РОЗДІЛ 1

МЕТОДИ ПОБУДОВИ Й СПОСОБИ ФУНКЦІОНУВАННЯ ОПЕРАЦІЙНИХ СИСТЕМ

ВСТУП

Перший розділ присвячений опису методів побудови й основ функціонування операційних систем (ОС) в обчислювальних системах. Розглянута класифікація поколінь ОС, їх функціональні особливості й тенденції розвитку. Показано, що в сучасній обчислювальній системі операційна система займає особливе місце, від ефективності роботи якої залежить як функціонування обчислювальної установки в цілому, так і ефективність проходження робіт через неї.

У розділі розглядаються наступні питання:

- Основні визначення.
- Історія розвитку операційних систем.
- Концепція побудови операційних систем (процес, файл, виклик, ядро).
- Класифікація операційних систем і їх структурні особливості.
- Поняття генерації, інсталяції, ініціалізації ОС.
- Ядро операційної системи, структура, склад, взаємодія.
- Загальна схема завантаження операційної системи.
- Загальна схема функціонування операційної системи (на прикладі MS DOS)

1.1. ОСНОВНІ ВИЗНАЧЕННЯ

Операційна система являє собою сукупність програмних модулів, орієнтованих на виконання функцій внутрішнього й зовнішнього характеру.

До перших з них відносяться функції: ефективного керування ресурсами системи, розподілу (якщо буде потреба) ресурсів між декількома користувачами, виділення ресурсів для одночасного виконання багатьох задач і контролю їх використання, контроль виконання задач користувача від надходження їх у ОС до одержання результату.

До функцій зовнішнього характеру відносяться функції надання набору послуг, що забезпечують користувачеві зручний інтерфейс — "віртуальна машина", який надає користувачеві набір засобів для керування інформацією й ресурсами обчислювальної системи, реалізацією прикладного програмного забезпечення. Реалізація цих функцій звільняє користувача від врахування особливостей апаратурних засобів обчислювальної системи.

Програми, що виконуються обчислювальною системою, за функціональним призначенням діляться на два класи: прикладні й системні. Операційні системи становлять важливу частину програм другого класу.

Операційна система *— програмне середовище, що забезпечує підтримку роботи всіх програм, їх взаємодію з апаратурними засобами обчислювальної системи, а також надає користувачеві можливості керування обчислювальною системою й використання ресурсів ОС.*

Операційні системи відрізняються архітектурою, можливостями, потрібними ресурсами для їхнього функціонування, набором сервісних функцій, орієнтацією на типи використовуваних мікропроцесорів і іншими особливостями. Розвиток операційних систем нерозривно зв'язаний й у значній мірі визначався розвитком технічного забезпечення обчислювальних систем, функціональними й структурними особливостями компонентів ОС і впровадженням нових принципів організації обчислювального процесу на апаратному й програмному рівні.

1.2. ЕТАПИ РОЗВИТКУ ОПЕРАЦІЙНИХ СИСТЕМ

Довгий часз моменту появи перших цифрових обчислювальних машин не існувало системного програмного забезпечення (операційних систем).

Користувач одержував можливість повністю розпоряджатися обчислювальною системою і її ресурсами на строго певний час. Це були машини значних розмірів, зроблені на ненадійних електронних лампах. Програми, написані машинною мовою, працювали на "голій" машині без будь-якого проміжного системного програмного забезпечення. Користувачі вводили свої програми, написані машинною мовою, у комп'ютер за допомогою машинних носіїв (перфокарт, перфострічок або, у найперших комп'ютерах, за допомогою комутаційної панелі вручну), завантажували програми, чекали їхнього виконання й відносили роздруківки з вихідними даними для подальшого аналізу й використання. У деякому сенсі користувачі-програмісти виконували функції операційної системи, тому що вони управляли всією послідовністю дій обчислювальної машини. Ефективність використання її ресурсів цілком залежала від якості їх дій. Пряма взаємодія з машиною (покрокове виконання програми, безпосередній контроль і зміна стану комірок пам'яті) було головним засобом відлагодження програми. На першому етапі програмісти повинні були самі писати програми виконання процедур вводу/виводу для своїх програм. Друк символу, наприклад, вимагав багато машинних команд, тому що в комп'ютері не було відповідної функції. Якщо цього не було у вашій програмі, то цього не було ніде. Потім з'явилися бібліотеки стандартних програм вводу/виводу й обчислення арифметичних, тригонометричних функцій і програмісти вже могли їх використовувати, вписуючи відповідні програмні коди в тексти своїх програм. Було відсутнє також загальне керування пам'яттю. Кожна програма мала доступ до всього простору пам'яті, а звідси і її неефективне використання.

Підготовка обчислювальної машини до виконання роботи містила в собі запис програм у двійковому коді за допомогою перемикачів на передній панелі в пам'ять машини, що займало дуже багато часу й передбачувало появу помилок у керуванні. Така робота була вкрай неефективною, тому що дороге обладнання використовувалося неекономно. Однак надання в

розпорядження користувача всіх ресурсів обчислювальної машини було безсумнівною перевагою такого режиму взаємодії людини з машиною й, пройшовши значний шлях розвитку, ми повернулися до цієї форми зараз із використанням персональних комп'ютерів. Однак, керування ресурсами обчислювальної машини здійснюється за допомогою зручного «дружнього інтерфейсу».

Першими програмними системними засобами, що підвищують ефективність роботи користувача на обчислювальній установці, а звідси й роботу програміста, були, з одного боку, засоби автоматизації розробки програм (асемблери, компілятори, відладники), а з іншого підпрограми вводу/виводу.

Покоління операційних систем:

— *ПЕРШЕ ПОКОЛІННЯ.* У середині 50-х років у комп'ютерах стала використовуватися нова елементна база (транзистори). Це зробило машини більш надійними, і сфера їх застосування розширилася. Однак принцип роботи практично не змінився: програміст або професійний оператор завантажував програму, чекав поки вона буде виконана, потім знову чекав, коли будуть надруковані результати, і тільки потім він завантажував наступну програму. Стало зрозуміло, що таке використання дорогого швидкодіючого обладнання є вкрай неефективним. Одна з перших проблем, яку повинні були розв'язати операційні системи, це звільнення центрального процесора від участі в керуванні вводом/виводом. Тому що фізичний ввід й вивід даних займали час, який, в основному визначався швидкодією пристроївводу/виводу, то обчислювальні можливості центрального процесора (ЦП) при цьому не використовувалися, тому що швидкості ЦП і пристроївводу/виводу непорівнянні. Очікуючи, поки відбудеться ввід/вивід, ЦП звичайно працював вхолосту, розтрачуючи свій ресурс.

Першим рішенням проблеми підвищення ефективності використання ЦП була проста пакетна обробка. При такому підході перфокарти багатьох програм зчитувалися додатковим низько швідкісним комп'ютером на

магнітну стрічку, підготовляючи вхідну чергу завдань - пакет. ЦП обробляв програми пакета одну за іншою й виводив результати на іншу магнітну стрічку. Тому що накопичувачі на магнітній стрічці були швидші, ніж обладнання перфовводу й принтери, то положення справ покращилося. Таким чином, проблема простою ЦП під час вводу/виводу в якійсь мірі вирішувалася за рахунок прискорення самого процесу вводу/виводу.

В кінці 50-х років з'явилися перші системні програми - пакетні монітори. Ці програми створювалися з метою автоматизації виконання всієї послідовності робіт (вводу/виводу даних і виконання прикладних програм), підготовлених заздалегідь, і процесу переходу від однієї роботи до іншої. Основною функцією такої системи було керування ресурсами: пам'яттю, процесором, вводом/виводом. Автоматизація керування ресурсами ОС викликала додавання в операційні системи контролюючих функцій для захисту ОС від можливих помилок при виконанні робіт, виконуваних послідовно в пакеті:

- контроль часу виконання процесором кожної роботи для усунення можливості блокування всієї роботи через зациклення однієї програми;
- контроль звертання до обладнань вводу/виводу, щоб уникнути циклічного звертання до периферійних обладнань помилково;
- захист області пам'яті, що постійно займана монітором (операційною системою), від помилок адресації в користувацьких програмах.

Забезпечення зазначених функцій привело до розширення можливостей використання машин. Став можливим ввід: служб виміру й обліку часу, обмеження на використання деяких інструкцій, захисту пам'яті.

Використання пакетних моніторів суттєво підвищило продуктивність обчислювальних систем і зменшило непродуктивні витрати машинного часу при зміні завдань. Однак це підвищення в значній мірі знецінювалося тим, що при виконанні операцій вводу/виводу процесор використовувався неефективно. Для підвищення ефективності роботи ЕОМ стали застосовувати спеціалізовані системи вводу/виводу на основі використання

допоміжних комп'ютерів. Програми виконувалися на одному комп'ютері (головному), оснащеному пристроєм з підвищеною швидкістю запису й зчитування даних з магнітної стрічки, а допоміжний (периферійний) спеціалізований комп'ютер або спеціальні пристрої служили для запису даних з перфокарт на стрічку й виводу результатів обчислень із магнітної стрічки на друк.

Операційна система цих систем мала резидентну (що постійно перебуває в оперативній пам'яті) керуючу програму, яка зчитувала, інтерпретувала й виконувала команди, написані з використанням спеціальної мови - JCL (мова керування завданнями), розташованими між програмами, що підлягають зчитуванню й виконанню.

Попереднє планування послідовності робіт дозволяло працювати обом машинам паралельно й тим самим більш повно використовувати можливості головного процесора. Недолік такої організації - негнучкість системи (тимчасові обмеження для вводу завдань, довге очікування результатів). Проте, системи послідовної пакетної обробки отримали широке поширення на початку 60-х років.

— *ДРУГЕ ПОКОЛІННЯ*. У період з 1960 по 1970 р. значний прогрес в галузі технології матеріалів і в розробці нових принципів функціонування операційних систем дозволив усунути недоліки систем пакетної обробки й перейти до систем з колективним доступом.

Використання автономних, спеціалізованих процесорів (каналів або пристроїв обміну) для передачі інформації між пристроямивводу/виводу й обчислювальною системою дозволило звільнити центральний процесор від функцій прямого керування пристроямивводу/виводу.

З'явилися перші системи з багатопрограмним режимом роботи на основі використання мультипрограмування. Мультипрограмування й пов'язаний із цим поділ пам'яті для кількох робіт, підвищило ефективність використання центрального процесора, за рахунок одночасної роботи процесора й кількох пристроївввводу/виводу.

Однак, для рівномірного завантаження обладнання в таких системах був потрібен ретельний добір задач, виконуваних у режимі мультипрограмування. У більшості випадків програми, що виконувалися, не підходили одні одним, і це ускладнювало забезпечення рівномірності завантаження процесора й пристроїв вводу/виводу. Однак, при запуску більш ніж двох програм одночасно, час центрального процесора, як найбільш дорогого ресурсу обчислювальної машини, використовувався більш ефективно.

Звичайно, функціонування ОС у багато задачному режимі вимагає використання більш складного системного програмного забезпечення (операційної системи), щоб стежити за тим, де розташована та або інша програма, у якій стадії виконання вона перебуває й, разом з апаратним забезпеченням, запобігати несанкціонованому впливу однієї програми на іншу.

Служба керування пам'яттю в перших мультипрограмних системах була досить простою: вона повинна була стежити, щоб кожна програма займала тільки виділену їй частину оперативної пам'яті й залишалася там аж до завершення й виводу.

Однак, програма може бути занадто великою, щоб уміститися у відведеній їй частині, або навіть настільки великою, щоб не вміститься у всій пам'яті машини. Для обліку цього стали застосовувати оверлейні структури програм. У цьому випадку програми повинні були ділитися на оверлеї - окремі частини програми, які викликали одна одну по мірі необхідності, але це створювало додаткові труднощі програмістові, тому що саме йому доводилося ділити програму на частини підходящого розміру й стежити за безконфліктним викликом однієї програми іншою.

Кращим рішенням цієї проблеми було використання віртуальної організації пам'яті, де доступний простір пам'яті ОС для кожної програми виглядав більше, ніж наявний у дійсності обсяг реальної оперативної пам'яті. Програма могла бути поділена на частини (сторінки, звичайно однакового

розміру). Ті сторінки програми, які були активні (тобто ті сторінки, команди з яких виконувалися в цей час або в них зберігалися необхідні дані), зберігалися в оперативній пам'яті, тоді як ті, які не використовувалися, тимчасово записувалися на швидкодіючий диск. Цей процес повністю управлявся операційною системою й був непомітний програмістові, який міг писати програми, що займають більше пам'яті, ніж було фізично доступно.

Почали застосовувати бібліотеки стандартних програм. Першою була створена бібліотека стандартних програм вводу/виводу, щоб користувачам не доводилося створювати їх з нуля. Операційні системи стали використовувати власні функції виводу символу на екран і виконували інші задачіпристроїв вводу/виводу. Логічніше просто використовувати ці функції, ніж щоразу повторювати їх у кожній програмі. Так народилася ідея системних функцій, які дотепер застосовуються в більшості операційних систем.

Хоча мультипрограмування й змушувало ЦП працювати більшу частину часу, воно ніяк не допомагало розв'язати інші проблеми: ефективну роботу користувача й ефективну роботу всієї обчислювальної установки в цілому. Програміст звичайно приносив програму у вигляді колоди перфокарт у машинний зал і віддавав їх операторові. Потім потрібно було чекати, поки вона виконається, що займало іноді кілька годин тому що завдання виконувалося в пакеті. І, нарешті, програміст забирав роздруківки з результатами з машинного залу й дивився, що вийшло. Часу звичайно вистачало на 2-3 таких прогона в день, тож налагодження було дуже тривалою процедурою (вражає, що якесь програмне забезпечення в той час все-таки було написане).

Рішенням проблеми недостачі машинного часу й підвищення ефективності праці програміста було введення систем поділу часу. При поділі часу кілька терміналів зв'язувалися з одним ЦП, і кожний одержував серію маленьких проміжків (квантів) часу ЦП. Такяк людина звичайно працює повільніше процесора, то користувач не зауважував, що між натисканням "1" і "s" при набиванні рядка "1s" ЦП обслуговує ще декількох

користувачів. Результатом стало інтерактивне обчислювальне середовище, де створюється ілюзія, що комп'ютер озивається відразу після вводу. Це режим удаваного суміщення. Основною метою функціонування систем з поділом часу є покращення роботи користувача. У найбільш розвинених системах з поділом часу результати користувачеві відправлялися по мірі їх одержання, що ще більше підсилювало ефект "індивідуального" використання ресурсів ОС користувачем.

Першою широко використовуваною системою з поділом часу була операційна система UNIX, яка з тих пір одержала широке поширення й була перенесена на безліч різних машин.

У середовищі з поділом часу перед операційною системою ставилися інші задачі, ніж при забезпеченні пакетної обробки. Вона усе ще залишається мультипрограмною системою, але акцент робить не на оптимізацію використання ЦП, а на мінімізацію часу відгуку обчислювальної системи на дії користувача. Це дозволяє користувачеві працювати комфортно й бачити результати обчислень через досить короткий час. Звичайно, що при підключенні до системи занадто великої кількості користувачів, час відгуку збільшиться, а загальні характеристики системи будуть погіршуватися. Система буде намагатися задовольнити всіх користувачів рівною мірою, забезпечивши кожного з них однаковою частиною машинного часу.

Задача розподілу часу ЦП між задачами в системах з поділом часу набагато більш складна, ніж в системах пакетної обробки. Вона вирішується спеціальною частиною операційної системи, що називається планувальником. Принципи, закладені в планувальнику, є важливими факторами, що впливають на характеристики системи. У системі з поділом часу кожна програма працює протягом короткого відрізка часу (говорять, що програма одержує квант часу). Коли проміжок часу, виділений програмі, закінчується, починає працювати інша програма, навіть якщо перша програма не була завершена. Це перемикання й принципи вибору наступної програми для виконання визначається пріоритетною системою й

застосовуваними в ОС дисциплінами обслуговування заявок (системами правил обслуговування заявок). Кожний процес або є пріоритетним для обробки, або відключається й заміщається іншим, більш пріоритетним. Правила можуть динамічно змінюватися виходячи з міркувань загальної ефективності системи обслуговування.

Зауважимо, що коли ми говоримо про дві й більше програми, що працюють одночасно в системі з поділом часу, ми не маємо на увазі точну, фізичну одночасність, тому що ЦП віддає квант часу тільки одній програмі. Хоча користувачам і здається, що багато програм працюють паралельно, але в дійсності ЦП запускає програму користувача А на кілька міллісекунд , потім програму користувача Б, і так далі, до кількох десятків. Потім він вертається, щоб дати користувачеві А наступний квант часу. Тільки швидкість цього "карусельного" перемикання часу ЦП створює для користувача ілюзію одночасності, тому що час реакції користувача значно вище часу реакції ОС.

Операційна система з поділом часу, крім цього, повинна управляти ресурсами. Як і в реальному світі, ресурси - це те, чого завжди не вистачає, тобто те, що потрібно ділити між багатьма користувачами. Прикладом ресурсів можуть служити процесор, пристрої вводу/виводу й пам'ять. Операційна система підтримує загальний порядок одночасної роботи всіх ресурсів, здійснюючи ввід завдань, вивід результатів роботи програм користувачів на принтер, розподіляє виділення часу процесора й пам'яті так ефективно й справедливо, як це можливо.

Методи розподілу й роботи з пам'яттю досить складні й різноманітні. Один із загальних підходів ефективної організації роботи з пам'яттю полягає в тому, щоб поділити кожну програму користувача на сегменти. Сегменти можуть відповідати логічним частинам програми, наприклад один сегмент - для команд, один сегмент - для даних і один - для стека. Сегменти можуть заноситися впам'ять й віддалятися з неї за необхідністю. Вони також можуть бути захищені так, що тільки одна програма буде мати до них доступ. Або

навпаки, вони можуть бути спільними, роблячи групу даних доступною багатьом користувачам.

Такяк в системі з поділом часу є безліч користувачів, то виникають додаткові складності. Повинна бути дотримана конфіденційність окремих програм, так щоб користувач не міг випадково або навмисно змінити (зіпсувати) чужу програму або дані або, що ще гірше, зруйнувати саму систему. Бажано, щоб система мала найпростіші методи контролю. Наприклад, вона повинна реєструвати, як довго кожний користувач працював із системою і які дії він робив. Саме при розробці систем з поділом часу й були введені поняття: переривання й логічних пристроїв. Переривання й логіка обробки переривань дозволили перейти до впровадження операційних систем із принципово новими функціональними можливостями.

— *ТРЕТЄ ПОКОЛІННЯ.* На 70-і роки припадає початок розквіту операційних систем. У цей час були закладені принципи сумісності операційних систем "знизу нагору". Розробляються багато режимні операційні системи, призначені для різних конфігурацій обчислювальних систем. Однак для цих систем характерне ускладнення доступу (безпосереднього керування) користувача до апаратних ресурсів обчислювальної системи. Користувачі могли управляти діями обчислювальної системи за допомогою, досить складної, спеціальної мови керування завданнями.

70-і роки відзначено двома подіями:

- Появою мікропроцесорів і постійним зростом їх можливостей, які дозволили забезпечити значні обчислювальні потужності при відносно низькій вартості. Саме з появою мікропроцесорів перестав діяти закон Гроша (HerbGrosh) - швидкодія процесора пропорційна квадрату його вартості.
- Розвитком техніки передачі даних (телеобробки) і постійно зростаючим ступенем інтеграції сучасних засобів зв'язку в інформаційно-обчислювальні системи.

Одночасно із цим досвід експлуатації обчислювальних систем сформував нові вимоги користувачів до них:

- необхідність поділу ресурсів між виконуваними роботами, яке було б виправдане з економічної точки зору й можливістю доступу до віддаленої інформації (ресурси й інформація можуть перебувати в географічно різних місцях);
- необхідність найкращої відповідності й адаптації обчислювальної системи до структури прикладних задач, що приводить до децентралізації системи й географічному розподілу елементів ОС;
- необхідність об'єднання прикладних задач (раніше роз'єднаних) у єдину сукупність об'єктів на єдиних системних принципах використання й взаємодії.

У результаті з'явилися нові типи обчислювальних систем:

- мережі телеобробки даних, що дозволяють організувати взаємозв'язок між існуючими системами, обмін даними між ними й доступ до віддаленихпристроїв інформаційного обслуговування.

- локальні мережі ЕОМ з високошвидкісними каналами передачі інформації (10 Мбіт/с-1Гбіт/с), географічно сконцентровані й виконуючі функції:

- систем керування;
- систем зв'язку й керування документацією (систем для установ), орієнтованих на обмін інформацією, створення й зберігання документів;
- систем загального користування, призначених, зокрема, для створення програмного забезпечення.

У той же час, передбачуваний розвиток мереж і персональних комп'ютерів не виключає існування великих централізованих систем з доступом у режимі поділу часу, які залишаються економічно виправданими для численних застосувань. Задачі обчислювальної математики в ряді спеціальних галузей вимагають розвитку обчислювальних систем дуже великої потужності, що

включають спеціалізовані мультипроцесори, для яких повинна бути розроблена своя архітектура операційних систем.

В 1965 році фірма Bell Telephone Laboratories, об'єднавши свої зусилля з компанією General Electric і проектом MAC Массачусетського технологічного інституту, розпочали розробку нової операційної системи, що реалізує нові вимоги до керування обчислювальними системами. Система одержала назву Multics. Перед системою Multics були поставлені задачі: забезпечення одночасного доступу до ресурсів ЕОМ великої кількості користувачів, забезпечення достатньо високої швидкості обчислень, доступу до даних і можливості користувачам якщо буде потреба спільно використовувати дані. Багато розробників, що згодом прийняли участь у створенні ранніх редакцій системи UNIX, брали участь у роботі над системою Multics у фірмі Bell Telephone Laboratories. Перша версія системи Multics була запущена в 1969 році на ЕОМ GE 645, але вона не забезпечувала виконання головних обчислювальних задач, для розв'язку яких вона призначалась.

Намагаючись удосконалити середовище програмування, Кен Томпсон, Деніс Річі й інші створили проект файлової системи, що одержав пізніше подальший розвиток у ранній версії файлової системи UNIX. Найбільш повну версію нової операційної системи Томпсон і Річі впровадили для ЕОМ PDP-7, що включила першу версію файлової системи UNIX, підсистему керування процесами й невеликий набір утиліт. Нова система більше не потребувала підтримки з боку системи GECOS у якості операційного середовища розробки й могла підтримувати себе сама. Нова система одержала назву UNIX, за подібністю з Multics. Цю назву придумав один зі співробітників Дослідного центру з інформатики Брайан Керніган.

Незважаючи на те, що ця рання версія системи UNIX уже була багатообіцяючої, вона не могла реалізувати свій потенціал доти, поки не набула застосування в реальному проекті. Перше реальне застосування вона знайшла для забезпечення функціонування системи обробки текстів для

патентного відділу фірми Bell Telephone Laboratories. В 1971 році система UNIX була перенесена на ЕОМ PDP-11. Система відрізнялася невеликим обсягом: 16 Кбайт для системи, 8 Кбайт для програм користувачів, обслуговувала диск обсягом 512 Кбайт і відводила під кожний файл не більш 64 Кбайт. Нова система вимагала появи нової мови. Спочатку Томпсон збирався написати для нової системи транслятор з Фортрану, але замість цього зайнявся мовою Бі (B), попередником якої була мова BCPL. Бі була мовою інтерпретуючого типу з усіма недоліками, властивими подібним мовам, тому Річі переробив його в новий різновид, що одержав назву Сі (C) і дозволив генерувати машинний код, повідомляти типи даних і визначати структуру даних. В 1973 році операційна система була написана заново на Сі – мові високого рівня. Це був крок, нечуваний для того часу, але він мав величезний резонанс серед широкого кола користувачів. Кількість машин фірми Bell Telephone Laboratories, на яких була інстальована система, зросло до 25, у результаті чого була створена група по системному супроводу UNIX усередині фірми.

У той час корпорація AT&T поширювала систему UNIX серед університетів, яким вона була потрібна в навчальних цілях. Додержуючись букві угоди, корпорація AT&T не рекламувала, не продавала й не супроводжувала систему. Незважаючи на це, популярність системи стійко росла. В 1974 році Томпсон і Річі опублікували статтю, що описує систему UNIX, у журналі Communication softhe ACM, що дало ще один імпульс до поширення системи. До 1977 року кількість машин, на яких функціонувала система UNIX, збільшилося до 500, при чому 125 з них працювали в університетах. Система UNIX здобула популярність серед телефонних компаній, оскільки забезпечувала гарні умови для розробки нових програм, обслуговувала роботу в мережі в режимі діалогу й роботу в реальному масштабі часу. Крім університетів, ліцензії на систему UNIX були передані комерційним організаціям. 1977 рік також був відзначений "переносом" системи UNIX на машину, відмінну від PDP (завдяки чому став можливий

запуск системи на іншій машині без змін або з невеликими змінами), а саме на Interdata 8/32.

З ростом популярності мікропроцесорів інші компанії стали переносити систему UNIX на нові машини, однак її простота і ясність спонукали багатьох розроблювачів до самостійного розвитку системи, у результаті чого було створено кілька варіантів базисної системи. За період між 1977 і 1982 роком фірма Bell Laboratories об'єднала кілька варіантів, розроблених у корпорації AT&T, в один, що одержав комерційну назву UNIX версія III. Надалі фірма Bell Telephone Laboratories додала у версію III кілька нових особливостей, назвавши новий продукт UNIX версія V, і ця версія стала офіційно поширюватися корпорацією AT&T із січня 1983 року. У той же час співробітники Каліфорнійського університету в Берклі розробили варіант системи UNIX, що одержав назву BSD 4.3 для машин серії VAX і відрізнявся деякими новими, цікавими особливостями.

До початку 1984 року система UNIX була вже інстальована приблизно на 100000 машин по усьому світу, причому на машинах із широким діапазоном обчислювальних можливостей - від мікропроцесорів до великих ЕОМ - і різних виготовлювачів. Ні про яку іншу операційну систему не можна було б сказати того ж. Популярність і успіх системи UNIX пояснювалися тим, що вона написана мовою високого рівня, завдяки чому її легко читати, розуміти, змінювати й переносити на інші машини. По оцінках, зроблених Річі, перший варіант системи на Сі мав на 20-40 % більший обсяг і працював повільніше в порівнянні з варіантом на асемблері, однак система мала ряд переваг обумовлених використанням мови високого рівня:

- наявність досить простого користувацького інтерфейсу, у якому є можливість надавати всі необхідні користувачеві послуги;
- наявність елементарних засобів, що дозволяють створювати складні програми з більш простих;
- наявність ієрархічної файлової системи, легкої в супроводі й ефективної в роботі;

- забезпечення узгодження форматів у файлах, робота з послідовним потоком байтів, завдяки чому полегшується читання прикладних програм;
- наявність простого, послідовного інтерфейсу з периферійними пристроями;
- система є багатокористувацької, багато задачною, кожний користувач може одночасно запускати кілька процесів;
- архітектура машини схована від користувача, завдяки цьому полегшений процес написання програм, що працюють на різних конфігураціях апаратних засобів.

Операційна система й більшість команд написана на Сі. Однак, система UNIX підтримує ряд інших мов, таких як Фортран, Бейсік, Паскаль, Ада, Кобол, Лісп і Пролог. Вона може підтримувати будь-якумову програмування, для якої є компілятор або інтерпретатор, і забезпечувати системний інтерфейс, що встановлює відповідність між користувацькими запитами до операційної системи, і набором запитів, прийнятих в UNIX.

— *ЧЕТВЕРТЕ ПОКОЛІННЯ.* Наприкінці 70-х років розвиток технології виготовлення БІС знизило вартість і розміри одного транзистора до таких меж, що стало доцільним робити персональні комп'ютери. Це викликало поворот до ситуації з однієї програмою й одним користувачем, як у часи перших комп'ютерів, і можливість повного володіння й керування ресурсами системи. Перший масовий ринок персональних комп'ютерів включав такі моделі, як IMSAI 8080, Radio Shach TRS-80 Model I, Apple II і пізніше IBM PC.

Перші операційні системи персональних комп'ютерів (ПК) були досить простими. Їм потрібно було обслуговувати дію тільки однієї програми, так що їх основним завданням стало керування файлами й забезпечення виконанням ряду корисних для програміста функцій. Для першого покоління ПК користувачем був програміст, що працював мовою асемблера, тому що перші машини не мали можливостей для виклику засобами мов високого рівня (зазвичай BASIC) системних функцій, вбудованих в операційну

систему. Першими операційними системами на оснащених процесорами Intel комп'ютерах були CP/M, а с появою IBM PC - MS-DOS. MS-DOS довгий час була найпоширенішою операційною системою. Однак незабаром стало очевидно, що робота персонального комп'ютера в однозадачному режимі не ефективна. Операційна система, яка може завантажувати й забезпечувати виконання тільки однієї програми, обмежує продуктивність праці користувачів і не забезпечує ефективне використання ресурсів машини.

Спочатку розроблювачі спробували забезпечити обмежені форми мульті програмності в MS-DOS за допомогою використання TSR - програм (програм які завершуються й залишаються резидентними в пам'яті), але цей розв'язок далекий від досконалості. З одного боку, TSR - програми не забезпечують реальну багато задачність. З іншого боку, TSR - програми, у зв'язку з обмеженнями в системі переривань, конфліктують одна з одною, тому що архітектура MS-DOS не створювалася для їхнього використання.

Термін "багатозадачность" часто плутають із терміном "мультипрограмування". Дійсно обоє вони відносяться до системи, у якій кілька програм можуть перебувати в пам'яті й виконуватися одночасно. Однак, з позицій операційної системи "задача" означає не зовсім те ж саме, що "програма". Це буде детально розглядатися пізніше, коли ми будемо розглядати процеси й ланцюжки.

Багато компаній пропонували операційні системи, які мали деякі риси багатозадачної системи, наприклад, Desqview фірми Quarterdeck Office System, Windows фірми Microsoft і PC-MOS фірми SoftwareLink. Серед систем такого класу в ті роки слід виділити операційну систему OS/2, що має ряд відмінних рис, що дозволяють відзначити її такими фірмами як IBM і Microsoft як найбільш перспективну ОС. OS/2 тоді була однієї з основних одно користувацьких багато задачних операційних систем, розроблених спеціально для персонального комп'ютера. Таким чином, можна відзначити, що в 80-і роки інтенсивно розробляються й поширюються операційні системи нового покоління.

—П'ЯТЕ ПОКОЛІННЯ. Основною особливістю сучасних операційних систем є забезпечення дружнього користувацького інтерфейсу для різних категорій користувачів. Вони реалізують такі елементи людино - машинної взаємодії, як багато віконний інтерфейс із користувачем, перемикання між програмами (і їх вікнами) і т.д. Операційні системи містять потужний набір графічних примітивів, що забезпечують стандартизацію роботи на різних типах графічних пристроїв.

Протягом останніх 40 років був розроблений ряд вдалих операційних систем, деякі з яких наведені в таблиці табл. 1.1.

1.3. КОНЦЕПЦІЯ ПОБУДОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Частина операційної системи, що постійно перебуває в основній пам'яті, називається *ядром системи або резидентною* частиною операційної системи. Програми, викликувані в основну пам'ять для виконання певних функцій, та які не зберігаються там постійно, називаються *транзитними програмами* або просто *транзитами*.

Табл. 1.1

Операційна система	Рікрозробки	Розроблювач
Atlas	1959	Манчестерський університет, Англія
CTSS	1963	Масачусетський технологічний інститут (MTI), США
OS/360 (1)	1964	IBM, США
THE	1967	Эйндховенський технологічнийуніверситет, Голландія
RC 4000	1970	Університет міста Орхуса, Данія
OS/360 (21)	1970	IBM, США
UNIX (1)	1972	BellLaboratories, США
Multics	1975	BellLaboratories, MTI, США
MVS	1976	IBM, США
VMS	1980	DigitalEquipmentCorp., США
CP/M	1983	DigitalResearchInc., США
MS-DOS	1983	Microsoft Corp., США
UNIX (7.5)	1983	BellLaboratories, США
UNIX (V)	1985	BellLaboratories, США
OS/2 Warp	1994	IBM, США
WINDOWS-95	1995	Microsoft Corp., США
WINDOWS NT	1993-94	Microsoft Corp., США
WINDOWS-98	1998	Microsoft Corp., США
WINDOWS-2000	2000	Microsoft Corp., США
WINDOWS - XP	2002	Microsoft Corp., США

Транзити викликаються в ділянку основної пам'яті, який називається транзитною областю керуючої програми. Уся область основної пам'яті, займана ядром і транзитами, називається областю керуючої програми або системною областю. Інша частина основної пам'яті утворює область проблемних програм або область користувача. Поділ керуючої програми на ядро й транзити дозволяє мінімізувати область представлення керуючої програми в основній пам'яті й, відповідно, збільшити область пам'яті виділюваної для проблемних програм.

У силу того, що транзити викликаються в міру необхідності, то вони можуть займати одну й ту саму область основної пам'яті, вони можуть перекривати одна одну. Через це їх часто називають програмами, що перекриваються, або програмами із запланованим перекриттям.

Операційна система спроектована для роботи в різних режимах. Тому на вхід системи може одночасно надходити велика кількість заявок (задач), виконати які відразу неможливо просто через те, що немає достатньої кількості фізичних ресурсів. До фізичних ресурсів системи відносяться центральний процесор, місце в основній і допоміжній пам'яті, окремі програми, пристрої керування вводом/виводом, канали, таймер, консоль оператора. Усі заявки, що надходять на вхід системи, являють собою незалежні ***завдання***, які слід розглядати як зовнішні (користувацькі), а отже незалежні одиниці роботи для операційної системи. Будь-яка робота, виконувана в системі, називається ***задачею***, яку слід розглядати як об'єкт системи, якій вона виділяє ресурси.

Реалізація користувацьких задач в операційній системі здійснюється на двох рівнях: рівні користувача й рівні ядра. Коли задача робить звертання до операційної системи, режим виконання перемикається з користувацького режиму на режим ядра й операційна система намагається обслужити запит користувача. Навіть якщо користувач не має потреби в яких-небудь певних послугах операційної системи й не звертається до неї із запитами, система самостійно повинна виконувати облікові операції, пов'язані з

користувацькими задачами. При цьому вона обробляє переривання, планує обслуговування задач, управляє розподілом пам'яті і т.д. Більшість обчислювальних систем різноманітної архітектури (і відповідні їм операційні системи) підтримують більше число режимів і рівнів, ніж зазначене тут, однак уже двох згаданих режимів цілком достатньо для функціонування операційної системи.

Основні відмінності між цими двома режимами:

- у режимі користувача, задачі мають доступ тільки до своїх власних інструкцій і даних, але не до інструкцій і даних ядра супервізора (або інших задач);
- у режимі ядра задачам уже доступні адресні простори ядра супервізора й користувачів.

При активізації задач віртуальний адресний простір пам'яті задач може бути поділений на адреси, доступні тільки в режимі ядра, і на адреси, доступні в будь-якому режимі.

Деякі машинні команди є привілейованими, а їх використання в режимі задачі викликає виникнення помилок. Наприклад, у машинній мові може бути команда, що управляє регістром стану процесора й задачам, що виконуються в користувацькому режимі, вона недоступна.

Простіше кажучи, будь-яка взаємодія з апаратурою описується в термінах режиму ядра й режиму задачі й протікає однаково для всіх користувацьких програм, що виконуються в цих режимах. Операційна система зберігає внутрішні записи про стан кожної з безлічі завдань, що виконуються в системі. Незважаючи на те, що система функціонує в одному із двох режимів, ядро супервізора діє для забезпечення виконання користувацької задачі й від її імені. Таким чином, ядро супервізора не є якоюсь особливою сукупністю задач, що виконуються паралельно з користувацькими. Воно саме виступає складовою частиною будь-якої користувацької задачі й призначене для забезпечення проходження роботи (завдання) користувача через обчислювальну систему.

При створенні операційних систем уперше широко стали застосовувати модульний принцип програмування. Модульний принцип програмування дозволяє досить швидко виконувати розробку складних програмних комплексів. Модульний принцип заснований на дотриманні для кожного модуля наступних угод: модулі повинні мати стандартну внутрішню структуру, мати параметричну універсальність, функціональну завершеність, взаємну незалежність. Подальший розвиток модульний принцип знайшов в об’єктно – орієнтованому програмуванні й IP (International Programming) технології програмування, яка використовується фірмою MICROSOFT. При цьому використовується добре розвинена система бібліотек і багаторівневі системи програмування.

У зв'язку з тим, що операційні системи сучасних ОС є складними керуючими програмними комплексами, то для підвищення ефективності роботи обчислювальної системи й прискорення прийняття рішень при керуванні застосовується табличний метод керування. Він заснований на використанні спеціальних системних об'єктів — керуючих таблиць. У них, у стислому виді зосереджена вся інформація про стан системи і її зміни. Системні таблиці формуються при завантаженні операційної системи при її ініціалізації й перебувають у системній області оперативної пам'яті (ядрі операційної системи - ядрі супервізора). Складність обчислювальної системи як об'єкта керування обумовила використання принципу багаторівневого керування. Цей принцип дозволяє користувачеві, при керуванні ресурсами ОС, працювати на логічному рівні не дбаючи про технологічні особливості обладнання.

1.4 КЛАСИФІКАЦІЯ ОПЕРАЦІЙНИХ СИСТЕМ І ЇХСТРУКТУРНІ ОСОБЛИВОСТІ

Розрізняють наступні типи операційних систем:

- Однопрограмні;
- Багатопрограмні;
- Однопроцесорні;

- Багатопроцесорні;
- Мережні;
- Розподілені;
- Віртуальні;
- Вбудовані (бортові).

Система працює в *однопрограмному* режимі, якщо в системі перебуває одне або кілька завдань, але всі ресурси системи віддано одній задачі й вона не може бути перервана до свого завершення, крім, зрозуміло, її аварійного завершення.

Система працює в *багатопрограмному* режимі, якщо в ній перебуває кілька задач у різній стадії виконання, і кожна з них може бути перервана іншою з наступним поверненням у точку переривання.

Багатопрограмний режим в однопроцесорній системі - це організація обчислювального процесу із плануванням у часі. При цьому операційна система забезпечує виконання активізованих задач на тому самому обладнанні, розділяючи й синхронізуючи їх роботу в часі. При цьому операційна система дбає про ефективність роботи всієї обчислювальної системи в цілому й поліпшення її макроекономічних показників (завантаження обладнання, часу очікування, продуктивності). У функції такої операційної системи входить також захист від впливу однією задачею на іншу. Основні турботи про синхронізацію взаємодіючих задач у рамках одного завдання покладені на програміста. У таких системах особливе значення набувають методи й способи реалізації двох основних функцій ОС. Невдала реалізація системних функцій може в значній мірі знизити показники загальної продуктивності ОС.

Багатопрограмний режим у багатопроцесорній системі— це планування в часі й у просторі. У цьому режимі операційна система, виконуючи одну зі своїх основних функцій (забезпечення ефективності роботи ресурсів), повинна розподіляти активізовані задачі по процесорах (у просторі) і планувати черговість їх роботи в часі. У тому випадку, якщо кількість задач

більше кількості процесорів, задача планування, диспетчеризації ускладнюється. У зв'язку зі складністю розв'язку питань розподілу задач по процесорах, їх розв'язок виконується або до активізації завдань у багатопроцесорній системі (статичне планування), або вирішуються простими способами, що не дають оптимального розв'язку.

У випадку застосування масштабованих систем масового розпаралелювання або розподілених систем обробки інформації, коли обчислювальна система може виділити стільки обчислювальних ресурсів, скільки потрібно для обслуговування вхідного потоку задач, планування в часі можна виключити. Це до деякої міри полегшує задачу операційної системи по плануванню, диспетчеризації й організації обчислювальних процесів.

- ***Режими експлуатації обчислювальної системи***

Широке впровадження засобів обчислювальної техніки в усі сфери людської діяльності призвело до появи й розвитку різноманітних обчислювальних систем (ОС), що різняться по архітектурі, структурі, режимах роботи, формах взаємодії, сукупності надаваних послуг. Поки рано говорити про можливість досить повної й точної класифікації ОС, тому що розвиток їх структур і способів функціонування триває, і майже кожна нова розробка ОС вносить нові ідеї, що змушують змінювати прийняту класифікацію. Проте, можна вказати деякі ознаки, що вже встановилися, по яких вдається класифікувати існуючі й проектовані ОС (рис.1.1).

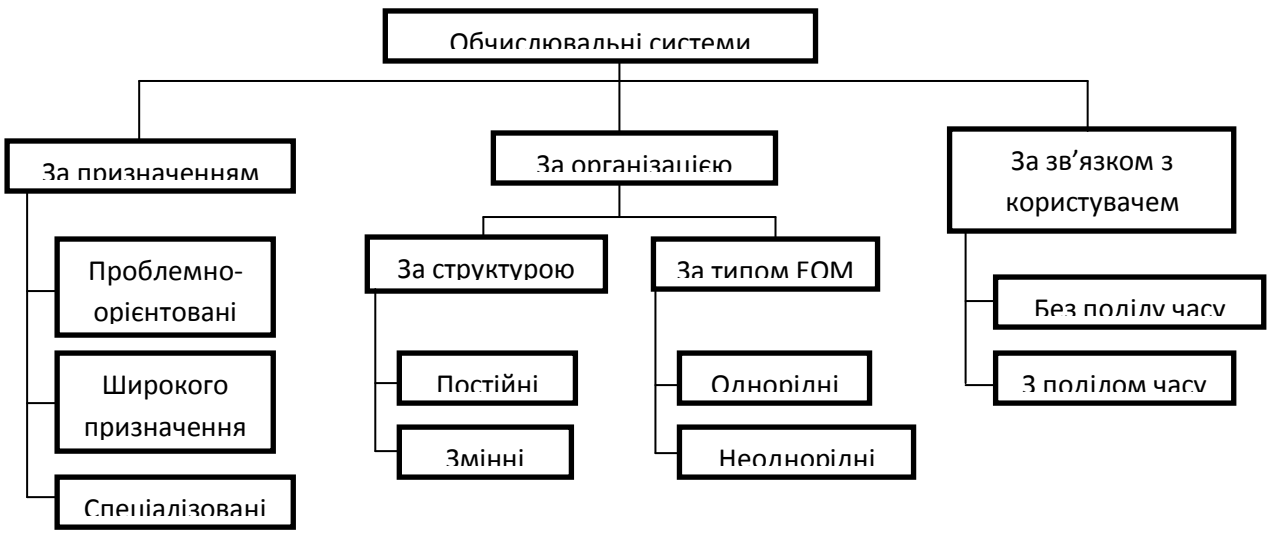


Рис.1.1

У наведеній схемі не знайшли відображення способи організації обчислювального процесу в обчислювальних системах і використання ресурсів ОС при рішенні задач. Перший досвід експлуатації ОС показав, що потенційно великі можливості цих систем можуть бути повністю розкриті й використані тільки при застосуванні ефективних методів і засобів організації їх роботи. Огляд літератури по застосуванню обчислювальної техніки дозволяє узагальнити різні форми експлуатації ОС і представити їх у вигляді наступної класифікаційної схеми (рис. 1.2). При цьому слід зауважити, що в цей час найбільша увага приділяється організації робіт у паралельних обчислювальних системах.

