залежно від завдання, яке він виконує, належить до одного з трьох визначених у системі пріоритетних класів: класу реального часу, класу системних процесів або класу процесів розділення часу. Призначення і оброблення пріоритетів виконуються для різних класів по-різному. Процеси системного класу, зарезервовані для ядра, використовують стратегію фіксованих пріоритетів. Рівень пріоритету процесу призначається ядром і ніколи не змінюється.

Процеси реального часу також використовують стратегію фіксованих пріоритетів, але користувач може їх змінювати. Оскільки за наявності готових до виконання процесів реального часу інші процеси не розглядаються, то процеси реального часу треба ретельно проектувати, щоб вони не захоплювали процесор на дуже тривалий час. Характеристики планування процесів реального часу включають дві величини: рівень глобального пріоритету і квант часу. Для кожного рівня пріоритету за замовчуванням є своя величина кванта часу. Процесу дозволяється захоплювати процесор на вказаний квант часу, а після його закінчення планувальник знімає процес з виконання.

Процеси розділення часу були до появи UNIX System V Release 4 єдиним класом процесів, і за замовчуванням UNIX System V Release 4 призначає новому процесу саме цей клас. Склад класу процесів розділення часу найбільш невизначений і часто змінюється на відміну від системних процесів та процесів реального часу. Для справедливого розподілу часу процесора між процесами в цьому класі використовується стратегія динамічних пріоритетів. Величина пріоритету, що призначається процесам розділення часу, обчислюється пропорційно значенням двох складових: частині, призначеній для користувача, і системній частині. Частина пріоритету, призначена для користувача, може бути змінена адміністратором і власником процесу, але в останньому випадку тільки у бік його зниження.

Системна складова дозволяє планувальнику керувати процесами залежно від того, як довго вони займають процесор, не переходячи в стан очікування. У тих процесів, які споживають великі періоди процесорного часу без відходу в стан очікування, пріоритет знижується, а для тих процесів, які часто перебувають у стані очікування після короткого періоду використання процесора, прі-

оритет підвищується. Таким чином, процесам, які ведуть себе не «по-джентльменськи», надається низький пріоритет. Це означає, що вони рідше вибираються для виконання. Це «ущемлення прав» компенсується тим, що процесам з низьким пріоритетом надаються більші кванти часу, ніж процесам з високими пріоритетами. Таким чином, хоча низькопріоритетний процес і не працює так часто, як високопріоритетний, зате, коли він нарешті вибирається для виконання, йому відводиться більше часу.

Інший приклад стосується ОС OS/2. Планування тут ґрунтується на використанні квантування і абсолютних динамічних пріоритетів. На множині потоків визначаються пріоритетні класи – критичний (*time critical*), серверний (*server*), стандартний (*regular*) і залишковий (*idle*), у кожному з яких є 32 пріоритетні рівні. Потоки *критичного класу* мають найвищий пріоритет. До цього класу можуть належати, наприклад, системні потоки, що виконують завдання керування мережею. Наступний за пріоритетністю клас призначений, як це випливає з його назви, для потоків, що обслуговують серверні додатки. До *стандартного класу* можуть бути віднесені потоки звичайних додатків. Потоки, що входять у *залишковий клас*, мають найнижчий пріоритет. До цього класу відноситься, наприклад, потік, що виводить на екран заставку, коли в системі не виконується ніякої роботи.

Потік з менш пріоритетного класу не може бути обраний для виконання, поки в черзі більш пріоритетного класу є хоча б один потік. Усередині кожного класу потоки вибираються також за пріоритетами. Потоки, що мають однакове значення пріоритету, обслуговуються в циклічному порядку.

Пріоритети можуть змінюватися планувальником у таких випадках:

– якщо потік перебуває в очікуванні процесорного часу довше, ніж це задано системною змінною *MAXWAIT*, то його рівень пріоритету буде автоматично збільшений ОС. При цьому результуюче значення пріоритету має не перевищувати нижньої межі діапазону пріоритетів критичного класу;

– якщо потік надійшов на виконання операції введення-виведення, то після її завершення він набуде найвищого значення пріоритету свого класу;

– пріоритет потоку автоматично підвищується, коли він надходить на виконання.

Операційна система динамічно встановлює величину кванта, що відводиться потоку для виконання. Величина кванта залежить від завантаження системи та інтенсивності підкачування. Параметри налаштувань системи дозволяють явно задавати межі зміни кванта. В будь-якому випадку він не може бути меншим за 32 мс і більшим ніж 65 536 мс. Якщо потік був перерваний до закінчення кванта, то наступний виділе-ний йому інтервал виконання буде збільшений на якийсь час, що дорівнює одному періоду таймера (близько 32 мс), і так до тих пір, поки квант не досягне наперед заданої під час налаштування ОС межі.

Завдяки такому алгоритму планування в OS/2 жоден потік не буде «забутий» системою і отримає достатньо процесорного часу.

2.2.5. Планування в системах реального часу

У системах реального часу, головним критерієм ефективності яких є забезпечення часових характеристик обчислювального процесу, планування має особливе значення. Будь-яка система реального часу повинна реагувати на сигнали керованого об'єкта протягом заданих тимчасових обмежень. Необхідність ретельного планування робіт полегшується тим, що в системах реального часу важливий набір виконуваних завдань відомий наперед. Крім того, часто в системі є інформація про час виконання завдань, моменти активізації, граничнодопустимі терміни очікування відповіді та ін. Ці дані можуть бути використані планувальником для створення статичного розкладу або для побудови адекватного алгоритму динамічного планування.

Розробляючи алгоритми планування для систем реального часу, необхідно враховувати, які наслідки в цих системах виника-ють у разі недотримання тимчасових обмежень. Якщо ці наслідки катастрофічні, як, наприклад, для системи керування польотами або атомною електростанцією, то ОС реального часу, на основі якої будується керування об'єктом, називається *жорсткою* (*hard*). Якщо ж наслідки порушення тимчасових обмежень не такі суттєві, тобто порівняно з тією користю, яку дає система керування об'єктом, то система є *м'якою* (*soft*) *системою реального часу*. Прикладом м'якої системи реального часу є система резервування квитків. Як-

що через тимчасові порушення оператору не вдається зарезервувати квиток, тоді можна просто послати запит на повторне резервування.

У жорстких системах реального часу час завершення виконання кожного з критичних завдань має бути гарантованим для всіх можливих сценаріїв роботи системи. Такі гарантії можуть бути у разі або вичерпного тестування всіх можливих сценаріїв поведінки керованого об'єкта і керувальних програм, або побудови статичного розкладу, або вибору математично обґрунтованого динамічного алгоритму планування. Складаючи розклад, треба мати на увазі, що для деяких наборів завдань неможливо побудувати розклад, згідно з яким задовольнялися б задані часові характеристики. Для визначення наявності розкладу можна використовувати різні критерії. Наприклад, простим критерієм може бути така умова: різниця між граничним терміном виконання завдання (після появи запиту на її виконання) і часом її обчислення (за умови безперервного виконання) завжди повинна бути додатною. Очевидно, що такий критерій є необхідним, але недостатнім. Точні критерії, що гарантують наявність розкладу, є дуже складними для обчислення.

У м'яких системах реального часу передбачається, що задані часові обмеження можуть іноді порушуватися, тому тут зазвичай застосовують менш витратні способи планування.

Залежно від характеру виникнення запитів на виконання завдань корисно розділяти їх на два типи: *періодичні* і *спорадичні*. По-чинаючи з моменту первинного запиту всі майбутні моменти запиту періодичного завдання можна визначити заздалегідь збільшивши до моменту початкового запиту величину, кратну відомому періоду. Час запитів на виконання спорадичних завдань наперед не був відомим.

Припустімо, що є періодичний набір завдань $\{T_i\}$ з періодами p_i, граничними термінами d_i та вимогами до часу виконання c_i. Для перевірки наявності розкладу досить проаналізувати розклад у період часу, що дорівнює принаймні найменшому загальному множнику періодів цих завдань. Необхідним критерієм існування розкладу для набору періодичних завдань є таке достатньо очевидне твердження: сума коефіцієнтів використання $m_i = c_i / p_i$ має бути меншою або дорівнювати k, де k – кількість доступних процесорів, тобто:

$$m_i = Sum(c_i / p_i) <= k.$$

Вибираючи алгоритм планування, слід враховувати дані про можливу залежність завдань. Ця залежність може мати, наприклад, вигляд обмежень на послідовність виконання завдань або їх синхронізації, викликаної взаємними виключеннями (замкнене виконання деяких завдань протягом певних періодів часу).

З практичного погляду алгоритми планування залежних завдань важливіші, ніж алгоритми планування незалежних завдань. За наявності дешевих мікроконтролерів немає сенсу організовувати мультипрограмне виконання великої кількості незалежних завдань на одному комп'ютері, оскільки при цьому значно ускладнюється ПЗ. Зазвичай одночасно виконувані завдання повинні обмінюватися інформацією та діставати доступ до загальних даних для досягнення загальної мети системи, тобто є залежними завданнями. Тому деяка перевага послідовності виконання завдань або взаємного виключення – це швидше норма для систем керування реального часу, ніж виключення.

Проблема планування залежних завдань дуже складна, знаходження її оптимального вирішення потребує великих обчислювальних ресурсів, порівняних з тими, які потрібні для власного виконання завдань керування. Вирішити цю проблему можна таким чином:

– розділити проблему планування на дві частини, щоб одна частина виконувалася наперед, перед запуском системи, а друга, простіша частина – під час роботи системи. Попередній аналіз набору завдань із взаємними виключеннями може полягати, наприклад, у виявленні заборонених діапазонів часу, протягом яких не можна призначати виконання завдань, що містять критичні секції;

– увести обмежувальні припущення про поведінку набору завдань.

За такого підходу планування наближається до статичного.

Планування незалежних завдань потребує класичного алгоритму для жорстких систем реального часу з одним процесором, розробленим в 1973 р. Ллю (*Liu*) і Лейландом (*Layland*). Алгоритм є динамічним, тобто використовує витісняльну багатозадачність і заснований на відносних статичних (незмінних протягом життя завдання) пріоритетах.

2.2.6. Алгоритм для жорстких систем реального часу з одним процесором

Алгоритм ґрунтується на таких припущеннях:

– запити на виконання всіх завдань набору, що мають жорсткі обмеження на час реакції, є періодичними;

– усі завдання незалежні. Між будь-якою парою завдань не існує ніяких обмежень на передування або на взаємне виключення;

– термін виконання кожного завдання дорівнює його періоду p_i;

– максимальний час виконання кожного завдання c_i відомий і постійний;

– час переключення контексту можна ігнорувати;

– максимальний сумарний коефіцієнт завантаження процесора Xc_i/p_i при існуванні n завдань не перевершує $n(2^{1/n} - 1)$. Ця величина при прагненні n до нескінченності приблизно дорівнює $ln\ 2$, тобто 0.7.

Суть алгоритму полягає в тому, що для всіх завдань призначаються статичні пріоритети відповідно до величини їх періодів виконання. Завдання з найкоротшим періодом отримує найвищий пріоритет, а завдання з найбільшим періодом виконання отримує найменший пріоритет. У разі дотримання всіх обмежень цей алгоритм гарантує виконання часових обмежень для всіх завдань у всіх ситуаціях.

Якщо ж періоди повторення завдань кратні періоду виконання найкоротшого завдання, то вимога до максимального коефіцієнта завантаження процесора зменшується – він може досягати значення 1.

Існують також алгоритми з динамічною зміною пріоритетів, які призначаються відповідно до таких поточних параметрів завдання як, наприклад, кінцевий термін виконання (*deadline*). У разі потреби призначати деяке завдання на виконання вибирається те, в якого поточне значення різниці між кінцевим терміном виконання і часом, потрібним для його безперервного виконання, є найменшим.

2.2.7. Моменти перепланування

Для реалізації алгоритму планування ОС повинна отримувати керування щоразу, коли в системі відбувається подія, яка потребує перерозподілу процесорного часу. Такими подями можуть бути такі:

– переривання від таймера сигналізує, що час, відведений для активного завдання на виконання, закінчився. Планувальник переводить завдання в стан готовності і виконує перепланування;

– активне завдання виконало системний виклик, пов'язаний із запитом на введення-виведення або на доступ до ресурсу, який зараз зайнятий (наприклад, файл даних). Планувальник переводить завдання в стан очікування і виконує перепланування;

– активне завдання виконало системний виклик, пов'язаний із звільненням ресурсу. Планувальник перевіряє, чи не чекає цей ресурс якого-небудь завдання. Якщо так, то це завдання переводиться зі стану очікування в стан готовності. При цьому, можливо, що завдання, яке отримало ресурс, має вищий пріоритет, ніж поточне активне завдання. Після перепланування більш пріоритетне завдання дістає доступ до процесора, витісняючи поточне завдання;

– зовнішнє (апаратне) переривання, яке сигналізує про завершення периферійним пристроєм операції введення-виведення, переводить відповідне завдання в чергу готових, і виконується планування.

– внутрішнє переривання сигналізує про помилку, яка відбулася внаслідок виконання активного завдання. Планувальник знімає завдання і виконує перепланування.

Із виникненням кожної з цих подій планувальник переглядає черги і визначає, яке завдання виконуватиметься наступним. Інші події (часто зумовлені системними викликами) потребують перепланування. Наприклад, запити додатків і користувачів на створення но-вого завдання або підвищення пріоритету вже існуючого завдання створюють нову ситуацію, яка вимагає перегляду черг і, можливо, перемикання процесора.

Фрагмент часової діаграми роботи планувальника в системі, де одночасно виконуються чотири потоки показано на рис. 2.9. У цьому випадку неважливо, за яким правилом вибираються потоки на виконання і яким чином змінюються їх пріоритети. Істотне значення мають лише події, що викликають активізацію планувальника.

Перші чотири цикли роботи планувальника (рис. 29) були ініційовані перериваннями від таймера після закінчення квантів часу (ці події позначені на рисунку як *T*).

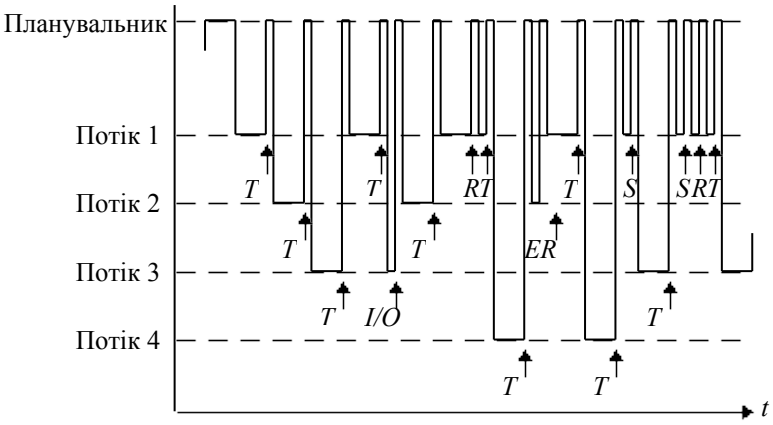


Рис. 2.9. Моменти перепланування потоків

Наступну передачу керування планувальнику було здійснено в результаті виконання потоком 3 системного запиту на введення-виведення (подія *I/O*). Планувальник перевів цей потік у стан очікування, а потім перемикнув процесор на потік 2. Потік 2 повністю використав свій квант, відбулося переривання від таймера, і планувальник активізував потік 1.

Під час виконання потоку 1 відбулася подія *R* – системний виклик, у результаті якого звільнився деякий ресурс (наприклад, був закритий файл). Ця подія викликала перепланування потоків. Планувальник переглянув чергу потоків і виявив, що потік 4 чекає звільнення цього ресурсу. Цей потік був переведений в стан готовності, але оскільки пріоритет потоку 1, що виконується в цей момент часу, вищий за пріоритет потоку 4, планувальник повернув процесор потоку 1.

У наступному циклі роботи планувальник активізував потік 4, а потім, після закінчення кванта та сигналу від таймера, керування отримав потік 2. Цей потік не встиг використати свій квант, оскільки був знятий з виконання в результаті виниклої помилки (подія *ER*).

Далі планувальник надавав процесорний час потокам 1, 4 і знову 1. Під час виконання потоку 1 відбулося переривання *S* від зовнішнього пристрою, що сигналізує про завершення операції передачі даних. Ця подія активізувала роботу планувальника, у результаті якої потік 3, що чекав завершення введення-виведення, витіснив потік 1, оскільки мав у цей момент часу вищий пріоритет.

Останній показаний на діаграмі період виконання потоку 1 переривався кілька разів. Спочатку це було переривання від зовнішнього пристрою (*S*), потім програмне переривання (*R*), що викликало звільнення ресурсу, і, нарешті, переривання від таймера (*T*). Кожне з цих трьох переривань викликало перепланування потоків. У двох перших випадках планувальник залишив виконуватися по-тік 1, оскільки в черзі не опинилося більш пріоритетних потоків, а квант часу, виділений потоку 1, ще не був вичерпаний. Перемикання потоків було виконане тільки за перериванням від таймера.

У системах реального часу для реалізації статичного розкладу планувальник активізується за перериваннями від таймера. Ці пе-реривання виникають через короткі постійні інтервали часу. Після кожного переривання планувальник переглядає розклад і перевіряє, чи не пора перемкнути завдання. Крім переривань від таймера, в системах реального часу перепланування завдань може відбуватися за перериваннями від зовнішніх пристроїв – різного виду датчиків і виконавчих механізмів.

2.3. Мультипрограмування на основі переривань. Системні виклики

2.3.1. Призначення і типи переривань

Переривання є основною рушійною силою будь-якої ОС. Якщо відключити систему переривань – «життя» в ОС негайно зупиниться. Періодичні переривання від таймера змінюють процеси в мультипрограмній ОС, а переривання від пристроїв уведення-виведення керують потоками даних, якими обчислювальна система обмінюється із зовнішнім середовищем.

Система переривань перекладає на процесор виконання потоку команд, відмінного від того, який виконувався до цих пір, з по-

дальшим поверненням до вихідного коду. Таким чином, можна зробити висновок про те, що механізм переривань дуже схожий на механізм виконання процедур. Це насправді так, хоча між цими механізмами є важлива відмінність. Перемикання за перериванням відрізняється від перемикання за командою безумовного або умовного переходу, передбаченою програмістом у потоці команд додатка. Перехід за командою відбувається в заздалегідь визначених програмістом точках програми залежно від вихідних даних, що обробляються програмою. Переривання ж відбувається в довільній точці потоку команд програми, яку програміст не може прогнозувати. Переривання залежить від зовнішніх відносно процесу виконання програми подій, або від появи непередбачених аварійних ситуацій в процесі виконання цієї програми. Схожість переривань з процедурами полягає в тому, що в обох випадках виконується деяка підпрограма, яка обробляє спеціальну ситуацію, а потім продовжується виконання основної гілки програми.

Залежно від джерела переривання поділяють на три великі класи:

– зовнішні;
– внутрішні;
– програмні.

Зовнішні переривання можуть спричиняти дії користувача чи оператора за терміналом, або сигнали, що надходять від апаратних пристроїв – сигнали завершення операцій введення-виведення, зовнішніх пристроїв комп'ютера, що виробляються контролерами, такими як принтер або нагромаджувач на жорстких дисках, або ж сигнали від датчиків керованих комп'ютером технічних об'єктів.

Зовнішні переривання називають також *апаратними*, відображаючи той факт, що переривання виникає унаслідок подачі деякою апаратурою (наприклад, контролером принтера) електричного сигналу, який передається (можливо, проходячи через інші блоки комп'ютера, наприклад контролер переривань) на спеціальний вхід переривання процесора. Цей клас переривань є асинхронним відносно потоку інструкцій програми, що переривається. Апаратура процесора працює так, що асинхронні переривання виникають між виконанням двох сусідніх інструкцій, при цьому система після оброблення переривання продовжує виконувати процес, вже починаючи з наступної інструкції.

Внутрішні переривання, названі також *виключеннями* (*exeption*), відбуваються синхронно до виконання програми в разі появи аварійної ситуації в ході виконання деякої інструкції програми. Прикладами виключень є ділення на нуль, помилки захисту пам'яті, звернення за неіснуючою адресою, спроба виконати привілейовану інструкцію в призначеному для користувача режимі й інші подібні виключення, що виникають безпосередньо в ході виконання тактів команди («усередині» виконання).

Програмні переривання відрізняються від попередніх двох класів тим, що вони за своєю суттю не є дійсними перериваннями. Програмне переривання відбувається під час виконання особливої команди процесора, виконання якої імітує переривання, тобто пере-хід на нову послідовність інструкцій.

Перериванням приписується пріоритет, за допомогою якого вони ранжируються за значущістю і терміновістю. Переривання, що мають однакове значення пріоритету, належать до одного рівня пріоритету переривань.

Переривання зазвичай обробляються модулями ОС, оскільки дії, що виконуються за перериванням, належать до керування поділюваними ресурсами обчислювальної системи (принтером, диском, таймером, процесором і т. ін.)

Процедури, що викликаються за перериваннями називають *обробниками переривань*, або *процедурами обслуговування переривань* (*Interrupt Service Routine*). Апаратні переривання оброб-ляються драйверами відповідних зовнішніх пристроїв, виключення – спеціальними модулями ядра, а програмні переривання – проце-дурами ОС, які обслуговують системні виклики. Окрім цих модулів в складі ОС може бути *диспетчер переривань*, який координує роботу окремих обробників переривань.

2.3.2. Механізм переривань

Механізм переривань підтримується апаратними засобами комп'ютера і програмними засобами ОС.

Апаратна підтримка переривань має свої особливості, що залежать від типу процесора та інших апаратних компонентів, які передають сигнал запиту переривання від зовнішнього пристрою до процесора (контролер зовнішнього пристрою, шини підключення

зовнішніх пристроїв, контролер переривань, що є посередником між сигналами шини та сигналами процесора). Особливості апаратної реалізації переривань впливають на засоби програмної підтримки переривань, що працюють у складі ОС.

Шини виконують переривання за допомогою двох основних способів: *векторним* (*vectored*) і *опитуваним* (*polled*). За обома способами процесору надається інформація про рівень пріоритету переривання на шині підключення зовнішніх пристроїв. У разі векторних переривань у процесор надходить також інформація про початкову адресу програми оброблення виниклого переривання – *обробника переривань*.

Пристроям, які використовують векторні переривання, призначається *вектор переривань*. Це електричний сигнал, що подається на відповідні шини процесора і містить інформацію про визначений, закріплений за цим пристроєм номер, який ідентифікує відповідний обробник переривань. Цей вектор може бути *фіксованим*, *конфігурованим* (наприклад, з використанням перемикачів) або *програмованим*.

Операційна система може передбачати процедуру реєстрації вектора оброблення переривань для певного пристрою, яка пов'язує деяку підпрограму оброблення переривань з певним вектором. Отримавши сигнал запиту переривання, процесор виконує спеціальний цикл підтвердження переривання, за яким пристрій повинен ідентифікувати себе. Протягом цього циклу пристрій відповідає, виставляючи на шину вектор переривань, а процесор використовує цей вектор для знаходження обробника цього переривання. Прикладом шини підключення зовнішніх пристроїв, яка підтримує векторні переривання, є шина Vmebus.

Використовуючи опитувані переривання, процесор отримує від пристрою, що запитав переривання, лише інформацію про рівень пріоритету переривання (наприклад, номер IRQ на шині ISA або номер IPL на шині Sbus комп'ютерів SPARC). З кожним рівнем переривань може бути пов'язано декілька пристроїв і відповідно декілька програм – обробників переривань. Коли виникає переривання, процесор повинен визначити, який пристрій з тих, що пов'язані з цим рівнем переривань, дійсно запитав переривання. Це досягається викликом усіх обробників переривань для цього рівня пріоритету, доки один з обробників не підтвердить, що перериван-ня

надійшло від обслуговуваного ним пристрою. Якщо від кожного рівня переривань залежить лише один пристрій, то потрібна програма оброблення переривання визначається негайно, як і у випадку векторного переривання. Опитувані переривання підтримують шини ISA, EISA, MCA, PCI і Sbus.

Механізм переривань деякої апаратної платформи може поєднувати і векторний, і опитуваний типи переривань. Типовим прикладом такої реалізації є платформа ПК на основі процесорів Intel Pentium. Шини PCI, ISA, EISA або MCA, використовувані в цій платформі як шини підключення зовнішніх пристроїв, підтримують механізм опитуваних переривань. Контролери периферійних при-строїв виставляють на шину не вектор, а сигнал запиту перериван-ня певного рівня IRQ. Проте в процесорі Pentium система перери-вань є векторною. Вектор переривань у процесор Pentium поставляє контролер переривань, який відображує сигнал IRQ, що надходить від шини, на певний номер вектора.

Вектор переривань, передаваний в процесор, є цілим числом в діапазоні від 0 до 255, що вказує на одну з 256 програм оброблення переривань, адреси яких зберігаються в таблиці обробників переривань. У тому випадку, коли до кожної лінії IRQ підключається лише один пристрій, процедура оброблення переривань працює так, як якби система переривань була чисто векторною, тобто процедура не виконує жодних додаткових опитувань для з'ясуван-ня того, який саме пристрій запитав переривання. Проте у разі спільного використання одного рівня IRQ декількома пристроями програма оброблення переривань повинна працювати відповідно до схеми опитуваних переривань, тобто додатково виконувати опитування всіх пристроїв залежно від рівня IRQ.

Механізм переривань найчастіше підтримує пріоритезацію і маскування переривань. Пріоритезація означає, що всі джерела переривань діляться на класи й кожному класу призначається свій рівень пріоритету запиту на переривання. Пріоритети можуть бути відносними й абсолютними. Обслуговування запитів переривань за схемою з відносними пріоритетами полягає в тому, що в разі одночасного надходження запитів переривань з різних класів вибирається запит, який має вищий пріоритет. Проте надалі під час обслуговування цього запиту процедура оброблення переривання вже не відкладається навіть у тому випадку, якщо з'являються більш пріоритетні запити – рішення про вибір нового запиту приймається

лише у момент завершення обслуговування чергового переривання.

Якщо ж більш пріоритетним перериванням дозволяється припиняти роботу процедур обслуговування менш пріоритетних переривань, то це означає, що працює схема пріоритезації з абсолютними пріоритетами. Якщо процесор (або комп'ютер, коли підтримка пріоритезації переривань винесена в зовнішній відносно процесора блок) працює за схемою з абсолютними пріоритетами, то він підтримує в одному зі своїх внутрішніх регістрів змінну, що фіксує рівень пріоритету обслуговуваного в цей момент часу переривання. Під час надходження запиту з певного класу його пріоритет порівнюється з поточним пріоритетом процесора, і якщо пріоритет запиту вищий, то поточна процедура оброблення переривань витісняється, а після закінчення обслуговування нового переривання повертається до перерваної процедури.

Упорядковане обслуговування запитів переривань поряд зі схемами пріоритетного оброблення запитів може виконуватися механізмом маскування запитів. В описаній схемі абсолютних пріоритетів виконується маскування – у процесі обслуговування дея-кого запиту всі запити з однаковим або нижчим пріоритетом маскуються, тобто не обслуговуються. Схема маскування передбачає можливість часового маскування переривань будь-якого класу незалежно від рівня пріоритету.

Послідовність дій апаратних і програмних засобів з оброблення переривання можна описати таким чином.

З надходженням сигналу (для апаратних переривань) або умови (для внутрішніх переривань) переривання відбувається первинне апаратне розпізнавання типу переривання. Якщо переривання цього типу заборонені (пріоритетною схемою або механізмом маскування), то процесор продовжує підтримувати природний хід виконання команд. Інакше залежно від інформації (рівня переривання, вектора переривання або типу умови внутрішнього пе-реривання), що надійшла в процесор, відбувається автоматичний виклик процедури оброблення переривання, адреса якої міститься в спеціальній таблиці ОС, розміщеній або в регістрах процесора, або у визначеному місці оперативної пам'яті.

Автоматично зберігається деяка частина контексту перерваного потоку, яка дозволить ядру відновити виконання потоку про-

цесу після оброблення переривання. У цю підмножину зазвичай включаються значення лічильника команд, слова стану машини, що зберігає ознаки основних режимів роботи процесора (приклад такого слова – регістр Efla6s в Intel Pentium), а також декількох регістрів загального призначення, які потрібні для програми оброблення переривання. Може бути збережений і повний контекст процесу, якщо ОС обслуговує переривання зі зміною процесу. Проте в загальному випадку це не є обов'язковим, часто переривання обробляється без витіснення поточного процесу. Рішення про перепланування процесів може бути прийняте в ході оброблення переривання, наприклад, якщо це переривання від таймера і після нарощування значення системного годинника з'ясовується, що процес вичерпав виділений йому квант часу. Проте це зовсім не обов'язково – переривання може виконуватися і без зміни процесу, наприклад чергова порція даних від контролера зовнішнього пристрою найчастіше надходить у межах поточного процесу, хоча дані, швидше, призначені для іншого процесу.

Одночасно із завантаженням адреси процедури оброблення переривань у лічильник команд може автоматично завантажуватися нове значення слова стану машини (або іншої системної структури, наприклад селектора кодового сегменту в процесорі Pentium), який визначає режими роботи процесора під час оброблення переривання, у тому числі роботу в привілейованому режимі. У деяких моделях процесорів перехід у привілейований режим за рахунок зміни стану машини у процесі оброблення переривання є єдиним способом заміни. Переривання майже у всіх мультипрограмних ОС обробляються в привілейованому режимі модулями ядра, оскільки при цьому зазвичай потрібно виконати ряд критичних операцій, від яких залежить життєздатність системи, – керувати зовнішніми пристроями, перепланувати потоки і т. ін.

Тимчасово забороняються переривання певного типу, щоб не утворилася черга вкладених один в один потоків однієї й тієї ж процедури. Деталі виконання цієї операції залежать від особливостей апаратної платформи, наприклад можна використовувати механізм маскування переривань. Багато процесорів автоматично встановлюють ознаку заборони переривань на початку циклу оброблення переривання, інакше це робить програма оброблення переривань.

Після оброблення переривання ядром ОС перерваний контекст відновлюється і робота потоку поновлюється з перерваного місця. Частина контексту відновлюється апаратно за командою повернення з переривань (наприклад, адреси наступної команди і слова стану машини), а частина – програмним способом, за допомогою явних команд витягання даних із стеку. Після повернення з переривання блокування повторних переривань цього типу припиняється.

2.3.3. Програмне переривання

Програмне переривання реалізовує один із способів переходу на підпрограму за допомогою спеціальної інструкції процесора, такої як *INT* у процесорах Intel Pentium, *trap* у процесорах Motorola, *syscall* у процесорах MIPS або *Ticc* у процесорах SPARC. У ході виконання команди програмного переривання процесор відпрацьовує ту ж послідовність дій, що й у разі виникнення зовнішнього або внутрішнього переривання, тільки відбувається це в передбаченій точці програми – там, де програміст помістив цю програму.

Усі сучасні процесори мають в системі команд інструкції програмних переривань. Однією з причин появи інструкцій програмних переривань у системі команд процесорів є те, що їх використання часто приводить до компактнішого коду програм порівняно з використанням стандартних команд виконання процедур. Це пояснюється тим, що розробники процесора зазвичай резервують для оброблення переривань невелику кількість можливих підпрограм, тому довжина операнда в команді програмного переривання, який вказує на потрібну підпрограму, менша, ніж в команді переходу на підпрограму. Наприклад, у процесорі x86 передбачена можливість вживання 256 програм оброблення переривань, тому в інструкції *INT* операнд має довжину один байт (а інструкція *INT* 3, яка призначена для виклику налагоджувача, вся має довжину один байт). Значення операнда команди *INT* просто є індексом у таблиці з 256 адрес підпрограм оброблення переривань, один з яких і використовується для переходу за командою *INT*. Якщо використовувати команду *CALL*, то потрібно вже не однобайтовий, а дво- або чотирибайтовий операнд. Іншою причиною використання програмних переривань замість звичайних інструкцій виклику підпрограм є можливість змінювати призначений для користувача режим на привілейований одночасно з викли-

ком процедури – ця властивість програмних переривань підтримується більшістю процесорів. У результаті програмні переривання часто використовуються для виконання обмеженої кількості викликів функцій ядра ОС, тобто системних викликів.

2.3.4. Диспетчеризація і пріоритезація переривань в операційній системі

Операційна система повинна відігравати активну роль в організації оброблення переривань. Переривання виконують дуже корисну для обчислювальної системи функцію – вони дозволяють реагувати на асинхронні до обчислювального процесу події. Водночас переривання створюють додаткові труднощі для ОС в організації обчислювального процесу. Ці труднощі зумовлені з непередбаченими переходами керування від однієї процедури до іншої, спричиненими перериваннями від контролерів зовнішніх пристроїв. Можуть також виникати в непередбачені моменти часу виключення, викликані помилками під час виконання інструкцій. Ускладнюють завдання планування обчислювальних робіт і запити на виконання системних функцій (системні виклики) від призначених для користувача застосувань, здійснювані за допомогою програмних переривань. Самі модулі ОС також часто викликають один одного за програмними перериваннями, ще більше заплутуючи обчислювальний процес.

Операційна система не може втрачати контроль над ходом виконання системних процедур, що викликаються за перериваннями. Вона повинна впорядковувати їх у часі так само, як планувальник упорядковує численні, призначені для користувача, потоки. Крім того, сам планувальник потоків є системною процедурою, що викликається за перериваннями (апаратним – від таймера або контролера пристрою введення-виведення, або програмним – від додатка або модуля ОС). Тому правильне планування процедур, що викликаються за перериваннями, є необхідною умовою правильного планування призначених для користувача потоків. Інакше в системі можуть виникати, наприклад, такі ситуації, коли ОС тривалий час займатиметься завданням, що потребує миттєвої реакції на керування стримером, який архівує дані, в той час, коли високошвидкісний диск простоюватиме і гальмуватиме роботу численних

застосувань, що обмінюються даними з цим диском. Приклад такої ситуації ілюструє рис. 2.10. Обробник переривань принтера блокує на тривалий час оброблення переривання від таймера, внаслідок чого системний час на деякий час «завмирає» і потік 2, критично важливий для користувача, не отримує керування в установлений час. Гостроту проблеми пом'якшує та обставина, що у багатьох випадках оброблення переривання залежить від виконання декількох операцій введення-виведення і тому має дуже малу тривалість. Проте ОС завжди повинна контролювати ситуацію і виконувати критичну роботу вчасно, а не покладатися на випадок.

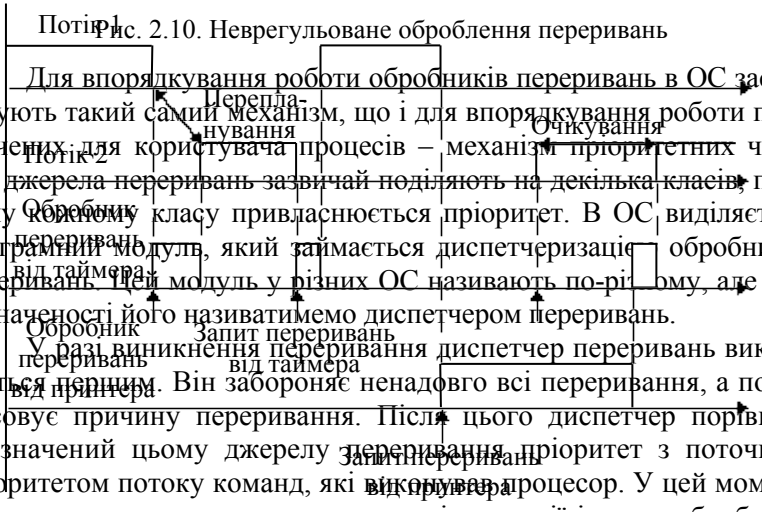


Рис. 2.10. Неврегульоване оброблення переривань

Для впорядкування роботи обробників переривань в ОС застосовують такий самий механізм, що і для впорядкування роботи призначених для користувача процесів – механізм пріоритетних черг. Усі джерела переривань зазвичай поділяють на декілька класів, причому кожному класу привласнюється пріоритет. В ОС виділяється програмний модуль, який займається диспетчеризацією обробників переривань. Цей модуль у різних ОС називають по-різному, але для визначеності його називатимемо диспетчером переривань.

У разі виникнення переривання диспетчер переривань викликається першим. Він забороняє ненадовго всі переривання, а потім з'ясовує причину переривання. Після цього диспетчер порівнює призначений цьому джерелу переривання пріоритет з поточним пріоритетом потоку команд, які виконував процесор. У цей момент часу процесор вже може виконувати інструкції іншого обробника переривань, що також має деякий пріоритет. Якщо пріоритет нового запиту вищий від поточного, то виконання інструкції поточного обробника припиняється і він поміщається у відповідну чергу обробників переривань. Інакше в чергу поміщається обробник нового запиту.

Примітка. *Пріоритет обробників переривань не збігається в загальному випадку з пріоритетом потоків, що виконуються в звичайній послідовності, визначуваній планувальником потоків. Будь-який потік, призначений на виконання планувальником, має найнижчий пріоритет, так що будь-який запит на переривання завжди може перервати виконання цього потоку.*

Функції централізованого диспетчера переривань на прикладі Windows NT. Деякі процесори або контролери переривань комп'ютера на апаратному рівні підтримують пріоритезацію запитів на переривання. Наприклад, у процесорах MIPS є декілька рівнів апаратних запитів на переривання і декілька рівнів програмних запитів. Процесор має внутрішню змінну, названу рівнем пере-ривань процесора. Переривання відбувається лише у тому випадку, коли рівень запиту на переривання вищий від поточного рівня переривань процесора. Є також привілейована інструкція, за допомогою якої код ядра ОС може змінити рівень переривань процесора. Необроблені запити на переривання мають зберігатися в контро-лерах пристроїв, щоб не загубитися й дочекатися обслуговування при зниженні рівня переривання процесора, коли він закінчить виконання терміновіших робіт. На такій апаратній платформі ОС може користуватися вбудованими в процесор засобами пріоритезації переривань для впорядковування процесу їх оброблення. Проте такі засоби є не у всіх процесорах, наприклад процесори сім'ї Pentium не мають вбудованої змінної для фіксації рівня переривань виконуваного коду – апаратні переривання можуть бути або повністю заборонені, або повністю дозволені (а найбільш критичні не можна забороняти взагалі).

Для усунення залежності від апаратної платформи в деякі ОС упроваджують власну програмну систему пріоритетів переривань. Прикладом такої ОС є Windows NT.

Диспетчер переривань Windows NT (Trap Handler) працює з програмною моделлю переривань, єдиною для всіх апаратних платформ, підтримуваних Windows NT. Джерела переривань (апаратних і програмних, а також деяких важливих для системи виключень, наприклад помилкове виключення шини) поділяться на декілька класів, і кожному класу привласнюється рівень запиту переривання (Interrupt Request Level – IRQL). Цей рівень і є пріоритетом цього класу. ОС програмним способом підтримує внутрішню змінну названу IRQL виконуваного процесором коду, яка за призначенням відповідає рівню переривання процесора. Якщо процесор, на якому працює ОС, підтримує таку змінну, то вона використовується і IRQL виконуваного коду відображується на ній, інакше відповідні функції процесора емулюються програмно.

Загальна схема планування оброблення переривань у Windows NT така. Якщо до процесора надходить сигнал запиту на переривання/виключення, викликається диспетчер переривань, який запам'ятовує інформацію про джерело переривання і аналізує його пріоритет. Якщо пріоритет запиту нижчий або дорівнює IRQL перерваного коду, то обслуговування цього запиту відкладається і дані про запит поміщаються у відповідну чергу запитів, після чого відбувається швидке повернення до перерваного обробника переривань. Після оброблення високопріоритетного переривання керування повертається диспетчеру переривань, який переглядає черги відкладених переривань і вибирає з них найбільш пріоритетне. При цьому рівень IRQL знижується до рівня вибраного переривання.

Якщо ж запит має вищий пріоритет, ніж IRQL поточного коду, то поточний обробник переривань витісняється і ставиться в чергу, що відповідає його значенню IRQL, а керування передається новому обробнику відповідно до запиту IRQL. Після цього рівень IRQL процесора дорівнюватиме рівню IRQL, прийнятого на виконання запиту.

Таким чином, запит на переривання приймається диспетчером переривань завжди незалежно від поточного рівня IRQL виконуваного коду, але диспетчер переривань не передає його на оброблення відповідній процедурі оброблення переривань, а поміщає в програмну чергу запитів, якщо в цей момент часу виконується більш пріоритетна процедура оброблення переривань. Операційна система цілком контролює ситуацію, не дозволяючи контролерам пристроїв введення-виведення приймати рішення про хід обчислювального процесу.

У Windows NT нижчий рівень IRQL відповідає звичайним потокам, призначуваним на виконання диспетчером потоків (рис. 2.11). Це є деяким допущенням, оскільки код потоків починає виконуватися процесором не в результаті запиту на переривання, але це допущення справедливе, оскільки дозволяє будь-якому «справжньому» запиту переривати код звичайного потоку.

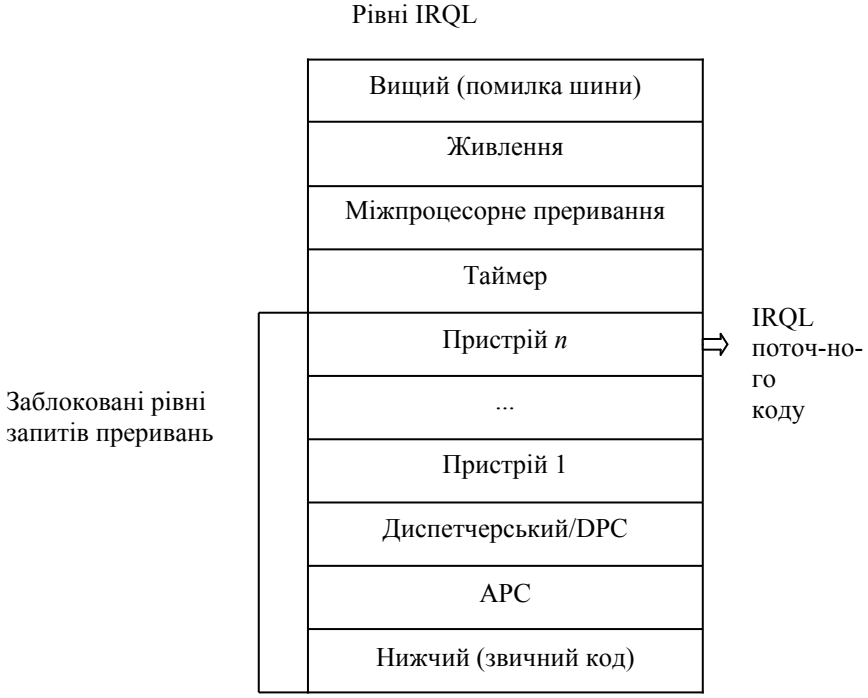


Рис. 2.11. Диспетчеризація переривань у Windows NT

Вищий рівень в ієрархії IRQL відводиться таким важливим подіям, як помилкове від'єднання шини і іншим важким апаратним збоям, далі розміщуються запит на переривання, зумовлене збоєм живлення і запит на міжпроцесорне переривання.

Переривання від зовнішніх пристроїв займають проміжні рівні IRQL. Конкретне співвідношення між пріоритетами зовнішніх пристроїв визначається пріоритетами, що задаються апаратною платформою, наприклад рівнем IRQ шини PCI, призначеним пристрою.

Особливу роль у функціонуванні обчислювальної системи відіграє системний таймер: на підставі його переривань оновлюється системний годинник, що визначає черговий момент виклику планувальника потоків, момент видачі дії, що керує, потоком реального часу і т. ін. Зважаючи на важливість негайного оброблення переривань від таймера, йому в Windows NT надається вищий рівень пріо-

оритету – вищий від рівня будь-якого пристрою введення-виведення.

У системі черг диспетчера переривань декілька черг відводиться для обслуговування відкладених програмних переривань.

Програмні переривання, що обслуговують системні виклики від додатків, виконуються з нижчим рівнем пріоритету, що відповідає концепції продовження одного й того ж процесу, але лише в системній фазі у ході виконання системного виклику. А для програмних переривань, що виходять від модулів ядра ОС, відводиться вищий рівень запитів, що має подвійну назву «диспетчерський/dpc».

Цей рівень пріоритету називають *диспетчерським*, оскільки саме в цю чергу поміщаються програмні запити, що викликають диспетчера потоків. Часто під час оброблення високопріоритетних переривань виникає ситуація, що потребує перепланування потоків. Наприклад, обробляючи чергове переривання від таймера, потрібно перевірити, чи не вичерпаний квант, виділений для поточного потоку, або оброблення переривання від контролера диска після завершення дискової операції можуть чекати декілька потоків. У всіх таких ситуаціях в Windows NT планувальник/диспетчер викликається високорівневими процедурами ядра не безпосередньо за допомогою виклику процедури, а побічно, за допомогою програмного переривання. Це дає можливість виокремити коротку, але таку, що потребує швидкої реакції системи, процедуру обслуговування високопріоритетного переривання (наприклад, нарощування системного годинника) від менш критичної операції перепланування призначених для користувача потоків. Прямий виклик планувальника/диспетчера потоків такої можливості б не дав, і критичні запити переривання від контролерів пристроїв уведення-виведення вимушені були б чекати, поки відпрацює планувальник. При цьому планувальник, можливо, вибрав би для виконання інший потік, якби працював після процедур обслуговування апаратних переривань, оскільки він отримав би нові відомості про завершення деяких операцій введення-виведення. Поміщення виклику планувальника потоків у чергу дозволяє виконувати його лише в тих ситуаціях, коли в системі немає очікуючих апаратних запитіви переривань.

Наявність окремого рівня для планувальника/диспетчера потоків не означає того, що він завжди викликається за допомогою програмних переривань. У тих випадках, коли він викликається з

коду, що має низький рівень запиту на переривання (якщо він виконується, це означає, що високопріоритетних запитів на переривання немає), планувальник може бути викликаний швидше шляхом безпосереднього внутрішньосегментного виклику процедури. Наприклад, системному виклику, що переводить потік за власним бажанням у стан очікування, немає сенсу викликати планувальника потоків за програмним перериванням.

Диспетчерський рівень DPC (Deffered Procedure Call – виклик відкладеної процедури) означає, що на цьому рівні чекають своєї черги відкладені виклики й інших процедур ОС, а не лише планувальника/диспетчера. Процедури ОС можуть викликати одна одну і безпосередньо, але за багатошарової побудови ядра існують більш і менш пріоритетні процедури, і виклик менш пріоритетних процедур з більш пріоритетних за допомогою механізму програмних переривань дозволяє, як і у випадку з планувальником, упорядковувати в часі їх виконання, що оптимізує функціонування ОС у цілому. Прикладом процедур ОС, що працюють на високому пріоритетному рівні, є ті частини драйверів пристроїв введення-виведення, які виконують нетривалі, але критичні до часу реакції дії. Водночас є й інші частини драйверів, які виконують менш термінову, але більшого обсягу роботу. У Windows NT такі частини драйверів оформляють як процедури, що викликаються за допомогою програмних переривань рівня «диспетчерський/dpc», а саме програмне переривання виконує критична частина драйвера. Природно, існують і інші модулі ОС, що оформляються так само.

В ОС сім'ї UNIX ці частини називають відповідно верхніми половинами (top half) і нижніми половинами (bottom half) обробника переривань.

Описана програмна реалізація пріоритетного обслуговування переривань приводить до однотипної роботи ОС Windows NT на різних апаратних платформах, що спрощує логіку роботи ОС та її перенесення на нові платформи. Негативним наслідком такого централізованого підходу є деяке уповільнення оброблення переривань, оскільки замість безпосереднього передавання керування драйверу пристрою або обробнику виключень виконується виклик посередника – диспетчера переривань. В ОС сім'ї UNIX використовується схожий, але менш централізований підхід до ведення й оброблення черг переривань. Замість єдиного диспетчера переривань

його функції виконують процедури, що обслуговують кожен пріоритетний клас переривань. Проте загальний підхід до впорядковування оброблення переривань за рахунок їх багаторівневої пріоритезації та ведення системи черг застосовується майже у всіх сучасних ОС.

2.3.5. Процедури оброблення переривань і поточний процес

Важливою особливістю процедур, що виконуються за запитами переривань, є те, що вони виконують роботу, найчастіше ніяк не пов'язану з поточним процесом. Наприклад, драйвер диска може отримати керування після того, як контролер диска записав у відповідні сектори інформацію, отриману від процесу *A*, але цей момент часу, імовірно, не збігатиметься з періодом чергової ітерації виконання процесу *A* або його потоку. У найбільш типовому випадку процес *A* перебуватиме в стані очікування завершення операції введення-виведення (за синхронного режиму виконання цієї операції) і драйвер диска перерве який-небудь інший процес, наприклад процес *B*. У Windows NT процедури, що викликаються як DPC, також можуть працювати в контексті процесу, що відрізняється від того, для якого вони виконують свої функції. У деяких випадках взагалі важко однозначно визначити, для якого процесу виконує роботу той чи інший програмний модуль ОС, наприклад планувальник потоків. Тому для такого роду процедур вводяться обмеження – вони не мають права використовувати ресурси (пам'ять, відкриті файли і т. ін.), з якими працює поточний процес, або ж від імені цього процесу запрошувати виділення додаткових ресурсів. Процедури оброблення переривань працюють з ресурсами, які були виділені ним під час ініціалізації відповідного драйвера або ініціалізації самої ОС. Ці ресурси належать ОС, а не конкретному процесу. Зокрема, пам'ять виділяється драйверам із системної ділянки, тобто тієї ділянки, на яку відображуються сегменти із загальної частини віртуального адресного простору всіх процесів. Тому зазвичай процедури оброблення переривань працюють поза контекстом процесу. Оскільки всі подібні процедури є частиною ОС за дотримання цих обмежень відповідає системний програміст. Змусити свої модулі виконувати ці обмеження ОС не може.

Прикладом того, що не буває правил без винятку, є ОС Windows NT. У ній є процедури оброблення переривань, які виконуються завжди в контексті певного процесу. Це процедури, що викликаються за допомогою програмного переривання APC (Asynchronous Procedure Call – виклик асинхронної процедури). Для них у диспетчері переривань передбачено власний рівень пріоритету IRQL, вищий за рівень для звичайного коду, але нижчий за рівень DPC. Ці процедури можуть перервати поточний код і виконуватися у разі дотримання двох умов: поточний код має нижчий рівень пріоритету (тобто виконується звичайний код), поточним процесом є певний процес, описувач якого задається в запиті на переривання для певної процедури APC. Процедури APC можуть користуватися ресурсами поточного процесу, і, власне, для цього вони і були впроваджені. Основне призначення APC-процедур – переміщення даних, отриманих драйвером від якого-небудь пристрою введення-виведення, з пам'яті системної ділянки пам'яті, куди вони поміщаються після зчитування з регістрів контролера цього пристрою, в індивідуальну частину адресного простору процесу, що запитував операцію введення-виведення. Така дія постійно виконується системою введення-виведення, і для її реалізації були введені такі специфічні процедури оброблення переривань, як APC.