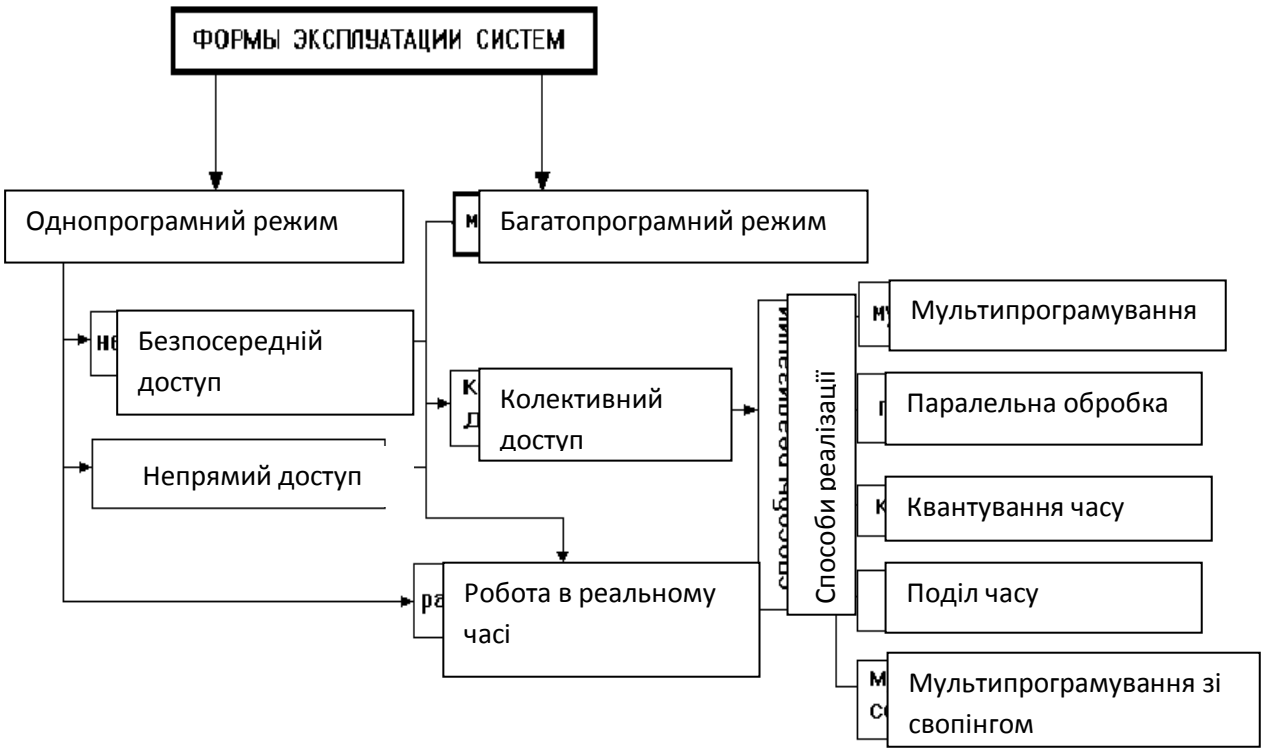


Рис. 1.2

- *Однопрограмний режим роботи*

Як ми вже відзначали, обчислювальна система (машина) працює в однопрограмному режимі, якщо в системі перебуває одна або кілька задач, але всі ресурси віддано одній задачі, і вона не може бути перервана для виконання іншої.

У початковій стадії застосування ЕОМ користувач сам працював за пультом керування, здійснюючи ввід своєї програми, її запуск і спостерігав за виходом результатів по мірі їх одержання.

Час реакції користувача при безпосередньому доступі великий в порівнянні з часом реакції машини, що надзвичайно невигідно з погляду ефективного використання потужності останньої. Дійсно, користувач певний час вводить за допомогою клавіатури інформацію, або обмірковує свою чергову дію, а в цей час ЕОМ простоює. При пакетній обробці в однопрограмному режимі без використання мультипрограмування доступ до ЕОМ здійснюється побічно: за допомогою керуючої програми, що забезпечує перехід від однієї задачі до іншої і контроль над їхнім виконанням. При цьому, значно скорочуються непродуктивні втрати машинного часу при переході від виконання одного завдання до іншого, тому що користувач не бере участь у процесі розв'язку задач й зміні задач, але підвищується ефективність використання ресурсів ЕОМ, і в той же час, повністю виключається можливість діалогу людина - машина.

При такому пакетному режимі виконується послідовна обробка задач. У будь-який момент часу тільки одна із задач пакета виконується й займає всі ресурси, а інші задачі (програми) пакета перебувають у стані очікування своєї черги. Під час виконання однієї із програм пакета забороняється її переривання з метою переходу до іншої програми пакета. Перехід до чергової програми дозволяється або після завершення поточної програми, або її аварійного завершення.

Для внесення найменшої зміни в програму необхідно очікувати вводу в машину чергового пакета задач. Таким чином, при непрямому доступі до

ЕОМ час реакції машини в переважній більшості випадків перевищує час реакції користувача, а це знижує ефективність роботи користувача.

При пакетній обробці задач в однопрограмному режимі функціонування ОС виникає необхідність вибору оптимальної послідовності виконання задач із заданими строками реалізації на одному обслуговуючомупристрої. У випадку, якщо можливості ОС є постійними, а заявки (задачі пакета), які повинні бути обслужені, уже є в наявності, необхідно визначити оптимальну черговість обслуговування заявок або з врахуванням макро характеристик обчислювальної системи, або з урахуванням вимог користувача до показників часу проходження заявок через ОС або часу очікування.

- *Багатопрограмний режим роботи*

Система працює в багатопрограмному режимі, якщо в системі перебуває кілька задач на різній стадії виконання, і кожна з них може бути перервана іншою з наступним відновленням.

Однією із проблем багатопрограмної роботи є організація захисту програм від взаємного впливу, як на рівні оперативної пам'яті, так і на різних рівнях зовнішньої пам'яті. Необхідність організації захисту виникає внаслідок того, що різні програми пишуться незалежно одна від одної, розраховуючи на одноособове використання ресурсів системи. Це створює небезпеку взаємного впливу програм одної на одну й повинне бути враховане при реалізації системи.

Поділ апаратних і програмних ресурсів системи ставить складні задачі керування перед Супервізором (сукупність керуючих програм операційної системи), які він вирішує завдяки наявності спеціальних алгоритмів розподілу ресурсів системи.

Якщо кількість задач перевищує кількість процесорів, то виникає проблема визначення (планування) черговості їх розв'язку в часі. Планування в часі - це розв'язок безлічі завдань на обмежених ресурсах за мінімальний

час. При організації (плануванні) роботи паралельної ОС і розподілі завдань між безліччю процесорів (ресурсів), планування необхідно виконувати як у просторі, так і в часі (рис. 1.3).

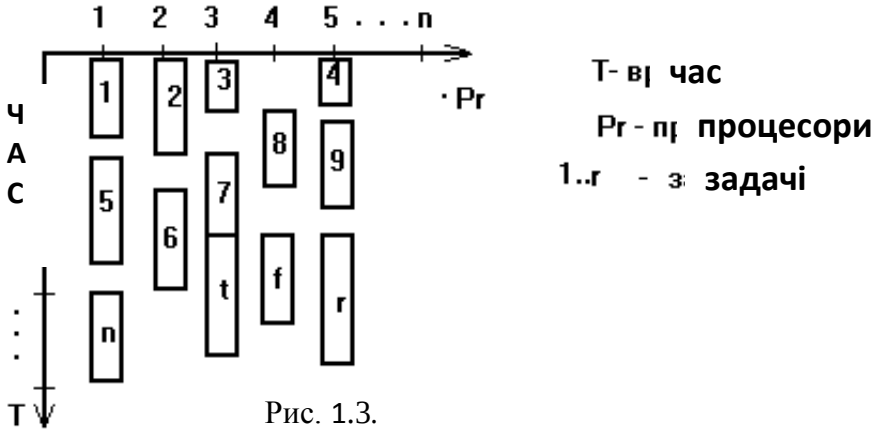


Рис. 1.3.

Як тільки кількість процесорів стає більше, ніж кількість незалежно розв'язуваних задач, потреба в часовій координаті планування пропадає.

Прийнято розрізняти наступні способи реалізації багатопрограмного режиму роботи:

- Класичне мультипрограмування;
- Паралельна обробка;
- Поділ часу;
- Мультипрограмування зі свопінгом.

Класичне мультипрограмування - це така організація обчислювальних процесів в ЕОМ, при якій одночасно виконується дві й більш програм. Ціль мультипрограмування полягає в підвищенні ефективності використання апаратурних засобів ОС за рахунок організації паралельної роботи її пристроїв. Необхідною умовою паралельної роботи пристроїв є їх автономність. Основою мультипрограмування є така організація обчислювальних процесів, при якій забезпечується максимальна продуктивність обчислювальної системи за рахунок паралельної роботи автономно діючих процесора (процесорів) і каналів вводу/виводу.

Режим мультипрограмування характеризується тим, що правила переходу від програми до програми встановлюються з міркувань досягнення

максимального використання ресурсів машини шляхом більш широкого використання сполучення їх дій. При пакетній обробці без мультипрограмування дорогий ЦП все-таки простоює, коли дані вводяться або виводяться, незважаючи на те, що сам процес вводу/виводу увесь час прискорюється. Тоді був запропонований інший розв'язок - помістити в пам’ять одночасно кілька програм. Пам'ять поділена на ділянки, і кожна програма розташовується у своїй ділянці. Тепер, при виконанні вводу/виводу для однієї програми, друга програма може робити обчислення, тобто процесор перемикається на виконання іншої програми (рис. 1.4). Цей підхід, при якім кілька програм ділять пам'ять і по черзі використовують ЦП і інші ресурси, що працюють автономно, називається мультипрограмуванням.

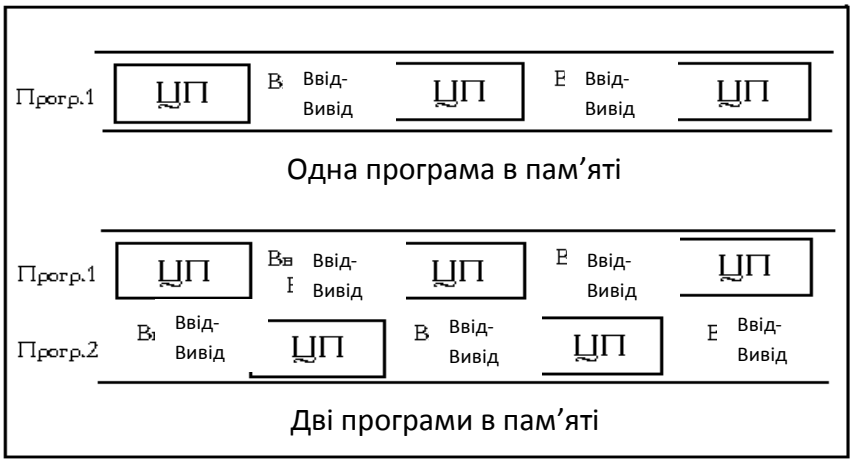


Рис. 1.4. Багато задачний і послідовний підхід

При мультипрограмуванні покращення якості обслуговування користувача не передбачається. Якщо одна із програм захоплює один з ресурсів, наприклад процесор, то виконання інших програм "блокується" і вони будуть перебувати в стані очікування. Крім того, мультипрограмування, у чистому виді, не сумісне з безпосереднім доступом, тому що це один з видів пакетної обробки задач, тобто всі користувачі одержують результати одночасно, як і в режимі непрямого доступу. Проте, загальний час обробки

пакета задач при мультипрограмуванні виявляється менше, ніж при послідовній пакетній обробці, за рахунок кращого використання обладнання.

Продуктивність системи, що працює в мультипрограмному режимі, багато в чому залежить від складу групи програм, що виконуються спільно. Дійсно, майже неминуче виникнення окремих моментів часу, коли всі програми очікують вводу або виводу даних, а центральнийпристрій простоює. Внаслідок цього виникає задача цілеспрямованого формування пакета програм і черговості їх проходження з метою скорочення непродуктивних простоїв обладнання.

На рис. 2.5.б показаний приклад виконання трьох задач у мультипрограмному режимі. При цьому необхідно враховувати пріоритети задач при виборі задачі, якій необхідно дати перевагу у випадку множинних вимог до деяких ресурсів.

При зменшенні абсолютного часу розв'язку, час розв'язку окремих заявок може збільшуватися в порівнянні з однопрограмним режимом роботи. рис. 1.5.а

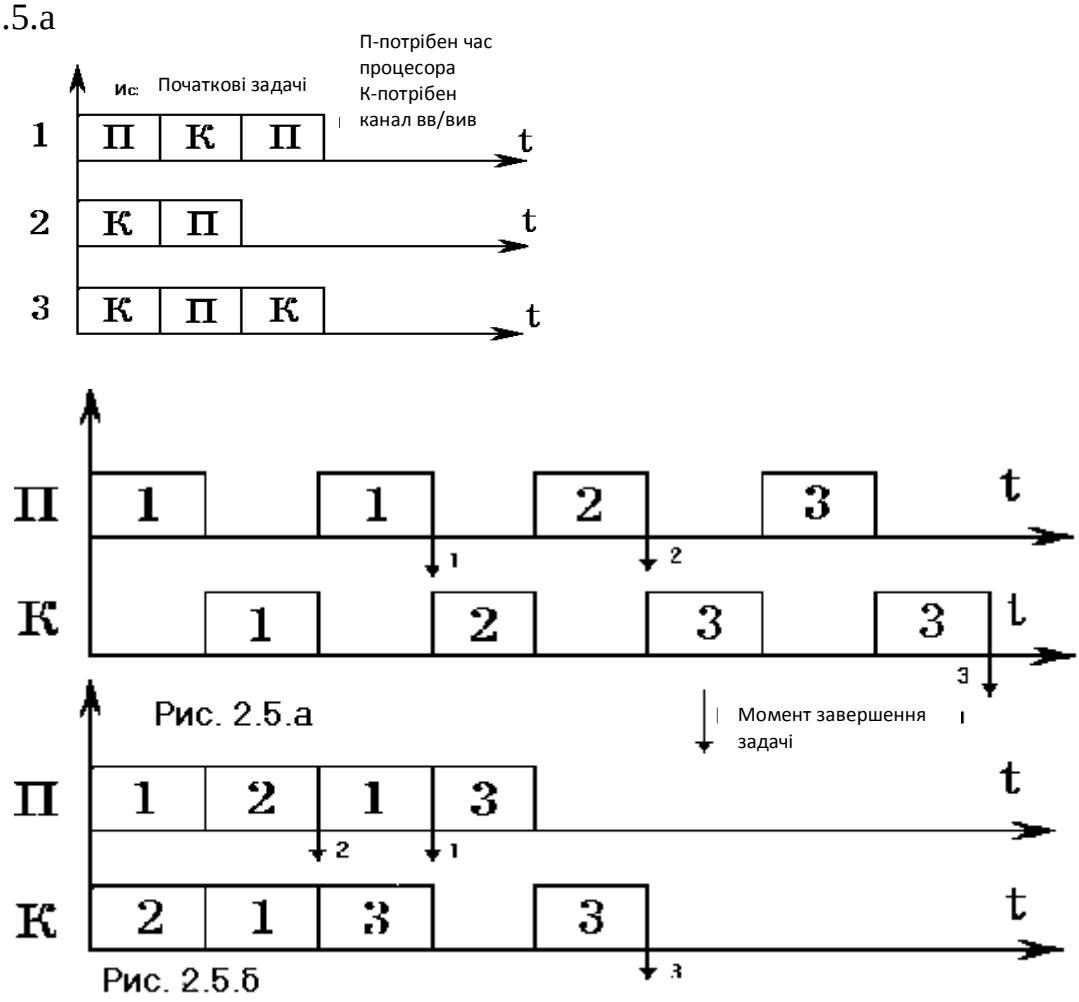


Рис. 2.5.б

Поліпшення якості обслуговування користувачів при збереженні переваг пакетної обробки може бути досягнуте на основі використання колективного (багатоканального) доступу до системи. Організація такого доступу базується на поділі часу роботи системи між кількома користувачами таким чином, що кожному з них періодично виділяються кванти машинного часу.

Найпростішою реалізацією режиму поділу часу є *паралельна обробка* даних. Під *паралельною обробкою* декількох задач розуміється такий багатопрограмний режим, у якім перехід від однієї задачі до іншої відбувається через досить короткі проміжки часу (кванти), щоб створити в користувача ілюзію одночасності виконання декількох програм (режим уявного суміщення).

Основною метою даного режиму є покращення обслуговування користувачів, що виражається в тому, що в останніх створюється враження відсутності черги, тому що розв'язок їх задач не переривається на тривалі відрізки часу. Крім того, якщо користувачеві надаються деякі засоби прямого доступу (хоча б для виводу інформації), то враження одноособової роботи на ЕОМ ще більш підсилюється видачею результатів у міру їх одержання. Паралельна обробка заснована на використанні значних відмінностей часів реакції користувача й машини внаслідок високої швидкості роботи останньої.

Однак, у загальному випадку, час відповіді при цьому режимі не оптимальний, тому що потрібен тонкий механізм врахування пріоритету задач, що враховується при визначенні черговості їх обробки. Найбільше застосування одержали алгоритми, засновані на принципі аналізу заявок, що стоять у черзі. Час виконання пакета задач при паралельній обробці перевищує час їх виконання, якби вони одноосібно використовували ресурси машини. Проте, паралельна обробка задач, при якій користувач може одержувати результати, не очікуючи кінця обслуговування всього пакета, зменшує час очікування відповіді в порівнянні з режимом послідовної обробки.

У більшості випадків паралельна обробка сполучається з мультипрограмуванням, що забезпечує наявність двох типів переривань, що викликають перехід до іншої програми: звичайне переривання під час очікування вводу/виводу (мультипрограмування); змушене переривання наприкінці відрізка часу, що приділяється кожної програмі (паралельна обробка). Саме цей режим дуже часто називають квантуванням часу.

У загальному випадку в сучасних системах під режимом *поділу часу* розуміється найбільш розвинена форма багатопрограмної роботи, що сполучає мультипрограмування із квантуванням і забезпеченням безпосереднього доступу деяких привілейованих користувачів до ресурсів системи. Безпосередній доступ до ОС, як правило, визначає необхідність роботи ОС у реальному часі. При цьому час реакції системи на зовнішні події повинен задовольняти певним вимогам. Час відповіді в такій системі близький до того, яке було при одноособовім використанні машини.

Як і при паралельній роботі, задачі виконуються по черзі протягом деякого відрізка часу (кванта), по завершенні якого відбувається змушене переривання. Тривалість цих відрізків часу (квантів) не є, як правило, фіксованої, а змінюється залежно від розглянутої задачі, а також від конкретних умов експлуатації в момент передачі керування від одного користувача іншому. Вибір користувача, задачі якого буде передане керування, і виділення цій задачі кванта часу здійснює керуюча програма, що є складовою частиною програм - диспетчерів, призначених для роботи з поділом часу. Диспетчер функціонує відповідно до обраної для даної системи дисципліни обслуговування заявок, які будуть розглянуті нижче.

Існує ще один спосіб реалізації багатопрограмного режиму роботи на одно процесорній машині - "мультипрограмування зі свопінгом". Він використовується при нестачі пам'яті, для зберігання всіх паралельно виконуваних програм.

Свопінг характеризується тим, що після певного проміжку часу (макрокванта) програми користувачів, що перебувають в оперативній пам'яті,

листуються на зовнішній носій інформації (диск) у спеціально виділені для цього своп - файли, у такий спосіб зберігається стан цих завдань на момент переривання. Наступна сукупність паралельно виконуваних програм займає місце, що звільнилося в оперативній пам'яті, і система виконує їх у режимі поділу часу, до завершення наступного макрокванта.

- *Форми взаємодії людини з машиною*

Враховуючи вищесказане, можна виділити наступні форми взаємодії користувача з машиною (рис. 1.6). :

1) Безпосередній доступ (БД)
2) Непрямий доступ (НД)
3) Колективний доступ (КД)

Безпосередній доступ (БД) має на увазі, що всі ресурси обчислювальної системи перебувають у повному розпорядженні єдиного користувача протягом усього періоду роботи з нею. При цьому користувач, працюючи за пультом керування машини, вводив програму, переводив машину в режим безперервного рахунку або в режим потактового виконання, спостерігаючи за ходом розв'язку задачі, користуючись індикацією даних на пульті керування, вводив зміни в програму, виводив результати рахунку на друк й т.п. Таким чином, від мистецтва оператора залежала ефективність використання машинного часу і якість розв'язку задачі.

Непрямий доступ (НД) - режим послідовного обслуговування завдань різних користувачів, сформованих у вигляді єдиного пакета, на основі керуючої програми, яка з врахуванням заздалегідь визначених вимог користувачів, здійснює автоматичне перемикання ресурсів ОС. Тобто користувач взаємодіє з обчислювальною системою побічно, за допомогою керуючої програми, для якої користувач повинен підготувати інформацію, що визначає її дії для виконання його роботи. При цьому використовується режим пакетної обробки завдань. У функції керуючої програми входить

керування проходженням задач пакета через обчислювальну машину й контроль над їх виконанням. При цьому ефективність праці програміста суттєво зменшується, тому що практично виключається діалоговий режим роботи людини з машиною. Таким чином, режим безпосереднього доступу дозволяє збільшити ефективність праці програміста, а режим пакетної обробки збільшує ефективність використання ресурсів обчислювальної машини.

Колективний доступ (КД) має на увазі можливість розподілу ресурсів (обладнання, часу, програм, даних) або їх поділ між різними користувачами. Для такого режиму доступу характерна мультипрограмна організація обчислень, при якій допускається виконання однієї програми й ввід-вивід для декількох програм. При такій організації роботи обчислювальної системи, значні перерви в роботі якої-небудь програми, під час яких користувач, може використовувати обладнання вводу/виводу або обмірковувати подальші кроки, не є настільки катастрофічними з погляду ефективності використання ресурсів машини, у порівнянні з режимом безпосереднього доступу. У цей час машина може бути завантажена обробкою програм інших користувачів. Очевидно, що при обмеженому числі користувачів, простої ОС неминучі. Ці простої можна зменшити або повністю ліквідувати за допомогою так званих фонових задач, тобто задач, введених у систему заздалегідь, на відміну від задач, розв'язуваних із зовнішнього термінала, що називаються основні. Фонові задачі, володіючи меншим пріоритетом у порівнянні з основними, включаються в обробку при недовантаженні системи основними задачами.

У теперішній час форми взаємодії користувача з ОС на основі використання ідей колективного доступу значно розширилися за рахунок використання інтелектуальних терміналів - персональних ЕОМ. Використання персонального комп'ютера повернуло нас до режиму безпосереднього доступу (БД1). Розвиток мережних технологій і відповідного програмного забезпечення привів до нової форми непрямого доступу (НД1). Мережні функції операційних систем дозволяють здійснити

непрямий доступ до ресурсів мережі. Однією з форм такого доступу є застосування технології клієнт-сервер. Користувачі побічно з використанням засобів мережних операційних систем мають доступ до тих самих ресурсів, де обробка запитів користувачів проводиться відповідно до прийнятих алгоритмів диспетчеризації. Комбінація обчислювальних можливостей персональної ЕОМ і використання, якщо буде потреба, усієї обчислювальної потужності обчислювального середовища з використанням засобів Internet, до якої цей термінал підключений, визначає практично необмежені можливості для виконання задач користувача. Таку форму взаємодії можна визначити як безпосередній доступ на новому якісному рівні, тобто як прямий доступ до ресурсів персонального комп'ютера з використанням, якщо буде потреба, як непрямого доступу, так і колективного ресурсів локальної або глобальної мережі. Безсумнівно, ця форма взаємодії людини з машиною (або краще з обчислювальним середовищем) вимагає зовсім нових принципів організації обчислювальної системи й організації обчислювальних процесів у ній.

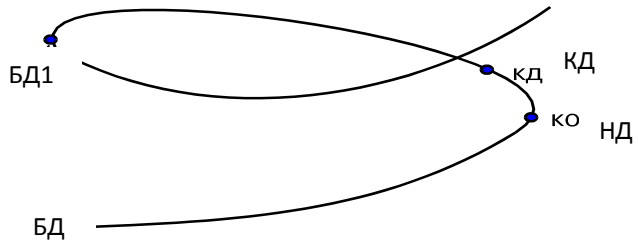


Рис. 1.6. Еволюційний розвиток форми взаємодії людина - машина

Для досягнення більшої продуктивності ОС у систему включають кілька незалежних процесорів, взаємодіючих через оперативну пам'ять, спеціальний або комутуючий канал зв'язку. Подібна організація ОС привела до мультипроцесорних обчислювальних систем або розподілених обчислювальних систем, здатних паралельно виконувати кілька незалежних програм або кілька незалежних областей однієї програми. Властивості мультипроцесорних обчислювальних систем відображають наступні дві основні тенденції сучасного розвитку обчислювальної техніки:

- Модульний принцип побудови технічного забезпечення, при якім кожнийпристрій виконується у вигляді незалежного модуля, що дозволяє: по-перше, варіювати можливостями системи як у кількісному, так і в якісному відношенні, по-друге, зменшити уразливість системи до відмов у силу взаємозамінності однотипних модулів.
- Паралельне виконання. Розпаралелювання програм (виділення паралельних ділянок) і паралельне виконання незалежних галузей однієї програми або виконання повністю незалежних програм, що дозволяє зменшити загальний час реалізації задач і підвищити ефективність завантаження обладнання.

Багатопроцесорна архітектура ОС містить у собі два й більш ЦП, що спільно використовують загальну пам'ять і периферійні пристрої (рис. 2.7), що збільшує можливості для підвищення продуктивності системи, на основі одночасного виконання задач на різних ЦП. Кожний ЦП функціонує незалежно від інших, але всі вони працюють із тим самим ядром операційної системи. Поведінка процесів у такій системі нічим не відрізняється від їхньої поведінки в однопроцесорній системі.

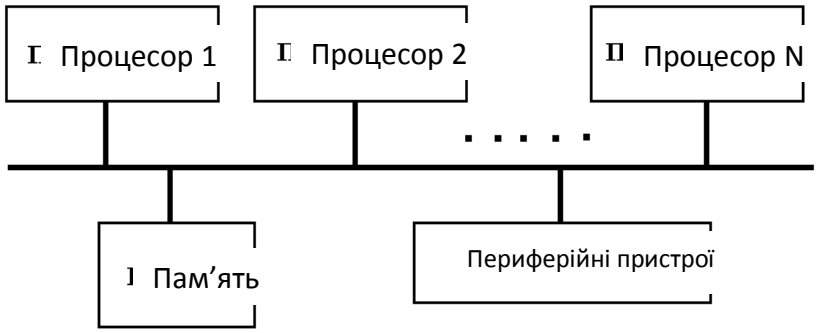


Рис.1.7. Багатопроцесорна конфігурація

Паралельна робота декількох процесорів у режимі обслуговування системних викликів (режим супервізора), пов'язана з виконанням різних процесів створює ряд проблем, по збереженню цілісності даних. Для дозволу даних проблем використовуються відповідні механізми захисту.

Якщо питання про порушення цілісності залишити відкритим — як би рідко подібні порушення не траплялися, ядро стане вразливим і його поведінка стане непередбаченою. Уникнути цього можна трьома способами:

- виконувати всі критичні операції на одному процесорі, спираючись на стандартні методи збереження цілісності даних використовуваних в одно процесорних системах;
- регламентувати доступ до критичних ділянок програми, використовуючи елементи блокування ресурсів;
- усунути конкуренцію за використання даних шляхом відповідного перетворення алгоритмів.

Головний процесор відповідає за обробку всіх звертань до операційної системи й усіх переривань. Підлеглі процесори реалізують процеси в режимі завдання й інформують головний процесор про всі звертання до системних функцій.

Мультипроцесорна система, на відміну від мультипрограмної, дозволяє зменшити час розв'язку кожної окремої задачі за рахунок можливості одночасного, паралельного виконання незалежних ділянок її програми. При цьому підвищується коефіцієнт використання обладнання, якщо вОС одночасно в обробці перебуває велика кількість програм.

Розподілені операційні системи призначені для керування й організації обчислень в ОС, що представляє собою сукупність обчислювальних засобів і периферійних пристроїв, об'єднаних між собою каналами зв'язку із забезпеченням можливості взаємного обміну інформацією, яка з погляду користувача являє собою єдине ціле. Більш докладно властивості й особливості розподілених ОС будуть розглянуто в розділі 3.

Властивості розподілених операційних систем:

- Організація обчислень у ОС із індивідуальною пам'яттю окремих обчислювальних вузлів. Це призводить до того, що не можна визначити загальний стан системи за допомогою безлічі спільних змінних. Неможливість спільного звертання до пам'яті й відмінності в затримці

передач повідомлень призводять до того, що при визначенні стану якого-небудь елемента системи із двох різних точок можна одержати різні результати, залежно від порядку попередніх подій;

- Розподілена операційна система планує й розподіляє виконувані роботи між вузлами системи, виходячи з міркувань підвищення пропускної здатності всієї системи;
- У розподіленій системі можлива організація паралельних обчислень.

Два фактори збільшують імовірність відмов окремих елементів (у порівнянні із централізованими системами):

1. Велике число незалежно працюючих елементів системи.
2. Надійність систем зв'язку звичайно менше, ніж надійність обчислювального вузла.

Проте надійність усієї обчислювальної системи в розподілених системах у цілому досить висока за рахунок специфічних властивостей, що дозволяють операційній системі виключити або послабити ефект фізичної відмови окремого елемента системи за рахунок локалізації елементів, що вийшли з ладу, реконфігурація системи й перерозподілу задач.

Архітектура розподіленої системи зовні збігається з архітектурою багатомашинного комплексу й представлена на рис. 2.8. Відмінність між ними полягає в зовсім різних способах організації обчислювального процесу. Кожний комп'ютер, показаний на малюнку, є автономним модулем (обчислювальним вузлом), що складається з ЦП, пам'яті й периферійних пристроїв. Відповідність моделі не порушується, навіть, незважаючи на те, що комп'ютер має в своєму розпорядженні тільки локальну файлову систему. Кожний комп'ютер має периферійні пристрої для зв'язку з іншими машинами, щоб якщо буде потреба, мати доступ до всіх необхідних йому файлам, які можуть розташовуватися на іншому комп'ютері. Фізична пам'ять, доступна кожній машині, не доступна задачам, виконуваним на інших машинах. Цією особливістю розподілені системи відрізняються від сильно зв'язаних багатопроцесорних систем. Відповідно, і ядро операційної системи

на кожній машині функціонує незалежно від зовнішніх умов експлуатації розподіленого середовища й роботи інших машин.

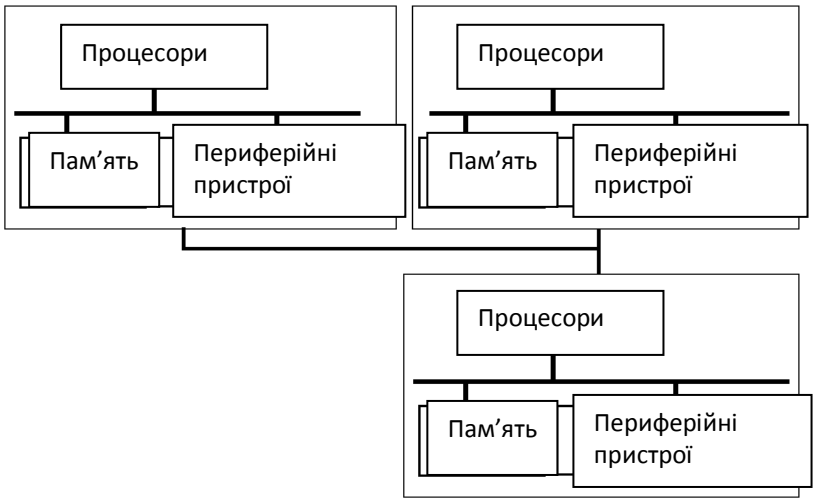


Рис. 1.8. Модель системи з розподіленою архітектурою

Розподілені системи традиційно діляться на наступні категорії:

- *периферійні системи*, що представляють собою групи машин, що відрізняються яскраво вираженою спільністю характеристик їх ресурсів і пов'язаних з однією (зазвичай більшої) машиною. Периферійні процесори ділять своє навантаження із центральним процесором і переадресовують йому всі звертання до операційної системи. Ціль периферійної системи полягає в збільшенні загальної продуктивності мережі й у наданні можливості виділення процесора одному процесу в операційній системі. Система запускається як окремий модуль; на відміну від інших моделей розподілених систем, периферійні системи не мають реальну автономію, за винятком випадків, пов'язаних з диспетчеризацією процесів і розподілом локальної пам'яті.

- *розподілені системи* типу "Newcastle", що дозволяють здійснювати дистанційний зв'язок по іменам віддалених файлів у бібліотеці.Віддалені файли мають специфікацію (складене ім'я), яка у зазначенні шляху пошуку містить спеціальні символи або додатковий компонент імені, що передує кореню файлової системи. Реалізація цього методу не припускає внесення

змін у ядро системи, внаслідок цього він більш простий, ніж інші методи, але менш гнучкий.

- *абсолютно "прозорі" розподілені системи*, у яких для звертання до файлів, розташованих на інших машинах, досить зазначення їх стандартних складених імен; розпізнавання цих файлів як віддалених входить в обов'язки ядра. Маршрути пошуку файлів, зазначені в їхніх складених іменах, перетинають машинні границі в точках монтування, скільки б таких точок не було сформовано при монтуванні файлових систем на дисках.

Віртуальна операційна система характеризується забезпеченням багатопрограмного режиму роботи, коли кожний користувач може працювати у власнім операційнім середовищі.

1.5. ПОНЯТТЯ ГЕНЕРАЦІЇ, ІНСТАЛЯЦІЇ, ІНІЦІАЛІЗАЦІЇ

При покупці обчислювальної системи набувається й програмне забезпечення для виконання тих функцій, заради яких ця ОС і купується. Як правило, у це ПЗ включені всі необхідні програми для повної підтримки всіх відповідних апаратурних складових і необхідних режимів роботи придбаної ОС. Таким чином, купується вихідний варіант операційної системи або *дистрибутив.* У нього входять різні варіанти драйверів для апаратних компонентів, а також програми для забезпечення різних режимів роботи ОС. Це драйвери Cd-Roma, мережевих Ethernet карт, SCSI-пристроїв, Sound Blastera, принтера і т.д. Причому до складу дистрибутива можуть входити варіанти драйверів для підтримки різних локальних шин (VESA, PCI) і апаратних платформ. Крім цього, дистрибутив має всі необхідні програми, що підтримують роботу обчислювальної системи в різних режимах роботи, наприклад, однопрограмному або багатопрограмному, багатопроцесорному, мережному й підтримки різних сервісів (файловий, поштовий, віддаленого термінала й т.п.).

У реальних умовах експлуатації, для нормального функціонування обчислювальної системи, уся сукупність програм вихідного ПО не потрібна. Із усієї їх кількості потрібно вибрати якусь підмножину програм, що забезпечують роботу обчислювальної установки у встановлених користувачем режимах і на наявнім обладнанні. Процес відбору з дистрибутива тих програмних модулів, які будуть використовуватися, називається *генерацією* операційної системи. Процес генерації виконується або з використанням спеціальної "мови генерації", або за допомогою організації діалогу користувача із системою, у результаті якого визначаються бажання й вимоги користувача до функціонування системи, а система відповідно до цього визначає сукупність необхідних програм, що підтримують ці вимоги.

Місце, де перебуває резиденція системи, називається "резидентним" томом, а сама система "резиденцією". Із усіх програмних модулів у резидентному томі, частина використовується в міру необхідності, а деякі з них, що забезпечують базове функціонування ОС становлять *резидентні* програми, що постійно перебувають у системній області оперативної пам'яті. Сукупність програм, що забезпечують функціонування ОС, що й перебувають у системній області оперативної пам'яті, становить ядро супервізора.

Деякі програми, пов'язані з виконанням функцій ОС, але, які не перебувають постійно в оперативній пам'яті називаються *транзитними*. Ці програми викликаються впам’ять в міру необхідності. Крім цього, система дозволяє користувачеві по своєму бажанню визначити деякі програми як резидентні. Усі інші програми є транзитами, викликаються динамічно й можуть займати те саме місце в оперативній пам'яті й затирати одна одну.

При роботі обчислювальної системи користувач може міняти свої уявлення про режими її функціонування. Крім цього може мінятися й склад обладнання. Процес локальної настройки ОС за бажанням користувача називається *інсталяцією*. Ознакою, по якій можна відрізнити інсталяцію від

генерації є те, що, якщо при настройці операційної системи не використовується дистрибутив або якщо він використовується, то тільки для локального додавання деяких функцій - то це інсталяція. Якщо для зміни або додавання функцій потрібен дистрибутив і виконується повна пере установка ОС - генерація.

При вмиканні обчислювальної системи виконується завантаження ОС. Під час завантаження виконується зв'язування й розміщення всіх частин, що входять у ядро супервізора в пам'яті ОС, тобто розміщення резидентних програм на своїх місцях, формування спеціальних системних структур даних в області даних операційної системи, формування постійної області, активізацію задач для початку роботи операційної системи. Цей процес називається *ініціалізацією* операційної системи й виконується у два етапи, спочатку виконується ініціалізація ядра, а потім ініціалізація системи.

1.6 КОНЦЕПЦІЯ РЕСУРСІВ У СУЧАСНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМАХ

Під **ресурсом** абстрактно можна розуміти "щось" (реально існуюче), яке може знадобитися для виконання програми (задачі, процесу). Усі ресурси можна розділити умовно на дві категорії:

фізичні пристрої, що входять до складу сучасної обчислювальної системи;

внутрішні ресурси, необхідні для виконання користувацьких програм (дані, підпрограми і т.д.).

У якості фізичних пристроїв можемо розглядати:

* оперативну пам'ять, де зберігаються: дані, програми, різні системні об'єкти для виконання функцій керування, програми обробки переривань, інформація про наявність і характеристики зовнішніх пристроїв, таблиці оверлеєв, системні буфери, блоки керування активізованих задач (ТСВ) і т.д.;

* процесор(и), який виконує роботу, обумовлену прикладною програмою (виконує обробку даних);

* клавіатура як пристрійвводу;

* принтер як пристрій друку (виводу);
* термінал як пристрійвиводу інформації;
* магнітні, жорсткі або гнучкі диски як пристроївводу/виводу.

Найчастіше при описі ресурсів обчислювальної системи розглядають процесор і оперативну пам'ять (ОП). ОП виділяється користувачеві для запису даних, програм, розміщення блоків керування задачами.

Якщо ОП досить для того, щоб розмістити всі необхідні програми, дані й допоміжні таблиці, то особливих ускладнень у розміщенні й використанні простору пам'яті немає.

Інакша справа із процесором. Цей ресурс із погляду системи керування обчислювальним процесом є найскладнішим для керування, планування й диспетчеризації його роботою.

Під **виділенням процесора** в однопроцесорній системі розуміється виділення його часу для обробки кожної із задач, що перебувають у системі. Під **виділенням процесора** в багатопроцесорній обчислювальній системі розуміється не тільки виділення його часу, але й власне визначення адреси процесора або обчислювального вузла, на який призначається активізованазадача.

Найбільші труднощі для ОС представляє планування задач для виконання на процесорі й відстеження станів взаємодіючих задач. Для цього існує безліч дисциплін керування чергами, відповідно до яких з наявних у системі й готових до виконання задач планувальник визначає черговість виділення їм часу процесора.

Найбільш пріоритетний процес із погляду планувальника й одержує у своє розпорядження час процесора (квант) для виконання. При розробці планувальників ОС прагнуть використовувати такі правила обслуговування черг задач, що претендують на захоплення процесора, щоб досягти найкращої ефективності роботи всієї обчислювальної системи.

До внутрішніх ресурсів системи можна також віднести дані (константи, змінні), підпрограми або навіть цілі програми (бібліотеки програм, використовувані користувацькими програмами в процесі виконання).

У багатокористувацьких або багатопроцесорних системах до деяких програм часто звертаються або вони можуть використовуватися декількома задачами (процесами) одночасно. Особливо це стосується програм, що реалізують вимоги користувацьких програм до виконання системних викликів. Для підвищення ефективності роботи системи ці програм включаються в ядро супервізора.

Ресурси можуть знадобитися відразу декільком задачам, однак, у будь-який момент часу він може бути виділений тільки одній з них. У такому випадку інші задачі вишиковуються в чергу, або сортуються по пріоритету для виділення їм необхідного ресурсу. Коли задача звільняє ресурс, то він може бути виділений першійзадачі із черги задач, що очікують, а якщо такоїнема, то ресурс вважається вільним.

Задача узгодження роботи декількох задач при використанні загального ресурсу називається*задачею взаємного виключення*, при цьому першазадача із черги очікування до загального ресурсу одержує його, а іншим доступ до ресурсу заборонений.

1.7 ЗАВДАННЯ, ПУНКТ, ЗАДАЧА, ПРОГРАМА І ЇХ ВИКОНАННЯ

Будь-яка робота обчислювальної системи ототожнюється з виконанням потоку завдань, який розуміється як послідовний ряд завдань. Таким чином, **завдання** — це одна з одиниць роботи, виконуваної обчислювальною системою, зміст якої полягає у виконанні обчислень по одній або декільком зв'язаним програмам для деякого застосування. Прикладом завдання може служити вимога здійснення трансляції, редагування й виконання модуля. Завдання можуть бути незалежні або залежні з умовами порядку виконання або умовами передування. Якщо завдання незалежні, то можуть

виконуватися одночасно, що особливо важливо при застосуванні багатопрограмних режимів роботи ОС. А якщо ні, то при виконанні завдань планувальник повинен ураховувати умови передування завдань.

Кожне завдання повинне бути описане мовою керування завданнями. Ввід завдання здійснюється через одне з пристроїв системного вводу. Послідовність завдань, що надходить у систему, називається вхідним потоком завдань.

Будь-яке завдання в загальному випадку може складатися з декількох логічних частин, називаних пунктами, або кроками завдання, які виконуються послідовно. Так для наведеного вище прикладу в якості пунктів завдання можна виділити наступні: пункт, що виконує трансляцію вихідного модуля; пункт, що виконує редагування, отриманого після трансляції об'єктного модуля; пункт, що здійснює виконання отриманого після редагування завантажувального модуля. Дані пункти повинні виконуватися строго послідовно, тому що кожний черговий з них використовує дані, отримані в результаті виконання попереднього пункту.

Як тільки система розпізнає крок завдання й визначить наявність необхідних ресурсів для його виконання, буде сформованазадача. Отже, при безпосередньому виконанні пункту завдання одиницею роботи є **задача.** Результатом такої роботи є виконання однієї або безлічі програм. Таким чином, у загальному випадку кожний пункт завдання викликає виконання деякої множини програм, ім'я яких вказується при описі пункту завдання мовою керування завданнями. Такяк яка програма буде виконуватися, визначається завданням, а потім задачею, то можна сказати, що програма підлегла задачі. Крім того, програма, як правило, перебуває на одному із зовнішніх носіїв інформації й може знадобитися декільком задачам одночасно. У цьому випадку програму можна віднести до категорії ресурсів ОС. Програма може бути або єдиною програмою в рамках пункту завдання, або може викликати інші програми в процесі свого виконання. Обчислювальна система, зафіксувавши задачу й виділивши їй ресурси,

зобов'язана її виконати. При цьому, у сучасних ОС вона виконується в одному з багатопрограмних режимів роботи й передбачає поділ ресурсів між декількома задачами. Для пояснення сутності динаміки процесів, що протікають в обчислювальній системі при проходженні безлічі завдань через ОС від входу завдань, до одержання результатів найчастіше використовується поняття **процесу**.

1.8. ПРОЦЕС — ОСНОВА ФУНКЦІОНУВАННЯ БАГАТОПРОГРАМНИХ ОС

Для користувачів знати, що відбувається в системі при виконанні програми, не обов'язково - наочно показує результат. Однак, для розроблювачів систем необхідно знати, що відбувається при виконанні тієї або іншої операції, які дії можуть привести до відмови системи, як уникнути проблем при зіткненні інтересів різних задач, що претендують на захоплення ресурсів і т.д.. Поняття процесу, опис динаміки його розвитку в часі й у просторі в обчислювальних системах допомагають створити деяку модель виконання програми й виявити критичні ділянки, що впливають на ефективність усієї системи в цілому.

Стан машини визначається станом процесора (вмістом регістрів, що адресуються і внутрішніх регістрів) і станом пам'яті (вмістом комірок пам'яті). Цей стан змінюється при виконанні процесором деяких операцій. Виконання операції - ця деяка дія, результатом якої є перехід за кінцевий час із поточного стану машини (до виконання операції) у наступне (після її виконання). У загальному випадку такою операцією може бути й вся програма. Однак, на такому рівні ми не можемо ввести поняття процесу. Тоді розглянемо кожну таку операцію як неподільну, тобто виконувану процесором за один крок (командний цикл, квант).

Із цього погляду виконання програми можна розглядати як безліч операцій. Будемо відслідковувати виконання програми протягом часу T (від

початку до кінця виконання). За цей час процесор встигне виконати N операцій (N як завгодно велике), які будемо ідентифікувати по номеру (1,..,N).

У таких елементарних операціях можна виділити два моменти: початок операції —П і кінець операції —К. У дані моменти, часів $t_п$ і $t_к$, будемо відзначати стан машини.

При виконанні програми маємо послідовність операцій 1,2,..,N, причому $t_п(1)<t_к(1)=<t_п(2)<...<t_к(N)$ (хоча кінець попередньої операції й початок наступної— це в принципі те саме). Для того, щоб програма могла бути запущена на виконання, операційна система спочатку, повинна створити *оточення* або *середовищевиконання*задачі, куди відносяться ресурси пам'яті, можливість доступу до пристроївводу/виводу й різних системних ресурсів, включаючи послуги ядра. Операційна система маніпулює *образом процесу,* який являє собою програмний код, і розділами даних процесу, що визначають середовище виконання. Події в моменти часу $t_п(1),t_п(2),...t_к(N)$ утворюють *часовий слід* (трасу або історію) виконання задачі або процесу. Для уточнення відмінності понять задача й процес слід зазначити, що задача, утворена в результаті обробки вхідного завдання фіксується в системі за допомогою спеціального системного об'єкта — блоку керування задачею (ТСВ). Відображення задачі в системі за допомогою ТСВ практично однаково, а опис процесу, який динамічно визначає стан виконуваної роботи в будь-який момент часу у взаємодії з апаратурою ОС і з іншими процесами весь час змінюється. При кожній активізації процесу вОС його траєкторія в просторі ресурсів ОС і в часі буде різнитися.