Диспетчеризація переривань є важливою функцією ОС, і ця функція реалізується майже у всіх мультипрограмних ОС. У загальному випадку в ОС реалізовується дворівневий механізм планування робіт. Верхній рівень планування виконується диспетчером переривань, який розподіляє процесорний час між потоком запитів, що надходять на переривання різних типів, – зовнішніх, внутрішніх і програмних. Процесорний час, що залишився, розподіляється іншим диспетчером – диспетчером потоків – на підставі дисциплін квантування й інших дисциплін.

2.3.6. Системні виклики

Системний виклик дозволяє додатку звернутися до ОС з тим, щоб вона виконала ту або іншу дію, оформлену як процедуру (або набір процедур) кодового сегменту ОС. Для прикладного програміста ОС виглядає як деяка бібліотека, що надає деякий набір корис-

них функцій, за допомогою яких можна спростити прикладну програму або виконати дії, заборонені в користувацькому режимі, наприклад обмін даними з пристроєм уведення-виведення.

Реалізація системних викликів має задовольняти такі вимоги:

– забезпечувати перемикання в привілейований режим;

– мати високу швидкість виклику процедур ОС;

– забезпечувати по можливості одноманітне звернення до системних викликів для всіх апаратних платформ, на яких працює ОС;

– допускати незначне розширення набору системних викликів;

– забезпечувати контроль з боку ОС за коректним використанням системних викликів.

Перша вимога для більшості апаратних платформ може бути виконаною лише за допомогою механізму програмних переривань. Тому вважатимемо, що останні вимоги потрібно забезпечити саме для такої реалізації системних викликів. Як це зазвичай буває, деякі з цих вимог взаємно суперечливі.

Для забезпечення високої швидкості було б корисно використовувати векторні властивості системи програмних переривань, притаманні багатьом процесорам, тобто закріпити за кожним системним викликом певне значення вектора. Додаток за такого способу виклику безпосередньо вказує в аргументі запиту значення вектора, після чого керування негайно передається необхідній процедурі ОС. Проте цей децентралізований спосіб передавання керування залежить від особливостей апаратної платформи, а також не дозволяє ОС легко модифікувати набір системних викликів і контролювати їх використання. Наприклад, у процесорі *Pentium* кількість системних викликів визначається кількістю векторів переривань, виділених для цієї мети із загального пулу з 256 елементів (частина яких використовується під апаратні переривання і оброблення виключень). Додавання нового системного виклику вимагає від системного програміста ретельного пошуку вільного елемента в таблиці переривань, якого на якомусь етапі розвитку ОС може і не виявитися.

У більшості ОС системні виклики обслуговуються за централізованою схемою, що ґрунтується на існуванні диспетчера системних викликів. За будь-якого системного виклику додаток виконує програмне переривання з певним і єдиним номером вектора. Наприклад, ОС Linux використовує для системних викликів команду

INT 80*h*. Перед виконанням програмного переривання додаток тим або іншим способом передає ОС номер системного виклику, який є індексом у таблиці адрес процедур ОС, що реалізовують системні виклики. Спосіб передавання залежить від реалізації, наприклад номер можна помістити в певний регістр загального призначення процесора або передати через стек (у цьому випадку після переривання і переходу в привілейований режим їх потрібно буде скопіювати в системний стек з користувацького режиму; ця дія в деяких процесорах автоматизована). Також деяким способом передаються аргументи системного виклику, вони можуть як поміщатися в регістри загального призначення, так і передаватися через стек або масив, що міститься в оперативній пам'яті. Масив зручний для випадків великого обсягу даних, що передаються як аргументи, при цьому в регістрі загального призначення вказується адреса цього масиву.

Диспетчер системних викликів зазвичай є простою прог-рамою, яка зберігає вміст регістрів процесора в системному стеку (оскільки в результаті програмного переривання процесор пере-ходить в привілейований режим), перевіряє, чи потрапляє запита-ний номер виклику в підтримуваний ОС діапазон (тобто чи не виходить номер за межі таблиці) і передає керування процедурі ОС, адресу якої задано в таблиці адрес системних викликів.

Процедура реалізації системного виклику витягує із системного стека аргументи і виконує задану дію. Ця дія може бути дуже простою, наприклад зчитування значення системного годинника, такою, що системний виклик оформлюється у вигляді однієї функції. Складніші системні виклики, такі як зчитування з файлу або виділення процесу додаткового сегменту пам'яті, вимагають звернення основної функції системного виклику до декількох внутрішніх процедур ядра ОС, що належать до різних підсистем, таких як підсистема введення-виведення або керування пам'яттю.

Після завершення роботи системного виклику керування повертається диспетчеру, при цьому він отримує код завершення цього виклику. Диспетчер відновлює регістри процесора, поміщає в певний регістр код повернення і виконує інструкцію повернення з переривання, яка відновлює непривілейований режим роботи процесора.

Для додатка системний виклик зовні нічим не відрізняється від виклику звичайної бібліотечної функції мови С, пов'язаної (ди-

намічно або статично) з об'єктним кодом додатка, і такої, що виконується в користувацькому режимі. Така ситуація дійсна для всіх системних викликів у бібліотеках, що надаються компілятором С. Кожна заглушка оформлена як *с*-функція, при цьому вона містить декілька асемблерних рядків, потрібних для виконання інструкції програмного переривання. Таким чином, призначена для користувача програма викликає заглушку, а та, у свою чергу, викликає процедуру ОС.

Для пришвидшення виконання деяких досить простих системних викликів, яким до того ж не необхідна робота в привілейованому режимі, необхідна робота повністю виконується бібліотечною функцією, яку несправедливо називати в такому випадку заглушкою. Точніше, така функція не є системним викликом, а є «чистою» бібліотечною функцією, що виконує всю свою роботу в призначеному для користувача режимі у віртуальному адресному просторі процесу, але прикладний програміст може про це й не знати – для нього системні виклики і бібліотечні функції виглядають однаково. Прикладний програміст має справу з набором функцій прикладного програмного інтерфейсу – API (наприклад, Win32 або POSIX), – що складається і з бібліотечних функцій, частина з яких використовується для завершення роботи системними викликами, а частина – ні.

Описаний табличний спосіб організації системних викликів прийнятий практично у всіх ОС. Він дозволяє легко модифікувати склад системних викликів, просто додавши в таблицю нову адресу і розширивши діапазон допустимих номерів викликів.

Операційна система може виконувати системні виклики в синхронному або асинхронному режимі. *Синхронний системний виклик* означає, що процес, який зробив такий виклик, припиняється (переводиться планувальником ОС у стан очікування) доти, доки системний виклик не виконає всю потрібну від нього роботу (рис. 2.12, *а*). Після цього планувальник переводить процес у стан готовності і під час чергового виконання процес гарантовано може скористатися результатами завершеного до цього часу системного виклику. Синхронні виклики називаються також *блокувальними*, оскільки процес, що викликав системну дію, блокується до його завершення.

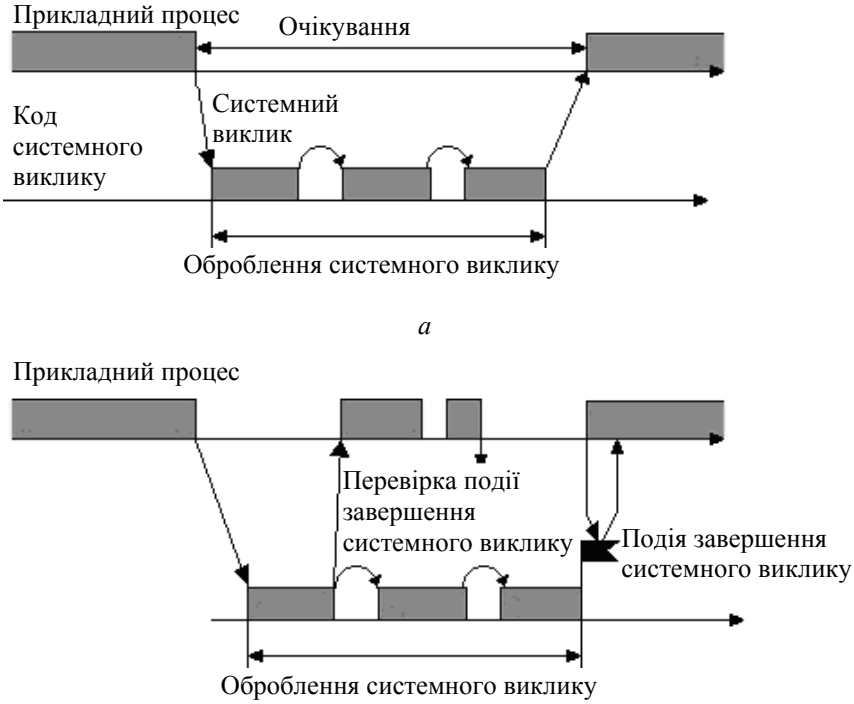


Рис. 2.12. Синхронні та асинхронні системні виклики

Асинхронний системний виклик не приводить до переведення процесу в режим очікування після виконання деяких початкових системних дій, наприклад запуску операції введення-виведення, керування повертається прикладному процесу (рис. 2.12, *б*).

Більшість системних викликів в ОС є синхронними, оскільки цей режим звільняє додаток від роботи зі з'ясування моменту появи результату виклику. Водночас у нових версіях ОС кількість асинхронних системних викликів поступово збільшується, що надає більше свободи розробникам складних застосувань. Особливо потрібні асинхронні системні виклики в ОС на основі мікроядерного підходу, оскільки в призначеному для користувача режимі працює декілька ОС, яким необхідно мати повну свободу в організації

своєї роботи, а таку свободу дає лише асинхронний режим обслуговування викликів мікроядром.

2.4. Визначення, класифікація та стани процесів. Система керування процесами

Процес – мінімальний програмний об'єкт, що має власні системні ресурси (запущена програма).

2.4.1. Класифікація процесів

За часовими характеристиками розрізняють *інтерактивні*, *пакетні процеси* та *процеси реального часу*. Час існування інтерактивного процесу визначається реакцією ЕОМ на запит обслуговування і становить секунди. Процеси реального часу мають гарантований час закінчення роботи і час реакції становить мілісекунди. Пакетні процеси запускаються один за одним і час реакції становить години і більше.

За генеалогічною ознакою розрізняють процеси *породжувальні* і *породжені*.

За результативністю розрізняють *еквівалентні*, *тотожні* та *однакові процеси*. Всі вони мають однаковий кінцевий результат, але *еквівалентні процеси* можуть реалізовуватися як на одному, так і на багатьох процесорах поодинці або за різними алгоритмами, тобто мають різні траси, які визначають порядок та час перебування процесу в різних станах. *Тотожні процеси* реалізовуються за однією і тією ж програмою, але мають різні траси. *Однакові процеси* реалізуються за однією програмою і мають однакові траси.

За часом розвитку процеси поділяють на *послідовні*, *паралельні* і *комбіновані* (останні мають точки, в яких є обидва процеси, і точки, в яких є тільки один процес).

За місцем розвитку процеси поділяють на *внутрішні* (реалізуються на центральному процесорі) і *зовнішні* (реалізуються на зовнішніх процесорах).

За належністю до ОС процеси бувають *системні* (виконують програму зі складу ОС) і *користувацькі*.

За *зв'язністю* розрізняють процеси:

– *взаємозв'язані*, які мають якийсь зв'язок (просторово-часовий, керувальний, інформаційний);

– *ізольовані* – слабкозв'язані;

– *незалежні*, які використовують сумісні ресурси, але мають власні інформаційні бази;

– *взаємодійні* – мають інформаційні зв'язки і розділяють загальні структури даних;

– *взаємозв'язані за ресурсами*;

– *конкуруючі*.

Порядок взаємозв'язку процесів визначається правилами синхронізації. Процеси можуть перебувати у співвідносності:

а) *передування* – один завжди перебуває в активному стані раніше, ніж інший;

б) *пріоритетності* – коли процес може переводитися в активний стан тільки в тому випадку, якщо в стані готовності немає процесів з вищим пріоритетом, або коли процесор вільний, або на ньому реалізується процес з меншим пріоритетом;

в) *взаємного виключення* – у процесі використовується загальний критичний ресурс, і процеси не можуть розвиватися одно-часно: якщо один з них використовує критичний ресурс, то інший перебуває в стані очікування.

2.4.2. Нитки

Багатозадачність є найважливішою властивістю ОС. Для підтримки цієї властивості ОС визначає й оформляє для себе ті внутрішні одиниці роботи, між якими і буде розділятися процесорний час та інші ресурси комп'ютера. Ці внутрішні одиниці роботи в різних ОС мають різні назви – завдання, процес, нитка. У деяких випадках сутності, що позначаються цими поняттями, принципово відрізняються одна від одної.

Операційна система підтримує таку відособленість процесів: кожний процес має власний адресний простір, кожному процесу призначаються власні ресурси – файли, вікна, семафори і т. ін. Така відособленість потрібна для того, щоб захистити один процес від іншого, оскільки вони, спільно використовуючи всі ресурси машини, конкурують один з одним. У загальному випадку процеси належать різним користувачам, що розділяють один комп'ютер, і ОС бере на себе роль арбітра «в суперечках» процесів за ресурси.

Мультипрограмування підвищує пропускну здатність системи, але окремий процес ніколи не може бути виконаний швидше, ніж якби він виконувався в однопрограмному режимі (будь-який поділ ресурсів сповільнює роботу одного з учасників за рахунок додаткових витрат часу на очікування звільнення ресурсу). Однак завдання, розв'язуване в межах одного процесу, може мати внутрішній паралелізм, що дозволяє пришвидшити його розв'язання. Наприклад, у ході виконання завдання відбувається звертання до зовнішнього пристрою, і на час цієї операції можна не блокувати цілком виконання процесу, а продовжити обчислення з іншої частини процесу. Для цього сучасні ОС пропонують використовувати порівняно новий механізм *багатониткового оброблення* (*multithreading*). При цьому вводиться нове поняття *нитка* (*thread*), а поняття *процес* значною мірою змінює зміст.

Мультипрограмування тепер реалізується на рівні ниток, і завдання, оформлене у вигляді декількох ниток у межах одного процесу, можна виконати швидше за рахунок псевдопаралельного (чи рівнобіжного в мультипроцесорній системі) виконання його окремих частин. Наприклад, якщо електронна таблиця була розроблена з урахуванням можливостей багатониткового оброблення, то користувач може запросити перерахування свого робочого аркуша й одночасно продовжувати заповнювати таблицю. Особливо ефективно можна використовувати багатонитковість для виконання розподілених додатків, наприклад, багатонитковий сервер може паралельно виконувати запити відразу декількох клієнтів.

Нитки, що належать до одного процесу, не настільки ізольовані одна від одної, як процеси в традиційній багатозадачній системі, між ними легко організовувати тісну взаємодію. Дійсно, на відміну від процесів, що належать різним, тобто, конкуруючим додаткам, усі нитки одного процесу завжди належать до одного додатка, тому програміст, що створює цей додаток, може заздалегідь продумати функціонування безлічі ниток процесу таким чином, щоб вони могли взаємодіяти, а не боротися за ресурси.

У традиційних ОС поняття *нитка* тотожно поняттю *процес*. Часто буває бажано мати кілька ниток, що розділяють єдиний адресний простір, але які виконуються квазіпаралельно, завдяки чому нитки стають подібними до процесів (за винятком поділюваного адресного простору).

Нитки іноді називають *полегшеними процесами* чи *міні-процесами*. Дійсно, нитки в багатьох випадках подібні до процесів. Кожна нитка виконується строго послідовно і має свій власний програмний лічильник та стек. Нитки, як і процеси, можуть, наприклад, породжувати нитки-нащадки, переходити зі стану в стан. Як і традиційні процеси (тобто процеси, що складаються з однієї нитки) нитки можуть перебувати в одному з таких станів: *виконання*, *очікування* і *готовність*. Поки одна нитка заблокована, інша нитка того ж процесу може виконуватися. Нитки розділяють процесор так, як це роблять процеси відповідно до різних варіантів планування.

Однак різні нитки в межах одного процесу не настільки незалежні, як окремі процеси. Всі такі нитки мають той самий адресний простір. Це означає, що вони розділяють ті самі глобальні змінні. Оскільки кожна нитка може мати доступ до кожної власної адреси, одна нитка може використовувати стек іншої нитки. Між нитками немає повного захисту, оскільки, по-перше, це неможливо, а по-друге, це не треба. Усі нитки одного процесу завжди вирішують загальне завдання одного користувача, і апарат ниток використовується тут для більш швидкого вирішення завдання його разпаралелюванням. При цьому програмісту дуже важливо мати зручні засоби організації взаємодії частин одного завдання. Крім поділу адресного простору, всі нитки розділяють також набір відкритих файлів, таймерів, сигналів і т. ін. Отже, нитки мають власні: програмний лічильник, стек, регістри, нитки-нащадки, стан. Нитки розділяють адресний простір, глобальні змінні, відкриті файли, таймери, семафори, статистичну інформацію.

Багатониткове оброблення підвищує ефективність роботи системи порівняно з багатозадачним обробленням. Наприклад, у багатозадачному середовищі Windows можна одночасно працювати з електронною таблицею і текстовим редактором. Однак, якщо користувач запитує перерахування свого робочого аркуша, електронна таблиця блокується доти, доки ця операція не завершиться, що може зайняти багато часу. У багатонитковому середовищі, якщо електронну таблицю було розроблено з урахуванням можливостей багатониткового оброблення, наданих програмісту, цієї проблеми не виникає, і користувач завжди має доступ до електронної таблиці.

2.4.3. Керування процесами

Найважливішою частиною ОС, що безпосередньо впливає на функціонування обчислювальної машини, є підсистема керування процесами. Для ОС процес являє собою одиницю роботи, заявку на споживання системних ресурсів. *Підсистема керування процесами* планує виконання процесів, тобто розподіляє процесорний час між декількома одночасно існуючими в системі процесами, а також займається створенням і знищенням процесів, забезпечує процеси необхідними системними ресурсами, підтримує взаємодію між процесами.

Щоб виконати процес, ОС повинна призначити йому ділянку оперативної пам'яті, в якій будуть розміщені коди і дані процесу, а також надати йому необхідну кількість процесорного часу. Крім того, процесу може знадобитися доступ до таких ресурсів, як файли та пристрої введення/виведення.

В інформаційні структури процесу часто включаються допоміжні дані, що характеризують історію перебування процесу в системі (наприклад, яку частку часу процес витратив на операції введення/виведення, а яку на обчислення), його поточний стан (активний чи заблокований), ступінь привілейованості процесу (значення пріоритету). Такі дані можуть враховуватися ОС під час прийняття рішення про надання ресурсів процесу.

У мультипрограмній ОС одночасно може існувати декілька процесів. Частина процесів породжується з ініціативи користувачів та їхніх додатків; такі процеси називають *користувацькими.* Інші процеси, названі *системними,* ініціюються самою ОС для виконання її функцій.

Оскільки процеси часто одночасно претендують на ті самі ресурси, то в обов'язки ОС входить підтримка черг заявок процесів на ресурси, наприклад черги до процесора, принтера та послідовного порту.

Важливим завданням ОС є захист ресурсів, виділених процесу, від інших процесів. Одними з ресурсів процесу, що захищаються більш ретельно, є ділянки оперативної пам'яті, в якій зберігаються коди та дані процесу. Сукупність ділянок оперативної пам'яті, виділених ОС процесу, називається його *адресним простором.* Кожен процес працює у своєму адресному просторі, маючи на увазі захист адресних просторів, здійснюваний ОС. Захищаються й інші типи ре-

сурсів, такі як файли, зовнішні пристрої і т. ін. Операційна система може не тільки захищати ресурси, виділені одному процесу, але й організовувати їх спільне використання, наприклад дозволяти доступ до деякої ділянки пам'яті декільком процесам.

Операційна система бере на себе також функції синхронізації процесів, які дозволяють процесу припиняти своє виконання до настання якої-небудь події в системі, наприклад завершення операції введення/виведення, здійснюваної за її запитом ОС.

В ОС немає однозначної відповідності між процесами і програмами. Один і той самий програмний файл може породити декілька паралельно виконуваних процесів, а процес у ході свого вико-нання змінити програмний файл і почати виконувати іншу програму.

2.4.4. Стани процесів

У багатозадачній системі процес може перебувати в одному з трьох основних станів.

Виконання – активний стан процесу, під час якого процес має всі необхідні ресурси і безпосередньо виконується процесором.

Очікування – пасивний стан процесу, процес заблокований, він не може виконуватися через свої внутрішні причини, чекає виконання деякої події, наприклад, завершення операції введення/виведення, отримання повідомлення від іншого процесу, звільнення якого-небудь необхідного йому ресурсу.

Готовність – також пасивний стан процесу, але в цьому ви-падку процес заблокований у зв'язку із зовнішніми щодо нього обставинами: процес має всі необхідні для нього ресурси, він готовий виконуватися, однак процесор зайнятий виконанням іншого процесу.

У ході життєвого циклу кожен процес переходить з одного стану в інший відповідно до алгоритму планування процесів, реалізованого в ОС. Типовий граф станів процесу показано на рис. 2.13.

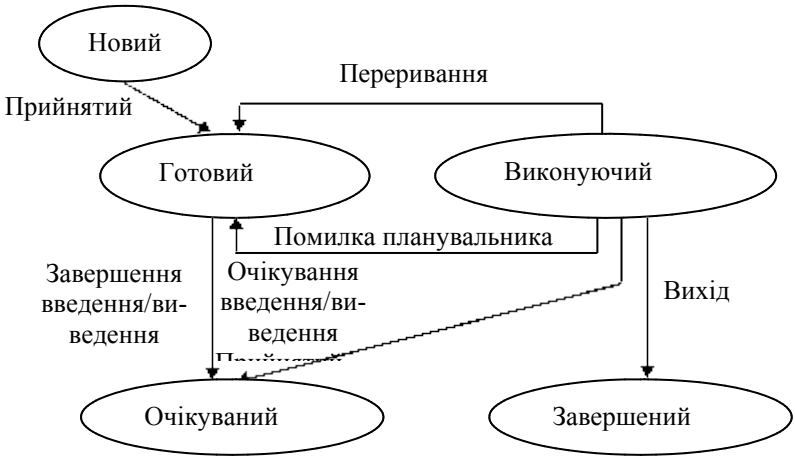


Рис.2.13. Граф станів процесу в багатозадачному середовищі

У стані виконання в однопроцесорній системі може перебувати тільки один процес, а в кожному зі станів очікування і готовності – кілька процесів; ці процеси утворять черги відповідно очікуючих і готових процесів. Життєвий цикл процесу починається зі стану готовності, коли процес готовий до виконання і чекає своєї черги. У разі активізації процес переходить у стан виконання і перебуває в ньому доти, доки він сам звільнить процесор, перейшовши в стан очікування якої-небудь події, або буде насильно витиснений із процесора, наприклад, унаслідок вичерпання відведеного процесу кванта процесорного часу. В останньому випадку процес повертається в стан готовності. У цей же стан процес переходить зі стану очікування після того, як очікувана подія відбудеться.

Таким чином, для однієї програми можуть бути створені декілька процесів у тому випадку, якщо за допомогою однієї програми в центральному процесорі виконується декілька послідовностей команд, що не збігаються. За час існування процес багато разів змінює свій стан. Розрізняють такі стани процесу:

– новий (процес тільки що створений);

– виконуваний (команди програми виконуються в центральному процесорі);

– очікуваний (процес чекає завершення деякого випадку, найчастіше операції введення/виведення);

– готовий (процес чекає звільнення центрального процесора);

– завершений (процес завершив свою роботу).

Перехід з одного стану в інший не може виконуватися довільним чином.

2.4.5. Контекст і дескриптор процесу

Протягом існування процесу його виконання може бути багаторазово перерване і продовжене. Для того, щоб відновити виконання процесу, необхідно відновити стан його операційного середовища. Стан операційного середовища відображається станом регістрів і програмного лічильника, режимом роботи процесора, вказівниками на відкриті файли, інформацією про незавершені операції введення/виведення, кодами помилок виконуваних процесом системних викликів і т.ін. Цю інформацію називають *контекстом процесу*.

Крім цього, ОС для реалізації планування процесів потрібна додаткова інформація про ідентифікатор процесу, стан процесу, дані про ступінь привілейованості процесу, місце перебування кодового сегмента й інша інформація. У деяких ОС (наприклад, в ОС UNIX) інформацію такого роду, використовувану ОС для планування процесів, називають *дескриптором процесу***.** Дескриптор процесу порівняно з контекстом містить більш оперативну інформацію, що має бути легко доступна підсистемі планування процесів. Контекст процесу містить менш актуальну інформацію і використовується ОС тільки після прийняття рішення про поновлення перерваного процесу.

Черги процесів являють собою дескриптори окремих процесів, об’єднані в списки. Таким чином, кожен дескриптор, крім всього іншого, містить принаймні один вказівник на інший дескриптор, що перебуває з ним у черзі. Така організація черг дозволяє легко їх перевпорядковувати, включати та виключати процеси, переводити процеси з одного стану в інший.

Програмний код тільки тоді почне виконуватися, коли для нього ОС буде створений процес. Створити процес – це означає:

1) створити інформаційні структури, що описують цей процес, тобто його дескриптор і контекст;

2) включити дескриптор нового процесу в чергу готових процесів;

3) завантажити кодовий сегмент процесу в оперативну пам’ять чи в ділянку свопінгу.

2.5. Планування та алгоритми планування процесів

2.5.1. Основні поняття планування процесів

Планування – забезпечення почергового доступу процесів до одного процесора.

Планувальник – відповідальна за планування частина ОС.

Алгоритм планування – використовуваний алгоритм для планування.

Ситуації, коли потрібне планування:

1) коли створюється процес;
2) коли процес завершує роботу;
3) коли процес блокується на операції введення-виведення, семафорі;
4) у разі переривання введення-виведення.

Алгоритм планування без перемикань (непріоритетний) – не потребує переривання за апаратним таймером, процес зупиняється тільки коли блокується або завершується робота.

Алгоритм планування з перемиканнями (пріоритетний) – потребує переривання за апаратним таймером, процес працює тільки у відведений період часу, після цього він припиняє роботу за таймером, щоб передати керування планувальнику.

Необхідність алгоритму планування залежить від завдань, для яких буде використовуватися ОС.

Основні три системи:

1. Системи пакетного оброблення – можуть використовувати непріоритетний і пріоритетний алгоритми (наприклад, для розрахункових програм).

2. Інтерактивні системи – можуть використовувати тільки пріоритетний алгоритм, не можна допустити, щоб один процес зайняв надовго процесор (наприклад, сервер загального доступу або ПК).

3. Системи реального часу – можуть використовувати непріоритетний і пріоритетний алгоритми (наприклад, система керування автомобілем).

Завдання алгоритмів планування:

1. *Для всіх систем*:

– справедливість – для кожного процесу відповідну частку процесорного часу;

– контроль виконання політики планування;

– баланс – підтримка зайнятості всіх частин системи (наприклад, зайнятість і процесору і обладнання введення-виведення);

2. *Системи пакетного оброблення*:

– пропускна здатність – кількість завдань за годину;

– обернений час – мінімізація часу на очікування обслуговування й оброблення завдань;

– використання процесору – щоб процесор завжди був зай-нятий.

3. *Інтерактивні системи*:

– час відгуку – швидка реакція на запити;

– домірність – виконання очікувань користувача (наприклад, користувач не готовий до довгого завантаження системи).

4. *Системи реального часу*:

– закінчення роботи до заданого терміну – запобігання втрати даних;

– передбачуваність – запобігання деградації якості в мультимедійних системах (наприклад, втрата якості звуку повинна бути меншою ніж відео).

2.5.2. Планування в системах пакетного оброблення

«Перший прийшов – першим обслужений» (FIFO – First In Fist Out). Процеси ставляться в чергу у міру надходження.

Переваги: простота; справедливість (як у черзі покупців, хто останній прийшов, той виявився наприкінці черги).

Недолік: процес, обмежений можливостями процесора може загальмувати більш швидкі процеси, обмежені обладнанням введення-виведення.

«Найкоротше завдання – перше». Цей алгоритм показано на рис. 2.14. Нижню чергу вибудовано з урахуванням цього алгоритму.

4 хв	6 хв	2 хв	4 хв	2 хв	2 хв

2 хв	2 хв	2 хв	4 хв	4 хв	6 хв

Рис. 2.14. Алгоритм «Найкоротше завдання – перше»

Переваги: зменшення оберненого часу; справедливість (як у черзі покупців).

Недолік: довгий процес, що зайняв процесор, не запустить нові більш короткі процеси, які надійшли пізніше.

Найменший час виконання, що залишився. Аналог попереднього алгоритму, але якщо надходить новий процес, його повний час виконання порівнюється з часом виконання, що залишився, поточного процесу.

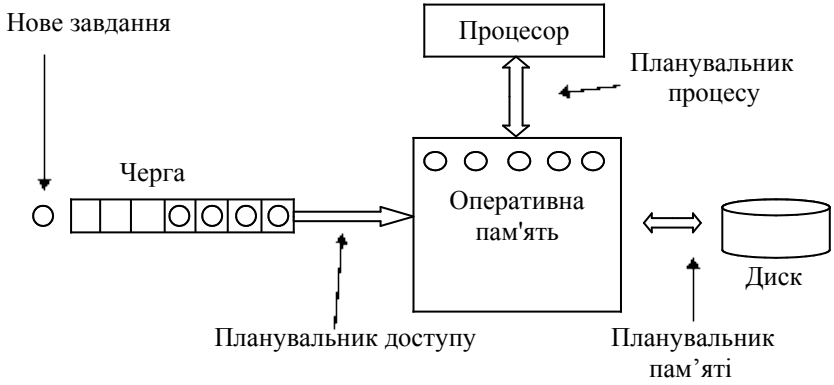


Рис. 2.15. Трирівневе планування

Трирівневе планування (рис. 2.15). Планувальник доступу вибирає завдання оптимальним чином (наприклад: процеси, обмежені процесором та введенням-виведенням).

Якщо процесів у пам’яті надто багато, планувальник пам’яті вивантажує й завантажує деякі процеси на диск. Кількість процесів, що перебувають у пам’яті, називають *ступенем багатозадачності*.

2.5.3. Планування в інтерактивних системах

Циклічне планування. Найпростіший алгоритм планування й часто використовуваний показано на рис. 2.16. Кожному процесу надається квант часу процесора. Коли квант закінчується, процес

переводиться планувальником у кінець черги. Під час блокування процесор випадає із черги.

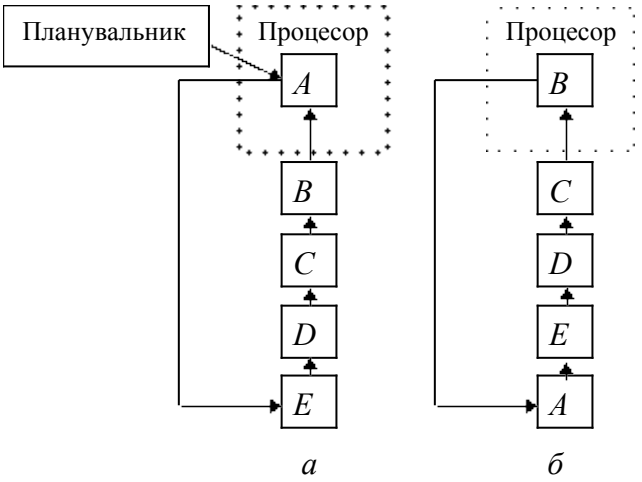


Рис. 2.16. Циклічне планування

Переваги: простота; справедливість (як в черзі покупців, кожному тільки по кілограму).

Недоліки:

– якщо часті перемикання (квант – 4 мс, а час перемикання дорівнює 1 мс), то відбувається зменшення продуктивності;

– якщо рідкісні перемикання (квант – 100 мс, а час перемикання дорівнює 1 мс), то відбувається збільшення часу відповіді на запит.

Пріоритетне планування. Кожному процесу привласнюється пріоритет, і керування передається процесу з найвищим пріоритетом. Пріоритет може бути *динамічний* і *статичний*. Динамічний пріоритет може встановлюватися так:

$$П = 1/T,$$

де T – частина використаного востаннє кванта.

Якщо використано 1/50 кванта, то пріоритет 50.

Якщо використано весь квант, то пріоритет 1.

Тобто процеси, обмежені введенням/виведенням, матимуть пріоритет над процесами, обмеженими процесором.

Часто процеси об’єднують за пріоритетами в групи та застосовують пріоритетне планування серед груп, але всередині групи використовують циклічне планування (рис. 2.17).

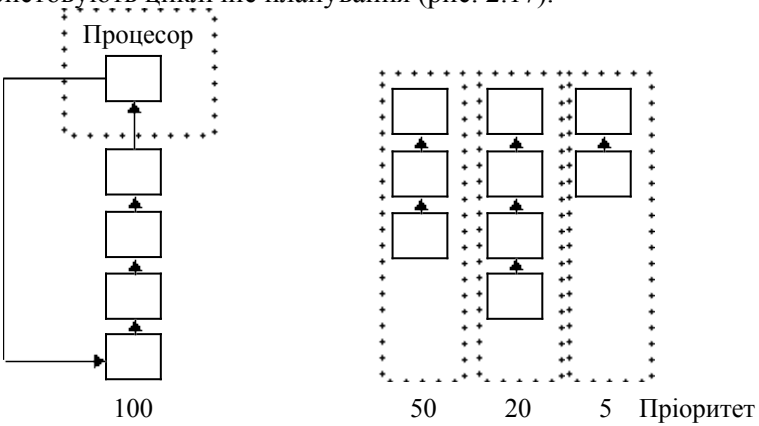


Рис. 2.17. Пріоритетне планування чотирьох груп

Методи поділу процесів на групи

Групи з різним квантом часу. Спочатку процес потрапляє в групу з найбільшим пріоритетом і найменшим квантом часу, якщо він використовує весь квант, то потрапляє в другу групу (рис. 2.18). Найдовші процеси виявляються в групі найменшого пріоритету й найбільшого кванта часу. Процес або закінчує роботу, або переходить в іншу групу

Цей метод нагадує алгоритм «Найкоротше завдання – перше».

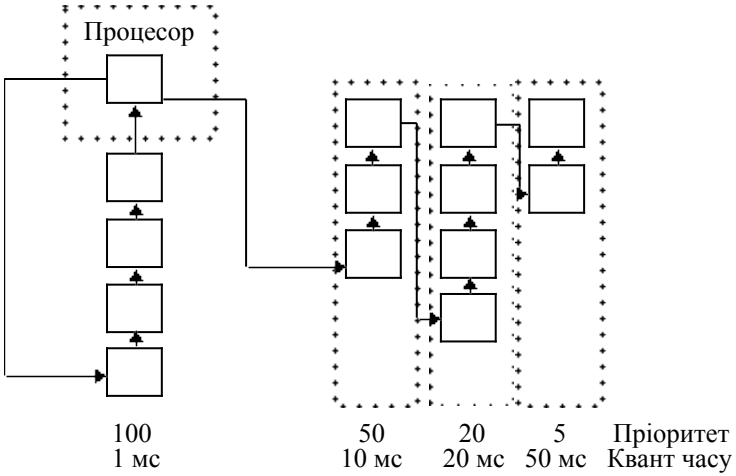


Рис. 2.18. Групи з різним квантом часу

Групи з різним призначенням процесів. Процес, що відповідає на запит, переходить у групу з найвищим пріоритетом (рис. 2.19). Такий механізм дозволяє підвищити пріоритет роботи з клієнтом.

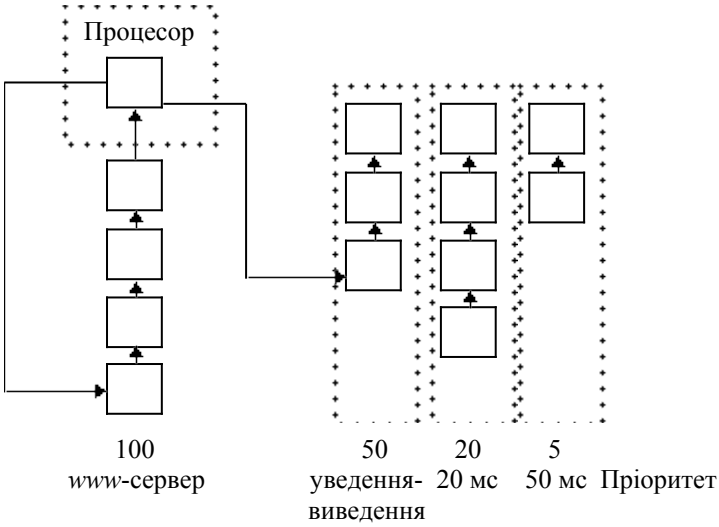


Рис. 2.19. Групи з різним призначенням процесів

Гарантоване планування. У системі з *n*-процесами, кожному процесу буде надано 1/*n* часу процесора.

Лотерейне планування. Процесам роздають «лотерейні квитки» на доступ до ресурсів. Планувальник може вибрати будь-який квиток, випадковим чином. Чим більше квитків опрацьовує процес, тим більше в нього шансів захопити ресурс.

Слушне планування. Процесорний час розподіляється серед користувачів, а не процесів. Це справедливо, якщо в одного користувача кілька процесів, а в іншого один.

2.5.4. Планування в системах реального часу

Системи реального часу поділяють на такі:

– *тверді* (тверді терміни для кожного завдання) – керування рухом;

– *гнучкі* (порушення часового графіка не бажані, але припустимі) – керування відео й аудіо.

Зовнішні події, на які система повинна реагувати:

– *періодичні* – потокове відео й аудіо;

– *неперіодичні* (непередбачені) – наприклад, сигнал про пожежу.

Щоб планувати систему реального часу, потрібно щоб виконувалася умова:

$$\sum_{i=1}^{m} \frac{T(i)}{P(i)} \leq 1,$$

де m – кількість періодичних подій; i – номер події; $P(i)$ – період надходження події; $T(i)$ – час, який витрачається на оброблення події.

Тобто переобтяжена система реального часу не є плановану.

Планування однорідних процесів. Як однорідні процеси можна розглядати відеосервер з декількома відеопотоками (декілька користувачів переглядають фільм). Оскільки всі процеси важливі, можна використовувати циклічне планування. Але через те, що кількість користувачів і розміри кадрів можуть змінюватися, для реальних систем він не підходить.

Загальне планування реального часу. Використовується модель, коли кожен процес змагається за процесор зі своїм завданням і графіком його виконання.

Планувальник повинен знати:

– частоту, з якою повинен працювати кожен процес;

– обсяг робіт, який йому належить виконати;

– найближчий термін виконання чергової порції завдання.

Розглянемо приклад з трьох процесів (рис. 2.20).

Процес *A* запускається кожні 30 мс, оброблення кадру 10 мс. Процес *B* – частота 25 кадрів, тобто кожні 40 мс, оброблення кадру 15 мс. Процес *C* – частота 20 кадрів, тобто кожні 50 мс, оброблення кадру 5 мс.

Перевіряємо, чи можна планувати ці процеси:

$$10/30+15/40+5/50 = 0{,}808 < 1.$$

Умова виконується, планувати можна.

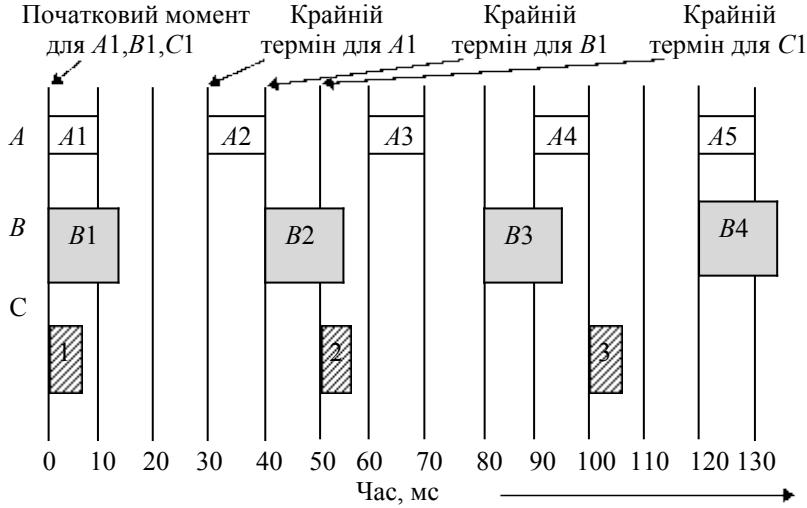


Рис. 2.20. Три періодичні процеси

Плануватимемо ці процеси *статичним* (пріоритет наперед призначається кожному процесу) і *динамічним* методами.

Статичний алгоритм планування RMS (Rate Monotonic Scheduling). Процеси мають задовольняти такі умови:

– процес повинен бути завершений за час його періоду;

– один процес не повинен залежати від іншого;

– кожному процесу потрібен однаковий процесорний час на кожному інтервалі;

– неперіодичні процеси не мають жорстких термінів;

– переривання процесу відбувається миттєво.

Пріоритет у цьому алгоритмі пропорційний частоті (рис. 2.21). Для процесу *A* він дорівнює 33 (частота кадрів), для процесу *B* – 25, для процесу *C* – 20. Процеси виконуються за пріоритетом.

Динамічний алгоритм планування EDF (Earliest Deadline First). Найбільший пріоритет виставляється процесу, у якого залишився найменший час виконання. У разі великих завантажень системи EDF має переваги.

Приклад, коли процесу *A* потрібен квант для оброблення кадру – 15 мс, показано на рис. 2.22.

Перевіряємо, чи можна планувати ці процеси:

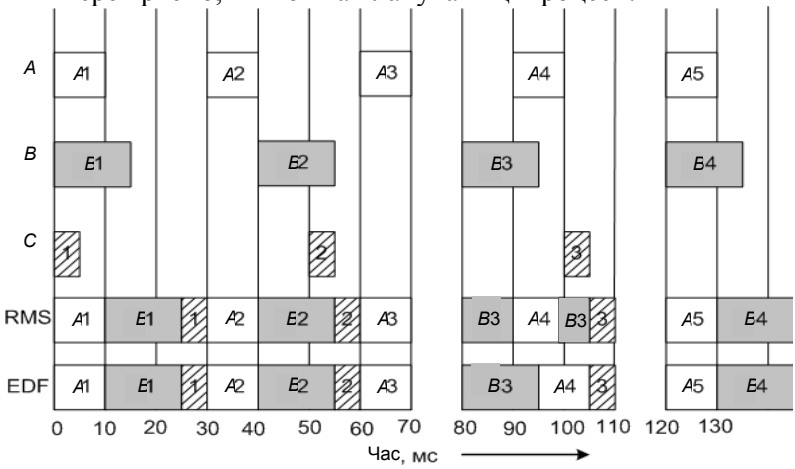


Рис. 2.21. Статичний алгоритм планування RMS

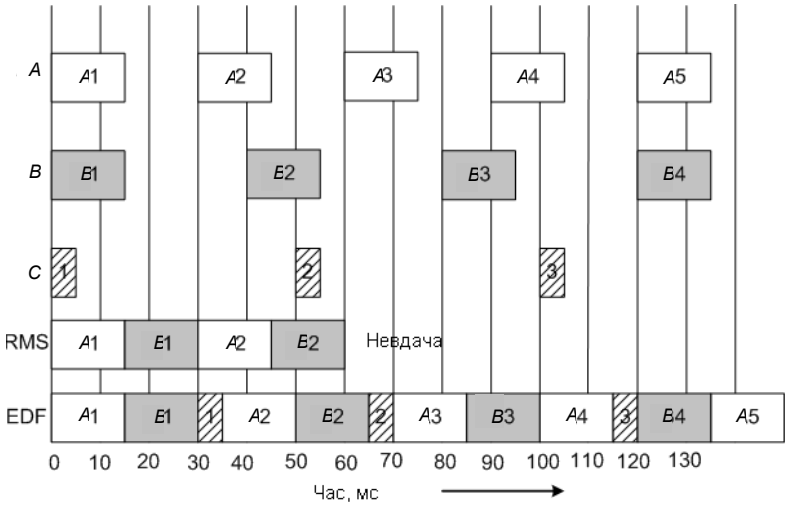


Рис. 2.22. Динамічний алгоритм планування EDF

$15/30+15/40+5/50 = 0.975 < 1.$

Завантаження системи становить 97.5%.
Алгоритм планування RMS виявився невдалим.

2.6. Взаємодія процесів. Методи і засоби синхронізації

2.6.1. Цілі та засоби синхронізації

Існує досить великий клас засобів ОС, за допомогою яких забезпечується взаємна синхронізація процесів і потоків. Потреба в синхронізації потоків виникає тільки в мультипрограмній ОС і залежить від спільного використання апаратних та інформаційних ресурсів обчислювальної системи. Синхронізація потрібна для запобігання перегонам та безвиході під час обміну даними між потоками, поділу даних, доступу до процесора і пристроїв введення-виведення.

У багатьох ОС ці засоби називаються засобами міжпроцесної взаємодії – Inter Process Communications (IPC), що відображає історичну первинність поняття *процес* відносно поняття *потік*. За-

звичай до засобів IPC належать не тільки засоби міжпроцесної синхронізації, але й засоби обміну даними.

Будь-яка взаємодія процесів або потоків залежить від їх синхронізації, яка полягає в узгодженні їх швидкостей через припинення потоку до настання деякої події й подальшої його активізації під час настання цієї події. Синхронізація лежить в основі будь-якої взаємодії потоків, незалежно від того, чи пов'язана ця взаємодія з розподілом ресурсів або з обміном даними. Наприклад, потік-одержувач повинен звертатися за даними тільки після того, як вони поміщені в буфер потоком-відправником. Якщо ж потік-одержувач звернувся до даних до моменту їх надходження в буфер, то він має бути припинений.

Синхронізація також потрібна у разі спільного використання апаратних ресурсів. Наприклад, коли активному потоку потрібен доступ до послідовного порту, а з цим портом у монопольному режимі працює інший потік, який перебуває у стані очікування, то ОС припиняє активний потік і не активізує його доти, доки потрібний йому порт не звільниться. Часто потрібна також синхронізація з подіями, які не належать до обчислювальної системи, наприклад реакція на натискання комбінації клавіш Ctrl+C.

Для синхронізації потоків прикладних програм програміст може використовувати як власні засоби та прийоми синхронізації, так і засоби ОС. Наприклад, два потоки одного прикладного процесу можуть координувати свою роботу за допомогою доступної для них обох глобальної логічної змінної, яка набуває значення одиниці при здійсненні деякої події, наприклад вироблення одним потоком даних, потрібних для продовження роботи іншого. Однак у багатьох випадках більш ефективними або навіть єдино можливими є засоби синхронізації, що надаються ОС у формі системних викликів. Так, потоки, що належать різним процесам, не мають можливості втручатися будь-яким чином у роботу один одного. Без посередництва ОС вони не можуть призупинити один одного або сповістити про подію, що відбулася. Засоби синхронізації використовуються ОС не тільки для синхронізації прикладних процесів, але й для її внутрішніх потреб. Зазвичай розробники ОС надають у розпорядження прикладних і системних програмістів широкий спектр засобів синхронізації. Ці засоби можуть утворювати ієрархію, коли на основі більш простих засобів будуються більш складні, а також бути функціонально спеціалізованими, наприклад засоби для синхронізації потоків одно-

го процесу, засоби для синхронізації потоків різних процесів під час обміну даними і т. ін. Часто функціональні можливості різних системних викликів синхронізації перекриваються, тому для вирішення одного завдання програміст може скористатися кількома викликами залежно від особистих переваг.

2.6.2. Взаємодія між процесами

Ситуації, коли процесам доводиться взаємодіяти;

– передавання інформації від одного процесу до іншого;

– контроль над діяльністю процесів (наприклад, коли вони змагаються за один ресурс);

– узгодження дій процесів (наприклад, коли один процес доставляє дані, а інший їх виводить на друк. Якщо узгодженості не буде, то другий процес може почати друк раніше, ніж надійдуть дані).

Два останні випадки стосуються і потоків. У першому випадку потоки не мають проблем, тому що вони використовують загальний адресний простір.

Передавання інформації від одного процесу до іншого. Інформація може передаватися кількома способами:

– *колективна пам'ять*;

– *канали* (труби) (рис. 2.23) – це псевдофайл, який один процес записує, а інший зчитує.

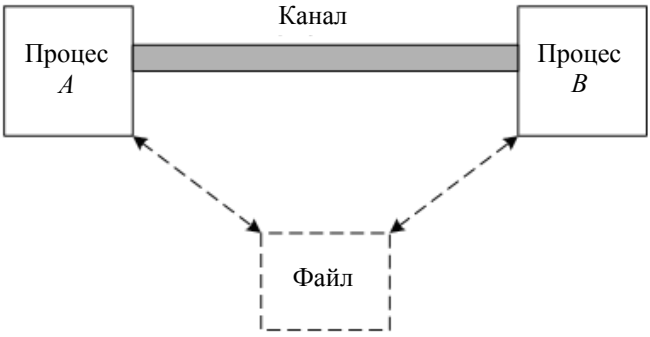


Рис. 2.23. Схема для каналу

– *сокети* (рис. 2.24) – підтримувані ядром механізми, що приховують особливості середовища і дозволяють взаємодіяти процесам, як на одному комп'ютері, так і в мережі.

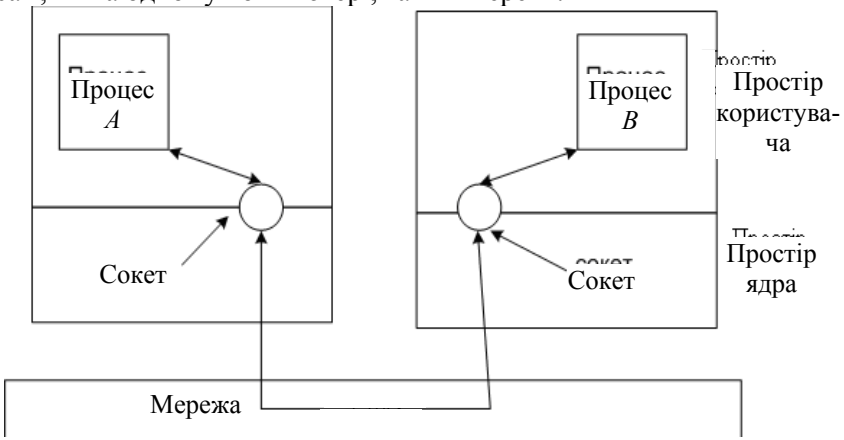


Рис. 2.24. Схема для сокетів

– поштові скриньки (тільки у Windows) – однонапрямлені з можливістю широкомовної розсилки;

– виклик віддаленої процедури – процес *A* може викликати процедуру в процесі *B* і отримувати назад дані.

Стан змагання (гонки). *Стан змагання* – ситуація, коли декілька процесів зчитують або записують дані (у пам'ять або у файл) одночасно.

Розглянемо приклад (рис. 2.25), коли два процеси намагаються роздрукувати файл. Для цього вони повинні помістити ім'я файлу в спулер друку, у вільний сегмент:

in – змінна вказує на наступний вільний сегмент;

out – змінна вказує на наступне ім'я файлу для друку.

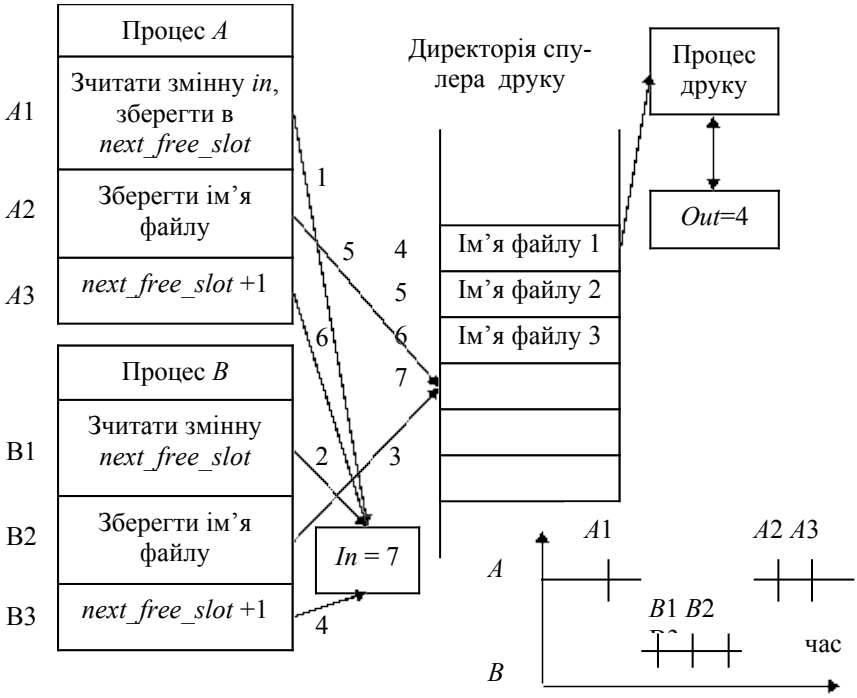


Рис. 2.25. Приклад змагання

Алгоритм змагання:

1. Процес *A* зчитує змінну *in* (дорівнює 7) і зберігає її у своїй змінній *next_free_slot*.

2. Відбувається переривання за таймером і процесор переключається на процес *B*.

3. Процес *B* зчитує змінну *in* (дорівнює 7) і зберігає її у своїй змінній *next_free_slot*.

4. Процес *B* зберігає ім'я файлу в сегменті 7.

5. Процес *B* збільшує змінну *next_free_slot* на одиницю (*next_free_slot* + 1) і замінює значення *in* на 8.

6. Керування переходить до процесу *A*, і він продовжується з того місця на якому зупинився.

7. Процес *A* зберігає ім'я файлу в сегменті 7, стираючи назву файлу процесу *B*.

8. Процес *A* збільшує змінну *next_free_slot* на одиницю (*next_free_slot* + 1) і замінює значення *in* на 8.

Як видно з алгоритму файл процесу *B* не буде надрукований.

Критичні області. *Критична область* – частина програми, в якій є звернення до спільно використовуваних даних.

Умови для уникнення змагання та ефективної роботи процесів:

1. Два процеси не повинні одночасно бути в критичних областях.

2. Процес, який перебуває поза критичною областю, не може блокувати інші процеси.

3. Неможлива ситуація, коли процес постійно чекає (зависає) потрапляння в критичну область.

Приклад взаємного виключення показано на рис. 2.26.

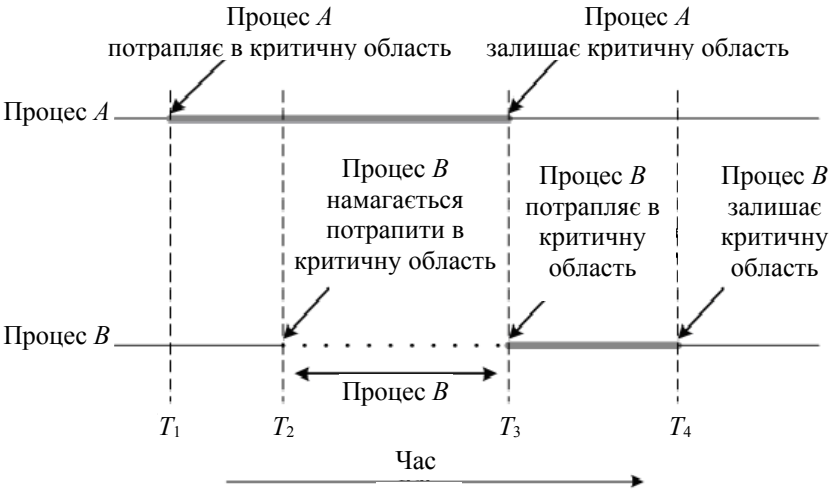


Рис. 2.26. Взаємне виключення з використанням критичних областей

Взаємне виключення з активним очікуванням. Розглянемо методи взаємного виключення.

Заборона переривань. Полягає в забороні всіх переривань при входженні процесу до критичної області.

Недолік цього методу: якщо відбудеться збій процесу, то він не зможе зняти заборону на переривання.

Змінні блокування (названі також дедлоками (*deadlocks*), клінчами (*clinch*), або безвиходями.)

Уводиться поняття змінної блокування *i*, тобто якщо значення цієї змінної одне, наприклад 1, то ресурс зайнятий іншим проце-сом, і другий процес переходить у режим очікування (блокується) доти, доки змінна не набуде значення 0 (рис. 2.27).

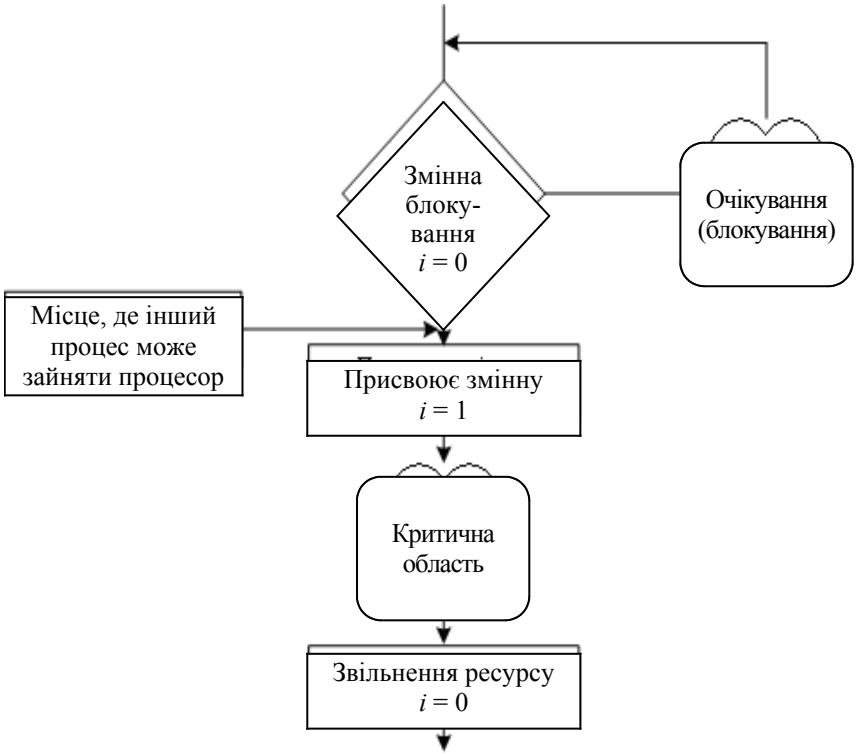


Рис. 2.27. Метод блокувальних змінних

Після того, як перший процес зчитує 0, другий може зайняти процесор і теж зчитати 0. Заблокований процес перебуває в режимі *активного очікування*, постійно перевіряючи, чи змінилася змінна блокування.

Чітке чергування. У цій моделі процеси можуть виконуватися чітко за чергою, використовуючи змінну (рис. 2.28).

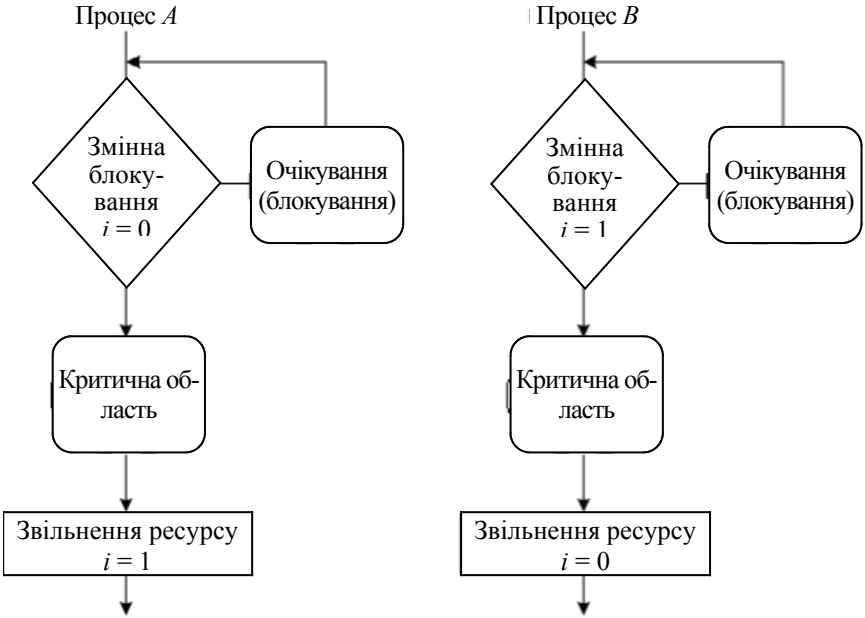


Рис.2.28. Чітке чергування

Недоліки моделі: заблокований процес постійно перебуває в циклі, перевіряючи, чи не змінилася змінна; чи суперечить він умові, коли процес, що перебуває поза критичною областю, може блокувати інші процеси.

Існують ще алгоритми з активним очікуванням (алгоритм Петерсона, команда *TSL*), але всі вони мають загальний недолік – витрачається безцільно час процесора на цикли перевірки змінювання змінної.

Примітиви взаємодії процесів. Уводиться поняття двох примітивів:

– *sleep* – системний запит, унаслідок якого викликаний процес блокується, доки його не запустить інший процес.

– *wakeup* – системний запит, унаслідок якого блокований процес буде запущений.

Основна перевага – відсутність активного очікування.

Недолік: якщо спулер порожній, то *wakeup* спрацьовує даремно (рис. 2.29).

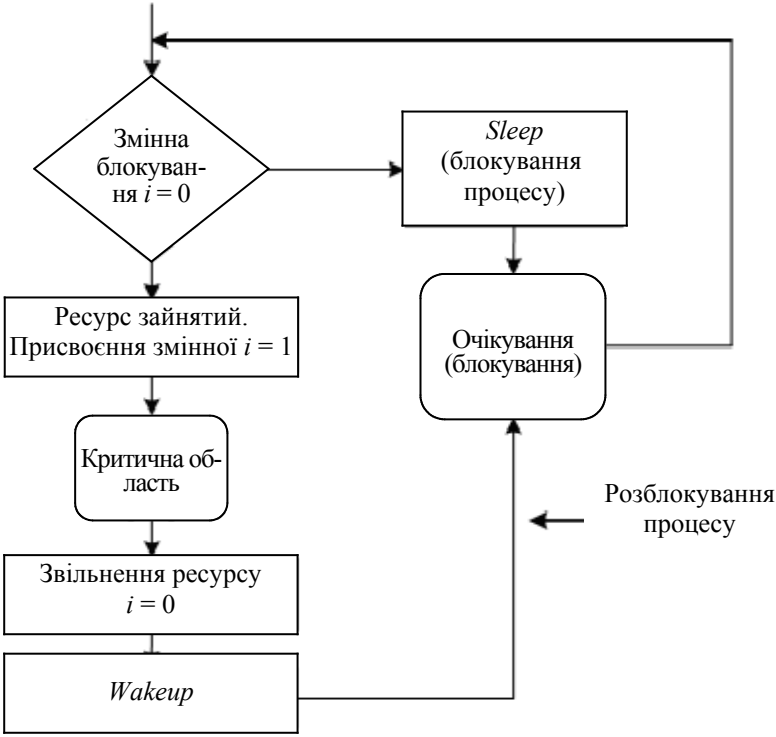


Рис. 2.29. Застосування примітивів

Проблема переповненого буфера (проблема виробника і споживача). Розглянемо два процеси, які спільно використовують буфер обмеженого розміру, один процес записує в буфер, другий зчитує дані.

Щоб перший процес не записував, коли буфер повний, а другий не зчитував, коли він порожній, вводиться змінна *count* для підрахунку кількості елементів у буфері (рис. 2.30). У цій ситуації обидва процеси можуть потрапити в стан очікування, якщо зникне сигнал активації.

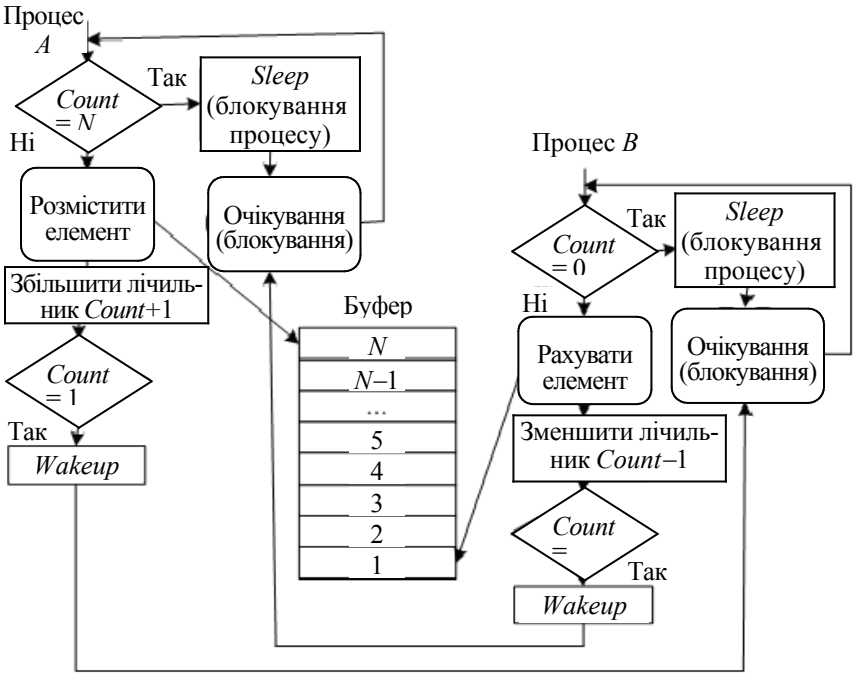


Рис. 2.30. Проблема переповненого буфера

Алгоритм такої ситуації:

1. Процес *B* зчитав *count* = 0 (заблокуватися він ще не встиг).

2. Планувальник передав керування процесу *A*.

3. Процес *A* виконав всі дії аж до *wakeup*, намагаючись розблокувати процес *B* (але він не заблокований, *wakeup* спрацьовує даремно).

4. Планувальник передав керування процесу *B*, і він заблокувався й більше сигналу на розблокування не отримає.

5. Процес *A* зрештою заповнить буфер і заблокується, але сигналу на розблокування не отримає.

Семафори. *Семафори* – змінні для підрахунку сигналів запуску, збережених для подальшого використання. Були запропоновані дві операції *down* та *up* (аналоги *sleep* і *wakeup*).

Перед тим, як заблокувати процес, *down* перевіряє семафор; якщо він дорівнює нулю, то він блокує процес, якщо ні, то процес знову стає активним і зменшує семафор на одиницю:

– *up* – збільшує значення семафора на 1 або розблоковує процес, який перебував в очікуванні;

– *down* – зменшує значення семафора на 1 або блокує процес, якщо семафор дорівнює 0.

Down та *up* виконуються як *елементарна дія*, тобто процес не може бути блокований під час виконання цих операцій. Отже, в ОС має бути заборона на всі переривання та переведення процесу в режим очікування.

Вирішення проблеми переповненого буфера за допомогою семафора. Застосуємо три семафори (рис. 2.31):

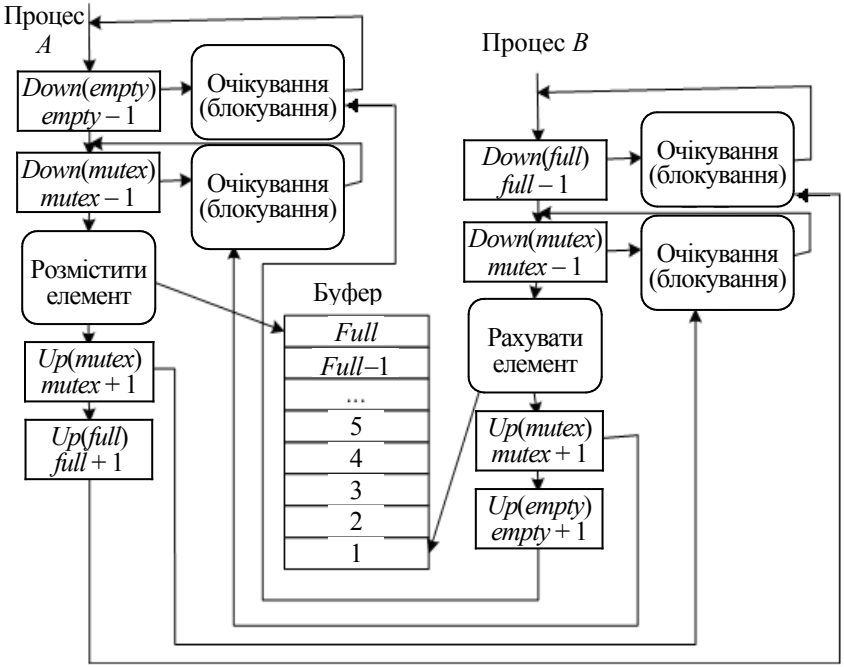


Рис. 2.31. Вирішення проблеми переповненого буфера за допомогою семафора

– *full* – підрахунок заповнених сегментів (на початку дорівнює 0);

– *empty* – підрахунок порожніх сегментів (на початку дорівнює кількості сегментів);

– *mutex* – для запобігання одночасному доступу до буфера двох процесів (на початку дорівнює 1).

М'ютекс – спрощена версія семафора; він керує доступом до ресурсу (показує, блокований він чи ні).

Застосування семафорів для пристроїв введення-виведення. Для пристроїв введення-виведення семафор виставляється рівним нулю. Після запуску керувального процесу виконується *down*, і тому семафор дорівнює нулю – процес блокується. Коли потрібно активізувати процес керування, виконується *up*.

2.7. Поняття ресурсу. Класифікація ресурсів системи за різними ознаками

2.7.1. Поняття ресурсу

Ресурс – будь-який споживаний об'єкт.

За запасами ресурси поділяють на *вичерпні* і *невичерпні*.

Споживачі ресурсів – процеси.

Ресурс – засіб обчислювальної системи, який виділяється для процесу на певний інтервал часу.

Процесор – будь-який пристрій у складі ЕОМ, здатен автоматично виконувати прийнятні для нього дії (процесори, канали та пристрої, що працюють з каналами).

Реалізація системи керування процесами у складі ОС ставить певні вимоги до властивостей процесорів.

2.7.2. Класифікація ресурсів

Ресурси класифікують за такими ознаками:

– *реальності* – *фізичні* та *віртуальні*. Віртуальні ресурси тільки за окремими властивостями подібні до фізичних ресурсів;

– *можливістю розширення властивостей* – *еластичні* та *жорсткі* (що не допускають віртуалізації);

– *ступенем активності* – *пасивні* та *активні* (можуть виконувати дії щодо інших ресурсів);

– *часом існування* – *постійні* (доступні протягом усього часу процесу: і до, і після його роботи) і *тимчасові*;

– *ступенем значущості – основні* та *другорядні* (допускають альтернативний розвиток процесу за їх відсутності);

– *функціональною надмірністю при розподілі – дорогий*, але надається швидко, і *дешевий*, але надається з очікуванням;

– *структурою – прості* (не містять складових елементів) і *складні*. (Вони розрізняються кількістю станів: простий може перебувати тільки в двох станах – доступному або зайнятому);

– *характером використання розподілюваних ресурсів – потрібні* і *відтворні* ресурси (допускають багатократне використання та звільнення);

– *характером використання – послідовно* і *паралельно* використовувані;

– *формою реалізації – жорсткі* (не допускають копіювання) і *м'які* (допускають копіювання і підрозділяються на програмні та інформаційні ресурси).

Дисципліна розподілу ресурсу визначає порядок використання багатьма процесами того або іншого ресурсу, який в кожен момент часу може обслуговувати тільки один процес.

2.7.3. Основні види ресурсів обчислювальної системи

Основні види ресурсів обчислювальної системи та способи їх поділу. Одним з найважливіших ресурсів є *процесорний час*. Процесорний час розподіляється попеременно (паралельно).

Іншим видом ресурсів обчислювальної системи вважають *пам'ять*. Оперативна пам'ять може розподіляться і одночасно і попеременно. Пам'ять – дуже важливий вид ресурсу. Річ у тім, що в кожен конкретний момент часу процесор під час виконання обчислень звертається до обмеженої кількості комірок оперативної пам'яті. З цього погляду пам'ять потрібно розділяти для більшої кількості паралельно виконуваних процесів. Чим більше пам'яті можна виділити, тим кращі будуть умови для виконання процесу. Тому проблема ефективного розподілення оперативної пам'яті між паралельно виконуваними обчислювальними процесами є однією із найбільш актуальних. Що стосується *зовнішньої пам'яті* (наприклад, пам'яті на магнітних дисках), то власне пам'ять і доступ до неї вважаються різними видами ресурсу. Але для повної роботи із зовнішньою пам'яттю необхідно мати обидва види цих ресурсів.

Власне зовнішня пам'ять може розподілятися одночасно, а доступ до неї – попеременно. Зовнішні пристрої можуть розподілятися паралельно, якщо використовуються механізми прямого доступу. Якщо ж пристрій працює з послідовним доступом, то він не може вважатися розподіленим ресурсом. Простими прикладами зовнішніх пристроїв, що не можуть бути розподілюваними, є принтер і нагромаджувач на магнітній стрічці.

Дуже важливим видом ресурсів є *програмні модулі*. Системні програмні модулі розглядаються як програмні ресурси і можуть бути розподілені між виконуваними процесами Як відомо, програмні модулі бувають *одноразово* і *багаторазово використовуваними*.

Одноразово використовуваними називають такі програмні модулі, які можна правильно виконати тільки один раз. Це означає, що в процесі виконання вони можуть пошкодити частину коду, або вихідні дані, від яких залежить хід обчислень. Очевидно, що однократно використовувані програмні модулі є неподільним ресурсом.

Повторно використовувані програмні модулі, у свою чергу, можуть бути *непривілейованими*, *привілейованими* і *рентабельними*.

Привілейовані програмні модулі працюють у привілейованому режимі, тобто за вимкненої системи переривань (переривання закриті), тому ніякі зовнішні події не можуть порушити природний порядок обчислень. У результаті програмний модуль виконується до кінця, після чого він може бути знову викликаний для виконання іншого завдання.

Непривілейовані програмні модулі – це звичайні програмні модулі, що можуть бути перервані під час роботи. Отже, у загальному випадку їх не можна вважати розподілюваними, оскільки якщо після переривання виконання такого модуля в межах одного обчислювального процесу запустити його ще раз за вимогою іншого обчислювального процесу, тоді проміжні результати для перерваних обчислень можуть бути загублені.

Рентабельні програмні модулі допускають повторне багаторазове переривання виконання і повторний їх запуск за звертанням з інших завдань (обчислювальних процесів). Для цього рентабельні програмні модулі мають бути створені таким чином, щоб забезпечувалося збереження проміжних обчислень для обчислень, що перериваються, і повернення до них, коли обчислювальний процес

відновлюється з перерваної раніше точки. Це може бути реалізовано двома способами: за допомогою статичних і динамічних методів виділення пам'яті для збережуваного значення. Основний, найбільш часто використовуваний динамічний спосіб, – виділення пам'яті для збереження всіх проміжних результатів обчислення, що належать до рентабельного програмного модуля.

Статичний спосіб виділення пам'яті полягає в такому: заздалегідь для фіксованої кількості обчислювальних процесів резервуються ділянки пам'яті, у яких розміщуватимуться змінні рентабельних програмних модулів: для кожного процесу є власна ділянка пам'яті. Найчастіше такі процеси є процесами введення/виведення, тобто ідеться про рентабельні драйвери.

Крім рентабельних програмних модулів є ще *повторно-вхідні*. Цим терміном називають програмні модулі, що теж до-пускають їх багаторазове використання, але на відміну від рента-бельних їх не можна переривати. Повторно-вхідні програмні моду-лі складаються з привілейованих секцій і повторне звернення до них можливе тільки після завершення якої-небудь з таких секцій. У повторно-вхідних програмних модулях чітко визначені всі припус-тимі (можливі) точки – входи.

2.8. Керування оперативною пам'яттю

Пам'ять комп'ютера, або основна пам'ять, або оперативна пам'ять відіграє особливу роль. Це зумовлено тим, що програма може виконуватися лише в тому разі, якщо вона міститься в пам'яті.

Пам'ять розподіляється між користувацькими і системними програмами ОС.

2.8.1. Функції операційної системи з керування пам'яттю

Основні функції ОС з керування пам'яттю:

– облік вільної та зайнятої пам'яті;

– виділення пам'яті процесам та її звільнення;

– витіснення кодів і даних процесів на диск, коли пам'яті не вистачає, і повернення на місце;

– налаштування адрес на конкретну ділянку фізичної пам'яті;

– дефрагментація;
– захист пам’яті.

2.8.2. Типи адрес

Для ідентифікації команд програми і даних використовуються адреси.

Адреси поділяють на такі:

– *символьні імена*. Привласнює програміст (наприклад, мітки);

– *віртуальні адреси*. Формує транслятор. Початкова адреса дорівнює нулю;

– *фізичні адреси* – номери елементів пам’яті, де насправді будуть розміщені команди та дані.

Сукупність віртуальних адрес складає віртуальний адресний простір (ВАП). Він визначається розрядністю комп’ютера. Для 32-розрядних комп’ютерів – це максимум FFFFFFFF, що становить 4 Гбайт.

Є два основні типи подання віртуальних адрес:

– *лінійне*, за якого адреса спочатку завжди дорівнює нулю, а адреса – ціле число;

– *поділ на сегменти*, за якого адреса – це пара чисел (*n*,*m*), де *n* – номер сегмента, *m* – зсув.

Максимально можливий ВАП – визначається розрядністю процесора. Для 32-розрядного *Intel Pentium* ця величина становить 4 Гбайт.

Призначений ВАП дійсно необхідний процесу для роботи. Його називають також *образом процесу*. Призначений ВАП може перевищувати фізичну ємність пам’яті. На цьому заснований механізм віртуальної пам’яті.

Віртуальний адресний простір і віртуальна пам’ять – це різні механізми для ОС. Операційна система може підтримувати ВАП, але механізму віртуальної пам’яті може при цьому не бути, наприклад, у разі перевищення фізичної пам’яті над ВАП будь-якого процесу.

2.8.3. Алгоритми розподілення пам’яті без використання зовнішньої пам’яті:

– розподілення пам’яті на фіксовані розділи;

– розподілення пам’яті на динамічні розділи.

У разі використання цього алгоритму пам’ять у початковий момент часу вважається вільною (за винятком пам’яті, відведеної для ОС). Кожному процесу відводиться вся необхідна пам’ять. Якщо її не вистачає, то процес не створюється. У довільний момент часу пам’ять є випадковою послідовністю зайнятих і вільних ділянок.

2.8.4. Алгоритми розподілу пам’яті з використанням зовнішньої пам’яті

Для повного завантаження процесора можуть знадобитися інколи сотні інтерактивних завдань. Всі вони мають бути розміще-ні в пам’яті, велика частина яких перебуває в стані очікування. Логічно було б на час очікування, в разі браку фізичної пам’яті, витісняти їх на диск, а коли необхідно, повертати в пам’ять. Така підміна (віртуалізація) оперативної пам’яті дисковою пам’яттю істотно підвищує рівень мультипрограмування. Важливо, що всі дії з переміщення відбуваються автоматично, без участі програміста.

Для віртуалізації застосовують два основні підходи:

– *свопінг* – образ процесу вивантажується на диск і повертається в пам’ять цілком (часто називають підкачуванням);

– *віртуальна пам’ять* – образ процесу вивантажується на диск і повертається в пам’ять частинами (сегментами, сторінками...).

Реалізацію віртуальної пам’яті, подано трьома класами: *сторінковим*, *сегментним*, *сегментно-сторінковим* розподілами.

Сторінковий розподіл. За сторінкового розподілу віртуальна пам’ять ділиться на частини однакового та фіксованого для системи розміру – *віртуальні сторінки*. Вся оперативна пам’ять також ділиться на частини такого ж розміру – *фізичні сторінки*. Розмір сторінки вибирається рівним мірі двійки: 512, 1024, 4096 і так далі.

Адреса сторінки входить в контекст процесу.

Таблиця сторінок складається з дескрипторів. Кожен дескриптор включає:

– номер фізичної таблиці;

– ознака наявності в оперативній пам’яті (формується апаратно);

– ознака модифікації (формується апаратно);

– ознака звернення (формується апаратно).

Віртуальна адреса, подана парою (*P*, *SV*), перетвориться в (*N*, *SF*).

Обсяг сторінки дорівнює мірі 2*k*, тоді зсув (*S*) можна отримати відділенням *k* розрядів.

Наприклад, якщо розмір сторінки становить 1 кбайт (2^{10}), то 50718 = 101 000 111 0012, 108 = 28 – номер сторінки.

Апаратно, з регістра витягується адреса таблиці сторінок. На підставі номера сторінки *P* і довжини запису *L* визначається адреса дескриптора ($A = AT + P \cdot L$).

З таблиці витягується номер фізичної сторінки *N*. До номера *N* приєднується зсув *S*.

Розмір сторінок, (часто 4096) впливає на розмір таблиць, а це, у свою чергу, позначається на продуктивності. Для усунення цього недоліку ВАП може ділитися на розділи, а в кожному розділі формується своя таблиця сторінок. Цей варіант пришвидшує пошук.

Сегментний розподіл. За сторінкового розподілу ВАП ділиться на рівні частини механічно без урахування важливого значення даних. На одній сторінці можуть одночасно виявитися код програми і вихідні дані. Такий підхід не дозволяє забезпечити роздільне оброблення, наприклад захист, спільний доступ і т. ін.

Розбиття адресного простору на «осмислені» частини усуває ці недоліки і називається *сегментним розподілом*. Приклади сегментів: код програми, масив вихідних даних та ін.

На етапі створення процесу ОС будує таблицю сегментів процесу, аналогічну таблиці сторінок.

Недоліки сегментного розподілу:

– використання операції складання під час формування фізичної адреси призводить до зниження продуктивності;

– надмірність. Оскільки сегмент у загальному випадку може бути більшим ніж сторінка, то і одиниця обміну між оперативною пам’яттю і диском більша, що приводить до уповільнення роботи.

Сегментно-сторінковий розподіл. Цей метод є комбінацією сторінкового і сегментного механізмів керування пам’яттю і спря-

мований на реалізацію переваг обох підходів. Віртуальна пам'ять ділиться на сегменти, а кожен сегмент – на сторінки. Усі сучасні ОС використовують саме такий спосіб організації.

2.9. Керування файлами та зовнішніми пристроями

2.9.1 Принципи апаратури введення-виведення

Пристрої введення-виведення. Пристрої поділяють на дві категорії (деякі не потрапляють в жодну з них):

– *блокові пристрої* – інформація зчитується і записується по блоках, блоки мають свою адресу (диски);

– *символьні пристрої* – інформація зчитується і записується посимвольно (принтер, мережеві карти, миші).

Контролери пристроїв. Пристрої введення-виведення зазвичай складаються з двох частин:

– *механічної* (не треба розуміти дослівно) – диска, принтера, монітора;

– *електронної* – контролера або адаптера.

Якщо інтерфейс між контролером та пристроєм стандартизова-ний (ANSI, IEEE або ISO), то незалежні виробники можуть випус-кати сумісні як контролери, так і пристрої, наприклад, диски IDE або SCSI.

Операційна система зазвичай працює не з пристроєм, а з контролером. *Контролер* виконує прості функції, наприклад, під час зчитування з диска, перетворює потік бітів у блоки, що складаються з байтів і здійснює контроль та виправлення помилок: перевіряє контрольну суму, якщо вона збігається із зазначеною в заголовку сектора, то блок зчитаний без помилок, якщо ні, то зчитується наново.

Контролер, відображуваний на адресний простір пам'яті введення-виведення. Кожен контролер має кілька регістрів, які використовуються для взаємодії з центральним процесором. За допомогою цих регістрів ОС керує (зчитує, пише, включає і т. ін.) і визначає стан (готовність) пристрою.

Багато пристроїв містять буфер даних (наприклад, відеопам'ять).

Реалізація доступу до керувальних регістрів і буфера (рис. 2.32):

Номер порту введення-виведення – призначається кожному керувальному регістру 8- або 16-розрядне ціле число. Адресні простори оперативної пам'яті та пристрої введення-виведення в цій схемі не перетинаються. Недоліки:

– для зчитування і запису застосовуються спеціальні команди, наприклад, *IN* і *OUT*;

– необхідний спеціальний механізм захисту від процесів;

– необхідно спочатку зчитати регістр пристрою в регістр процесора.

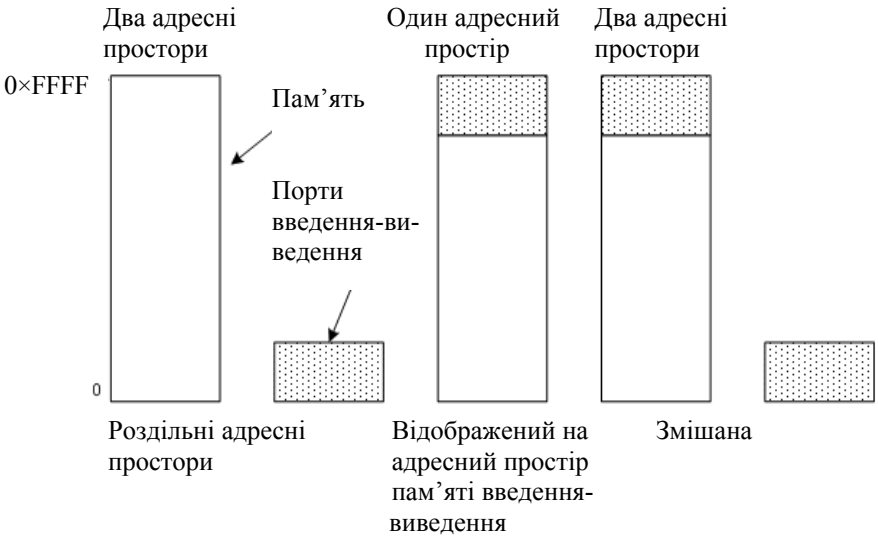


Рис. 2.32. Способи реалізації доступу до керувальних регістрів і буфер

Контролер, відображуваний на адресний простір пам'яті введення-виведення – регістри відображаються на адресний простір пам'яті. Недоліки:

– під час кешування пам'яті можуть кешуватися і регістри пристроїв;

– усі пристрої повинні перевіряти всі звернення до пам'яті, щоб визначити, на які їм реагувати. На одній загальній шині це реалізується легко, але на декількох будуть виникати труднощі.

Змішана реалізація – використовується в *x*86 і Pentium, від 0 до 64Кбайт відводиться для портів, від 640 до 1Мбайт зарезервується для буферів даних.

Прямий доступ до пам'яті. Прямий доступ до пам'яті (DMA – Direct Memory Access) реалізується за допомогою DMA-контролера (рис. 2.33).

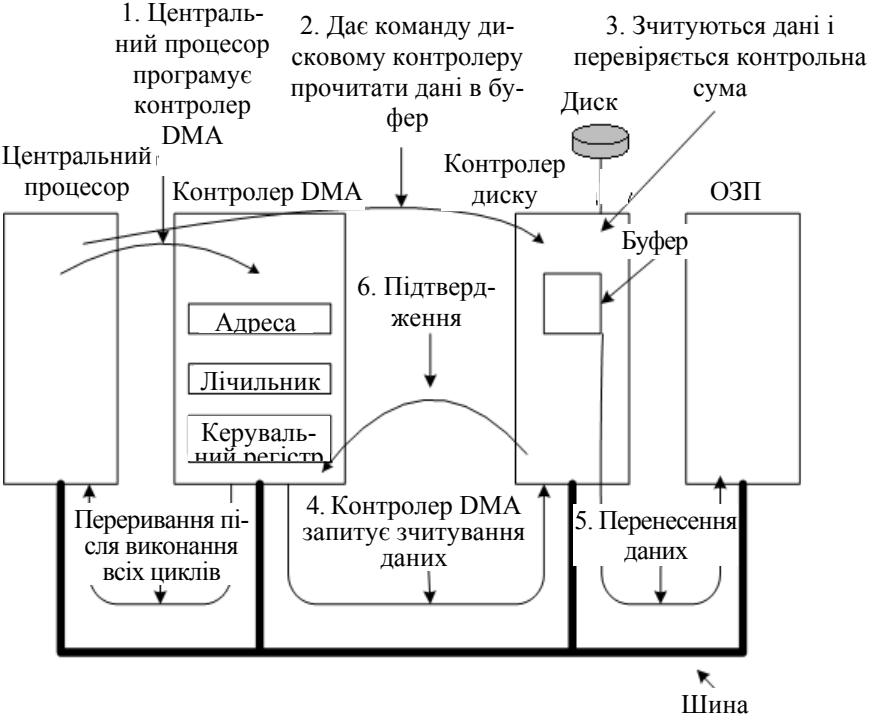


Рис. 2.33. Робота DMA-контролера: ОЗП – оперативний запам'ятовувальний пристрій

Контролер містить декілька регістрів:

1. Регістр адреси пам'яті.
2. Лічильник байтів.
3. Керувальні регістри, які можуть містити:
   – порт введення-виведення;
   – регістри зчитування або запис;
   – одиниці перенесення (побайтно або послівно).

Операції, виконувані без контролера:

1. Процесор дає команду дисковому контролеру зчитати дані в буфер.
2. Зчитуються дані в буфер, контролер перевіряє контрольну суму зчитаних даних (перевірка на наявність помилок). Процесор до переривання перемикається на інші завдання.
3. Контролер диска ініціює переривання.
4. Операційна система починає працювати і може зчитувати дані з буфера в пам'ять.

Операції, виконувані з контролером:

1. Процесор програмує контролер (які дані і куди переміс-тити).
2. Процесор дає команду дискового контролера зчитати дані в буфер.
3. Зчитуються дані в буфер, контролер диска перевіряє контрольну суму зчитаних даних (процесор до переривання перемикається на інші завдання).
4. Контролер DMA надсилає запит на зчитування дискового контролера.
5. Контролер диска поставляє дані на шину, адреса пам'яті вже міститься на шині, відбувається запис даних у пам'ять.
6. Коли запис закінчений, контролер диска посилає підтвердження контролеру DMA.
7. Контролер DMA збільшує використовувану адресу та зменшує значення лічильника байтів.
8. Операції повторюються з пункту 4, поки значення лічильника не дорівнюватимеме нулю.
9. Контролер DMA ініціює переривання. Операційна система не повинна копіювати дані в пам'ять, вони вже там.

Переривання. Після того як пристрій введення-виведення розпочав роботу, процесор перемикається на інші завдання.

Щоб сигналізувати процесору про закінчення роботи, пристрій ініціює переривання, виставляючи сигнал на виділену для пристрою лінію шини (а не виділений провід).

Контролер переривань обслуговує переривання, що надходять від пристроїв:

1. Якщо необроблених переривань немає, переривання виконується негайно.

2. Якщо є необроблені переривання, контролер ігнорує переривання. Але пристрій продовжує утримувати сигнал переривання на шині доти, доки воно не буде оброблено.

Алгоритм роботи переривань (рис. 2.34):

– пристрій виставляє сигнал переривання;

– контролер переривань ініціює переривання, указуючи номер пристрою;

– процесор починає обробляти переривання, викликаючи процедуру; ця процедура підтверджує отримання переривання контролеру переривань.

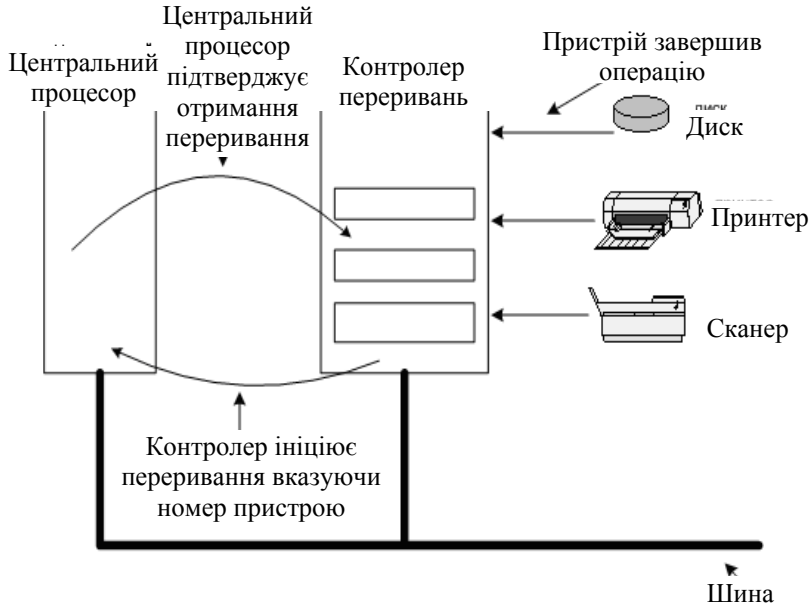


Рис. 2.34. Робота переривань

2.9.2. Принципи програмного забезпечення введення-виведення

Завдання програмного забезпечення введення-виведення. Основні завдання, які має вирішувати ПЗ введення-виведення:

1. Незалежність від пристроїв – наприклад, програма, що зчитує дані з файлу не повинна «замислюватися» з чого вона зчитує (CD, HDD та ін.). Усі проблеми повинна вирішувати ОС.
2. Однакове іменування – назва файлу або пристрою не повинні відрізнятися. (У системах UNIX виконується дослівно).
3. Оброблення помилок – помилки можуть бути виявлені на рівні контролера, драйвера і т. ін.
4. Передавання даних – синхронна та асинхронна (в останньому випадку процесор запускає перенесення даних і перемикається на інші завдання до переривання).
5. Буферизація.
6. Проблема виділених (принтер) і невиділених (диск) пристроїв – принтер повинен надаватися тільки одному користувачу, а диск багатьом; ОС повинна вирішувати всі виникаючі проблеми.

Три основні способи виконання операцій введення-виведення:

– програмне введення-виведення;
– кероване перериваннями введення-виведення;
– введення-виведення з використанням DMA.

Програмне введення-виведення. У цьому випадку всю роботу виконує центральний процесор. Розглянемо процес друку рядка *ABCDEFGH* цим способом (рис. 2.35).

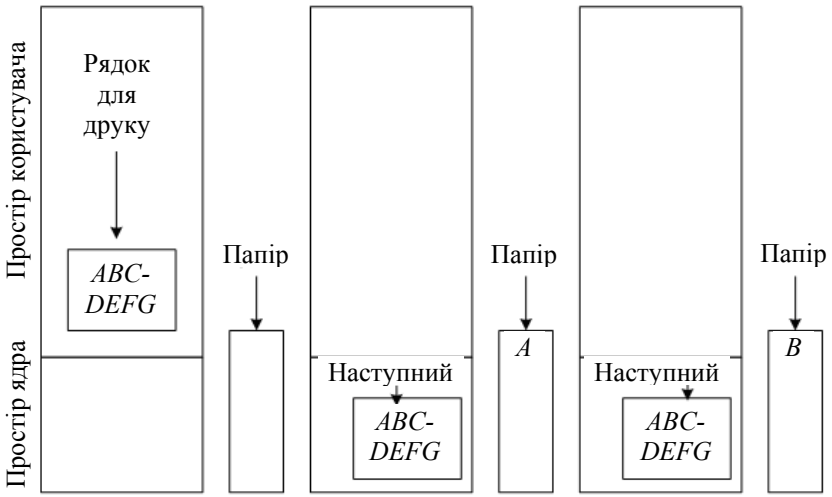


Рис. 2.35. Етапи друку рядка *ABCDEFGH*

Алгоритм друку рядка:

1. Фраза для друку складається у просторі користувача.
2. Звертаючись до системного виклику, процес отримує принтер.
3. Звертаючись до системного виклику, процес просить роздрукувати рядок на принтері.
4. Операційна система копіює рядок у масив, розміщений в режимі ядра.
5. Операційна система копіює перший символ у регістр даних принтера, який відображений у пам'яті.
6. Символ друкується на папері.
7. Указівник установлюється на наступний символ.
8. Процесор чекає, коли біт готовності принтера виставиться у стан готовності.
9. Процес друку повторюється.

У разі використання буфера принтера спочатку весь рядок копіюється в буфер, після цього починається друк.

Кероване перериваннями введення-виведення. Якщо для програми введення-виведення буфер не використовується, а принтер друкує 100 символів за секунду, то на кожен символ буде витрача-

тися 10 мс, у цей час процесор простоюватиме, очікуючи готовність принтера.

Розглянемо цей приклад, але з деяким удосконаленням.

Алгоритм друку рядка:

1. До пункту 8 попереднього алгоритму процес друку аналогічний.
2. Процесор не чекає готовності принтера, а викликає планувальника і перемикається на інше завдання. Друкувальний процес блокується.
3. Коли принтер буде готовий, він надсилає переривання процесору.
4. Процесор перемикається на процес друку.

Уведення-виведення з використанням DMA. Недолік попереднього методу полягає в тому, що переривання відбувається у процесі друку кожного символа.

Алгоритм друку не відрізняється від попереднього, але всю роботу виконує контролер DMA.

Програмні рівні та функції введення-виведення. Чотири рівні введення-виведення зображено на рис. 2.36.

Рівень користувача
Пристрій – незалежне програмне забезпечення ОС
Драйвери пристроїв
Обробники переривань
Апаратура

Рис. 2.36. Рівні введення-виведення

Обробники переривань. Переривання повинні бути сховані якомога глибше в тіло ОС, щоб менша частина ОС мала до них доступ. Найкраще блокувати драйвер, який почав введення-виведення.

Алгоритм блокування драйвера:

1. Драйвер починає операцію введення-виведення.

2. Драйвер блокує сам себе, виконавши за допомогою семафора процедуру *down*; змінної стану – процедуру *wait*; повідомлення – процедуру *receive*.

3. Відбувається переривання.

4. Оброблювач переривань починає роботу.

5. Оброблювач переривань може розблокувати драйвер (наприклад, виконавши на семафорі процедуру *up*).

Драйвери пристроїв. Драйвер пристрою необхідний для кожного пристрою. Для різних ОС потрібні різні драйвери.

Драйвери мають бути частиною ядра (в монолітній системі), щоб одержати доступ до регістрів контролера (рис. 2.37).

Це одна з основних причин, що призводять до знищення ОС, оскільки драйвери, як правило, створюються виробниками пристроїв, і вставляються в ОС.

Насправді обмін даними між контролерами і драйверами відбувається по шині. Драйвери мають взаємодіяти з ОС через стандартні інтерфейси.

Стандартні інтерфейси, які повинні підтримувати драйвери:

– для блокових пристроїв;

– для символьних пристроїв.

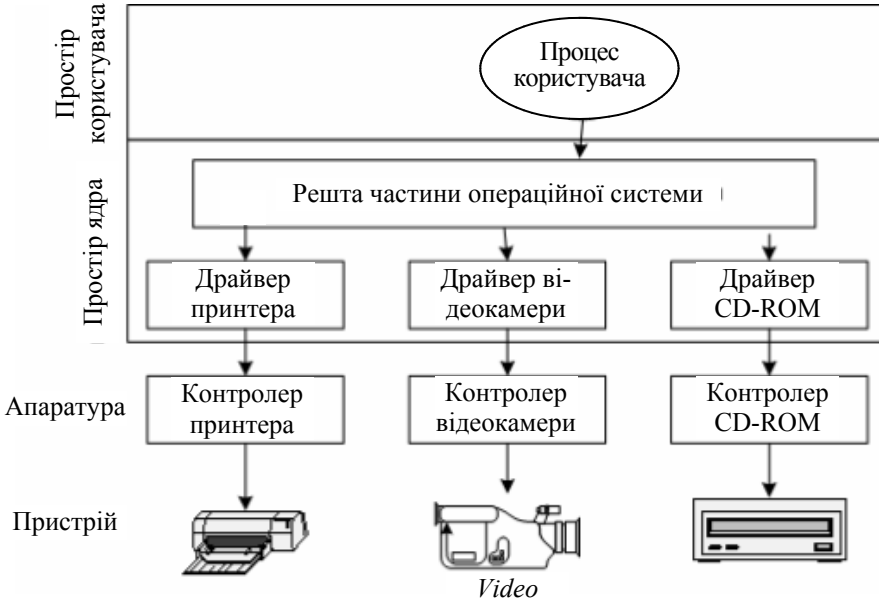


Рис. 2.37. Логічне розміщення драйверів пристроїв

Раніше для встановлення ядра доводилося перекомпільовувати ядра системи. Тепер ОС завантажують драйвери. Деякі драйвери можуть бути завантажені в «гарячому» режимі.

Функції, які виконують драйвери:

– оброблення запитів зчитування або запису;
– ініціалізація пристрою;
– керування енергоспоживанням пристрою;
– прогрівання пристрою (сканера);
– увімкнення пристрою або запуск двигуна.

2.9.3. Незалежне від пристроїв програмне забезпечення введення-виведення

Функції незалежного від пристроїв ПЗ уведення-виведення:

– забезпечення однакового інтерфейсу для драйверів пристроїв;

– буферизація;

– повідомлення про помилки;

– захоплення та звільнення виділених пристроїв (блокування);

– забезпечення розміру блока, що не залежить від пристроїв.

Забезпечення однакового інтерфейсу для драйверів пристроїв. Крім самого інтерфейсу, до нього також входять-іменування та захист пристроїв.