Виходячи з вищеописаного, можна виділити наступні, найбільш уживані в літературі й прийнятні визначення процесу.

Процес **:**

* це траєкторія процесора в адресному просторі обчислювальній системі при виконанні програми;

* сукупність даних ядра системи, необхідних для опису образа програми в пам'яті й керування її виконанням;

* це будь-яка програма, що перебуває в стадії виконання вОС;

* Програма, що ідентифікується унікальним чином, яка потребує одержання доступу до ресурсів комп'ютера;

* це динамічний об'єкт системи, якому вона виділяє ресурси;

* динамічно мінливе середовище виконання задачі.

* абстракція операційної системи, що представляє все, що необхідно для виконання однієї програми в певний момент часу.

* одиниця активності, яку можна охарактеризувати єдиним ланцюжком послідовних дій, поточним станом і пов'язаним з нею набором системних ресурсів.

Перше визначення процесу виділяє дві основні складові для опису процесу. Це послідовність виконуваних команд процесора й набір адрес пам'яті (*адресний простір*), у якім розташовуються ці команди й дані для них.

Виділення цих частин виправдане ще й тим, що в рамках одного адресного простору може бути декілька паралельно виконуваних послідовностей команд, що спільно використовують дані. Такий поділ послідовності команд і адресного простору й привів до поняття *потоку*.

1.8.1. Класифікація процесів

По способу створення процеси діляться на:

- Системні процеси. Вони створюються при завантаженні системи. Програми, виконувані при їхній активізації (наданні часу процесора), мають оверлейну або динамічно - послідовну структуру, є частиною ядра й, або резидентно розташовані в оперативній пам'яті, або викликаються динамічно в транзитну область і запускаються особливим образом при ініціалізації ядра системи. Програми й дані цих процесів недоступні для несистемних процесів. Системними процесами є: диспетчер свопінга, диспетчер

сторінкового заміщення, диспетчер буферного кеша, диспетчер пам'яті ядра і т.д.

- Прикладні (проблемні, користувацькі) процеси. Створюються при активізації роботи (задачі користувача) операційною системою або активізації користувацького сеансу роботи. Під активізацією процесу тут розуміється початок виконання програми, підлеглої цьому процесу, тобто початок процесу. Користувацькі процеси можуть виконуватися як в інтерактивному так і фоновому режимі. Час життя процесу обмежений сеансом роботи користувача.

У сучасних багатопрограмних операційних системах виділений ще один тип процесів — демони. Демони — це не інтерактивні процеси, які запускаються звичайним чином — шляхом завантаження впам’ять відповідних їм програм і виконуються у фоновому режимі. Звичайно демони запускаються при ініціалізації системи й забезпечують роботу різних підсистем ОС: системи термінального доступу, системи друку, системи мережного доступу й т.п. Демони не зв'язані з жодним користувацьким сеансом роботи й не можуть управлятися користувачем. Вони очікують поки той або інший процес запросить певну послугу, наприклад доступ до файлової системи або друку документа.

Користувацький процес може бути активізований тільки іншим процесом, для цього й існують системні процеси, що активізуються при запуску ОС. Процес, який активізував інший процес, вважається батьківським, а створений породженим або дочірнім (або процес нащадок). У свою чергу активізований користувацький процес може створити один або кілька породжених процесів, для яких він стане батьківським. Таким чином, вОС створюється ієрархія процесів, що перебувають в «родинних відносинах».

По способу розвитку процеси діляться на:

* Послідовні процеси. Це процеси, які виконуються один за одним, тобто коли закінчиться виконання першого процесу, починається другий і т.д.

* Паралельні. Процеси, які можуть виконуватися одночасно. *Паралельні процеси* можуть бути:

Незалежні, тобто виконують різні операції, займають різні сегменти пам'яті й не мають загальних частин.

Асинхронні - це процеси, які при виконанні синхронізуються й взаємодіють один з одним. Взаємодія - це передача даних між процесами. Також процеси можуть мати спільні частини - процедури, дані, необхідні (але ще не виділені) ресурси. У цьому випадку операційна система, що відслідковує стани процесів повинна вирішувати задачу взаємного виключення й задачі пов'язані з можливим потраплянням у тупикову ситуацію (дедлок).

Маючи можливість створювати кілька паралельно працюючих процесів, операційна система може одночасно обслуговувати кілька користувачів і виконувати кілька задач. Здатність процесу створювати нові процеси дає наступні переваги:

1. Будь-який користувач може створювати багато задачні додатки.
2. Тому що породжений процес виконується у власному віртуальному адресному просторі, то успіх або невдача при його виконанні на батьківський процес не впливає.
3. Процеси створюють породжені процеси, які виконують нові програми. Це дозволяє користувачам писати програми, які можуть шляхом виклику інших програм розширювати свої функціональні можливості.

При активізації операційної системи першими повинні бути активізовані наступні процеси: адміністратора пам'яті - це процес, який управляє ефективним розподілом пам'яті для розміщення різних програм, процес для роботи із зовнішніми пристроями, керування даними (керування файловою системою).

Окремо виділяється таймерний процес, підлеглий годиннику, який активізується на апаратному рівні.

1.8.2. Стани процесів

Під час свого існування процес може перебувати в наступних станах: ***підготовлений (новий)***, ***готовий (здійснимий), активний (виконуваний), заблокований або очікуючий*** (рис.1.10). У кожний конкретний момент часу процес перебуває в одному із цих станів, який фіксується в блоці керування процесом (PCB).

Підготовлений (новий) процес це тільки що створений процес, який ще не поміщений операційною системою в чергу здійсненних процесів. Однак операційна система може виконати попередню роботу із присвоєння йому ідентифікатора й сформувати всі необхідні для керування процесом таблиці. Новий процес ще не завантажений в оперативну пам'ять. Тобто програма, підлегла даному процесу не завантажена впам’ять, а даним, що відносяться до цього процесу, не виділений адресний простір.

Процес перебуває в **готовому** стані або в стані готовому для виконання (активізований), якщо йому вже виділені ресурси (програма, необхідна для виконання, перебуває в оперативній пам'яті), однак йому не виділений час процесора для виконання (процесор зайнятий іншим процесом).

Процес, якому виділяється час процесора, переводитися в **активний** стан або стан фізичного виконання.

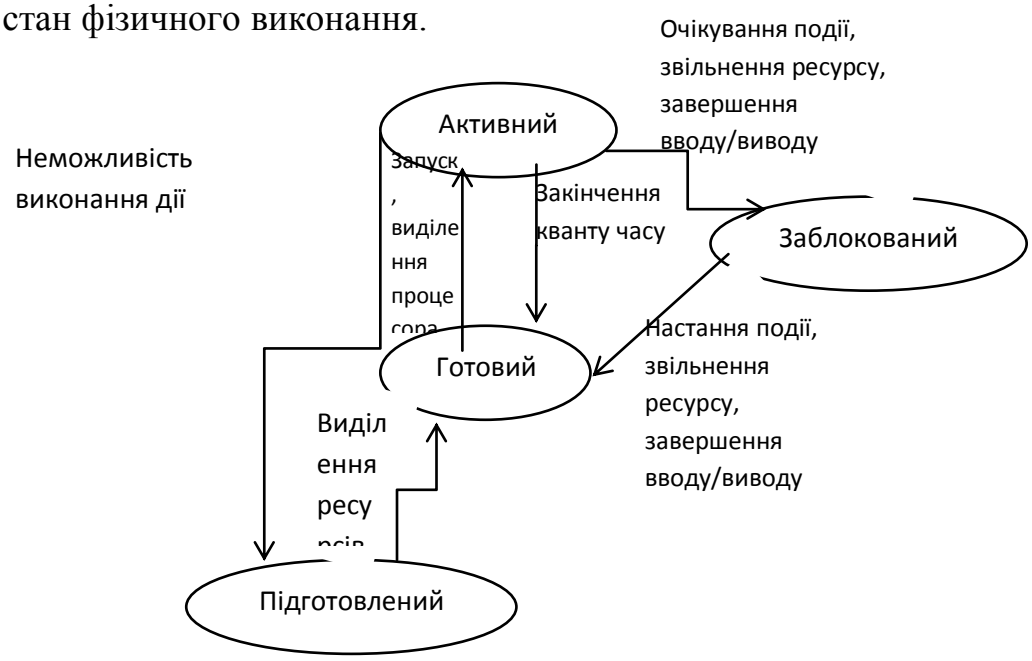


Рис. 1.10. Схема переходу процесу зі стану в стан

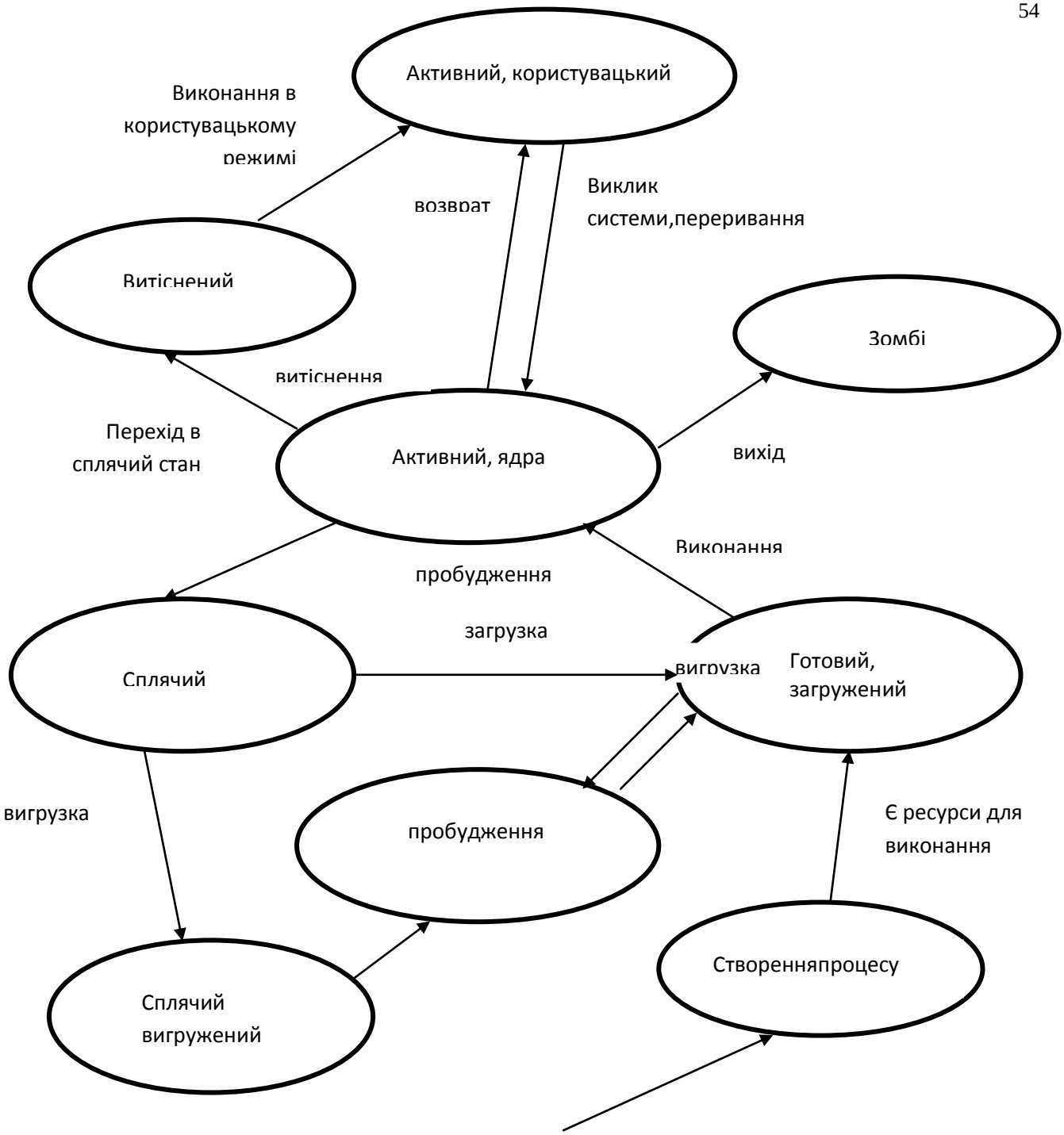
Під час виконання процесу йому можуть знадобитися додаткові ресурси, але для їхнього одержання необхідний якийсь час. Тобто у цьому випадку процес повинен очікувати деякої події (звільнення ресурсу, захопленого іншим процесом) або закінчення виконання операції, яку він сам активізував (наприклад, закінчення операції вводу/виводу для одержання/видачі даних при зв'язку з іншим процесом). Запит, що посилається операційній системі звичайно має вигляд виклику системної функції, тобто виклику процедури, що є частиною операційної системи. Такий процес переводиться в **заблокований** стан або стан **очікування**. Інтерпретатор команд користувача й системні демони проводять у цьому стані більшу частину часу, очікуючи вводу з термінала або мережного з'єднання.

Станів процесів, показаних на рис. 1.10 досить, що б зрозуміти принцип керування процесами операційними системами. У сучасних операційних системах кількість станів більша. Більш повна схема станів і переходів станів процесів на прикладі системи UNIX наведена на рис. 1.11. Активний (виконуваний) має два стани – користувацький і ядра. Додані переходи в стани, витиснуті або свопіровані процеси. Є останній стан процесу - зомбі

Свопірований процес — це процес, переписаний на диск і видалений з оперативної пам'яті комп'ютера. Операційна система свопірує процес на диск, коли конкуренція за оперативну пам'ять настільки висока, що виконання процедур заміщення сторінок (paging) починає погіршувати показники ефективності роботи обчислювальної системи, тому що ОС витрачає невиправдано багато часу на обробку переривань по відсутності потрібних сторінок у пам'яті.

Припинений процес — це процес, виконання якого заборонено. Цей стан аналогічний стану очікування, але вийти з нього можна тільки за допомогою іншого процесу. Процес переводиться в цей стан у тому випадку, якщо подальше виконання процесу неможливе через відсутність в обчислювальній системі необхідних ресурсів.

Процес, що виконав свої функції переходить у стан **зомбі** (zombie, defunct). Як такого процесу вже не існує, але залишаються записи, що містять код повернення й тимчасову статистику його виконання, доступну для батьківського процесу. Цей стан є кінцевим у життєвому циклі процесу. При перекладі процесу в цей стан ядро звільняє ресурси, що належать процесу. У цьому стані процес перебуває доти, поки батьківський процес не виконає один із системних викликів завершення (wait(..)), після чого вся інформація про процес буде знищена, а батько одержить код повернення завершеної програми.



Процеси при надходженні в систему перебувають у підготовленому стані й накопичуються у вхідних чергах завдань. Практично, це ще не процеси, а завдання. Слід розрізняти процеси, що надходять у систему й потребуючі безумовного виконання відповідно до однієї з дисциплін обслуговування, і процеси, що перебувають у підготовленому стані після завантаження операційної системи. Процеси першого виду, як правило, належать користувачам, а другого виду - ОС. Такі процеси перебувають у підготовленому стані доти, поки для виконання функцій системи вони не

будуть активізовані. Користувацькі процеси активізуються або робиться спроба їх активізації при наявності в системі ресурсів. Активізація процесу може проводитися системою при наявності в системі оперативної пам'яті, достатньої, для розміщенні програми, підлеглої процесу.

Тому в системі може перебувати одночасно багато підготовлених і готових процесів, для яких організують черги (одна для готових і одна для підготовлених процесів). Дисципліни обслуговування їх можуть бути різними, а вибір дисципліни обслуговування визначається вимогами, що пред’являються до системи планування.

Якщо система визначила необхідність активізації процесу й виділяє потрібні йому ресурси, крім часу процесора, то вона переводить його в готовий стан. При цьому система визначає (якщо може) чи є в системі ресурси, необхідні для виконання даного процесу. Процес, який переводиться в готовий стан, повинен мати більший пріоритет, ніж процеси, що вимагають таких же, як і він, типів ресурсів (тобто він перебуває на початку черги підготовлених процесів). При цьому сортування процесів у черзі по пріоритетах і планування переходу в готовий стан (стежачи за звільненням ресурсів) повинен здійснювати планувальник даної черги.

Якщо процесор звільнився, то перший (найбільш пріоритетний) процес із черги готових процесів одержує час процесора й переходить в активний стан. Виділення часу процесора процесу здійснює диспетчер.

Якщо активний процес не виконався за виділений йому квант часу, то він переходить знову в готовий стан.

Процес, який вимагає додаткових ресурсів, крім часу процесора, або результати деякої операції виконуваної іншим процесом, переводиться в стан очікування (заблоковане) і очікує настання події, яка потрібна для його продовження. Як тільки очікувана подія відбулася, процес переводиться в готовий стан.

Якщо процес вимагає виконання дій або ресурсів, а операційна система в цей момент ці вимоги не може виконати, він переводиться в підготовлений стан і вважається припиненим.

Задача ОС полягає в спостереженні за станом усіх процесів, що перебувають у системі, особливо в активізованому стані, оскільки в активному стані процес монопольно володіє виділеним йому процесором. Тому йому й виділяється тільки квант часу процесора, щоб процес не займав його на тривалий час, що може погіршити загальну ефективність роботи ОС. Вибір процесів із черги здійснюється відповідно до прийнятої у ОС дисципліною обслуговування заявок на підставі пріоритетного або без пріоритетного обслуговування. У сучасних ОС використовуються різні дисципліни обслуговування заявок у чергах, що дозволяють ураховувати вимоги по поліпшенню характеристик використання обладнанняОС і виконання завдань користувачів.

Система враховує пріоритети користувацьких процесів, що мають різні пріоритети залежно від процесів, що породжують їх і від часу виконання їх на процесорі. Їй також необхідно стежити за чергою, очікуючих (заблокованих) процесів, для того, щоб серед них не було "нескінченно" очікуючих (завислих процесів). Цілком можливо, що подія, яку вони очікують, взагалі може не відбутися - тоді їх треба вивести із системи, тобто виконати припинення процесу. Також може виявитися, що процес, що перебуває в стані «готовності» маючи занадто низький пріоритет довго перебуває в режимі «нескінченного відкладання» - тоді використовуються дисципліни обслуговування з динамічною зміною пріоритетів.

Зі сказаного випливає, що ефективність системи керування процесами в значній мірі впливає на характеристики ОС, пов'язані із пропускною здатністю й ефективністю. Невдалий вибір дисципліни обслуговування процесів і організації керування процесами може привести до того, що деякі процеси можуть так і залишитися в підготовленому або припиненому станах,

що рівносильне втраті заявок, а це суперечить одній з основних функцій роботи операційних систем.

1.8.3. Керування процесами

Виконання функцій ОС, пов'язаних з керуванням процесами, здійснюється за допомогою спеціальних структур даних, що утворюють оточення процесу, середовище виконання або образ процесу. Образ процесу складається із двох частин: даних режиму завдання й режиму ядра. Образ процесу в режимі завдання складається із сегмента коду програми, яка підпорядкована процесу, даних, стека, бібліотек і інших структур даних, до яких він може одержати безпосередній доступ. Образ процесу в режимі ядра складається зі структур даних, недоступних процесу в режимі завдання, які використовуються ядром для керування процесом. Воно містить різну допоміжну інформацію, необхідну ядру під час роботи процесу.

Функціонально образ процесу можна розділити на три частини: контекст користувацького рівня, контекст регістрів, контекст системного рівня.

Контекст користувацького рівня містить основні елементи користувацької програми, отримані зі скомпільованих об'єктних файлів. Він містить дві основні частини - текст програми й область даних. Крім цього кожному процесу виділений користувацький стек, використовуваний процесором для викликів і повернень із процедур, а також при передачі даних і може бути область пам'яті спільно використовувана з іншими процесами для обміну інформацією.

У той час, коли процес не виконується, інформація про стан процесора зберігається в області контекст регістрів

У контекст системного рівня є інформація для операційної системи, на підставі якої ОС виконує керування процесами. Ця інформація знаходится в оперативній пам'яті в області ядра ОС і залишається в пам'яті протягом

усього часу життя процесу. Ця інформація складається із запису фіксованого розміру в таблиці процесів і динамічно мінливої користувацької області.

Кожному процесу в ядрі операційної системи відповідає **блок керування процесом** (PCB – process control block), що являється частиною таблиці процесів контексту системного рівня. **Вхід у процес** (фіксація системою процесу) — це створення його блоку керування (PCB), а **вихід ізпроцесу**— це його знищення, тобто знищення його блоку керування.

Таким чином, для кожного активізованого завдання система створює свій PCB, у якому, у стислому виді, міститься використовувана при керуванні інформація про процес. PCB (це системна структура даних, що містить певні відомості про процес із наступними полями:

1. Ідентифікатор процесу (ім'я);
2. Ідентифікатор батьківського процесу;
3. Поточний стан процесу (виконання, припинений, сон і т.д.);
4. Пріоритет процесу;
5. Прапори, що визначають додаткову інформацію про стан процесу;
6. Список сигналів, що очікують доставки;
7. Список областей пам'яті виділеній програмі, підлеглої даному процесу;
8. Покажчики на опис виділених йому ресурсів;
9. Область збереження регістрів;
10. Права процесу (список дозволених операцій);
11. Пускова адреса програми, підлеглої даному процесу.
12. Статус пам'яті.

Дані структур PCB для всіх процесів, у будь-який момент часу, повинні бути присутнім у пам'яті, хоча інші структури даних, включаючи образ процесу, можуть бути переміщені у вторинну пам'ять, — область свопінгу. Це дозволяє ядру мати під рукою мінімальну й достатню інформацію, необхідну для визначення місцезнаходження інших даних, що ставляться до процесу, навіть якщо вони відсутні в пам'яті. Структура PCB є окремим

записом у системній таблиці процесів. Запис цієї таблиці для часу, що виконується в даний момент, процесу адресується вмістом регістру TR.

Коли ОС перемикає процесор із процесу на процес, вона використовує області збереження регістрів у РСВ для запам'ятовування інформації, необхідної для рестарту (повторного запуску) кожного процесу із точки переривання, коли він наступного разу отримає у своє розпорядження процесор. Кількість процесів у системі обмежена й визначається самою системою або користувачем під час генерації ОС. Невдале визначення кількості програм, що одночасно виконуються, може привести до зниження корисної ефективності роботи системи, тому що перемикання процесів вимагає виконання додаткових операцій по збереженню й відновленню стану процесів. Блоки керування системних процесів створюються при завантаженні системи. Це необхідно, щоб система виконувала свої функції досить швидко, а час реакції ОС було мінімальним. Однак, кількість блоків керування системними процесами менше, чим кількість самих системних процесів. Це пов'язане з тим, що структура ОС має або оверлейну, або динамічно-послідовну або паралельні структури ієрархічного типу, і немає необхідності створювати окремі РСВ для процесів, які ніколи не будуть активізуватися одночасно. При такій організації легко враховувати пріоритети системних процесів, вибудувавши їх по пріоритетах заздалегідь при ініціалізації системи. Блоки керування проблемними (користувацькими) процесами створюються в процесі активізації процесів динамічно, а їх пріоритети можуть змінюватися під час життєвого шляху процесів. Усі РСВ перебувають у виділеній системній області пам'яті.

У кожному РСВ є поле стану процесу. Усі блоки керування системними процесами розташовуються в порядку убування пріоритетів і знаходятся у системній області пам'яті. Якщо пріоритети системних блоків можна визначити заздалегідь, то для проблемних процесів необхідна таблиця пріоритетів проблемних програм. Кожний блок РСВ має стандартну структуру, фіксований розмір, точку входу й містить, як правило, зазначену

вище інформацію й додаткову інформацію для синхронізації процесів. Для синхронізації в РСВ використовуються поля:

— поля для організації зв'язку (два поля):

1. Ім'я викликаного процесу.
2. Ім'я процесу, що викликав.

— поля для організації очікування (два поля).

1. Поле очікування. Ім'я процесу, звільнення якого очікує даний.
2. Лічильник очікування. Число процесів, що очікують даний.

У поле організації зв'язки в породженого процесу вказується ідентифікатор батьківського процесу й код повернення, переданий батьківському процесу при завершенні своїх дій нащадком. Батьківський процес одержує від породженого процесу свій ідентифікатор.

У полях організації очікування, вказується адреса РСВ викликуваного процесу, якщо викликуваний процес зайнятий. У відповідному полі зайнятого процесу записується число процесів, його, що очікують.

Якщо процес А намагається викликати процес В, а в процесу В у РСВ зайнятий ланцюжок зв'язків, тобто він є зайнятим стосовно інших процесів, тоді адреса процесу В записується в поле очікування РСВ процесу А, а до вмісту поля лічильника очікування РСВ процесу В додається 1. Як тільки процес В завершує виконання своїх функцій, він передає керування зухвалому процесу в такий спосіб: В перевіряє стан свого лічильника очікування, і, якщо лічильник більше 0, то серед РСВ інших процесів шукається перший (по пріоритету або іншим ознаках) процес, у поле очікування РСВ якого стоїть ім'я очікуваного процесу, у цьому випадку В, тоді керування передається цьому процесу.

У сучасних ОС для синхронізації доступу декількох процесів до поділюваних ресурсів використовуються семафори. Вони виконують функції дозволу або заборони процесу використання того або іншого ресурсу. Допустимо, є поділюваний ресурс (файл, програма, поділювана пам'ять, загальні змінні…). Необхідно блокувати доступ до ресурсу для інших

процесів, коли якийсь процес робить операцію над ресурсом (наприклад, записує у файл). Для цього зв'яжемо з даним ресурсом якусь ціло чисельну величину — лічильник (семафор), доступний для всіх процесів. Приймемо, що значення 1 семафора означає доступність ресурсу, 0 – його неприступність. Якщо воно рівне 0 — ресурс зайнятий, і операція неприпустима, процесу необхідно чекати. Він переходить у режим очікування або сну (заблокований), що відображається в полі стану процесу в PCB. У системну таблицю процесів при цьому заноситься відповідний прапор стану й визначається подія пробуджуюча процес. Події сповіщають про доступність того або іншого ресурсу. Настання цієї події (звільнення ресурсу) може очікувати кілька процесів. Оскільки перехід з одного стану в іншу дію скоріше логічне, то і пробуджуються ці процеси одночасно. Однак це не означає, що один з них відразу почне виконуватися. Це приведе до того, що їх стан змінитися від «сну» до «готовий до виконання», і вони можуть конкурувати з іншими процесами, що перебувають в аналогічному стані за захоплення часу процесора, тобто переходу в стан «виконання». Якщо воно рівно 1 – можна працювати з ресурсом. Для цього, насамперед, необхідно заблокувати ресурс, тобто змінити значення семафора на 0. Для нормальної роботи необхідно забезпечити виконання наступних умов:

- Значення семафора повинне бути доступно різним процесам. Тому семафор перебуває в адресному просторі ядра.
- Операція перевірки й зміни значення семафора повинна бути реалізована у вигляді однієї атомарної стосовно інших процесів операції. Це виключає ситуацію, коли після перевірки значення семафора виконання процесу буде перервано іншим процесом, який у свою чергу перевірить семафор і змінить його стан. Для цього всі операції над семафорами повинні виконуватися в режимі ядра.

Таким чином, семафори є системним ресурсом, дії над якими проводяться через інтерфейс системних викликів.

Для кожної групи семафорів ядро ОС підтримує спеціальну структуру даних. Наприклад, для кожного семафора System V у цій структурі є наступні поля:

- Значення семафора.
- Ідентифікатор процесу, що виконав останню операцію над семафором.
- Число процесів, що очікують доступність ресурсу асоційованого із семафором, тобто очікуючих зміни значення семафора.

Зміна значення семафора з «0» на «1» при наявності процесів, що очікують доступність ресурсу, приводить до виконання системної операції переведення їх у стан «готовності». Завдання вибору процесу для запуску потім вирішує планувальник процесів.

1.8.4. Операції над процесами

Над процесами можна робити наступні операції: створення, знищення, відновлення, зміна пріоритету, блокування, активізація (пробудження), запуск (вибір), припинення.

1. **Створення** процесу містить у собі:

а) присвоєння імені процесу,

б) включення цього імені в список імен процесів, відомих системі,

в) визначення початкового пріоритету,

г) формування блоку керування процесом,

д) виділення початкових ресурсів.

2. **Знищення** процесу означає його видалення із системи. Ресурси, виділені цьому процесу, вертаються системі, ім'я в системних списках стирається, блок керування процесом звільнюється.

3. **Зміна пріоритету** означає модифікацію значення пріоритету в блоці керування даним процесом. Операційна система збільшує пріоритет у процесів, коли передбачається, що вони будуть перебувати в стані

нескінченного відкладання або зменшує пріоритет процесу, що надовго захопив процесор.

4. **Запуск** (вибір) процесу, здійснюваний планувальником, який виділяє процесу час процесора.

5. **Припинений** процес не може продовжити своє виконання доти, поки його не активізує будь-який інший процес (короткочасне виключення певних процесів у періоди пікового навантаження). У випадку тривалого припинення процесу, його ресурси повинні бути звільнені. Рішення про звільнення ресурсу залежить від його природи. Основна пам'ять повинна звільнятися негайно, а от принтер може й не бути звільнений. Припинення процесу звичайно проводиться в наступних випадках:

а) якщо в системі виникло переривання від схем контролю (наприклад, відбулося перемикання на безперебійний блок живлення), то поточні процеси треба призупинити, виправити причину несправності й виникнення переривання, а потім активізувати припинені процеси;

б) якщо проміжні результати роботи процесу викликали сумнів, то потрібно його призупинити, знайти помилку й запустити або спочатку, або з контрольної точки;

в) якщо ОС перевантажена (у системі активізоване занадто багато процесів), що викликало зниження її ефективності;

г) якщо для продовження нормального виконання процесу необхідні ресурси, які ОС не може йому надати.

Ініціатором припинення може бути або сам процес, або інший процес. В одно процесорній ОС при однопрограмному режимі роботи процес, що виконується, може призупинити сам себе або бути примусово зупинений з пульта керування без можливості відновлення, тому що жоден інший процес не може виконуватися одночасно з ним. У мультипроцесорній системі або при багатопрограмному режимі роботи будь-який процес, що виконується, може бути припинений іншим процесом (системним, батьківським або який відноситься до одногой того ж користувача).

6. **Відновлення або активізація** процесу — операція підготовки процесу до повторного запуску виконувана операційною системою, з тієї точки, у якій він був припинений.

1.8.5. Ієрархія процесів

Процес може породити один або кілька нових процесів. При цьому перший процес, що породжує, називається батьківським, а **другий** — породженим **процесом** або дочірнім. Для створення деякого процесу необхідний тільки один батьківський процес (рис. 1.12.).

Таким чином, у системі створюється ієрархічна структура процесів. На самому верху ієрархії (рис. 1.12) перебувають системні процеси (процес А на малюнку), які у свою чергу породжують дочірні процеси і т.д. Одержуємо деяке дерево процесів, яке може рости вниз і мати стільки ярусів, скільки дозволяє система. Причому в дочірніх процесів може бути тільки один батьківський, але в батьківського - кілька дочірніх.

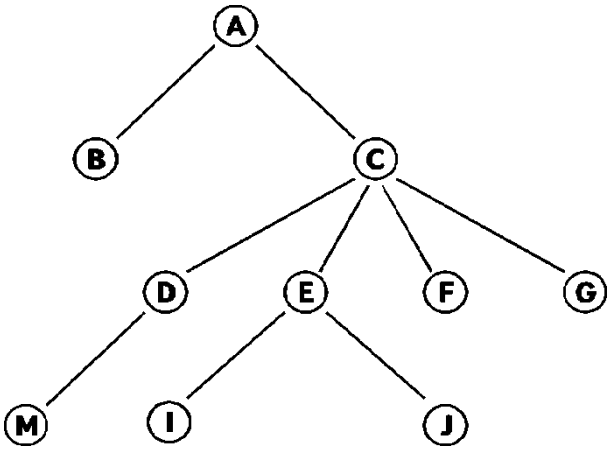


Рис. 1.12. Ієрархія процесів

Усі процеси в UNIX створюються за допомогою системного виклику *forc(2).* Запуск на виконання нових задач здійснюється за схемою forc-and-exec, або за допомогою *exec(2).* “Прабатьком” усіх процесів є процес *init*, називаний також розподільником процесів. Якщо побудувати граф

«родинних відносин між процесами, то вийде дерево, коренем якого є *init*(Рис. 1.13)

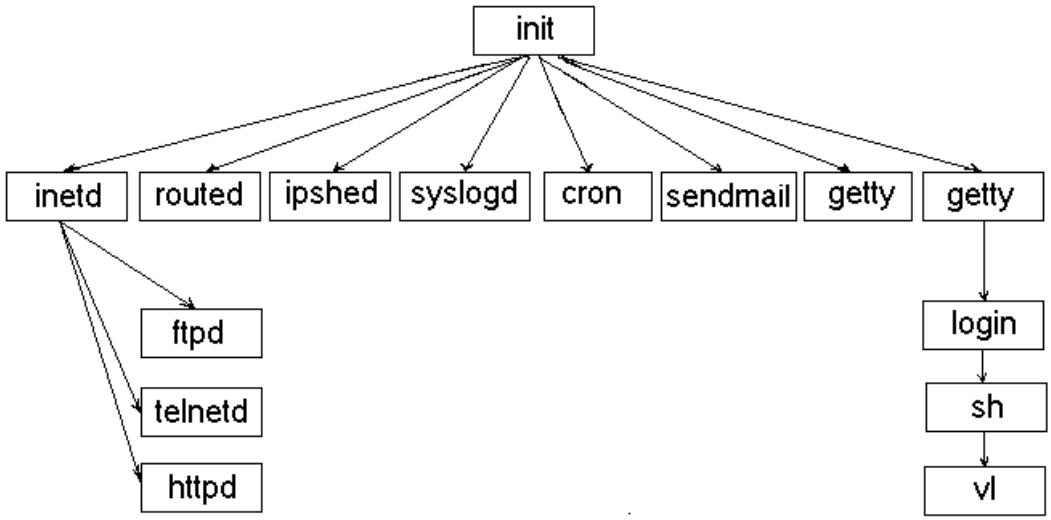


Рис. 1.13 Типове “дерево” процесів в UNIX

Знищення процесів на нижньому ярусі ієрархії не викликає ускладнень, тому що вони не мають дочірніх процесів. При знищенні батьківського процесу в деяких системах видаляються й дочірні процеси, а в деяких дочірні процеси починають існувати незалежно від батьківського, знищеного. При створенні процесу в системі UNIX поточний процес розгалужується на два незалежні паралельні процеси: батьківський і дочірній. Вони не мають загальної первинної пам'яті, але можуть спільно використовувати всі відкриті файли. Для дочірніх процесів створюються копії всіх сегментів даних, у які дозволений запис. Процес створюється системним викликом fork(2). Батьківський процес може бути знищений раніше дочірніх у двох випадках:

1. його виконання не залежить від результатів, щоформуються у дочірніх процесах, і він має всі ресурси, які необхідні для свого завершення. Тоді батьківський процес завершується незалежно від завершення дочірніх. У цьому випадку всі батьківські права для нащадків переходять до батьківських процесів, що стоять вище по ієрархічних відносинах або до системного процесу init;

2. ОС вважає, що процес є причиною створення тупикової ситуації й видаляє його із системи.

Однак, звичайно батьківський процес очікує завершення дочірніх, одержує від них результати й завершується сам.

1.8.6. Перемикання контексту процесу

Кожний процес має контекст, під яким розуміється вся інформація, необхідна для опису процесу. Ця інформація зберігається, коли виконання процесу припиняється, і відновлюється, коли планувальник надає обчислювальні ресурси.

Контекст процесу складається з декількох частин:

- **Адресний простір процесу в режимі задачі**. Сюди входять сегменти коду, даних і стека процесу, а також інші області, наприклад, поділювана пам'ять або код і дані динамічних бібліотек.
- **Керуюча інформація.** Ядро використовує спеціальні структури (РСВ) даних для керування процесом. Сюди ж входять дані, необхідні для відображення віртуального адресного простору у фізичне.
- **Оточення процесу.** Змінні середовища процесу являють собою стоки пар (змінна – значення), які успадковуються дочірнім (породженим) процесом від батьківського й зберігаються в нижній частині стека.
- **Апаратний контекст.** Сюди входять значення загальних і ряду системних регістрів процесора. До системних регістрів відносяться:

— Покажчик інструкцій, що містить адресу наступної інструкції, яку необхідно виконати;

— покажчик стека, що містить адресу останнього елемента стека;

— регістри плаваючої коми;

— регістри керування пам'яттю, відповідальні за перетворення віртуальної адреси процесу у фізичний.

Перемикання між процесами, необхідне для справедливого розподілу часу процесора, по суті виражається в перемиканні контексту, коли контекст процесу, що виконувався, запам'ятовується, а контекст процесу, обраного планувальником відновлюється. Перемикання контексту є досить трудомісткою операцією. Крім збереження стану регістрів процесу, ядро виконує й інші дії.

Можна виділити чотири основні ситуації, при яких проводиться перемикання контексту:

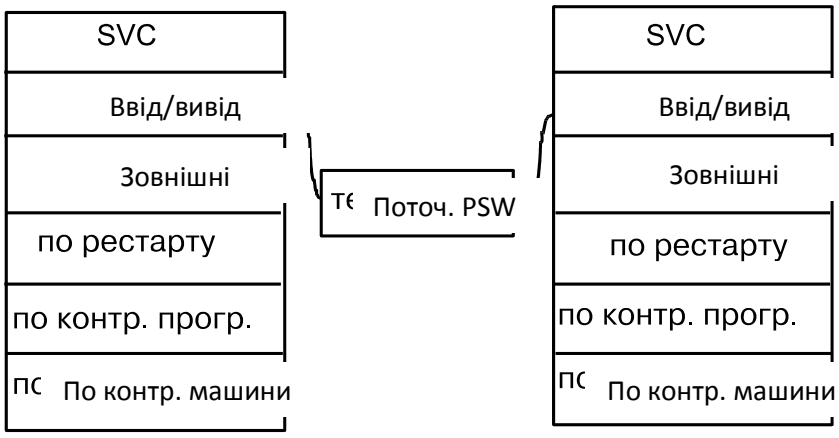
1. Поточний процес переходить у стан сну, очікуючи недоступного ресурсу.
2. Поточний процес завершує своє виконання.
3. Після перерахування пріоритетів у черзі на виконання береться більш високо пріоритетний процес.
4. Відбувається пробудження більш високо пріоритетного процесу.

Перемикання контексту виконується з появою у ОС переривання. Переривання (це порушення природньої послідовності виконання дій (команд), коли після поточного дії (команди) виконується не наступна команда, а деяка інша дія.

З появою запиту переривання керування передається ОС, яка запам'ятовує стан перерваного процесу. Далі ОС аналізує тип переривання й передає керування відповідній до програми обробки цього переривання. Ініціатором переривання може бути процес, що виконується, або воно може бути викликане деякою подією, зв'язаною або не зв'язаною із цим процесом.

Розглянемо механізм передачі керування **програмі обробки переривання (ІН)**. Як було сказано вище, ОС запам'ятовує стан перерваного процесу й передає керування ІН. Ця операція називається **перемиканням контексту.** При реалізації перемикання використовуються **слова стану програми (PSW)**, за допомогою яких здійснюється керування порядком виконання команд. В **PSW** міститься інформація про стани процесу на момент переривання, що забезпечує продовження перерваної програми.

Існують три типи PSW: поточне, нове й старе PSW (мал.1.14.). Адреса наступної команди (активної програми), що підлягає виконанню, міститься в поточному PSW. У ньому також вказуються й типи переривань, дозволених і заборонених у даний момент.



Старі PSW Нові PSW

Рис. 1.14

Процесор реагує тільки на демасковані переривання, а масковані ігнорує або затримує їхнє виконання. Адреса наступної команди, що зберігається в **поточному PSW**, при виконанні командного циклу передається лічильнику команд. Адреса нової команди, що виконується, записується в PSW (по першим двом бітам команди визначається її довжина). Крім цього, в PSW перебувають: ознака результату попередньої команди, поля масок деяких переривань (наприклад, каналів і переривань виняткових ситуацій).

У **нових PSW** містяться адреси програм обробки переривань. При виникненні дозволеного переривання, в одному зі **старих PSW** (відповідному до типу переривання) система зберігає вміст поточного PSW. При цьому, зберігається адреса наступної команди поточного процесу, яка повинна виконатися по закінченню обробки переривання й передачі керування перерваному процесу (тобто адреса команди, яка іде за поточною в даному процесі і яка повинна була б виконатися при відсутності сигналу переривання). Таким чином, забезпечується коректне повернення в перерваний процес після обробки переривання.

Одне з нових PSW, яке містить адреси оброблювача переривання, відповідного надійшовшому запиту переривання, записується в поточне PSW, а потім відбувається перехід на відповідну програму обробки переривання. Таким чином, за допомогою механізму зміни PSW організує вхід в перериваючу програму й повернення в перервану. Старі PSW можуть зберігатися в постійній області пам'яті разом з векторами переривань; у стеці тієї програми, яка переривається або в PCB перерваного процесу. Якщо старе PSW перебуває в постійній області пам'яті, то кількість виділених для цього PSW дорівнює кількості класів переривань. У цьому випадку глибина переривань визначається кількістю старих PSW.

При виконанні процедури виходу з переривання, ОС може передати керування процесору для продовження виконання перерваного процесу, якщо ОС не допускає перехоплення процесора більш пріоритетним процесом (першому в черзі готових процесів), тоді даний процес переводиться в готовий стан.

1.8.7. Потоки

Потоком (потоком керування, ниткою, thread) називається набір виконуваних команд процесора, що використовують загальний адресний простір процесу. У системі може одночасно перебувати безліч процесів, а в рамках виконання кожного процесу кілька потоків, що використовують його адресний простір. Завданням ОС є організація перемикання процесора між ними й планування їх виконання. У багатопроцесорних системах код окремих потоків може виконуватися на окремих процесорах.

Захищеність адресного простору процесу є його найважливішою характеристикою. Код і дані процесу не можуть бути прямо прочитані або перезаписані іншим процесом. З іншого боку, потоки в рамках процесу розпоряджаються загальною пам'яттю, у більшості випадків потік не захищений від доступу до його даних з інших потоків за умови, що всі вони виконуються в одному адресному просторі. Це надає програмістові більші можливості, дозволяючи йому розпаралелювати виконання програм і

підвищувати їх продуктивність (створення потоку вимагає менше ресурсів, чим створення адресного простору процесу), але при цьому робить багато поточне програмування більш складним і менш захищеним від помилок, тому що доступ до глобальних змінних по читанню й запису мають усі потоки одного процесу.

1.8.8. Моделі процесів і потоків

Операційна система може підтримувати набір потоків, що одночасно виконуються, і набір одночасно існуючих захищених адресних просторів, пов'язаних із процесами. Максимально припустиме число процесів і потоків залежить від системи. Наприклад, в одно задачних системах є тільки один адресний простір, у якому у даний момент часу може виконуватися один потік. З іншого боку, у вбудованих системах теж може бути один адресний простір на всю систему, але в ньому може бути припустиме виконання безлічі потоків. У цьому випадку весь код виконується в режимі ядра, але при цьому зберігається можливість організації паралельних обчислень. Ще одним прикладом системи такого роду може служити віртуальна машина Java.

У системах з підтримкою мультипрограмування може бути одночасно виділена безліч захищених адресних просторів, тобто може існувати безліч процесів.

Можна виділити два основні підходи.

- Для систем, подібних традиційних систем UNIX (Version 7, System V Release 3, BSD) допускається безліч захищених адресних просторів (процесів) у даний момент часу, але в рамках одного адресного простору процесу допускається тільки один потік. Це – традиційна одно потокова *модель процесів*, коли по суті справи одиницею виконання коду є саме процес. Насправді в такій моделі про потоки взагалі не говорять, уживаючи такі терміни, як «перемикання між процесами», «планування виконання процесів», «послідовність команд процесу» і т.д., розуміючи в цих випадках

під процесами їх єдині потоки. Таке уявлення про процеси було загальноприйнятим до початку 90-х років XX століття.

- Для більшості сучасних ОС (таких, як лінія Windows 2000, сучасні версії UNIX, такі як Solaris і Linux) допускається існування безлічі потоків у рамках адресного простору процесу й допускається безліч процесів у системі. Говорять, що такі системи *підтримують багатопоточність* або реалізують *модель потоків*. Процес у такій системі ми будемо називати багатопоточним процесом.

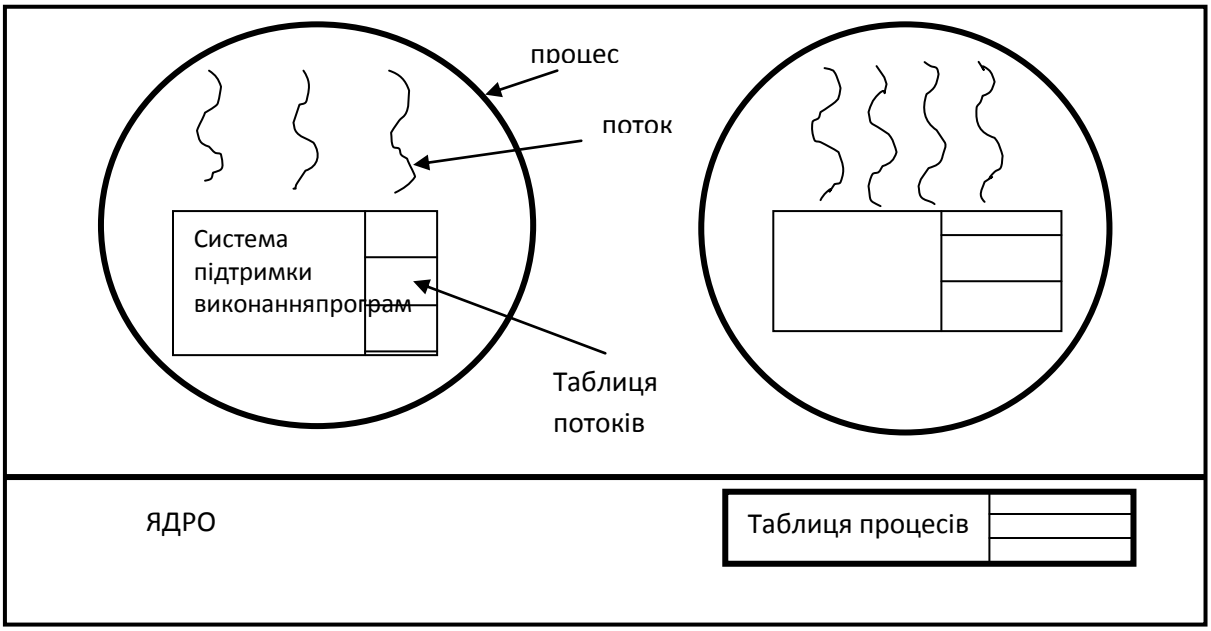


Рис. 1.14 Пакет процесів у просторі користувача

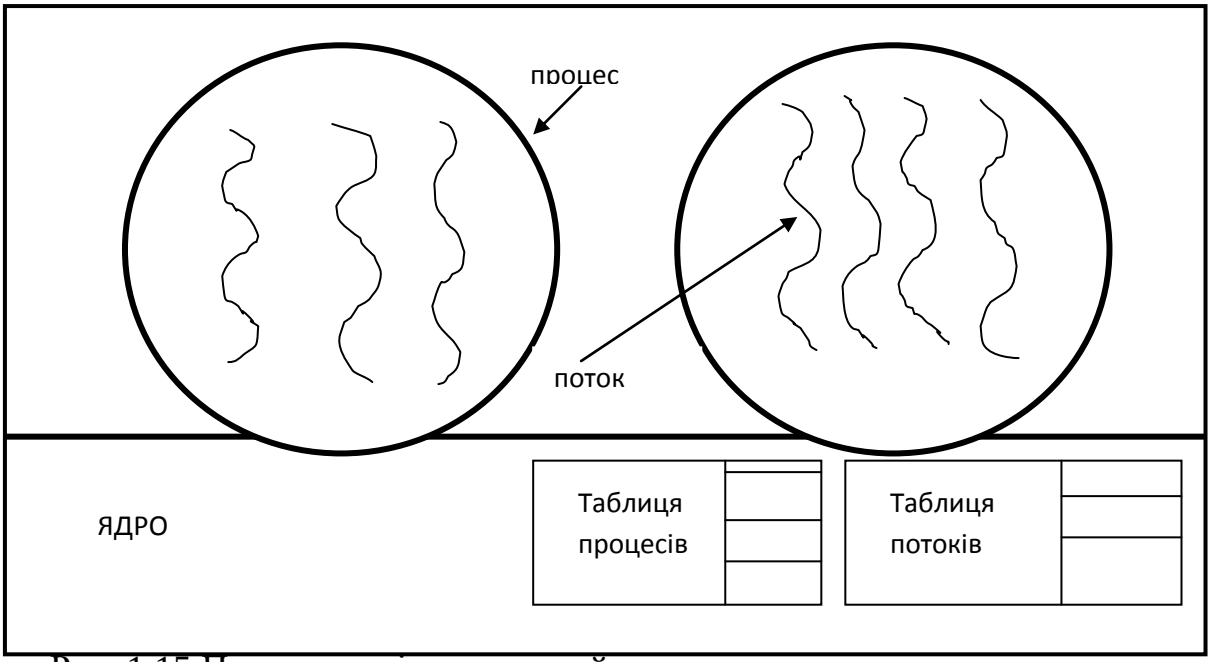


Рис. 1.15 Пакет потоків, керований ядром

1.8.9. Складові елементи процесів і потоків

До елементів процесу відносяться:

- виділений, захищений адресний простір;
- глобальні змінні, загальні для всього процесу (ці дані можуть спільно використовуватися всіма потоками даного процесу);
- інформацію про використання ресурсів (відкриті файли, мережні з'єднання і т.д.);
- інформацію про потоки даного процесу

Потік містить:

- лічильник поточної інструкції процесора;
- стан процесора (набір поточних даних з його регістрів);
- стік потоку (область пам'яті, яка містить локальні змінні потоку й адреси повернення функцій, які викликає даний потік).

Насправді лічильник поточної інструкції теж міститься в спеціальному регістрі процесора і, отже, є частиною його стану, але зручно розглядати його окремо. Покажчик на стек теж є частиною стану процесора.

1.9. Багатопоточність і її реалізація

1.9.1. Поняття паралелізму

Використання декількох потоків у додатку означає внесення в цей додаток *паралелізму*. *Паралелізм означає одночасне виконання дій різними фрагментами коду програми*. Звісно, що поняття «одночасне» тут означає «одночасно з погляду прикладного програміста», така одночасність насправді може бути реалізована на одному процесорі на основі перемикання задач, а може бути заснована на паралельному виконанні коду на декількох процесорах. Насправді, потоки абстрагують цю відмінність, дозволяючи розробляти програми, які в однопроцесорних системах використовують псевдопаралелізм, а при додаванні процесорів – справжній паралелізм.

1.9.2. Підходи до реалізації моделі потоків

Модель потоків є базовою моделлю організації виконання програмного коду в сучасних операційних системах. Насамперед цим необхідно визначити відмінність між користувацьким потоком і потоком ядра.

Користувацький потік це послідовність виконання команд у рамках процесу користувача. Ядро ОС може не підозрювати про існування користувацьких потоків. Тоді планування таких потоків і перемикання між ними проводиться в користувацькому режимі в рамках програми. У цьому випадку програміст реалізує таке планування через бібліотеку підтримки потоків за допомогою виклику бібліотечних функцій.

Бібліотеки підтримки потоків у цей час найчастіше реалізують набір функцій, визначений стандартом POSIX (відповідний розділ стандарту називається POSIX 1b), у цьому випадку говорять про підтримку бібліотекою *потоків POSIX.*

Потік ядра являє собою послідовність виконання команд, видиму для ядра ОС. ОС повністю управляє потоками ядра, для перемикання між такими потоками необхідний перехід у привілейований режим, а за планування виконання потоків ядра відповідає ОС. Відзначимо, що серед потоків ядра можуть бути потоки, відповідні до користувацьких потоків, а можуть бути потоки, що виконують команди, потрібні тільки ядру. Звичайно в ОС код керування потоками ядра доступний через системні виклики, але прикладний програміст не має справи із цими викликами безпосередньо – він у будь-якому разі використовує виклики бібліотеки підтримки потоків.

Взаємовідношення між двома видами потоків визначає реалізацію моделі потоків. Існує кілька варіантів (*моделей*) такої реалізації, розглянемо найбільш важливі з них.

Найбільш рання модель являє собою реалізацію багато потоковості винятково в користувацькому режимі. У даній ситуації кожний процес може містити безліч користувацьких потоків, при цьому про існування цих потоків ОС не знає. При перемиканні контексту й плануванні ОС працює винятково

із процесами, усю роботу з керування потоками в рамках процесів бере на себе бібліотека підтримки потоків. Така модель називається *моделлю M:1*. До її переваг відноситься висока продуктивність перемикання контексту й планування (оскільки для цих задач немає необхідності переходити в режим ядра), а також те, що для його реалізації не потрібно змінювати ядро ОС. Тим не менше, у наш час вона практично не використовується через два важливих недоліки, що йдуть врозріз із ідеологією баготопоточності.

1. Оскільки визначити, який саме код буде виконуватися на кожному із процесорів, може тільки ядро ОС, такий підхід не дозволяє скористатися багатопроцесорними архітектурами (усі потоки одного процесу будуть завжди виконуватися на одному процесорі).

2. Оскільки системні виклики обробляються на рівні ядра ОС системний виклик, що блокує (наприклад, виклик, який очікує вводу користувача) заблокує всі потоки даного процесу, а не тільки той, який цей виклик виконав.

Друга модель (*модель 1:1*) ставить у відповідність кожному користувацькому потоку один потік ядра. У цьому випадку планування й перемикання контексту торкається тільки потоків ядра, у користувацькому режимі ці функції не реалізовані. При цьому, оскільки ядро ОС знає про потоки, ми звільняємося від недоліків моделі M:1 (тепер різні потоки можуть виконуватися на різних процесорах, а при блокуванні одного потоку інші продовжують роботу). Дана модель є найпростішою в реалізації й у цей час використовується найбільш широко. Теоретично, її недоліком є те, що при керуванні потоками потрібно постійно перемикатися з користувацького режиму в режим ядра й назад, але на практиці цей програш у продуктивності виявився не настільки значним, більш істотною представляється простота й надійність реалізації.

Ще одна важлива модель – *модель M:N*. У даній моделі існують як потоки ядра, так і користувацькі потоки, при цьому користувацькі потоки відображаються на потоки ядра так, що один потік ядра може відповідати

декільком користувацьким. Число потоків ядра може змінюватися програмістом з метою досягнення максимальної продуктивності. Розподіл користувацьких потоків по потоках ядра проводиться в користувацькому режимі, планування потоків ядра – у режимі ядра. Насправді можуть існувати два планувальники – один для користувацького режиму, інший для режиму ядра. Модель M:N є найбільш складною в реалізації й у цей час уступає свої позиції більш простій і надійній моделі 1:1.

1.10. Ядро операційної системи

Програми, що виконують дії, пов'язані з керуванням процесами, входять у базове системне програмне забезпечення машини. Вони включаються в комплекс використовуваних засобів, сукупність яких становить **ядро ОС**. Таким чином, усі операції, пов'язані із процесами, здійснюються під управлінням ядра ОС, яке представляє лише невелику частину коду ОС у цілому. Оскільки ці програми часто використовуються, то резидентно розміщуються в ЗП і становлять ядро супервізора. Інші ж частини ОС переміщаються в ЗП у міру необхідності. Ядро ОС приховує від користувача приватні особливості фізичної машини, надаючи йому всі необхідні засоби для організації обчислень.

Виконання програм ядра може здійснюватися двома способами:

1. Викликом примітива керування процесами (створення, знищення, синхронізація і т.д.); ці примітиви реалізовані у вигляді звертань до супервізора (керуючим програмам ядра);

1. Перериванням: програми обробки переривань становлять частину ядра, тому що вони безпосередньо пов'язані з операціями синхронізації й ізольовані від вищих рівнів керування.

Ядро ОС містить програми для реалізації наступних функцій:

- обробки переривань;
- створення й знищення процесів;

- перемикання процесів зі стану в стан;
- диспетчеризації завдань, процесів і ресурсів;
- припинення й активізації процесів;
- синхронізації процесів;
- організації взаємодії між процесами;
- маніпулювання РСВ;
- підтримки операцій вводу/виводу;
- підтримки розподілу й перерозподілу пам'яті;
- підтримки роботи файлової системи;
- підтримки механізму виклику - повернення при звертанні до процедур;
- підтримки певних функцій по веденню обліку роботи ОС;

Одна з найважливіших функцій, реалізована в ядрі - обробка переривань.

У системі постійно виникає велика кількість переривань і потрібна швидка реакція на них для того, щоб ефективно використовувалися ресурси системи, а користувач якнайшвидше одержував відповідь (на запит у вигляді переривання). При обробці (дешифрації) поточного переривання ядро забороняє інші переривання й дозволяє їх тільки після завершення обробки поточного. При великій інтенсивності появи запитів переривань може скластися така ситуація, коли ядро буде блокувати переривання протягом значного проміжку часу, тобто не буде мати можливості ефективно реагувати на переривання. Тому, ядро супервізора ОС звичайно здійснює лише мінімально можливу попередню обробку кожного переривання, а потім передає це переривання на подальшу обробку відповідному до системного процесу, після початку роботи якого, ядро могло б реагувати на наступні переривання. Таким чином, поліпшується реакція системи на переривання (сигнал переривання).

1.11 Класичні завдання, пов'язані з доступом до спільних ресурсів (СР)

* *Виробники —споживачі. Поняття кільцевого буфера*

Розглянемо пари "виробник - споживач" (рис. 1.15).

Рис. 1.15

У якості СР виступає деякий **буфер**, куди заносяться й звідки зчитуються дані. Один **процес**—джерело або **виробник**, генерує інформацію, яку інший **процес**—одержувач або **споживач** використовує. Ця взаємодія проводиться за допомогою буфера. Процес-виробник генерує дані або здійснює деякі обчислення й результат заносить у буфер, звідки процес-споживач зчитує дані й друкує їх. Якщо обидва процеси працюють із приблизно однаковими швидкостями, то проблем передачі/прийому практично не існує; а якщо ні, то, при різних швидкостях уведення й виведення, виникають труднощі синхронізації.

Якщо процес-виробник працює швидше, то він може кілька разів перезаписувати дані в буфері, перш ніж споживач зчитає їх. Відбувається втрата інформації, чого не можна допустити. Тоді процес-виробник повинен чекати, поки споживач не зчитає дані, тільки тоді він зможе знову заносити дані в буфер. При цьому не буде втрати інформації (швидкість обміну процесу-споживача). Якщо ж процес-споживач швидше, то він буде кілька разів зчитувати ті самі дані до заміни їх виробником. Має місце дублювання

інформації. У цьому випадку споживач повинен чекати заміни інформації в буфері виробником.

В СР передбачається виділення деякої кількості комірок пам'яті для використання як буфера передач. Цей буфер можна представити у вигляді масиву заданого розміру. Процес — виробник поміщає передані дані в послідовні елементи цього масиву. Споживач зчитує дані в порядку їх надходження. Виробник може випереджати споживача на кілька кроків. Згодом, процес—виробник заповнить останній елемент виділеного буфера—масиву. Коли він сформує чергові дані для передачі, то повинен буде вернутися до початку буфера й продовжити запис, починаючи з першого елемента буфера. Такий масив працює як замкнене кільце й носить назву **кільцевого буфера**. Так як розмір буфера обмежений, процес-виробник може зіштовхнутися з тим, що всі елементи масиву зайняті. У цьому випадку виробник повинен почекати, поки споживач не зчитає й не звільнить хоча б один елемент масиву. Може виникнути ситуація, коли споживач хотів би прочитати дані, а масив порожній. Тоді споживач повинен буде очікувати, поки виробник не почне поміщати дані в буфер. Таким чином, такий буфер добре використовувати для передачі даних від процесу-виробника процесу-споживачеві, якщо вони мають різні швидкості. Прикладом реалізації такої схеми є буфер клавіатури ПК.

* *Читачі—письменники*

Процеси—письменники записують інформацію в деякий ресурс, наприклад, порт (рис. 1.16).

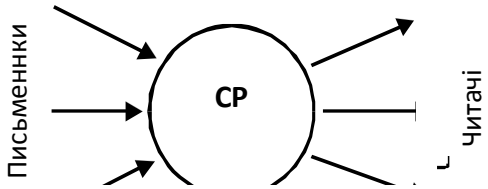


Рис. 1.16

Процеси—читачі повинні зчитувати цю інформацію, але записувати не мають права. Може скластися ситуація, коли є багато процесів-письменників, які по черзі записують інформацію в порт. При цьому процеси-читачі можуть перебувати в ситуації нескінченного відкладання, тобто очікування, поки процеси-письменники закінчать запис і читачі отримають доступ до СР (порту) для зчитування. Тому, з появою нового процесу-письменника пріоритет віддається наявним у системі процесам-читачам. Однак, тепер уже процеси-читачі можуть надовго заблокувати доступ до СР письменникам. У цьому випадку, з появою нових процесів-читачів, більший пріоритет мають процеси-письменники, що перебувають у системі.

1.12. Тупики. Тупикові ситуації й методи боротьби з ними

Тупик — це:

1. ситуація, з якої система не може вийти;
2. ситуація, коли процес чекає події, яка ніколи не відбудеться.

Прикладом з життя може служити транспортна пробка на перехресті.

Другий приклад: круговий ланцюг очікування (рис. 1.17).

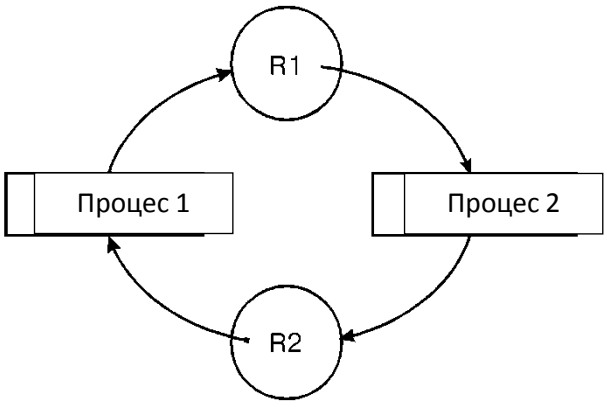


Рис. 1.17

Тут ресурс 2 виділений процесу 1, а ресурс 1 – процесу 2. У якийсь момент процес 1 вимагав додатково ресурс 1, а процес 2 – ресурс 2. Звісно,

жоден із цих ресурсів не може бути виділений процесу, що його вимагав. Одержуємо тупикову ситуацію, при якій жоден із двох процесів не може закінчити своє виконання, поки не буде звільнений необхідний ресурс, чого в цьому випадку ніколи не зможе відбутися. Вихід полягає у видаленні одного із процесів, що потрапили в тупикову ситуацію із системи.

Тупик ***неминучий***, якщо присутні чотири умови його виникнення:

1. Умова *взаємного виключення*. Процес має монопольне право володіння ресурсами під час усього існування процесу.

2. Умова *очікування*. Процес, володіючи монопольним правом, намагається захопити нові ресурси.

3. Умова *перерозподілу*. Ресурс не можна відібрати до повного завершення процесу.

4. Умова: *круговий ланцюг очікування*. Кожний із процесів ланцюжка вимагає додатковий ресурс, який уже виділений процесу даного ланцюжка.

* *Тупики в системі спулінга*

При введенні завдань використовується режим **спулінга**— режим буферизації для вирівнювання швидкостей при введенні й зчитуванні інформації з буфера. Система, що використовує режим спулінга, сильно піддана тупикам.

У таких системах буфер може бути повністю заповнений до того, як дані з нього стануть надходити на друкувальний пристрій, а його розміру недостатньо для буферизації всіх даних виводу. При цьому виникає тупикова ситуація, тому що розмір буфера недостатній для розміщення всіх даних, що надійшли, а принтер не зможе почати виведення. У цьому випадку єдиним виходом є рестарт системи, але при цьому втрачається вся інформація.

У таких системах гарним рішенням було б неповне заповнення буфера (на 75-80%) і передача після цього керування друкувальному пристрою. Після

звільнення буфера можна його знову не повністю заповнити, потім знову перейти до роздруківки і т.д.

У багатьох сучасних ОС принтер починає роздруківку до заповнення буфера (по мірі заповнення). При цьому частина буфера (роздрукована) звільняється й туди можна заносити нову інформацію. У таких системах імовірність виникнення тупикової ситуації значно менше.

1.13. Способи боротьби з тупиками

Наслідки тупиків рівносильні ситуації нескінченного відкладання. Тупики й ситуація нескінченного відкладання – це ситуації рівносильні "втраті процесу".

Можна виділити 4 стратегії боротьби з тупиками:

1. **Запобігання тупика**. Потрібно створити умови, що виключають можливість виникнення тупикових ситуацій. Однак цей метод часто приводить до нераціонального використання ресурсів.

2. **Обхід тупика**. Цей метод передбачає менш жорсткі обмеження, ніж перший і забезпечує краще використання ресурсів. Метод ураховує можливість виникнення тупиків, і при збільшенні ймовірності тупикової ситуації вживаються заходи щодо обходу тупика.

3. **Виявлення тупиків**. Потрібно встановити сам факт виникнення тупикової ситуації, причому точно визначити ті процеси й ресурси, які в неї включені.

4. **Відновлення після тупиків**. Необхідно усунути тупикову ситуацію, щоб система змогла продовжити роботу, а процеси, що потрапили в тупикову ситуацію, змогли завершитися й звільнити займані ними ресурси. Відновлення – це серйозна проблема, так що в більшості систем воно здійснюється шляхом повного виведення з розв'язку одного або більше процесів, що потрапили в тупикову ситуацію. Потім виведені процеси звичайно перезапускаються із самого початку, так що вся або майже вся виконана процесами робота втрачається.

1.14 Запобігання тупиків

Виникнення тупика неможливо, якщо порушується одне із чотирьох необхідних умов виникнення тупикових ситуацій. Першу умову при цьому можна й не порушувати, тобто вважаємо, що процес може мати монопольне право володіння ресурсами. Можна виділити наступні три способи запобігання тупиків:

І спосіб. Процес запитує й отримує всі ресурси відразу. Якщо процес не має можливості одержати всі необхідні ресурси відразу, тобто деякі з них на даний момент зайняті, то він повинен очікувати звільнення необхідних ресурсів. При цьому він не повинен утримувати за собою які-небудь ресурси. Порушується умова "очікування додаткових ресурсів" (2-е) і тупикова ситуація неможлива. При цьому способі можливий простій процесів, що очікують виділення ресурсів, і роботу значної частини ресурсів вхолосту. Можна вдатися до поділу програми на кілька програмних кроків, що працюють незалежно друг від друга. При цьому виділення ресурсів можна здійснювати для кожного кроку програми, однак, при цьому збільшуються витрати на етапі проектування прикладних програм.

Цей спосіб має два істотні недоліки:

1). Знижується ефективність роботи системи;

2). Підсилюється ймовірність нескінченного відкладання для всіх процесів.

ІІ спосіб. Якщо процес, що втримує за собою певні ресурси, зажадав додаткові й отримав відмову в їхньому отриманні, він повинен звільнити усі раніше отримані ресурси. При необхідності, процес повинен буде запросити їх знову разом з додатковими. Тут порушується умова "перерозподілу" (3-є). При цьому способі кожний процес може кілька разів запитувати, отримувати й віддавати ресурси системі. При цьому, система буде виконувати масу зайвої роботи – відбувається деградація системи з погляду виконання корисної роботи.

Недоліки цього способу:

1. Якщо процес протягом деякого часу втримував ресурси, а потім звільнив їх, він може втратити інформацію, яка в нього була.

2. Тут також можливо нескінченне відкладання процесів – процес запитує ресурси, одержує їх, однак не може завершитися, тому що потрібні додаткові ресурси; далі він, не завершившись, віддає наявні в нього ресурси системі; потім він знову запитує ресурси і т.д.

Маємо нескінченний ланцюжок відкладання завершення процесу.

III спосіб. Стратегія лінійності. Усі ресурси вибудовані в порядку присвоєних пріоритетів, і захват нових ресурсів може бути виконаний тільки в порядку зростання пріоритетів. При цьому порушується умова "круговий ланцюг очікування" (4-е).

Труднощі й недоліки даного способу:

1. Оскільки запит ресурсів здійснюється в порядку зростання пріоритетів, а вони призначаються при установці машини, то у випадку введення нових типів ресурсів може виникнути необхідність переробки існуючих програм і систем;

2. Пріоритети ресурсів повинні відображати нормальний порядок, в якому більшість завдань використовують ресурси. Однак, якщо завдання вимагає ресурси не в цьому порядку, то воно буде захоплювати й утримувати деякі ресурси задовго до того, як з'явиться необхідність їх використання – втрачається ефективність.

3. Спосіб не надає зручності користувачам.

1.15. Обхід тупиків

Алгоритм "банкіра".

При використанні алгоритму "банкіра" мається на увазі, що:

1) системі заздалегідь відома кількість наявних ресурсів;

2) система знає, скільки ресурсів може максимально знадобитися процесу;

3) число користувачів (процесів), що звертаються до ресурсів, відоме й постійне;

4) система знає, скільки ресурсів зайнято в цей момент часу цими процесами (загалом, і кожним з них);

5) процес може зажадати ресурсів не більше, чим є в системі.

В алгоритмі "банкіра" введено поняття надійного стану. Поточний стан обчислювальної машини називається **надійним**, якщо ОС може забезпечити всім поточним користувачам (процесам) завершення їх завдань протягом кінцевого часу. Інакше стан вважається **ненадійним**. Надійний стан – цей стан, при якому загальна ситуація з ресурсами така, що всі користувачі мають можливість згодом завершити свою роботу. Ненадійний стан може згодом привести до тупика.

Система задовольняє тільки ті запити, при яких її стан залишається надійним. При запиті ресурсу система визначає, чи зможе вона завершити хоча б один процес, після цього виділення.

Приклад розв'язку завдання виділення ресурсів по алгоритму "банкіра".

Маємо 6 ресурсів і 6 процесів (Табл. 1.2). У таблиці показаний стан зайнятості ресурсів на даний момент (1 вільний ресурс).

Табл. 1.2

Процес	Виділена кількість ресурсів	Потрібне число ресурсів
A	2	3
B	1	3
C	0	2
D	1	2
E	1	4
F	0	5

Ця таблиця відображає надійний стан, тому що існує така послідовність виділення – звільнення ресурсів, при якій усі процеси за кінцевий час будуть завершені.

Недоліки алгоритму банкіра:

1) Якщо кількість ресурсів велика, то стає досить складно обчислити надійний стан.

2) Досить складно спланувати, яка кількість ресурсів може знадобитися системі.

3) Число працюючих користувачів (процесів) залишається постійним.

4) Користувачі повинні заздалегідь зазначати максимальну потребу в ресурсах, однак, потреба може визначатися динамічно в міру виконання процесу.

5) Користувач завжди замовляє ресурсів більше, ніж йому може знадобитися.

6) Алгоритм вимагає, щоб процеси повертали виділені їм ресурси протягом деякого кінцевого часу. Однак, для систем реального часу потрібні більш конкретні гарантії. Тобто у системі повинно бути задано максимальний час захоплення всіх ресурсів процесом (за цей час ресурси повинні бути звільнені).

Графічний метод обходу тупика (рис. 1.18).

Маємо 2 процеси Р1 і Р2 і кожному з них для виконання потрібні по 2 ресурси: Д - диск, П - процесор. Якщо Р1 захопив процесор, а Р2 - диск, то виникає тупикова ситуація. Однак, якщо Р1 захопив П и Д і завершив своє виконання, то Р2 захоплює ці ж ресурси й закінчує виконання - безтупикова взаємодія.

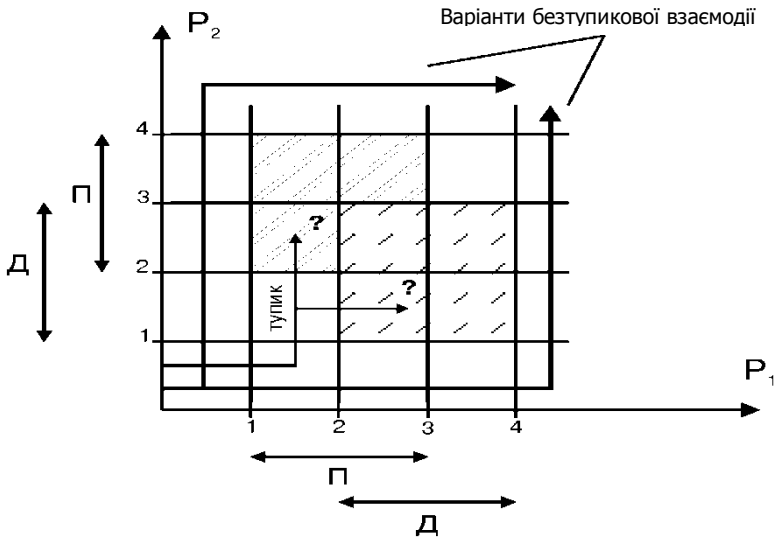


Рис. 1.18

1.16. Виявлення тупиків

Виявлення тупика— це встановлення факту того, що виникла тупикова ситуація й визначення процесів і ресурсів, залучених у тупикову ситуацію. Для виявлення тупикової ситуації використовують операцію **редукції графа.** Ця операція виконується системою, коли є підозріння, що якісь процеси перебувають у тупику. При цьому визначається, які процеси можуть успішно завершитися (по графу) і звільнити ресурси. Якщо запити ресурсів для деякого процесу можуть бути задоволені, то граф можна редукувати на цей процес. При цьому процес видаляється із графа, включаючи всі дуги від нього до необхідних ресурсів і від виділених ресурсів до нього, тобто ці ресурси можуть уважатися вільними й виділятися іншим процесам. Якщо граф можна редукувати на всі процеси, виходить, тупикової ситуації немає, а якщо цього зробити не можна, то всі процеси, що не редукуються, утворюють групу процесів, що перебувають у тупиковій ситуації.

Приклад графа, що редукується та містить 5 процесів і 4 типи ресурсів, показаний на рис. 1.19.

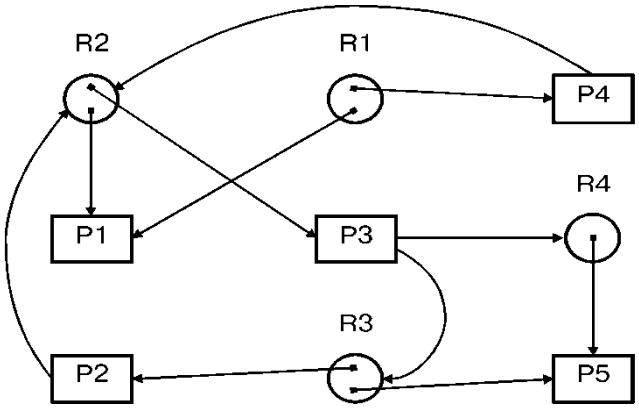


Рис. 1.19

Примітка до рис.1.19.: дуга від процесу до ресурсу (колу) — вимога ресурсу цього типу; тип ресурсу - кружок, а самі ресурси - квадратики

усередині кола; P —позначення процесу; R —типу ресурсу; дуга від ресурсу (квадрата) до процесу — ресурс виділений процесу.

На рис. 1.19 показаний граф без тупикової взаємодії.

* *Відновлення після тупиків.*

Систему, що опинилась в тупику, необхідно вивести з нього, порушивши одне або декілька необхідних умов його існування. При цьому кілька процесів втратять (повністю або частково) виконану роботу. Однак, це дешевше, ніж залишати систему в тупиковій ситуації. Виникають наступні складності відновлення системи:

1) У перший момент може бути неочевидно, що система потрапила в тупикову ситуацію.

2) У більшості систем немає досить ефективних засобів, що дозволяють призупинити процес на невизначено довгий час, вивести його із системи й відновити згодом. Деякі процеси (наприклад, процеси реального часу) повинні працювати безупинно й не допускають призупинення й наступного відновлення.

3) Якщо в системі існують ефективні засоби призупинення/поновлення, то їх використання вимагає значних витрат машинного часу й уваги висококваліфікованого оператора.

4) Відновлення після тупика невеликих масштабів вимагає й невеликої кількості роботи; великий тупик може вимагати досить великого обсягу робіт по його виключенню із системи.

У сучасних системах відновлення зазвичай виконують шляхом примусового виводу одного або декількох процесів із системи, щоб їх ресурси могли бути використані. Тоді ці процеси втрачаються разом з інформацією в них, однак процеси, що залишилися, одержують можливість завершити свою роботу. Тобто у цьому випадку кілька процесів просто вбиваються.

При видаленні процесів можуть виникнути проблеми із пріоритетністю (видаляються зазвичай найменш пріоритетні процеси):

1) Процеси можуть не мати конкретних пріоритетів.

2) Значення пріоритетів можуть виявитися неправильними або можуть бути порушені (наприклад, для процесу, що нескінченно відкладається, збільшується пріоритет).

3) Труднощі в оптимальному визначенні, які з процесів вивести із системи.

Найефективніший спосіб відновлення після тупиків —**механізм призупинення/поновлення**. Він дозволяє короткочасно переводити процеси в стан очікування, а потім активізувати процеси, що очікують, причому без втрати інформації.

Часто при тупикових ситуаціях потрібен перезапуск (рестарт) системи. При цьому існує **перезапуск спочатку** (уся отримана до рестарту інформація втрачається) або **з контрольної точки** (втрачається тільки інформація після контрольної точки). Система не визначає контрольні точки (їх повинен передбачити програміст (для конкретного процесу)).

1.17. Синхронізація процесів

Активізовані в системі процеси можуть виконуватися паралельно, якщо їм для виконання не потрібно тих самих ресурсів. Однак, це досить рідка ситуація в системі. У загальному випадку кілька процесів можуть виконуватися паралельно лише деякий малий час. В інший час вони синхронізуються або взаємодіють один з одним. **Взаємодія** — це передача даних від одного процесу до іншого. **Синхронізація** необхідна для коректності передачі даних або для звертання до **загального ресурсу** (ЗР).

Синхронізація *при передачі даних* полягає в наступному: поки процес — передавач не сформує передані дані й не перешле їх, процес — приймач не повинен виконувати ніяких дій, тобто він повинен очікувати пересилання даних, і тільки тоді він зможе продовжити своє виконання. Процес — приймач може й сам створити процес — передавач, наприклад, при

виникненні необхідності введення даних. При цьому процес — приймач блокується й передає керування дочірньому процесу, по закінченню виконання якого він може продовжити (відновити) своє виконання. При реалізації такого виду синхронізації необхідно визначити порядок передування в часі деяких точок трас процесів і визначити умову, що дозволяє перехід деяких точок трас на певні процеси. Ці виділені точки називаються **точками синхронізації.** Якщо два процеси, що виконуються, підійшли до точки синхронізації, то необхідно визначити, хто з них передавач і хто приймач; або необхідно зафіксувати момент, коли створюється й запускається дочірній процес (передається керування від батьківського).

При *звертанні до СР* необхідна синхронізація, для того щоб виключити одночасний доступ до СР відразу декількох процесів. Тут виникає завдання взаємного виключення, коли тільки один процес може в цей момент "захопити" СР, а інші повинні очікувати. **Критична ділянка (КД)—** це частина процесу, де відбувається звертання до СР. Процес, який у цей момент має СР, перебуває в КД. У період знаходження в КД процес є монополістом, що використовує ОР. Тому бажане, щоб процес якнайшвидше проходив свою КД й не блокувався в ньому, інакше може скластися ситуація, коли декілька менш пріоритетних процесів будуть нескінченно довго очікувати виділення СР. При виході процесу з КД, вхід туди може бути дозволено одному з процесів, які очікують у черзі до СР. Це також свого роду синхронізація. Таким чином, процеси, що звертаються до СР можуть виконуватися лише послідовно.

- **Примітиви синхронізації**

Для синхронізації процесів використовуються примітиви. Обробка їх реалізована в ядрі для синхронізації введення/виводу, перемикання контексту, переходу по перериванню і т.д.

Примітиви синхронізації можуть бути реалізовані в такий спосіб:

1. Найпростіші примітиви —**прапорці** (примітиви взаємовиключення). Якщо прапорець рівний 0, тобто умова хибна, то процес не може ввійти в КД, тому що там уже перебуває інший процес; якщо прапорець рівний 1 (умова є істинною), те процес входить у КД, обнуляючи прапорець (якщо є черга процесів до СР, то в КД входить найбільш пріоритетний з них). На основі прапорців реалізовані алгоритми синхронізації Деккера. Прапорці – програмна реалізація розв'язку задачі взаємного виключення. Це найнижчий рівень.

2. **Команда test_and_set**— апаратний засіб синхронізації (більш високий рівень). Це спеціальна команда, що виконує 3 дії:

- читання значення зміної;
- запис її значення в область збереження;
- установка нового значення цієї змінної;

3. **Семафор** — деяка (захищена) змінна, значення якої можна зчитувати й змінювати тільки за допомогою спеціальних операцій P і V. Перевага в порівнянні з test_and_set – поділ дій на окремі атомарні операції.

Операція **P(S)**: перевіряє значення семафора S; якщо S>0, то S:=S-1, інакше (S=0) чекати (по S) .

Операція **V(S)**: перевіряє, чи їсти процеси, припинені по даному семафору; якщо є, то дозволити одному з них продовжити своє виконання (увійти в КД); якщо ні, те S:=S+1.

Операція P повинна стояти до входу в КД, а операція V – після виходу з КД.

Недоліки:

1) громіздкість (у кожному процесі кожна КД повинна бути оздоблена операціями P і V;

2) неможливість розв'язку цілого ряду задач за допомогою таких примітивів.

4. **Монітор** — примітив високого рівня. Тут "забір" ставиться навколо СР, що зручніше (ніж семафори), тому що ресурсів у кілька разів менше, ніж

процесів (їх КД), а ставити "забори" навколо КД занадто громіздко. Можна сказати, що монітор — це деякий програмний модуль, у якому об'єднані СР і підпрограми для роботи з ними. При звертанні до СР, тобто відповідній процедурі монітора для роботи з СР, здійснюється вхід у монітор. У кожний конкретний момент часу може виконуватися тільки одна процедура монітора — це і є розв'язок задачі взаємного виключення. Якщо процес звернувся до процедури монітора, а інший процес уже перебуває усередині монітора, то процес, що звернувся, блокується й переводиться в зовнішню чергу процесів - блокованих по входу в монітор. Процес, що ввійшов у монітор (тобто виконуючий його процедуру), також може бути блокований їм, якщо не виконується деяка *умова* X для виконання процесу. Умова X у моніторі - це деяка змінна типу condition. З нею зв'язується деяка внутрішня черга процесів, блокованих за умовою X. Внутрішня черга пріоритетніше зовнішньої. Для блокування процесу по змінній X і розміщення його у внутрішній черзі по цій змінній використовується операція монітора WAIT(X). При виконанні операції SIGNAL(X) із черги, пов'язаної зі змінною X, вилучається й розблоковується перший процес. Якщо внутрішня черга порожня, то в монітор дозволяється ввійти першому процесу із зовнішньої черги.

1.18. ЗАСОБИ МІЖПРОЦЕСНОЇ ВЗАЄМОДІЇ У БАГАТОПРОГРАМНИХ ОБЧИСЛЮВАЛЬНИХ СИСТЕМАХ

Процеси виконуються у власному адресному просторі й по суті ізольовані друг від друга. Тим самим зведені до мінімуму можливість впливу процесів один на одного, що є необхідною умовою в багато задачних операційних системах. Однак від одиночного ізольованого процесу мало користі. Сама концепція багатопрограмних операційних систем полягає в модульності , тобто заснована на взаємодії між окремими процесами.

Розглянемо методи й способи організації міжпроцесної взаємодії на прикладі їх реалізації в ОС UNIX.

Для реалізації взаємодії потрібно забезпечити засоби взаємодії між процесами й одночасно виключити небажаний вплив одного процесу на іншій.

Взаємодія між процесами необхідна для розв'язку наступних задач:

- *Передача даних.* Один процес передає дані іншому процесу, при цьому їх обсяг може варіюватися від десятків байтів до декількох мегабайтів.
- *Спільне використання даних.* Замість копіювання інформації від одного процесу до іншого, процеси можуть спільно використовувати одну копію даних, причому зміни, зроблені одним процесом, будуть відразу ж помітні для іншого. Кількість взаємодіючих процесів може бути більше двох. При спільному використанні ресурсів процесам може знадобитися деякий протокол взаємодії для збереження цілісності даних і виключення конфліктів при доступі до них.
- *Сповіщення.* Процес може сповістити інший процес або групу процесів про настання деякої події. Це може знадобитися, наприклад, для синхронізації виконання декількох процесів.

Очевидно, що вирішувати дану задачу засобами самих процесів неефективно, а в рамках багато задачної системи — це небезпечно й тому неможливо. Таким чином, операційна система повинна забезпечити механізми міжпроцесної взаємодії (Inter-Process Communication, IPC).

До засобів міжпроцесної взаємодії можна віднести:

- сигнали
- канали
- FIFO (іменовані канали)
- повідомлення (черги повідомлень)
- семафори
- поділювану пам'ять
- сокети (гнізда)

1.18.1. Сигнали

Сигнали від початку були запропоновані як засіб повідомлення про помилки, але можуть використовуватися й для елементарного IPC, наприклад, для синхронізації процесів або для передачі найпростіших команд від одного процесу до іншого. Однак використання сигналів як засобу IPC обмежене через те, що сигнали дуже ресурсномісткі. Відправлення сигналу вимагає виконання системного виклику, а його доставка — переривання процесу-одержувача й інтенсивних операцій зі стеком процесу для виклику функції обробки й продовження його нормального виконання. При цьому сигнали слабо інформативні і їх число досить обмежене.

У деякому змісті сигнали забезпечують найпростішу форму міжпроцесної взаємодії, дозволяючи повідомляти процес або групу процесів про настання деякої події.

❖ **Керування сигналами**

Сигнали забезпечують механізм виклику певної процедури при настанні деякої події. Кожна подія має свій ідентифікатор і символьну константу. Деякі із цих подій мають асинхронний характер, наприклад, коли користувач натискає клавішу <*Del*>або <*Ctrl*>+<*C*> для завершення виконання процесу, інші є повідомленням про помилки й особливі ситуації, наприклад, при спробі доступу до неприпустимої адреси або виклики неприпустимої інструкції. Кажучи про сигнали, необхідно розрізняти дві фази цього механізму: генерація або відправлення сигналу і його доставка й обробка. Сигнал відправляється, коли відбувається певна подія, про настання якої повинен бути сповіщений процес. Сигнал вважається доставленим, коли процес, якому був відправлений сигнал, одержує його й виконує його обробку. У проміжку між цими двома моментами сигнал очікує доставки.

❖ **Відправлення сигналу**

Ядро генерує й відправляє процесу сигнал у відповідь на ряд подій, які можуть бути викликані самим процесом, іншим процесом, перериванням або якими-небудь зовнішніми подіями. Можна виділити основні причини відправлення сигналу:

Особливі ситуації	Коли виконання процесу викликає особливу ситуацію, наприклад, поділ на нуль, процес одержує відповідний сигнал.
Термінальні *переривання*	Натискання деяких клавіш термінала, наприклад, *<Del>, <Ctrl>+<C>або<Ctrl>+<\>,* викликає відправлення сигналу поточному процесу, пов'язаному з терміналом.
Інші процеси	Процес може відправити сигнал іншому процесу або групі процесів за допомогою системного виклику *kill(2).* У цьому випадку сигнали є елементарною формою міжпроцесної взаємодії.
Керування завданнями	Командні інтерпретатори, що підтримують систему керування завданнями, використовують сигнали для маніпулювання фоновими і поточними завданнями. Коли процес, що виконується у фоновому режимі, робить спробу читання або запису на термінал, йому відправляється сигнал останову. Коли дочірній процес завершує свою роботу, батько повідомляється про це також за допомогою сигналу
Квоти	Коли процес перевищує виділену йому квоту обчислювальних ресурсів або ресурсів файлової системи, йому відправляється відповідний сигнал.
Повідомлення	Процес може запросити повідомлення про настання тих або інших подій, наприклад, готовності пристрою і т.д. Таке повідомлення відправляється процесу у вигляді сигналу.
Алармы	Якщо процес установив таймер, йому буде відправлений сигнал, коли значення таймера стане рівним нулю

❖ **Доставка й обробка сигналу**

Для кожного сигналу в системі визначена обробка за замовчуванням, яку виконує ядро, якщо процес не вказав іншої дії. У загальному випадку існують наступні можливі дії на появу сигналу: завершити виконання, ігнорувати сигнал, зупинити процес і продовжити процес. Процес може змінити дію обробки сигналу за замовчуванням, зареєструвавши власний оброблювач сигналу, або указати, що сигнал слід ігнорувати. Процес також може заблокувати сигнал, відклавши на якийсь час його обробку. Це можливо не для всіх сигналів. Наприклад, для сигналів SIGKILL і SIGSTOP єдиною дією є дія за замовчуванням, ці сигнали не можна ні перехопити, ні заблокувати, ні ігнорувати. Для ряду сигналів, переважно пов'язаних з апаратними помилками й особливими ситуаціями, обробка, відмінна від дій за замовчуванням, не рекомендується, тому що може призвести до непередбачених (для процесу) результатам.