Буферизація. Кілька прикладів буферизації (рис. 2.38):

а) *небуферизоване введення* – після введення кожного символа відбувається переривання;

б) *буферизація в просторі користувача* – доводиться тримати завантаженими необхідні сторінки пам'яті у фізичній пам'яті;

в) *буферизація в ядрі з копіюванням у простір користувача* – сторінка завантажується тільки коли буфер ядра повний, дані з буфера ядра в буфер користувача копіюються за одну операцію. Проблема може виникнути, коли буфер ядра повний, а сторінка буфера користувача ще не завантажена;

г) *подвійна буферизація в ядрі* – якщо один буфер заповнений, і поки він вивантажується, символи пишуться в другий буфер.

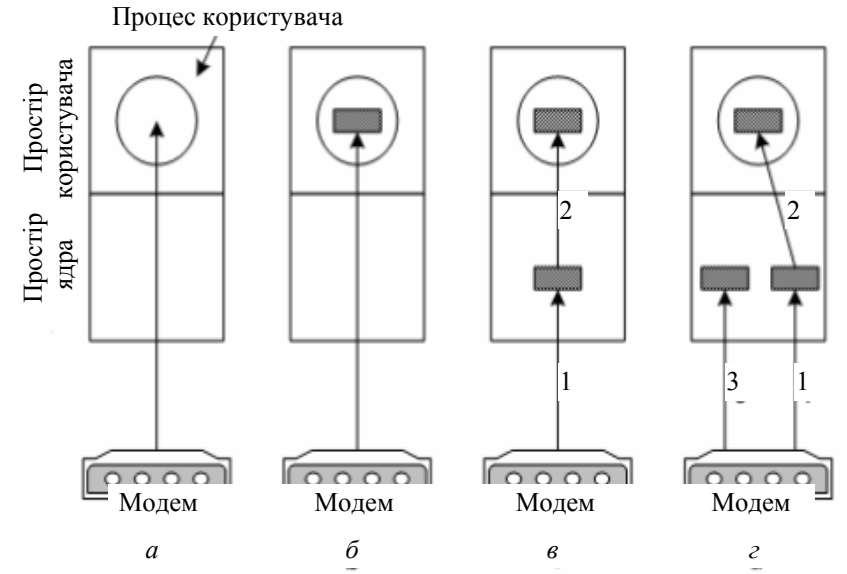


Рис. 2.38. Приклади буферизації

Повідомлення про помилки. Найбільше помилок виникає внаслідок операції введення-виведення, тому їх потрібно виявляти якомога раніше. Помилки можуть бути дуже різні залежно від пристроїв.

Захоплення і звільнення виділених пристроїв. Для пристроїв (наприклад принтера) з якими повинен працювати в один час тільки один процес, необхідна можливість захоплення і звільнення пристроїв. Коли один процес зайняв пристрій, інші стають в чергу.

Розмір блока має бути однаковий для верхніх рівнів, і не залежати від пристроїв (розмірів секторів на диску).

Функції ПЗ введення-виведення простору користувача:

– звернення до системних викликів введення-виведення (через бібліотечні процедури);

– форматне введення-виведення (змінюють формат, наприклад, в ASCII);

– спулінгування (для виділених пристроїв) – створюється процес і каталог спулеру.

Рівні та основні функції системи введення-виведення показано на рис. 2.39.

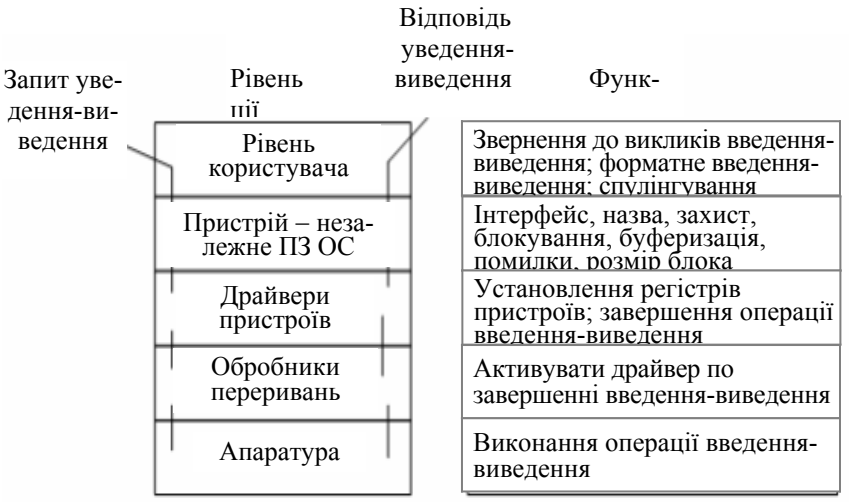


Рис. 2.39. Рівні та основні функції системи введення-виведення

Запитання для самоперевірки

1. Наведіть визначення терміна *процес*.
2. Наведіть визначення терміна *потік* або *нитка*.
3. Розкрийте стани процесу, потоку.
4. Що таке планування процесів і потоків?
5. Які види планування ви знаєте?
6. Що таке диспетчеризація процесів і потоків?
7. Чим розрізняються процеси і потоки?
8. Наведіть визначення процесора.
9. Наведіть визначення обчислювальної системи.
10. Наведіть визначення паралелізму.
11. Назвіть типи паралелізму та коротко їх охарактеризуйте.
12. Назвіть критерії оцінювання ефективності використання багатопроцесорних систем.
13. Наведіть визначення терміна *переривання*.
14. Назвіть та охарактиризуйте призначення і типи переривань.
15. Наведіть визначення терміна *система переривань*.
16. Наведіть визначення терміна *ресурс*.
17. Назвіть та охарактиризуйте призначення і типи ресурсів.
18. Назвіть функції ОС з керування пам’яттю.
19. Які бувають адреси?
20. Назвіть і охарактеризуйте алгоритми розподілу пам’яті.
21. На які категорії діляться пристрої?
22. Охарактеризуйте контролер DMA.
23. Охарактеризуйте контролер переривань.
24. Назвіть основні завдання, які має вирішувати ПЗ уведення-виведення.
25. Що таке драйвер пристрою?
26. Назвіть функції незалежного від пристроїв ПЗ уведення-виведення.
27. Назвіть функції ПЗ уведення-виведення простору користувача.

3. СИСТЕМИ ПРОГРАМУВАННЯ

3.1. Сучасні технології та етапи розроблення програмного забезпечення

3.1.1. Системне та прикладне програмне забезпечення

В основу роботи комп'ютерів покладено програмний принцип керування, який полягає в тому, що комп'ютер виконує дії за заздалегідь заданою програмою. Цей принцип забезпечує універсальність використання комп'ютера: у певний момент часу розв'язується завдання відповідно до вибраної програми. Після її завершення у пам'ять завантажується інша програма. Програма – це запис алгоритму розв'язання завдання у вигляді послідовності команд або операторів мовою, яку розуміє комп'ютер. Кінцевою метою будь-якої комп'ютерної програми є керування апаратними засобами.

Для належного розв'язання завдань на комп'ютері потрібно, щоб програма була налагоджена, не потребувала доробок і мала відповідну документацію. Тому стосовно роботи на комп'ютері часто використовують термін *програмне забезпечення* (*software*), під яким розуміють сукупність програм, процедур і правил, а також доку-ментації, що стосуються функціонування системи оброблення даних.

Системне програмне забезпечення. Програмне та апаратне забезпечення у комп'ютері функціонують у нерозривному зв'язку та взаємодії. Склад ПЗ обчислювальної системи називають *програмною конфігурацією*. Між програмами існує взаємозв'язок, тобто багато програм працюють, базуючись на програмах нижчого рівня.

Міжпрограмний інтерфейс – це розподіл ПЗ на декілька пов'язаних між собою рівнів. Рівні ПЗ являють собою піраміду, де кожен вищий рівень ґрунтується на ПЗ попередніх рівнів: системному, службовому та прикладному.

До системного (базового) рівня належить ПЗ, що керує базовим інтерфейсом взаємодії користувача та комп'ютера, тобто ОС.

Різні ОС (ОС MS-DOS, ОС Windows, ОС Linux, ОС Unix та ін.) використовують ті чи інші можливості обслуговування компонентів комп'ютера і організації діалогу з користувачем. Основними

характеристиками ОС є розрядність, а також підтримання багатопроцесорності, багатозадачності та багатокористувацького режиму.

Операційні системи містять інтерфейс взаємодії з користувачем ПК, що включає в себе вигляд екрана під час роботи з тією чи іншою програмою, способи керування нею, зокрема всі ті засоби, за допомогою яких вона контактує з користувачами.

Операційна система є складним комплексом програм та програмних засобів. Ці програми та файли згруповані у певну структуру деяким чином за функціями, які вони виконують під час функціонування системи – забезпечення введення, прорисовування інтерфейсу, робота з файлами та ін.

3.1.2. Службове програмне забезпечення

Програми цього рівня взаємодіють як із програмами базового рівня, так і з програмами системного рівня. Призначення службових програм (утиліт) полягає у автоматизації робіт з перевірки та налаштування комп'ютерної системи, а також для покращення функцій системних програм. Деякі службові програми (програми обслуговування) відразу додають до складу ОС, доповнюючи її ядро, але більшість є зовнішніми програмами і розширюють функції ОС.

Найбільш поширені службові програми:

1. Диспетчери файлів (файлові менеджери). За їх допомогою виконується більшість операцій з обслуговування файлової структури копіювання, переміщення, перейменування файлів, створення каталогів (папок), знищення об'єктів, пошук файлів та навігація у файловій структурі. Ці базові програмні засоби містяться у складі програм системного рівня і встановлюються разом з ОС.

2. Засоби стиснення даних (архіватори). Призначені для створення архівів. Архівні файли мають підвищену щільність запису інформації і відповідно ефективніші для зберігання та перенесення інформації.

3. Засоби діагностики. Призначені для автоматизації процесів діагностування програмного та апаратного забезпечення. Їх використовують для виправлення помилок і оптимізації роботи комп'ютерної системи.

4. Програми інсталяції (установлення). Призначені для контролю за додаванням у поточну програмну конфігурацію нового ПЗ. Вони слідкують за станом і зміною програмного середовища, відслідковують та протоколюють утворення нових зв'язків, загублені під час знищення певних програм. Прості засоби керування встановленням та знищенням програм містяться у складі ОС, але можуть використовуватись і додаткові службові програми.

5. Засоби комунікації. Дозволяють установлювати з'єднання з віддаленими комп'ютерами, передають повідомлення електронної пошти, пересилають факсимільні повідомлення тощо.

6. Засоби перегляду та відтворення. Застосовують переважно для роботи з файлами, їх завантажують у «рідну» прикладну систему і вносять необхідні виправлення.

7. Засоби комп'ютерної безпеки. До них належать засоби пасивного та активного захисту даних від пошкодження, несанкціонованого доступу, перегляду та зміни даних. Засоби пасивного захисту – це службові програми, призначені для резервного копіювання. Засоби активного захисту застосовують антивірусне ПЗ. Для захисту даних від несанкціонованого доступу, їх перегляду та зміни використовують спеціальні системи, базовані на криптографії.

3.1.3. Прикладне програмне забезпечення

Програмне забезпечення цього рівня являє собою комплекс прикладних програм, за допомогою яких виконуються конкретні завдання (від виробничих до творчих, розважальних та навчальних). Між прикладним та системним програмним забезпеченням існує тісний взаємозв'язок.

Універсальність обчислювальної системи, доступність прикладних програм і широта функціональних можливостей комп'ютера безпосередньо залежать від типу наявної ОС, системних засобів, що містяться у її ядрі й взаємодії комплексу людина–програма–обладнання. Існує така класифікація прикладного ПЗ:

1. Текстові редактори. Основними функціями є введення та редагування текстових даних. Для операцій введення, виведення та зберігання даних текстові редактори використовують системне ПЗ.

З цього класу прикладних програм починають ознайомлення з ПЗ і набувають перших навичок роботи з комп'ютером.

2. Текстові процесори. Дозволяють форматувати, тобто оформлювати текст. Основними засобами текстових процесорів є засоби забезпечення взаємодії тексту, графіки, таблиць та інших об'єктів, що складають готовий документ, а також засоби автоматизації процесів редагування та форматування. Сучасний стиль роботи з документами має два підходи: робота з паперовими документами та робота з електронними документами. Прийоми та методи форматування таких документів різняться між собою, але текстові процесори спроможні ефективно опрацьовувати обидва види документів.

3. Графічні редактори. Широкий клас програм, призначені для створення та оброблення графічних зображень. Розрізняють три категорії:

– растрові редактори;

– векторні редактори;

– 3-D редактори (тривимірна графіка).

4. Системи керування базами даних (СКБД). Базою даних називають великі масиви даних організовані у табличні структури. Основні можливості СКБД:

– створення порожньої структури бази даних;

– забезпечення засобів її заповнення або імпорту даних із таблиць іншої бази;

– уможливлення доступу до даних, наявність засобів пошуку й фільтрації.

У зв'язку з поширенням мережевих технологій від сучасних СКБД вимагається можливість роботи з віддаленими й розподіленими ресурсами, розміщеними на серверах Інтернету.

5. Електронні таблиці. Надають комплексні засоби для збереження різних типів даних та їх оброблення. Основний акцент зміщений на перетворення даних; надано широкий спектр методів для роботи з числовими даними. Основна особливість електронних таблиць полягає в автоматичній зміні вмісту всіх комірок у разі зміни відношень, заданих математичними або логічними формулами. Широке застосування знаходять у бухгалтерському обліку, аналізі фінансових і торговельних ринків, засобах оброблення результатів експериментів, тобто в автоматизації регулярно повторюваних обчислень великих обсягів числових даних.

6. Системи автоматизованого проектування (CAD-системи). Призначені для автоматизації проектно-конструкторських робіт. Застосовуються у машинобудуванні, приладобудуванні, архітектурі. Окрім графічних робіт, дозволяють виконувати прості розрахунки та вибирати готові конструктивні елементи з існуючої бази даних. Особливість CAD-систем полягає у автоматичному забезпеченні на всіх етапах проектування технічних умов, норм та правил. Система автоматизованого проектування є необхідним компонентом для гнучких виробничих систем та автоматизованих систем керування технологічними процесами (АСК ТП).

7. Настільні видавничі системи. Автоматизують процес верстання поліграфічних видань. Займає проміжний стан між текстовими процесами та САПР. Видавничі системи відрізняються розширеними засобами керування взаємодії тексту з параметрами сторінки і графічними об'єктами, але мають слабші можливості щодо автоматизації введення та редагування тексту. Їх доцільно застосовувати до документів, зазделегідь оброблених у текстових процесорах та графічних редакторах.

8. Редактори HTML (Web-редактори). Особливий клас редакторів, що об'єднують у собі можливості текстових та графічних редакторів. Призначені для створення і редагування Web-сторінок Інтернету.

Програми цього класу можна також використовувати під час підготовки електронних документів і мультимедійних видань.

9. Браузери (засоби перегляду Web-документів). Програмні засоби призначені для перегляду електронних документів, створених у форматі HTML. Відтворюють, окрім тексту та графіки, також музику, людську мову, радіопередачі, відеоконференції і дозволяють працювати з електронною поштою.

10. Системи автоматизованого перекладу. Розрізняють електронні словники та програми перекладу мови. Електронні словники – це засоби для перекладу окремих слів у документі. Потрібні для професійних перекладачів, які самостійно перекладають текст. Програми автоматичного перекладу отримують текст однією мовою і видають текст іншою, тобто автоматизують переклад. Автоматизованний переклад не забезпечує якісного вихідного тексту, оскільки все зводиться до перекладу окремих лексичних одиниць.

11. Інтегровані системи діловодства. Засоби для автоматизації робочого місця керівника. Зокрема, це функції створення, реда-

гування і форматування документів; централізація функцій електронної пошти, факсимільного та телефонного зв'язку; диспетчеризація та моніторинг документообігу підприємства, координація дій підрозділів; оптимізація адміністративно-господарської діяльності й постачання оперативної та довідкової інформації.

12. Бухгалтерські системи. Містять у собі функції текстових, табличних редакторів та СКБД. Призначені для автоматизації підготовки початкових бухгалтерських документів підприємства та їх обліку, регулярних звітів за підсумками виробничої, господарської та фінансової діяльності у формі прийнятної для податкових органів, позабюджетних фондів та органів статистичного обліку.

13. Фінансові аналітичні системи. Використовують у банківських та біржових структурах. Дозволяють контролювати та прогнозувати ситуацію на фінансових, торговельних та ринків сировини, виконувати аналіз поточних подій, готувати звіти.

14. Експертні системи. Призначені для аналізу даних, що містяться у базах знань, і видачі результатів за запитом користувача. Такі системи використовуються, коли для прийняття рішення потрібні широкі спеціальні знання. Використовуються у медицині, фармакології, хімії, юриспруденції. З використанням експертних систем пов'язана галузь науки – інженерія знань. Інженери знань – це фахівці, які є проміжною ланкою між розробниками експертних систем (програмістами) та провідними фахівцями у конкретних галузях науки й техніки (експертами).

15. Геоінформаційні системи. Призначені для автоматизації картографічних та геодезичних робіт на основі інформації, отриманої топографічним або аерографічними методами.

16. Системи відеомонтажу. Призначені для цифрового оброблення відеоматеріалів, монтажу, створення відеоефектів, виправлення дефектів, додавання звуку, титрів та субтитрів. Окремі категорії є навчальними, довідковими та розважальними системами й програмами. Характерною особливістю є підвищені вимоги до мультимедійної складової.

17. Інструментальні мови та системи програмування. Ці засоби призначені для розроблення нових програм. Комп'ютер «розуміє» і може виконувати програми у машинному коді. Кожна команда при цьому має вигляд послідовності нулів й одиниць. Писати програми машинною мовою дуже незручно, а їх надійність низка.

Тому програми розробляють мовою, зрозумілою людині (інструментальною мовою або алгоритмічною мовою програмування), після чого спеціальною програмою, яка називається транслятором, текст програми перекладається (транслюється) на машинний код.

До мов низького рівня належать асемблери, а високого – Pascal, Basic, C/C++, мови баз даних і т. ін. Систему програмування, крім транслятора, складають текстовий редактор, компонувальник, бібліотека стандартних програм, налагоджувач, візуальні засоби автоматизації програмування. Прикладами таких систем є Delphi, Visual Basic, Visual C++, Visual FoxPro та ін.

3.1.4. Сучасні технології розроблення програмного забезпечення

Технологія розроблення ПЗ – система інженерних принципів для створення ПЗ, що надійно та ефективно працює в реальних комп'ютерах.

Розрізняють методи, засоби та процедури технології розроблення ПЗ.

Методи забезпечують розв'язання таких завдань:

– планування й оцінювання проекту;

– аналіз системних та програмних вимог;

– проектування алгоритмів, структур даних та програмних структур;

– кодування;

– тестування;

– супроводження.

Засоби (утиліти) технології розроблення ПЗ потрібні для автоматизованої підтримки методів технології розроблення ПЗ. Для спільного використання утиліти можуть об'єднуватися в системи автоматизованого конструювання ПЗ. Такі системи називають CASE-системами (CASE – Computer Software Engineering – програмна інженерія з комп'ютерною підтримкою).

Процедури є тим, що поєднує методи й утиліти для того, щоб вони забезпечували безперервний технологічний ланцюг розроблення.

Процедури визначають:

– порядок використання методів та утиліт;

– формування звітів, форм за відповідними вимогами;

– контроль, який допомагає забезпечувати якість та координувати зміни;

– формування точок контролю, за якими керівники оцінюють прогрес у розробленні програми.

Процес конструювання ПЗ складається з послідовності кроків, що використовують методи, утиліти та процедури. Ці послідовності кроків часто називають парадигмами технології розроблення ПЗ.

Використання парадигм технології розроблення ПЗ гарантує систематичний, впорядкований підхід до промислового розроблення, використання та супроводження ПЗ.

Найдавнішою парадигмою процесу розроблення ПЗ є класичний життєвий цикл (автор Уїнстон Ройс, 1970).

Досить часто класичний життєвий цикл називають *каскадною* або *водоспадною моделлю*, підкреслюючи цим, що розроблення розглядається як послідовність етапів, до того ж перехід на наступний, ієрархічно нижчий етап здійснюється лише після повного завершення робіт на поточному етапі (рис 3.1).

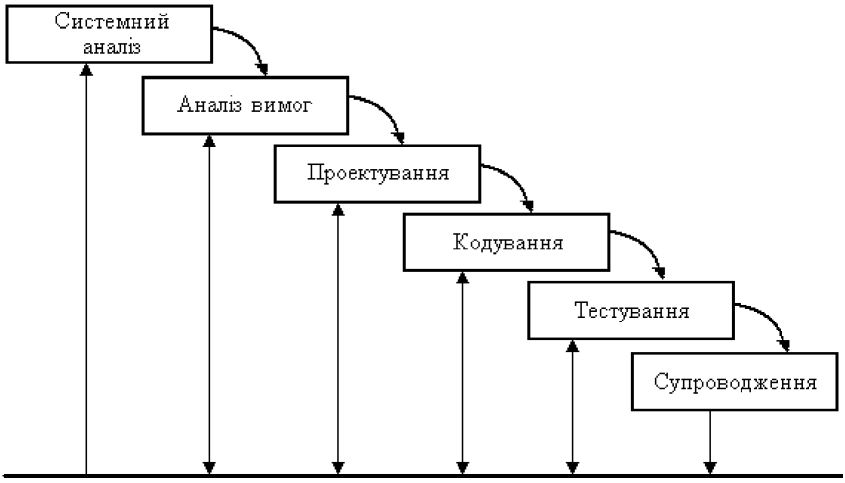


Рис. 3.1. Класичний життєвий цикл розроблення ПЗ

Вважається, що розроблення починається на системному рівні і проходить етапи аналізу, проектування, кодування, тестування та

супроводження. При цьому моделюють дії стандартного інженерного циклу.

Системний аналіз задає функцію кожному елементу в комп'ютерній системі, взаємодію елементів одне з одним. Аналіз починається з визначення вимог до всіх системних елементів та визначення підмножини цих вимог до програми. Необхідність системного підходу явно проявляється, коли формується інтерфейс ПЗ з іншими елементами (апаратурою, людьми, базами даних). На цьому ж етапі починається розв'язання завдань з планування проекту ПЗ. У процесі планування проекту визначаються обсяг проектних робіт та їх ризик, необхідні трудовтрати, формуються завдання і план-графік робіт.

Аналіз вимог стосується ПЗ (на відміну від системного аналізу, коли розглядається система в цілому). Уточнюють і деталізують функції ПЗ, характеристики та інтерфейс. Усі визначення документують у с*пецифікації аналізу***.** Тут же завершується розв'язання завдання з планування проекту.

Проектування ґрунтується на створенні:

– архітектури ПЗ;

– модульної структури ПЗ;

– алгоритмічної структури ПЗ;

– структури даних;

– вхідного та вихідного інтерфейсів (вхідних та вихідних форм даних).

Вихідні дані для проектування містяться у *специфікації аналізу*, тобто протягом проектування здійснюється перехід вимог до ПЗ на множину проектних рішень. Під час розв'язання завдань проектування основна увага приділяється якості майбутнього програмного продукту.

Кодування полягає у переведенні результатів проектування на текст мовою програмування.

Тестування – виконання програми для виявлення дефектів у функціях, логіці та формі реалізації програмного продукту.

Супроводження – це внесення змін у ПЗ, що експлуатується. Призначення змін:

– виправлення помилок;

– адаптація до змін зовнішнього відносно ПЗ середовища;

– удосконалення ПЗ відповідно до вимог замовника.

Супроводження ПЗ полягає в повторному використанні кожного з попередніх кроків (етапів) життєвого циклу до існуючої програми, а не в розробленні нової програми.

Переваги класичного життєвого циклу: складає план та часовий графік на всіх етапах проекту, упорядковує хід конструювання.

*Недоліки класичного життєвого циклу***:** 1) реальні проекти часто потребують відхилення від стандартної послідовності кроків; 2) цикл ґрунтується на точному формулюванні вихідних вимог до ПЗ (що не завжди властиве реальним проектам).

Модель швидкого розроблення додатків RAD (*Rapid Application Development*) – одна з найбільш використовуваних моделей технології розроблення ПЗ (рис. 3.2).

Рис. 3.2. Модель швидкого розроблення додатків

Модель RAD забезпечує екстремально короткий цикл розроблення. RAD – це надшвидкісна адаптація лінійної послідовної моделі, в якій швидке розроблення досягається за рахунок використання компонентно-орієнтованого конструювання. RAD – підхід орієнтований на розроблення інформаційних систем і виокремлює такі етапи:

*Бізнес-моделювання***.** Моделюється інформаційний потік бізнес-функцій, у якому зазначається вид інформації, що керує бізнес-процесом та генерується; хто її генерує; де інформація використовується та хто її обробляє.

Моделювання даних. Визначаються перетворення об'єктів даних, що забезпечують реалізацію бізнес-функцій. Створюються описи оброблення для додавання, модифікації, видалення або знаходження (виправлення) об'єктів даних.

Генерація додатка. Припускається використання методів, орієнтованих на мови програмування четвертого покоління. Замість створення ПЗ за допомогою мов програмування третього покоління RAD-процес працює з програмними компонентами, що використовуються повторно або створють утиліти автоматизації.

Тестування та об'єднання. Це зменшує час на тестування (оскільки численні програмні елементи вже протестовані, хоча нові елементи мають бути також протестовані).

Використовувати RAD можна в тому випадку, коли кожна головна функція завершиться за три місяці. Кожна головна функція адресована окремій групі розробників, а потім інтегрується в цілу систему.

Переваги та недоліки RAD:

1) для великих проектів у RAD необхідні істотні людські ресурси (необхідно створити достатню кількість груп);

2) RAD можна використовувати лише для таких додатків, які можуть бути розбиті на окремі модулі і в яких продуктивність не є критичною величиною;

3) RAD не можливо використовувати в умовах високих технічних ризиків (тобто в разі використання нових технологій).

3.1.5. Спіральна модель

Для розроблення великих складних сучасних програм використовують еволюційні стратегії конструювання.

Спіральна модель – один з найпоширеніших варіантів використання цієї стратегії.

Спіральна модель (Баррі Боем, 1988) ґрунтується на властивостях класичного життєвого циклу, до яких додається новий елемент – аналіз ризику (рис 3.3).

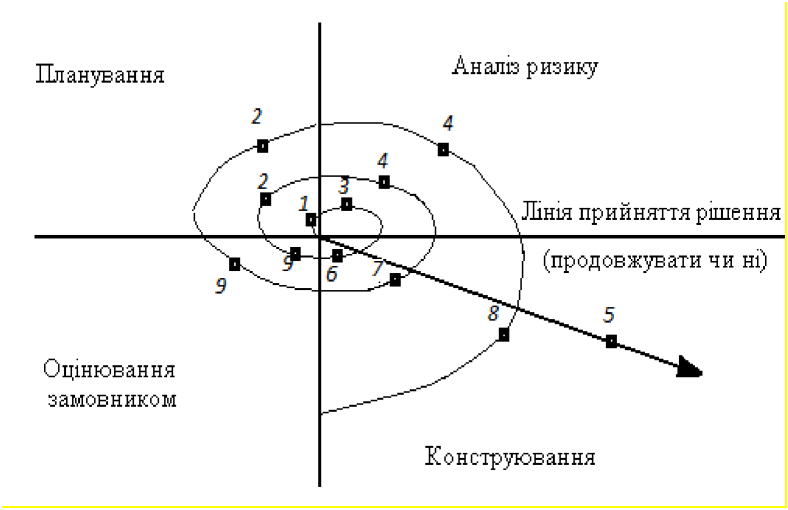


Рис. 3.3. Спіральна модель*: 1* – початковий збір вимог та планування проекту; *2* – збір вимог вже на основі рекомендацій замовника*; 3*- аналіз ризику на основі початкових вимог; *4* – аналіз ризику на основі реакції замовника; *5* – перехід до комплексної системи*; 6* – початковий макет системи; *7* – наступний рівень макету; *8* – побудована система*; 9* – оцінювання замовником

Як показано на рис. 3.3, модель визначає чотири дії, які подано чотирма квадрантами спіралі:

– планування – визначення цілей, варіантів та вимог;

– аналіз ризику – аналіз варіантів та оцінювання ризику(ів);

– конструювання – розроблення продукту наступного рівня;

– оцінювання – оцінювання замовником поточних результатів конструювання.

За рахунок руху по радіальній складовій спіралі досягається інтегральний ефект. З кожною інтеграцією по спіралі (просуванням від центра до периферії) будуються більш повні версії ПЗ. На першому витку спіралі визначають початкові цілі, варіанти та обмеження, розпізнається та аналізується ризик. Якщо аналіз ризику показує невизначеність вимог, то на допомогу розробнику та замовнику приходить макетування (яке використовують у квадранті конструювання). Для подальшого визначення проблемних та уточнених вимог можна скористатися моделюванням. Замовник оцінює інженерну (конструк-

торську) роботу і вносить пропозиції з модифікації (квадрант оцінювання замовником). Наступна фаза планування та аналізу ризику базується на пропозиціях замовника. У кожному циклі по спіралі результати аналізу ризику формуються у вигляді «продовжувати, не продовжувати». Якщо ризик надто великий, від проекту можна відмовитися.

У більшості випадків рух по спіралі триває і з кожним кроком просуває розробників до більш загальної моделі системи. У кожному циклі по спіралі потрібне конструювання (нижній правий квадрант), яке можна реалізувати класичним життєвим циклом або макетуванням.

Переваги спіральної моделі: 1) найреальніше (у вигляді еволюції) відображає розроблення ПЗ; 2) дозволяє явно враховувати ризик на кожному витку еволюції розроблення; 3) містить крок системного підходу в інтеграційну структуру розроблення; 4) використовує моделювання для зменшення ризику та вдосконалення програмного виробу.

Недоліки спіральної моделі: 1) новизна (бракує достатньої статистики ефективності моделі); 2) підвищені вимоги до замовника; 3) ускладнення під час контролю та керування часом розроблення.

3.1.6. Етапи розроблення програмного забезпечення

Відомо, що технологічний цикл конструювання ПЗ містить три процеси:

1) аналіз;

2) синтез;

3) супроводження.

Протягом аналізу виявляються цілі створюваної системи, тобто вказуються функції та зміни стану системи залежно від оточення та керованих параметрів.

У процесі синтезу вказуються способи реалізації запропонованих на першому етапі функцій системи, тобто виконується програмна реалізація системи. Виокремлюють три ***етапи синтезу:***

1) проектування ПЗ;

2) кодування ПЗ;

3) тестування ПЗ (рис. 3.4).

Рис.3.4. Інформаційні потоки процесу синтезу ПЗ

Етап проектування доповнює вимоги до ПЗ, які подані моделями аналізу: інформаційною, функціональною моделями та моделлю поведінки. *Інформаційна модель* описує інформацію, яку, за задумом, має обробляти ПЗ. *Функціональна модель* визначає перелік функцій оброблення. *Модель поведінки* фіксує бажану динаміку системи (режими її роботи). На виході з етапу проектування виконується розроблення даних, архітектури та процедурне розроблення ПЗ.

Розроблення даних – це результат перетворення інформаційної моделі аналізу в структури даних для реалізації програмної системи.

Розроблення архітектури – виокремлення основних структурних компонентів та фіксація зв'язків між ними.

Процедурне розроблення – опис послідовності дій у структурних компонентах, тобто визначення їх умісту.

Далі створюють тести програмних модулів, виконують тестування та перевірку системи програмування. На проектування, кодування і тестування відводиться 75 % вартості конструювання системи програмування.

Рішення, які приймаються протягом проектування, роблять цей процес провідним етапом процесу синтезу. Важливість проектування визначається ще і *якістю.* Проектування – етап, на якому «вирощують» якість розроблення системи програмування, це єдиний шлях, який забезпечує правильну трансляцію вимог замовника в кінцевий програмний продукт.

3.1.7. Розроблення програмних модулів

Методології розроблення ПЗ корисні для розроблення великих складних продуктів або розподілених інформаційних систем. Розробляючи відносно невеликі програми або реалізуючи конкретний програмний модуль, достатньо притримуватися такої послідовності кроків:

1. Постановка завдання. Формування завдання природною мовою. Визначення мети. Підготовка технічного завдання на розроблення програми.

2. Обґрунтований вибір засобів розроблення (програмування). Розроблення форматів введения вихідних даних і відображення результатів.

3. Вибір методу розв'язання завдання. Аналіз можливості використання раніше розробленого й доступного для програміста ПЗ.

4. Розроблення алгоритму рішення завдання. Декомпозиція завдання на підзадачі. Визначення послідовності розв'язання підзадач. Розроблення структури програми.

5. Кодування засобами обраної мови програмування.

6. Верифікація й перевірка коректності. Аналітичний доказ правильності програми.

7. Тестування програми. Розроблення тестів і контрольних прикладів. Зіставлення реальних і очікуваних результатів.

8. Налагодження програми у випадку виявлення помилок. Локалізація виявлених помилок. Коректура помилок. Повернення до етапу тестування.

9. Розроблення документації. Текстовий опис програми. Розроблення інструкцій користувачу – особі, що застосовує розроблену програму у своїй роботі, інструкцій для експлуатації, що містять інформацію, потрібну програмістам, що відповідають за нормальне функціонування програми.

10. Експериментальна експлуатація. Уточнення вимог замовника до подання вихідних даних і результатів роботи програми. У разі потреби повернення до попередніх етапів.

11. Промислова експлуатація. Супровід програми. Обробленя вимог до нових версій програми.

3.2. Система програмування. Класифікація систем програмування за надаваними можливостями та основні їх функції і компоненти

3.2.1. Уявлення про систему програмування

Стрімкий розвиток обчислювальної техніки зумовив потребу у створенні нових засобів спілкування програмістів з ЕОМ. Найбільш універсальними та потужними засобами такого спілкування є мови програмування.

Під *мовою програмування* розуміють правила подання даних і запису алгоритмів їх оброблення, що автоматично виконуються ЕОМ. У більш абстрактному вигляді мова програмування є засобом створення програмних моделей об'єктів і явищ зовнішнього світу. Натепер вже створено десятки різних мов програмування (як примітивних, або мов нижнього рівня, команди яких є певним аналогом машинних команд (наприклад, мова асемблер), так і близьких до мови людини, наприклад, мова LISP).

Мова програмування призначена для того, щоб людина (програміст) вказувала операції, сценарії та дії, які має виконувати машина. Для цього програміст створює програми, які і є записами цих дій в мові програмування.

Мови програмування можна розглядати з різних точок зору. По-перше, мова програмування є інструментом програміста для створення програм. Для створення якісних програм потрібні зручні мови програмування. Тому розробники мов програмування прагнуть до створення більш досконалих програм.

По друге, процес розроблення програми можна порівнювати з промисловим виробництвом, у якому визначальними чинниками є продуктивність праці колективу програмістів, собівартість і якість програмної продукції. Створюються різноманітні технології розро-

блення програм (структурне, модульне, об'єктно-орієнтоване програмування та ін.), що мають підтримуватися мовою програмування.

Програми можна розглядати як аналог електронних приладів оброблення інформації, в яких замість радіодеталей і мікросхем використовують конструкції мови програмування (елементна база програми). Як і електронні прилади, програми можуть бути найпростішими (рівня детекторного приймача) і дуже складними (рівня автоматичної космічної станції), при цьому рівень інструменту повинен відповідати складності виробу. Крім того, людині зручніше описувати об'єкт, що моделюється, використовуючи терміни предметної галузі, а не мову цифр. Тому важливою рушійною силою, що веде до створення нових, спеціалізованих, орієнтованих на проблемну галузь, потужних мов програмування, є збільшення різноманітності і підвищення складності завдань, які розв'язуються за допомогою ЕОМ.

Удосконалення самих ЕОМ також потребує створення мов, що максимально реалізують нові можливості ЕОМ.

Мова програмування повинна забезпечувати тривалий життєвий цикл програми.

Перші ЕОМ, створені людиною, мали невеликий набір команд і вбудованих типів даних, але дозволяли виконувати програми машинною мовою. Машинна мова – єдина мова, яку розуміє ЕОМ. Вона реалізується апаратно: кожну команду виконує певний електронний пристрій. Програма машинною мовою являє собою послідовність команд і даних, заданих у цифровому вигляді. Наприклад, команда вигляду 1A12 або 0001101000010010 означає операцію додавання (1A) вмісту регістрів 1 і 2. Машинною мовою дані подаються у вигляди чисел і символів. Операції є елементарними і з них будується вся програма. Уведення програми в цифровому вигляді виконувалося безпосередньо в пам'ять з пульта ЕОМ або з примітивних пристроїв уведення. Природно, що процес програмування дуже трудомісткій, а ефект від застосування ЕОМ незначний.

Цей етап розвитку мови програмування показав, що програмування є складною проблемою, що важко піддається автоматизації, а саме програмне забезпечення визначає ефективність застосування ЕОМ. Тому на всіх наступних етапах зусилля спрямовувалися на вдосконалення інтерфейсу між програмістом і ЕОМ – мови

програмування. Прагнення програмістів оперувати не цифрами, а символами зумовило створення мнемонічної мови програмування, що називають асемблером, мнемокодом, автокодом. Ця мова має певний синтаксис запису програм, у якому зокрема цифровий код операції замінений мнемонічним кодом. Наприклад, команда додавання записується у вигляді *AR* 1,2 й означає додавання (*addition*) типу регістр – регістр (*register*) для регістрів 1 і 2. Тепер програма має зручнішу для читання форму, але її не розуміє ЕОМ. Тому потрібно було ще створити спеціальну програму-транслятор, що перетворює програму з мови асемблеру на машинну мову. Це, в свою чергу, потребувало глибоких наукових досліджень і розроблення різноманітних теорій, наприклад, теорії формальних мов, що лягли в основу створення трансляторів. Будь-який клас ЕОМ має свою мову асемблеру. Тепер мова асемблеру використовується для створення системних програм, що використовують специфічні апаратні можливості цього класу ЕОМ.

Наступний етап характеризується створенням мов високого рівня. Ці мови є універсальними (дають змогу створювати будь-які прикладні програми) й алгоритмічно повними, мають більш широкий спектр типів даних і операцій, підтримують технології програмування. Цими мовами створюється безліч різноманітних прикладних програм. Принциповими відмінностями мов високого рівня від мов низького рівня є:

1) використання змінних;
2) можливість запису складних виразів;
3) можливість розширення набору типів даних за рахунок конструювання нових типів з базових;
4) можливість розширення набору операцій за рахунок підключення бібліотек підпрограм;
5) низька залежність від типу ЕОМ. З ускладненням мови програмування модернізуються й транслятори для них.

У набір інструментів програміста, окрім транслятора, входить текстовий редактор для введення тексту програм, настроювач для усунення помилок, бібліотекар для створення бібліотек програмних модулів і багато інших службових програм. Усе це разом називається *системою програмування*. Найбільш яскравими представниками є FORTRAN, PL/1, Pascal, C, Basic, Ada.

Подальший розвиток мов став визначатися новими технологіями програмування. Водночас з розвитком універсальних мов високого рівня стали розвиватися проблемно-орієнтовні мови програмування, що вирішували економічні завдання (COBOL), завдання реального часу (Modula-2, Ada), символьного оброблення (Snobol), моделювання (GPSS, Simula, SmallTalk), числово-аналітичні задачі (Аналітик) та ін. Ці спеціалізовані мови давали змогу адекватніше описувати об'єкти і явища реального світу, наближаючи мови програмування до мови фахівця у проблемній галузі.

Іншим напрямом розвитку мови програмування є створення мов надвисокого рівня. Мовою високого рівня програміст задає процедуру (алгоритм) отримання результату на основі відомих вихідних даних, тому їх називають процедурними мовами програмування. Мовою надвисокого рівня програміст задає відношення між об'єктами в програмі, наприклад, систему лінійних рівнянь, і визначає, що потрібно знайти, але не вказує як отримати результат. Такі мови називають також непроцедурними, оскільки сама процедура пошуку розв'язку вбудована в мову (в її інтерпретатор). Вони використовуються, наприклад, для розв'язування задач штучного інтелекту (Lisp, Prolog) і дають змогу моделювати розумову діяльність людини в процесі пошуку розв'язків. До непроцедурних мов можна віднести і мови запитів систем керування базами даних (QBE, SQL).

Мови програмування можуть бути *компіляторами* або *інтерпретаторами.*

Компілятор (*compiler* від англ. *to compile* – збирати в ціле) – комп'ютерна програма (або набір комп'ютерних програм), що перетворює (компілює) програмний код, написаний певною мовою програмування (мова джерела – *source language*), на семантично еквівалентний код в іншій мові програмування (мова цілі – *target language*). Це, зазвичай потрібно для виконання програми на комп'ютері.

Інтерпретатор (*interpreter*) – програма чи технічні засоби, необхідні для виконання інших програм; вид транслятора, який здійснює пооператорне (покомандне) оброблення, перетворення у машинні коди та виконання програми або запиту (на відміну від компілятора, який транслює у машинні коди всю програму без її виконання).

Інтерпретатори можуть працювати як з вихідним кодом програми, написаним мовою програмування, так і з байт-кодом (інтерпретатори байт-коду).

Система програмування (*programming system*) – 1) те саме, що й інструментальна система; 2) система автоматичного програмування, що складається з мови програмування, компілятора або інтер-претатора програм, які написані цією мовою, відповідної документації, а також допоміжних засобів для підготовки програм до виконання.

Інструментальна система (*development environment*) – комплекс програмних або програмних і технічних засобів, який використовується фахівцями з програмування як інструмент для розроблення ПЗ (програм, програмних комплексів та систем тощо).

Від складання програми до її виконання комп'ютером – досить тривалий процес, що здійснюється спеціальними службовими програмами, що складають систему автоматизації програмування. З часом слово «автоматизація» випущено із наведеного словосполучення, в результаті чого воно перетворилося на систему програмування. Ця система складається з кількох компонент, а саме: препроцесора (*preprocessor*), компілятора (*compiler*), компонувальника (*linker*), налаштувача (*debugger*), об'єднаних спільним інтерфейсом в універсальне середовище розроблення програм.

Головна особливість підготовки програми до виконання полягає у використанні багатьох різнорідних складових частин. Деякі з них підготовлені заздалегідь; вони зберігаються в бібліотеках, системних або власних, інші складають частини програми, розміщені в різних файлах. У мові **C++** лише поглибилася тенденція винесення значної частини мови на рівень бібліотеки, яка була закладена ще авторами C. Зокрема на бібліотеку, а не на мову покладено відповідальність за зв'язок програми з ОС.

Одночасне використання багатьох файлів з текстами різних частин програми – роздільна компіляція – одне з найбільших досягнень системи прграмування. Завдяки йому поділяють програму на файли, які називаються одиницями трансляції (translation unit), групуючи в одному файлі тісно пов'язані між собою частини програми. У такий спосіб великі за розмірами програми діляться на частини, якими легше керувати.

Програмування – це діяльність, яка потребує великої організованості. Тому під час складання програм дотримуються певних

правил, одне з яких полягає в розділенні визначень і обчислень між файлами двох типів: файлів заголовків (*header*) і файлів реалізації (*source file*). Файли заголовків обробляються препроцесором, файли реалізації готуються препроцесором для подальшого оброблення компілятором. Тому їх називають початковими файлами. Сукупність взаємопов'язаних заголовних і початкових файлів складають вхідну програму для системи програмування. Вхідну програму розміщують у програмному проекті, у межах якого система програмування будуватиме об'єктні коди та виконавчу програму.

3.2.2. Класифікація систем програмування за надаваними можливостями

Системи програмування, так само, як і мови програмування, можна класифікувати за ступенем орієнтації на специфічні можливості. Зокрема, мови та системи програмування поділяють на машиннозалежні та машиннонезалежні.

До машиннозалежних належать машинні мови, асемблери і автокоди, що використовуються в системному програмуванні. Програма машиннозалежною мовою програмування може виконуватися лише на ЕОМ цього типу. Програма машиннонезалежною мовою після трансляції на машинну мову стає машиннозалежною. Ця ознака визначає мобільність створюваних програм (можливість перенесення на ЕОМ іншого типу).

За ступенем деталізації алгоритму отримання результату системи та мови програмування поділяються на такі:

– мови низького рівня;

– мови високого рівня;

– мови надвисокого рівня.

Рівень мови програмування визначається на основі складності елементів мови, з яких будується програма. Найнижчим рівнем є машинна мова, а найвищим, мабуть, рівень природної мови людини, якою формулюється задача типу «Дано:. . . треба:. . . ». До мов низького рівня належать машинні мови, асемблери та автокоди. До мов високого рівня належать мови, які мають типи даних (наприклад, масиви) та програмні конструкції (наприклад, цикли), що безпосередньо не реалізовуються машинною мовою. Взагалі, одному оператору мови високого рівня відповідає певна програма машинною мовою. Традицій-

но мовами високого рівня є мови класу FORTRAN, PL/1, Pascal, C, Basic, Ada. До мов надвисокого рівня належать мови, які мають найбільш абстрактні механізми опису завдання та вбудовані засоби його розв'язання. Ці мови дають змогу зосередитися на самому завданні, а не на деталях його розв'язання. Такими мовами є мови класу Lisp, Prolog.

За ступенем орієнтації на розв'язання завдань певного класу системи програмування та мови програмування поділяються на проблемно-зорієнтовані та універсальні.

Проблемно-зорієнтовані системи програмування містять спеціалізовані засоби опису та розв'язання завдань певного класу. Універсальні системи програмування не містять таких засобів, дозволяють розв'язувати будь-які завдання, але більшими зусиллями.

За можливістю доповнення новими типами даних і операціями системи програмування поділяються на такі, що розширюються і на такі, що не розширюються.

Системи програмування, що дозволяють розширювати склад типів даних і операцій, фактично містять механізм адаптації мови для розв'язання певних завдань. Вони є замкненими системами програмування.

За можливістю керування реальними об'єктами і процесами системи програмування поділяються на системи реального часу та системи умовного часу.

Мови систем реального часу забезпечують створення систем керування реальними об'єктами, в яких час є важливим чинником. Однією з таких мов є мова Ada, що використовується в системах військового призначення США.

3.2.3. Основні функції та компоненти системи програмування

Однією з найпоширеніших мов програмування серед сучасних мов високого рівня, що використовуються в ПК, є мова Visual C++.

Слово Visual означає, що за допомогою цієї мови реалізовано візуальний стиль програмування. Це зовсім новий стиль, за якого програми не пишуть, а проектують. Програмістів, які використовують його, доцільніше називати інженерами-проектувальниками програмних

засобів, оскільки перед тим, як почати набирати перший рядок коду, створюється інтерфейс, тобто проектується зовнішній вигляд робочого середовища, за яким працюватиме користувач ПК.

C++ у назві мови свідчить про те, що остання є розвитком давно відомої мови C++, яка, в свою чергу, є об'єктно-орієнтовною модифікацією популярної мови C. Ця мова здобула популярність своєю потужністю та продуманістю реалізації.

Мова C++ динамічно розвивається разом з комп'ютерами та комп'ютерними технологіями. Крім того, конструкції цієї мови універсальні як для різних типів комп'ютерів, так і різних ОС.

Для зміни однозадачних ОС (типу Windows) потрібен принципово новий підхід до розроблення програм у багатовіконному середовищі, що полягає не тільки в написанні тексту програми, а, що головне, в наявності графічного інструмента розроблення, який може працювати в середовищі системи Windows, створювати додатки, здатні використати всі переваги графічних, мультимедійних, діалогових і багатопроцесорних можливостей ОС Windows. У зв'язку з цим створено нове середовище програмування Visual C++. Саме завдяки відмінним візуальним засобам розроблення прикладних програм система й дістала таку назву.

Попри те, що мова Visual C++ суттєво відрізняється від мови C++, але покладений в її основу принцип гнучкості та точності залишився незмінним.

Це мова, на яку «роблять ставку» майже всі провідні фірми в галузі розроблення комп'ютерних технологій. Нові версії мови Visual C++ використовуються для розроблення системного ПЗ.

Visual C++ – це сучасна потужна система програмування, яку складає низка компонентів.

Одним із компонентів будь-якої сучасної системи програмування, у тому числі системи Visual C++ є інтегроване середовище – оболонка, що включає вбудований редактор тексту, систему інформаційної контекстуальної допомоги, транслятор-компілятор, компонувальник і настроювач програм, а також елементи, призначені для користувача інтерфейсу.

Безпосередня взаємодія програміста з інтегрованим середовищем здійснюється за допомогою засобів керування, розміщених у головному вікні проекту. Це вікно з'являється на екрані монітора

щоразу під час запуску Visual C++. Розглянемо основні його елементи.

Меню *Edit* містить відомі команди редагування: *Cut* – вирізати; *Copy* – копіювати; *Paste* – уставити; *Find* – знайти; *Delete* – вилучити; *Can't Undo* – скасувати; *Can't Redo* – повернути; *Select All* – виділити все.

Меню *View* містить команди, що дають змогу розкрити в головному вікні проекту інструментальні вікна, необхідні для розроблення на налаштування додатка. Основні з них наведено в табл. 3.1.

Таблиця 3.1

Основні вікна меню *View*

Ім'я команди	Назва вікна, що розкривається
Code	Вікно редактора коду
Object	Вікно екранної форми
Properties Windows	Вікно властивостей
Project Explorer	Вікно провідника проекту
Toolbox	Вікно елементів керування
Object Browser	Вікно характеристик об'єкта
Form Layout	Вікно розміщення форми

Зазначені в табл. 3.1 команди з меню *View* дають можливість розкрити в головному вікні проекту сім вікон.

У першому з розкритих вікон розміщується екранна форма, яка проектується, у другому – стандартний набір піктограм (інструментів) для створення на екранній формі об'єктів керування, у третьому – опис структури проекту, що створюється, у четвертому – список властивостей активного (виділеного) об'єкта.

За допомогою команд із меню Windows можна змінювати порядок розміщення вікон у головному вікні: горизонтально, вертикально або каскадно.

Вікно елементів керування *Toolbox* забезпечує проектувальника набором інструментів, необхідних для розроблення прикладної програми під час розміщення елементів керування на екранній формі.

Елементи керування – це візуальний засіб для створення об'єктів на формі, наприклад, рисунки, написи, кнопки керування, списки, смуги прокручування, меню і геометричні фігури.

3.3. Засоби створення програм

Будь-який компілятор є складовою частиною системного ПЗ. Основне призначення компіляторів – розроблення нових прикладних і системних програм за допомогою мов високого рівня.

Програма, як системна, так і прикладна, має етапи життєвого циклу, починаючи від проектування і закінчуючи впровадженням та супроводом. Компілятори – це засоби для створення ПЗ на етапах кодування, тестування й налагодження.

Однак сам по собі компілятор не вирішує повністю всіх завдань, пов'язаних з розробленням нової програми. Засобів одного компілятора недостатньо для того, щоб забезпечити проходження програмою етапів життєвого циклу. Тому компілятори – це програмне забезпечення, що функціонує в тісній взаємодії з іншими технічними засобами, що застосовуються на кожному етапі створення ПЗ.

Стандартний набір засобів системного програмування: текстовий редактор, компілятор, редактор зв'язків (збирач), бібліотеки стандартних функцій.

У зручній *інтегрованій системі* обов'язково міститься текстовий редактор, що має засоби виділення ключових слів, команд та інших операцій різними кольорами й шрифтами. Він може містити підказки для програмістів – формати команд, приклади, основні помилки та ін. Усі етапи створення програми в ній автоматизовані: після введення вихідного тексту програми його компіляція і складання здійснюються одним натисканням клавіші.

У сучасних інтегрованих системах є ще один компонент – *налагоджувач (debugger)*. Він дозволяє аналізувати роботу програми по кроках під час її виконання, спостерігаючи, як змінюються значення різних змінних.

Таким чином, комлект інтегрованої системи програмування для створення програми вибраною мовою програмування має такі компоненти:

– текстовий редактор;

– компілятор;

– редактор зв'язків;

– бібліотеку функцій.

Текстовий редактор використовується для формування вихідного тексту програми. Можна використовувати будь-який ре-

дактор, але краще використовувати спеціалізовані редактори, які орієнтовані на конкретну мову програмування, тобто мають відповідні можливості редагування, підказки та форматування коду програми.

Вихідний текст програми переводиться в машинний код за допомогою програми-компілятора. Якщо знайдено синтаксичні помилки, то результуючого кода створено не буде.

На цьому етапі можна отримати готову програму, але дуже часто в ній не вистачає деяких компонентів і тому *компілятор видає проміжний об'єктний код* (із розширенням *obj*).

Вихідний текст великої програми завичай містить декілька модулів (файлів з вихідним текстом). Кожен модуль компілюється в окремий файл з об'єктним кодом, які потім об'єднуються в одне ціле. Крім того, потрібно додати машинний код підпрограм, що реалізують різні стандартні функції. Такі функції містяться в бібліотеках (файлах із розширенням *lib*). Об'єктний код обробляється спеціальною програмою – *редактором зв'язків*, який об'єднує об'єктні модулі в готовий програмний продукт – *виконавчу програму* (*виконавчий код*).

Виконавчий код – це готовий програмний продукт, який можна запустити. Цей файл має розширення *exe*, *com*.

Зазвичай створюють інтегровані системи програмування, що містять усі компоненти.

Основним модулем системи програмування завжди є *компілятор*. Саме технічні характеристики компілятора, насамперед, впливають на ефективність результуючих програм, породжуваних системою програмування.

Крім основного компілятора, більшість систем програмування можуть містити у своєму складі ряд інших компіляторів. Так, більшість систем містять компілятор з мови асемблеру й компілятор із вхідної мови опису ресурсів. Але вони рідко безпосередньо взаємодіють із користувачем.

3.3.1. Розвиток систем програмування

Спочатку компілятори являли собою відособлені програмні модулі, що вирішують винятково завдання перекладу вихідного тексту програми з вхідної мови мовою машинних кодів. Компілято-

ри розроблялися поза зв'язком з іншими технічними засобами, з якими їм доводилося взаємодіяти. Розробник програми повинен забезпечувати взаємозв'язок усіх використовуваних технічних засобів:

– подати вхідні дані у вигляді тексту вихідної програми на вхід компілятора;

– отримати від компілятора результати його роботи у вигляді набору об'єктних файлів;

– подати весь набір отриманих об'єктних файлів разом з необхідними бібліотеками підпрограм на вхід збирачу програм;

– одержати від збирача єдиний виконавчий файл програми і підготувати його до виконання за допомогою завантажувача;

– поставити програму на виконання, у разі потреби використати настроювач для перевірки правильності виконання програми.

Усі ці дії виконувалися за допомогою послідовності команд, що ініціювали запуск відповідних програмних модулів з передачею їм всіх необхідних параметрів. Параметри передавалися кожному модулю в командному рядку і являли собою набір імен файлів і налаштувань, реалізованих у вигляді спеціальних ключів. Користувачі могли виконувати ці команди послідовно вручну, а з розвитком засобів командних процесорів ОС стали поєднувати їх у командні файли.

Згодом розробники компіляторів доклали зусиль, щоб полегшити працю користувачів, надавши їм необхідну кількість програмних модулів у складі однієї поставки компілятора. Тепер компілятори поставлялися вже разом з усіма необхідними супровідними технічними засобами. Крім того, були уніфіковані формати об'єктних файлів і файлів бібліотек підпрограм. Тепер розробники, маючи компілятор від одного виробника, могли в принципі користуватися бібліотеками й об'єктними файлами, отриманими від іншого виробника компіляторів.

Для написання командних файлів компіляції була запропонована спеціальна командна мова Makefile. Вона дозволяла у досить гнучкій і зручній формі описати весь процес створення програми від породження вихідних текстів до підготовки її до виконання. Це був зручний, але досить складний технічний засіб, що вимагав від розробника високого ступеня підготовки й професійних знань, оскільки сама командна мова Makefile була за складністю порівнянна

із простою мовою програмування. Мова Makefile стала стандартним засобом, єдиним для компіляторів всіх розробників.

Така структура засобів розроблення існувала досить довгий час, а в деяких випадках вона використовується й нині (особливо під час створення системних програм). Її поширення було пов'язане з тим, що сама по собі вся ця структура засобів розроблення була дуже зручною для пакетного виконання програм на комп'ютері, що сприяло її повсюдному застосуванню в епоху mainframe.

Наступним кроком розвитку засобів розроблення стала поява інтегрованого середовища розроблення. Інтегроване середовище об'єднало в собі можливості текстових редакторів вихідних текстів програм і командну мову компіляції. Користувач (розробник вихідної програми) тепер не повинен був виконувати всю послідовність дій від породження вихідного коду до його виконання, від нього також не вимагалося описувати цей процес за допомогою системи команд у Makefile. Тепер йому було досить тільки вказати в зручній інтерфейсній формі склад необхідних для створення програми вихідних модулів і бібліотек. Ключі для компілятора й інших технічних засобів також задавалися у вигляді інтерфейсних форм налаштування.

Після цього інтегроване середовище розроблення саме автоматично підготувало всю необхідну послідовність команд Makefile, виконувало їх, отримувало результат і повідомляло про помилки, якщо вони є. Причому сам текст вихідних модулів користувач міг змінити тут же, не перериваючи роботу з інтегрованим середовищем, щоб потім за необхідності просто повторити весь процес компіляції.

Створення інтегрованих середовищ розроблення стало можливим завдяки бурхливому розвитку ПК і появі розвинених засобів інтерфейсу користувача (спочатку текстових, а потім і графічних). Поява їх на ринку визначила подальший розвиток такого роду технічних засобів. Мабуть, першим вдалим середовищем такого роду можна визнати інтегроване середовище програмування Turbo Pascal на основі мови Pascal виробництва фірми Borland. Її широка популярність визначила той факт, що згодом всі розробники компіляторів звернулися до створення інтегрованих засобів розроблення для своїх продуктів.

Розвиток інтегрованих середовищ трохи знизив вимоги до професійних навичок розробників вихідних програм. Тепер у найпростішому випадку розробникам потрібно знати тільки вихідну мову (її синтаксис й семантику). Створюючи прикладну програму, розробник міг навіть не бути обізнаним з архітектурою цільової обчислювальної системи.

Подальший розвиток засобів розроблення також тісно пов'язаний з поширенням розвинених засобів графічного інтерфейсу користувача. Такий інтерфейс став невід'ємною складовою частиною багатьох сучасних ОС і графічних оболонок, а згодом – стандартом дефакто майже у всіх сучасних прикладних програмах.

Ці системи включали, крім вбудованого редактора текстів, підсистеми роботи з файлами, систему допомоги, підсистеми керування компіляцією й редактор зв'язків, компілятор, убудований налагоджувач. Це не могло не позначитися на вимогах, запропонованих до засобів розроблення ПЗ. У їх склад були спочатку включені відповідні бібліотеки, що забезпечують підтримку розвиненого графічного інтерфейсу користувача і взаємодія з функціями API (application program interface, прикладний програмний інтерфейс ОС). А потім для роботи з ними потрібні були додаткові засоби, які забезпечували б розроблення зовнішнього вигляду інтерфейсних модулів. Така робота вже більш характерна для дизайнера, аніж для програміста.

Для опису графічних елементів програм потрібні були відповідні мови. На їхній основі склалося поняття *ресурси* (*resources*) *прикладних програм*.

Ресурсами прикладної програми називають множину даних, що забезпечують зовнішній вигляд інтерфейсу користувача цієї програми, і не зв'язаних прямо з логікою виконання програми. Характерними прикладами ресурсів є:

– тексти повідомлень, що видаються програмою;

– кольорова гама елементів інтерфейсу;

– напис на таких елементах, як кнопки й заголовки вікон та ін.

Для формування структури ресурсів, у свою чергу, потрібні були редактори ресурсів, а потім і компілятори ресурсів, що обро-

бляють результат їх роботи. Ресурси, отримані з виходу компіляторів ресурсів, стали оброблятися компоновниками й завантажниками.

Будь-яка система програмуваннння може працювати тільки у відповідній ОС, для якої вона й створена, однак при цьому вона може дозволяти розробляти ПЗ й для інших ОС.

3.3.2. Структура сучасної системи програмування

Системою програмування називають весь комплекс ПЗ, призначених для їх кодування, тестування й налаштування. Нерідко системи програмування й інші технічні засоби, використовувані для створення ПЗ на більше ранніх етапах життєвого циклу (від формулювання вимог і аналізу до проектування) є взаємозалежними.

Структуру сучасної СП можна подати у вигляді схеми (рис. 3.5).

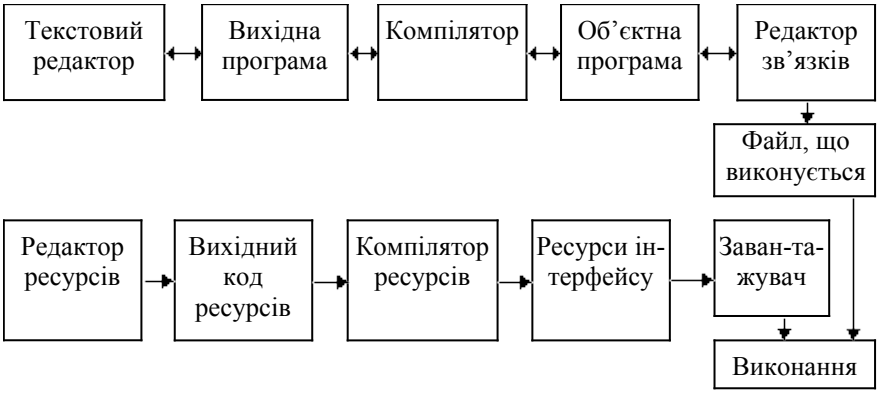


Рис 3.5. Структура системи програмування

Системи програмування в сучасному світі домінують на ринку засобів розроблення. Майже всі фірми-розробники компіляторів поставляють продукти в складі відповідної системи програмування у комплексі всіх інших технічних засобів. Окремі компілятори є рідкістю і, як правило, використовуються тільки для вузькопрофільних цілей. Тенденція така, що розвиток системи програмування спрямований на неухильне підвищення їхньої кооперації і сервісних можливостей. Це зумовлено тим, що на ринку лідирують ті

системи програмування, які дозволяють істотно знизити трудовитрати, необхідні для створення ПЗ на етапах життєвого циклу, пов'язаних з кодуванням, тестуванням і налагодженням програм. Показник зниження трудозатрат вважається більш істотним, ніж показники, що визначають ефективність результуючої програми, побудованої за допомогою системи програмування.

Як основні тенденції розвитку сучасних систем програмування варто відзначити впровадження в них засобів розроблення на основі мов четвертого покоління 4GL (four generation languages), а також підтримання систем швидкого розроблення ПЗ RAD (rapid application development).

Мови четвертого покоління 4GL являють собою широкий набір засобів, орієнтованих на проектування й розроблення ПЗ. Вони будуються на основі оперування не синтаксичними структурами мови й описами елементів, а їхніми графічними візуальними образами. На такому рівні проектувати й розробляти прикладне ПЗ може користувач, що не є кваліфікованим програмістом, але має уявлення про предметну галузь, на яку орієнтовано прикладну програму. Мови четвертого покоління є наступним (четвертим) етапом розвитку систем програмування.

Опис програми, побудованої на базі мов 4GL, транслюється потім у вихідний текст і файл опису ресурсів інтерфейсу, що являють собою звичайний текст відповідною вхідною мовою високого рівня. З цим текстом вже може працювати професійний програміст-розробник: він може коригувати й доповнювати його необхідними функціями. Такий підхід дозволяє розділити роботу проектувальника, відповідального за загальну концепцію всього проекту створюваної системи, дизайнера, відповідального за зовнішній вигляд інтерфейсу користувача, і професійного програміста, що відповідає безпосередньо за створення вихідного коду створюваного ПЗ.

У цілому мови четвертого покоління вирішують уже більш широкий клас завдань, ніж традиційні системи програмування. Вони становлять частину засобів автоматизованого проектування й розроблення ПЗ, що підтримують всі етапи життєвого циклу CASE-систем.

Приклади сучасних систем програмування

– Turbo Pascal, Borland Pascal, Borland Delphi, Borland C++ Builder.

– Microsoft Visual Basic, Microsoft Visual C++. Новітньою є система, побудована на базі мови C#, і системи, орієнтовані на концепцію .NET.

– Системи програмування мови С для ОС Linux і UNIX (функції завантажника виконуються самою ОС) тривалий час не потребували інтегрованого середовища й цілком могли обходитися командними файлами компіляції. Однак стали створюватися системи програмування, побудовані на базі інтегрованих середовищ розроблення. Вони будуються передусім у графічному середовищі на базі стандартного графічного інтерфейсу користувача у середовищі Windows.

3.3.3. Середовища швидкого проектування

В останні роки в програмуванні намітився візуальний підхід (особливо в програмуванні для ОС Windows). Достатньо глянути на вікно будь-якої Windows-програми, яке містить багато стандартних елементів керування (кнопки, пункти меню, списки, перемикачі й т. ін.). Звичайними мовами програмування четвертого покоління описувати процес створення цих елементів – дуже трудомістка операція. Тому процес створення стандартних елементів керування за допомогою графічних додатків автоматизований в середовищах швидкого проектування. Усі необхідні елементи оформляються за допомогою готових візуальних компонентів, які мишкою переміщуються у вікно, де створюється проект. Таким чином, вихідний текст програми формується автоматично, а це дозволяє зосередитися тільки на логіці завдання, яке розв'язується. У результаті програмування замінюється на проектування – такий підхід називається візуальним програмуванням. Таким чином, створені системи є системами *швидкого проектування*. У проектоване вікно готові візуальні компоненти перетягуються за допомогою мишки, потім властивості й поводження компонентів настроюються за допомогою редактора. Вихідний же текст програми, відповідальний за роботу цих елементів, генерується автоматично за допомогою *середовища швидкого проектування*, що називається *RAD-середовищем*. Подібний підхід називається *візуальним програмуванням (visual programming)*.

3.3.4. Основні системи програмування

Основні мови програмування та візуальні середовища наведено в таблиці 3.2.

Таблиця 3.2

Мови програмування та візуальні середовища

Найбільш популярні мови програмування	Відповідні візуальні середовища швидкого проектування програм для Windows
Бейсік (Basic)	Microsoft Visual Basic
Паскаль (Pascal)	Borland Delphi
Сі++ (C++)	Microsoft Visual C++
Ява (Java)	Java: Borland JBuilder

Компоненти можуть створюватися самостійно, із них потім формуються бібліотеки компонентів. Такий компонентний підхід до створення програми вважається дуже перспективним, оскільки без зайвих зусиль і на законних підставах допускає повторне використання чужої праці.

3.3.5. Системи програмування компанії Borland/Inprise

Популярність і поширеність цих систем програмування визначила, насамперед, простота їх використання, оскільки саме в системах цієї компанії були вперше реалізовані на практиці ідеї інтегрованого середовища програмування.

Turbo Pascal. Система програмування Turbo Pascal була створена компанією Borland на основі розширення мови Pascal, що отримала назву Borland Pascal. Звідси походить й сама назва системи програмування.

Компанія Borland побудувала й реалізувала ефективний однопрохідний компілятор мови Borland Pascal. За рахунок цього в системі програмування вдалося домогтися високої швидкості компіляції вихідних програм. Для пришвидшення роботи компоновника компанією Borland був запропонований власний унікальний формат об’єктних файлів модулів вихідної програми TPU (Turbo Pascal Unit). Із цієї причини модулі, створені в системі програмування Turbo Pascal, не могли використовуватися в інших системах.

У склад Turbo Pascal, крім компілятора з мови Borland Pascal, входив також компілятор з мови асемблеру (а з появою можливості розроблення результуючих програм для середовища Microsoft Windows компілятор ресурсів). Середовище програмування дозволяло компонувати як єдині виконавчі файли, що є і оверлейними програмами для ОС типу MS DOS.

Спочатку система програмування Turbo Pascal будувалася на основі бібліотеки RTL (Run Time Library) мови Borland Pascal. Ця бібліотека не надавала користувачу широкого набору функцій, вона тільки реалізовувала базові математичні функції й функції мови. Однак можна виокремити одну характерну особливість цієї бібліотеки – вона включала у свій склад об'єктний код менеджера пам'яті для керування розподілом динамічної пам'яті, що автоматично підключалася до кожної результуючої програми, створеної за допомогою даної системи програмування.

Незважаючи на недоліки, Turbo Pascal набула поширення й завоювала своє місце на ринку, оскільки ця система вперше була побудована у вигляді інтегрованого середовища. Цей факт визначив її широке поширення, і, насамперед, в університетському середовищі, де були потрібні прості й зрозумілі для використання засоби розроблення.

Перші версії системи програмування були орієнтовані тільки на функціонування ОС MS DOS для ПК на базі процесорів типу Intel 80×86. На виконання в середовищі цієї ОС були орієнтовані й результуючі програми, розроблювані за допомогою цього середовища програмування.

Система програмування Turbo Pascal набула поширення й подальшого розвитку. Компанія Borland створила декілька їх реалізацій (найпоширеніші з них версії 5.5 і 7.0). Останні реалізації системи програмування могли створювати результуючі програми, орієнтовані на роботу як в ОС типу MS DOS, так і в середовищі Microsoft Windows. Вони реалізовували основні переваги, надані інтегрованим середовищем програмування, такі, як лексичний аналіз програм під час виконання і вбудовані контекстні підказки.

У міру поширення Turbo Pascal розроблялися бібліотеки підпрограм і функцій для неї. Були створені такі бібліотеки, як Turbo Professional (TP), Turbo Vision, Object Window Library (OWL) для середовища MS DOS і ObjectWindows для середовища Microsoft

Windows. Поширенню цих бібліотек, як і раніше, заважав той факт, що в Turbo Pascal використається унікальний, нестандартний формат об'єктних файлів. Відсутність стандарту мови Borland Pascal багато в чому стримувало розвиток цієї системи програмування й не сприяло її застосуванню як професійного засобу розроблення.

Системі програмування Turbo Pascal приділено багато уваги оскільки це одна з найпоширеніших тепер система програмування навчального призначення. Крім того, це перша система, що з'явилася на ринку і повністю реалізувала в собі ідеї інтегрованого середовища програмування. Ці ідеї, закладені в Turbo Pascal, знайшли застосування в багатьох сучасних системах програмування.

Borland Delphi. Система програмування Borland Delphi стала логічним продовженням і подальшим розвитком ідей, закладених компанією-розробником ще в Turbo Pascal.

Як основні в новій системі програмування можна виокремити такі принципові зміни:

– нова мова програмування Object Pascal, що виявилася істотним перероблennям колишньої версії мови Borland Pascal;

– компонентна модель середовища розроблення орієнтована передусім на технологію розроблення RAD.

Компонентна модель середовища розроблення передбачає створення основної частини програми у вигляді набору взаємозалежних компонентів класів об'єктно-орієнтованої мови. Під час розроблення вихідної програми (design time) компоненти набувають вигляду графічних зображень і позначень, зв'язаних між собою. Кожний компонент має певний набір властивостей (properties), подій (events) і методів (methods). Кожному з них відповідає свій фрагмент вихідного коду програми, що відповідає за оброблення методу або реакції на якусь подію. Розробник може розміщувати на екрані й об'єднувати компоненти, а також редагувати пов'язаний з ними вихідний код програми. Причому поводження компонентів під час виконання програми (run time) повністю визначається їхнім взаємозв'язком, вихідним кодом програми й об'єктним кодом самого компонента.

Система програмування Borland Delphi призначена для створення результуючих програм, що виконуються в середовищі ОС Windows різних типів.

Основу системи програмування Borland Delphi та її компонентної моделі становить бібліотека VCL (Visual Component Library). У цій бібліотеці реалізовано у вигляді компонентів основні органи керування й інтерфейсу ОС. Також у її склад входять класи, що забезпечують розроблення додатків для архітектури клієнт – сервер і трирівневої архітектури (у сучасних реалізаціях Borland Delphi). Розробник має можливість не тільки використовувати будь-які компоненти, що входять до складу бібліотеки VCL, але також розробляти свої власні компоненти, засновані на кожному із класів бібліотеки. Ці нові компоненти стають частиною системи програмування й згодом можуть бути використані іншими розробниками.

Для підтримання розроблення результуючих програм для архітектури сервер – сервер та клієнт – сервер до складу Borland Delphi входить засіб BDE (Borland Database Engine). Він забезпечує результуючим програмам можливість доступу до широкого діапазону серверів бази данних за допомогою класів бібліотеки VCL. За допомогою BDE результуюча програма може взаємодіяти із серверами баз данних типу Microsoft SQL Server, Interbase, Sybase, Oracle та ін. Система програмування Borland Delphi підтримує також створення результуючих програм, що виконуються в архітектурі сервер – сервер та клієнт – сервер, на базі інших технологій, наприклад ADO (Active Data Objects).

Borland C++ Builder. Система програмування Borland C++ Builder об'єднала в собі ідеї інтегрованого середовища розроблення, реалізовані компанією в системі програмування Turbo Pascal і Borland Delphi з можливостями мови програмування C++. Історія цієї системи починається з інтегрованого середовища розроблення Borland Turbo C.

Удале поширення систем програмування Turbo Pascal і Borland Delphi сприяло і впровадженню на ринок Borland C++ Builder від тієї ж компанії-розробника. Ця система займає міцну позицію на ринку засобів розроблення для мови C++, де існує досить жорстка конкуренція.

3.3.6. Системи програмування фірми Microsoft

Компанія Microsoft є в цей час виробником ОС і ПЗ і домінує на ринку ПК, побудованих на базі процесорів типу Intel 80×86. Це, насамперед, стосується до всіх варіантів ОС типу Microsoft Windows. Цей факт став однією з головних причин, які зумовили міцну позицію компанії на ринку засобів розроблення програмних продуктів для ОС типу Microsoft Windows. Усі види ОС типу Microsoft Windows створювалися як закриті системи. Тому безумовне знання компанією-розробником структури і внутрішнього устрою ОС найчастіше було визначальним у ситуації, коли треба було створити засіб розроблення додатків для цієї ОС.

Microsoft Visual Basic. Цей засіб розроблення має довгу історію під керівництвом компанії Microsoft. Історія мови Basic на ПК почалася із примітивних інтерпретаторів цієї мови. Сама по собі мова Basic дозволяла легко організовувати інтерпретацію вихідного коду програм, а його синтаксис і семантика досить прості для розуміння навіть непрофесійними розробниками.

Система програмування Microsoft Visual Basic також спочатку була орієнтована на інтерпретацію вихідного коду. Однак вимоги й умови на ринку засобів розроблення підштовхнули компанію-виробника до створення компілятора, що ввійшов до складу системи програмування. Основні функції бібліотеки мови були винесені в окрему бібліотеку, що підключається динамічно (VBRun), яка має бути в ОС для виконання результуючих програм, створених за допомогою системи програмування. Різні версії Microsoft Visual Basic орієнтовані на різні версії цієї бібліотеки. Інтерпретатор мови був збережений і впроваджений компанією-розробником до складу модулів іншого програмного продукту Microsoft Office.

Розвиток системи програмування Visual Basic потребував істотної зміни синтаксису й семантики самої мови.

Остання версія цієї системи Microsoft Visual Basic є одним з ефективних засобів для створення результуючих програм, орієнтованих на виконання під керуванням ОС типу Microsoft Windows. Ця система програмування орієнтована на технологію розроблення RAD. Microsoft Visual Basic містить інтегровані засоби візуальної роботи з базами даних, що підтримують проектування й доступ до баз даних SQL Server, Oracle та ін. До цих засобів належать Visual Database Tools, ADO/OLE DB, Data Environment Designer, Report Designer та ін.

У системі програмування підтримується також створення серверних Web-додатків, що працюють із будь-яким засобом перегляду на базі нових Web-класів. У новій версії забезпечується й налагодження додатків для сервера IIS (Internet Information Server) виробництва компанії Microsoft. У Microsoft Visual Basic можливе створення інтерактивних Web-сторінок.

Microsoft Visual Basic забезпечує просте створення додатків, орієнтованих на дані. Visual Basic 6.0 дозволяє створювати результуючі програми, виконувані в архітектурі клієнт – сервер , які можуть працювати з будь-якими базами даних. Система програмування Microsoft Visual Basic орієнтована насамперед на створення клієнтської частини додатків.

Тепер Visual Basic підтримує універсальний інтерфейс доступу до даних Microsoft за допомогою технології ADO. Visual Basic забезпечує перегляд таблиць, зміну даних, створення запитів SQL із середовища розроблення для будь-якої сумісної з ODBC або OLE DB бази даних. Так само, як і в редакторі Visual Basic, синтаксис SQL виділяється кольорами й негайно перевіряється на наявність помилок. Це робить код SQL більш зручним і менш схильним до випадкових помилок.

Нова версія продукту підтримує колективне розроблення, масштабованість, створення компонентів проміжного шару, придатних до багаторазового використання в будь-якому COM-сумісному продукті. Підтримання широкого спектру інтерфейсів доступу до даних дає можливість застосовувати цю систему програмування для розроблення клієнтської частини додатків, що виконуються в трирівневій архітектурі.

Розроблення ПЗ має багато нових можливостей, таких як виділення синтаксису й автоматичне завершення ключових слів. Система програмування Microsoft Visual Basic інтегрується у сім'ю програмних продуктів Microsoft BackOffice, що забезпечує середовище для виконання й створення складних додатків для підприємства для роботи в локальних мережах або Інтернеті. Використання нових інтегрованих візуальних засобів роботи з даними полегшує виконання рутинних завдань із забезпечення доступу до них; ці засоби доступні безпосередньо із середовища розроблення Visual Basic.

Система програмування Visual Basic поєднує в собі простоту й ефективність розроблення. Всі недоліки, властиві цій системі, здебільшого виникають через недоліки використовуваної вихідної мови програмування. Засоби мови Basic навіть після значної модифікації обмежують можливості її застосування в сучасних архітектурах взаємодії додатків, які значною мірою засновані на об'єктно-орієнтованому підході. Крім того, мова програмування в системі Visual Basic не є визнаним стандартом, а тому виникають труднощі з використання створених на його основі модулів і компонентів в інших засобах розроблення.

Microsoft Visual C++. Система програмування Microsoft Visual C++ являє собою реалізацію середовища розроблення для поширення мови системного програмування C++, виконану компанією Microsoft. Ця система програмування побудована у вигляді інтегрованого середовища розроблення, що включає в себе всі необхідні засоби для розроблення результуючих програм, орієнтованих на виконання під керуванням ОС типу Microsoft Windows різних версій.

Основу системи програмування Microsoft Visual C++ становить бібліотека класів MFC (Microsoft foundation classes). У цій бібліотеці реалізовано у вигляді класів C++ всі основні органи керування й інтерфейсу ОС. У її склад також входять класи, що забезпечують розроблення додатків для архітектури клієнт – сервер і трирівневої архітектури (у сучасних версіях бібліотеки). Система програмування Microsoft Visual C++ дозволяє розробляти будь-які додатки, що виконуються в середовищі ОС типу Microsoft Windows, у тому числі серверні або клієнтські результуючі програми, що взаємодіють між собою за однією з зазначених вище архітектур.

Класи бібліотеки MFC орієнтовані на використання технологій COM/DCOM, а також побудованої на їхній основі технології Active для організації взаємодії між клієнтською й серверною частинами розроблюваних додатків. На основі класів бібліотеки користувач може створювати власні класи в мові C++ та організовувати власні структури даних.

На відміну від систем програмування компанії Borland система програмування Microsoft Visual C++ орієнтована на використання стандартних засобів зберігання й оброблення ресурсів інтерфейсу користувача в ОС Windows. Це не дивно, оскільки всі версії ОС

типу Windows розробляються самою компанією Microsoft. Microsoft Visual C++ забезпечує всі необхідні засоби для створення професійних Windows-додатків. Від версії до версії продукт стає простішим для використання, розширюються можливості застосування, підвищується продуктивність.

Система програмування Microsoft Visual C++ витримала кілька реалізацій. Серед нових версій системи програмування було випущено й кілька версій бібліотеки MFC, на якій ґрунтується ця система.

Бібліотека MFC є реалізацією широкого набору класів мови C++, орієнтованого на розроблення результуючих програм під керуванням ОС типу Microsoft Windows. Це зумовлюється тим, що розробник бібліотеки компанія Microsoft одночасно є і розробником ОС типу Microsoft Windows, на які орієнтований об'єктний код бібліотеки. Бібліотека може бути підключена до результуючої програми за допомогою звичайного компоновника, або використовуватися як динамічна бібліотека, що підключається до програми під час її виконання. Бібліотека MFC є досить поширеною. Її можна використовувати не тільки в складі системи програмування виробництва компанії Microsoft, але й у системах програмування інших виробників.

3.4. Архітектура програмних систем. Основні системи програмування

Проектування програмних систем – ітераційний процес, за допомогою якого вимоги до програмної системи транслюють в інженерні. Проектування має дві стадії:

1) попереднє проектування;

2) детальне проектування.

Попереднє проектування формулює абстракції архітектурного рівня, а детальне проектування уточнює ці абстракції, додає деталі алгоритмічного рівня. Крім цього, у багатьох випадках відокремлюють ще й інтерфейсне проектування, мета якого – формування графічного інтерфейсу користувача (GUI, рис. 3.6).

Попереднє проектування містить етап декомпозиції підсистем на модулі – кожна підсистема розбивається на модулі та визначаються типи модулів і з'єднання між модулями.

Рис. 3.6. Інформаційні зв'язки процесу проектування

3.4.1. Архітектура програмної системи

Архітектуру має будь-яка система програмування, незалежно від того, чи розроблялась архітектура цілеспрямовано, чи ні. Архітектура програмної системи – це розміщення та взаємодія її модулів. Попри значущість програмної архітектури, не завжди на ній акцентується увага, оскільки її визначення не завжди піддається чіткому формулюванню, концепція не достатньо виразна (між керуванням вимогами та поняттям системи), немає узагальненого способу подання архітектури і не описано процесу її розроблення.

Разом із тим без програмної архітектури або неякісне її виконання є основним технічним ризиком програмних проектів.

Описувати систему треба таким чином, щоб зацікавлені особи (розробники, програмісти, користувачі, замовники) могли:

– розуміти, що робить система та як вона працює;

– попрацювати з частиною системи (зокрема, використати її повторно);

– розширити систему.

Якщо цей опис займає не більше 60 сторінок, то це й буде описом програмної архітектури.

Іноді з опису вилучають частини. *Архітектура* – це коли вилучити більше нічого не можна, щоб система лишалася зрозумілою.

Модель – це архітектура (моделі складних систем можуть бути дуже великими).

Більшість визначень програмної архітектури ґрунтуються на таких поняттях програмної системи:

– статична структура (елементи та взаємозв’язок між ними);

– динамічна структура (відношення, що є динамічними аспектами);

– композиція чи декомпозиція (підсистеми, модулі);

– компоненти та їх взаємодія;

– рівні та їх взаємодія;

– організація фізичного розгортання елементів;

– деякі обмеження (оточення, мова програмування тощо);

– стиль, що визначає розроблення та розвиток;

– функціональні можливості;

– інші аспекти (повторне використання, продуктивність, масштабованість);

Архітектурно значущий елемент справляє вплив на структуру системи та її продуктивність, стійкість, можливість розвитку та модульного нарощування.

Архітектурно значущими елементами є:

– основні класи;

– архітектурні механізми, що визначають поведінку класів, зокрема, механізми зв'язку;

– шаблони та контури;

– рівні та підсистеми;

– інтерфейси та компоненти;

– основні процеси чи потоки керування.

Архітектура об'єднує значущі рішення стосовно:

– організації системи програмування;

– вибору структурних елементів та їх інтерфейсів, за допомогою яких система об'єднується в одне ціле, а також поведінки цих інтерфейсів, об'єднаної сумісним функціонуванням елементів;

– об'єднання цих елементів у підсистеми, що поступово збільшуються;

– архітектурного стилю, що направляє описану структуру, елементи, їх інтерфейси, їх сумісну роботу та об'єднання.

Проте програмна архітектура пов'язана не тільки зі структурою та поведінкою системи програмування, але й з її контекстом: використанням, функціональними можливостями, продуктивністю, еластичністю (гнучкістю), повторним використанням, можливістю

розуміння, економічними й технологічними обмеженнями та компромісами, а також з питаннями естетики.

Наведені різні визначення програмної архітектури відображають складність поняття «програмна архітектура».

Зрозуміло, що для різних зацікавлених сторін важливими є різні аспекти програмної архітектури. Як наслідок, використовуються різні подання однієї й тієї ж програмної архітектури.

Архітектура:

– спрощує розуміння системи програмування;

– дозволяє отримати повний інтелектуальний контроль на всіх етапах життєвого циклу системи програмування, забезпечуючи її гнучкість та адаптивність, спрощуючи розроблення та супровід;

– надає ефективну основу широкомасштабного повторного використання;

– уможливлює керування проектом (наприклад, організація планування, кадрове забезпечення – за рівнями, підсистемами).

Таким чином, опис програмної архітектури можна зобразити у вигляді рисунка (рис. 3.7).

Рис. 3.7. Опис програмної архітектури

За сучасними стандартами для опису програмної архітектури використовують найчастіше графічне зображення програмної архітектури. Відокремлюють такі класи графічних позначень:

Компонент – абстрактна одиниця, яка забезпечує оброблення інформації в системі за певним інтерфейсом (обчислення, переведення в інший формат, запис інформації тощо). Поведінка (функції) компонентів описується з урахуванням їх впливу на інші компоненти.

Зв'язок – абстрактний механізм, що забезпечує передавання даних, керування та взаємодію компонентів між собою. Прикладами зв'язків є виклики процедур (модулів), протоколи передавання даних, потоки даних.

Дані – елементи інформації, що передаються між компонентами. Виділяють такі типи даних: файли, записи, повідомлення, аргументи функцій, масиви.

Кожна архітектура характеризується своїми властивостями, серед яких виділяються надійність, складність, можливість реор-ганізації, функціональна гнучкість та функціональні властивості.

3.4.2. Найпопулярніші архітектури програмних систем

Серед архітектур системи програмування, що найчастіше використовуються в сучасному програмуванні можна виділити такі:

– модель сховища даних;
– потік даних (послідовна архітектура);
– об'єктна модель (стиль виклик-повернення);
– модель клієнт – сервер;
– трирівнева модель;
– модель віртуальної (абстрактної) машини.

У моделі сховища даних (рис. 3.8) підсистеми розділяють дані, що містяться в загальній пам'яті. Як правило, подібні дані утворюють бази даних. Передбачається система керування базами даних (СКБД).

Модель потоку даних використовують у простих системах, що виконують послідовні обчислення чи оброблення даних.

Об'єктна модель полягає в тому, що створюється один або декілька модулів (об'єктів), що взаємодіють один з одним за допомогою викликів один одного.

Модель клієнт – сервер використовують для розподілених систем, у яких дані розподілені по серверах (рис. 3.9). Для передавання даних використовують мережевий протокол, наприклад TCP/IP.

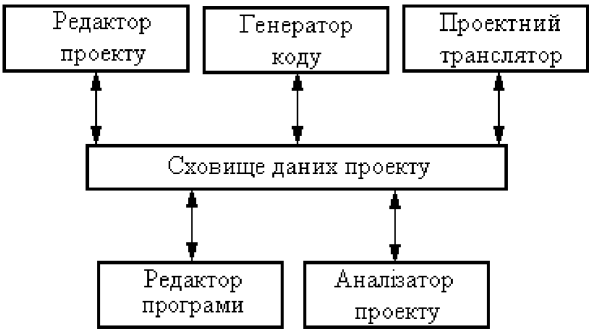


Рис. 3.8. Модель сховища даних

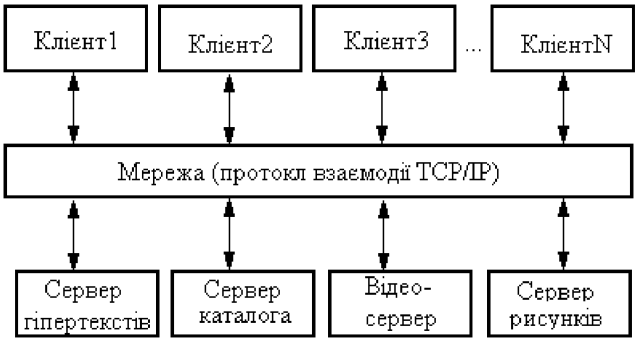


Рис. 3.9. Модель клієнт-сервер

Трирівнева модель є моделлю розвитком моделі клієнт – сервер (рис. 3.10).

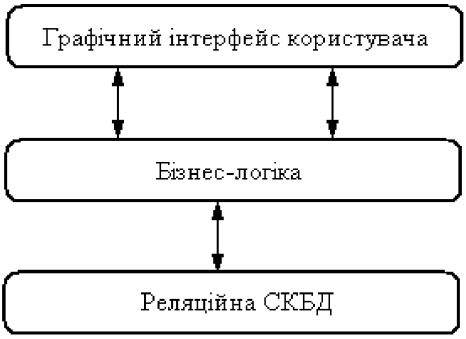


Рис. 3.10. Трирівнева модель

Рівень графічного інтерфейсу користувача запускається на машині клієнта. Бізнес-логіку утворюють модулі, які здійснюють функціональні обов'язки системи. Цей рівень запускається на сервері додатка. Реляційна СКБД зберігає дані, які необхідні для рівня бізнес-логіки. Цей рівень запускається на іншому сервері – сервері бази даних.

Переваги трирівневої моделі:

– спрощується така модифікація рівня, яка не впливає на інші рівні;

– відокремлення прикладних функцій від функцій керування базами даних спрощує оптимізацію всієї системи.

Модель абстрактної машини віддзеркалює багатошарову систему (рис. 3.11).

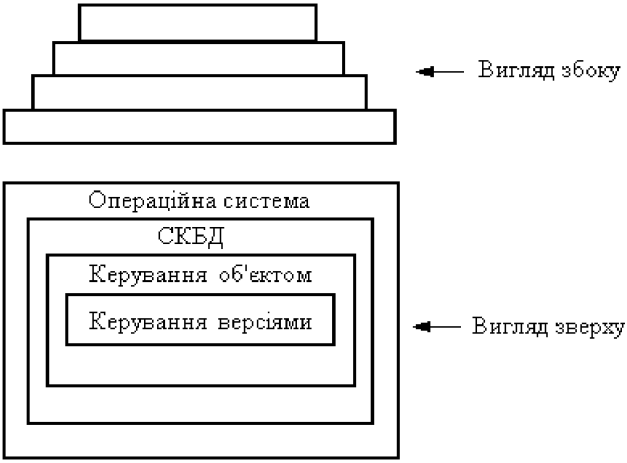


Рис. 3.11. Модель віртуальної машини

У такій моделі кожний шар реалізується з використанням засобів, які забезпечуються шаром-фундаментом.

3.4.3. Стандартизація мов програмування

Мову програмування можна подати у вигляді набору специфікацій, що визначають його синтаксис і семантику.

Для багатьох поширених мов програмування створено міжнародні стандарти. Спеціальні організації регулярно відновлюють і публікують специфікації і формальні визначення відповідної мови. У межах таких організацій триває розроблення й модернізація мов програмування й вирішуються питання про розширення або підтримання вже існуючих і нових мовних конструкцій.

Основні особливості архітектур мов програмування полягають в особливостях використання таких складових:

*Типи даних***.** Сучасні цифрові комп'ютери зазвичай є двійковими і дані зберігають у двійковому (бінарному) коді (хоча можливі реалізації й в інших системах числення). Ці дані, як правило, відображають інформацію з реального світу (імена, банківські рахунки, виміри й ін.), що являє собою високорівневі концепції.

Особлива система, за якою дані організовуються в програмі, – це система типів мови програмування; розроблення й вивчення

систем типів, яка відома як теорія типів. Мови можуть бути класифіковані як системи з типізацією і мови з типізацією.

Статично-типізовані мови можуть бути надалі підрозділені на мови з обов'язковою декларацією, де кожна змінна й оголошення функції має обов'язкове оголошення типу, і мови з виведеними типами. Іноді динамічно-типізовані мови називаються латентно-типізованими.

Структури даних. Системи типів у мовах високого рівня дозволяють визначати складні, складові типи, структури даних. Структурні типи даних утворюються як декартовий добуток базових (атомарних) типів і раніше визначених складних типів.

Основні структури даних (списки, черги, хеш-таблиці, двійкові дерева й пари) часто виражаються особливими синтаксичними конструкціями в мовах високого рівня. Такі дані структуруються автоматично.

Мова програмування – формальна знакова система, призначена для запису програм, що задають алгоритм у формі, зрозумілій для виконавця (наприклад, комп'ютера). Мова програмування визначає набір лексичних, синтаксичних і семантичних правил, що використовуються для складання комп'ютерної програми. Вона дозволяє програмісту точно визначити: на які події буде реагувати комп'ютер, як будуть зберігатися й передаватися дані, які саме дії варто виконувати над цими даними за різних обставин. Із часу створення перших програмувальних машин людство створило вже більше восьми з половиною тисяч мов програмування. Щороку їх кількість поповнюється новими. Деякими мовами вміє користуватися тільки невелика кількість їхніх власних розробників, інші стають відомі мільйонам людей. Професійні програмісти іноді застосовують у своїй роботі більше десятка різноманітних мов програмування.

Семантика мов програмування. Існує кілька підходів до визначення семантики мов програмування, та найбільш поширені три їх різновиди:

– операційний;
– денотаційний (математичний);
– деривaційний (аксіоматичний).

Під час описування семантики в межах операційного підходу звичайне виконання конструкцій мови програмування інтерпретується за допомогою деякої уявної (абстрактної) ЕОМ.

Деривaційна семантика описує наслідки виконання конструкцій мови за допомогою мови логіки й задання передумови і післяумови. *Денотаційна семантика* оперує поняттями, типовими для математики – множини, відповідності, судження, твердження й ін.

Мова програмування будується відповідно до тієї або іншої базової моделі обчислень і парадигми програмування.

Незважаючи на те, що більшість мов орієнтована на послідовну модель обчислень, що задається фон-неймановскою архітектурою ЕОМ, існують й інші моделі: мови зі стековою обчислювальною моделлю (Forth, Factor, Postscript і ін.), функціональне (Лісп, Haskell, ML і ін.) і логічне програмування (Пролог) і мова Рефал, заснована на моделі обчислень, яка введена радянським математиком А.А. Марковим-молодшим.

Водночас активно розвиваються проблемно-орієнтовані, декларативні й візуальні мови програмування.

3.4.4. Компільовані й інтерпретовані мови

Мови програмування можна поділити на компільовані й інтерпретовані.

Програма компільованою мовою за допомогою спеціальної програми компілятора перетворюється (компілюється) у набір інструкцій для цього типу процесора (машинний код) і далі записується в модуль, що виконується, і може запускатися на виконання як окрема програма. Інакше кажучи, компілятор перекладає вихідний текст програми з мови програмування високого рівня у двійкові коди інструкцій процесора. Зазвичай такі мови мають послідовну або об’єктну архітектуру.

Якщо програма написана інтерпретованою мовою, то інтерпретатор безпосередньо її виконує (інтерпретує) без попереднього перекладу. При цьому програма залишається вихідною мовою й не може бути запущена без інтерпретатора. Отже, процесор комп’ютера – це інтерпретатор машинного коду. Такий підхід є найбільш поширеним для послідовної та трирівневої архітектури.

Поділ на компільовані й інтерпретовані мови є трохи умовним. Так, для будь-якої традиційно компільованої мови, як, наприклад, Паскаль, можна написати інтерпретатор. Крім того, більшість сучасних «чистих» інтерпретаторів не виконують конструкції мови безпосередньо, а компілюють їх у деяке високорівневе проміжне подання (наприклад, з поіменуванням змінних і розкриттям макросів).

Для будь-якої інтерпретованої мови можна створити компілятор – наприклад, мова Ліспі, що є інтерпретованою, може компілюватися без жодних обмежень. Створюваний під час виконання програми код може так само динамічно компілюватися під час виконання.

Скомпільовані програми виконуються швидше й не потребують додаткових програм, оскільки вони вже перекладені машинною мовою. Разом з тим, з кожною зміною тексту програми потрібно її перекомпільовувати, що створює труднощі під час розроблення. Крім того, скомпільована програма може виконуватися тільки на тому типі комп'ютерів і з тією ж ОС, на яку був розрахований компілятор. Щоб створити виконавчий файл для машини іншого типу, потрібна нова компіляція.

Інтерпретовані мови мають деякі специфічні додаткові можливості, крім того, програми, виконані цими мовами, можна запускати відразу ж після зміни, що полегшує процес розроблення. Програма інтерпретованою мовою може бути запущена на різних типах машин і ОС без додаткових зусиль.

Однак інтерпретовані програми виконуються помітно повільніше, ніж компільовані, крім того, вони не можуть виконуватися без додаткової програми-інтерпретатора.

Деякі мови, наприклад, Java і C#, перебувають між компільованими й інтерпретованими. Зокрема програма компілюється не в машинну мову, а в машиннонезалежний код низького рівня – байт-код. Далі байт-код виконується віртуальною машиною. Для виконання байт-коду зазвичай використовується інтерпретація, хоча окремі його частини для пришвидшення функціонування програми можуть бути трансльовані в машинний код безпосередньо під час виконання програми за технологією компіляції «на льоту» (Just-in-time compilation, JIT). Для Java байт-код виконується віртуальною машиною Java (Java Virtual Machine, JVM), для C# – Common Language Runtime. Тут засто-

совується архітектура «віртуальної машини» або клієнт-серверна архітектура.

Подібний підхід у деякому змісті дозволяє використовувати переваги як інтерпретаторів, так і компіляторів. Варто згадати також оригінальну мову Форт (Forth), що має й інтерпретатор, і компілятор.

3.4.5. Процедурні мови програмування

Процедурне (імперативне) програмування є відбиттям архі-тектури традиційних ЕОМ, яка була запропонована фон Нейманом у 1940-х роках. Теоретичною моделлю процедурного програмування є алгоритмічна система, названа Машиною Т'юрінга.

Програма процедурною мовою програмування складається з послідовності операторів (інструкцій), що задають процедуру розв'язання завдання. Основним є оператор присвоювання, призначений для зміни вмісту пам'яті. Концепція пам'яті як сховища значень, уміст якого може обновлятися операторами програми, є фундаментальною в імперативному програмуванні.

Виконання програми зводиться до послідовного виконання операторів для перетворення вихідного стану пам'яті, тобто значень вихідних даних, у заключний, тобто в результати. Таким чином, з погляду програміста є програма й пам'ять, причому перша послідовно обновляє вміст останньої.

Процедурна мова програмування дає змогу програмісту визначати кожний крок у процесі розв'язання завдання. Особливість таких мов програмування полягає в тому, що завдання розбиваються на кроки й розв'язуються крок за кроком. Використовуючи процедурну мову, програміст визначає мовні конструкції для виконання послідовності алгоритмічних кроків.

Як приклад найбільш популярних систем програмування можна навести такі:

Асемблер. Мова низького рівня, що перетворює вихідні програми, написані мовою програмування, безпосередньо в коди машинних команд. Термін «асемблер» походить від англійського слова assembler (збирач частин в одне ціле). Частинами тут є оператори, а результатом збирання – послідовність машинних команд.

Асемблер – машиннозалежна мова, що відображає особливості архітектури конкретного типу комп'ютера. Вихідна програма, написана мовою асемблеру, складається з одного або декількох вихідних модулів, а кожний модуль – з операторів.

Basic. Бейсик (від BASIC, скорочення від Beginner's All-purpose Symbolic Instruction Code – універсальний код символічних інструкцій для початківців; basic– основний, базовий) – сім'я високорівневих мов програмування.

Pascal. Паскаль (Pascal) – високорівнева мова програмування загального призначення. Це одна з найбільш відомих мов програмування, широко застосовується в промисловому програмуванні, навчанні програмування у вищій школі, є базою для великої кількості інших мов.

Особливостями мови є чітка типізація і наявність засобів структурного (процедурного) програмування.

Сі. Сі (C) – стандартизована мова програмування, розроблена на початку 1970-х років співробітниками Bell Labs Кеном Томпсоном і Денисом Рітчі як розвиток мови Бі. Сі була створена для використання в ОС UNIX. Відтоді вона імпортована в інші ОС і стала однією з найбільш використовуваних мов програмування. Сі цінують за її ефективність; це найпопулярніша мова для створення системного та прикладного ПЗ. Незважаючи на те, що Сі не розроблялася для новачків, її активно використовують для навчання програмування. Надалі синтаксис мови Сі став основою для багатьох інших мов.

3.4.6. Об`єктно-орієнтовані мови програмування

Обєктно-орієнтована мова програмування – мова, побудована на принципах об`єктно-орієнтованого програмування.

В основу концепції об`єктно-орієнтованого програмування покладено поняття об'єкта – якоїсь субстанції, що поєднує в собі поля (дані) і методи (виконувані об'єктом дії). Наприклад, об'єкт «людина» може мати поля «ім'я», «прізвище» і мати методи «бути» і «спати».

Ідея об`єктно-орієнтованого програмування вперше була висунута в мові Smalltalk. В об`єктно-орієнтоване програмування введено поняття «об'єкту» й реалізовано механізми обчислень, що дозволяють:

– описувати структуру об’єкта;

– описувати дії з об’єктами;

– використовувати спеціальні правила спадкування об’єктів;

– установлювати різні ступені захисту компонентів об’єктів і визначити різні права доступу до них.

Становленню об’єктно-орієнтовного програмування значною мірою сприяв розвиток функцій машинної графіки.