Слід помітити, що будь-яка обробка сигналу, у тому числі обробка за замовчуванням, передбачає, що процес активний. На системах з високим завантаженням це може призвести до істотних затримок між відправленням і доставкою сигналу, тому що процес не отримає сигнал, поки не буде обраний планувальником, і йому не будуть надані обчислювальні ресурси. Доставка сигналу відбувається після того, як ядро від імені процесу викликає системну процедуру issig(), яка перевіряє, чи існують сигнали, що чекають доставки, адресовані даному процесу. Функція issig() викликається ядром у трьох випадках:

1. Безпосередньо перед поверненням процесу з режиму ядра в режим задачі після обробки системного виклику або переривання.

2. Безпосередньо перед переходом процесу в стан сну із пріоритетом, що допускає переривання сигналом.

3. Відразу ж після пробудження після сну із пріоритетом, що допускає переривання сигналом.

Якщо процедура issig() виявляє сигнали, що очікують доставки, ядро викликає функцію доставки сигналу, яка виконує дії за замовчуванням або викликає спеціальну функцію sendsig(), що запускає оброблювач сигналу, зареєстрований процесом. Функція sendsig() повертає процес у режим завдання, передає керування оброблювачеві сигналу, а потім відновлює контекст процесу для продовження перерваного сигналом виконання процесу.

Розглянемо типові ситуації, пов'язані з відправленням і доставкою сигналів. Допустимо, користувач, працюючи за терміналом, натискає клавішу переривання (<Del> або <*Ctrl*>+<*C*> для більшості систем). Натискання будь-якої клавіші викликає апаратне переривання (наприклад, переривання від послідовного порту), а драйвер термінала при обробці цього переривання визначає, що була натиснута спеціальна клавіша, що генерує сигнал, і відправляє поточному процесу, пов'язаному з терміналом, сигнал SIGINT. Коли процес буде обраний планувальником і запущений на виконання, при переході в режим задачі він виявить надходження сигналу й обробить його. Якщо ж у момент генерації сигналу термінальним драйвером процес, якому був адресований сигнал, уже виконувався (тобто був перерваний оброблювачем термінального переривання), він також обробить сигнал при поверненні в режим задачі після обробки переривання.

Робота із сигналами, пов'язаними з особливими ситуаціями, незначно відрізняється від вищеописаної. Особлива ситуація виникає при виконанні процесом певної інструкції, що викликає в системі помилку (наприклад, поділ на нуль, звертання до неприпустимої області пам'яті, неприпустима інструкція або виклик і т.д.). Якщо таке відбувається, викликається системний оброблювач особливої ситуації, і процес переходить у режим ядра, майже так само, як і при обробці будь-якого іншого переривання. Оброблювач відправляє процесу відповідний сигнал, який доставляється, коли виконання повертається в режим задачі.

При переході процесу в стан сну виділяють дві категорії подій, пов'язаних з наявністю сигналу: ті, що допускають переривання сигналом, і ті, що не допускають такого переривання. В останньому випадку сигнал буде терпляче очікувати нормального пробудження процесу, наприклад, після завершення операції дискового введення/виводу.

У першому випадку, доставка сигналу буде перевірена ядром безпосередньо перед переходом процесу в стан сну. Якщо такий сигнал надійшов, буде викликаний оброблювач сигналу, а системний виклик, який виконувався процесом, буде аварійно завершений з помилкою EINTR. Якщо генерація сигналу відбулася протягом сну процесу, ядро буде змушено розбудити його й зняти перерваний системний виклик (помилка EIHTR). Після пробудження процесу або внаслідок одержання сигналу, або через настання очікуваного події, ядром буде викликана функція issig(), яка виявить наджодження сигналу й викличе відповідну обробку.

1.18.2. Канали

Синтаксис організації програмних каналів при роботі в командному рядку shell: **cat myfile | wc**

При цьому (стандартний) вивід програми *cat(1),* яка виводить уміст файлу **myfile,** передається на (стандартне) уведення програми *wc(1),* яка, у свою чергу підраховує кількість рядків, слів і символів. У результаті ми одержимо щось на зразок: 12 45 260, що буде означати кількість рядків, слів і символів у файлі **myflle.**

Таким чином, два процеси обмінялися даними. При цьому використовувався програмний канал, що забезпечує *односпрямовану передачу даних* між двома задачами.

Для створення каналу використовується системний виклик *pipe(2):*

int pipe (int *filedes) ;

Він повертає два файлові дескриптори — fildes [0] для запису в канал і fildes [1] для читання з каналу. Тепер, якщо один процес записує дані в fildes

[0], інший зможе одержати ці дані з fildes [1]. Питання тільки в тому, як інший процес зможе одержати сам файловий дескриптор fildes [1]?

Для цього використовуються наслідувані атрибути при створенні процесу. Дочірній процес успадковує й розділяє всі призначені файлові дескриптори батьківського. Тобто доступ до дескрипторів fildes каналу може одержати сам процес, що викликав *pipe(2),* і його дочірні процеси. У цьому полягає серйозний недолік каналів, оскільки вони можуть бути використані для передачі даних тільки між родинними процесами. Канали не можуть використовуватися в якості засобу міжпроцесної взаємодії між незалежними процесами (рис. 2.20).

Хоча в наведеному прикладі може здатися, що процеси *cat(1)* і *wc(1)* незалежні, насправді обоє цих процесу створюються процесом shell і є родинними.

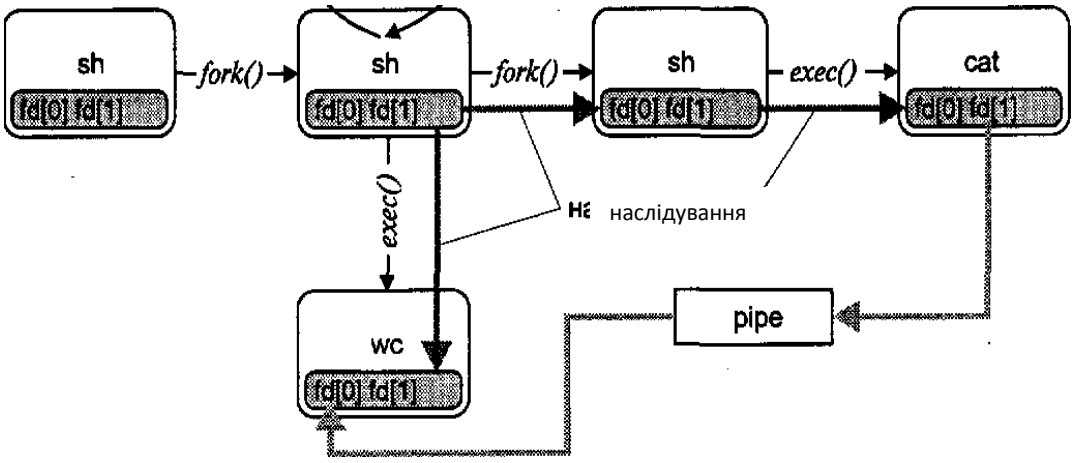


Рис. 1.20 Створення каналу між задачами *cat(1)* і *wc(1)*

1.18.2. FIFO (іменовані канали)

Назва каналів FIFO походить від виразу First In First Out (перший увійшов — перший вийшов). FIFO дуже схожі на канали, оскільки є односпрямованим засобом передачі даних, причому читання даних відбувається в порядку їх запису. Однак на відміну від програмних каналів, FIFO мають імена, які дозволяють незалежним процесам одержати до цих об'єктів доступ. Тому іноді FIFO також називають *іменованими каналами.* FIFO є окремим типом файлу у файловій системі UNIX. Для створення FIFO використовується системний виклик *mknod(2):*

int mknod(char *pathname, int mode, int dev) ;

де pathname — ім'я файлу у файловій системі (ім'я FIFO),

mode — прапори володіння, прав доступу і т.д.,

dev — при створенні FIFO ігнорується.

FIFO може бути створений і з командного рядка shell:

$ mknod *name* p

Після створення FIFO може бути відкритий на запис і читання, причому запис і читання можуть відбуватися в різних незалежних процесах.

Канали FIFO і звичайні канали працюють за наступними правилами:

1. При читанні меншого числа байтів, ніж перебуває в каналі або FIFO, вертається необхідне число байтів, залишок зберігається для наступних читань.

2. При читанні більшого числа байтів, ніж перебуває в каналі або FIFO, вертається доступне число байтів. Процес, що читає з каналу, повинен відповідним чином обробити ситуацію, коли прочитано менше, чим замовлено.

3. Якщо канал порожній і жоден процес не відкрив його на запис, при читанні з каналу буде отримано 0 байтів. Якщо один або більш процесів відкрили канал для запису, виклик *read(2)* буде заблокований до появи даних (якщо для каналу або FIFO не встановлений прапор відсутності блокування O_NDELAY).

4. Запис числа байтів, меншого ємності каналу або FIFO, гарантовано атомарно. Це означає, що у випадку, коли кілька процесів одночасно записують у канал, порції даних від цих процесів не перемішуються.

5. При записі більшого числа байтів, ніж це дозволяє канал або FIFO, виклик *write(2)* блокується до звільнення необхідного місця. При цьому атомарність операції не гарантується. Якщо процес намагається записати дані в канал, не відкритий жодним процесом на читання, процесу генерується сигнал SIGPIPE, а виклик *write(2)* повертає 0 з установкою помилки

(errno=EPIPE) (якщо процес не встановив обробки сигналу SIGPIPE, проводиться обробка за замовчуванням — процес завершується).

1.18.3. Ідентифікатори й імена в IPC

Відсутність імен у каналів робить їхніми недоступними для незалежних процесів. Цей недолік усунутий в FIFO, які мають імена. Інші засоби міжпроцесної взаємодії, що є більш складними, вимагають додаткових угод за іменами й ідентифікаторами. Множина можливих імен об'єктів конкретного типу міжпроцесної взаємодії називається *простором імен* (name space). Імена є важливим компонентом системи міжпроцессної взаємодії для всіх об'єктів, крім каналів, оскільки дозволяють різним процесам одержати доступ до загального об'єкта. Так, іменем FIFO є ім'я файлу іменованого каналу. Використовуючи обумовлене ім'я створеного FIFO, два процеси можуть звертатися до цього об'єкта для обміну даними.

Для таких об'єктів IPC, як черги повідомлень, семафори й поділювана пам'ять, процес призначення імені є більш складним, ніж проста вказівка імені файлу. Ім'я для цих об'єктів називається *ключем* (key) і генерується функцією *ftok(3C)* із двох компонентів — імені файлу й ідентифікатора проекту:

```
#include <sys/types.h>
#include <sys/ipc.h>
key_t ftok(char *filename, char proj);
```

У якості filename можна використовувати ім'я деякого файлу, відоме взаємодіючим процесам. Наприклад, це може бути ім'я програми-сервера. Важливо, щоб цей файл існував на момент створення ключа. Також небажане використовувати ім'я файлу, який створюється й видаляється в процесі роботи розподіленої програми, оскільки при генерації ключа використовується номер inode файлу. Знову створений файл може мати інший inode і згодом процес, що бажає мати доступ до об'єкта, одержить невірний ключ.

Простір імен дозволяє створювати й спільно використовувати IPC не родинним процесам. Однак для посилань на вже створені об'єкти використовуються ідентифікатори, точно так само, як файловий дескриптор використовується для роботи з файлом, відкритим по імені.

Кожне з перерахованих IPC має свій унікальний дескриптор (ідентифікатор), використовуваний ОС (ядром) для роботи з об'єктом. Унікальність дескриптора забезпечується унікальністю дескриптора для кожного з типів об'єктів (черги повідомлень, семафори й поділювана пам'ять), тобто яка-небудь черга повідомлень може мати той же чисельний ідентифікатор, що й поділювана область пам'яті (хоча будь-які дві черги повідомлень повинні мати різні ідентифікатори).

Робота з об'єктами IPC System V багато в чому схожа на роботу з файлами в UNIX. Однією з відмінностей є те, що файлові дескриптори мають значимість у контексті процесу, у той час як значимість дескрипторів об'єктів IPC поширюється на всю систему. Так файловий дескриптор 3 одного процесу в загальному випадку ніяк не пов'язаний з дескриптором 3 іншого неспорідненого процесу (тобто ці дескриптори посилаються на різні файли). Інакше виглядає справа з дескрипторами об'єктів IPC. Усі процеси, що використовують, скажемо, одну чергу повідомлень, одержать однакові дескриптори цього об'єкта.

Для кожного з об'єктів IPC ядро підтримує відповідну структуру даних, відмінну для кожного типу об'єкта IPC (черги повідомлень, семафора або поділюваної пам'яті). Загальною в цих даних є структура ipc_perm, що описує права доступу до об'єкта, подібно тому, як це робиться для файлів.

Права доступу (як і для файлів) визначають можливі операції, які може виконувати над об'єктом конкретний процес (одержання доступу до існуючого об'єкта, читання, запис і видалення).

Помітимо, що система не видаляє створені об'єкти IPC навіть тоді, коли жоден процес не користується ними. Видалення створених об'єктів є обов'язком процесів, яким для цього надаються відповідні функції керування

msgctl(2), semctl(2), shmctl(2). За допомогою цих функцій процес може одержати й установити ряд полів внутрішніх структур, підтримуваних системою для об'єктів IPC, а також вилучити створені об'єкти. Безумовно, як і в багатьох інших випадках використання об'єктів IPC процеси попередньо повинні "домовитися", який процес і коли вилучить об'єкт. Найчастіше, таким процесом є сервер.

1.18.4. Повідомлення

Черги повідомлень є складовою частиною операційної системи, вони обслуговуються операційною системою, розміщаються в адресному просторі ядра і є поділюваним системним ресурсом. Кожна черга повідомлень має свій унікальний ідентифікатор. Процеси можуть записувати й зчитувати повідомлення з різних черг. Процес, що послав повідомлення в чергу, може не очікувати читання цього повідомлення яким-небудь іншим процесом. Він може закінчити своє виконання, залишивши в черзі повідомлення, яке буде прочитано іншим процесом пізніше.

Дана можливість дозволяє процесам обмінюватися структурованими даними, що мають наступні атрибути:

- Тип повідомлення (дозволяє мультиплексувати повідомлення в одній черзі).
- Довжина даних повідомлення в байтах (може бути нульовою).
- Власне дані (якщо довжина ненульова, можуть бути структурованими).

Черга повідомлень зберігається у вигляді внутрішнього односпрямованого зв'язаного списку в адресному просторі ядра. В адресному просторі ядра є таблиця черг повідомлень, у якій відслідковуються всі черги повідомлень, створювані в системі. Для кожної черги повідомлень у таблиці повідомлень ядро створює структуру даних (msqid_ds), де міститься інформація про права доступу до черги (msg_perm), її поточний стан: msg_cbytes — число байтів, msg qnum — число повідомлень у черзі, вказівники на перше msg_first і

останнє (msg last) повідомлення, що зберігаються у вигляді зв'язаного списку. Кожний елемент цього списку є окремим повідомленням. Коли процес передає повідомлення в чергу, ядро створює для нього новий запис зі структурою struct_msg і поміщає її в кінець зв'язаного списку записів відповідних повідомлень зазначеної черги. У кожному такому записі вказується тип повідомлення, число байтів даних, що містяться в повідомленні, і вказівник на іншу область даних ядра (msg_spot), де фактично перебувають дані повідомлення.

На рис. 2.21 зображена структура даних ядра, що використовується для маніпулювання чергою повідомлень.

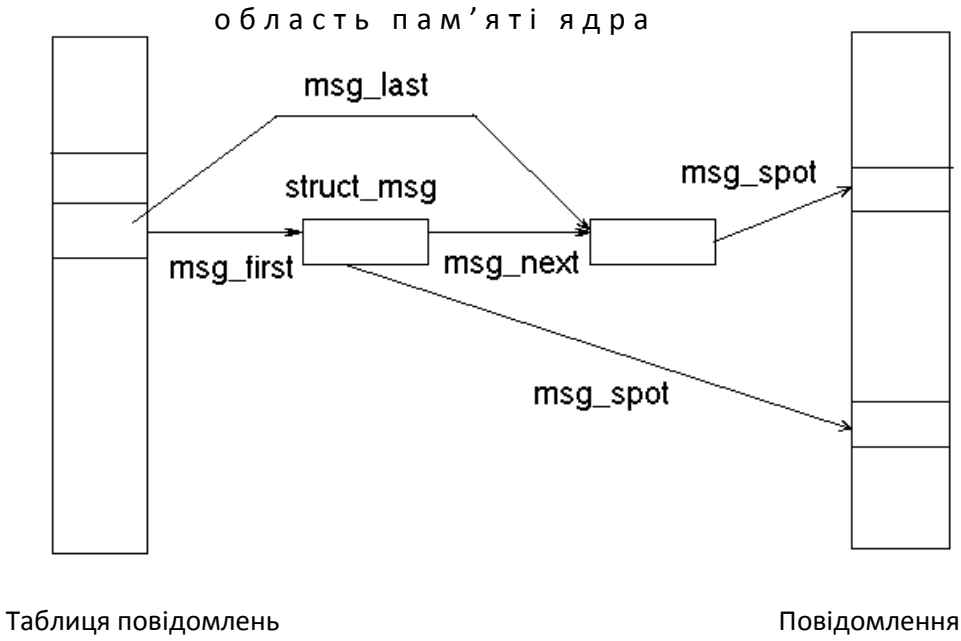


Рис. 1.21 Структура даних ядра, що використовуються для маніпулювання чергою повідомлень

Для створення нової черги повідомлень або для доступу до існуючої використовується системний виклик *msgget(2}:*

```
#include <sys/types,h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msggett key_t key, int msgflag);
```

Функція повертає дескриптор об'єкта - черги, або -1 у випадку помилки. Подібно файловому дескриптору, цей ідентифікатор використовується процесом для роботи із чергою повідомленнь. Зокрема, процес може:

- Поміщати в чергу повідомлення за допомогою функції *msgsnd(2);*
- Отримувати повідомлення визначеного типу із черги за допомогою функції *msgrcv(2);*
- Керувати повідомленнями за допомогою функції *msgctl(2).*

Черги повідомлень мають досить корисну властивість — в одній черзі можна мультиплексувати повідомлення від різних процесів. Для демультиплексування використовується атрибут msgtype, на підставі якого будь-який процес може фільтрувати повідомлення за допомогою функції *msgrcv(2),* як це було показано вище.

Розглянемо типову ситуацію взаємодії процесів, коли серверний процес обмінюється даними з декількома клієнтами. Властивість мультиплексування дозволяє використовувати для такого обміну одну чергу повідомлень. Для цього повідомленням, що направляються від кожного із клієнтів серверу, будемо привласнювати значення типу, скажемо, рівним 1. Якщо в тілі повідомлення клієнт яким-небудь чином ідентифікує себе (наприклад, передає свій PID), то сервер зможе передати повідомлення конкретному клієнтові, привласнюючи тип повідомлення рівним цьому ідентифікатору.

Оскільки функція *msgrcv(2)* дозволяє ухвалювати повідомлення певного типу (типів), сервер буде ухвалювати повідомлення з типом 1, а клієнти — повідомлення з типами, рівними ідентифікаторам їх процесів. Схема такої взаємодії представлена на мал. 1.22.

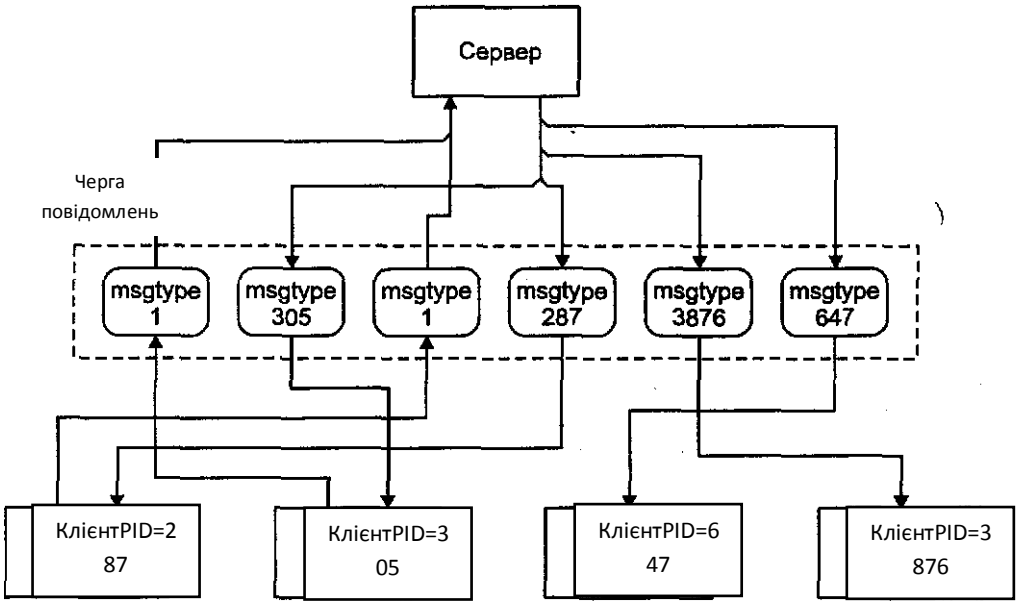


Рис. 1.22 Мультиплексування повідомлень в одній черзі

Атрибут msgtype також можна використовувати для зміни порядку вилучення повідомлень із черги. Стандартний порядок одержання повідомлень аналогічний принципу FIFO — повідомлення отримуються у порядку їх запису. Однак використовуючи тип, наприклад, для призначення пріоритету повідомлень, цей порядок легко змінити.

1.18.5. Семафори

Для синхронізації процесів, а точніше, для синхронізації доступу декількох процесів до поділюваних ресурсів, як уже згадувалося, використовуються *семафори.* Будучи однієї з форм IPC, семафори не призначені для обміну великими обсягами даних, як у випадку FIFO або черг повідомлень. Замість цього, вони виконують функцію, повністю відповідну до своєї назви — дозволяти або забороняти процесу використання того або іншого поділюваного ресурсу.

Застосування семафорів пояснимо на простому прикладі. Припустимо, є якийсь поділюваний ресурс (наприклад, файл). Необхідно блокувати доступ до ресурсу для інших процесів, коли якийсь процес робить операцію над ресурсом (наприклад, записує у файл). Для цього зв'яжемо з даним ресурсом якусь ціло чисельну величину — лічильник, доступний для всіх процесів. Приймемо, що значення 1 лічильника означає доступність ресурсу, 0 — його

неприступність. Тоді перед початком роботи з ресурсом процес повинен перевірити значення лічильника. Якщо воно дорівнює 0 — ресурс зайнятий, і операція неприпустима — процесу залишається чекати. Якщо значення лічильника рівно 1 — можна працювати з ресурсом. Для цього, насамперед, необхідно заблокувати ресурс, тобто змінити значення лічильника на 0. Після виконання операції для звільнення ресурсу значення лічильника необхідно змінити на 1. У наведеному прикладі лічильник відіграє роль семафора.

Для нормальної роботи необхідно забезпечити виконання наступних умов:

1. Значення семафора повинне бути доступно різним процесам. Тому семафор перебуває не в адресному просторі процесу, а в адресному просторі ядра супервізора.

2. Операція перевірки й змінм значення семафора повинна бути реалізована у вигляді однієї атомарної стосовно інших процесів (тобто не перериваємою іншими процесами) операції. Інакше можлива ситуація, коли після перевірки значення семафора виконання процесу буде перервано іншим процесом, який у свою чергу перевірить семафор і змінить його значення. Єдиним способом гарантувати атомарність критичних ділянок операцій є виконання цих операцій у режимі ядра (див. режими виконання процесу).

Таким чином, семафори є системним ресурсом, дії над яким проводяться через інтерфейс системних викликів.

Семафори в System V мають наступні характеристики:

- Семафор являє собою не один лічильник, а групу, що складається з декількох лічильників, об'єднаних загальними ознаками (наприклад, дескриптором об'єкта, правами доступу і т.д.).
- Кожне із цих чисел може приймати будь-яке ненегативне значення в межах, визначених системою (а не тільки значення 0 і 1).

Крім власне значення семафора, у структурі sem зберігається ідентифікатор процесу, що викликав останню операцію над семафором, число процесів, що очікують збільшення значення семафора, і число процесів, що очікують, коли значення семафора стане рівним нулю. Ця

інформація дозволяє ядру здійснювати операції над семафорами, які ми обговоримо трохи пізніше. Для одержання доступу до семафора (і для його створення, якщо він не існує) використовується системний виклик *semget(2):*

У випадку успішного завершення операції функція повертає дескриптор об'єкта, у випадку невдачі — -1. Аргумент nsems задає число семафорів у групі. У випадку, коли ми не створюємо, а лише одержуємо доступ до існуючого семафора, цей аргумент ігнорується. Аргумент semflag визначає права доступу до семафора й прапорці для його створення (IPC_CREAT, IPC_EXCL).

Після одержання дескриптора об'єкта процес може здійснювати операції над семафором, подібно тому, як після одержання файлового дескриптора процес може читати й записувати дані у файл. Для цього використовується системний виклик *semop(2):*

У якості другого аргументу функції передається покажчик на структуру даних, що визначає операції, які потрібно здійснити над семафором з дескриптором semid. Операцій може бути декілька, і їхнє число вказується в останньому аргументі nops. Важливо, що ядро забезпечує атомарність виконання критичних ділянок операцій (наприклад, перевірка значення — зміна значення) стосовно інших процесів.

UNIX допускає три можливі операції над **семафором,** обумовлені полем semop:

1. Якщо величина semop позитивна, то поточне значення семафора збільшується на цю величину.

2. Якщо значення semop дорівнює нулю, процес очікує, поки семафор не обнулюється.

3. Якщо величина semop негативна, процес очікує, поки значення семафора не стане більшим або рівним абсолютній величині semop. Потім абсолютна величина semop віднімається зі значення семафора.

Можна помітити, що перша операція змінює значення семафора (безумовне виконання), друга операція тільки перевіряє його значення

(умовне виконання), а третя — перевіряє, а потім змінює значення семафора (умовне виконання).

При роботі із семафорами взаємодіючі процеси повинні домовитися про їхнє використання й кооперативно проводити операції над семафорами. Операційна система не накладає обмежень на використання семафорів. Зокрема, процеси можуть вирішувати, яке значення семафора є дозволяючим, на яку величину змінюється значення семафора й т.п.

Таким чином, при роботі із семафорами процеси використовують різні комбінації із трьох операцій, визначених системою, по-своєму трактуючи значення семафорів.

1.18.6. Поділювана пам'ять

Інтенсивний обмін даними між процесами з використанням розглянутих механізмів міжпроцесної взаємодії (канали, FIFO, черги повідомлень) може викликати падіння продуктивності системи. Це, у першу чергу, пов'язане з тим, що дані, передані за допомогою цих об'єктів, копіюються з буфера передаючого процесу в буфер ядра й потім у буфер приймаючого процесу. Механізм поділюваної пам'яті дозволяє позбутися накладних витрат передачі даних через ядро, надаючи двом або більше процесам можливість безпосереднього одержання доступу до однієї області пам'яті для обміну даними.

Безумовно, процеси повинні попередньо "домовитися" про правила використання поділюваної пам'яті. Наприклад, поки один із процесів робить запис даних у поділювану пам'ять, інші процеси повинні втриматися від роботи з нею. На щастя, задача кооперативного використання поділюваної пам'яті, що полягає в синхронізації виконання процесів, легко вирішується за допомогою семафорів.

Зразковий сценарій роботи з поділюваною пам'яттю виглядає таким чином:

1. Сервер отримує доступ до поділюваної пам'яті, використовуючи семафор.

2. Сервер робить запис даних у поділювану пам'ять.

3. Після завершення запису сервер звільнює поділювану пам'ять за допомогою семафора.

4. Клієнт отримує доступ до поділюваної пам'яті, замикаючи ресурс за допомогою семафора.

5. Клієнт здійснює читання даних з поділюваної пам'яті й звільняє її, використовуючи семафор.

Для кожної області поділюваної пам'яті, ядро підтримує спеціальну структуру, основними полями якої є: права доступу засновника області, розмір поділюваної пам'яті, число процесів, що використовують поділювану пам'ять, часи останнього приєднання, відключення й зміни.

Для створення або для доступу до вже існуючої поділюваної пам'яті використовується системний виклик *shmget(2):*

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmgettkey t key, int size, int s"hmfiag) ;
```

Функція повертає дескриптор поділюваної пам'яті у випадку успіху, і -1 у випадку невдачі. Аргумент size визначає розмір створюваної області пам'яті в байтах. Значення аргументу shmflag задають права доступу до об'єкта й спеціальні прапори IPC_CREAT і IPC_EXCL. Помітимо, що виклик *shmget(2)* лише створює або забезпечує доступ до поділюваної пам'яті, але не дозволяє працювати з нею. Для роботи з поділюваною пам'яттю (читання й запис) необхідно спочатку приєднати (attach) область викликом *shmat(2):*

```
#incluude <sys/types,h>
#include <sys/ipc.h>
#include <sys/shm.h>
char *shmat(int shmid, char *shmaddr, int shmflag);
```

Виклик *shmat(2)* повертає адресу початку області в адресному просторі процесу розміром size, заданим попереднім викликом *shmget(2).* У цьому

адресному просторі взаємодіючі процеси можуть розміщати необхідні структури даних для обміну інформацією. Правила одержання цієї адреси наступні:

1. Якщо аргумент shmaddr нульовий, то система самостійно вибирає адресу.

2. Якщо аргумент shmaddr відмінний від нуля, значення адреси, що повертається, залежить від наявності прапорця SHM RND в аргументі shmflag:

• Якщо прапорець SHM_RND не встановлений, система приєднує поділювану пам'ять до зазначеного shmaddr адресі.

• Якщо прапорець SHM_RND установлений, система приєднує поділювану пам'ять до адреси, отриманої округленням у меншу сторону shmaddr до деякої визначеної величини SHMLBA.

За замовчуванням поділювана пам'ять приєднується із правами на читання й запис. Ці права можна змінити, указавши прапорець SHM_RDONLY в аргументі shmflag.

Таким чином, декілька процесів можуть відображати область поділюваної пам'яті в різні ділянки власного віртуального адресного простору, як це показане на рис. 2.23.

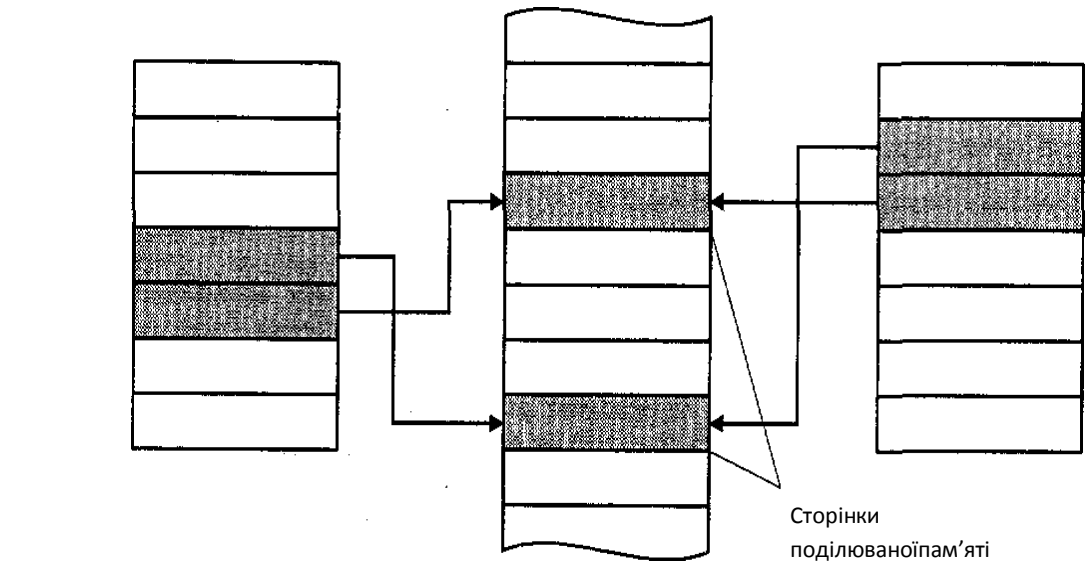


Рис. 1.23 Спільне використання поділюваної пам'яті

Закінчивши роботу з поділюваною пам'яттю, процес відключає (detach) область викликом *shmctt(2):*

```
#incluude <sys/types,h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmd(char *shrnaddr);
```

При роботі з поділюваною пам'яттю необхідно синхронізувати виконання взаємодіючих процесів: коли один із процесів записує дані в поділювану пам'ять, інші процеси очікують завершення операції. Звичайно синхронізація забезпечується за допомогою семафорів, призначення й число яких визначається конкретним використанням поділюваної пам'яті.

1.18.7. Сокети.

Міжпроцесна взаємодія, реалізована одним з вищезазначених способів, має загальний істотний недолік — вони можуть бути реалізовані для процесів, що виконуються на одному комп'ютері.

Розроблювачі нового засобу міжпроцесної взаємодії BSD UNIX керувалися рядом міркувань:

По-перше, взаємодія між процесами повинна бути уніфікована, незалежно від того, чи виконуються вони на одному комп'ютері або на різних машинах мережі. Найбільш оптимальна реалізація міжпроцесної взаємодії, що задовольняє цій вимозі, повинна мати модульну структуру й базуватися на загальній підсистемі підтримки мережі UNIX. При цьому можуть бути використані різні схеми адресації об'єктів, їх розташування, протоколи передачі даних і т.д. У цьому зв'язку було введене поняття *комунікаційний домен* (communication domain), що описує набір позначених характеристик взаємодії.

Для позначення комунікаційного вузла, що забезпечує приймання й передачу даних для об'єкта (процесу), був запропонований спеціальний об'єкт — *сокет* (socket). Сокети створюються в рамках певного комунікаційного домену, подібно тому, як файли створюються в рамках

файлової системи. Сокети мають відповідний інтерфейс доступу у файловій системі, і так само як звичайні файли, адресуються деяким цілим числом — дескриптором. Однак на відміну від звичайних файлів, сокети являють собою віртуальний об'єкт, який існує, поки на нього посилається хоча б один із процесів.

По-друге, комунікаційні характеристики взаємодії повинні бути доступні процесам у деякій уніфікованій формі. Інакше кажучи, програма повина мати можливість вимагати визначений тип зв'язку, наприклад, заснований на віртуальному каналі (virtual circuit) або датаграмах (datagram), причому ці типи повинні бути погоджені для всіх комунікаційних доменів.Усі сокети умовно можна розділити на кілька типів, залежно від надаваних комунікаційних характеристик. Повний набір цих характеристик включає:

- Упорядковану доставку даних
- Відсутність дублювання даних
- Надійну доставку даних
- Збереження границь повідомлень
- Підтримку передачі екстрених повідомлень
- Попереднє встановлення з'єднання

Наприклад, канали, розглянуті раніше, забезпечують тільки перші три характеристики. При цьому дані мають вигляд суцільного потоку, вичленовування повідомлень з якого повинне при необхідності бути забезпечене взаємодіючими програмами.

Підтримка передачі екстрених повідомлень припускає можливість доставки даних поза нормальним потоком. Як правило, це повідомлення, пов'язані з деякими терміновими подіями, що вимагають негайної реакції.

Взаємодія з попереднім установленням з'єднання припускає створення віртуального каналу між джерелом і одержувачем даних. Це рятує від необхідності ідентифікувати передавальну сторону в кожному пакеті даних. Ідентифікація відбувається на початковому етапі встановлення зв'язки й

потім зберігається для всіх пакетів, що належать даному віртуальному каналу.

В BSD UNIX реалізовані наступні основні типи сокетів:

- *Сокет датаграм* (datagram socket), через який здійснюється теоретично ненадійна, незв'язна передача пакетів.
- *Сокет потоку* (stream socket), через який здійснюється надійна передача потоку байтів без збереження границь повідомлень. Цей тип сокетів підтримує передачу екстрених даних.
- *Сокет пакетів* (packet socket), через який здійснюється надійна послідовна передача даних без дублювання з попереднім установленням зв'язку. При цьому зберігаються границі повідомлень.
- *Сокет низького рівня (raw* socket), через який здійснюється безпосередній доступ до комунікаційного протоколу.

Нарешті, для того щоб незалежні процеси мали можливість взаємодіяти один з одним, для сокетів повинно бути визначено *простір імен.* Ім'я сокету має сенс тільки в рамках комунікаційного домену, в якому він створений. Якщо для IPC System V використовуються ключі, то імена сокетів представлені *адресами.*

Для створення сокету процес повинен указати тип сокету й комунікаційний домен, у рамках якого буде використовуватися сокет. Оскільки комунікаційний домен може підтримувати використання декількох протоколів, процес може також указати конкретний комунікаційний протокол для взаємодії. Якщо такий не зазначений, система вибере найбільш підходящий зі списку протоколів, доступних для даного комунікаційного домену. Якщо ж у рамках зазначеного домену створення сокету даного типу неможливо, тобто відсутній відповідний комунікаційний протокол, запит процесу завершиться невдало.

Для створення сокету використовується системний виклик *socket(2)* наступний вид, що має:

```
#include <sys/types.h>
```

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

Тут аргумент domain визначає комунікаційний домен, type — тип сокету, а protocol — використовуваний протокол (може бути не зазначений, тобто прирівняно 0). У випадку успіху системний виклик повертає позитивне ціле число, аналогічне файловому дескриптору, яке служить для адресації даного сокету в наступних викликах.

По суті комунікаційний домен визначає *сімейство протоколів* (protocol family), припустимих у рамках даного домену.

Створення сокету не означає створення комунікаційного вузла. Для однозначної ідентифікації сокету його необхідно позиціонувати в просторі імен даного комунікаційного домену. У загальному випадку кожний комунікаційний канал визначається двома вузлами — джерелом і одержувачем даних, і може бути охарактеризовано п'ятьома параметрами:

1. Комунікаційним протоколом
2. Локальною адресою
3. Локальним процесом
4. Віддаленою адресою
5. Віддаленим процесом

Як правило, адреса визначає вузол мережі, а процес — конкретна програма, що одержує або передає дані. Однак конкретні значення й формат цих параметрів визначаються комунікаційним доменом.

Оскільки при створенні сокету вказується тільки один параметр — комунікаційний протокол, то перш ніж передача даних між взаємодіючими процесами стане можливою, необхідно вказати чотири додаткові параметри для комунікаційного каналу. Очевидно, що взаємодіючі сторони повинні робити це узгоджено, використовуючи або заздалегідь визначені адреси, або домовляючись про них у процесі встановлення зв'язку. Процедура установки цих параметрів істотно залежить і від типу створюваного каналу,

обумовленого типом використовуваного сокету й комунікаційного протоколу.

Ілюстрація взаємодії між процесами при віртуальному комунікаційному каналі з попереднім установленням зв'язку наведена на рис. 1.24, а взаємодія, заснована на датаграмах без установлення зв'язку показане на рис. 1.25.

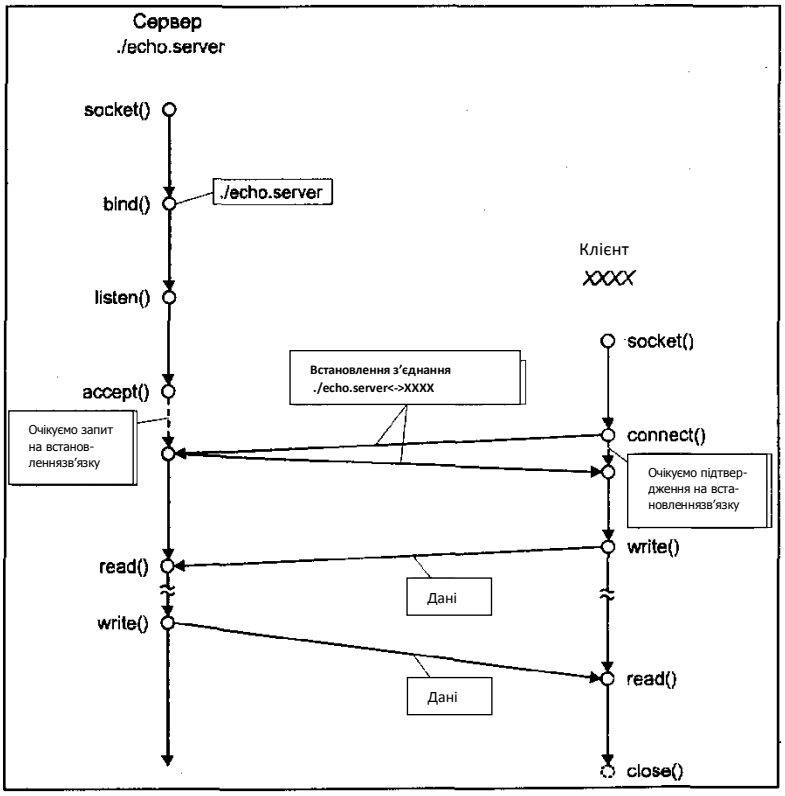


Рис. 1.24. Взаємодія між процесами при створенні віртуального каналу (з попереднім установленням з'єднання)

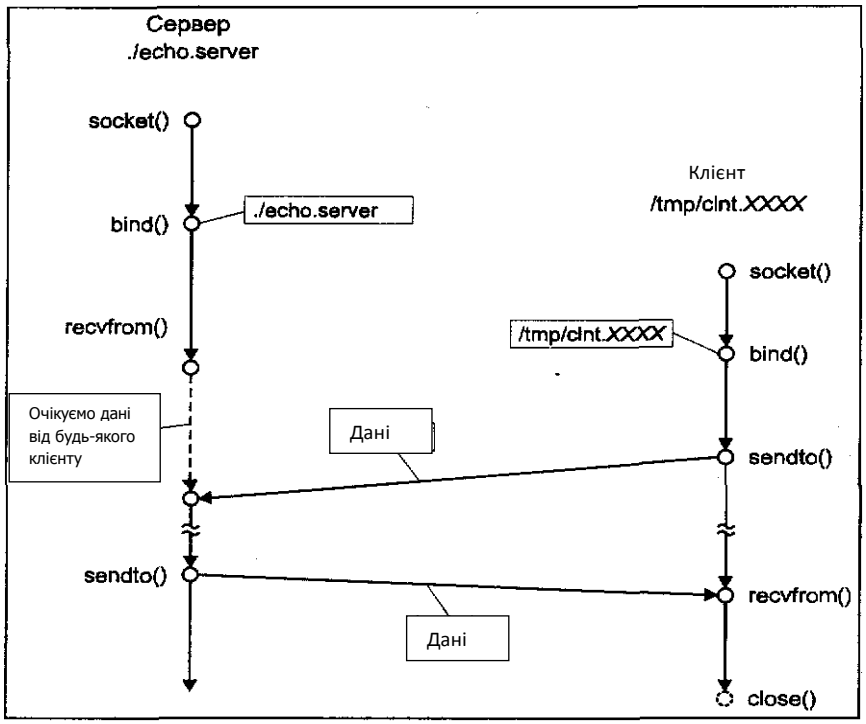


Рис. 1.25 Взаємодія між процесами, заснована на датаграмах (без попереднього встановлення з'єднання)

Як видно з рисунків, фактичній передачі даних передує початкова фаза *зв'язування* (binding) сокету, коли визначається додаткова інформація, необхідна для створення комунікаційного вузла. Зв'язування може бути здійснене за допомогою системного виклику *bind(2):*

#include <sys/types-h>

#include <sys/socket.h>

int bind(int sockfd, struct sockaddr *localaddr, int addrien);

Тут sockfd є дескриптором сокету, отриманим при його створенні; аргумент localaddr визначає локальну адресу, з якою необхідно зв'язати сокет; параметр addrlen визначає розмір адреси. Помітимо, що мова йде про зв'язування з локальною адресою, у загальному випадку визначальним два параметри комунікаційного каналу (комунікаційний вузол): локальна адреса й локальний процес.

На відміну від локального міжпроцесної взаємодії, для мережного обміну даними необхідна вказівка: як локального процесу, так і хосту, на якому виконується даний процес.

Отже, зв'язування необхідне для присвоєння сокету локальної адреси й, таким чином, для визначення комунікаційного вузла. Можна виділити три випадки використання для цього функції *bind(2)'*.

1. Сервер реєструє свою адресу. Ця адреса повинна бути заздалегідь відома клієнтам, що бажають "спілкуватися" із сервером. Зв'язування необхідне, перш ніж сервер буде готовий до приймання запитів від клієнтів.

2. При взаємодії без попереднього встановлення зв'язку й створення віртуального каналу клієнт також повинен попередньо зареєструвати свою адресу. Ця адреса повинна бути унікальною у рамках комунікаційного домену. У випадку домену UNIX про це повинна подбати сама програма. Ця адреса не повинна бути заздалегідь відома серверу, оскільки запит завжди ініціює клієнт, автоматично передаючи разом з ним свою адресу. Отримана адреса віддаленого вузла потім використовується сервером для мультиплексування повідомлень різним клієнтам, що відправляються.

3. Навіть у випадку взаємодії з використанням віртуального каналу клієнт може побажати зареєструвати власну адресу, не покладаючись при цьому на систему.

Призначення адреси для клієнта також можна виконати за допомогою системного виклику *connect(2), що* встановлює зв'язок із сервером і автоматично зв’язує сокет клієнта з локальним комунікаційним вузлом.

Характер цього виклику припускає створення віртуального каналу й, таким чином, використовується для попереднього встановлення зв'язку між комунікаційними вузлами. У цьому випадку клієнтові немає необхідності явно зв'язувати сокет за допомогою системного виклику *bind(2).* Локальний вузол комунікаційного каналу вказується дескриптором сокету sockfd, для якого система автоматично вибирає прийнятні значення локальної адреси й процесу. Віддаленний вузол визначається аргументом servaddr, який указує на адресу сервера, а addrlen задає його довжину.

Виклик *connect(2)* може також застосовуватися й клієнтами, що використовують сокети датаграм без створення віртуального каналу. У цьому

випадку *connect(2)* не викликає фактичного з'єднання із сервером, а є зручним способом збереження параметрів адресата (сервера), якому будуть направлятися датаграми. При цьому клієнт буде позбавлено від необхідності вказувати адресу сервера при кожному відправленні даних.

Якщо в момент одержання запиту на встановлення зв'язку черга запитів, що очікують, досягла свого максимального значення, *виклик connect(2)* клієнта завершиться з помилкою ECONNREFUSED для домену UNIX (AF_UNIX). Для інших доменів результат залежить від того, чи підтримує протокол повторну передачу запиту. Наприклад, протокол TCP (домен AF_INET) буде передавати повторні запити, поки число запитів у черзі не зменшиться, або не відбудеться тайм-аут, визначений для протоколу. В останньому випадку виклик клієнта завершиться з помилкою ETIMEDOUT.

Фактичну обробку запиту клієнта на встановлення зв'язку здійснює системний виклик *accept(2).* Виклик *accept(2)* вилучає перший запит із черги й створює новий сокет, характеристики якого не відрізняються від сокета sockfd, і в такий спосіб завершує створення віртуального каналу з боку сервера. Одночасно *accept(2)* повертає параметри віддаленого комунікаційного вузла — адресу клієнта clntaddr і його розмір addrlen. Новий сокет використовується для обслуговування створеного віртуального каналу, а отримана адреса клієнта виключає анонімність останнього.

У цьому сценарії, у той час як дочірній процес забезпечує фактичний обмін даними із клієнтом, батьківський процес продовжує "прослуховувати" запити, що надходять, породжуючи для кожного з них окремий процес-оброблювач. Черга дозволяє буферизувати запити на час, поки сервер завершує виклик *accept(2)* і потім створює дочірній процес. Помітимо, що новий сокет newsockfd, отриманий у результаті виклику *accept(2),* адресує повністю визначений комунікаційний канал: протокол і повні адреси обох вузлів — клієнта й сервера. Навпаки, для сокета sockfd визначена тільки локальна частина каналу. Це дозволяє серверу продовжувати використовувати sockfd для "прослуховування" наступних запитів.

Нарешті, якщо для сокетів потоку при прийманні й передачі даних можуть бути використані стандартні виклики *read(2)* і *write(2),* то сокети датаграм повинні користуватися спеціальними системними викликами (ці виклики також доступні для сокетів інших типів):

Функції *send(2)* і *sendto(2)* використовуються для передачі даних віддаленому вузлу, а функції *recv(2)* і *recvfrom(2)* — для їхнього приймання. Основною відмінністю між ними є те, що функції *send(2)* і *recv(2)* можуть бути використані тільки для "приєднаного" сокету, тобто після виклику *connect(2).*

Усі ці виклики використовують як перший аргумент дескриптора сокету, через який здійснюється обмін даними. Аргумент msg містить повідомлення довжиною len, яке повинне бути передане за адресою toaddr, довжина якого становить tolen байтів. Для функції *send(2)* використовується адреса одержувача, установлена попереднім викликом *connec1(2).* Аргумент buf являє собою буфер, у який копіюються отримані дані.

1.18.8. Порівняння різних систем міжпроцесної взаємодії

Закінчуючи розмову про міжпроцессну взаємодію в UNIX, приведемо зведену порівняльну таблицю розглянутих систем (табл. 2.3).

Якщо говорити про продуктивність **IPC,** то найбільш швидким способом передачі даних між неспорідненими процесами є поділювана пам'ять. Поділювана пам'ять є частиною адресного простору для кожного із взаємодіючих процесів, тому читання й запис у цю область не відрізняються, наприклад, від читання й запису в область власних даних процесу. Однак при використанні поділюваної пам'яті необхідно забезпечити синхронізацію процесів. При використанні семафорів, необхідно мати на увазі наступні обставини:

- Застосування семафорів може збільшити число процесів у черзі на виконання, оскільки кілька процесів, що очікують дозволяючого сигналу семафора, будуть одночасно розбуджені й переведені в чергу на виконання.
- Застосування семафорів збільшує число перемикань контексту, що, у свою чергу, збільшує навантаження на систему.
- У той же час, використання семафорів є найбільш стандартним (POSIX.lb), хоча й неефективним способом забезпечення синхронізації.

Черги повідомлень призначені для обміну короткими (звичайно менше 1 Кбайт) структурами даних. Якщо обсяг даних перевищує цю величину, використання повідомленнь може значно збільшити число системних викликів і зменшити продуктивність операційної системи.

Табл. 1.3

	Канали	**FIFO**	Повідомлення	Поділювана пам'ять	Сокети (домен
Простір імен	—	Ім'я файлу	Ключ	Ключ	Ім'я файлу
Об'єкт	Системний канал	Іменований	Черга повідомле	Поділювана область	Комунікаційний вузол
Створення об'єкта	pipe ()	niknod ()	msgget ()	shmget()	socket ()
Зв'язуванн	pipe ()	open()	msgget()	shmat ()	bind ()
Передача даних	read() write ()	read() write ()	msgrcv() msgsnd()	Безпосередній доступ memcpy(	read () write () recv() send ()
Знищення	close 0	close() unlink()	msgcti()	Shnidtf()	close () unlink()

1.19. МЕХАНІЗМ ПЕРЕРИВАНЬ — ОСНОВА КЕРУВАННЯ ПРОЦЕСАМИ

У цій частині розглянута концепція переривань – одного з основних механізмів організації багатопрограмних режимів роботи й обміну інформацією із зовнішніми пристроями. Реалізація системи обробки переривань розглянута на прикладі операційних систем MS-DOS, WINDOWS 95(98) і деяких особливостей, застосовуваних в ОС UNIX. Описані загальні питання організації системи переривань, дані основні поняття й визначення, обговорюються проблеми й різні методи реалізації апаратних і програмних засобів підтримки механізму переривань.

Практично будь-яка обчислювальна система, крім одного або декількох процесорів має, у своєму складі безліч пристроїв, що обробляють дані паралельно із процесором, що підвищує загальну продуктивність системи. Це зовнішні накопичувачі, пристрої комутації, додаткові спеціалізовані процесори, таймери та інші пристрої. При такій організації системи виникає проблема реалізації взаємодії між процесором і периферійними пристроями.

1.19.1. Взаємодія із зовнішніми пристроями

Розглянемо таку взаємодію докладніше. Пристрій може *незалежно* від процесора виконувати операції введення/виводу, а іноді й розміщати результат в оперативну пам'ять (використовуючи прямий доступ до пам'яті або захват циклу пам'яті). Якщо процесору необхідно прийняти або передати дані з/у зовнішнього пристрою, він посилає пристрою запит на виконання певних дій. Якщо пристрій готовий його обробити, воно виконує програму введення/виводу й задовольняє запит, а процесор отримує результат його роботи. Таким чином, процесору необхідно активізувати пристрій, призначити йому роботу й одержати результат (або інформацію про завершення обробки). Розглянемо можливі варіанти організації такого обміну:

1. Процесор передає пристрою необхідну для роботи інформацію, а потім *постійно* перевіряє стан пристрою до появи в ньому ознаки закінчення обробки, після чого (якщо необхідно) приймає результат. Такий режим називають **режимом безпосереднього (програмного) опитування** *(Polling mode).*

2. Процесор розміщає необхідні пристрою дані в визначене місце, звідки пристрій їх зчитує, потім виконує необхідну роботу й розташовує результат також у визначеному місці, з якого процесор зможе їх отримати. Цей розвиток режиму безпосереднього опитування, що виключає пряма взаємодія процесора із пристроєм, одержало назву **системи з поштовою скринькою** *(Mailbox system).*

3. Процесор посилає пристрою запит і виконує процес, не пов'язаний із закінченням роботи зовнішнього пристрою. Пристрій виконує свої функції, а після формування результату посилає процесору спеціальний сигнал, що повідомляє про те, що обробка закінчена. Процесор звертається до пристрою й одержує необхідну інформацію. Саме такий метод реалізується за допомогою **системи переривань** *(Interrupt system).*

Ці методи різні за обсягом необхідних для їхньої реалізації ресурсів, за часом одержання результату взаємодії процесора й периферійного пристрою й по тимчасових витратах на очікування взаємодії. Так, режим безпосереднього опитування вимагає мінімальної кількості ресурсів і дозволяє одержати результат найбільш швидким чином. Однак, на очікування готовності пристрою процесор витрачає досить багато часу, особливо, якщо пристрій має невисоку швидкість обробки даних (для деяких пристроїв уведення/виводу, наприклад, для друкувальних пристроїв, швидкість обробки інформації в тисячі й мільйони раз нижче, ніж у процесора), а це значить, що під час взаємодії процесор більшу частину часу витрачає даремно.

Системи з "поштовими скриньками" дозволяють організувати більш незалежну роботу процесора й зовнішнього пристрою, але це збільшує час одержання результату й витрати на додаткове встаткування.

Переривання зводять час даремного очікування до нуля, але реалізація системи переривань вимагає найбільших витрат устаткування, до того ж необхідність перемикання виконуваної в процесорі задачі на програму взаємодії із пристроєм вимагає додаткового часу.

Саме тому, в обчислювальних системах одночасно можуть використовуватися різні методи організації взаємодії процесорів і допоміжних пристроїв залежно від вимог, які пред’являються до такої взаємодії. Одні пристрою можуть використовувати переривання, інші — безпосередній зв'язок із процесором або "поштові скриньки". Існують також комбінації цих методів. Наприклад, різновидом методу безпосереднього опитування є режим, у якому процесор, виконуючи основну роботу, *переривається* сигналом таймера через певні проміжки часу. При цьому він перевіряє стан усіх підключених до нього пристроїв, відзначаючи що відбувся із часу останнього опитування зміни, і, якщо такі присутні, перевіряє, є чи в пристрої необхідні йому дані, і отримує наявну для нього інформацію. Завершивши опитування, процесор продовжує виконання перерваного процесу. Аналогічний варіант можливий і при перегляді "поштових скриньок". Однак, найбільш часто механізм переривань використовується, у баготозадачних системах, тому що з погляду витраченого часу цей режим найбільш економний.

Механізм переривань використовується не тільки при взаємодії процесора із зовнішніми пристроями. Адже в якості ресурсу можуть виступати не тільки зовнішні пристрої, але й процесор по відношенню до процесів, процес стосовно іншого процесу при їхній взаємодії, а також операційна система стосовно програми користувача. Тому переривання є основою функціонування ОС (особливо багатозадачної).

1.19.2. Обробка переривань

У загальному випадку під **перериванням** розуміють подію, що вимагає від системи припинення роботи активного процесу й перехід до обробки іншого завдання. Обчислювальна система, що володіє можливістю використання переривань, повинна містити цілий комплекс програмно-апаратних засобів, що забезпечують реєстрацію сигналу переривання, призупинення активного процесу, перехід до обробки переривання й відновлення перерваного процесу. Сукупність дій, виконуваних після реєстрації сигналу переривання, називається **обробкою переривання. Система обробки переривань**— комплекс програмно-апаратних засобів, що забезпечують припинення активного процесу й перехід до обробки переривання, тобто виконання спеціальної програми обробки переривання.

Розглянемо роботу системи переривань докладніше. У реальному житті нам дуже часто доводиться зустрічатися з аналогами переривань, реалізованих в обчислювальних системах. Приміром, припустимо, що ви читаєте книгу, і в цей час лунає телефонний дзвінок. Тоді ви відкладаєте книгу, підходите до телефону, відповідаєте на дзвінок, а потім повертаєтеся до перерваного на найцікавішому місці заняття. У цей момент вам необхідно визначити, з якого місця продовжити читання. Повернемося до того моменту, коли ви відклали книгу. Якщо ви очікували важливого дзвінка, то ви могли закрити книгу, навіть не запам'ятавши місця, на якому зупинилися. В іншому випадку ви могли б залишити позначку олівцем у тому місці, де вас *перервав* телефонний дзвінок, а на потрібній сторінці покласти закладку. Але якщо книга вас досить зацікавила, то ви могли б не відразу реагувати на дзвінок, а дочитати до кінця речення, абзацу, а то й до кінця "найцікавішого місця".

Аналогічним чином працюють переривання в обчислювальних системах. Коли процесор отримує сигнал переривання, він повинен перервати виконуваний у ньому процес. При цьому можливі кілька варіантів реакції процесора на сигнал переривання, що називаються моментом переривання.

Можливо три варіанти реалізації моменту переривання:

1. Переривання можливе тільки після певних команд, при цьому необхідно збереження мінімальної кількості інформації про стан системи.

2. Переривання можливе після завершення будь-якої команди процесора.

3. Переривання можливе після завершення чергового такту процесора. При цьому потрібне збереження великого обсягу інформації.

Більшість процесорів побудована таким чином, що сигнал переривання перевіряється ними тільки після завершення виконання поточної команди процесу, що виконується. Однак, у системі можуть існувати переривання, для яких неможливе очікування завершення чергової команди, наприклад, якщо виникає помилка в самій команді. Тому в системі можуть використовуватися одночасно декілька з перерахованих способів для обробки різних переривань.

Виявивши сигнал переривання, процесор повинен запам'ятати стан перерваного процесу для того, щоб продовжити його виконання після обробки переривання. Після цього в процесор завантажується новий процес, що виконує обробку переривання. Ця процедура одержала назву **перемикання контексту** або **перемикання стану**. Вона є однієї з найважливіших в організації роботи операційної системи при реалізації багатопрограмного режиму роботи обчислювальної системи, причому її реалізація вимагає підтримки й виконання деяких функцій на апаратному рівні.

Кожний процесор оснащений апаратурою для керування послідовністю виконання команд. Це спеціальний регістр, що постійно зберігає адреса наступної команди. Він називається **словом стану програми** *(Program Status Word, PSW),* інакше називаний також **словом стану процесу** *(Process Status Word, PSW),* **лічильником програми** *(Program Counter, PC),* **лічильником адреси програми** *(Program Address Counter, PAC),* **вказівником команди** *(Instruction Pointer, IP).* Як мінімум, такий регістр повинен містити адреса наступної команди (або послідовності команд), а на додаток до цього – різну інформацію, яку система зв'язує із процесом. Розрізняють три види PSW: поточне, старе й нове. Поточне PSW або PC (PAC), як уже було сказано,

містить адресу наступної виконуваної команди. Старе PSW містить адресу команди, перерваної програми, з якої вона буде виконуватися після обробки переривання й передачі керування до перерваної програми. Нове PSW містить адреса точки входу в програму обробки переривання. З появою переривання система обробки переривань виконує зміну PSW при вході в перериваючу програму, а по її завершенню виконується зворотна зміна PSW для продовження виконання перерваної програми.

При перемиканні контексту процесор зберігає вміст PSW, потім поміщає в нього нове значення, відповідне до процесу, називаного **оброблювачем переривання** *(Interrupt Handler, IH),* який і виконує обробку самого переривання. Після завершення обробки переривання в PSW відновлюється вміст збереженого раніше PSW. Для розуміння процесів, що протікають в обчислювальній системі з появою переривання можна ввести умовні стани процесора P1-P4 (рис.1.26). У стані P1 процесор виконує прикладну програму.

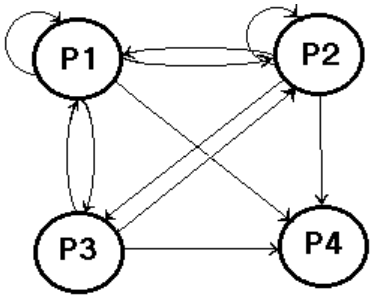


Рис.1.26 Зміна станів процесора при обробці сигналу переривання.

У стані P2 процесор виконує програму обробку переривання. У стані P3 виконується дешифрація сигналу переривання й у стані P4 виконуються дії, пов'язані з перериванням від схем контролю ВР.

Відповідності до схеми, при виникненні переривання від схем контролю машини ОС переходить у стан P4. При виникненні сигналу переривання й фіксації його на фізичному рівні у ОС, що приводить до появи загального сигналу переривання, процесор переходить у стан P3 і виконує дії по визначенню джерела сигналу переривання і його дешифрації. Система перебуває в стані P3 до завершення всіх операцій, пов'язаних із цим. Під час

цього стану система не може реагувати на інші джерела переривань, тому що контролер переривань заблокований. Блокування контролера переривань знімається при переході ВР зі стану Р3 у стани Р1 або Р2. Вибір стану Р1 або Р2 визначається пріоритетністю процесів у цих станах і обраної у ВР дисципліною їх обслуговування.

У процесі обробки переривання можна виділити наступні **фази переривання** (мал. 1.27):

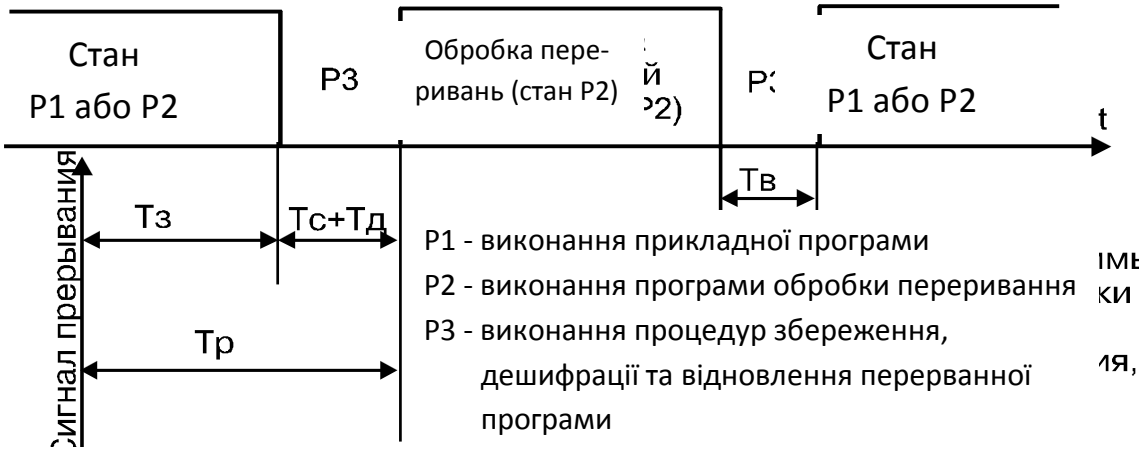


Рис.1.27 Фази переривання

1. $T_з$ — **час затримки** між моментом виникнення сигналу переривання й перериванням активного процесу. Воно залежить від прийнятого в системі (процесорі) способу обробки сигналу переривання (моменту переривання).

2. $T_з$ — **час збереження** необхідної інформації програми, що переривається. Залежить від кількості інформації, що зберігається, при прийнятому способі обробки сигналу переривання.

3. $T_д$ — **час дешифрації** сигналу переривання (визначення джерела й типу переривання). Залежить від апаратури, дешифруючої сигнал переривання.

4. $T_в$ — **час відновлення** перерваного процесу. Залежить від кількості відновлюваної інформації.

Час між виникненням сигналу переривання й початком виконання оброблювача переривання називається **часом реакції системи на сигнал переривання** ($T_р$).

Для реалізації механізму переривань необхідна апаратна схема фіксації сигналу запиту на переривання. Така схема звичайно містить регістр, на

якому фіксується наявність сигналів у вхідних лініях **запитів на переривання**. Об'єднані схемою "АБО", сигнали з розрядів регістру формують загальний сигнал про наявність запиту на переривання. Потім усі розряди регістру опитуються в порядку пріоритетів вхідних ліній. При цьому використовуються 2 варіанта реалізації опитування:

1. Повно впорядкована схема**.** Регістри опитуються за порядком пріоритетів, від вищого до нижчого. Це проста схема, однак, при виникненні переривань із низьким пріоритетом необхідно перевірити *всі* регістри запитів на переривання з більш високим пріоритетом.

2. Частково впорядкована схема**.** Усі переривання діляться на **класи,** і вводиться 2-рівнева система пріоритетів: перший рівень — серед класів, другий — усередині кожного класу. При цьому, спочатку за повно впорядкованою схемою визначається клас переривання, на яке надійшов запит, а потім, також за повно впорядкованою схемою усередині встановленого класу, визначається саме переривання. Це прискорює пошук переривання причини переривання, однак ускладнює процедуру пошуку.

1.19.3. Класи переривань

По своєму призначенню, причині виникнення переривання діляться на різні **класи**. Традиційно виділяють наступні класи:

1. *Переривання від схем контролю машини*. Виникають при виявленні збоїв у роботі апаратури, наприклад, при розбіжності парності в мікросхемах пам'яті.

2. *Зовнішні переривання*. Збуджуються сигналами запитів на переривання від різних зовнішніх пристроїв: таймера, клавіатури, іншого процесора та ін.

3. *Переривання по введенню/виводу*. Ініціюються апаратурою введення/виводу при зміні стану каналів уведення/виводу, а також при завершенні операцій уведення/виводу.

4. *Переривання по звертанню до супервізора*. Викликаються при виконанні процесором **команди звертання до супервізора** (виклик функції операційної системи). Звичайно така команда ініціюється виконуваним

процесом при необхідності одержання додаткових ресурсів або при взаємодії із пристроями введення/виводу.

5. *Програмні переривання*. Виникають при виконанні **команди виклику переривання** або при виявленні помилки у виконуваній команді, наприклад, при арифметичному переповненні.

Переривання 4 і 5 класів можна об'єднати в один клас програмних переривань. При виконанні програми вони не можуть виникнути одночасно, тому що активізуються процесором, причому, залежно від причини, що викликала переривання, серед них виділяють такі підтипи:

- переривання, викликане виконанням процесором *командипереходу до підпрограми обробки переривання;*
- переривання, що виникають у результаті *виняткової* (аварійної) *ситуації* в процесорі (поділ на "0", переповнення і т.д.).

У зв'язку з різноманіттям різних обчислюваних систем і їх постійним розвитком змінюється й організація системи переривань. Так, з появою віртуальної пам'яті, з'явилися сторінкові переривання, які можна віднести й до класу виняткових ситуацій у процесорі; у системах з кеш-пам'яттю реалізуються переривання, при кеш промахах і необхідністю підкачування сторінок у кеш-пам'ять і т.д.

Під час виконання програми обробки одного з переривань можливенадходження на обробку іншої програми обробки переривання, тому система повинна використовувати певну *дисципліну обслуговування заявок* на переривання. Звичайно різним класам або окремим перериванням привласнюються різні *абсолютні пріоритети.* При надходженні запиту з вищим пріоритетом поточний оброблювач знімається, а його місце займає оброблювач переривання, що знов надійшло. Кількість рівнів вкладеності переривань називають **глибиною системи переривань**. Значення цієї характеристики визначається обраним у системі способом і правилами обробки переривань (наприклад, обсягом пам'яті, виділеної для зберігання старих PSW). Оброблювачі переривань або дисципліна обслуговування

заявок на переривання повинні мати також спеціальні засоби для випадку, коли при обробці переривання виникає запит на обробку переривання від того ж джерела. У тому випадку, якщо при обробці переривання від того ж джерела приходить повторне переривання, то виникає ситуація називана насиченням системи переривання. При виникненні такої ситуації потрібна зміна стратегії обробки переривань, що звичайно приводить до зміни всієї системи обробки переривання у ОС.

1.20. РЕАЛІЗАЦІЯ МЕХАНІЗМУ ПЕРЕРИВАНЬ У КОМП'ЮТЕРАХ ТИПУ IBM PC

У комп'ютерах сімейства IBM PC, що використовують операційну систему MS-DOS, механізм переривань реалізується як за допомогою апаратних, так і за допомогою програмних засобів.

Процесор підтримує обробку 256 різних переривань, кожне з яких ідентифікується номером у діапазоні 00h-Ffh. Сегментні адреси оброблювачів переривань, називані **векторами переривань**, містяться в **таблиці векторів переривань**, яка розташовується на початку оперативної пам'яті (за адресою 0000:0000). Кожний вектор переривання має розмір 4 байта, таким чином, таблиця векторів займає 1024 байта. Для визначення адреси оброблювача будь-якого переривання потрібно номер переривання помножити на 4.

У захищеному режимі старші моделі сімейства процесорів разом з таблицею векторів переривань використовують **таблицю дескрипторівпереривань,**подібну по своєму призначенню з таблицею векторів (схема реалізації системи переривання в захищеному режимі розглядається в наступному розділі).

Операційна система не контролює вміст таблиці векторів, хоча і є засоби для читання й зміни векторів. За допомогою функцій MS-DOS значення векторів можна змінити, помістивши потрібну адресу безпосередньо в таблицю векторів переривань. У цьому випадку контроль над умістом

векторів і їх відповідністю розміщенню оброблювачів у пам'яті повністю лягає на програміста, а наслідки у випадку помилки можуть бути непередбаченими.

Активізація оброблювача переривання може відбутися в результаті виникнення наступних ситуацій:

– виникнення сигналу внутрішнього переривання усередині мікросхеми процесора (**внутрішні** або **логічні переривання**);

– надходження у процесор сигналу запиту на переривання від зовнішнього (стосовно процесора) пристрою (**апаратні переривання**);

– виконання процесором команди INT виклику процедури обробки переривання (**програмні переривання**).

1.20.1. Внутрішні переривання

Для обслуговування внутрішніх переривань мікропроцесори i8086 використовували 5 перших векторів переривань (00h-04h). Переривання з номерами 05h-31h фірма Intel зарезервувала для використання в подальших розробках, однак фірма IBM при створенні IBM PC використовувала ці вектора за своїм розсудом. У наступних реалізаціях процесорів ці ж вектора були задіяні для обробки нових внутрішніх переривань, у результаті чого стали можливі конфлікти. Оскільки всі додані переривання використовуються в захищеному режимі процесора, то для запобігання конфліктів при переході в захищений режим контролер переривань перепрограмується таким чином, що при вступі сигналу на лінію IRQ0 викликається процедура обробки з номером, відмінним від 08h (звичайно 50h або 60h). Для наступних ліній IRQ викликаються процедури з наступними (відносно базового номера для IRQ0) номерами.

Сигнали внутрішніх переривань виникають при порушеннях у роботі самого мікропроцесора або при помилках у виконуваних командах і формуються усередині схеми мікропроцесора при виникненні однієї із ситуацій, зазначених у табл. 2.4.

1.20.2. Апаратні переривання

Апаратні переривання ініціюються різними пристроями комп'ютера, зовнішніми стосовно процесора (системний таймер, клавіатура, контролер диска та ін.). Ці пристрої формують сигнали запитів на переривання (IRQ), що надходять на **контролер переривань**.

1.20.2. 1. *Програмувальний контролер переривань (ПКП)*

Програмувальний контролер переривань *(Programmable Interrupt Controller, PIC),* реалізований на мікросхемі i8259, призначений для підтримки апаратних переривань від восьми різних пристроїв. Кожний пристрій має власний пріоритет при обслуговуванні заявок на переривання. Крім того, сигнал запиту переривання може бути **замаскований**, тобто з появою цього сигналу контролер переривань може його ігнорувати. Для збільшення кількості зовнішніх пристроїв, що обслуговуються, мікросхеми контролерів можна каскадувати, підключаючи виходи додаткових мікросхем до входів основної.

Контролер переривань можна запрограмувати для роботи в декількох режимах:

1. Режим фіксованих пріоритетів *(Fixed Priority, Fully Nested Mode).* У цьому режимі запити переривань мають жорсткі пріоритети від 0 (вищий) до 7 (нижчий) і обробляються відповідно до цих пріоритетів. Якщо запит з меншим пріоритетом виникає під час обробки запиту з більш високим пріоритетом, то він ігнорується. BIOS при включенні живлення й при перезавантаженні програмує контролер переривань на роботу саме в цьому режимі. Надалі, цей режим можна змінити програмним способом при необхідності.

Табл. 1.4 Внутрішні переривання процесорів i80x86

Виняткова ситуація	Номер переривання

Виняткова ситуація	Номер переривання
Поділ на 0	00h
Завершення виконання чергової команди процесора при встановленому прапорі TF (використовується в режимі налагодження програм)	01h
Особлива ситуація в режимі налагодження (80386+)	01h
Виконання команди INTO при встановленому прапорі OF (використовується при виявленні переповнення)	04h
Вихід величини, що перевіряється, за межі діапазону при виконанні команди BOUND (80186+)	05h
Спроба виконання неприпустимої операції (наприклад привілейованої команди в реальному режимі) (80286+)	06h
Спроба виконання команди співпроцесора при відсутності останнього (80286+)	07h
Виникнення двох виняткових ситуацій в одній команді (80286+)	08h
Спроба співпроцесора одержати доступ до пам'яті за границями сегмента (80286,80386)	09h
Спроба перемкнутися на TSS, що містить невірну інформацію (80286+)	0Ah

Виняткова ситуація	Номер переривання
Звертання до сегмента, вивантаженого на диск із оперативної пам'яті (80286+)	0Bh
Переповнення стека (80286+)	0Ch
Відмова загального захисту (80286+)	0Dh
Помилка звертання до сторінки віртуальної пам'яті (80386+)	0Eh
Звернення за неправильно вирівняною адресою пам'яті (486+)	11h

1. Режим з автоматичним зсувом пріоритетів *(Automatic Rotation).* У цьому режимі після обробки переривання даному рівню призначається мінімальний пріоритет, а всі інші пріоритети змінюються циклічно таким чином, що запити наступного за тільки що обробленим рівнем мають найвищий, а попереднього — нижчий пріоритет. Наприклад, після обробки запиту рівня 6 пріоритети встановлюються в такий спосіб: рівень 7 має пріоритет 0, рівень 8 — пріоритет 1 і т.д., рівень 5 — пріоритет 6 і рівень 6 — пріоритет 7. Таким чином, у цьому режимі всім запитам дається можливість одержати обробку переривання за прийнятний час.

2. Режим із програмно-керованим зсувом пріоритетів *(Specific Rotation).* Пріоритети можна змінювати так само, як і в режимі автоматичного зсуву, однак для цього необхідно подати спеціальну команду, у якій вказується номер рівня, якому потрібно призначити максимальний пріоритет.

3. Режим автоматичного завершення обробки переривання *(Automatic End of Interrupt).* У цьому режимі, на відміну від інших, оброблювач переривання *не повинен* при своєму завершенні посилати контролеру спеціальний **сигнал завершення обробки апаратного переривання** *(End Of Interrupt, EOI),* який дозволяє обробку переривань із поточним і більш низьким пріоритетами. У даному режимі ці переривання дозволяються в момент початку обробки

переривання. Режим використовується рідко, тому що всі оброблювачі переривань повинні бути **повторно входимі (реєнтерабельними)** у випадку, якщо до завершення обробки виникне запит на переривання від того ж джерела.

4. Режим спеціальної маски *(Special Mask Mode).* У даному режимі можна замаскувати окремі рівні переривань, змінивши пріоритетне впорядкування обробки запитів. Після скасування цього режиму відновлюється колишній порядок обробки переривань.

5. Режим опитування *(Polling Mode).* У цьому режимі обробка переривань не проводиться автоматично, сигнали запиту переривань лише фіксуються у внутрішніх регістрах контролера. Процедуру обробки переривання необхідно викликати "вручну", відповідно до номера рівня з максимальним пріоритетом, що зберігається в контролері, по якому є запит.

Крім цього, є можливість програмувати номер процедури переривання, яка буде викликатися при виникненні сигналу на першій лінії IRQ (при цьому номера процедур для інших IRQ будуть мати подальші значення). Ця можливість використовується при переході в захищений режим процесора, тому що в цьому режимі переривання з номерами 05h-11h використовуються для обробки критичних ситуацій (логічні переривання). Щоб виключити можливі конфлікти, апаратні переривання обробляються процедурами з більшими номерами, наприклад, OS/2 v1.x переміщає оброблювачі переривань від пристроїв, підключених до IRQ0-IRQ7, на процедури з номерами 50h-57h. Крім цього, зазначену процедуру застосовують у тих випадках, коли необхідно контролювати виконання оброблювачів переривань, тому що перепрограмування контролера ускладнює перехоплення переривань.

Приведемо приклад такої програми:

```
IDEAL
MODEL TINY
CODESEG
```

```
    Startupcode
    JMP INIT    ; перехід до ініціалізації резидента
IRQ0:           ; підпрограма обробки переривання таймера
    STI         ; дозвіл переривань
    INT 08h     ; виклик оригінального оброблювача
    IRET        ; повернення з оброблювача
IRQ1:           ; підпрограма обробки переривання клавіатури
    STI
    INT 09h
    IRET
IRQ2:           ; підпрограма обробки запиту по IRQ2
    STI
    INT 0Ah
    IRET
IRQ3:           ; обробка переривання від COM2,COM4
    STI
    INT 0Bh
    IRET
IRQ4:           ; обробка переривання від COM1,COM3
    STI
    INT 0Ch
    IRET
IRQ5:           ; підпрограма обробки запиту по IRQ5
    STI
    INT 0Dh
    IRET
IRQ6:           ; підпрограма обробки запиту по IRQ6
    STI
    INT 0Eh
    IRET
```

```
IRQ7:                ; підпрограма обробки запиту по IRQ7
    STI
    INT 0Fh
    IRET
INIT:
    MOV AX,2560h     ;  занесення нових значень у таблицю векторів
                     ;    25 - номер функції
                     ;    60 - номер переривання
    MOV BX,OFFSET IRQ ;   адреса початку таблиці адрес нових
                     ;   оброблювачів переривань
A1:     MOV DX,[BX]      ; значення чергового елемента таблиці
                     ;  адрес нових оброблювачів переривань
    INT 21h          ; виклик функції MS-DOS
    ADD BX,2         ; зміна індексу в таблиці
    INC AL           ; збільшення номера переривання
    CMP AL,67h
    JNG A1
    CLI              ; заборона переривань
    MOV AL,0Ffh      ; процедура перепрограмування контролера
    OUT 21h,AL
    MOV AL,11h
    OUT 20h,AL
    MOV AL,60h       ; номер першої процедури обробки переривань
    OUT 21h,AL
    MOV AL,4
    OUT 21h,AL
    MOV AL,1
    OUT 21h,AL
    MOV AL,0
    OUT 21h,AL
```