Серед найбільш популярних мов об’єктно-орієнтованого програмування можна виділити такі:

– Сі++. Компільована статично типізована мова програмування загального призначення. Підтримує різні парадигми програмування, але порівняно з її попередником – мовою Сі – найбільша увага приділена підтриманню об’єктно-орієнтованого й узагальненого програмування.

– JAVA. Java – об’єктно-орієнтована мова програмування, яка розроблена компанією Sun Microsystems. Додатки Java зазвичай компілюються у спеціальний байт-код, тому вони можуть працювати на будь-якій віртуальній Java-машині незалежно від комп’ютерної архітектури.

Java – так називають не тільки саму мову, але й платформу для створення й виконання додатків на основі цієї мови.

Програми на Java транслюються у байт-код, виконуваний віртуальною машиною Java (JVM) – програмою, що обробляє байтовий код і передавальні інструкції обладнанню як інтерпертатор, але байтовий код на відміну від тексту обробляється значно швидше.

Перевага подібного способу виконання програм полягає у повній незалежності байт-коду від ОС та обладнання, що дозволяє виконувати Java-додатки на будь-якому пристрої, для якого є відповідна віртуальна машина. Іншою важливою особливістю технології Java є гнучка система безпеки завдяки тому, що виконання програми повністю контролюється віртуальною машиною. Будь-які операції, які перевищують установлені повноваження програми (наприклад, спроба несанкціонованого доступу до даних або з’єднання з іншим комп’ютером) спричиняють негайне переривання.

Часто недоліком концепції віртуальної машини вважають те, що виконання нею байт-коду може знижувати продуктивність програм і алгоритмів, реалізованих мовою Java. Це твердження справедливе для перших версій віртуальної машини Java, однак оста-

ннім часом воно майже втратило актуальність. Цьому сприяли такі удосконалення:

– застосування технології трансляції байта-коду в машинний код безпосередньо під час роботи програми (JIT-технологія) з можливістю збереження версій класу в машинному коді,

– широке використання платформно-орієнтованого коду (native-код) у стандартних бібліотеках,

– апаратні засоби, що забезпечують прискорене оброблення байт-коду (наприклад, технологія Jazelle, підтримувана деякими процесорами фірми ARM).

Ідеї, закладені в концепцію, й різні реалізації середовища віртуальної машини Java, надихнули безліч ентузіастів на розширення переліку мов, які можна використовувати для створення програм, виконуваних на віртуальній машині. Ці ідеї знайшли також вираження в специфікації загальномовної інфраструктури CLI, покладеної в основу платформи .NET компанією Microsoft.

– SQL. SQL принципово відрізняється від традиційних алгоритмічних мов програмування, насамперед тим, що вона належить до непроцедурних мов. Мова SQL дозволяє задавати тільки те, «що потрібно робити», зокрема виконання окремих операцій («як робити») покладається безпосередньо на СКБД. Такий підхід значною мірою визначається самою філософією баз даних. Система керування базами даних у цьому випадку розглядається як «чорний ящик», і те, що відбувається всередині нього, користувача не повинно стосуватися. Він має цікавитися тільки внесенням у базу даних потрібних змін і отриманням правильної відповіді на запит. Отже, найбільш близькою архітектурою тут буде модель сховища даних.

Іншою особливістю SQL є тризначна логіка. У більшості мов булеве вираження може набувати тільки двох значень: істини й хибності. Мова SQL дозволяє записувати в базу даних значення *NULL* (порожнє значення). *NULL* – це спеціальний код, поміщений в стовпець таблиці, якщо з якої-небудь причини в ньому немає даних. Коли значення *NULL* бере участь в операціях порівняння, булевим результатом буде не істина і не хибність, а невизначеність.

Мова SQL не є мовою програмування відповідно до визначення цього терміна. SQL являє собою субмову даних, призначену для використання лише як інтерфейсу із бази даних. Сама по собі SQL не містить тих засобів, які необхідні для розроблення закінче-

них програм, і може застосовуватися у формі однієї із трьох прикладних реалізацій:

1. Інтерактивна (або автономна) SQL дає змогу користувачам безпосередньо витягати інформацію з бази або записувати в неї дані. Інформація, одержувана за запитом SQL, може бути видана на екран, переадресована у файл або на принтер.

2. Статична SQL дозволяє записувати фіксований виконавчий код SQL, якщо він зазвичай використовується в додатках.

Є два різновиди статичної SQL: вбудована і модульна.

Убудована SQL визначається як код SQL, що включений у вихідний текст програми, написаної іншою мовою програмування.

У модульному варіанті оператори SQL записані в окремих модулях, які компонуються з модулями основної мови.

3. Динамічна SQL дає змогу генерувати код SQL під час виконання додатка й використовується замість статичної SQL у тих випадках, коли під час розроблення додатка необхідний код SQL ще не може бути визначений або залежить від того, який вибір зробить користувач.

Оператори динамічної SQL зазвичай застосовуються в діалогових середовищах для побудови запитів і в графічних засобах розроблення додатків бази даних.

– Object PAL. Object PAL являє собою об'єктно-орієнтовану, керовану по подіях, візуальну мову програмування.

На початковому рівні функціональності Object PAL можна виконувати операції з даними, створювати спеціальні меню, а також керувати сеансом уведення даних. Події в Object PAL породжують команди, які імітують ефект використання Paradox в інтерактивному режимі. Є можливість автоматизувати часто виконувані завдання, а також виконувати з таблицями, формами й звітами дії, які були недоступні під час інтерактивної роботи.

Object PAL надає всі засоби повнофункціональної мови програмування в середовищі Windows. Можна використовувати Object PAL для створення закінчених систем, у яких реалізовані спеціальні системи меню, довідкова система, а також перевірки даних.

Object PAL може використовуватися як інструмент для створення автономних програм. Можна написати закінчений Windows-додаток і запустити його для Paradox.

Object PAL підтримує механізм динамічного обміну даними і як клієнта, і як сервера. Крім того, Object PAL підтримує як клієнт механізм роботи зі комплексними документами. Є також можливість включати у додаток мультимедійні засоби з додаванням до виконуваних додатків звукових і анімаційних ефектів.

– dBase. Створена фірмою Борланд реалізація мови dBase являє собою гібрид об'єктної орієнтації й традиційних способів програмування, що дозволяє створювати системи за допомогою об'єктного дизайну й використовувати звичайні методи обробления записів. Отже, це застосування підходу сховища даних та об'єктної архітектури.

Це дозволяє створювати нові класи об'єктів, які мають властивості спадкування, інкапсуляції й поліморфізму. Вона також дозволяє програмувати ці об'єкти за допомогою традиційних команд dBase, що ідеально підходять для керування простими табличними базами даних. Усе це дає незаперечну перевагу – здійснювати без особливих зусиль перехід до прийомів об'єктного програмування, наприклад, як у СКБД Paradox.

– HTML. Термін HTML (HyperText Markup Language) означає *мова маркування* (*розмітки*) *гіпертекстів*. Мова HTML була необхідна для статичного розміщення сторінок у всесвітній павутині WWW (World Wide Web) за допомогою клієнт-серверного підходу.

Із часу створення першої версії HTML зазнала деяких змін. Як і зазвичай в комп'ютерному середовищі версії, або специфікації, HTML пронумеровані. Відомі специфікації 2.0, 3.0 і 3.2, 4.0.

– Perl і PHP. Ці мови програмування є інтерпретованими мовами для оброблення великих текстів і файлів. Вони найбільше підходять до трирівневої та клієнт-серверної архітектури. За допомогою цих мов можна створити скріпти, що відкривають один або кілька файлів, обробляти інформацію й записувати результати.

З винаходом World Wide Web, Perl та PHP виявилися дієвим засобом для взаємодії з web-серверами через Common Gateway Interface (CGI) – загальний інтерфейс взаємодії. Команди Perl та PHP можуть легко отримувати дані з форми HTML або іншого джерела й виконувати з ними будь-які дії.

3.5. Методологія структурного аналізу і проектування – SADT

3.5.1. Сутність структурного підходу

Сутність структурного підходу до розроблення ІС полягає в її декомпозиції (розбитті) на автоматизовані функції: функціональні підсистеми, які, в свою чергу, діляться на підфункції, останні на завдання і так далі. Процес розбиття триває аж до конкретних процедур. При цьому автоматизована система зберігає цілісне подання, у якому всі складові компоненти взаємопов'язані. Під час розроблення системи «знизу вверх» від окремих завдань до всієї системи цілісність втрачається, виникають порушення у процесі інформаційного стикування окремих компонентів.

Найбільш поширені методології структурного підходу ґрунтуються на загальних принципах. Як два базові принципи використовують:

– «розділяй і володарюй» – принцип вирішення складних проблем шляхом їх розбиття на множину менших незалежних завдань, легких для розуміння і розв'язання;

– ієрархічного впорядкування – принцип організації складових частин завдання в ієрархічні деревоподібні структури з додаванням нових деталей на кожному рівні.

Виділення двох базових принципів не означає, що інші принципи є другорядними, оскільки ігнорування будь-якого з них може призвести до непередбачуваних наслідків (у тому числі і до провалу всього проекту). Основні принципи:

– абстрагування – виділення головних аспектів системи і відволікання від несуттєвих;

– формалізації – потреба суворого методичного підходу до вирішення проблеми;

– несуперечності – обґрунтованість та узгодженість елементів;

– структурування даних – дані мають бути структуровані й ієрархічно організовані.

У структурному аналізі використовують переважно дві групи засобів, що ілюструють функції, які виконуються системою, і співвідношення даних. Кожній групі засобів відповідають певні види моделей (діаграм), найпоширеніші з-поміж них такі:

– SADT (Structured Analysis and Design Technique) моделі і відповідні функціональні діаграми;

– DFD (Data Flow Diagrams) діаграми потоків даних (підрозділ 3.5.6);

– ERD (Entity-Relationship Diagrams) діаграми «сутність – зв'язок».

На стадії проектування ІС моделі розширюються, уточнюються і доповнюються діаграмами, що відображають структуру ПЗ: архітектуру ПЗ, структурні схеми програм і діаграми екранних форм.

Перераховані моделі в сукупності дають повний опис ІС незалежно від того, чи вона існує чи розробляється. Склад діаграм у кожному конкретному випадку залежить від необхідної повноти опису системи.

В основу різних методологій моделювання предметної галузі ІС покладено принципи послідовної деталізації абстрактних категорій. Зазвичай моделі будуються на трьох рівнях: зовнішньому (визначення вимог), концептуальному (специфікації вимог) і внутрішньому (реалізації вимог). Так, модель визначає:

– на зовнішньому рівні – що має робити система, тобто склад основних компонентів системи (об'єктів, функцій, подій, організаційних одиниць, технічних засобів);

– на концептуальному рівні – як повинна функціонувати система, тобто характер взаємодії компонентів системи одного і різних типів;

– на внутрішньому рівні – за допомогою яких програмно-технічних засобів реалізуються вимоги до системи.

З позиції життєвого циклу ІС описуються рівні моделей відповідно будуються на етапах аналізу вимог, логічного (технічного) і фізичного (робочого) проектування.

3.5.2. Об'єктна структура

Об'єкт – це сутність, яка використовується під час виконання певної функції або операції (перетворення, оброблення, формування і т. ін.). Об'єкти можуть мати динамічну чи статичну природу: динамічні об'єкти використовуються в одному циклі відтворення (наприклад, замовлення на продукцію, рахунки на

оплату, платежі); статичні об'єкти – у багатьох циклах відтворення, наприклад, обладнання, персонал, запаси матеріалів.

На зовнішньому рівні деталізації моделі виділяються основні види матеріальних об'єктів (наприклад, сировина і матеріали, напівфабрикати, готові вироби, послуги) та основні види інформаційних об'єктів або документів (наприклад, замовлення, накладні, рахунки і т. ін.).

На концептуальному рівні побудови моделі предметної галузі уточнюється склад класів об'єктів, визначаються їх атрибути і взаємозв'язки. Таким чином будується узагальнене уявлення структури предметної галузі.

Далі концептуальна модель на внутрішньому рівні відображається у вигляді файлів бази даних, вхідних і вихідних документів ЕІС. Причому динамічні об'єкти задаються одиницями змінної інформації або документами, а статичні об'єкти – одиницями умовно-постійної інформації у вигляді списків, номенклатур, цінників, довідників, класифікаторів. Модель бази даних як постійно підтримуваний інформаційний ресурс відображає зберігання умовно-постійної і нагромадженої змінної інформації, яка використовується у повторюваних інформаційних процесах.

3.5.3. Методологія функціонального моделювання SADT

На основі методології SADT, розробленої Дугласом Россом, створено відому методологію IDEF0 (Icam DEFinition), яка є основною частиною програми ICAM (Інтеграція комп'ютерних та промислових технологій), що проводиться за ініціативою ВПС США.

Методологія SADT являє собою сукупність методів, правил і процедур, призначених для побудови функціональної моделі об'єкта будь-якої предметної галузі. Функціональна модель SADT відображає функціональну структуру об'єкта, тобто вироблені ним дії й зв'язки між цими діями. Основні елементи цієї методології ґрунтуються на таких концепціях, як графічне зображення блокового моделювання та суворість і точність.

Графічне подання блокового моделювання. Графіка блоків і дуг SADT-діаграми зображує функцію у вигляді блока, а інтерфейси входу/виходу – дуги, відповідно входять у блок і виходять з нього. Взаємодія блоків один з одним описуються за допомогою інтер-

фейсних дуг, що виражають «обмеження» які, в свою чергу, визначають, коли і яким чином функції виконуються й керуються;

Суворість і точність. Виконання правил SADT потребує достатньої суворості й точності, не накладаючи водночас надмірних обмежень на дії аналітика.

Правила SADT включають:

– обмеження кількості блоків на кожному рівні декомпозиції (правило 3–6 блоків);

– зв'язність діаграм (номера блоків);

– унікальність міток і найменувань (відсутність повторюваних імен);

– синтаксичні правила для графіки (блоків і дуг);

– поділ входів і керувань (правило визначення ролі даних);

– відокремлення організації від функції, тобто запобігання впливу організаційної структури на функціональну модель.

Методологію SADT можна використовувати для моделювання багатьох систем та визначення вимог і функцій, а потім для розроблення системи, яка задовольняє ці вимоги і реалізує ці функції. Для вже існуючих систем SADT може бути використана для аналізу функцій, виконуваних системою, а також для визначення механізмів, за допомогою яких вони здійснюються.

Склад функціональної моделі. Результатом застосування методології SADT є модель, яка складається з діаграм, фрагментів текстів і глосарію, що мають посилання один на одного. Діаграми – головні компоненти моделі, всі функції ІС і інтерфейси на них подані як блоки і дуги. Місце з'єднання дуги з блоком визначає тип інтерфейсу. Керувальна інформація входить у блок зверху, у той час, як оброблювана інформація показана з лівого боку блока, а результати виходу – з правого. Механізм (людина або автоматизована система), що виконує операцію, зображується дугою, що входить до блока знизу (рис. 3.12).

Однією з найважливіших особливостей методології SADT є поступове введення щоразу більших рівнів деталізації у міру створення діаграм, що відображають модель.

Структуру SADT-моделі показано на рис. 3.12 (чотири діаграми та їх взаємозв'язок). Кожен компонент моделі може бути декомпозований на іншій діаграмі (рис. 3.13). Кожна діаграма ілюструє внутрішню будову блока на батьківській діаграмі.

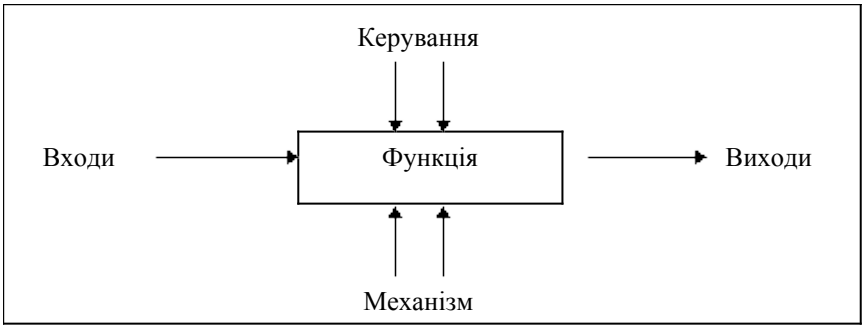


Рис. 3.12. Функціональний блок та інтерфейсні дуги

Ієрархія діаграм. Побудова SADT-моделі починається з надання всій системи вигляду найпростішої компоненти – одного блока і дуг, що зображують інтерфейси з функціями поза системою. Оскільки єдиний блок являє собою всю систему як єдине ціле, ім'я, вказане в блоці, є спільним. Це справджується і для інтерфейсних дуг – вони також є повним набором зовнішніх інтерфейсів системи в цілому.

Блок, який є системою як єдиний модуль, деталізується на іншій діаграмі за допомогою декількох блоків, з'єднаних інтерфейсними дугами. Ці блоки являють собою основні підфункції вихідної функції. Така декомпозиція є повним набором підфункцій, кожна з яких являє собою блок, межі якого визначені інтерфейсними дугами. Кожна з цих підфункцій може бути декомпозиційована подібним чином для більш детального представлення.

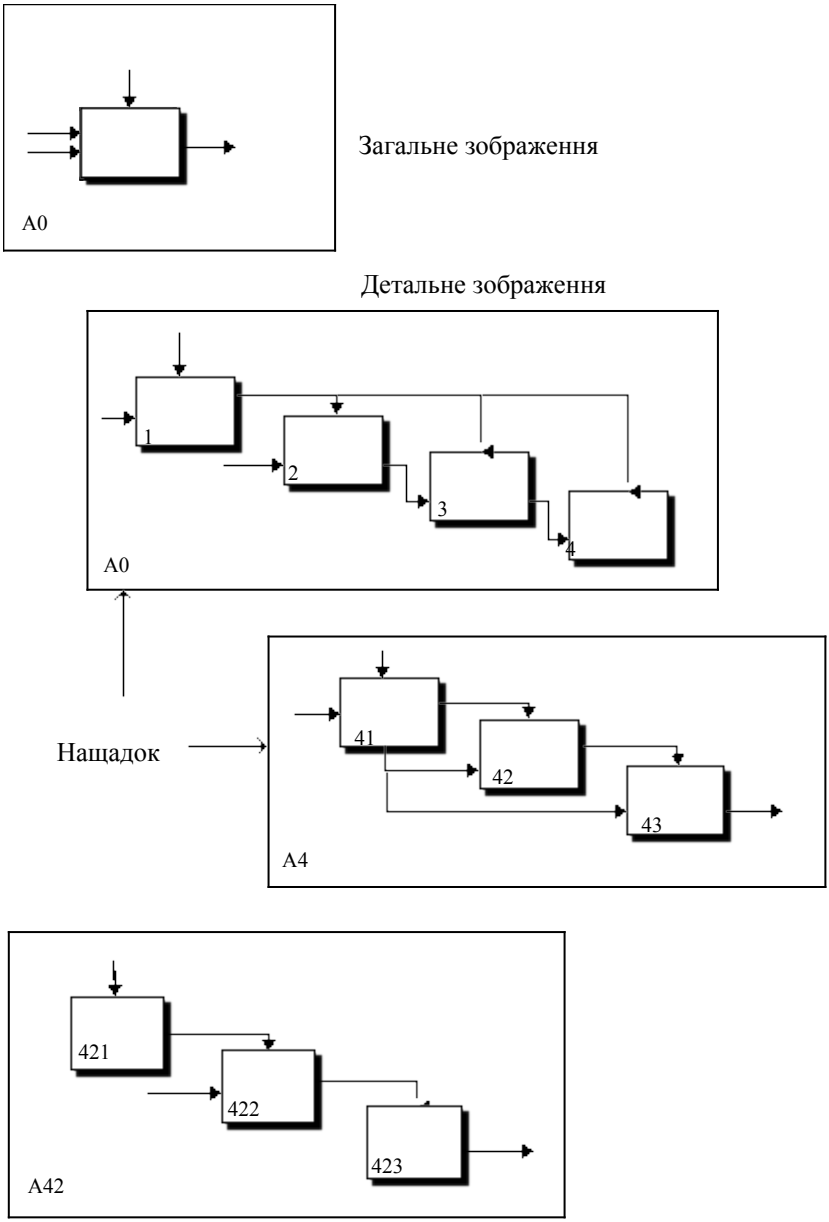


Рис. 3.13. Структура SADT-моделі та декомпозиція діаграм

У всіх випадках кожна підфункція може містити тільки ті елементи, які входять у вихідну функцію. Крім того, модель не може випустити будь-які елементи, тобто батьківський блок і його інтерфейси забезпечують контекст. До нього не можна нічого додати, і з нього не можна нічого видалити.

Модель SADT являє собою серію діаграм із супровідною документацією, що розбивають складний об'єкт на складові частини, які мають вигляд блоків. Деталі кожного з основних блоків показані у вигляді блоків на інших діаграмах. Кожна детальна діаграма є декомпозицією блока з більш узагальненої діаграми. На кожному кроці декомпозиції більш узагальнена діаграма називається батьківською для більш детальної діаграми.

Дуги, що входять у блок і виходять з нього на діаграмі верхнього рівня, є точно такими самими, що й дуги, що входять у діаграму нижнього рівня і виходять з неї, оскільки блок і діаграма складають одну і ту ж частину системи.

Різні варіанти виконання функцій і з'єднання дуг із блоками показано на рис. 3.14 – 3.16.

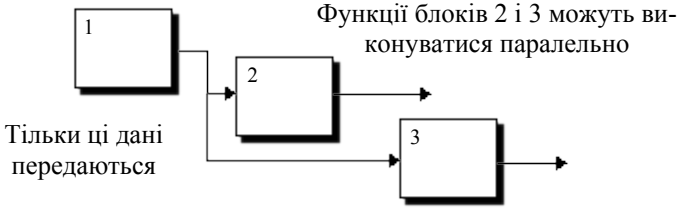


Рис. 3.14. Одночасне виконання

Деякі дуги приєднані до блоків діаграми обома кінцями, в інших же один кінець залишається неприєднаним. Неприєднані дуги відповідають входам, керуванням і виходам батьківського блока. Джерело або одержувач цих дотичних дуг виявляється тільки на батьківській діаграмі. Неприєднані кінці повинні відповідати дугам на вихідної діаграмі. Всі дотичні дуги повинні продовжуватися на батьківській діаграмі для забезпечення її повноти і несуперечливості.

На SADT діаграмах не вказані явно ні послідовність, ні час. Зворотні зв'язки, ітерації, що тривають, процеси і функції, які пере-

криваються (за часом), можна зобразити за допомогою дуг. Зворотні зв'язки можуть мати вигляд коментарів, зауважень, виправлень і т. ін. (рис. 3.16).

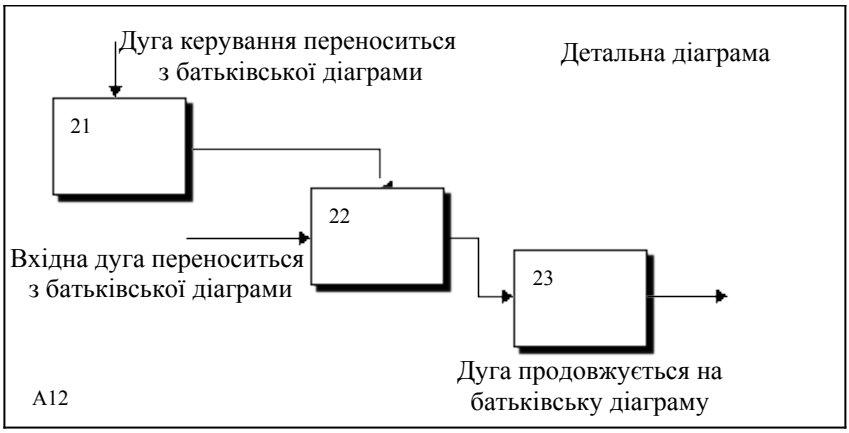


Рис. 3.15. Повна та несуперечлива відповідність

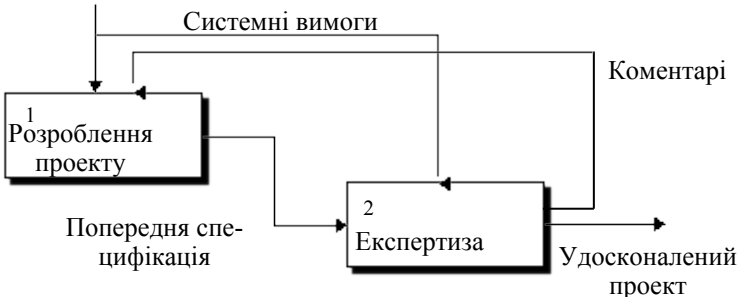


Рис. 3.16. Приклад зворотного зв'язку

Механізми (дуги з нижнього боку) показують засоби, за допомогою яких виконуються функції. Механізм може бути людиною, комп'ютером або будь-яким іншим пристроєм, що допомагає виконувати цю функцію (рис. 3.17).

Кожен блок на діаграмі має свій номер. Блок будь-якої діаграми можна описати діаграмою нижнього рівня, яку потім деталізувати за допомогою необхідної кількості діаграм. Таким чином, формується ієрархія діаграм.

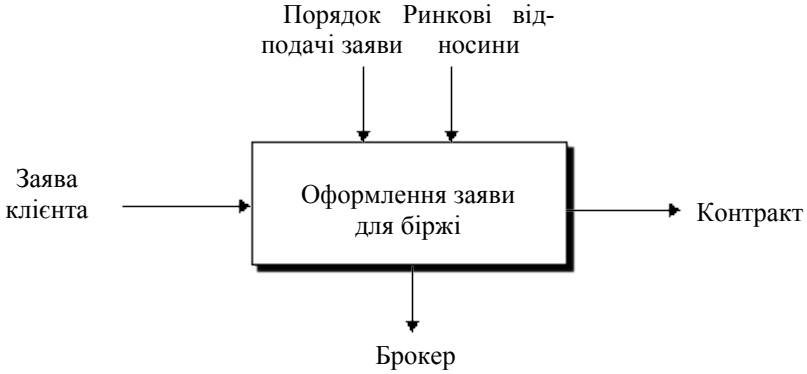


Рис. 3.17. Приклад механізму

Щоб указати положення будь-якої діаграми або блока в ієрархії, використовують номери діаграм. Наприклад, A21 є діаграмою, яка деталізує блок 1 на діаграмі A2. Аналогічно, A2 деталізує блок 2 на діаграмі A0, яка є верхньою діаграмою моделі. Типове дерево діаграм показано на рис. 3.18.

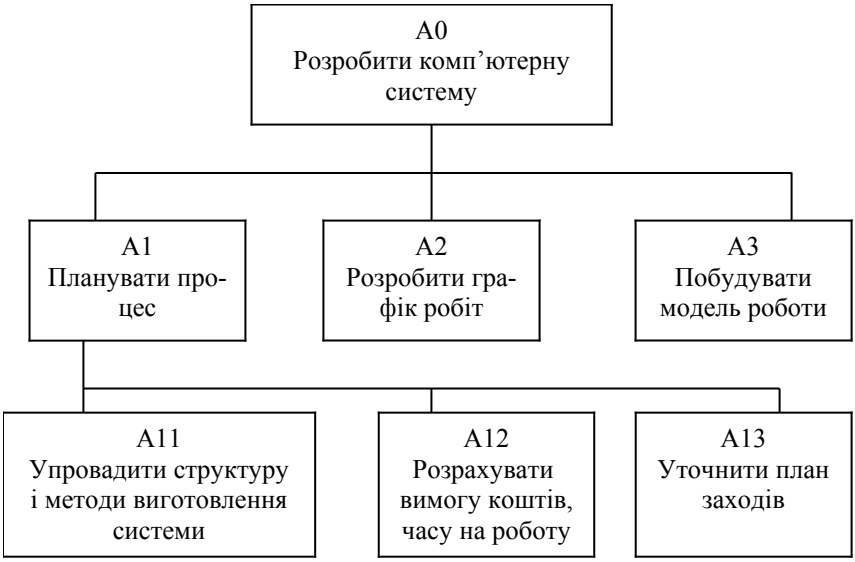


Рис. 3.18. Ієрархія діаграм

Типи зв'язків між функціями. Одною з важливих особливостей проектування ІС за допомогою методології SADT є точна узгодженість типів зв'язків між функціями. Розрізняють принаймні сім типів зв'язування (табл. 3.3):

Таблиця 3.3

Типи зв'язку

Тип зв'язку	Відносна значущість
Випадкова	0
Логічна	1
Тимчасова	2
Процедурна	3
Комунікаційна	4
Послідовна	5
Функціональна	6

На прикладі SADT розглянемо кожен тип зв'язку.

(0) Тип випадкової зв'язності: найменш бажаний. Випадкова зв'язність виникає, коли конкретний зв'язок між функціями є або його немає. Це випадок, коли імена даних на SADT дугах в одній діаграмі мало зв'язані між собою. Крайній варіант цього випадку показано на рис. 3.19.

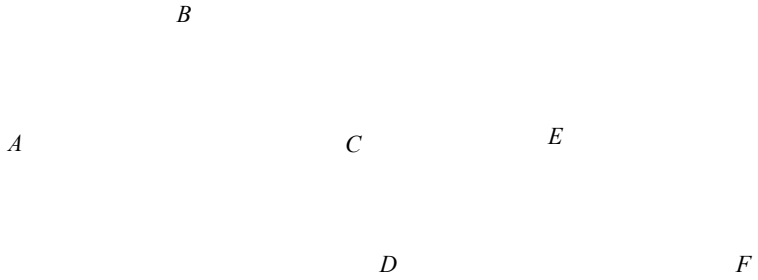


Рис. 3.19. Випадкова зв'язність

(1) Тип логічної зв'язності. Логічне зв'язування відбувається тоді, коли характеристики та функції збираються разом унаслідок їх потрапляння в загальний клас або набір елементів, але необхідних функціональних відносин між ними не виявляється.

(2) Тип тимчасової зв’язності. Зміни, узгоджені в часі елементи, виникають унаслідок того, що вони виражають функції, узгоджені в часі, якщо дані використовуються одночасно або функції включаються паралельно, а не послідовно.

(3) Тип процедурної зв’язності. Процедурно-зв’язані елементи виявляються згрупованими внаслідок того, що вони виконуються протягом однієї й тієї ж частини циклу або процесу. Приклад процедурно-зв’язаної діаграми показано на рис. 3.20.

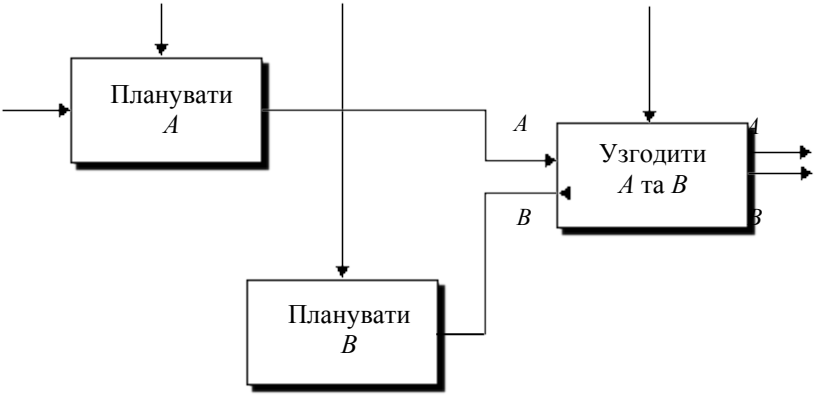


Рис. 3.20. Процедурна зв’язність

(4) Тип комунікаційної зв’язності. Діаграми демонструють комунікаційні зв’язки, коли блоки групуються внаслідок використання одніх і тих самих вхідних даних та / або виробляють одні й ті ж вихідні дані (рис. 3.21).

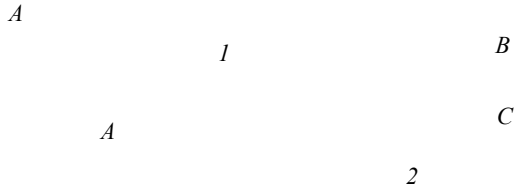


Рис. 3.21. Комунікаційна зв’язність

(5) Тип послідовної зв'язності. На діаграмах, що мають послідовні зв'язки, вихід однієї функції є вхідними даними для наступної функції. Зв'язок між елементами на діаграмі більш тісний на відміну від рівнів зв'язності (0)–(4), оскільки моделюються причинно-наслідкові залежності (рис. 3.22).

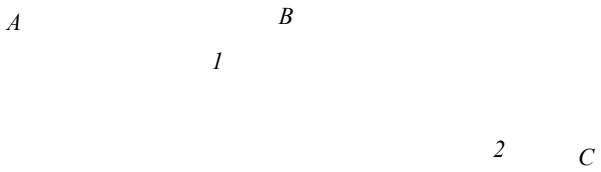


Рис. 3.22. Послідовна зв'язність

(6) Тип функціональної зв'язності. Діаграма відображає повну функціональну зв'язність за наявності повної залежності однієї функції від іншої. Діаграма, яка є чисто функціональною, не містить сторонніх елементів, що належать до послідовного або більш слабкого типу зв'язності. Одним зі способів визначення функціонально-зв'язаних діаграм є розгляд двох блоків, з'єднаних керувальними дугами, як показано на рис. 3.23.

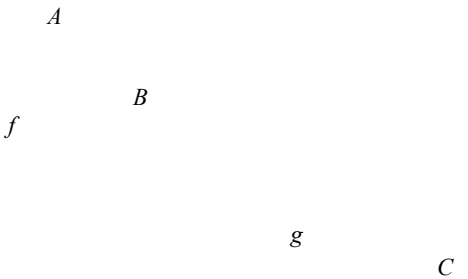


Рис. 3.23. Функціональна зв'язність

У математичних термінах необхідна умова для простого типу функціональної зв'язності (рис. 3.23) має такий вигляд:

$$C = g(B) = g(f(A)).$$

Типи зв’язності наведено в табл. 3.4. Важливо відзначити, що рівні (4) – (6) установлюють типи зв’язності, які розробники вважають найважливішими для побудови діаграм високої якості.

Таблиця 3.4

Типи зв’язності для функцій та даних

Значущість	Тип зв’язності	Для функцій	Для даних
0	Випад-кова	Випадкова	Випадкова
1	Логічна	Опції однієї і тієї ж множини або типу (наприклад, «коригувати всі входи»)	Дані однієї і тієї самої множини або типу
2	Тимча-сова	Опції одного й того ж періоду часу (наприклад, «операції ініціалізації»)	Дані, використовувані в будь-якому часовому інтервалі
3	Процедурна	Функції, які працюють в одній і тій же фазі або ітерації (наприклад, «перший прохід компілятора»)	Дані, використовувані під час однієї і тієї ж фази або ітерації
4	Комунікаційна	Функції, що використовують одні й ті ж дані	Дані, на які впливає одна й та сама діяльність
5	Послі-довна	Функції, що виконують послідовні перетворення одних і тих самих даних	Дані, перетворені послідовними функціями
6	Функціональна	Функції, об’єднані для виконання однієї функції	Дані, пов’язані з однією функцією

Моделювання потоків даних (процесів). В основу цієї методології (методології Gane / Sarson [11]) покладено побудову моделі

аналізованої ІС – проектованої або реально існуючої. Відповідно до методології модель системи визначається як ієрархія діаграм потоків даних (ДПД або DFD), що описують асинхронний процес перетворення інформації від часу її введення в систему до видачі користувачу. Діаграми верхніх рівнів ієрархії (контекстні діаграми) визначають основні процеси або підсистеми ІС із зовнішніми входами і виходами. Вони деталізуються за допомогою діаграм нижнього рівня. Така декомпозиція триває, створюючи багаторівневу ієрархію діаграм, доти, доки не буде досягнуто такого рівня декомпозиції, на якому процеси стають елементарними і деталізувати їх далі неможливо.

Джерела інформації (зовнішні сутності) породжують інформаційні потоки (потоки даних), що переносять інформацію до підсистем або процесів. Ті, в свою чергу, перетворюють інформацію і породжують нові потоки, які переносять інформацію до інших процесів або підсистем, нагромаджувачів даних або зовнішніх сутностей – до споживачів інформації. Таким чином, основними компонентами діаграм потоків даних є:

– зовнішні сутності;
– системи / підсистеми;
– процеси;
– нагромаджувачі даних;
– потоки даних.

Зовнішні сутності. Зовнішня сутність являє собою матеріальний предмет або фізичну особу, що являє собою джерело або приймач інформації (наприклад, замовники, персонал, постачальники, клієнти, склад). Визначення деякого об’єкта або системи як зовнішньої сутності вказує на те, що вона перебуває за межами аналізованої ІС. У процесі аналізу деякі зовнішні сутності можуть бути перенесені всередину діаграми аналізованої ІС, якщо це необхідно, або, навпаки, частина процесів ІС може бути винесена за межі діаграми і подана як зовнішня сутність.

Зовнішня сутність позначається квадратом (рис. 3.24), розміщеним немовби «над» діаграмою та кидає на неї тінь, що дозволяє виділити цей символ серед інших позначень.

Системи і підсистеми. Для побудови моделі складної ІС її можна подати в загальному вигляді на контекстній діаграмі – у

вигляді однієї системи як єдиного цілого, або декомпозувати на ряд підсистем.

Рис. 3.24. Зовнішня сутність

Підсистема (або система) на тематичній діаграмі зображується таким чином, як показано на рис. 3.25.

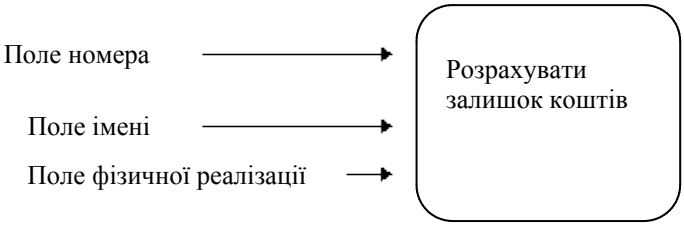


Рис. 3.25. Підсистема

Номер підсистеми призначений для її ідентифікації. У поле імені вводиться найменування підсистеми у вигляді пропозиції з підметом і відповідними визначеннями та доповненнями.

Процеси. Процес являє собою перетворення вхідних потоків даних у вихідні згідно з певним алгоритмом. Фізично процес може бути реалізований різними способами: це може бути підрозділ організації (відділ), що обробляє вхідні документи і складає звіти, програма, апаратно реалізований логічний пристрій і т. ін.

Процес на діаграмі потоків даних зображується так, як показано на рис. 3.26.

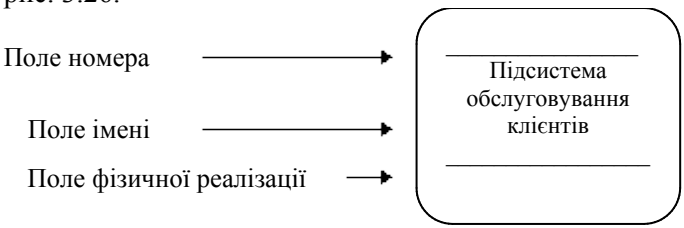


Рис. 3.26. Процес

Номер процесу призначений для його ідентифікації. У поле імені вводиться найменування процесу у вигляді пропозиції з активним недвозначним дієсловом у невизначеній формі (обчислити, розрахувати, перевірити, визначити, створити, отримати), за яким слідують іменники в знахідному відмінку, наприклад:

– «Увести відомості про клієнтів»;
– «Видати інформацію про поточні витрати»;
– «Перевірити кредитоспроможність клієнта».

Використання таких дієслів, як «опрацювати», «модернізувати» або «відредагувати» означає недостатньо глибоке розуміння даного процесу і потребує подальшого аналізу.

Інформація в полі фізичної реалізації показує, які підрозділи організації, програма або апаратний пристрій виконують процес.

Нагромаджувачі даних. Нагромаджувач даних являє собою абстрактний пристрій для зберігання інформації, яку можна в будь-який момент часу помістити в нагромаджувач і через деякий час вилучити, причому спосіб поміщення та вилучення може бути будь-яким.

Нагромаджувач даних може бути реалізований фізично у вигляді мікрофільму, скриньки в картотеці, таблиці в оперативній пам'яті, файлу на магнітному носії та ін. Нагромаджувач даних на діаграмі потоків даних зображується так, як показано на рис. 3.27.

D1	Отримані розрахунки

Рис. 3.27. Нагромаджувач даних

Нагромаджувач даних ідентифікується літерою D і довільним числом. Назва нагромаджувача вибирається з міркування найбільшої інформативності для проектувальника.

Нагромаджувач даних у загальному випадку є прообразом майбутньої бази даних і опис даних, що зберігаються в ньому, має бути пов'язаний з інформаційною моделлю.

Потоки даних. Потік даних визначає інформацію, яка передається через підключення від джерела до приймача. Реальний потік даних може бути інформацією, що передається по кабелю між двома пристроями, пересилається по пошті листами, магнітними стрічками або дискетами і т. ін.

Потік даних на діаграмі зображується лінією, що закінчується стрілкою, яка вказує напрямок потоку (рис. 3.28).

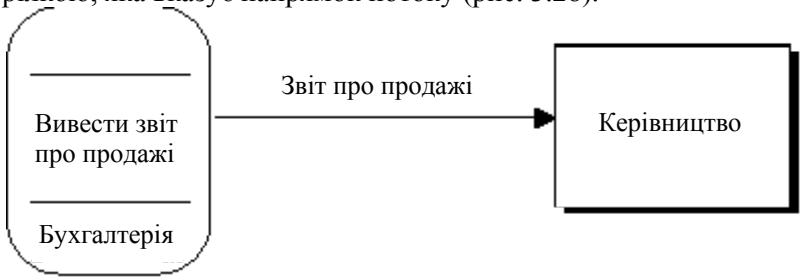


Рис. 3.28. Потік даних

Кожен потік даних має ім'я, що відображає його зміст.

Побудова ієрархії діаграм потоків даних. Перший крок побудови ієрархії ДПД – побудова тематичних діаграм. Зазвичай під час проектування простих ІС будується єдина контекстна діаграма зі зіркоподібною топологією, в центрі якої міститься головний процес, сполучений з приймачами і джерелами інформації, за допомогою яких із системою взаємодіють користувачі та інші зовнішні системи.

Якщо ж для складної системи обмежитися єдиною контекстною діаграмою, то вона буде містити дуже велику кількість джерел і приймачів інформації, які важко розмістити на аркуші паперу нормального формату; крім того, єдиний головний процес не розкриває структури розподіленої системи. Ознаками складності (в сенсі контексту) можуть бути:

– наявність великої кількості зовнішніх сутностей (десять і більше);

– розподілена природа системи;

– багатофункціональність системи з уже сформованим або виявленим групованням функцій в окремі підсистеми.

Для складних ІС будується ієрархія контекстних діаграм. При цьому контекстна діаграма верхнього рівня містить не єдиний головний процес, а набір підсистем, з'єднаних потоками даних. Контекстні діаграми наступного рівня деталізують контекст і структуру підсистем.

Ієрархія контекстних діаграм визначає взаємодію основних функціональних підсистем проектованої ІС як між собою, так і з

зовнішніми вхідними і вихідними потоками даних та зовнішніми об'єктами (джерелами і приймачами інформації), з якими взаємодіє ІС.

Розроблення тематичних діаграм вирішує проблему визначення функціональної структури ІС на найбільш ранній стадії її проектування, що особливо важливо для складних багатофункціональних систем, у розробленні яких беруть участь різні організації та колективи розробників.

Після побудови тематичних діаграм отриману модель слід перевірити на повноту вихідних даних про об'єкти системи та ізольованість об'єктів (відсутність інформаційних зв'язків з іншими об'єктами).

Для кожної підсистеми, наявної на контекстних діаграмах, виконується її деталізація за допомогою ДПД. Кожен процес на ДПД, у свою чергу, може бути деталізований за допомогою ДПД або мініспеціфікації. Під час деталізації повинні виконуватися такі правила:

– правило балансування – під час деталізації підсистеми або процесу деталізується діаграма. Як зовнішні джерела / приймачі даних можуть бути лише ті компоненти (підсистеми, процеси, зовнішні сутності, нагромаджувачі даних), з якими має інформаційний зв'язок деталізована підсистема або процес на батьківській діаграмі;

– правило нумерації – під час деталізації процесів повинна підтримуватися їх ієрархічна нумерація. Наприклад, процеси деталізують процес з номером 12, отримують номери 12.1, 12.2, 12.3 і т.д.

Мініспеціфікація, або опис логіки процесу – це формулювання його основних функцій таким чином, щоб фахівець, який реалізовує проект, зміг виконати їх або розробити відповідну програму.

Мініспеціфікація є кінцевою вершиною ієрархії ДПД. Рішення про завершення деталізації процесу і використання мініспеціфікації приймається аналітиком, виходячи з таких критеріїв:

– наявності у процесі відносно невеликої кількості вхідних та вихідних потоків даних (2 – 3 потоки);

– можливості опису перетворення даних процесом у вигляді послідовного алгоритму;

– виконання процесом єдиної логічної функції перетворення вхідної інформації у вихідну;

– можливості опису логіки процесу за допомогою мініспеціфікації (не більше 20 – 30 рядків).

Під час побудови ієрархії ДПД переходити до деталізації процесів слід тільки після визначення змісту всіх потоків і нагромаджувачів даних, що описується за допомогою структур даних. Структури даних конструюються з елементів даних і можуть містити альтернативи, умовні входження та ітерації. Умовне входження означає, що деякого компонента може не бути у структурі. Альтернатива означає, що в структуру може входити один з елементів даних. Ітерація означає входження будь-які кількості елементів у зазначеному діапазоні. Для кожного елемента даних може вказуватися його тип (безперервні або дискретні дані), для безперервних даних – одиниця вимірювання, діапазон значень, точність подання та форма фізичного кодування, а для дискретних даних – таблиця допустимих значень.

Після побудови закінченої моделі системи її необхідно верифікувати (перевірити на повноту і узгодженість). Повна модель усі її об’єкти (підсистеми, процеси, потоки даних) повинна докладно описувати й деталізовати. Виявлені недеталізовані об’єкти слід деталізувати, повернувшись до попередніх кроків розроблення. Відповідно до узгодженої моделі для всіх потоків даних і нагромаджувачів даних має виконуватися правило збереження інформації: всі дані, що входять куди-небудь, повинні бути прочитані, а всі дані, які зчитуються,– бути записані.

3.6. Методологія RUP

Усі провідні компанії – розробники технологій і програмних продуктів (IBM, Oracle, Borland, Computer Associates та ін.) мають у своєму розпорядженні розвинені технології створення ПЗ, які створювалися як власними силами, так і за рахунок придбання продуктів і технологій, створених невеликими спеціалізованими компаніями. Провідною методологією, згідно з якою инструментально підтримуються всі етапи життєвого циклу розроблення ПЗ, є RUP, що являє собою програмний продукт, розроблений компанією Rational Software, яка входить до складу IBM. Вона спи-

рається на перевірені практикою методи аналізу, проектування й розроблення ПЗ, методи керування проектами. RUP забезпечує прозорість і керованість процесу й дозволяє створювати ПЗ відповідно до вимог замовника та до можливостей інструментальних засобів підтримання розроблення.

Rational Unified Process пропонує ітеративну модель розроблення, що включає чотири фази:

– початок,
– дослідження,
– побудова,
– упровадження.

Кожну фазу можна розбити на етапи (ітерації), у результаті яких випускається версія для внутрішнього або зовнішнього використання. Проходження крізь чотири основні фази називається циклом розроблення, кожний цикл завершується генерацією версії системи. Якщо після цього робота над проектом не припиняється, то отриманий продукт продовжує розвиватися й знову пройде ті самі фази. Сутність роботи в межах RUP – це створення й супровід моделей на базі Universal Modelling Language (UML).

В основі методології RUP, як і багатьох інших програмних методологій, що поєднують інженерні методи створення ПЗ, лежить «покроковий підхід». Він визначає етапи життєвого циклу, контрольні точки, правила робіт для кожного етапу й тим самим упорядковує проектування й розроблення ПЗ. Для кожного етапу життєвого циклу методологія задає:

– склад і послідовність робіт, а також правила їх виконання;
– розподіл повноважень серед учасників проекту (ролі);
– склад і шаблони формованих проміжних і підсумкових документів;
– порядок контролю й перевірки якості.

3.6.1. Принципи розроблення програмного забезпечення RUP

Методологію розроблення ПЗ RUP створила компанія Rational Software.

В основу RUP покладено такі основні принципи:

1) рання ідентифікація й безперервне (до закінчення проекту) усунення основних ризиків;

2) концентрація на виконанні вимог замовників до виконання програм (аналіз і побудова моделі прецедентів);

3) очікування змін у вимогах, проектних рішеннях і реалізації в процесі розроблення;

4) компонентна архітектура, яка реалізується й тестується на ранніх стадіях проекту;

5) постійне забезпечення якості на всіх етапах розроблення проекту (продукту);

6) робота над проектом у згуртованій команді, ключова роль у якій належить архітекторам;

7) життєвий цикл розроблення.

3.6.2. Технологія RUP

Технологія RUP значною мірою відповідає стандартам і нормативним документам, пов'язаним із процесами життєвого циклу ПЗ й оцінюванням технологічної зрілості організацій- розробників (ISO 12207, ISO 9000, CMM і ін.). Її основними принципами є:

1. Ітераційний та інкрементний (нарощуваний) підхід до створення ПЗ.

2. Планування й керування проектом на основі функціональних вимог до системи – варіантів використання.

3. Побудова системи на базі архітектури ПЗ.

Перший принцип є визначальним. Відповідно до нього систему розробляють у вигляді декількох короткострокових мініпроектів фіксованої тривалості (від 2 до 6 тижнів), названих ітераціями. Кожна ітерація включає власні етапи аналізу вимог, проектування, реалізації, тестування, інтеграції й завершується створенням працюючої системи.

Ітераційний цикл ґрунтується на постійному розширенні й доповненні системи в процесі декількох ітерацій з періодичним зворотним зв'язком і адаптацією модулів, що додаються до ядра системи. Система постійно розростається крок за кроком, тому такий підхід називають ітераційним та інкрементним.

Загальний вигляд RUP у двох вимірах показано на рис.3.29. Горизонтальний вимір відображає час, динамічні аспекти процесів і оперує такими поняттями, як стадії, ітерації й контрольні точки. Вертикальний вимір відображає статичні аспекти процесів і оперує такими

поняттями, як види діяльності (технологічні операції), робочі продукти, виконавці й дисципліни (технологічні процеси).

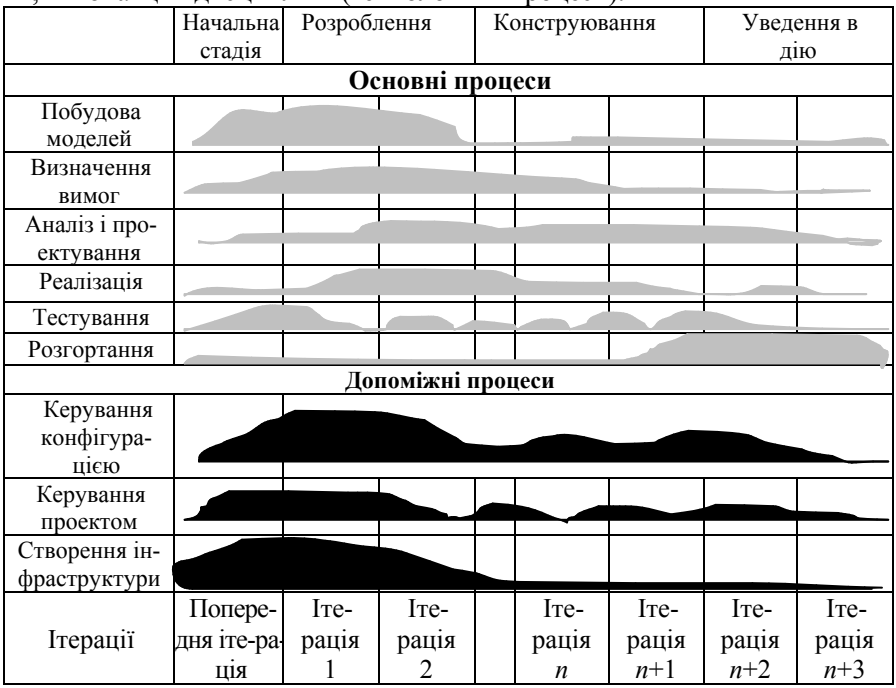


Рис.3.29. Загальне зображення RUP

Згідно з RUP життєвий цикл ПЗ розбивається на окремі цикли, у кожному з яких створюється нове покоління продукту. Кожний цикл, у свою чергу, містить чотири послідовні стадії:

– початкову (*inception*);
– розроблення (*elaboration*);
– конструювання (*construction*);
– упровадження в дію (*transition*).

Кожна стадія завершується в чітко визначеній контрольній точці (*milestone*). У цей момент часу мають досягатися важливі результати й прийматися критично важливі рішення про подальше розроблення.

1. *Початок*. На цьому етапі:

1) формуються бачення й межі проекту;
2) створюється економічне обґрунтування;

3) визначаються основні вимоги, обмеження й ключова функціональність продукту;

4) створюється базова версія моделі прецедентів;

5) оцінюються ризики.

Після завершення початкової стадії оцінюється досягнення цілей життєвого циклу (Lifecycle Objective Milestone – LOM), що припускає угоду зацікавлених сторін про продовження проекту. Результатами початкової стадії є:

– загальний опис системи: основні вимоги до проекту, його характеристики й обмеження;

– початкова модель варіантів використання (ступінь готовності – 10–20%);

– початковий проектний глосарій (словник термінів);

– початковий бізнес-план;

– план проекту, що відображає стадії й ітерації;

– один або декілька прототипів.

2. *Розроблення або проектування*. На етапі проектування виконуються аналіз предметної галузі й побудова виконавчої архітектури. Цей етап містить:

1. документування вимог (включаючи детальний опис для більшості прецедентів);

2. спроектовану, реалізовану й тестовану виконавчу архитектуру;

3. оновлене економічне обґрунтування й більш точні оцінки строків і вартості;

4. знижені основні ризики.

Результатами стадії розроблення є:

– модель варіантів використання (завершена принаймні на 80%), що визначає функціональні вимоги до системи;

– перелік додаткових вимог, включаючи вимоги нефункціонального характеру й вимоги, не пов’язані з конкретними варіантами використання;

– опис базової архітектури майбутньої системи;

– працюючий прототип;

– уточнений бізнес-план;

– план розроблення всього проекту, що відображає ітерації й критерії оцінювання для кожної ітерації.

Найважливішим результатом стадії розроблення є опис базової архітектури майбутньої системи. Ця архітектура включає:

– модель предметної галузі, яка відображає розуміння бізнесу і є відправним пунктом для формування основних класів предметної галузі;

– технологічну платформу, що визначає основні елементи технології реалізації системи і їх взаємодію.

Ця архітектура є основою всього подальшого розроблення, вона є своєрідним проектом для наступних стадій. Надалі неминучі незначні зміни в деталях архітектури, однак істотні зміни малоймовірні.

Стадія розроблення займає близько п'ятої частини загальної тривалості проекту. Основними ознаками завершення стадії розроблення є дві події:

– розробники в стані оцінити з досить високою точністю, скільки часу буде потрібно на реалізацію кожного варіанта використання;

– ідентифіковані всі найбільші ризики, і ступінь розуміння найбільш важливих з них такий, що відомо, як впоратися з ними.

3. *Конструювання*. Під час цієї фази реалізується більша частина функціональності продукту. Фаза побудови завершується першим зовнішнім релізом системи й початковою функціональною готовністю (*Initial Operational Capability*).

Сутність планування полягає у визначенні послідовності ітерацій конструювання й варіантів використання, реалізованих на кожній ітерації. Ітерації на стадії конструювання є одночасно інкрементними й повторюваними:

– ітерації є інкрементними відповідно до тієї функції, яку вони виконують. Кожна ітерація додає чергові конструкції до варіантів використання, реалізації під час попередніх ітерацій;

– ітерації є повторюваними стосовно розроблювального коду. На кожній ітерації деяка частина існуючого коду листується для надання йому більшої гнучкості.

Результатом стадії конструювання є продукт, готовий до передавання кінцевим користувачам. Він зокрема містить:

– програмне забезпечення, інтегроване на необхідних платформах;

– посібник користувача;

– опис поточної реалізації.

4. *Упровадження*. Під час фази впровадження створюється фінальна версія продукту й передається від розробника до замовника. Це включає в собе програму бета-тестування, навчання користувачів, а також визначення якості продукту. Якщо якість не відповідає очікуванням користувачів або критеріям, установленим у фазі «Початок», фаза «Упровадження» повторюється знову. Виконання всіх цілей означає досягнення готового продукту і завершення повного циклу розроблення.

Призначенням стадії впровадження в дію є передавання готового продукту в розпорядження користувачів. Ця стадія включає такі кроки:

– бета-тестування, що дозволяє переконатися у відповідності нової системи очікуванням користувачів;

– паралельне функціонування з існуючою системою, яка підлягає поступовій заміні;

– конвертування баз даних;

– оптимізацію продуктивності;

– навчання користувачів і фахівців служби супроводження.

Кроки класифікують на такі етапи:

Етап формування вимоги. На цьому етапі визначаються функціональні, технічні і прикладні вимоги до проекту. На основі вимог замовника й користувачів система описується так, щоб досягти розуміння між ними й проектною групою. Інформація збирається з урахуванням особливостей існуючих систем і документів, підготовлених замовником, і включає:

– модель ПЗ;

– модель схем використання з описом функціональних і загальних вимог у формі результатів опитування, наборів діаграм і детального опису кожної схеми;

– дизайн і прототип інтерфейсу користувача для кожного актора;

– список вимог, які не ставляться до конкретних схем використання.

Етап аналізу. Сформульовані вимоги уточнюються й відображаються в моделі сценаріїв використання. Крім того, створюється аналітична модель системи, яка включає формалізацію для аналізу

внутрішньої структури системи, визначення класів і перетворення цієї моделі в проектні концепції й схему їх реалізації.

Етап проектування. На цьому етапі уточнюються класи й описуються їх чотири рівні: інтерфейсу користувачу, бізнесів-рішень, рівня доступу й рівня даних. Створювана проектна модель системи складається зі структури підсистем; їх розподілу між рівнями; інтерфейсів класів і об'єктів; зв'язків класів з вузлами розгортання (модель розгортання).

Етап реалізації – це побудова прототипу з компонентів; створення тестів за схемами використання; тестування й інтеграція компонентів; перевірка архітектури; перехід до наступної ітерації. Із кожної ітерації тестова модель уточнюється вилученням неактуальних тестів, створенням схеми регресійного тестування й додаванням тестів для складальних компонентів. Кожний тест створюється за допомогою варіантів використання й реалізує конкретний метод перевірки функцій системи за вхідними даними.

Методологія RUP оформлена й розміщена в Web базі знань пошукової системи. В ній регламентовані етапи розроблення ПЗ, документи й інструментальні засоби для забезпечення кожного етапу життєвого циклу.

Іноді до наведених етапів додають ще такі процеси:

– тестування;

– розгортання;

– конфігураційне керування й керування змінами;

– керування проектом;

– керування середовищем.

Тестування дозволяє визначати й контролювати якість створюваних продуктів, зокрема:

– наскільки якісно здійснена інтеграція компонентів і підсистем;

– чи реалізовані всі вимоги до системи;

– чи всі виявлені помилки усунуті до того, як система буде розгорнута на устаткуванні кінцевого користувача.

Розгортання є процесом, у ході якого розроблювальний продукт доставляється до кінцевого користувача. У ході цього процесу розробляється нова версія системи, поширення ПЗ, його установлення на боці кінцевого користувача та його навчання навичок

ефективної роботи з поставленим ПЗ, надання послуг з технічного підтримання, бета-тестування і т. ін.

Конфігураційне керування й керування змінами дають змогу організовувати ефективну роботу з артефактами проекту, контролювати й керувати доступом до них, вести історію змін, забезпечувати ефективну взаємодію учасників проекту як у простих командах, так і в розподілених, що значно віддалені один від одного.

Керування проектом передбачає безпосереднє формування умов для ефективного ходу всього проекту, визначення керівництв і керівних принципів для планування, формування команди та моніторингу проекту, виявлення й керування ризиками, організацію роботи учасників проекту, формування бюджету, планування фаз та ітерацій.

Керування середовищем дозволяє підтримувати всіх учасників проекту щодо вибору інструментарію і його придбання, настроювання й установлення, конфігурування процесу, дороблення й адаптації методології, використовуваної для ведення проекту, навчання.

Статичний аспект RUP містить чотири основні елементи:

– роль;

– види діяльності;

– робочі продукти;

– дисципліна.

Поняття «роль» визначає поведінку й відповідальність особи або групи осіб, що складають проектну команду. Одна особа може відіграти в проекті багато різних ролей.

Під *видом діяльності* конкретного виконавця розуміють одиницю виконуваної ним роботи. Вид діяльності відповідає поняттю технологічної операції. Він має чітко визначену мету, що зазвичай виражається термінами одержання або модифікації деяких *робочих продуктів*, таких, як модель, елемент моделі, документ, вихідний код або план. Кожний вид діяльності пов’язаний з конкретною роллю. Тривалість виду діяльності становить від декількох годин до декількох днів, виконується зазвичай одним виконавцем і породжує тільки один або досить невелику кількість робочих продуктів. Будь-який вид діяльності має бути елементом процесу планування. Прикладами видів діяльності можуть бути планування ітерації, визначення варіантів використання й діючих осіб, виконання тесту на

продуктивність. Кожний вид діяльності супроводжується набором директив методики, що являють собою, виконання технологічних операцій.

Дисципліна відповідає поняттю технологічного процесу і являє собою послідовність дій, що приводить до значущого результату.

У межах RUP виокремлено шість основних дисциплін: побудову бізнес- моделей; визначення вимог; аналіз і проектування; реалізацію; тестування; розгортання і три допоміжні: керування конфігурацією й змінами; керування проектом та створення інфраструктури.

Методологія RUP як продукт входить до складу комплексу Rational Suite, причому кожна із зазначених вище дисциплін підтримується певним інструментальним засобом комплексу. Фізична реалізація RUP являє собою Web-Сайт, що включає такі компоненти:

– опис усіх елементів динамічного й статичного аспекту RUP;

– навігатор для всіх елементів RUP, глосарій і засіб швидкого навчання технології;

– керівництва для всіх учасників проектної команди, що охоплюють увесь життєвий цикл ПЗ.

Розрізняють два види керівництва: для осмислення процесу на верхньому рівні і у вигляді детальних настанов з повсякденної діяльності:

– настанови з використання інструментальних засобів, що входять до складу Rational Suite;

– прикладів і шаблонів проектних рішень для Rational Rose;

– шаблонів проектної документації для Soda;

– шаблонів у форматі Microsoft Word, призначених для підтримання документації за всіма процесами і діями життєвого циклу ПЗ;

– планів у форматі Microsoft Project, що відображають ітераційний характер розроблення ПЗ.

Адаптація RUP до потреб конкретної організації або проекту забезпечується за допомогою Rational Process Workbench (RPW) – спеціального набору інструментів і шаблонів для налаштування і публікації Web-сайтів на основі RUP. RPW підтримує три основні функції моделювання технологічних процесів:

– визначення процесу;

– опис процесу;

– вигляд процесу.

Бібліотека елементів процесу містить текстову інформацію про кожний елемент у моделі процесу, усі текстові сторінки RUP, а RPW – необхідні шаблони для створення нових сторінок опису. RPW генерує опис процесів, що включає текст і графіку, у вигляді Web-сайту, з'єднуючи моделі процесів і бібліотеку описів у єдине ціле.

RUP спирається на інтегрований комплекс інструментальних засобів Rational Suite. Він існує в таких варіантах:

– Rational Suite Analyststudio – призначений для визначення й керування повним набором вимог до розроблювальної системи;

– Rational Suite Developmentstudio – призначений для проектування й реалізації ПЗ;

– Rational Suite Teststudio – набір продуктів, призначених для автоматичного тестування додатків;

– Rational Suite Enterprise – підтримує повний життєвий цикл ПЗ і призначений як для менеджерів проекту, так і окремих розробників, що виконують кілька функцій у команді розробників.

Таким чином, RUP – це процес моделювання й побудови системи програмування з об'єктів із застосуванням мови UML. Він включає теоретичний і прикладний аспекти подання й тлумачення створюваних моделей для проектованої предметної галузі.

Теоретичний аспект процесу моделювання моделей систем програмування підтримується методами і поняттями формальних теорій. Формалізація моделей в RUP забезпечується засобами UML і дає змогу чітко описувати вимоги й адаптувати їх до готового продукту.

Основу процесу моделювання становлять *прецеденти* – варіанти використання для визначення вимог до системи. Головний елемент проектування – модель варіантів використання, на основі якої розробляються моделі аналізу, проектування й реалізації системи. Кожна модель аналізується на відповідність моделі варіантів використання, у яку входять вхідні дані для пошуку й специфікації класів та підсистем, для підбору й специфікації тестів, а також для планування ітерацій розроблення й інтеграції системи програмування. У процесі моделювання створюються такі моделі:

– варіантів використання, що відображують взаємодію між користувачами й системами програмування;

– аналізу, що забезпечує специфікації вимог до системи й опис варіантів використання як взаємодії між концептуальними класифікаторами;

– проектування, орієнтованого на створення статичної структури й інтерфейсів системи, реалізацію варіантів використання у вигляді набору кооперацій (взаємодій) між підсистемами, класами й інтерфейсами;

– реалізації, що включає компонент системи у вихідному вигляді мовою програмування;

– тестування;

– розміщення компонентів і виконання в операційному середовищі комп'ютерів.

Ці моделі виражаються різними видами діаграм, наприклад, у моделі варіантів – діаграмою use-case, у моделях аналізу – діаграмами класів, кооперацій і станів. Ці моделі – взаємозалежні, семантично перетинаються й визначають систему як єдине ціле. Наприклад, варіант використання у відповідній моделі може мати відношення залежності до кооперації в моделі проектування, що задає реалізацію. Деякі моделі на кожній ітерації процесу RUP уточнюються або розширюють моделі попередніх ітерацій процесу. Типи моделей і їхніх зв'язків показані на рис 3.30, кожна з моделей задається відповідними діаграмами. Наприклад, модель аналізу складається з діаграм класів, станів і кооперації.

Артефакти однієї моделі пов'язані між собою й мають бути сумісні між собою. Відносини між моделями є не повністю формальними, оскільки частини моделей специфіковані мовою метамоделі, а інші описані тільки неформально, природною мовою. Специфікації діаграм UML є також полуформальними.

Основу *моделі аналізу* становить діаграма варіантів використання, що містить у собі діаграми класів, взаємодії, що задають можливі сценарії варіантів використання в термінах взаємодії об'єктів на етапі аналізу.

Варіанти використання специфікують тип відносин між діючою особою (актором), користувачем і системою. На високому рівні абстракції вони набувають упорядкованої послідовності дій або альтернатив.

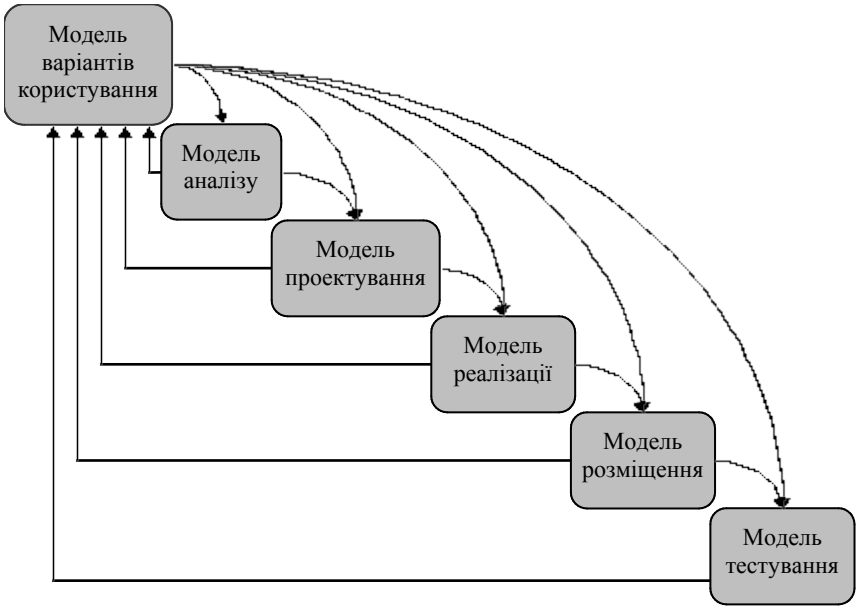


Рис 3.30. Типи моделей проектування

Варіант використання в UML – це різновид класифікатора, операціями якого є повідомлення, які отримують екземпляри конкретного варіанта використання. Методи задають реалізацію операцій у термінах послідовностей дій, виконуваних екземплярами варіанта використання.

Приклад. Нехай *uc* – варіант використання (*uc – use case*), операція яко-го виконується над обліковим записом і має таке визначення:

uc.operations = <*op1*>

op1.name = запит і відновлення облікового запису

op1.method.body = {<перевірка ідентифікації користувача, наявності сервісу, запиту про борги, відновлення облікового запису>,

<перевірка ідентифікації користувача, відхилення облікового запису>,

< перевірка ідентифікації користувача, сервісу, відхилення облікового запису >,

<перевірка ідентифікації користувача, наявності сервісу, запиту про борги, запиту на оплату, відновлення облікового запису>}

Тіло методу – процедура, що специфікує реалізацію операцій у вигляді послідовності дій *op.method.body* або *op.action Sequence*. Між іменами дій варіанта використання й іменами дій у кооперації встановлюється відображення, яке забезпечує гнучкість у процесі розроблення й модифікацію імен дій. Між кооперацією й варіантом використання *uc* створюється відношення реалізації.

Варіант використання реалізується кооперацією, якщо класифікатори у ній взаємодіють для забезпечення поводження. Якщо кооперація має більш складне поводження, ніж специфіковане варіантом використання, то цей варіант використання – часткова специфікація поводження кооперації. Варіанти використання специфікують дії, які є видимими за межами системи, але не специфікують внутрішніх дій (створення й видалення екземплярів класифікаторів, взаємодія між екземплярами класифікаторів і т. ін.).

Визначення розширення включає як умову розширення, так і посилання на точку розширення в цільовому варіанті використання, що є позицією усередині варіанта використання. Як тільки екземпляр варіанта використання досягає точки розширення, на яку посилається це відношення, перевіряється його умова.

Якщо умова виконується, послідовність, що задовольняє умови в екземплярі варіанта використання, розширюється таким чином, щоб включити послідовність розширюваного варіанта використання.

Із практичного погляду RUP являє собою впорядкований набір кроків і етапів життєвого циклу, які виконуються ітеративно. Цей процес є керованим як щодо задання вимог, так і реалізації функціональних можливостей ПЗ із заданим рівнем якості й гарантованих витрат згідно з графіком робіт. Оцінка якості всіх кроків і дій учасників процесу ґрунтується на певних критеріях.

Кроки виконання RUP керуються прецедентами, тобто технологічним маршрутом від ділового моделювання й вимог до випробувань. *Екземпляр прецеденту* – це послідовність дій, виконуваних системою із спостережуваним результатом для конкретного суб'єкта. Функціональні можливості системи визначаються набором прецедентів, кожний з яких являє собою деякий потік подій. Опис пре-

цеденту визначає ті події, що відбудуться в системі, коли прецедент буде виконаний. Кожний прецедент орієнтований на завдання, яке він повинен виконати. Набір прецедентів установлює всі можливі маршрути виконання системи. Прецеденти відіграють роль у кожному з п'яти основних робіт процесу: формуванні вимог, аналізі, проектуванні, реалізації й випробуванні.

Методологія RUP містить п'ять основних етапів, виконуваних на всіх фазах процесу розроблення системи програмування. Завершення цих етапів називається ітерацією, яка закінчується випуском проміжного продукту. На кожній ітерації цикл повторюється, починаючи зі збирання й уточнення вимог. RUP використовує ітеративну модель розроблення. Наприкінці кожної ітерації (яка в ідеалі триває від 2 до 6 тижнів) проектна команда повинна досягти запланованих на цю ітерацію цілей, створити або доробити проектні артефакти й отримати проміжну, але функціональну версію кінцевого продукту. Ітеративна розробка дозволяє швидко реагувати на мінливі вимоги, виявляти й усувати ризики на ранніх стадіях проекту, а також ефективно контролювати якість створюваного продукту.

3.6.3. Найважливіші акценти RUP

Головна мета будь-якої організації, що займається створенням інформаційних систем – працювати ефективніше, а отже, швидше створювати більш якісні продукти й досягати бізнес-переваг від успішного ведення проектів. Упровадження передової методології, подібної до RUP, дозволяє гарантувати вироблення і подальший розвиток в організації необхідних для цього навичок.

Однак упровадження методології – не настільки простий процес, як це може видатися на перший погляд. Дуже важливо, прагнучи до якомога ефективнішого ведення проектів, не зруйнувати те, чого вже досягнуто. Особливість методології RUP полягає в тому, що вона може бути налаштована й адаптована відповідно до особливостей і вимог організації-розробника, при цьому варіанти впровадження RUP можуть варіюватися залежно від конкретних умов.

Для спрощення переходу до методології RUP допускається поступове його впровадження. Але при цьому RUP акцентує увагу

на декількох найважливіших елементах, без яких складно гарантувати успіх проекту.

Опис проекту в методології має такий шаблон:

1. ***Назва*** <коротка фраза у вигляді дієслова в невизначеній формі зробленого виду, що відображає мету проекту>

2. ***Контекст використання*** <уточнення мети, у разі необхідності – умови її нормального завершення>.

3. ***Галузь дії*** <посилання на межі проекту>. Наприклад, підсистема бухгалтерського обліку.

4. ***Рівень*** <один із трьох: узагальнений, мета користувача, підфункції>. Автор задає трирівневу класифікацію вимог, що в цілому відповідає класифікації вимог до бізнес-вимог, вимог користувачів і функціональних вимог.

Після опису цих чотирьох пунктів переходять до описання акторів та їх взаємодії.

Основна діюча особа <ім'я ролі основного актора або його опис>.

Учасники й інтереси <список інших акторів-учасників прецеденту з указанням на їх інтереси>.

Передумова <те, що як очікується, вже відбувається>.

Мінімальні гарантії <що гарантується акторам-учасникам>. Наприклад, у випадку невдалої транзакції всі дані, що були в системі до її початку, зберігаються незмінними.

Гарантії успіху <що отримають актори-учасники у випадку успішного досягнення мети>.

Тригер <пристрій, що «запускає» варіант використання, зазвичай – подія в часі >.

Основний сценарій <тут перераховуються кроки основного сценарію, починаючи від тригера й закінчуючи досягненням гарантії успіху>.

Формат опису: <Номер кроку> <Опис дії>

Розширення <тут послідовно описуються всі альтернативні сценарії>. Кожна з альтернатив залежить від кроку основного сценарію.

Формат опису: <Номер кроку. Номер розширення> <Умова>:<Дія або посилання на підпорядкований варіант використання>.

Будь-який з кроків основного сценарію може мати одне або більше розгалужень. Кожне розгалуження оформляється у вигляді розширення. У блоці «Розширення» всі розширення описуються послідовно.

Якщо альтернативний сценарій не вдається описати одним рядком, застосовується наступний формат.

Починаючи з рядка, що випливає після опису розширення, описуються його дії у форматі основного сценарію:

<Номер кроку. Номер розширення. Номер кроку розширення> <Дія>

Опис розширення закінчується описом виходу з розширення. Основні варіанти виходу з розширення:

– повернення до чергового за номером кроку основного сценарію,
– закінчення прецеденту,
– перехід до іншого кроку основного сценарію.

Список змін у технології й даних <що гарантується акторам-участникам>. Наприклад, у випадку невдалої транзакції всі дані, що були в системі до її початку, зберігаються незмінними.

Допоміжна інформація <додаткова інформація, корисна для опису варіанта використання>.

Табличні подання варіанта використання. Іноді зручно поміщувати сценарії варіантів використання в таблицю (табл. 3.5). Інформація набуває більш структурованого вигляду (табл. 3.5–3.6).

Таблиця 3.5

Зразок таблиці з двох стовпчиків

Актор	Дія
Користувач	Формує запит на пошук замовлень
Система	Відображає список замовлень
Користувач	Вибирає необхідне замовлення
Система	Показує докладну інформацію про замовлення

Таблиця 3.6

Зразок таблиці з трьох стовпчиків

Номер кроку	Користувач	Система
1	Робить запит на пошук замовлень	Відображає список замовлень
2	Вибирає необхідне замовлення	Показує докладну інформацію про замовлення

3.7. Методи об'єктно-орієнтованого аналізу і проектування програмного забезпечення.

Концептуальною основою об'єктно-орієнтованого аналізу і проектування ПЗ є об'єктна модель. Її основні принципи (абстрагування, інкапсуляція, модульна та ієрархія) і поняття (об'єкт, клас, атрибут, операція, інтерфейс та ін) найбільш чітко сформульовані Граді Бучем у його фундаментальній книжці і подальших працях.

Більшість сучасних методів об'єктно-орієнтованого аналізу і проектування основані на використанні мови UML. Уніфікована мова моделювання UML є мовою для визначення, подання, проектування та документування програмних систем, організаційно-економічних систем, технічних систем та інших систем різної природи, а також містить стандартний набір діаграм і нотацій різноманітних видів.

Стандарт UML версії 1.1, прийнятий OMG у 1997 р., містить такий набір діаграм:

Структурні (structural) моделі:

– діаграми класів (*class diagrams*) – для моделювання статичної структури класів системи і зв'язків між ними;

– діаграми компонентів (*component diagrams*) -- для моделювання ієрархії компонентів (підсистем) системи;

– діаграми розміщення (*deployment diagrams*) – для моделювання фізичної архітектури системи.

Моделі поведінки (behavioral):

– діаграми варіантів використання (*use case diagrams*) – для моделювання функціональних вимог до системи (у вигляді сценаріїв взаємодії користувачів із системою);

– діаграми взаємодії (*interaction diagrams*):

– діаграми послідовності (*sequence diagrams*) і кооперативні діаграми (*collaboration diagrams*) – для моделювання процесу обміну повідомленнями між об'єктами;

– діаграми станів (*statechart diagrams*) – для моделювання поведінки об'єктів системи при переході з одного стану в інший;

– діаграми діяльності (*activity diagrams*) – для моделювання поведінки системи в межах різних варіантів використання, або потоків керування.

Діаграми варіантів використання показують взаємодії між варіантами використання та дійовими особами, відображаючи функціональні вимоги до системи з позиції користувача. Мета побудови діаграм варіантів використання – це документування функціональних вимог у самому загальному вигляді, тому вони мають бути гранично простими.

Варіант використання являє собою послідовність дій (транзакцій), виконуваних системою у відповідь на подію, що ініціюється деякими зовнішнім об'єктом (дійовою особою). Варіант використання описує типову взаємодію між користувачем і системою і відображає уявлення про поведінку системи з позиції користувача. У простому випадку варіант використання визначається в процесі обговорення з користувачем тих функцій, які він хотів би реалізувати, чи цілей, які він ставить щодо розробленої системи.

Діаграма варіантів використання є найбільш загальним поданням функціональних вимог до системи. Подальше проектування системи потребує більш конкретного опису документа – сценарія варіантів використання або потоку подій (*flow of events*). Сценарій документує докладно процес взаємодії дійової особи із системою, що реалізується варіантами використання. Основний потік подій описує нормальний хід подій (якщо немає помилок). Альтернативні потоки описують відхилення від нормального перебігу подій (помилкові ситуації) та їх оброблення.

Переваги моделі варіантів використання:

– визначає користувачів і межі системи, а також системний інтерфейс;

– зручна для спілкування користувачів з розробниками;

– використовується для написання тестів;

– є основою написання для користувача документації;

– добре вписується в будь-які методи проектування (як об'єктно-орієнтовані, так і структурні).

Діаграми взаємодії описують поведінку взаємодійних груп об'єктів (за варіантами використання або деякої операції класу). Діаграма взаємодії охоплює поведінку об'єктів у межах лише одного потоку подій варіанта використання. На такій діаграмі відображаються об'єкти і ті повідомлення, якими вони обмінюються між собою. Є два види діаграм взаємодії: діаграми послідовності та кооперативні діаграми.

Діаграми послідовності відображають хронологію подій, що відбуваються відповідно до варіантів використання, а кооперативні діаграми концентрують увагу на зв'язках між об'єктами.

Діаграма класів визначає типи класів системи і різні статичні зв'язки між ними. Діаграми класів містять також атрибути класів, операції класів і обмеження, які накладаються на зв'язок між класами. Вид та інтерпретація діаграми класів істотно залежить від рівня абстракції: класи можуть виражати сутності предметної галузі (у процесі аналізу) або елементи програмної системи (у процесах проектування та реалізації).

Діаграми станів визначають всі можливі стани, в яких може перебувати конкретний об'єкт, а також процес зміни станів об'єкта в результаті настання деяких подій. Діаграми станів не треба створювати для кожного класу, вони застосовуються тільки в складних випадках. Якщо об'єкт класу може перебувати в декількох станах і в кожному з них поводиться по-різному, для нього може знадобитися така діаграма.

Діаграми діяльності, на відміну від більшості інших засобів UML, запозичують ідеї з декількох різних методів, зокрема, методу моделювання станів SDL і мереж Петрі. Ці діаграми особливо корисні в описі поведінки, що включає велику кількість паралельних процесів. Діаграми діяльності є також корисними під час паралельного програмування, оскільки можна графічно зобразити всі гілки і визначити, коли їх необхідно синхронізувати.

Діаграми діяльності можна застосовувати для опису потоків подій у варіантах використання. За допомогою текстового опису можна достатньо детально розповісти про потік подій, але в складних і заплутаних потоках з безліччю альтернативних гілок буде важко зрозуміти логіку подій. Діаграми діяльності надають ту ж ін-

формацію, що й текстовий опис потоку подій, але в наочній графічній формі.

Діаграми компонентів моделюють фізичний рівень системи. На них зображаються компоненти ПЗ і зв'язки між ними. На такій діаграмі зазвичай виділяють два типи компонентів: виконувані компоненти і бібліотеки коду.

Кожен клас моделі (або підсистема) перетворюється в компонент початкового коду. Між окремими компонентами відображаються залежності, що відповідають залежностям на етапі компіляції або виконання програми.

Діаграми компонентів застосовують ті учасники проекту, які відповідають за компіляцію і побудову системи. Вони потрібні там, де починається генерація коду.

Діаграма розміщення відображає фізичні взаємозв'язки між програмними і апаратними компонентами системи; вказує на розміщення об'єктів і компонентів у розподіленій системі та на фізичне розміщення мережі різних компонентів і в ній. Її основними елементами є вузол (обчислювальний ресурс) і з'єднання – канал взаємодії вузлів (мережа).

Діаграму розміщення використовує менеджер проекту, користувачі, архітектор системи й експлуатаційний персонал, щоб зрозуміти фізичне розміщення системи і розміщення її окремих підсистем.

Мова UML має механізми розширення, призначені адаптації мови моделювання до конкретних потреб розробника, не змінюючи при цьому його метамодель. Наявність механізмів розширення принципово відрізняє UML від таких засобів моделювання, як IDEF0, IDEF1X, IDEF3, DFD і ERM. Вони сильно типізуються (за аналогією з мовами програмування), оскільки не допускають довільної інтерпретації семантики елементів моделей. UML, допускаючи таку інтерпретацію (здебільшого за рахунок стереотипів), є мовою, що слабо типізується. Її механізмами розширення є:

– стереотипи;
– теговані (іменовані) значення;
– обмеження.

Стереотип – це новий тип елемента моделі, який визначається на основі вже існуючого елемента. Стереотипи розширюють нотацію моделі і можуть застосовуватися до будь-яких елементів моделі.

Стереотипи класів – це механізм, що дозволяє розділяти класи на категорії. Розробники ПЗ можуть створювати свої власні набори стереотипів, формуючи тим самим спеціалізовані підмножини UML (наприклад, для опису бізнес-процесів, Web-додатків, баз даних і т. ін.). Такі підмножини (набори стереотипів) у стандарті мови UML називають профілями мови.

Іменоване значення – це пара рядків «тег = значення», або «ім'я = вміст», у яких зберігається додаткова інформація про якийнебудь елемент системи, наприклад, час створення, статус розроблення або тестування, час закінчення роботи над ним і т. ін.

Обмеження – це семантичне обмеження, що має вигляд текстового виразу природною або формальною мовою (OCL – Object Constraint Language), який неможливо виразити за допомогою нотації UML.

Починаючи працювати над уніфікацією методів створення ПЗ, Р. Буч, Дж. Румбах і А. Джекобсон сформулювали такі вимоги до мови моделювання:

– дозволяти моделювати не лише програмне забезпечення, але й ширші класи систем і бізнес-додатків з використанням об'єктно-орієнтованих понять;

– забезпечувати взаємозв'язок між базовими поняттями для моделей концептуального фізичного рівнів;

– забезпечувати масштабованість моделей, що є важливою особливістю складних багатоцільових систем;

– мова має бути зрозумілою аналітикам та програмістам і підтримуватися спеціальними інструментальними засобами, реалізованими на різних комп'ютерних платформах.

На основі технології UML Microsoft, Rational Software та інші постачальники засобів розроблення програмних систем створили єдину інформаційну модель, яка отримала назву UML Information Model. Передбачається, що за цією моделлю різні програми, що підтримують ідеологію UML, зможуть обмінюватися компонентами й описами, що дозволить створити стандартний інтерфейс між засобами розроблення додатків і засобами візуального моделювання.

Натепер уже розроблено засоби візуального програмування на основі UML, що забезпечують інтеграцію, включаючи пряму й обернену генерацію кодів програм, з найбільш поширеними мовами і середовищами програмування, такими як MS Visual C++, Java,

Object Pascal/delphi, Power Builder, MS Visual Basic, Forte, Ada, Smalltalk.

Запитання для самоперевірки

1. Які класи комп'ютерних програм використовуються в авіації? Які з цих програм належать до системних, а які до прикладних?
2. Укажіть етапи побудови програмних модулів.
3. Що таке життєвий цикл ПЗ? З яких етапів складається та як стосується технологій побудови ПЗ?
4. Що таке мова програмування і навіщо вона потрібна програмістам?
5. Як класифікують мови програмування?
6. Що таке система програмування та з яких модулів вона складається?
7. Що таке компілятор? З чого складається процес компіляції? Які існують види компіляторів?
8. Які відмінності між транслятором і компілятором? Назвіть мови-транслятори, мови-інтерпретатори. Чи існують мови, які є і компіляторами, і інтерпретаторами?
9. Навіщо потрібні інтегровні програмні середовища? З яких модулів вони складаються? Які функції цих модулів?
10. Назвіть особливості та складові середовищ швидкого програмування.
11. Що таке архітектура ПЗ?
12. Які найбільш популярні типи архітектури ПЗ?
13. Що таке трирівнева архітектура і в яких випадках її доцільно використовувати?
14. Якими поняттями зазвичай описують програмну архітектуру?
15. Які принципи покладено в основу SADT?
16. Для чого використовують SADT?
17. Розкладіть згідно з методологією SADT процес роботи авіаційного диспетчера.
18. Що таке зв'язок у SADT та які його типи?
19. Що таке потік даних та як він моделюється?
20. Які обмеження на діаграмі містить методологія DFD?
21. Яка сутність методології RUP?

22. На які стадії згідно з методологією RUP поділяється розроблення ПЗ?

23. Які додаткові процеси містить ця модель?

24. У чому полягає процес посадки літака в шаблоні RUP.

4. ПРИКЛАДНІ ПРОГРАМНІ СЕРЕДОВИЩА

4.1. Сумісність і множинні прикладні середовища. Способи реалізації прикладних програмних середовищ

4.1.1. Множинні прикладні середовища

Тоді як деякі ідеї (наприклад, об'єктно-орієнтований підхід) безпосередньо стосуються лише розробників і лише побічно впливають на кінцевого користувача, концепція множинних прикладних середовищ дозволяє користувачу виконувати на власній ОС програми, написані для інших ОС та інших процесорів.

Множинні прикладні середовища забезпечують сумісність власної ОС з додатками, написаними для інших ОС і процесорів, на двійковому рівні, а не на рівні вихідних текстів.

Реалізуючи множинні прикладні середовища, розробники стикаються із суперечливими вимогами. З одного боку, завданням кожного прикладного середовища є виконання програми по можливості так, як якби вона виконувалася б на «рідній» ОС. Але потреби цих програм можуть конфліктувати з конструкцією сучасної ОС. Спеціалізовані драйвери пристроїв можуть суперечити вимогам безпеки. Можуть конфліктувати схеми керування пам'яттю і віконні системи. Економічні питання (наприклад, вартість ліцензування програм і загроза судового переслідування) також можуть впливати на дизайн чужих прикладних середовищ. Але найбільшою потенційною проблемою є продуктивність – прикладне середовище має виконувати програми з прийнятною швидкістю.

Цю вимогу не можуть задовольняти широковикористовувані раніше емулювальні системи. Для скорочення часу на виконання чужих програм прикладні середовища використовують імітацію програм на рівні бібліотек. Ефективність цього підходу зумовлюється тим, що більшість нинішніх програм працюють під керуванням GUI типу Windows, Mac або UNIX Motif, при цьому додатки витрачають велику частку часу на вироблення деяких передбачених дій. Вони безперервно виконують виклики бібліотек GUI для маніпулювання вікнами та інших пов'язаних з GUI дій, що дозволяє прикладним середовищам відшкодувати час, витрачений на емуляцію команди за командою. Ретельно створене прикладне середовище має в своєму

складі бібліотеки, що імітують внутрішні бібліотеки GUI, але написані за допомогою «рідного» коду, тобто вона сумісна з програмним інтерфейсом іншої ОС. Інколи такий підхід називають трансляцією для того, щоб відрізняти його від повільнішого процесу емуляції кодів за однією командою за один раз.

З позиції використання прикладних середовищ переважає спосіб написання програм, згідно з яким програміст для виконання деякої функції звертається з викликом до ОС, а не намагається ефективніше реалізувати еквівалентну функцію, самостійно, працюючи безпосередньо з апаратурою. Програмісти не стануть «звертатися до металу», якщо бібліотеки міститимуть потужні і складні програми, до яких набагато простіше звертатися, ніж писати самому.

Модульність ОС нового покоління дозволяє набагато легше підтримувати множинні прикладні середовища. На відміну від старих ОС, що складаються з одного великого блока для всіх практичних застосувань, довільно розбитого на частини, нові системи є модульними, з чітко певними інтерфейсами між складовими. Це робить створення додаткових модулів, що об'єднують емуляцію процесора і трансляцію бібліотек, значно простішим.

До вдосконалених ОС, що явно містять засоби множинних прикладних середовищ, належать: IBM Os/2 2.x і Workplace OS, Microsoft Windows NT, Poweropen компанії Poweropen Association і версії UNIX від Sun Microsystems, IBM і Hewlett-packard. Крім того, деякі компанії переробляють свої інтерфейси користувача у вигляді модулів прикладних середовищ, а інші постачальники пропонують продукти для емуляції і трансляції прикладних середовищ, що працюють прикладними програмами.

Існує багато різних стратегій зі втілення ідеї множинних прикладних середовищ, і деякі з цих стратегій діаметрально протилежні. В разі UNIX транслятор прикладних середовищ зазвичай виконують, як і інші прикладні програми, плаваючим на поверхні ОС. У сучасніших ОС типу Windows NT або Workplace OS модулі прикладного середовища виконуються тісніше пов'язаними з ОС, хоча і мають, як і раніше, високу незалежність. А в Os/2 з її простішою, слабоструктурованою архітектурою засоби організації прикладних середовищ вбудовані глибоко в ОС.

Використання множинних прикладних середовищ забезпечить користувачам велику свободу вибору ОС і легший доступ до якіснішого ПЗ.

Тоді як багато архітектурних особливостей ОС безпосередньо стосуються лише системних програмістів, концепція множинних прикладних середовищ безпосередньо пов'язана з потребами кінцевих користувачів. Можливість ОС виконувати додатки, написані для інших ОС, називають *сумісністю*.

4.1.2. Способи реалізації прикладних програмних середовищ

Створення повноцінного прикладного середовища, повністю сумісного із середовищем іншої ОС, є досить складним завданням, пов'язаним із структурою ОС. Існують різні варіанти побудови множинних прикладних середовищ, що вирізняються як особливостями архітектурних рішень, так і функціональністю (рис. 4.1). У багатьох версіях ОС UNIX транслятор прикладних середовищ реалізується у вигляді звичайного застосування. У Windows NT прикладні середовища виконані у вигляді серверів, а в Os/2 організація прикладних середовищ вбудована в ОС (рис. 4.2).

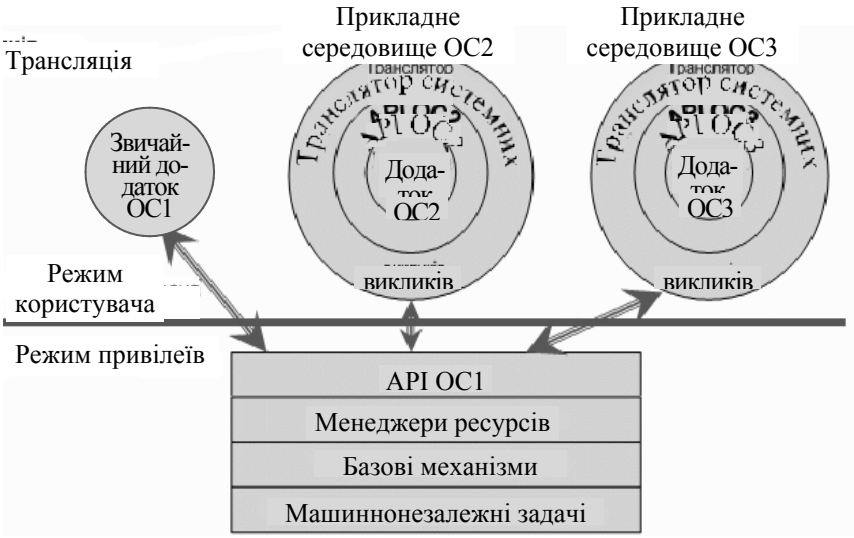


Рис. 4.1. Способи реалізації прикладних програмних середовищ

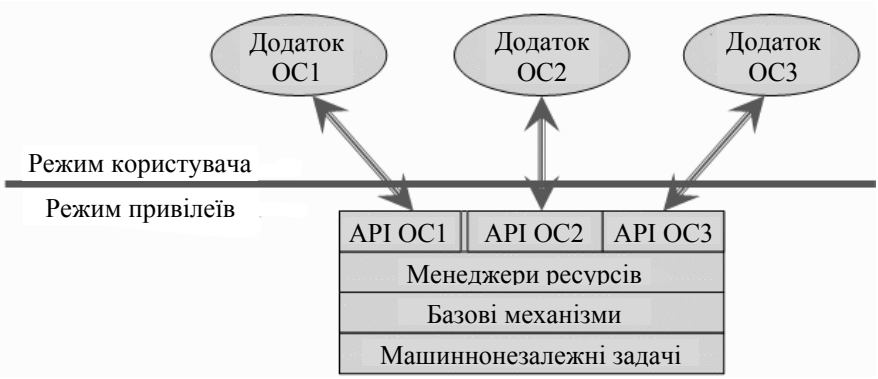


Рис. 4.2. Організація прикладних середовищ вбудована в ОС

4.2. Використання прикладних програмних середовищ для проектування програмних засобів оброблення інформаційно-технологічних процесів

Процес бізнес-моделювання можна реалізувати за різними методиками, що вирізняються насамперед розумінням того, що являє собою модельована організація. Відповідно до різних уявлень про організацію методики поділяють на об'єктні й функціональні (структурні).

Згідно з об'єктними методиками модельована організація розглядається як набір взаємодійних об'єктів – виробничих одиниць. Об'єкт визначається як реальність – предмет або явище з чітко визначеною поведінкою. Метою застосування цієї методики є виділення об'єктів, тобто організація, і розподіл між ними відповідальностей за їх дії.

Відповідно до функціональних методик, найбільш відомою з яких є методика IDEF, організація розглядається як набір функцій, що перетворює потік інформації у вихідний потік. Процес перетворення інформації споживає певні ресурси. Основна відмінність від об'єктної методики полягає в чіткому відділенні функцій (методів оброблення даних) від самих даних.

Із погляду бізнес-моделювання обидва підходи мають свої переваги. Об'єктний підхід дозволяє будувати більш стійку до змін систему, добре корелюється зі структурами організації. Функціо-

нальне моделювання застосовують у тих випадках, коли організаційна структура змінюється або оформлена неналежним чином. Підхід від виконуваних функцій інтуїтивно краще розуміють виконавці при отриманні від них інформації про їх поточну роботу.

4.2.1. Функціональна методика IDEF0

Методологію IDEF0 можна вважати наступним етапом розвитку добре відомої графічної мови опису функціональних систем SADT. Історично IDEF0 як стандарт був розроблений в 1981 р. у межах великої програми автоматизації промислових підприємств ICAM. Сім'я стандартів IDEF успадкувала позначення від назви цієї програми (IDEF = Icam DEFinition); остання редакція – грудень 1993 р. – виконана національним інститутом за стандартами і технологіями США (NIST).

Метою методики є побудова функціональної схеми досліджуваної системи, що описує всі необхідні процеси з точністю, достатньою для однозначного моделювання діяльності системи.

В основу методології покладено чотири основні поняття: функціональний блок, інтерфейсна дуга, декомпозиція, глосарій.

Функціональний блок (*Activity Box*) являє собою деяку конкретну функцію в межах цієї системи. За вимогами стандарту назва кожного функціонального блока має бути сформульована відповідно до дієслівного способу (наприклад, «реалізовувати послуги»). На діаграмі (рис. 4.3) функціональний блок зображується прямокутником. Кожна з чотирьох сторін функціонального блока має своє певне значення (роль):

– верхня – «Керування»;
– ліва – «Вхід»;
– права – «Вихід»;
– нижня – «Механізм».

Інтерфейсна дуга відображає елемент системи, яка обробляється функціональним блоком або справляє інший вплив на функцію, подану цим функціональним блоком. Інтерфейсні дуги часто називають потоками або стрілками.

За допомогою інтерфейсних дуг зображують різні об'єкти, визначають процеси, що відбуваються в системі. Такими об'єктами

можуть бути елементи реального світу (деталі, вагони, співробітники і т. ін.) або потоки даних та інформації (документи, дані, інструкції тощо).

Рис. 4.3. Функціональний блок

Залежно від того, якій зі сторін функціонального блока підходить інтерфейсна дуга, її називають «вхідною», «вихідною» чи «керувальною».

Будь-який функціональний блок за вимогами стандарту повинен мати, принаймні, одну керувальну інтерфейсну дугу і одну вихідну, оскільки кожен процес має відбуватися за якимось правилом (що відображається керувальною дугою) і видавати певний результат (виходить дуга), інакше він позбавлений сенсу.

Обов'язкова наявність керувальних інтерфейсних дуг є одною з головних відмінностей стандарту IDEF0 від інших методологій класів DFD (Data Flow Diagram) і WFD (Work Flow Diagram).

Декомпозиція (*Decomposition*) є основним поняттям стандарту IDEF0. Принцип декомпозиції застосовується для розбиття складного процесу на складові його функції. При цьому рівень деталізації процесу визначає безпосередньо розробник моделі.

Декомпозиція дозволяє поступово і структуровано подавати модель системи у вигляді ієрархічної структури окремих діаграм, що робить її менш важкою і легко засвоюваною.

Останнім з понять IDEF0 є глосарій (Glossary). Для кожного з елементів IDEF0 – діаграм, функціональних блоків, інтерфейсних дуг – за існуючим стандартом має створюватися і підтримуватися набір відповідних визначень, ключових слів тощо, які характеризують об'єкт, що міститься в елементі. Цей розділ називається глоса-

рієм і є описом суті елемента. Глосарій гармонійно доповнюється графічною мовою, забезпечуючи діаграми необхідною додатковою інформацією.

Модель IDEF0 завжди починається з подання системи як єдиного цілого – одного функціонального блока з інтерфейсними дугами, що виходять за його межі. Така діаграма з одним функціональним блоком називається контекстною діаграмою.

У пояснювальному тексті до тематичної діаграми має бути зазначена мета (*Purpose*) побудови діаграми у вигляді короткого опису і зафіксована точка зору (*Viewpoint*).

Визначати та формалізувати мету розроблення IDEF0-моделі дуже важливо. Фактично мета визначає відповідні аспекти в досліджуваній системі, на яких необхідно фокусувати увагу в першу чергу.

Точка зору визначає основний напрям розвитку моделі і рівень необхідної деталізації. Чітке фіксування точки зору дозволяє розвантажити модель без потреби деталізувати і досліджувати окремі елементи, виходячи з обраної точки зору на систему. Правильний вибір точки зору істотно скорочує тимчасові витрати на побудову кінцевої моделі.

Виділення підпроцесів. У процесі декомпозиції функціональний блок, який у тематичній діаграмі відображає систему як єдине ціле, піддається деталізації на іншій діаграмі. Діаграма другого рівня містить функціональні блоки, що відображають головні підфункції функціонального блока контекстної діаграми, і є дочірньою (*Child Diagram*) відносно нього (кожен з функціональних блоків, що належать дочірній діаграмі, відповідно називається дочірнім блоком – *Child Box*). У свою чергу, функціональний блок – предок – називається батьківським блоком відносно дочірньої діаграми (*Parent Box*), а діаграма, до якої він належить, – батьківською діаграмою (*Parent Diagram*). Кожна з підфункцій дочірньої діаграми може бути далі деталізована аналогічною декомпозицією відповідного її функціонального блока. У кожному випадку декомпозиції функціонального блока інтерфейсні дуги, що входять у цей блок або виходять з нього, фіксуються на дочірній діаграмі. Цим досягається структурна цілісність IDEF0-моделі.

Іноді окремі інтерфейсні дуги вищого рівня немає сенсу розглядати на діаграмах нижнього рівня, або навпаки – окремі дуги

нижнього рівня відображати на діаграмах більш високих рівнів – це тільки перевантажуватиме діаграми і ускладнюватиме їх сприйняття. Для вирішення подібних завдань у стандарті IDEF0 передбачено поняття тунелювання. Позначення «тунеля» (*Arrow Tunnel*) у вигляді двох круглих дужок навколо початку інтерфейсної дуги означає, що ця дуга не була успадкована від функціонального батьківського блока і з'явилася (з «тунелю») тільки на цій діаграмі. У свою чергу, таке позначення навколо кінця (стрілки) інтерфейсної дуги поблизу блока-приймача означає той факт, що в дочірній відносно цього блока діаграмі ця дуга зображуватися і розглядатися не буде. Найчастіше окремі об'єкти і відповідні їм інтерфейсні дуги не розглядаються на деяких проміжних рівнях ієрархії, – вони спочатку «занурюються у тунель», а потім у разі потреби «повертаються з «тунелю».

Зазвичай IDEF0-моделі містять складну і концентровану інформацію, і для того, щоб обмежити перевантаженість і полегшити їх читання, у стандарті передбачено відповідні обмеження складності.

Рекомендується на діаграмі зображувати від трьох до шести функціональних блоків, при цьому передбачається кількість придатних для одного функціонального блока (що виходять з одного функціонального блока) інтерфейсних дуг не більше чотирьох.

Стандарт IDEF0 містить набір процедур, що дозволяє розробляти та погоджувати модель великою групою фахівців різних галузей. Процес розроблення є ітеративним і складається з таких умовних етапів.

Створення моделі групою фахівців з різних сфер діяльності підприємства. Ця група в термінах IDEF0 називається авторами (*Authors*). Побудова початкової моделі є динамічним процесом, протягом якого автори опитують компетентних осіб про структуру різних процесів, створюючи моделі діяльності підрозділів. Їх цікавлять відповіді на питання: Що надходить до підрозділу «на вході»; які функції і в якій послідовності виконуються в межах підрозділу; хто відповідає за виконання кожної функції; чим керується виконавець виконуючи кожноу з функцій; що є результатом роботи підрозділу (на виході).

На основі положень, документів і результатів опитувань створюється чернетка (*Model Draft*) моделі.

Поширення чернетки для розгляду, погоджень та коментарів. На цій стадії відбувається обговорення чернетки моделі широким колом компетентних осіб (у термінах IDEF0 – читачів) на підприємстві. При цьому кожна діаграма чорнової моделі письмово критикується і коментується, а потім передається автору. Автор, у свою чергу, також письмово погоджується з критикою або відкидає її з викладенням логіки прийняття рішення і знову повертає відкориговану чернетку для подальшого розгляду. Цей цикл продовжується доти, доки автори та читачі не дійдуть єдиної думки.

Офіційне затвердження моделі. Узгоджену модель затверджує керівник робочої групи в тому випадку, якщо автори моделі і читачі не мають розбіжності з приводу її адекватності. Остаточна модель підприємства (системи) має бути узгодженою.

Графічна мова IDEF0 робить модель зрозумілою і для осіб, які не брали участі в проекті її створення, а також ефективною для показів і презентацій. Надалі на базі побудованої моделі можна організовувати нові проекти, націлені на зміни до моделі.

4.2.2. Функціональна методика потоків даних

Метою методики є побудова моделі ІС у вигляді потоків даних DFD, що забезпечує правильне описання виходів (відгуку системи у вигляді даних) за заданого впливу на вхід системи (подання сигналів через зовнішні інтерфейси). Діаграми потоків даних є основним засобом моделювання функціональних вимог до проектованої системи.

Для створення діаграми потоків даних використовуються чотири основні поняття: потоки даних, процеси (роботи) перетворення вхідних потоків даних у вихідні, зовнішні сутності, нагромаджувачі даних (сховища).

Потоки даних є абстракціями, що використовуються для моделювання передавання інформації (або фізичних компонентів) з однієї частини системи в іншу. Потоки на діаграмах зображуються іменованими стрілками, орієнтація яких вказує напрям руху інформації.

Призначення процесу (роботи) полягає в продукуванні вихідних потоків із вхідних відповідно до дії. Назва процесу має містити дієслово невизначеної форми з подальшим доповненням (наприклад, «отримати документи з відвантаження продукції»). Кожен

процес має унікальний номер для посилань на нього всередині діаграми, який може використовуватися разом з номером діаграми для отримання унікального індексу процесу у всій моделі.

Сховище (нагромаджувач) даних дозволяє на зазначених ділянках визначати дані, які будуть зберігатися в пам'яті між процесами. Фактично сховище являє собою «зрізи» потоків даних у часі. Інформація, зазначена у ньому, може використовуватися в будь-який час після її отримання, при цьому дані можуть вибиратися в будь-якій послідовності. Назва сховища має визначати його вміст і бути іменником.

Зовнішня сутність являє собою матеріальний об'єкт поза контекстом системи, що є джерелом або приймачем системних даних. Її ім'я має містити іменник, наприклад, «склад товарів». Передбачається, що об'єкти, подані як зовнішні сутності, не повинні брати участі в жодному процесі оброблення.

Крім основних елементів, до складу DFD входять словники даних і мініспеціфікації.

Словники даних є каталогами всіх елементів даних, наявних у DFD, включаючи групові та індивідуальні потоки даних, сховища і процеси, а також всі їх атрибути.

Мініспеціфікації оброблення являють DFD опис процесів нижнього рівня. Фактично мініспеціфікації – це алгоритми опису завдань, що виконуються процесами: численні мініспеціфікації є повною специфікацією системи.

Процес побудови DFD починається зі створення основної діаграми типу «зірка», на якій зображено модельований процес і всі зовнішні сутності, з якими він взаємодіє. У разі складного основного процесу він відразу набуває вигляду декомпозиції на ряд взаємодійних процесів. Критеріями складності в цьому випадку є наявність великої кількості зовнішніх сутностей, багатофункціональність системи, її розподілений характер. Зовнішні сутності виділяються стосовно основного процесу. Для їх визначення необхідно виділити постачальників і споживачів основного процесу, тобто всі об'єкти, які взаємодіють з основним процесом. На цьому етапі опис взаємодії полягає у виборі дієслова, що дає уявлення про те, як зовнішня сутність використовує основний процес або використовується ним. Наприклад, основний процес – «облік звернень громадян», зовнішня сутність – «громадяни», опис взаємодії – «подає заяви і

отримує відповіді». Цей етап є принципово важливим, оскільки саме він визначає межі моделювання системи.

Для всіх зовнішніх сутностей будується таблиця подій, що описує їх взаємодію з основним потоком. Таблиця подій включає в себе найменування зовнішньої сутності, подію, його тип (типовий для системи або винятковий, реалізується за певних умов) і реакцію системи.

На наступному кроці відбувається декомпозиція основного процесу на набір взаємопов'язаних процесів, що обмінюються потоками даних. Самі потоки не конкретизуються, визначається лише характер взаємодії. Декомпозиція завершується, коли процес стає простим, тобто:

– має дві-три вхідні та вихідні потоки;

– може бути описаний у вигляді перетворення вхідних даних у вихідні;

– може бути описаний у вигляді послідовного алгоритму.

Для простих процесів будується мініспеціфікація – формальний опис алгоритму перетворення вхідних даних у вихідні.

Мініспеціфікація задовольняє такі вимоги: для кожного процесу будується одна специфікація; специфікація однозначно визначає вхідні та вихідні потоки для процесу, вона не визначає способу перетворення вхідних потоків у вихідні; посилається на наявні елементи, не вводячи нових; по можливості використовує стандартні підходи та операції.

Після декомпозиції основного процесу для кожного підпроцесу будується аналогічна таблиця внутрішніх подій.

Наступним кроком після визначення повної таблиці подій виділяються потоки даних, якими обмінюються процеси і зовнішні сутності. Найпростіший спосіб їх виділення полягає в аналізі таблиць подій. Події перетворюються в потоки даних від ініціатора події до запитуваного процесу, а реакції – у зворотний потік подій. Після побудови вхідних та вихідних потоків так само будуються внутрішні потоки. Для їх виділення для кожного з внутрішніх процесів виділяються постачальники та споживачі інформації. Якщо постачальник або споживач інформації являє собою процес збереження або запиту інформації, то вводиться сховище даних, для якого цей процес є інтерфейсом.

Після побудови потоків даних діаграма має бути перевірена на повноту і несуперечність. Повнота діаграми забезпечується, якщо в системі немає «завислих» процесів, що не використовуються в процесі перетворення вхідних потоків у вихідні. Несуперечність системи забезпечується виконанням наборів формальних правил про можливі типи процесів: на діаграмі не може бути потоку, що зв'язує дві зовнішні сутності – це взаємодія видаляється з розгляду; жодна сутність не може безпосередньо отримувати або віддавати інформацію в сховище даних – сховище даних є пасивним елементом, керованим за допомогою інтерфейсного процесу; два сховища даних не можуть безпосередньо обмінюватися інформацією – ці сховища мають бути об'єднані.

Переваги методики DFD:

– можливість однозначно визначати зовнішні сутності, аналізуючи потоки інформації всередині і поза системою;

– можливість проектувати зверху вниз, що полегшує побудову моделі «як має бути»;

– наявність специфікацій процесів нижнього рівня, що дозволяє подолати логічну незавершеність функціональної моделі та скласти повну функціональну специфікацію розроблюваної системи.

Недоліки моделі: необхідність штучного введення керувальних процесів, оскільки керувальні дії (потоки) та керувальні процеси з точки зору *DFD* нічим не відрізняються від звичайних; відсутність поняття часу, тобто аналізу часових проміжків при перетворенні даних (всі обмеження за часом мають бути введені в специфікаціях процесів).

4.2.3. Об'єктно-орієнтована методика

Принципова відмінність функціонального підходу від об'єктного полягає в способі декомпозиції системи. Об'єктно-орієнтований підхід використовує об'єктну декомпозицію, при цьому статична структура описується в термінах об'єктів і зв'язків між ними, а поведінка системи – у термінах обміну повідомленнями між об'єктами. Метою методики є побудова бізнес-моделі організації, що дозволяє перейти від моделі сценаріїв використання до моделі, що визначає окремі об'єкти, які беруть участь у реалізації бізнес-функцій.

Концептуальною основою об'єктно-орієнтованого підходу є об'єктна модель, яка будується з урахуванням таких принципів:

– абстрагування;
– інкапсуляція;
– модульність;
– ієрархія;
– типізація;
– паралелізм;
– стійкість.

Основними поняттями об'єктно-орієнтованого підходу є *об'єкт* і *клас*.

Об'єкт – предмет або явище, що має чітко визначене поводження, стан, поведінку і індивідуальність. Структура та поведінка схожих об'єктів визначають загальний для них клас. Клас – це безліч об'єктів, пов'язаних спільністю структури і поведінки. Ще однією групою важливих понять об'єктного підходу є успадкування і поліморфізм. Поняття поліморфізму може бути інтерпретовано як здатність класу належати більш ніж до одного типу. Спадкування означає побудову нових класів на основі існуючих з можливістю додавання або перевизначення даних і методів.

Важливою якістю об'єктного підходу є узгодженість моделей діяльності організації і моделей проектованої інформаційної системи від стадії формування вимог до стадії реалізації. За об'єктними моделями можна простежити відображення реальних сутностей модельованої предметної галузі (організації) в об'єкти і класи інформаційної системи.

Більшість методів об'єктно-орієнтованого підходу включають мову моделювання та опис процесу моделювання. *Процес* – це опис кроків, які необхідно виконати під час розроблення проекту. Як мову моделювання об'єктного підходу використовують уніфіковану мову моделювання UML, яка містить стандартний набір діаграм для моделювання.

Діаграма (*Diagram*) – це графічне зображення безлічі елементів. Найчастіше вона зображується у вигляді зв'язного графу з вершинами (сутностями) і ребрами (відносинами) і являє собою певну проекцію системи.

Об'єктно-орієнтований підхід має такі переваги:

– дає змогу створювати моделі меншого розміру шляхом використання загальних механізмів, що забезпечують необхідну економію виразних засобів. Використання об'єктного підходу істотно підвищує рівень уніфікації розроблення та придатність для повторного використання, що зумовлює створення середовища розроблення та переходу до складного створення моделей;

– дозволяє уникнути створення складних моделей, оскільки вона припускає еволюційний шлях розвитку моделі на базі відносно невеликих підсистем;

– є природною, оскільки орієнтована на людське сприйняття світу.

Недоліком об'єктно-орієнтованого підходу є високі початкові витрати. Цей підхід не забезпечує негайної віддачі. Ефект від його застосування позначається після розроблення двох-трьох проектів та нагромадження повторно використовуваних компонентів. Діаграми, що відображають специфіку об'єктного підходу, менш наочні.

4.2.4. Порівняння існуючих методик

У функціональних моделях (DFD-діаграм потоків даних, SADT-діаграм) головними структурними компонентами є функції (операції, дії, роботи), які на діаграмах з'єднуються між собою потоками об'єктів.

Безперечною перевагою функціональних моделей є реалізація структурного підходу до проектування ІС за принципом «зверху-вниз», коли кожен функціональний блок може бути декомпозований на безліч підфункцій, виконуючи, таким чином, модульне проектування ІС. Для функціональних моделей характерні процедурна строгість декомпозиції ІС і наочність подання.

За функціонального підходу об'єктні моделі даних у вигляді ER-діаграм «об'єкт – властивість – зв'язок» розробляються окремо. Для перевірки коректності моделювання предметної галузі між функціональними і об'єктними моделями встановлюються взаємно однозначні зв'язку.

Недоліки функціональних моделей: процеси й дані існують окремо один від одного, крім функціональної декомпозиції, наявна структура дани; не визначені умови виконання процесів оброблення інформації, які динамічно можуть змінюватися.

Недоліки функціональних моделей нівелюються в об'єктно-орієнтованих моделях, головним компонентом яких є структуроутворювальний клас об'єктів з набором функцій, які можуть звертатися до атрибутів цього класу.

Для класів об'єктів характерна ієрархія узагальнення, що дозволяє успадковувати не тільки атрибути (властивості) об'єктів від вищого класу об'єктів до нижчого класу, але і функцій (методів).

У разі спадкування функцій можна абстрагуватися від конкретної реалізації процедур (абстрактні типи даних), які різняться для певних підкласів ситуацій. Це дає змогу звертатися до подібних програмних модулів за загальними іменами (поліморфізм) і повторно використовувати програмний код для модифікації ПЗ. Таким чином, адаптивність об'єктно-орієнтованих систем до зміни предметної галузі порівняно з функціональним підходом значно вища.

За об'єктно-орієнтованого підходу змінюється і принцип проектування ІС. Спочатку виділяються класи об'єктів, а далі залежно від можливих станів об'єктів (життєвого циклу об'єктів) визначаються методи оброблення (функціональні процедури), що забезпечує найкращу реалізацію динамічного поводження ІС.

Для об'єктно-орієнтованого підходу розроблені графічні методи моделювання предметної галузі, узагальнені в мові уніфікованого моделювання UML. Однак за наочністю подання моделі користувачу-замовнику об'єктно-орієнтовані моделі явно поступаються функціональним моделями.

Для вибору методики моделювання предметної галузі зазвичай критерієм виступає ступінь її динамічності. Для більш регламентованих завдань більше підходять функціональні моделі, для більш адаптивних бізнес-процесів (керування робочими потоками, реалізації динамічних запитів до інформаційних сховищ) – об'єктно-орієнтовані моделі. Проте в межах однієї і тієї ж ІС для різних класів завдань можуть вимагатися різні види моделей, що описують одну й ту саму проблемну галузь. У такому випадку потрібно використовувати комбіновані моделі предметної галузі.

4.2.5. Синтетична методика

Кожна з розглянутих методик дозволяє створювати формальний опис робочих процедур досліджуваної системи. Всі методики дозволяють будувати модель «як є» і «як має бути». З іншого боку, кожна з цих методик має істотні недоліки. Недоліки застосування окремої методики полягають не в описі реальних процесів, а в неповноті методичного підходу.

Функціональні методики в цілому дають уявлення про функції в організації, про методи їх реалізації, причому чим вищий ступінь деталізації досліджуваного процесу, тим краще вони дозволяють описати систему. Під кращим описом у цьому випадку розуміють найменшу помилку при спробі за отриманою моделлю передбачити поведінку реальної системи. На рівні окремих робочих процедур їх опис однозначно збігається з фактичною реалізацією в потоці робіт.

На рівні загального опису системи функціональні методики допускають значний ступінь довільності щодо вибору загальних інтерфейсів системи, її механізмів, тобто визначення меж системи. Описати систему на цьому рівні належним чином дозволяє об'єктний підхід, заснований на понятті сценарію використання. Ключовим є поняття про сценарій використання як про сеанс взаємодії дійової особи із системою, в результаті якого дійова особа отримує щось, що має для нього цінність. Використання критерію цінності для користувача дає можливість відкинути незначні деталі потоків робіт і зосередитися на тих функціях системи, які виправдовують її існування. Однак і в цьому випадку завдання визначення меж системи, виділення зовнішніх користувачів є складним.

Технологія потоків даних, що історично виникла першою, легко вирішує проблему меж системи, оскільки дозволяє за допомогою аналізу інформаційних потоків виділити зовнішні сутності і визначити основний внутрішній процес. Однак без виділених керувальних процесів, потоків і дієвої орієнтованості не можливо запропонувати цю методику, як єдину.

Найкращим способом усунення недоліків розглянутих методик є формування синтетичної методики, що поєднує різні етапи окремих методик. При цьому з кожної методики необхідно взяти частину методології, найбільш повну і формально викладену, і уможливити обмін результатами на різних етапах застосування си-

нергетичної методики. У бізнес-моделюванні неявним чином формується така синергетична методика.

Ідея синтетичної методики полягає у послідовному застосуванні функціонального та об'єктного підходів з урахуванням можливості реінжинірингу існуючої ситуації.

Розглянемо застосування синтетичної методики на прикладі розроблення адміністративного регламенту.

При побудові адміністративних регламентів виділяються наступні стадії:

1. Визначення меж системи – за допомогою аналізу потоків даних виділяють зовнішні сутності і власне модельовану систему.

2. Виділення сценаріїв використання системи – за допомогою критерію корисності формують для кожної зовнішньої сутності набір сценаріїв використання системи.

3. Додавання системних сценаріїв використання – визначають сценарії, необхідні для реалізації цілей системи, відмінних від цілей користувачів.

4. Побудова діаграми активностей сценаріїв використання – формують набір дій системи, що призводять до реалізації сценаріїв використання.

5. Функціональна декомпозиція діаграм активностей як контекстних діаграм методики IDEF0.

6. Формальний опис окремих функціональних активностей у вигляді адміністративного регламенту (із застосуванням різних нотацій).

Засоби для моделювання ділових процесів: інструментальне середовище BPwin. Принципи побудови моделі IDEF0: контекстна діаграма, суб'єкт моделювання, мета і точка зору. Діаграми IDEF0: контекстна діаграма, діаграми декомпозиції, діаграми дерева вузлів, діаграми лише для експозиції (FEO).

Ділові процеси зазвичай моделюють за допомогою case-засобів. До таких засобів належать BPwin (PLATINUM technology), Silverrun (Silverrun technology), Oracle Designer (Oracle), Rational Rose (Rational Software) та ін.

BPwin підтримує три методології моделювання: функціональне моделювання (IDEF0); опис бізнес-процесів (IDEF3); діаграми потоків даних (DFD).

Запитання для самоперевірки

1. Що таке сумісність програмних середовищ?
2. Що таке GUI? У чому полягають сутність та призначення?
3. Які поняття в основі методології IDEF?
4. Поясніть поняття «глосарій» та «мета».
5. Навіщо створюється чорновий варіант моделі?
6. Як створюють моделі?
7. Як виконати декомпозицію процесів у Bpwin?

СПИСОК ЛІТЕРАТУРИ

1. Брауде Э. Технологии разработки программного обеспечения. – СПб: Питер, 2004. – 655 с.
2. Вельбицкий И.В. Технология программирования. – К.: Техніка, 1984. – 279 с.
3. Вигерс Карл. Разработка требований к программному обеспечению; [Пер. с англ.] – М: Русская Редакция, 2004. – 576 с.
4. Гайдамакин Н. А. Автоматизированные информационные системы, базы и банки данных. Вводный курс: Учеб. пособие. – М.: Гелиос АРВ, 2002. – 368 с.
5. Гайфуллин Б. Н. Автоматизированные системы управления предприятиями стандарта ERP/MRPII [Текст] Производственное издание / Б. Н. Гайфуллин, И. А. Обухов. – М.: Богородский печатник, 2001. – 104 с.
6. Зосимович М.В. Технологія програмування: Конспект лекцій. Ч.1 – Житомир: ЄУФІМБ, 2005. – 44 с.
7. Зосимович Н.В. Технология программирования: Конспект лекций. Ч.2 – Житомир: ЕУФИМБ, 2005. – 43 с.
8. Коберн А. Современные методы описания функциональных требований к системам. – М: Лори, 2002. – 263 с.
9. Костромин В. Linux для пользователя. – СПб.: БХВ-Петербург, 2002. – 352с.
10. Кратчен Ф. Введение в Rational Unified Process. – М.: Вильямс, 2002. – 240 с.
11. Леонтьев В.П. Новітня енциклопедія персонального комп'ютера. – М: ОЛМА-ПРЕСС, 2003. – 920 с.
12. Макарова Н.В. Информатика: Учебник. – М.: Финансы и статистика, 2003. – 768 с.
13. Маклаков С.В. BPwin и ERwin: CASE-средства для разработки информационных систем. – М: ДиалогМифи, 2000. – 256 с.
14. Марка Д.А. Методология структурного анализа и проектирования. – СПб.: Питер, 1995. – 235 с.
15. Мацяшек Л. А. Анализ требований и проектирование систем. Разработка информационных систем с использованием UML. – М.: Вильямс, 2002. – 432 с.

16. Меняев М.Ф. Информационные технологии управления.– М.: Омега-Л, 2003.
Книга 3: Системы управления организацией.– 2003.– 464 с.
17. Мэнсфилд Рон. Windows 95 для занятых. – СПб.: Питер , 1997. – 384 с.
18. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб.: Питер, 2002. – 504 с.
19. Орлов С.А. Технологии разработки программного обеспечения: Учебник для вузов. 3-е изд. – СПб.: Питер, 2004. – 527 с.
20. О'Лири Д. ERP системы. Современное планирование и управление ресурсами предприятия. Выбор, внедрение, эксплуатация. – М.: Вершина, 2004. – 272 с.
21. Ричард Петерсен. Linux: руководство по операционной системе: [В 2 т.]; [пер с англ. С. М. Тимачева]; Под ред. М. В. Коломыцева. – К: СПб. ЭлектроникаБизнесИнформатика, 1998. – 527 с.
22. Петров В. Н. Информационные системы. – СПб: Питер, 2002. – 688 с.
23. Потапкин А.В. Операционная система Windows 95: Руководство к действию: Практическое пособие. – М.: ЭКОМ, 1996. – 432 с.
24. Раскин А.Л. Руководство по применению стандарта ИСО 9001:2000 при разработке программного обеспечения. – М.: Стандарты и качество, 2002. – 104 с.
25. Таненбаум Е. Сучасні операційні системи. – СПб.: Пітер, 2002. – 1040 с.
26. Фойц Стефан. Windows 3.1. – К.: BHV, 1996. – 557 с.
27. Черемных С.В. Моделирование и анализ систем. IDEF-технологии: практикум/ Черемных С.В., Семенов И.О., Ручкин В.С. – М.: Финансы и статистика, 2006. – 192 с.
28. Шафрин Ю. А. Информационные технологии: [В 2ч.] – М.: Лаборатория Базовых Знаний, – 2000.
Ч.2: Офисная технология и информационные системы – 2000. – 336с.
29. Якобсон А., Буч Г., Рамбо Дж. Унифицированный процесс разработки программного обеспечения. – СПб.: Питер, 2002. – 496 с.
30. Analyzing requirements and defining Microsoft .Net solution architectures. – Microsoft Press 2000. – 491 p.

Операційна система MS DOS

MS-DOS (Microsoft Disk Operating System – дискова ОС від Microsoft) – комерційна ОС фірми Microsoft для IBM PC-сумісних ПК. MS-DOS – найвідоміша ОС із сім'ї DOS, що раніше встановлювалась на більшість IBM PC сумісних комп'ютерів. З часом вона була витіснена ОС сім'ї Windows 9x та Windows NT.

Мінімальний набір файлів MS-DOS:

Файли ядра:

- *IO.SYS* – розширення BIOS;
- *MSDOS.SYS* – оброблення переривань.

Командний процесор:

- *COMMAND.COM* – командний процесор (підтримка інтерфейсу командного рядка).

Файли конфігурації:

- *CONFIG.SYS* – конфігурування системи та завантаження драйверів пристроїв на етапі ініціалізації *MSDOS.SYS.*
- *AUTOEXEC.BAT* – стартовий пакетний файл. Виконується у процесі запуску командного процесора під час завантаження ОС.

Команди MS DOS

Команди DOS для роботи з каталогами

Зміна поточного каталогу

Формат команди:
cd [дисковод:][шлях]

ПРОДОВЖЕННЯ ДОД. 1

Приклади:
**cd ** – перехід у кореневий каталог поточного диска;
cd .. – перехід у наддиректорію;
cd – повідомляє поточний диск і каталог.

Перегляд каталогу

Формат команди:
dir [дисковод:][шлях\][ім'я файлу] [параметри]

Параметри:
/p – поекранне виведення;
/w – виведення у широкому форматі;
/s – зміст зазначеного в команді каталогу й усіх його підкаталогів;
/b – тільки імена файлів без заголовних і підсумкових відомостей;
/a атрибут – відомості про файли, що мають зазначені атрибути.

Сортування:
/on – за іменем;
/oe – за розширенням;
/od – за часом;
/og – спочатку виводити відомості про підкаталоги.

Приклади:
dir – зміст поточного каталогу;
dir *.exe – відомості про всі файлах *.exe* поточного каталогу;
dir a: – зміст поточного каталогу диска *a*:;
dir c:*.exe /s – відомості про всі файли *.exe* на диску *c*:.

Виведення змісту у файл або на принтер:
dir > prn – вивести зміст поточного каталогу на принтер;
dir c:*.txt > txtfiles.txt – створити у файлі *txtfiles.txt* список усіх файлів з розширенням *txt*, що перебувають у кореневому каталозі диска *c:*.

ПРОДОВЖЕННЯ ДОД. 1

Створення каталогу

Формат команди:

md [дисковод:][шлях\] ім'я каталогу

Приклад:
md c:\users\my – створити каталог *my* у каталозі *users* у кореневому каталозі диска *c*:.

Видалення каталогу

Видалення порожнього каталогу

Формат команди:
rd [дисковод:][шлях\] ім'я каталогу

Приклад:
rd c:\users\my – вилучити каталог *my* з підкаталогу *users* кореневого каталогу диска *c*:.

Видалення каталогу з усім умістом

Формат команди:
deltree [/y] ім'я файлу або каталогу

Команда **deltree** може видаляти як каталоги, так і файли. В **імені файлу або каталогу** можна використовувати символи * і ?.

Приклади:
deltree temp – вилучити каталог або файл з іменем *temp* з поточного каталогу;
deltree /y d* – вилучити з поточного каталогу всі каталоги й файли, ім'я яких починається на *d*, не запитуючи підтвердження.

ПРОДОВЖЕННЯ ДОД. 1

Перейменування каталогу

Формат команди:
move [дисковод:][шлях\] ім'я каталогу нове ім'я каталогу

Приклад:
move a:\temp tmp – перейменувати каталог *temp* кореневого каталогу диска *a*: в *tmp*.

Установлення списку каталогів для пошуку виконуваних програм

Формат команди:
path [дисковод:][шлях\] ім'я каталогу [; [дисковод:][шлях\] ім'я каталогу]

Приклади:
path ; – пошук програм повинен вестися тільки в поточному каталозі;
path c:\exe; c:\exe\program; d:\msdos – пошук програм робити в каталогах *exe*, *program*, *msdos*.

Команди DOS для роботи з файлами

Створення текстових файлів

Формат команди:
copy con ім'я файлу

Ctrl+Z, F6 – ознака кінця файлу.
Enter – ознака кінця рядка.

Приклади:
copy con work.txt – створити в поточному каталозі текстовий файл *work.txt*.

ПРОДОВЖЕННЯ ДОД. 1

Видалення файлів

Формат команди:
del ім'я файлу

Приклади:
del *.txt – вилучити всі файли з розширенням *txt* з поточного каталогу;
del name.doc – вилучити з поточного каталогу файл із іменем *name.doc*.

Перейменування файлів

Формат команди:
ren ім'я файлу1 ім'я файлу2
У параметрі **ім'я файлу1** можна вказувати дисковод і шлях, в **ім'я файлу2** – ні.
Команда *ren* не обробляє сховані файли.

Приклади:
ren xxx.doc xxx.txt – перейменувати файл *xxx.doc* поточного каталогу в *xxx.txt*;
ren a:*.txt *.doc – перейменувати всі файли поточного каталогу на диску *a*: з розширенням *txt* у файли з такими ж іменами й розширеннями *.doc*.

Копіювання файлів

Формат команди:
copy ім'я файлу1 ім'я файлу2
copy ім'я файлу1 [ім'я каталогу2]

В іменах файлів можна вживати символи * і ?, а також указувати ім'я диска й шлях.
Команда *copy* не копіює сховані файли й файли нульової довжини.

ПРОДОВЖЕННЯ ДОД. 1

Якщо файл із таким же іменем, як у копії, створюваній командою, вже існує, то він заміщається.

Приклади:

copy x.txt z.txt – скопіювати файл *x.txt* у поточний каталог з іменем *z.txt*;

copy \text*.txt a:*.doc – скопіювати з підкаталогу *text* поточного каталогу всі файли з розширенням *txt* у поточний каталог диска *a*:. Файли отримають розширення *doc*.

Використання обладнання:

copy t1.txt prn – копіювання файлу *t1.txt* на принтер;

copy t1.txt con – копіювання файлу *t1.txt* на екран монітора.

З'єднання (конкатенація) файлів

Формат команди:

copy ім'я файлу [+ ім'я файлу]…[ім'я файлу]

Якщо ім'я вихідного файлу (або одного з файлів) збігається з іменем створюваного командою *copy* файлу, то існуючий файл заміщається. Наприклад, якщо файл *all.doc* уже існує, то команда copy **.doc all.doc* буде помилковою, тому що файл *all.doc* буде знищений на початку копіювання.

Приклади:

copy .lst+*.ref *.prn – до кожного файлу поточного каталогу з розширенням *lst* додати файл із тим же іменем і розширенням *ref*, результат записується у файл із тим же іменем і розширенням *prn*;

copy f1.doc+f2.doc – об'єднати файли *f1.doc* і *f2.doc*, уміст об'єднаного файлу записується у файл *f1.doc*;

copy *.txt all.prn – уміст усіх файлів з розширенням *txt* записується у файл *all.prn*.

ПРОДОВЖЕННЯ ДОД. 1

Переміщення файлів в інший каталог

Формат команди:
move [/y] ім'я файлу ім'я каталогу
move [/y] ім'я файлу [дисковод:][шлях] нове ім'я файлу

З параметром */y* при існуванні в каталозі- приймачі файлів з тими ж іменами, що й пересилаються, виконується заміщення цих файлів без запиту. Завдання нового імені можливе тільки при пересиланні одного файлу. Наприклад, команда *move *.bac a:*.old* помилкова.

Приклади:
move *.doc d: – перемістити файли з розширенням *doc* з поточного каталогу в кореневий каталог диска *d:*;
move f1.txt tmp\f2.txt – перемістити файл *f1.txt* у каталог *tmp* з перейменуванням в *f2.txt*.

Порівняння файлів

Формат команди:
fc [параметри] ім'я файлу ім'я файлу [ім'я файлу-протоколу]

Якщо ім'я файлу-протоколу не задане, відомості про порівняння виводяться на екран.

Параметри:
/l – порядкове порівняння. Програма виявляє відмінності у файлах, намагається знайти після точки неузгодженості місця, починаючи з яких файли знову стають однаковими. На виході – рядки, що різняться;
/b – побайтове порівняння. Після виявлення відмінності файлів порівняння припиняється.

ПРОДОВЖЕННЯ ДОД. 1

За замовчуванням режим порівняння вибирається за розширенням:

/b – *.exe*, *.com*, *.sys*, *.obj*, *.lib*, *.bin*,

/l – інші.

Параметри порядкового порівняння:

/c – ігнорувати відмінності між рядковими й прописними буквами;

/n – виведення номерів рядків;

/l число – скільки рядків файлів має збігатися, щоб файли вважалися знову погодженими; за замовчуванням – 2 рядки;

/b число – розмір внутрішнього буфера для знаходження відповідностей у файлах після неузгодженості; за замовчуванням – 100 рядків.

Приклад:

fc doclad.doc doclad1.doc > diff – порівняти файли, звіт помістити у файл *diff*.

Виведення файлу на екран

Формат команди:

type ім'я файлу

Приклад:

type t1.doc – виведення на екран файлу *t1.doc* з поточного каталогу.

Команда копіювання xcopy

Формат команди:

xcopy файл-або-каталог … [параметри]

Переваги команди xcopy:

– працює швидше, ніж copy;

– дозволяє копіювати файли з усіх підкаталогів зазначеного каталогу;

ПРОДОВЖЕННЯ ДОД. 1

– дозволяє вибіркове копіювання файлів залежно від значення атрибута «архівний» або дати файлу.

Обмеження команди хсору:

– не підтримує копіювання з логічного обладнання або на логічне обладнання;

– не дозволяє поєднувати файли;

– завжди копіює цілі файли, команда сору може припиняти копіювання, якщо у вихідному файлі трапляється символ кінця файлу.

Параметри:

/s – копіювання файлів із зазначених каталогів і всіх їхніх підкаталогів. Файли з підкаталогів копіюються у відповідні підкаталоги того каталогу, у який копіюються файли;

/e – копіювання всіх підкаталогів, навіть якщо вони порожні. Використовується тільки разом з режимом /s;

/a – копіювання тільки тих файлів, у яких установлений атрибут «архівний»;

/m – копіювання тільки тих файлів, у яких установлений атрибут "архівний". Після копіювання, атрибут «архівний» відміняється;

/d дата – копіювання файлів, створених або змінених, починаючи із зазначеної дати. Якщо дата не зазначена, то копіюються тільки файли, більш нові, ніж файли, які вони заміщають;

/p – запит на копіювання кожного файлу;

/y – перезапис наявних файлів з тими ж іменами без запитів;

/n – перевірка правильності копіювання кожного файлу.

Приклади:

xcopy a:\ b:\ /s /e – копіювання всіх каталогів з диска *a*: на *b*:;

xcopy *.doc a:\ /s – копіювання всіх файлів з розширенням *doc* з поточного каталогу в кореневий каталог диска *a*:. Файли з розширенням *.doc* з підкаталогів поточного каталогу копіюються в однойменні підкаталоги кореневого каталогу диска *a*:;

xcopy *.* a:\ /s /e – копіювання всіх файлів і підкаталогів поточного каталогу в кореневий каталог і відповідні підкаталоги диска *a*: (створення архівної копії файлів поточного каталогу);

ПРОДОВЖЕННЯ ДОД. 1

xcopy *.* a:\ /m /s /e – копіювання всіх змінених файлів і підкаталогів поточного каталогу в кореневий каталог і відповідні підкаталоги диска *a*: (відновлення архівної копії файлів поточного каталогу).

Команди DOS для роботи з дисками

Зміна поточного дисководу

Формат команди:
ім'я дисководу:

Приклади:
c: – установити поточним диск *c*:;
a: – установити поточним диск *a*:.

Режим перевірки під час запису на диски

Формат команди:
verify [on/off]

Приклади:
verify on – увімкнути режим перевірки під час запису на диски;
verify off – відключити режим перевірки під час запису на диски;
verify – вивести інформацію про те, ввімкнений або вимкнений режим перевірки.

Форматування дискет

Формат команди:
format дисковод: [параметри]

Параметри:
/s – створити системний диск;
/v:мітка – задання мітки диска;
/u – безумовне форматування зі знищенням наявних даних;

ПРОДОВЖЕННЯ ДОД. 1

/q – швидке очищення без контролю наявності збійних ділянок.

Якщо параметри не зазначені, то програма перевіряє, чи відформатована дискета; залишає формат таким самим, що й наявний; стирає інформацію про всі файли й каталоги із системних ділянок; тестує ділянку даних на наявність збійних ділянок.

Приклад:
format a: /u – безумовне форматування дискети *a*:;

Перенесення на диск системних файлів DOS

Формат команди:
sys [шлях] диск:

Якщо путь не заданий, системні файли беруться з кореневого каталогу поточного диска.

Приклад:
sys a: – перенести на диск *a*: системні файли з кореневого каталогу поточного диска.

Завдання мітки диска

Формат команди:
label дисковод:

Мітка – позначення довжиною до 11 символів.

Недопустимі символи: * ? / \ | ., ; : + = [ ] ( ) & < > ^ " .

Щоб узнати мітку диска, можна використовувати команду *vol*. (*vol* дисковод:)

Приклади:
label a: – повідомити мітку диска *a*:.

ПРОДОВЖЕННЯ ДОД. 1

Програми й команди DOS загальносистемного призначення

Виведення інформації про дату і встановлення дати в комп'ютері

Формат команди:
date

Приклад: **date**
Запит: Уведіть нову дату (дд-мм-гггг.): (*Enter new date (dd-mm-yyyy*))
рядок, що вводиться: 11-09-2000

Виведення інформації про час і встановлення часу в комп'ютері

Формат команди:
time

Приклад:
time – вивести поточний час; time 11:29 – установити час 11 годин 29 хв.

Зміна виду запрошення DOS

Формат команди:
prompt [текст]

Спеціальні комбінації символів:
$p – поточний дисковод і каталог;
$n – поточний дисковод;
$d – поточна дата;
$t – поточний час;
$v – версія DOS;
$_ – перехід на новий рядок;
$s – пробіл;

ПРОДОВЖЕННЯ ДОД. 1

$g – символ ">";
$h – видалення попереднього символа.

Приклади:
prompt pg – установлює запрошення виду c:\users\doc>;

Отримання інформації про версію DOS

Формат команди:
ver

На екран виводиться версія використовуваної ОС.

Операційна система Linux

Linux – багатозадачна й багатокористувацька ОС для освіти, бізнесу, індивідуального програмування. Linux належить до сім'ї Unix-подібних ОС.

Операційна система Linux була написана Лінусом Торвальдсом, а потім багаторазово поліпшувалася. Вона є клоном комерційної ОС Unix, однієї з перших потужних ОС.

Режим командного рядка є основним режимом ОС Linux. Тому без знання основних команд та розуміння принципів їх дії повноцінна робота з ОС Linux неможлива.

Команди Linux

Отримання справки про команди

Опис команди або файлу конфігурації

Формат команди:
man

Перегляд опису стрілками, вихід клавішею *q*.

Приклад:
man fstab

Пошук за описом man

Формат команди:
apropos

Приклад:
apropos iso

Команди роботи з каталогами

Зміна поточного каталогу

Формат команди:
cd <ім'я каталогу>

Приклади:
cd .. – перехід у надкаталог;
cd . – немає переходу, тому що «.» є посиланням на поточний каталог;
cd – перехід у домашній каталог поточного користувача;
cd /bin – перехід у каталог *bin* абсолютним шляхом;
cd bin – перехід у каталог *bin* поточного каталогу;
cd /lab – перехід у каталог *lab***.**

Вивід списку файлів каталогу

Формат команди:
ls <ім'я файлу1> ... < ім'я файлуN>

Параметри:
-F – виведення інформації про типи файлів;
-l – виведення інформації про файли, власників, права доступу, дату створення і т. ін.).

Приклади:
ls -lF /home/user – зміст каталогу */home/user*;
ls – зміст поточного каталогу;
ls /proc – виведення всіх процесів, що функціонують.

Вивід змісту поточного каталогу

Формат команди:
pwd

Створення нового каталогу

Формат команди:
mkdir <ім'я каталогу1> ... <ім'я каталогуN>

Приклад:
mkdir /home/user/tests – створення каталогу *tests* у каталозі */home/user*.

Видалення порожнього каталогу

Поточний робочий каталог має бути не в каталозі, що видаляють.

Формат команди:
rmdir <ім'я каталогу1> ... <ім'я каталогу N>

Приклад:
rmdir /home/labs/M1 видаляє каталог */home/labs/M1*, якщо він порожній.

Команди роботи з файлами

Поекранний вивід змісту названих файлів

Формат команди:
more <ім'я файлу1> ... <ім'я файлуN>

Приклад:
more labs/lab1 – вивід файлу *lab1* каталогу *labs***.**

Перенесення файлів

Формат команди:
mv <ім'я файлу1> ... <ім'я файлуN> <ім'я файлу/каталогу>

де **<ім'я файлу1> ... <ім'я файлуN>** – імена файлів, що переносять, а **<ім'я каталогу>** – ім'я файлу/каталогу, куди переносять файли.

Приклад:
mv ../lab labs – перенесення файлу *../lab* до файлу/каталогу *labs*.

Копіювання файлів

Формат команди:
cp <ім'я файлу1> ... <ім'я файлуN> <ім'я файлу/каталогу>

де **<ім'я файлу1> ... <ім'я файлуN>** – імена файлів, що копіюють, а **<ім'я файлу/каталогу>** – ім'я файлу/каталогу, куди копіюють файли.

Приклад:
cp ../lab labs – копіювання файлу *../lab* до файлу/каталогу *labs*.

Видалення файлів

Формат команди:
rm <ім'я файлу1> ... <ім'я файлуN>

Параметри:
-i – запит підтвердження перед видаленням файлу;
-R – видалення у підкаталогах.

Приклад:
rm -i /home/user/lab1 /home/user/lab2 – видалення файлів *lab1* і *lab2* з каталогу */home/user*.

Вивід на екран змісту файла з можливістю навігації

Формат команди:
less <ім'я файлу>

Навігація здійснюється за допомогою клавіш зі стрілками, *PgUp/PgDn* і т. ін.

Вивід на екран ідентифікатора типа файлу з дослідженням його змісту з дуже високим ступенем точності

Формат команди:
file <ім'я файлу>

Пошук файлу/каталогу на вказаному диску з указанням всіх знайдених місць перебування

Формат команди:
locate <ім'я файлу/ім'я каталогу>

Можна вказувати часткове им'я або частину повного шляху.

Права доступа до файлів та каталогів

Зміна власника для одного або кількох файлів

Формат команди:
chown [параметри] <користувач[:група]> <ім'я файлу/ім'я каталогу> [ім'я файлу/ім'я каталогу...]

Параметри:

-R – зміна власника для всіх файлів і підкаталогів у зазначеному каталозі;

-v – вивід на екран всіх дій, виконуваних chown: для яких файлів були змінені власники в результаті виконання команди, і які файли залишилися без змін;

-c – вивід інформації про змінені файли.

Приклад:
chown teacher /labs/lab1.doc – змінює власника файлу */labs/lab1.doc* на *teacher*.

Зміна групи для одного чи декількох файлів

Формат команди:
chgrp [параметри] <група> <ім'я файлу/ім'я каталогу> [ім'я файлу/ім'я каталогу...]

Параметри і спосіб використання такі самі, як у chown.

Приклад:
chgrp disk /dev/l* – змінює належність всіх файлів в каталозі */dev/* з іменами, що починаються з *l*, до групи *disk*.

Зміна прав доступа

Формат команди:
chmod [a,u,g,o][+,-][r,w,x] <ім'я файлу>

Параметри:
a – доступ для всіх (*all*);
u – доступ для користувача (*user*);
g – доступ для групи користувачів (*group*);
o – доступ для інших (*other*);
+,- – надання/позбавлення прав доступу;
r – доступ для читання (*read*);
w – доступ для запису (*write*);
x – доступ для виконання (*execute*).

Права доступа задають у символьному або числовому вигляді.

Приклади:
chmod 0000 f1 – для *f1* задані права доступа 0000;
chmod a+r lab1 – надання всім користувачам права доступа для читання файлу *lab1*;
chmod +r lab1 – теж саме (*a* – за замовчуванням).
chmod og-x lab1 – позбавлення права доступа на виконання файлу *lab1* всіх, окрім власника.

chmod u+rwx lab1 – дозвіл власнику на всі дії з файлом *lab1* (read, write та execute).

chmod o-rwx lab1 – заборона на всі дії з файлом *lab1* (read, write та execute) користувачам категорії «інші» (other).

Конкатенація файла або вивід файла на екран

Формат команди:
cat <ім'я файлу1> ... <ім'я файлуN>

Приклад:
cat labs/lab1 – вивід на екран файла *labs/lab1***.**

Вивід аргументів

Формат команди:
echo <аргумент1> ... <аргументN>

де <**аргумент**1> ... <**аргумент**N> – аргументи, що повторюються.

Приклад:
echo "Hello world" – вивід на екран «Hello world».

Вивід усіх рядків у названому файлі(лах), які містять заданий зразок

Формат команди:
grep <зразок> <ім'я файлу1> ... <ім'я файлуN>

де **<зразок>** – зразок (регулярний вираз), а **<ім'я файлу1> ... <ім'я файлуN>** – файли, у яких проводиться пошук.

Приклад:
grep text /labs/lab1 – вивід всіх рядків, де файл */labs/lab1* містить зразок *text***.**

Створення посилання на файл

Формат команди:
ln [-f] <ім'я файлу1> [<ім'я файлу2> ...] ім'я цільового файлу

Команда *ln* робить *цільовий файл* посиланням на *файл1*. *Файл1* не повинен збігатися із цільовим файлом. Якщо цільовий файл є каталогом, то в ньому створюються посилання на *файл1*, *файл2*, ... з тими ж іменами. Тільки в цьому випадку можна вказувати кілька вихідних файлів.

Якщо цільовий файл існує й не є каталогом, його старий уміст губиться. Якщо при цьому виявляється, що в цільовий файл не дозволений запис, то виводиться режим доступу до цього файлу й запитується рядок зі стандартного введення. Якщо цей рядок починається із символу *y*, то необхідні дії все-таки виконуються, за умови що у користувача досить прав для видалення цільового файлу. Якщо була зазначена опція -f або стандартне введення призначене не на термінал, то необхідні дії виконуються без усяких запитів. Цільовий файл успадковує режим доступу до *файл1*.

Зведення використання дискового простору

Формат команди:
du [-s] [-a] [-r] [ім'я файлу...]

Команда *du* видає кількість кілобайтних блоків, що втримуються у всіх заданих файлах, серед яких можуть бути каталоги. Обхід каталогів виконується, починаючи із зазначених, рекурсивно на будь-яку глибину. При підрахунку враховуються непрямі блоки файлів. Якщо файли не задані, робота йде з поточним каталогом.

Параметри:

-s – видається тільки загальна сума для кожного заданого файлу (навіть якщо він є каталогом);

-a – для кожного файлу, що зустрівся при обході, видається рядок, що містить кількість блоків та ім'я файлу.

Якщо жодний із цих параметрів не задано, вихідні рядки генеруються тільки для каталогів, що зустрілися при обході.

-r – викликає видачу повідомлень про каталоги, які не можуть бути прочитані, про файли, які не можна відкрити і т. ін., на відміну від режиму без повідомлень (за замовчуванням).

Файл, на який існує два й більше посилання, ураховується тільки один раз.

Редагування файлів

Формат команди:
nano <ім'я файлу>

Вихід з редактора *CTRL+X*.

Приклад:
nano /texts/text1 – внесення змін до файлу */texts/text1*.

Монтування й розмонтування дисків

Монтування пристрою

Формат команди:
mount [параметри] пристрій шлях

Приклад:
mount /dev/sda1 /mnt/Disk1.

Розмонтування

Формат команди:
umount пристрій/шлях

Приклад:
umount /mnt/Disk1.

Розмонтування й викид CDROM

Формат команди:
eject

Команди роботи з користувачами

Отримання інформації про користувачів, що увійшли до системи

Формат команди:
who

Параметр:
-i – інформація про час простою кожного користувача або коли він останній раз входив у систему.

Приклади:

```
who
jene      tty3    Jan    29     08:29
root      tty7    Jan    29     08:00

who -i
jene      tty3    Jan    29     08:29   09:30
root      tty7    Jan    29     08:00   00:10
```

Отримання інформації про поточного користувача

Формат команди:
who am i
або
whoami

Зміна користувача

Формат команди:
su <им'я користувача>

Корисно, коли потрібно виконати команду або програму із правами адміністратора. У цьому випадку не потрібно виходити, починати новий сеанс і т. ін.

Команди завершення роботи системи

Безпечне вимкнення комп'ютера

Формат команди:
shutdown <параметри> <час> <застережуюче повідомлення>

де **<час>** – указує час, коли має бути виконано команду (не обов'язково виконувати її негайно).

Параметри:
-h – повне зупинення системи (комп'ютер буде вимкнено);
-r – перезавантаження системи.

Команда *shutdown* може бути виконана тільки користувачем *root*. Час можна вказати у формі затримки від теперішнього моменту.

Приклади:
shutdown -r +5 – зупинення системи через 5 хвилин та перезавантаження після коректного завершення роботи;
shutdown -h 0 – вимкнення комп'ютера.

Вимкнення комп'ютера

Формат команди:
halt

Перезавантаження комп'ютера

Формат команди:
reboot

Вимкнення живлення

Формат команди:
poweroff

Навчальне видання

ХАРЧЕНКО Володимир Петрович
ЗНАКОВСЬКА Євгенія Анатоліївна
БОРОДІН Віктор Анатолійович

ОПЕРАЦІЙНІ СИСТЕМИ ТА СИСТЕМИ ПРОГРАМУВАННЯ

Навчальний посібник

Редактор Р. М. Шульженко
Технічний редактор
Комп'ютерна верстка

Підп. до друку . .10. Формат . Папір офс. Офс. друк. Ум. друк. арк. . Обл.-вид. арк. .
Тираж 100 пр. Замовлення № . Вид. № .

Видавництво НАУ
03680. Київ-680, проспект Космонавта Комарова, 1.

Свідоцтво про внесення до Державного реєстру ДК № 977 від 05.07.2002