```
        OUT 20h,AL
        STI                     ;  дозвіл переривань
        MOV AX,3100h            ;  завершення резидентної програми
        MOV DX,OFFSET INIT      ;  обчислення розміру резидентної
частини
        MOV CL,4        ;  у параграфах
        SHR DX,CL
        INC DX
        INT 21h         ;  кінець коду, що виконується
              ; таблиця адрес нових оброблювачів переривань
    IRQ     DW IRQ0,IRQ1,IRQ2,IRQ3,IRQ4,IRQ5,IRQ6,IRQ7
    ENDS
    END @Startup
```

Дана програма програмує контролер таким чином, що поява сигналів на лініях IRQ0-IRQ7 викликає відповідно процедури INT60h-int67h. Нові оброблювачі переривань у даному прикладі не виконують ніяких додаткових функцій, вони просто викликають старі процедури обробки, однак їх неважко наповнити потрібним змістом.

Так як після виконання цієї процедури замість переривань INT08h-int0Fh будуть викликатися процедури INT60h-int67h, то необхідно попередньо занести в таблицю векторів адреси нових оброблювачів переривань, які вже повинні знаходитися в пам'яті. Таблицю векторів можна змінити, безпосередньо записавши потрібні значення в пам’ять за адресами, починаючи з 0000:0180 (60h*4=180h), або використовуючи функцію 25h переривання 21h, надавану MS-DOS, як і зроблено в наведеному прикладі. При виклику переривання необхідно вказати наступні параметри:

- у регістрі AH — номер функції
- у регістрі AL — номер змінюваного вектора
- у регістрах DS:DX — адреса нового оброблювача

Програмування контролера полягає в посилці певних кодів у керуючі регістри ПКП (з номерами 20h і 21h). Серед цих кодів вказується й номер вектора, який контролер надалі буде виставляти на шину при вимозі переривання процесора.

Програма складається із двох частин: резидентної частини й ініціалізуючої. При запуску програми виконується код, від мітки INIT, за якою, у циклі у таблицю векторів за допомогою функції MS-DOS заносяться нові значення. Далі йде процедура програмування контролера, яка для того, щоб виключити обробку переривань до завершення перепрограмування, обрамлена командами CLI і STI.

Програма завершується викликом функції 31h переривання 21h. При цьому частина програми, розташована перед міткою INIT, залишається в пам'яті після завершення програми. Резидентна частина містить нові оброблювачі переривань, кожний з них починається міткою IRQx (відповідно до оброблюваного запиту). Адреси оброблювачів розміщуються в таблицю, що знаходитьсяв кінці програми, для зручності їх використання при занесенні в таблицю векторів.

Функція 31h вимагає вказівки в регістрі DX розміру резидентної частини в параграфах. Для цього розмір резидента в байтах (обумовлений вираженням OFFSET INIT) ділиться на 16 за допомогою зсуву вправо на 4 розряди й округляється в більшу сторону.

Дана програма припускає, що вектора 60h-67h не використовуються іншими програмами, інакше виникне конфлікт між програмами, що спільно використовують ці вектора переривань, і переривання будуть оброблятися некоректно.

Схема взаємодії зовнішніх пристроїв, ПКП і центрального процесора показана на рис.1.28. У комп'ютерах класу IBM PC і PC/XT до контролера підключено 8 пристроїв, для обробки переривань, від яких використовуються підпрограми, на які вказують вектора 08h..0Fh (табл.1.5).

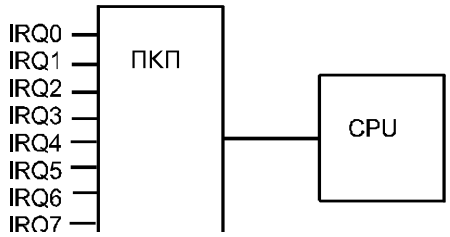


Рис. 1.28 Загальна схема з'єднання джерел переривань, ПКП і мікропроцесора в IBM PC і IBM PC/XT

У зв'язку зі збільшенням кількості зовнішніх пристроїв, що вимагають підтримки апаратними перериваннями, комп'ютери IBM PC/AT і наступні моделі оснащуються двома контролерами переривань. Додатковий контролер, називаний **веденим**, підключається до входу IRQ2 основного контролера, який називають **провідним** (рис.1.29.). На входи веденого ПКП підключені лінії IRQ8-IRQ15, причому лінія IRQ8 має максимальний пріоритет у веденому контролері, а IRQ15 — мінімальний. У результаті такого каскадування запити від пристроїв, підключених до веденого контролера, обробляються по дворівневій системі пріоритетів (молодший рівень — усередині веденого ПКП, а старший — усередині ведучого). Оскільки вихід веденого контролера підключений до лінії IRQ2, пріоритети ліній IRQ8-IRQ15 розташовані між пріоритетами IRQ1-IRQ3. Пристрої, що мають можливість обслуговування апаратними перериваннями в комп'ютерах IBM PC/AT, перераховано в таблиці 1.6.

Табл. 1.5 Апаратні переривання IBM PC/XT

Лінія запиту	Вектор переривання	Підключений пристрій
IRQ0	08h	таймер
IRQ1	09h	клавіатура
IRQ2	0Ah	резерв
IRQ3	0Bh	COM2,COM4
IRQ4	0Ch	COM1,COM3
IRQ5	0Dh	жорсткий диск

IRQ6	0Eh	гнучкий диск
IRQ7	0Fh	LPT1

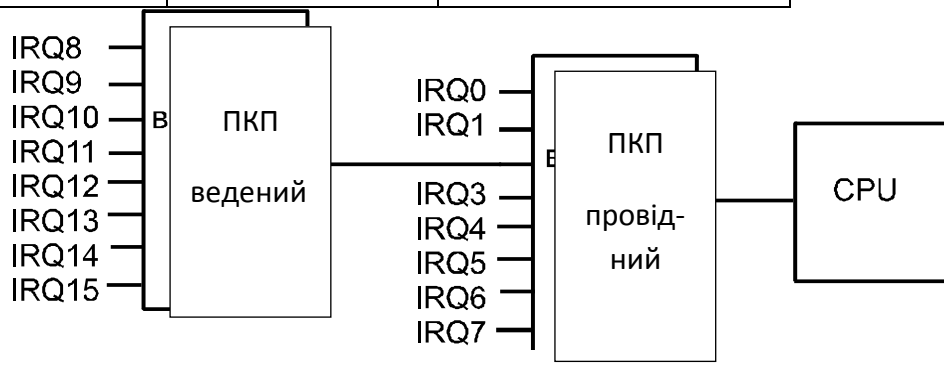


Рис. 1.29 Загальна схема з'єднання джерел переривань, ПКП і мікропроцесора в IBM PC/AT і наступних моделях

Розглянемо внутрішню структуру контролера. У ньому можна виділити наступні основні вузли (рис. 1.30):

- регістр запитів на переривання *(Interrupt Requests Register, IRR).* На входи регістру надходять сигнали з ліній IRQ.
- регістр маскування переривань *(Interrupt Mask Register, IMR).* Одиниця в розряді регістру блокує проходження сигналу по відповідній до лінії запиту.
- регістр оброблюваних запитів *(In-Service Register, ISR).* Одиниця в розряді регістру вказує на те, що обробляється переривання даного рівня.
- схему обробки пріоритетів *(Priority Resolver)*. Визначає запит з максимальним пріоритетом.

Табл. 1.6 Апаратні переривання IBM PC AT

Лінія запиту	Вектор переривання	Підключений пристрій
IRQ0	08h	таймер
IRQ1	09h	клавіатура
IRQ2	0Ah	ведений ПКП
IRQ8	70h	мікросхема КМОП
IRQ9	71h	переспрямовано на int 09h

IRQ10	72h	резерв
IRQ11	73h	резерв
IRQ12	74h	резерв (миша в PS/2)
IRQ13	75h	співпроцесор
IRQ14	76h	жорсткий диск
IRQ15	77h	резерв
IRQ3	0Bh	COM2,COM4
IRQ4	0Ch	COM1,COM3
IRQ5	0Dh	LPT2
IRQ6	0Eh	гнучкий диск
IRQ7	0Fh	LPT1

Сигнал від зовнішнього пристрою по лінії IRQ надходить на регістр запитів на переривання, установлюючи в одиницю відповідний розряд *(1).* Далі, якщо в регістрі маскування переривань розряд з тим же номером не встановлений в "1", сигнал проходить у схему аналізу пріоритетів *(2).* Якщо в цей момент не виконується переривання з більш високим пріоритетом, сигнал запиту переривання надходить на вхід регістру запитів, що обслуговуються, де дозволяє установку відповідного біта в "1" (але не встановлює його), і одночасно по лінії INT у *мікропроцесор (3).*

Мікропроцесор реагує на вступ сигналу INTтільки в тому випадку, якщо в регістрі прапорів установлений прапор IF дозволу переривань (таким чином, скинувши прапор IFкомандою CLI, можна заборонити всі апаратні переривання, що регіструються контролером переривань). Одержавши сигнал INT, мікропроцесор завершує обробку поточної команди, і переходить до обробки переривання, що починається з посилки контролеру переривань сигналу INTA *(Interrupt Acknoledge*, підтвердження переривання), який установлює дозволений раніше біт регістру запитів, що обслуговуються, і скидає біт регістру *запитів (4).*

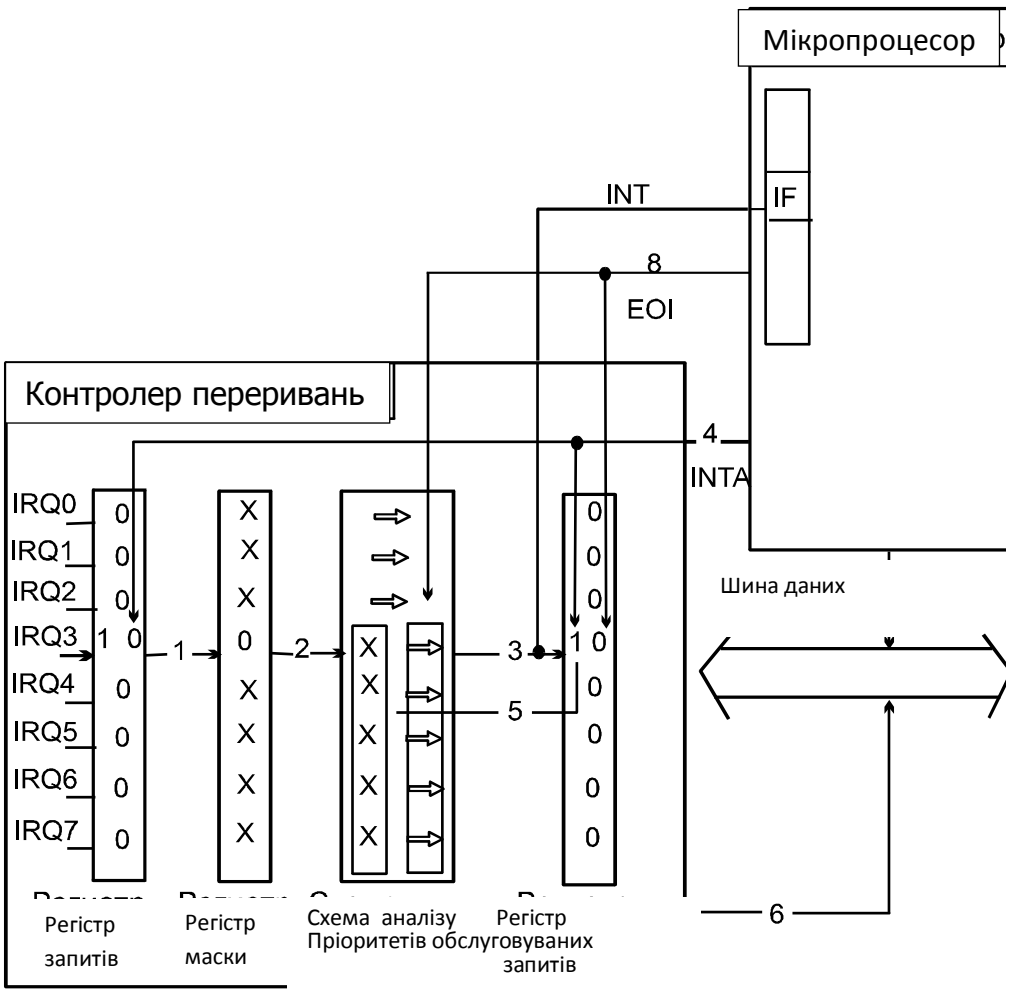


Рис. 1.30 Схема взаємодії ПКП і мікропроцесора

Із цього моменту переривання переводиться в розряд обслуговуваних, а на регістрі запитів може реєструватися й чекати обслуговування новий сигнал запиту переривання, можливо, від того ж пристрою. Після установки біта в регістрі обслуговуваних переривань, у схемі аналізу пріоритетів блокуються всі переривання, починаючи з поточного рівня й нижче по ланцюжкові *пріоритетів (5)*. Після одержання другого сигналу підтвердження від процесора по лінії *INTA*, ПКП виставляє на шину 8-бітовий номер вектора переривання *(6)*, яке одержало обслуговування (BIOS програмує провідний контролер на номери 0-7, а ведений — на номери 8-15). Далі процесор запам'ятовує в стеці програми, що переривається, значення регістру прапорів, лічильника команд (IP) і регістру адреси кодового сегмента (CS), потім скидає прапор IF. В останні два регістри розміщаються значення, що

зберігаються у векторі переривання, номер якого процесор читає на шині даних *(7)*. Після цього починається виконання оброблювача переривання.

Для забезпечення коректної роботи системи, оброблювач переривання повинен зберегти вміст усіх регістрів процесора для того, щоб при виході з оброблювача перервана програма могла продовжити своє функціонування. Звичайно процедура обробки переривання зберігає всі змінювані регістри в стеці програми, що переривається, до першої їхньої зміни, а потім відновлює їх перед виходом. Із цього випливає, що стек будь-якої програми, яка може бути перервана, повинен містити достатньо вільного місця для збереження в ньому адрес повернення й регістрів процесора декількома перериваннями. Через те, що переривання можуть бути вкладеними й дуже рідко створюють свій стек, то при вкладених перериваннях усі оброблювачі можуть зберігати дані в одному стеці програми.

Виходячи зі сказаного, можна вважати, що значення регістрів CS, IP і Flags, що розміщають в стек при ініціалізації переривання, а також інших регістрів процесора, що зберігаються оброблювачем, становлять PSW процесора i80x86 у реальному режимі. При цьому старе PSW запам'ятовується в стеці програми, що переривається, а нове PSW перебуває у векторі викликуваного переривання й має скорочений вигляд (у реальному режимі система виконує тільки одну програму процесом, що перериває, може бути тільки оброблювач переривання, а він завжди виконується з початку, тому нове PSW не містить ініціалізуючих значень регістрів). Нове PSW заміщає старе в регістрах процесора при перемиканні контексту.

Обробка переривання завершується після виконання процесором **команди** *IRE T* **повернення з оброблювача переривання**. Процесор відновлює значення збережених у стеці регістрів, при цьому встановлюється прапор IF і переривання знову дозволяються. Однак, багато оброблювачів переривань містять команду STI дозволу переривань у своєму тілі, дозволяючи, таким чином, переривання з більш високим пріоритетом. При цьому необхідно бути впевненим у тому, що при виникненні вкладених переривань не виникнуть

конфлікти. Звичайно ця умова виконується, і переривання з вищим пріоритетом (які мають більшу важливість для нормального функціонування системи) одержують можливість бути обробленими.

Що стосується переривань даного й нижчого рівнів пріоритетів, то вони залишаються заблокованими схемою аналізу пріоритетів, навіть після завершення обробки переривання. Для дозволу цих переривань існує спеціальна **команда завершення переривання** *(End Of Interrupt, EOI),* яка полягає в надсиланні певного коду на керуючий регістр контролера переривань (8). Ця команда скидає біт у регістрі обслуговуваних переривань і дозволяє проходження сигналів запиту переривань через схему аналізу пріоритетів. Звичайно така команда завершує роботу оброблювача переривання, однак вона може розташовуватися в будь-якому місці усередині оброблювача. В останньому випадку можливо вкладене переривання того ж рівня, що й оброблюване, тобто буде виконуватися та ж сама процедура обробки, що обробляється в теперішній момент. Така процедура повинна задовольняти певним вимогам, вона повинна бути реєнтерабельної. Турбота про це лягає на програміста. Слід також пам'ятати, що якщо команду STI в оброблювачі задавати не обов'язково (тому що при поверненні з переривання команда IRET відновлює зі стека значення регістру прапорів, збережене в момент виклику переривання, із установленим прапором IF), то команду завершення переривання (EOI) потрібно подати обов'язково, інакше будуть заборонені переривання на схемі аналізу пріоритетів, а поточне переривання залишиться в стадії виконання.

1.20.2.2. ***Немасковані переривання***

Існує одне апаратне переривання, відмінне від усіх інших. Це **немасковане переривання***(Non-Maskable Interrupt, NMI).* Його відмінність полягає в тому, що хоч ініціатором його і є зовнішній пристрій, сигнал запиту надходить у процесор, минаючи контролер переривань. Тому такий

запит на переривання неможливо замаскувати. Крім цього, запит по лінії NMI викликає негайну реакцію процесора й має максимальний пріоритет у системі, хоча оброблювач NMI може бути перерваний будь-яким маскованим перериванням, якщо їх дозволити командою STI. Це дозволяє використовувати дане переривання в особливо критичних ситуаціях: звичайно воно використовується при спаданні напруги живлення й при помилках пам'яті, однак на деяких моделях IBM – сумісних комп'ютерів NMI використовується також для обслуговування співпроцесора, клавіатури, каналів уведення/виводу, дискових контролерів, часів реального часу, таймерів, контролерів прямого доступу до пам'яті та ін.

У процесорах i80286 і вище у випадку, якщо під час обробки NMI виникне повторний запит на немасковане переривання, він не перерве виконання поточного оброблювача, а запам'ятається й буде оброблений після завершення поточного оброблювача. Якщо повторних запитів під час виконання оброблювача буде декілька, то збережеться тільки перший, наступні будуть проігноровані. При поверненні з оброблювача командою IRET процесор не переходить на виконання наступної команди, а намагається виконати команду по тійсамій адресі, що призвела до виникнення критичної ситуації.

1.20.2.3. *Програмні переривання*

Програмні переривання ініціюються спеціальною командою процесору INT n, де n указує номер переривання, оброблювач якого необхідно викликати. При цьому в стек заноситься вміст регістрів прапорів, вказівника команд і сегмента коду, а прапор дозволу переривання скидається (аналогічно тому, як при обробці апаратних переривань).

Далі в процесор завантажується нове PSW, значення якого міститься у векторі переривання з номером n, і керування переходить до оброблювача переривання. Вимоги до оброблювача програмного переривання такі ж, як і

до оброблювача апаратного переривання, за винятком того, що команда завершення апаратного переривання (EOI) не потрібна. Обробка переривання завершується командою IRET, що відновлює зі стека старе PSW.

Обробка програмного переривання є більш м'яким видом переривання стосовно програми, що переривається, ніж обробка апаратного переривання, тому що воно виконується на вимогу самої програми, що переривається, і вона очікує від цього якого-небудь результату. Апаратне ж переривання одержує керування без усякого відома програми, що виконується, і найчастіше не виявляє на неї ніякого впливу.

1.20.2.4. *Підтримка переривань в BIOS*

BIOS бере на себе основну роботу із програмування периферійних пристроїв комп'ютера, у тому числі й контролера переривань. Крім цього BIOS установлює оброблювачі всіх апаратних переривань, а також надає безліч інших функцій по керуванню практично всіма пристроями комп'ютера за допомогою програмних переривань. Адреси цих оброблювачів BIOS заносить в таблицю векторів переривань.

Усі службові функції BIOS реалізовані в оброблювачах програмних переривань і можуть бути викликані за допомогою команди переривання *INT* із вказівкою номера переривання. Звичайно одне переривання містить у собі не одну, а кілька функцій, що виконують подібні дії або працюючих з одним пристроєм. Такий підхід дозволяє реалізувати велику кількість функцій, використовуючи набагато меншу кількість переривань, що дуже важливо, тому що кількість переривань, що обслуговуються системою, обмежене. Список програмних переривань, надаваних BIOS, наведено в таблиці 1.7.

Табл. 1.7 Службові переривання BIOS

Номер переривання	Призначення
05h	Друк екрана
10h	Робота з відеоадаптером
11h	Одержання списку підключеного встаткування
12h	Одержання розміру пам'яті
13h	Функції роботи з дисками
14h	Робота з COM-портом
15h	Робота з касетним магнітофоном і додаткові функції BIOS
16h	Доступ до клавіатури й буферу клавіатури
17h	Функції роботи із друкувальним пристроєм
18h	Виклик недискового завантажника (напр. ROM-BASIC в PC фірми IBM)
19h	Виклик програми початкового завантаження
1Ah	Робота з годинником реального часу
1Bh	Обробка натискання Ctrl-Break
1Ch	Викликається оброблювачем переривання таймера (IRQ0, INT 08h)
1Dh	Доступ до таблиці параметрів відеосистеми
1Eh	Доступ до таблиці параметрів дисків
1Fh	Доступ до області опису шрифту 8x8
4Ah	Оброблювач "будильника" годин реального часу
Feh,Ffh	Використовуються при поверненні в реальний режим i80286

1.20.2.4. *Переривання в MS-DOS*

Система MS-DOS активно використовує механізм переривань. Усі функції, надавані ОС, як і функції BIOS, реалізовані саме у вигляді програм

обробки переривань. Якщо деякому процесу потрібен сервіс операційної системи, він повинен викликати програмне переривання (аналог звертання до супервізора), передавши через регістри загального призначення мікропроцесора необхідні параметри DOS і функції DOS, викликуваної процедури.

Дотримуючись термінології довідкового керівництва MS-DOS, службові процедури операційної системи розділені на дві категорії: переривання DOS мають унікальні номери програмних переривань, а для виклику функції необхідно викликати переривання й конкретизувати номер функції, задавши його в одному з регістрів (звичайно AX).

MS-DOS 3.00 надає переривання 20h-28h і 2Fh, п'ять із них (20h, 25h-27h, 2Fh) є "справжніми" (що містять одну процедуру оброблювача) перериваннями DOS; усі функції реалізовані в оброблювачі переривання 21h; три переривання (22h-24h) містять адреси процедур обробки виняткових ситуацій, які можуть настроюватися кожною програмою. Існують також недокументовані переривання й службові функції, частково документовані в наступних версіях MS-DOS. Нові версії MS-DOS привнесли також і нові переривання (табл.2.8.):

Зміна векторів переривань досить часто використовується при написанні програм що залишаються в пам'яті, резидентно. Однак, при написанні таких програм слід приділяти особливу увагуреєнтерабельності, адже, одержуючи керування, резидентний процес перериває інший процес, яким може бути й системний оброблювач переривання. Але переривання MS-DOS не є реєнтерабельними (реєнтерабельність переривань BIOS замовчується, тобто звичайно проблем з функціями BIOS не виникає, але все-таки гарантії на цей рахунок немає), тому використовувати функції MS-DOS можна тільки переконавшись, що перерваний процес не був системним. Для цього можна використовувати функцію 34h переривання 21h, яка документована починаючи з версії MS-DOS 5.0, але існує й у більш ранніх версіях. Вона повертає в ES:BX адресу прапора InDOS, що сигналізує про виконання

системного процесу. Іншою досить корисної недокументованою функцією MS-DOS є функція 52h переривання 21h, що повертає в ES:BX адресу "списку списків" *(List of lists)* інакше називаного блоком змінних DOS *(DOS vars block)*, у якому утримуються адреси керуючих таблиць MS-DOS і покажчики на системні драйвери.

Табл. 1.8 Переривання MS-DOS

Номер переривання	Призначення переривання
20h	Завершити програму
21h	Службові функції MS-DOS
22h	Адреса процедури, що одержує керування після завершення програми
23h	Оброблювач натискання Ctrl-Break
24h	Оброблювач критичної помилки (апаратури)
25h	Абсолютне читання диска
26h	Абсолютний запис на диск
27h	Резидентне завершення програми
28h	Викликається при очікуванні введення із клавіатури
29h	Вивід символу на екран
2Bh-2Dh	Зарезервовані
2Fh	Мультиплексне переривання
30h	Міжсегментний перехід на процедуру підтримки CP/M

Операційна система дозволяє створювати нові, змінювати, доповнювати й повністю заміщати будь-які, у тому числі й власні, оброблювачі переривань. Це дозволяє досить гнучко настроїти систему для використання з різними пристроями й для оптимального виконання певних завдань. Хоча MS-DOS і надає кошти для роботи з векторами й оброблювачами переривань, але й вектора переривань, і процедури обробки переривань можна змінювати, минаючи операційну систему. Це сприяє поширенню в такому середовищі, як

MS-DOS, комп'ютерних вірусів, а також створює проблеми при спільному використанні одного ресурсу декількома процесами.

1.20.2.5. ***Використування власних обробників переривань***

Хоча BIOS і операційна система надають дуже широкий спектр службових процедур, нерідко при використанні нестандартних пристроїв для розширення функціональності стандартних пристроїв чи системних функцій виникає необхідність в написанні власних обробників переривань. Новий обробник може або повністю замінити системний, або доповнити його. Другий варіант застосовується частіше, оскільки його легше реалізувати.

Існують 3 основних метода доповнення існуючого обробника переривань:

1. Виконати додаткові дії для виклику існуючого обробника
2. Виконати додаткові дії після виклику існуючого обробника
3. Виконати додаткові дії до и після виклику існуючого обробника

Для зміни (доповнення) процедури обробки переривання необхідно змінити адресу цього обробника в таблиці векторів переривань. Це можливо зробити, безпосередньо записав адресу нового кода в таблицю чи при допомозі функції MS-DOS. Якщо новий обробник буде лише доповнювати старий, то попередньо необхідно зберегти адресу старого обробника для того, щоб мати можливість його виклику. Отримати цю адресу також можливо двома шляхами: прочитав його значення в таблиці векторів або викликав функцію MS-DOS.

Припустимо, що ми бажаємо добавити свій код до обробки переривань клавіатури. Тоді змінення вектора переривання може виглядати наступним чином:

```
OldInt9h DD ?          ; адреса старого обробника переривань
PROC NewInt9h          ; новий обробник
    . . . . . .
ENDP
    . . . . . .
```

```
    MOV AX,0                ; занесення в ES сегмента таблиці векторів
(0000)
    MOV ES,AX
    CLI
    MOV AX,[ES:9*4]                 ; отримання і збереження адреси
    MOV [WORD PTR OldInt9h],AX      ;  старого обробника
    MOV AX,[ES:9*4+2]
    MOV [WORD PTR OldInt9h+2],AX
    MOV AX,OFFSET NewInt9h          ; занесення нової адреси
    MOV [WORD PTR ES:9*4],AX
    MOV AX,SEG NewInt9h
    MOV [WORD PTR ES:9*4+2],AX
    STI
. . . . . .
```

В даному прикладі функції MS-DOS не використовуються, тому фрагмент кода по звертанню до таблиці векторів оточена командами CLI і STI для того, щоб виключити можливість змінення таблиці векторів другим процесом в цей же час.

Того ж результату можна досягнути, використовуючи функції MS-DOS (INT 21h): 35h и 25h для змінення вектора переривання. Обидві функції потребують задання в регістрі AL номера вектора, функція 35h повертає значення цого вектора (сегмент: зміщення) в регістрах ES:BX, а 25h записує в указаний вектор значення регістрів DS:DX. Функції DOS гарантують коректне використання таблиці векторів, тому команди CLI и STI використовувати не потрібно:

```
. . . . . .
OldInt9h DD ?               ; адреса старого обробника переривання
PROC NewInt9h               ; новий обробник
    . . . . . .
ENDP
```

```
. . . . . .
MOV AX,3509h          ; отримання і збереження адреси
INT 21h               ; старого обробника
MOV [WORD PTR OldInt9h],BX
MOV [WORD PTR OldInt9h+2],ES
MOV AX,2509h          ; занесення нової адреси
MOV AX,SEG NewInt9h
MOV DS,AX
MOV DX,OFFSET NewInt9h
INT 21h
. . . . . .
```

Якщо додаток до обробника викликається після системної процедури обробки переривання, необхідно передати їй управління таким чином, щоб після виконання команди IRET системного обробника переривань управління перейшло до доданого коду. Для цього перехід на старий обробник виконується командою CALL з атрибутом FAR. Хоча ця команда є у всіх процесорах i80x86, вона не реалізована в багатьох Асемблерах, тому її необхідно задавати безпосередньо її машинним кодом: 9Ah <довга адреса виклику>. Команда CALL зберігає в стеку тільки регістри IP і CS, а IRET витягує з стека ще й Flags, то перед викликом CALL необхідно помістити в стек регістр прапорів командою PUSHF.

З урахуванням сказаного, обробник переривань може мати наступний вигляд:

```
PROC NewInt9h       ; новий обробник
   PUSHF
   DB 9Ah      ; CALL FAR
OldInt9h DD ?          ; адреса старого обробника переривань
   . . . . . .         ; доданий код
   IRET
ENDP
```

Якщо ж новий код додається перед викликом системного обробника,

можна використовувати команду JMP FAR, яку також потрібно ставити її машинним кодом: 0EAh <довга адреса переходу>. При цьому, після виконання старого обробника, команда IRET виконає повернення в перервану програму, а не в доданий обробник.

Розглянемо приклад розширення процедури обробки переривання клавіатури з метою виконання спеціальних дій при натисненні на клавішу ScrollLock:

```
IDEAL
MODEL TINY
CODESEG
    STARTUPCODE
    JUMP Init
PROC NewInt9h       ; новий обробник
    PUSHF                   ; збереження змінюваних регістрів
    PUSH AX
    IN AL,60h       ; зчитування порта клавіатури
    CMP AL,70
    JNE _Jump
    . . . . . .     ; дії при натисненні ScrollLock
_Jump:
    POP AX          ; відновлення зміненних регістрів
    POPF
    DB 0EAh         ; JMP FAR
OldInt9h DD ?       ; адреса старого обробника переривання
ENDP
Init:
    MOV AX,3509h            ;  отримання и збереження адреси
    INT 21h                 ;  старого обробника
    MOV [WORD PTR OldInt9h],BX
```

```
    MOV [WORD PTR OldInt9h+2],ES
    MOV AX,2509h              ; занесення нової адресси
    MOV DX,OFFSET NewInt9h
    INT 21h
    MOV AX,3100h              ; завершення резидентної програми
    MOV DX,OFFSET Init        ; обчислення розміру резидентної
                              ; частини
    MOV CL,4                  ; в параграфах
    SHR DX,CL
    INC DX
    INT 21h
ENDS
END @StartUp
```

Новий обробник переривання читає порт 60h, в якому при виклику INT 9 знаходиться скан-код натиснутої клавіші, і порівнює його з кодом ScrollLock (70). Якщо була натиснута клавіша ScrollLock, виконуються відповідні дії, після чого командою JMP FAR здійснюється перехід в системний обробник, який і завершує обробку переривання, тому в новій процедурі не використовується команда IRET.

Необхідно пам'ятати, що обробник переривання повинен зберігати всі змінювані їм регістри процесора і відновлювати їх при виході, крім тих випадків, коли через регістри передається результат обробки переривання.

У випадку, якщо новий обробник повністю замінює колишній, необхідно пам'ятати, що при вході в процедуру обробки забороняються всі переривання, тому по можливості їх потрібно дозволити командою STI. Крім того, якщо новий обробник обслуговує апаратне переривання, необхідно перед його завершенням послати контролеру переривань команду EOI. Це робиться наступним чином:

– при обробці запитів головного ПКП (IRQ0-IRQ7) — командами

```
MOV AL,20h
```

OUT 20h,AL

– при обробці запитів другого ПКП (IRQ8-IRQ15) — командами

MOV AL,20h

OUT 0A0h,AL

1.20.2.6. Реалізація виконання операцій введення / виводу в MS-DOS

- *Ввід/вивід через COM —порт*

Розглянемо взаємодію користувальницької програми, операційної системи та апаратури на прикладі виконання введення з зовнішнього пристрою, підключеного до послідовного інтерфейсу. У IBM PC такий інтерфейс відповідає стандарту RS-232-C. Зазвичай в комп'ютері встановлено 2 або 4 порта, що позначаються COM1-COM4, проте порти можуть формувати запити на переривання тільки по двох лініях: IRQ3 (COM2 і COM4) і IRQ4 (COM1 і COM3). Тому не можна одночасно працювати з портами COM1 і COM3 або COM2 і COM4, використовуючи апаратні переривання (можна працювати з одним портом в режимі програмного опитування, а з іншим - або через переривання, або в режимі опитування).

Інтерфейс RS-232-C реалізований у вигляді семи регістрів, базовий адресу (адреса першого регістра) для COM1 зазвичай 3F8h (адреси портів можна дізнатися в таблиці параметрів BIOS за адресою 0040:0000 h):

Перший регістр (3F8h) використовується для читання при прийомі і для запису при передачі даних, а також для встановлення швидкості обміну COM-порту з підключеним до нього пристроєм.

Другий регістр (3F9h) управляє режимом видачі сигналу запиту переривання. Можливі наступні режими:

1. переривання можливо при готовності прийнятих даних;

2. переривання можливо після передачі останнього біта з буфера при передачі;

3. переривання можливо при виявленні помилки і при отриманні сигналу BREAK;

4. переривання можливо при зміні стану вхідних ліній COM-порту.

Якщо відбулося переривання від COM-порту, то його причину можна дізнатися, прочитавши третій регістр (за адресою 3FAh).

Четвертий регістр (3FBh) є керуючим: записана в ньому інформація визначає довжину переданих слів, кількість стоп-біт, тип контролю.

Решта регістри використовуються для управління лінією і підключеним до COM-порту пристроєм.

MS-DOS надає 2 функції для роботи з COM-портом:

1. читання символу з COM-порту - функція 03 переривання 21h;

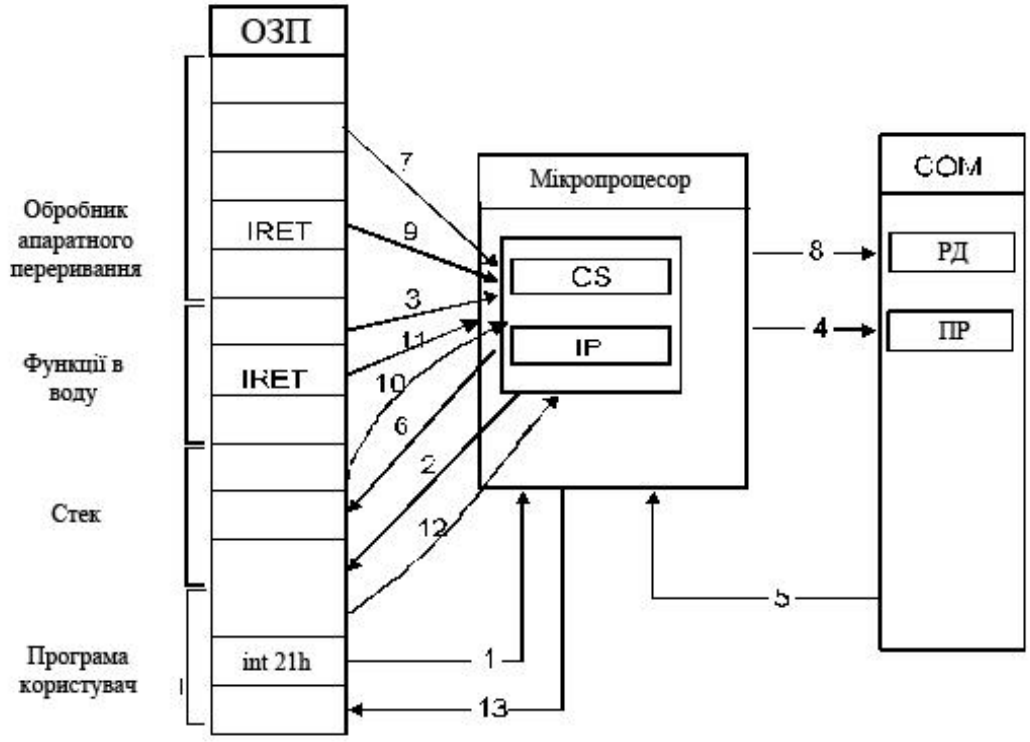
2. запис символу в COM-порт - функція 04 переривання 21h.

Розглянемо малюнок 2.31. Програма користувача, що бажає ввести символ з COM-порту, повинна завантажити в регістр AH номер функції - 03 і подати (1) команду INT 21h. При цьому поточне PSW зберігається (2) в стеку, а його місце заміщає (3) адреса процедури введення MS-DOS, яка ініціалізує (4) керуючий регістр. Далі система очікує отримання COM-портом символу, про що сповіщає сигнал на лінії IRQ4, що викликає (5) обробник переривання 0Ch. Для цього поточне PSW зберігається (6) в стеку, його замінює (7) нове PSW обробника апаратного переривання, який визначає причину переривання і читає (8) дані з регістра даних. Після цього робота обробника завершується (9) і управління повертається (10) функції введення, яка поміщає введений через COM-порт символ в регістр AL і повертає (11) управління користувальницької програмі. Її PSW відновлюється (12) з стека. Виконується (13) наступна за INT 21h команда, і програма може прочитати надійшов в порт символ в регістрі AL.

- *Введення даних з клавіатури*

При натисканні або відпущенні клавіші клавіатура посилає контролеру переривань по лінії IRQ1 запит на обробку переривання і встановлює в порту з номером 60h скан-код натиснутої (відпущеної) клавіші. Якщо переривання даного рівня не замасковано і дозволено, контролер посилає вимогу переривання процесору. Процесор перевіряє стан прапора IF і, якщо прапор

не встановлений, підтверджує запит контролера. Контролер виставляє на шину даних номер вектора, який повинен вказувати на обробник переривання клавіатури (в реальному режимі це зазвичай INT9), а процесор запам'ятовує адресу повернення і регістр прапорів в стеку активної програми, а потім поміщає в регістри CS і IP значення, що знаходяться на початку оперативної пам'яті в таблиці векторів, у зазначеному КПП векторі переривань. Після цього управління отримує обробник переривання.



Мал. 1.31 Виконання операції введення з COM-порта

Процедура обробки переривання читає з порту 60h скан-код і визначає, чи була клавіша натиснута або відпущена, а також з якою саме клавішею було вироблено цю дію. Якщо дії проводилися з клавішею перемикачем (Insert, CapsLock і т.п.) або зі спеціальною клавішею (Ctrl, Shift, Alt), обробник змінює статус (натиснута, відпущена) цієї клавіші в спеціальному слові стану клавіатури, що знаходиться в області даних BIOS по адресою 0040:0017. Крім цього, обробник перевіряє також, чи не була натиснута спеціальна клавіша (або комбінація клавіш): PrtScr, SysRq, Pause, Ctrl-Alt-Del, Ctrl-Break, Ctrl-NumLock та інші - і, якщо це так, обробник викликає інше

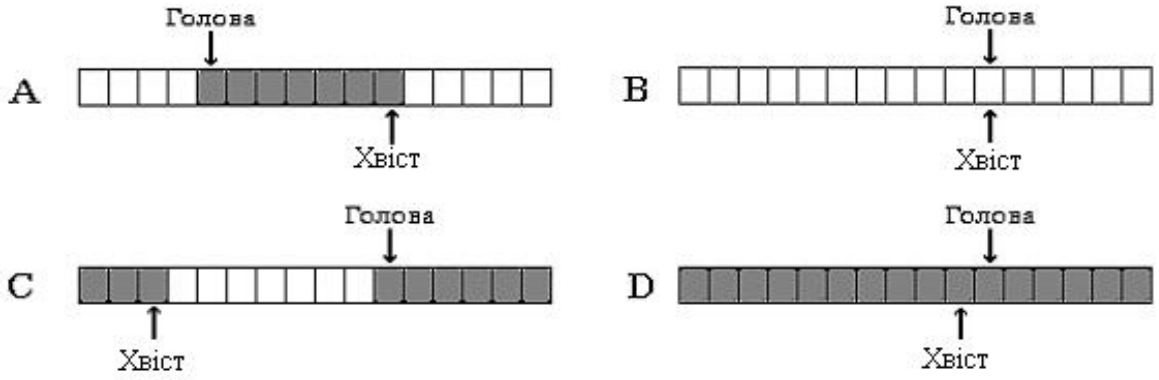
переривання, обробляє відповідну ситуацію (наприклад, 1Bh при натисканні Ctrl-Break).

В інших випадках обробник зіставляє натиснуту клавішу зі станом клавіш перемикачів і формує відповідний код (ASCII для клавіш з символом ASCII або розширений для не символьних клавіш і комбінацій різних клавіш з Shift, Ctrl та Alt). Цей код має довжину 2 байта і містить:

– для ASCII символу його ASCII код в молодшому байті і скан-код клавіші в старшому;

– для розширеного коду його значення в старшому байті і 0 в молодшому.

Цей код заноситься в спеціальний буфер клавіатури, також знаходиться в області даних BIOS (мал. 2.32). Буфер має дисципліну обслуговування FIFO і розмір 32 байта, отже, він може зберегти значення 16-ти натискань клавіш. Значення в буфер поміщаються послідовно (спочатку молодший байт, за ним старший) і циклічно, тобто при досягненні кінця буфера значення поміщаються в його початок. Для визначення початку і кінця черги маються покажчики на початок (голову) і кінець (хвіст). Ці покажчики зберігаються по адресам 0040:001 A і 0040:001 C відповідно і містять зміщення від початку сегмента 0040:0000 (мал. 1.32.A).



Мал. 1.32 Буфер клавіатури

При запису нового значення в буфер покажчик на хвіст збільшується на 2 і записується значення поміщається за цією адресою. При читанні з буфера витягується значення, що знаходиться за адресою, що міститься в покажчику на початок, а потім цей покажчик збільшується на 2. Якщо значення

покажчиків однакові, буфер вважається порожнім (мал. 1.32.B). Якщо який-небудь покажчик досягає кінця буфера, він переходить на початок, таким чином покажчик на хвіст може мати менше значення, ніж покажчик на голову (мал. 2.32.C). При досягненні покажчика на кінець буфера значення, на 2 меншого, ніж значення покажчика на початок, буфер вважається заповненим, і нове значення ігнорується (мал. 1.32.D).

Таким чином, програми, які виконують введення з клавіатури, насправді зазвичай читають буфер клавіатури. Значення, що знаходяться в буфері, можна отримати за допомогою функцій BIOS (INT 16h) або MS-DOS (INT 21h, функції 01, 06, 07, 08, 0Ah, 0Bh, 0Ch, 3Fh). Ці функції дозволяють також отримувати стан буфера і змінювати його.

- *Введення команд для виконання*

Розглянемо інший приклад, що демонструє роботу системи переривань, а саме, відстеження командним процесором вводяться користувачем з клавіатури команд. При цьому виконується наступна послідовність дій:

Користувач набирає на клавіатурі ім'я і параметри завдання. При натисканні кожної клавіші клавіатура посилає запит на переривання, він обробляється процедурою INT 9h, яка заносить значення введеного символу в буфер клавіатури.

Командний процесор в цей же час перевіряє буфер клавіатури на наявність в ньому введених даних, і читає їх у міру появи (за допомогою функцій MS-DOS). Так як ці процедури виконуються набагато швидше, ніж введення символів на клавіатурі, то велика частина процесорного часу при очікуванні введення витрачається без користі. Для підвищення ефективності роботи системи функції MS-DOS введення з клавіатури при очікуванні викликають спеціальне мультиплексному переривання INT 28h, яке виконує фонові процеси (наприклад, друк файлів на принтер командою PRINT). Отримавши символ CR (Enter), який свідчить про закінчення введення завдання, командний процесор перевіряє його коректність і, в разі цього, запускає команду чи програму на виконання за допомогою відповідного переривання.

1.20.2.7. *Реалізація системи переривань в операційних системах Microsoft Windows 95(98)*

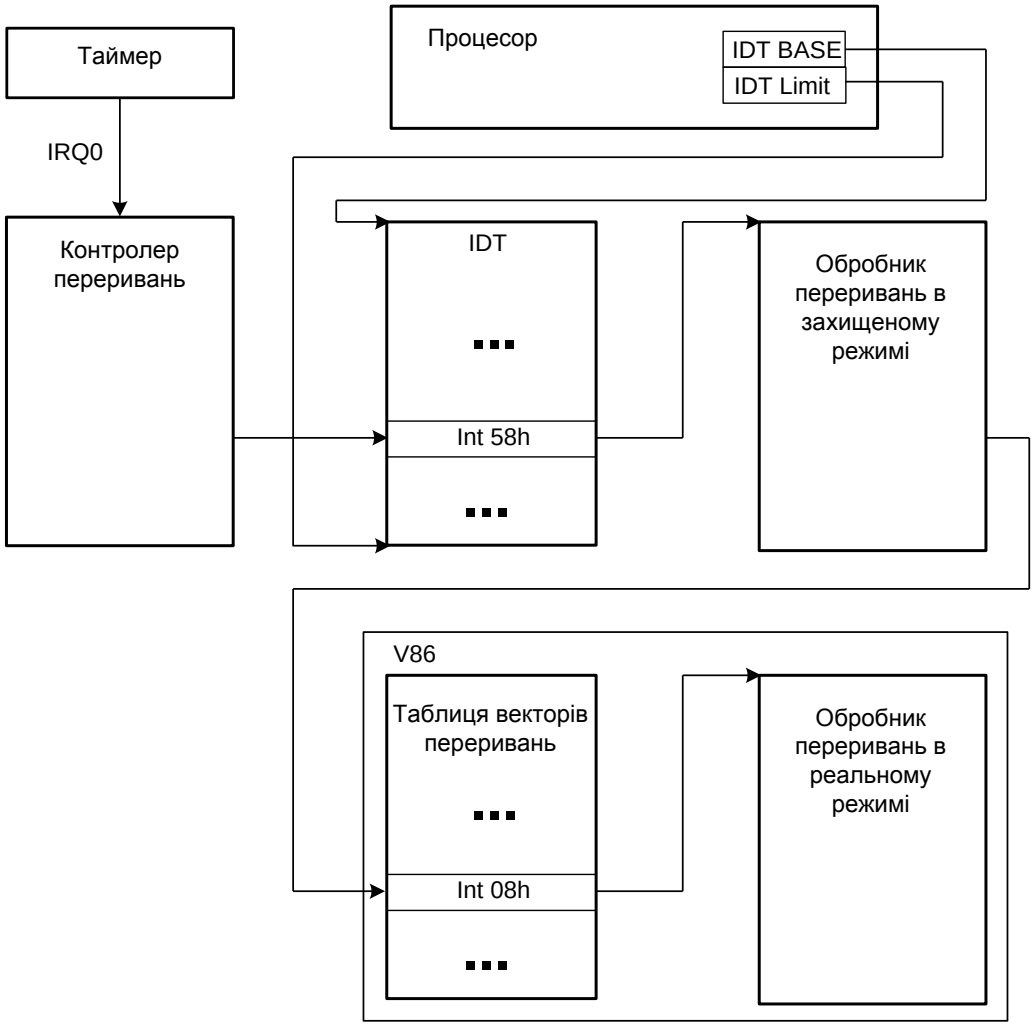
Операційна система (ОС) Microsoft Windows'95 (98) є багатозадачною операційною системою для однопроцесорних платформ, побудованих на базі процесорів 80386 і вище. При її ініціалізації спочатку завантажується ОС MS-DOS (для сумісності з попередніми версіями), а потім завантажується ОС Windows з послідовним витісненням драйверів і базових підпрограм DOS і BIOS 32-бітовим кодом, що виконуються в захищеному режимі. Це можливо, внаслідок використання механізму драйверів віртуальних пристроїв (VxD), що має ряд переваг і недоліків. Цей механізм дозволяє динамічно (прямо під час роботи операційної системи) підключати нові пристрої, вивантажувати непотрібні драйвери, замінювати і доповнювати вже завантажені. Це надає гнучкість системі і дозволяє досить легко підключати нове устаткування і істотно полегшує завдання розробки нових драйверів. Проте ця технологія має на увазі виконання коду цих драйверів в кільці з нульовим рівнем привілеїв. При цьому в кільці з нульовим рівнем привілеїв з'являється досить багато коду, який часто ніколи не виконується, а в зв'язку з тим, що він розроблений різними фірмами - виробниками, в ньому можливо виникнення конфліктів між різними драйверами, що зазвичай призводить до зависання операційної системи.

Переривання, як уже було сказано, діляться на три категорії: програмні, апаратні та переривання у виняткових ситуаціях, що виникають внаслідок появи помилок або відмов. Програмні переривання підтримуються в 16-бітових (NE) додатках і в програмах, написаних під DOS. У 32-бітових (PE) додатках немає можливості виклику програмних переривань. При цьому виникає переривання неправильний код операції і керування передається обробнику винятків, що виконуються в кільці нульового рівня привілеїв, що дозволяє в деяких випадках зберегти систему від некоректно написаних програм.

Для підтримки старого програмного забезпечення в ОС Windows використовується дворівнева система переривань: спочатку викликаються переривання захищеного режиму, що визначаються таблицею IDT, а потім у разі потреби, їх обробники викликають переривання реального режиму, описані в попередньому розділі, в спеціально створюваної системної віртуальній машині V86 (рис .1.33). Наприклад, при виклику переривання таймера IRQ0, викликається переривання захищеного режиму Int 58h, яке надходить на обробники внутрішніх віртуальних таймерів, на планувальник завдань. Потім програмно імітується переривання Int 8h (адреса його обробника береться з таблиці переривань в цій віртуальній машині, яка розташована на початку її адресного простору починаючи з адреси 0000h: 0000h) у системному віртуальної машині V86, яке використовується для зміни показань системних годин і для потреб програм, завантажених резидентно до ОС Windows. Системна VM завжди має номер 1 і ніколи не вивантажується з пам'яті. Коли в пам'яті присутні інші V86, то відбувається виклик Int 8h в кожній з них, що дозволяє використовувати прикладні програми, що працюють під DOS і використовують переривання таймера.

Існує і зворотний зв'язок: DOS програми можуть викликати переривання захищеного режиму, через драйвера віртуальних пристроїв. Так, наприклад, при відкритті файлу на диску програма викликає INT 21h з AH = 3Dh. Оброблювач цього переривання при завантаженні Windows підміняється обробником з драйвера VMM.VxD, частина з якого виповнюється в режимі V86, а частина в захищений режимі. При виклику INT 21h спочатку в режимі V86 виконується обробка переданих параметрів, приведення імені файлу до канонічного виду, визначення режиму доступу до файлу і т.д. Потім за допомогою спеціальної інструкції APPL управління передається в захищений режим до драйвера файлової системи (зазвичай IFS.VxD), яка викликає переривання по виняткової ситуації через неправильний код операції (ця інструкція недопустима в режимі V86), за якою VMM перемикається в

захищений режим і продовжує там подальшу обробку функції відкриття файлу.



Мал. 1.33

У режимі емуляції MS-DOS або в сеансі, запущеному під MS Windows, для підтримки програм, написаних під DOS, в пам'яті створюється образ віртуальної машини (V86), кожен екземпляр якої містить: таблицю переривань, область даних BIOS, область даних DOS, набір підпрограм, що обробляють програмні переривання і імітують виконання апаратних переривань (зазвичай покажчики на всі апаратні переривання встановлені на команду IRET), відеобуфер і свій локальний буфер клавіатури.

При виклику апаратних переривань процесор вибирає адресу процедури обробки переривання не з таблиці переривань, а з таблиці, на початок якої вказує регістр IDTR, доступний тільки з коду, що виконується з нульовими привілеями (нульовий - максимальний рівень привілеїв). Ця таблиця містить

список адрес процедур обробки переривань, які можуть бути як 16-ти так і 32-бітними і виконуються з максимальними привілей-ми. Обробники апаратних переривань розташовані в кільці нульового рівня привілеїв і при необхідності після обробки переривання імітують переривання DOS, отримуючи їх адреси з таблиці переривання для кожного сеансу DOS окремо для підтримки функціонування старих програм, наприклад резидентних русифікаторів, драйверів миші, комп'ютерних ігор, які самостійно обробляють переривання.

Обробники програмних переривань, що викликаються з режиму V86, мають посилання в IDT і при їх виклик управління отримують підпрограми операційної системи. Потім вони вибирають адресу процедури обробки переривання з таблиці переривань і викликають ці процедури. Вони зазвичай розташовані всередині блоку пам'яті, виділеного для конкретної VM і виконуються, як якщо б вони були викликані безпосередньо командою INT. Деякі з них все-таки обробляються в нульовому кільці, так як не можуть оброблятися в автономному режимі (наприклад, дискові операції, виділення EMS і XMS блоків пам'яті, процедури, безпосередньо працюють з апаратурою).

Крім цього, MS Windows надає для DOS-додатків набір функцій, що здійснюють зв'язок з програмами захищеного режиму. Наприклад, для визначення версії Windows можна використовувати наступний фрагмент програми:

```
mov ax,16h
int 2fh
```

Після виконання цих команд в регістрі AH буде номер версії Windows, а в регістрі AL - номер під версії. Для роботи з Clipboard з DOS-додатків існують такі функції переривання 2Fh: 1701h (відкрити Clipboard), 1702h (очистити), 1703h (помістити дані в Clipboard), 1704h (отримати розмір даних), 1705h (отримати дані). Наступний фрагмент програми демонструє отримання текстової рядки з Clipboard і виведення її на екран:

```
    mov ax,1701h        ; відкрити Clipboard
    int 2fh
mov ax,1705h            ; функція отримання даних
mov dx,01               ; використовуємо текстовий формат
lea bx,TextBuffer       ; ES:BX вказують на буфер для прийому рядка
int 2fh                 ; викликаємо мультиплексному переривання
test ax,ax
jz ClipFail             ; якщо відбулася помилка
mov di,offset TextBuffer
mov al,0
mov cx,255
repnz    scasb          ; шукаємо в рядку ознака кінця рядка (0h)
    mov byte ptr [di],'$'       ; замінюємо 0h на '$'
    mov ah,9h           ; функція виведення рядка на екран
    lds dx,TextBuffer
    int 21h
ClipFail:
    mov ax,1708h        ; закрити Clipboard
    int 2fh
```

У разі виконання 16-бітного коду в захищеному режимі допускається прямий виклик програмних переривань (так, наприклад, практично всі функції аналогії API можуть бути викликані через переривання Int 31h). При цьому не має значення, на 16-ти або 32-бітний код вказує процедура обробки переривання - вони завжди виконуються в своєму контексті. Апаратні переривання не дублюються для окремих додатків.

У 16-розрядних додатках існує можливість перехоплення переривань як реального, так і захищеного режиму. Так, наприклад, для отримання адреси процедури обробки переривання 21h реального режиму системної VM можна скористатися функцією DPMI 0200h:

```
mov ax, 0200h               ; функція отримання вектора переривання
```

```
mov bl,21h                ; номер вектора реального режиму
int 31h                   ; виклик DPMI
mov word ptr dwRet,dx     ; зсув
mov word ptr dwRet+2,cx   ; сегмент
```

Для установки нового обробника переривання реального режиму існує функція 0201h, яка використовується наступним чином:

```
mov ax, 0201h             ; функція установки вектора переривання
mov bl,21h                ; номер вектора реального режиму
mov cx,seg NewInt21h      ; сегмент
mov dx,offset NewInt21h   ; зсув
int 31h                   ; виклик DPMI
```

Після чого всі виклики переривання 21h реального режиму будуть передаватися в новий обробник переривань. Але іноді виникає необхідність перехоплювати вектора переривань захищеного режиму (наприклад, те ж переривання DPMI INT 31h). Для цих цілей можна скористатися функціями 204h і 205h переривання Int 31h:

```
mov ax,204h               ; отримати PM вектор переривання
mov bl,31h                ; номер вектора переривання захищеного режиму
int 31h                   ; виклик DPMI
mov word ptr OldInt31h,cx     ; селектор захищеного режиму
mov word ptr OldInt31h+2,dx   ; зсув
mov ax,205h                   ; встановити PM вектор переривання
mov bl,31h                    ; номер вектора переривання
mov cx,seg NewInt31h          ; селектор захищеного режиму
mov dx,offset NewInt31h       ; зсув
Int 31h                       ; виклик DPMI
```

Якщо обробник переривання 32-розрядний, то необхідно використовувати 32-розрядні зсуву, і вони повинні поміщатися не в регістр DX а в EDX.

Для імітації виклику переривання реального режиму існує функція 300h переривання int 31h. Її можна використовувати для зв'язку DOS-додатків з

Windows-додатками або для імітації виклику апаратного переривання. Наприклад, для імітації виклику переривання таймера в реальному режимі (так робить Windows 18.2 рази в секунду) використовується наступний фрагмент програми в захищеному режимі:

```
mov ax,300h          ;  зімітувати виклик переривання
mov bl,8             ;  номер вектора переривання
mov bh,0             ;  скинути лінію A20 (нема доступу до HMA)
mov cx,0             ;  не передавати нічого через стек
les di,CallStruct    ;  структура, що визначає регістри при виклику переривання (в даному випадку це не має значення, так як ніякі регістри переривання 8h не використовує)
int 31h              ; виклик DPMI
```

Якщо виникає необхідність передачі параметрів в викликається переривання, то для цього існує структура CallStruct, значення полів якої визначає вміст регістрів процесора при виклику зазначеного переривання:

Зсув	Розмір	Опис
00h	DWORD	EDI
04h	DWORD	ESI
08h	DWORD	EBP
0Ch	DWORD	зарезервовано (00h)
10h	DWORD	EBX
14h	DWORD	EDX
18h	DWORD	ECX
1Ch	DWORD	EAX
20h	WORD	flags
22h	WORD	ES
24h	WORD	DS
26h	WORD	FS
28h	WORD	GS
2Ah	WORD	IP

2Ch	WORD	CS
2Eh	**WORD**	**SP**
30h	WORD	SS

Якщо викликається переривання вимагає передачі параметрів через стек, то параметри просто поміщаються в стек захищеного режиму і в регістрі CX вказується їх розмір. При виклику переривання ці параметри будуть передані в стек реального режиму.

При виконанні 32-бітного коду програмний виклик переривань дозволений тільки з нульового кільця, в іншому випадку це призводить до виняткової ситуації - «недозволений код команди» переривання Int 6h. При цьому вибирається відповідний 8-байтовий дескриптор з таблиці переривань IDT, що адресується за допомогою регістра IDTR. Вміст (IDTR) вказує базову адресу і межі таблиці. Значення вектора переривань, помножене на 8, дає відносний адресу першого байта дескриптора в таблиці IDT. Дескриптор містить 16-розрядний селектор, який заноситься в CS, і 16-ти або 32-розрядний відносний адресу, який заноситься в IP або EIP. Крім того, дескриптор визначає число 16-ти або 32-розрядних параметрів з вершини стека, які при виконанні команд переривань переносяться в стек програми обслуговування переривань для використання або збереження.

Переривання з 32-бітної програми викликати не можна, але зате їх можна перехоплювати. Для цього досить з регістра IDT отримати адресу початку таблиці векторів переривань захищеного режиму, і у відповідний вектор занести нову адресу обробника переривань. Цей спосіб перехоплення переривань демонструє наступний фрагмент програми:

```
push    eax                 ; виділити в стеку 4 байта
sidt    [esp-02h]           ; Отримати базовий адресу IDT
pop     ebx                 ; отримати з стека вміст IDT
add     ebx, 5*08h+04h      ; обчислимо адресу вектора 6-го переривання (обробник неправильного коду операції)
cli                         ; забороняємо переривання
```

```
lea     esi,NewInt06h     ; встановлюємо вектор
mov     [ebx-04h], si     ; 6-го переривання
shr     esi, 16           ; на нашу процедуру обробки
mov     [ebx+02h], si     ; неправильного коду операції
sti                       ; дозволяємо переривання
```

Тепер при виконанні неприпустимою операції управління буде передано процедурі NewInt06h, а не операційній системі, а ця процедура буде виконуватися в нульовому кільці привілеїв, з якого можна викликати будь-які переривання і дозволено звернення до портів вводу-виводу. Цей спосіб переходу в нульове кільце був використаний у вірусі WinCih і є недокументированним.

Мікропроцесори i486 і вище здатні обробляти до 256 різних типів переривань (або виключень). Перші 32 типу зарезервовані для внутрісистемних цілей, інші надані користувачеві.

Переривання 0 - помилка ділення на нуль. Виникає при виконанні команд поділу DIV, IDIV, якщо дільник дорівнює нулю.

Переривання 1 - виняток для налагодження. Це виключення виникає: в покроковому режимі (прапор TF = 1) після виконання кожної команди; при зверненні до сегменту TSS, що має біт зупинки T = 1; при зупинці в контрольних точках, які встановлюються за допомогою регістрів DR0-DR3.

Переривання 2 - немасковане переривання. Виникає при вступі активного сигналу на вхід NMI (наприклад в NoteBook'ах в разі розрядки батарей живлення для термінового збереження всіх не збережених даних), або при виконанні команди INT 2 (ніколи не використовується).

Переривання 3 - це виняток має місце при виконанні команди INT 3, яка використовується зазвичай при налагодженні програмного забезпечення. Оскільки команда INT 3 є однобайтове, її можна використовувати для заміни першого байта будь-якої команди в програмі, викликаючи при необхідності перехід до процедури налагодження. Після налагодження початковий код відновлюється.

Переривання 4 - виняток при переповненні. Цей виняток має місце, якщо мікропроцесор виконує команду INT0 при встановленому прапорі OF = 0. Таким чином, здійснюється контроль за появою переповнення після арифметичних операцій над числами із знаком.

Переривання 5 - перевищення межі масиву. Має місце при виконанні команди BOUND, якщо вміст адресується регістра виходить за вказані межі.

Переривання 6 - недозволений код команди. Має місце при спробі виконання команди з невірним кодом або неправильної адресацією операнда, а також при використанні префікса LOCK з командою, для якої блокування магістралі заборонено.

Переривання 7 - процесор FPU недоступний. Це виключення виникає при виконанні команд FPU, якщо біти EM або TS в регістрі управління CR0 мають значення «1». Установка EM = 1 забороняє використання внутрішнього процесора FPU. В цьому випадку для виконання надійшла команди процедура обслуговування повинна реалізувати програмну емуляцію FPU.

Переривання 8 - подвійна помилка. Цей виняток має місце, коли при виклику процедур обслуговування винятків, пов'язаних з помилками сегментів (виключення 10-13), мікропроцесор виявляє нове виняток, відмінне від виключення 14 (помилка сторінки). Подвійна помилка також має місце, коли при виклику процедури обробки виключення 14 виявляється нове виняток, відмінне за типом від повторної помилки сторінки. Якщо при обробці виключення 8 знову виникає якесь виключення, то мікропроцесор припиняє свою роботу.

Переривання 10 - недозволений сегмент TSS (таблиця стану завдання). Виняток цього типу виникає при перемиканні задач.

Переривання 11 - відсутність сегмента. Реалізується при зверненні до сегменту, в дескрипторі якого біт присутності P = 0 (сторінка знаходиться в своп файлі).

Переривання 12 - помилка звернення до стеку. Виникає при спробі завантаження регістра SS відсутнім дескриптором або при зверненні до стека з порушенням його межі.

Переривання 13 - порушення загального захисту. Це найбільш загальний тип винятку, причинами якого можуть бути:

1. порушення правил захисту при зверненні до пам'яті чи зовнішнім пристроям;
2. порушення кордонів сегмента при зверненні до таблиці дескрипторів або сегментів CS, DS, ES, FS або GS;
3. перемикання на зайняту задачу;
4. завантаження регістра CR0 вмістом, мають PE = 0 (незахищений режим) або PG = 1 (сторінкова адресація);
5. запис в сегмент команд або даних, доступний тільки для читання;
6. читання з сегменту, дозволеного тільки для використання як сегмента коду або стека;
7. передача управління сегменту, який є сегментом даних;
8. завантаження регістрів DS, ES, FS, GS або SS дескриптором системного сегмента;
9. перевищення максимальної допустимої довжини команди.

Переривання 14 - відсутність доступу до сторінки. Цей виняток має місце при дозволеної сторінкової адресації, якщо поточна процедура не має достатнього рівня привілеїв для виклику зазначеної сторінки або відбувається звернення до не існуючого розділу або сторінці.

Переривання 15 - зарезервовано.

Переривання 16 - помилка операції FPU. Має місце, якщо виявлена будь-яка з помилок FPU.

Переривання 17 - помилка вирівнювання. Виникає при формуванні не вирівняних адрес операндів в режимі примусового вирівнювання.

Ці переривання завжди обробляються операційною системою, а їх обробники знаходяться в модулі VMM.VxD. Переривання зовнішніх

пристроїв обробляються драйверами, що обслуговують ці пристрої. Це спеціальним чином, файли програми у форматі VxD (LX) і виконувані в кільці нульового рівня. Драйвери пристроїв, як правило, - найбільш критична частина програмного забезпечення комп'ютерів і безпосередньо з ними працювати вкрай небажано. Прикладним програмам надано API - інтерфейс для роботи практично з усіма ресурсами комп'ютера.

1.20.3. *Сигнали*

В операційній системі UNIX сигнали є способом передачі від одного процесу іншому або від ядра операційної системи якомусь процесу повідомлення про виникнення певної події. Сигнали можна розглядати як просту форму між процесії взаємодії. У той же час сигнали більше нагадують програмні переривання, - засіб, за допомогою якого нормальне виконання процесу може бути перервано. Наприклад, якщо процес виробляє ділення на 0, ядро посилає йому сигнал SIGFPE, а при натисканні клавіш переривання, звичайно «DEL», поточного процесу посилається сигнал SIGINT. Для відправлення сигналу служить команда kill (1) kill sig_no pid

Де: sig_no - номер або символічну назву сигналу, а pid ідентифікатор процесу, якому надсилається сигнал. Адміністратор системи може посилати сигнали будь-яким процесам, звичайний же користувач може посилати сигнали тільки процесам, власником яких він є.

1.21. ЗАГАЛЬНА СХЕМА ЗАВАНТАЖЕННЯ ОПЕРАЦІЙНОЇ СИСТЕМИ

1.21.1. MS DOS

При включенні комп'ютера процесор встановлює стан скидання, здійснює контроль парності, встановлює в регістрі CS значення 0FFFFh, а в регістрі IP - нуль. Таким чином, перша виконувана команда знаходиться за адресою FFFF: 0, що є точкою входу в BIOS. При цьому на апаратному рівні

виконується тестування пристроїв (POST-тест), заповнення векторів переривань. BIOS перевіряє порти комп'ютера для визначення та ініціалізації зовнішніх пристроїв. Потім BIOS створює в пам'яті (за адресою 0) таблицю переривань і виконує дві операції: INT 11h (запит списку приєднаного устаткування) і INT 12h (запит розміру фізичної пам'яті).

Далі BIOS визначає наявність OS DOS. Якщо виявлена системна дискета, то BIOS виконує переривання INT 19h для доступу до першого сектору диска, який містить блок початкового завантаження (Master boot record), Зчитується 0 доріжка 1 сектор циліндра 0, визначається активний розділ диска (для Hard Drive), зчитується перший сектор активного розділу, на початку якого знаходиться адреса програми початкового завантаження операційної системи, вона зчитується і їй передається керування.

Для функціонування операційної системи MS-DOS в оперативну пам'ять програмою початкового завантаження завантажується обов'язкова частина MS-DOS: MSDOS.SYS, IO.SYS, COMMAND.COM.

- • IO.SYS - забезпечує логічний інтерфейс низького рівня між системою і апаратурою (через BIOS); містить набір резидентних драйверів основних периферійних пристроїв. При ініціалізації програма визначає стан усіх пристроїв і устаткування, а також управляє операціями обміну між пам'яттю і зовнішніми пристроями.
- • В MSDOS.SYS розташовуються програми управління пам'яттю, файлами, даними, зовнішніми пристроями, завданнями, процесами. Це центральний компонент DOS, що реалізує основні функції ОС - керування ресурсами ПК і виконуваними програмами. Основу складають обробники перерви-ний верхнього рівня.
- • COMMAND.COM - командний процесор, що забезпечує інтерфейс користувача з обчислювальною системою

Файл IO.SYS, завантажений з диска, зазвичай складається з двох модулів: BIOS і SYSINIT. SYSINIT викликається за допомогою програми ініціалізації BIOS. Модуль визначає величину безперервної пам'яті, доступний системі, і

потім розпорядженні SYSINIT в старші адреси. Далі модуль переносить ядро системи DOS, MSDOS.SYS, з області її початкового завантаження в область остаточного розташування.

Далі SYSINIT викликає програму ініціалізації в модулі MSDOS.SYS. Ядро DOS ініціалізує її внутрішні таблиці і робочі області, встановлює вектора переривань за адресами з 20h по 2Fh і перебирає пов'язаний список резидентних драйверів пристроїв, викликаючи функцію ініціалізації для кожного з них.

Коли ядро DOS проініціалізувати і всі резидентні драйвери стають доступними, модуль SYSINIT може викликати звичайний файловий сервіс системи MS-DOS і відкрити файл CONFIG.SYS. Драйвери, відповідно до директив-вами DEVICE =, зазначеними в CONFIG.SYS, послідовно завантажуються в пам'ять, активізуються за допомогою викликів відповідних модулів ініціалізації і заносяться в зв'язаний список драйверів. Функції ініціалізації кожного з них повідомляють SYSINIT розмір пам'яті, відведений під відповідний драйвер.

Після завантаження всіх встановлених драйверів, SYSINIT закриває все дескриптор файлів і відкриває консоль, принтер і послідовний порт. Це дозволяє заміщати резидентні драйвери BIOS стандартних пристроїв. Модуль SYSINIT переміщує себе до старших адресами пам'яті і зрушує ядро системи DOS в сторону молодших адрес пам'яті в область остаточного розташування.

В кінці свого виконання SYSINIT викликає системну функцію EXEC для завантаження інтерпретатора командного рядка, тобто COMMAND.COM.

COMMAND.COM - командний процесор, який забезпечує інтерфейс користувача з установкою.

Система завантажує дві частини програми Command.com в пам'ять:

1. резидентна частина - обробляє всі помилки дискових операцій вводу / виводу і управляє перериваннями:

• • INT 22h - адреса обробки завершення завдання;

• INT 23h - адреса реакції на Ctr / Break;

• INT 24h - адреса реакції на фатальну помилку.

транзитна частина - завантажується в найстарші адреси пам'яті. Вона містить налаштовуючий завантажувач і призначена для завантаження. COM або. EXE файлів з диска в пам'ять для виконання. В результаті отримуємо остаточний розподіл пам'яті. Залежно від модифікації персонального комп'ютера і складу його периферійного обладнання, розподіл адресного простору може дещо відрізнятися. Тим не менш, розміщення основних компонентів системи досить строго уніфіковано. Типова схема використання адресного простору приведена в табл. 1.9.

Перші 640 Кбайт адресного простору з адресами 00000h до 9FFFFh відводиться під основну оперативну пам'ять, яка ще називають стандартної (Conventional). Початковий кілобайт оперативної пам'яті зайнятий векторами переривань (256 векторів по 4 байти). Слідом за векторами переривань Наявна область даних BIOS, яка займає адреси від 00400h до 004FFh. У цій області зберігаються різноманітні дані, використовувані програмами BIOS в процесі управління периферійним обладнанням.

Так, тут розміщуються:

• вхідний буфер клавіатури з системою вказівників;

• адреси послідовних і паралельних портів;

• дані, що характеризують настройку відеосистеми (форма курсора, його поточне місце розташування на екрані, поточний відеорежим і пр.);

• комірки для відліку поточного часу; область межзадачного зв'язків і т.д.

Табл. 1.9. Остаточний розподіл адресного простору

1 Кбайт	Вектори переривань	00400h
256 байт	Область даних BIOS	00500h
512 байт	Область даних DOS	00700h
	IO.SYS и MSDOS.SYS	
	Завантажувані драйвери	

	Command.com (резидентна частина)	
	Вільна пам'ять для за-розвантажувальних прикладних і системних програм	A0000h
64 Кбайт	Графічний буфер	B0000h
32 Кбайт	UMB	B8000h
32 Кбайт	Текстовий буфер	C0000h
64 Кбайт	ПЗП — розширення BIOS	D0000h
64 Кбайт	UMB	E0000h
128 Кбайт	ПЗП — розширення BIOS	1000000h
64 Кбайт	HMA	10FFF0h
До 15Мбайт	XMS	

Область даних BIOS заповнюється інформацією в процесі початкового завантаження комп'ютера і динамічно модифікується системою в міру необхідності.

В області пам'яті, починаючи з 500h, містяться деякі системні дані MS-DOS. Слідом за областю даних MS-DOS розташовується власне операційна система, завантажувана з файлів IO.SYS і MSDOS.SYS (IBMBIO.COM і IBMDOS.COM для системи PC-DOS).

Вся решта пам'ять до кордону 640 Кбайт називається транзитної областю. Вона вільна для завантаження будь-яких системних або прикладних програм. Решта 384 Кбайта вище 640 Кбайт адресного простору називаються старшої пам'яттю (Upper Memory). Спочатку вони були призначені для розміщення постійних запам'ятовуючих пристроїв (ПЗП). Практично під ПЗП зайнята тільки частина адрес. Тому частина старшої пам'яті виявляється вільною. Ці ділянки використовуються для адресації до додатковим блокам оперативної

пам'яті, які носять назву блоків старшої пам'яті (Upper Memory Blocks, UMB). Для підтримки старшої пам'яті використовується драйвер HIMEM.SYS, який дозволяє ефективно використовувати UMB, завантажуючи в них встановлюються драйвери пристроїв, а також резидентні програми. Завантаження системних програм в UMB звільняє від них стандартну пам'ять, збільшуючи її транзитну область. Поряд із стандартною пам'яттю (640 Кбайт) використовується розширена пам'ять об'ємом до 15 Мбайт.

У транзитну частину вантажиться програма діалогу з користувачем, після чого система готова до сприйняття команд користувача.

Завдання фіксуються в кільцевому буфері, по <CR> інтерпретатор починає виконання надійшов завдання (виконання передається MS-DOS.SYS).

Якщо знайдений файл з розширенням. COM або. EXE, то COMMAND.COM, використовуючи системну функцію EXEC, здійснює завантаження і виконання знайденого файлу. Якщо в області транзитних програм достатньо вільної пам'яті, то EXEC виділяє блок пам'яті під нову програму, будує PSP за його базовому адресою, а потім зчитує програму в пам'ять безпосередньо за PSP. В кінці своєї роботи EXEC встановлює регістри сегмента і стека і передає управління програмі.

При ініціалізації ядра та системи активізуються і виконуються наступні процеси операційної системи: адміністратор пам'яті, програма роботи з ВУ, програма підтримки файлової системи і процес, підлеглий таймеру. Після запуску системи управління передається внутрішнім процедурам COMMAND.COM, серед яких можна виділити програму системного введення, аналізує вхідний потік завдань. Програма працює в режимі сторожа; вхідний потік завдань накопичується у вхідних чергах (перший рівень класичного планування процесів). Як тільки система визначає, що у неї є ресурси (найбільш важлива наявність ОП), операційна система завантажує планувальник як транзитну програму, яка аналізує наявність інших ресурсів, ініціалізувавши чергу процесів до процесора.

Після завершення виконання транзитної програми викликається спеціальна функція завершення, яка в свою чергу звільняє пам'ять від транзитної програми і повертає керування програмі, що викликала завантаження транзитної програми (Command.com).

При завантаженні COM - програми DOS встановлює в чотирьох сегментних регістрах адресу першого байта PSP. Потім встановлює покажчик стека на кінець сегмента обсягом 64Кбайт (FFFFh). У вершину стека заноситься нульове слово. В командний покажчик поміщається 100h (розмір PSP). Після цього управління передається за адресою реєстрової пари CS: IP. Ця адреса є початком виконуваної COM-програми. При виході з програми команда RET заносить в регістр IP нульове слово. У цьому випадки в CS: IP виходить адресу першого байта PSP, де знаходиться команда INT 20h. При цьому управління передається в резидентну частину Command.com.

Програма у форматі EXE складається з двох частин:

1. заголовка - записи, що містить інформацію з управління та налаштування програми;

2. завантажувального модуля.

Після ініціалізації регістри CS і SS містять правильні адреси сегментів, а регістр DS (і ES) повинен бути налаштований у програмі на власний сегмент даних. При завершенні програми команда RET заносить в регістр IP нульове значення. У реєстрової парі CS: IP виходить адресу першого байта PSP. Після виконання команди управління передається в DOS.

MS-DOS відводить префіксу область в 256 байт на початку блоку пам'яті, що виділяється транзитної програмі (Табл. 1.10.) Префікс має декілька зв'язків з MS-DOS, які можуть використовуватися транзитної програмою, крім того, певну інформацію записує в нього MS-DOS як для власних це-лей, так і для передачі транзитної програмі, яка в разі необхідності може цю інформацію використовувати.

Табл. 1.10. Структура PSP

Системне переривання int 20h

Сегмент, кінець виділеного блоку
зарезервовано
Довгий виклик диспетчера функцій MS-DOS
Попередні зміст вектора переривання обробки завершення (int 22h)
Попереднє зміст вектора переривання Ctr-C (int 23h)
Попереднє зміст вектора переривання обробки критичної помилки (int 24h)
зарезервовано
Сегментний адресу блоку оточення
зарезервовано
Стандартний блок управління файлами № 1
Стандартний блок управління файлами № 2
(перекривається якщо FCB 1 відкритий)

1.21.2. WINDOS

Апаратна ініціалізація комп’ютера

Апаратна ініціалізація комп'ютера

Загальна схема завантаження операційних систем схожа на завантаження MS-DOS. Проте зміни та особливості структур і способів організації обробки інформації в сучасних ОС вплинули на виконання деяких процедур завантаження і ініціалізації операційної системи.

Для початку процедури завантаження на процесор подається команда RESET (скидання), деякі регістри процесора встановлюються в фіксовані значення і починає виконуватися код з фізичного адресою 0xffffff0. Апаратне забезпечення відображає цю адресу на Basic Input / Output System (BIOS), який включає набір керованих перериваннями низькорівневих процедур, які можуть бути використані для керування пристроями, підключеними до комп'ютера.

Більшість сучасних ОС використовують BIOS тільки на етапі початкового завантаження (такий етап ще називається bootstrapping). Процедура початкового завантаження BIOS (boostrap procedure) зводиться до виконання чотирьох операцій.

1. Виконання набору тестів (Power-On Self-Test, POST) визначають наявність пристроїв в системі і їх працездатність.

2. Ініціалізація апаратних пристроїв. Цей етап перевіряє без конфліктність роботи при використанні переривань або портів вводу-виводу.

3. Пошук і виконання програми початкового завантаження. В залежності від установок BIOS, проводиться спроба доступу до першого сектору гнучкого диска, жорсткого диска до головної завантажувальної записом (MBR) або компакт-диска.

4. Після того, як пристрій знайдено, BIOS копіює вміст першого сектора в оперативну пам'ять з фіксованого адреси 0x00007c00 і виконує команду переходу на цю адресу для виконання завантаженого коду. У пам'ять завантажується бутовий завантажувач (boot loader) і йому передається управління

Завантаження ОС

Бутовим завантажувачем, записаним в MBR, називається програма, яка викликається кодом BIOS в ході виконання процедури початкового завантаження для завантаження образу ядра операційної системи в оперативну пам'ять.

MBR містить таблицю розділів і програму, яка завантажує перший сектор, одного з обраних розділів диска в пам'ять. Починає виконуватися програма, яка в ньому знаходиться - зазвичай ця програма і називають програми початкового завантаження ОС. Зазвичай вибір розділу, з якого потрібно завантажити програму початкового завантаження ОС проводиться на підставі меню із зазначенням операційної системи згенерованої в кожному розділі. Таким чином, можна завантажити тільки ту ОС, ядро якої знаходиться на

активному розділі, інші підходи ми розглянемо пізніше. Така схема завантаження отримала назву схемою двоетапної завантаження.

Початковий завантажувач ОС зазвичай записується в завантажувальний сектор при її інсталяції, тоді ж виконується відповідний запис у MBR. Найпростіший завантажувача виконує пошук в активному розділі диска програм ініціалізації ядра і ініціалізації системи (зазвичай ці файли, знаходяться у фіксованому місці кореневого каталога). Їх запис туди виконується під час інсталяції операційної системи.

В даному випадку завантажувач розбивається на дві частини: завантажувач першого етапу і завантажувач другого етапу. Завантажувач першого етапу, знаходиться в завантажувальному секторі диска і його основним його завданням є пошук на диску та завантаження в пам'ять не ядра, а завантажувача другого етапу. При такій схемі завантаження можна керувати завантаженням декількох операційних систем. Особливо зручно це робити в завантажувача, що приймають управління від MBR, в цьому випадку завантажувач бере на себе пошук активного розділу і завантаження системи з нього, конфігурацію такого завантажувача, наприклад, можна динамічно змінювати при зміні розділів диска. Завантажувач може містити код доступу до різних файловим системам, код завантаження різних ядер і т. д. Крім цього можна мати користувальницький інтерфейс. Такий інтерфейс може зводитися до відображення меню вибору завантажується операційної системи, крім того, можна організувати передачу параметрів, введених користувачем, в ядро перед його завантаженням. Можна надавати користувачеві можливість задавати конфігурацію завантажувача зсередини завантаженої їм операційної системи. Для цього, наприклад, можна задати текстовий конфігураційний файл, зберегти його на диску і запустити спеціальну утиліту, яка зробить синтаксичний розбір цього файлу, перетворює його у внутрішнє представлення і збереже це подання на диску у фіксованому місці, яке відоме завантажувачу. Такий завантажувач не обмежений одним розділом і одним диском, він може працювати з усіма

дисками комп'ютера, завантажувати ядра, що знаходяться в різних місцях на диску (у тому числі всередині ієрархії каталогів).

Двоетапні завантажувачі поширені надзвичайно широко, вони можуть поставлятися разом з ОС (наприклад, lilo або GRUB для Linux, завантажувач Windows 2000), а можуть бути реалізовані як окремі продукти - менеджери завантаження (boot managers).

Код ядра зазвичай зберігається в окремому файлі на диску, завантажувач повинен вміти знайти цей файл. Є різні підходи до реалізації завантаження ядра:

- ядро може завантажитися в пам'ять відразу повністю;
- ядро може завантажуватися по частинах, при цьому окремі частини можуть завантажувати жати інші частини у міру потреби;
- ядро може завантажитися в пам'ять в частково стислому стані, після виконання деяких попередніх операцій воно може бути розпаковано.

Після завантаження з фіксованих файлів частини ядра в пам'ять управління передається спеціальним процедурам, які входять до складу завантажуваних файлів. Після їх виконання вони можуть бути видалені з пам'яті. Це програми ініціалізації ядра і ініціалізації системи. Спочатку виконується ініціалізація ядра. В ході ініціалізації проводиться опитування та ініціалізація обладнання, ініціалізація підсистем ядра, завантаження і ініціалізація необхідних драйверів, монтування кореневої файлової системи, формування керуючих таблиць операційної системи.

Після того, як ядро завантажилося в пам'ять і виконався етап ініціалізації ядра, починається завантаження різних компонентів системи та ініціалізація системи. У першу чергу, до таких компонентів відносяться додаткові драйвери (які не були потрібні при завантаженні), системні фонові процеси. Велика частина цієї роботи здійснюється в режимі користувача. В результаті до моменту, коли користувач може почати користуватися системою, в системі активізовані процеси, які потрібні для повноцінної роботи обчислювальної системи.

Розглянемо більш докладно етапи завантаження WINDOWS. Спочатку блок початкового завантаження Windows шукає файл WINBOOT.SYS. З того моменту як завантажився WINBOOT.SYS починається процес завантаження Windows. Використовуючи відладчик, можна простежити по кроках процес завантаження. Встановивши контрольні точки на функції DOS: File Open, Extended Open / Create, Find First File і Exec. Розглянемо першу стадію завантаження Windows:

Func	File	Comment
OPEN	\Logo.sys	завантаження логотипу
RENAME	IO.SYS→IO.DOS	
RENAME	MSDOS.SYS→MSDOS.DOS	
FIND	\DRVSPACE.BIN	відмова - DRVSPACE відсутня
OPEN	C:\DBLSPACE.BIN	відмова - DBLSPACE відсутня
EXEC	WINBOOT.SYS	завантажуємо WINBOOT.SYS
FIND	C:\SYSTEM.DAT	знаходить реєстру
OPEN	C:\SYSTEM.DAT	читає реєстру
OPEN	CONFIG$	Частина Plug & Play Configuration Manager для реального режиму
OPEN	C:\WINDOWS\HIMEM.SYS	загрузка верхньої пам'яті
EXEC	C:\WINDOWS\HIMEM.SYS	загрузка верхньої пам'яті
OPEN	IFSHLP	
OPEN	C:\WINDOWS\IFSHLP.SYS	
EXEC	C:\WINDOWS\IFSHLP.SYS	
OPEN	SETVERXX	
OPEN	C:\WINDOWS\COMMAND\SETVER.EXE	
EXEC	C:\WINDOWS\COMMAND\SETVER.E	

	XE	
OPEN	CON	
OPEN	AUX	
OPEN	PRN	
OPEN	\AUTOEXEC.BAT	відмова - AUTOEXEC.BAT відсутня
FIND	WIN.???	
FIND	C:\WINDOWS\WIN.???	
OPEN	C:\WINDOWS\WIN.COM	
EXEC	C:\WINDOWS\WIN.COM	
OPEN	C:\WINDOWS\VMM32.VXD	
OPEN	C:\WINDOWS\SYSTEM\VMM32.VXD	
EXEC	C:\WINDOWS\SYSTEM\VMM32.VXD	завантажити VMM і драйверів VxD

З таблиці видно, що після завантаження WINBOOT.SYS, шукається конфігураційна інформація, визначена користувачем. У Windows кращий-ним місцем для інформації конфігурації є реєстр. Файл C: \ SYSTEM.DAT є головним файлом реєстру.

Після відкриття реєстру Windows відкриває пристрій CONFIG $ в реальному режимі (частина Plug-and-Play Configuration Manager - менеджер конфігурації «підключи і грай») є частиною WINBOOT.SYS. Компонент захищеного режиму живе в VxD під назвою CONFIGMG. VxD CONFIGMG зв'язується з пристроєм CONFIG $ реального режиму за допомогою функції DOS IOCTL і передає CONFIG $ V86 обробник непрямого виклику так, що CONFIG $ може визвати CONFIGMG. Крім того, що Plug-and-Play Configuration Manager теж має компонент реального режиму, варто зазначити, що CONFIG $ надає новий інтерфейс, дозволяючи драйверам

пристроїв і TSR - програмами DOS запитувати інформацію про конфігурацію.

Далі WINBOOT.SYS завантажує драйвер додаткової (extended) пам'яті (XMS). Після завантаження Windows V86MMGR (V86 Memory Manager) VXD забезпечує свої власні функції XMS. Сервер XMS V86MMGR заміняє їм драйвер, присутній до запуску Windows.

Після HIMEM.SYS WINBOOT.SYS завантажує IFSHLP.SYS. Код реального режиму в IFSHLP є ключовою частиною файлової системи, інстальоване Windows. IFSMGR не зможе завантажитися, якщо не виявить драйвер IFS $ HLP $. IFSMGR викликає IFSHLP, використовуючи функцію DOS IOCTL передає IFSHLP покажчик на V86-обробник непрямого виклику, який в подальшому використовуються IFSHLP для зв'язку з IFSMGR.

Потім WINBOOT.SYS завантажує SETVER.EXE (підставляють для програм номер версії DOS - саме DOS, як не дивно) і, за певних обставин, - менеджер розширеної пам'яті Microsoft, EMM386.EXE. Можна зауважити, що V86MMGR VxD забезпечує функції не тільки XMS, а й EMM.

Потім WINBOOT.SYS шукає файл AUTOEXEC.BAT і завантажує командну оболонку DOS (COMMAND.COM) для обробки AUTOEXEC.BAT. Але якщо AUTOEXEC.BAT немає, то немає необхідності завантажувати COMMAND.COM, і WINBOOT.SYS переходить до безпосередньої завантаженні WIN.COM. WIN.COM Зазвичай знаходиться в C: \ WINOWS, але це розташування закріплено з WINBOOT.SYS не жорстко. Це випливає з реєстру (HKEY_LOCAL_MACHINE \ Software \ Microsoft \ Windows \ CurrentVersion \ SystemRoot). WIN.COM - це ще не Windows. Це маленька (менше ніж 20 Кбайт) програма для завантаження Windows. WIN.COM загружає файл VMM32.VXD і грає важливу роль в ОС Windows. Саме, вона оберігає нас від повернення в DOS. Коли завершується робота Windows, саме WIN.COM виводить на екран повідомлення «You can now safely turn off the computer. If you want to restart your computer, press Ctrl-Alt-Del »(Можете

тепер спокійно вимикати комп'ютер. Якщо хочете перезапустити ваш комп'ютер, натисніть <Ctrl-Alt-Del>).

Отже перша стадія завантаження Windows завершується тим, що WIN.COM ви-виконуваних VMM32.VXD. VMM32.VXD містить саму суть операційної системи Windows (саме операційну систему, а не операційне середовище).

Далі розглянемо наступну стадію завантаження Windows:

Func	File	Comment
EXEC	C:\WINDOWS\SYSTEM\VMM32.VXD	
OPEN	QEMM386$	Quarterdeck QEMM/386
OPEN	386MAX$	Qualitas 386Max
OPEN	SMARTAAR	SmartDrive
OPEN	C:\SYSTEM.DAT	
OPEN	$DebugDD	WDEB386.EXE (ядро відладчика Windows)
OPEN	NDISHLP$	NDISHLP.SYS (для NDIS 2.0)
OPEN	C:\WINDOWS\SYSTEM.INI	
CREAT	C:\WINDOWS\WNBOOTNG.STS	
FIND	C:\WINDOWS\SYSTEM\VMM32*.VDX	
OPEN	C:\WINDOWS\SYSTEM\nwlink.386	Windows NetWare – сумісний протокол
OPEN	C:\WINDOWS\SYSTEM\wsipx.386	Windows Sockets для Novell IPX
OPEN	C:\WINDOWS\SYSTEM\vnetsup.386	Win32 віртуальне

		забезпечення мережі
OPEN	IFSHLP	IFSHLP.SYS
OPEN	C:\WINDOWS\SYSTEM\ndis.386	NDIS 3.0
OPEN	IFSHLP	IFSHLP.SYS
OPEN	C:\WINDOWS\SYSTEM\ndis2sup.386	Перетворення між NDIS 2.0 and 3.0
OPEN	NDISHLP$	
OPEN	C:\WINDOWS\SYSTEM\msodisup.386	Перетворювач забезпечення ODI в ODI MLID
OPEN	C:\WINDOWS\SYSTEM\vnetbios.386	Віртуальна NetBIOS NetBIOS (INT 5Ch)
OPEN	C:\WINDOWS\SYSTEM\wsock.386	Windows Sockets
OPEN	CONFIG$	CONFIG$ в WINBOOT.SYS
OPEN	C:\WINDOWS\SYSTEM\vserver.386	Сервер для однорангового доступу
OPEN	IFSHLP	
OPEN	C:\WINDOWS\SYSTEM\vredir.386	Мережевий редиректор
OPEN	C:\WINDOWS\SYSTEM\dva.386	
OPEN	C:\WINDOWS\SYSTEM\vpmtd.386	
OPEN	IFSHLP	
OPEN	SMARTAAR	

OPEN	C:\WINDOWS\SYSTEM\ISAPNP.vxd	
OPEN	C:\WINDOWS\SYSTEM\MMDEVLDR.vxd	
OPEN	C:\WINDOWS\SYSTEM\MSSB16.vxd	
OPEN	C:\WINDOWS\SYSTEM\VJYD.vxd	
OPEN	SCSIMGR$	
OPEN	C:\WINDOWS\IOS.INI	
OPEN	C:\WINDOWS\IOS.LOG	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS*.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\apix.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\cdfs.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\cdtsd.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\cdvsd.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\disktsd.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\scsilhl.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\voltrack.vxd	
OPEN	C:\WINDOWS\SYSTEM\IOSUBSYS\rmm.pdr	
OPEN	C:\WINDOWS\INDISLOG.TXT	
OPEN	C:\WINDOWS\SYSTEM\ee16.386	Intel EtherExpress 16 (NDIS 3.0)
OPEN	C:\WINDOWS\SYSTEM\netbeui.386	Розширений інтерфейс користувача NetBIOS
OPEN	C:\WINDOWS\WINSTART.BAT	
OPEN	C:\WINDOWS\SYSTEM\LPTENUM.vxd	
OPEN	C:\WINDOWS\SYSTEM\UNICODE.BIN	
OPEN	IFSHLP	
EXEC	C:\WINDOWS\SYSTEM\krnl386.exe	

Отже почнемо спочатку WIN.COM завантажує VMM32.VXD. Незважаючи на розширення імені файлу, VMM32.VXD є не єдиним Virtual Device Driver (драйвер віртуального пристрою - VxD), а цілим зборами VxD. Програма установки Windows формує VMM32.VXD з вільних VxD, тому файл може змінюватися від однієї машини до іншої. Але зазвичай VMM32.VXD включає близько 45 VxD.

Диспетчер віртуальної машини (Virtual Machine Manager) - це, по суті справи, серце операційної системи. Він включає код, який реалізує всі дії базової системи з управління завданнями, діями над віртуальною пам'яттю, завантаженням і завершенням програм, а також підтримкою взаємодії програм. Крім різних сервісів, VMM також містить первинні системні обробники переривань, помилок і виняткових ситуацій. Наприклад, всі апаратні переривання від таймера, клавіатури, миші, COM-портів, і т.д. спочатку надходять на VMM, який, зазвичай, передає їх на Virtual Programmable Interrupt Controller (PIC) Device (VPICD) VxD. Аналогічним чином, загальні помилки захисту пам'яті (GP) і помилки віртуальних сторінок пам'яті спочатку надходять на VMM, а той чи обробляє їх самостійно, або видає зацікавленим VxD.

У таблиці 2 ми бачимо, як Windows шукає різні драйвери пристроїв DOS в реальному режимі. Також видно, що Windows завантажує велику кількість додаткових VxD з розширеннями 386 або VXD, які не розташовуються всередині VMM32.VXD.

Всі ці описи міститися в SYSTEM.INI, і описуються через директиви device = для завантаження 32-бітового програмного забезпечення.

Крім відображення в процесі завантаження Windows тих VxD, що в явній формі вказуються в директивах device = в SYSTEM.INI, у таблиці також показано динамічне завантаження супервізором вводу-виводу (IOS) Windows VxD з підкаталогу IOSUBSYS. Сюди включені VxD RMM, трасувальник носіїв (voltrack) і DiskTSD. RMM - це Real Mode Mapper, який перетворює дисковий драйвер реального режиму в драйвер Fastdisk захищеного режиму.

DiskTSD - це драйвер специфічного типу, відповідальний за перетворення логічного запиту до запиту до фізичного диску.

В самому кінці таблиці Windows запускає KRNL386.EXE. Це ядро операційної системи. Після завантаження KRNL386.EXE управління переходить в нього. KRNL386.EXE в свою чергу завантажує драйвер VWIN32. VWIN32 завантажує три бібліотеки динамічної компоновки, і передає управління 16-ти бітовому модулю Kernel, а той викликає функцію ініціалізації KERNEL32.DLL. Як тільки ця функція відпрацює, підсистема Win32 готова до експлуатації.

Після завантаження KERNEL32.DLL Windows починає завантажувати драйвера пристроїв Win16. Для завантаження цих драйверів Win16 KRNL386.exe використовує динамічну компоновку часу виконання. У наступній таблиці наведено список найбільш поширених драйверів Win16:

	C:\WINDOWS\SYSTEM\system.drv
	C:\WINDOWS\SYSTEM\keyboard.drv
	C:\WINDOWS\SYSTEM\mouse.drv
	C:\WINDOWS\MOUSE.INI
	C:\WINDOWS\system.INI
	C:\WINDOWS\SYSTEM\framebuf.drv
	C:\WINDOWS\SYSTEM\DIBENG.DLL
	C:\WINDOWS\SYSTEM\sound.drv
	C:\WINDOWS\SYSTEM\comm.drv

Також Windows завантажує величезна кількість DLL.

В кінці завантаження Windows система завантажує саму помітну частину операційної системи - оболонку. У Windows оболонка називається Explorer або Cabinet. Оболонкою Windows є Win32-додаток, CAB32.EXE. CAB32.EXE, в свою чергу використовує багато функцій з завантажується частини операційної системи, такої як SHELL32.DLL.

1.21.3 Особливості завантажувача Linux

При завантаженні Linux використовується двоетапний завантажувач. Насправді, існує кілька програмних продуктів, реалізують такі завантажувачі, найбільш відомим з них є lilo (від linux loader). Даний завантажувач може бути встановлений як в MBR (замінивши там код, який завантажує перший сектор активного розділу), так і в завантажувальному секторі активного розділу диска. Перша частина lilo, записана в завантажувальний сектор або в MBR, при своєму виконанні готує пам'ять і завантажує в неї другу частину. Друга частина зчитує з диска двійкове уявлення карти існуючих на комп'ютері варіантів завантаження (різні ОС, різні установки Linux і т. д.) і пропонує користувачеві вибрати один з них (за допомогою підказки «LILO boot:»). Зазначимо, що початкова версія карти варіантів завантаження створюється користувачем (системним адміністратором) у вигляді звичайного текстового файлу / etc / lilo.conf, після кожного її зміни необхідно оновлювати уявлення карти на диску, що використовується завантажувачем, для цього можна просто виконати команду (з правами root):

```
# lilo
```

Після вибору користувачем одного з варіантів завантаження поведінку завантажувача залежить від характеру файлової системи для розділу. При виборі розділу з іншого ОС (наприклад, Windows) зчитується в пам'ять і виконується завантажувальний сектор цього розділу (тим самим за допомогою lilo можна завантажити будь-яку ОС), при виборі ж розділу з Linux в пам'ять завантажується ядро системи (його адресу на диску знаходиться в карті варіантів завантаження). Після завантаження в пам'яті виявляється стислий ядро системи і придатний для виконання код двох функцій завантаження: setup () і startup_32 (). Код завантажувача переходить до виконання функції setup (), починаючи ініціалізацію ядра.

На першому етапі, ініціалізація проходить при роботі процесора в реальному режимі. Спочатку виконується ініціалізація апаратних пристроїв. Функція setup () визначає фізичний обсяг пам'яті в системі, ініціалізує

клавіатуру, вигляді карту, контролер жорсткого диска, перевизначає таблицю переривань, потім переводить процесор в захищений режим і передає управління функції startup_32 ().

Функція startup_32 () задає сегментні регістри і стек, розпаковує образ ядра і розташовує його в пам'яті. Після цього виконується код ядра. Виконання починається з функції, яка теж називається startup_32 (). Ця функція створює середовище виконання для першого потоку ядра Linux в рамках процесу 0, створює його стек, ініціалізує підтримку сторінкової організації пам'яті, задає початкові (порожні) обробники переривань, визначає модель процесора і переходить до виконання функції start_kernel ().

Функція start_kernel () також виконується в в рамках процесу 0 в потоці ядра, завершуючи ініціалізацію ядра. Вона призводить до робочого стану практично кожний компонент ядра, зокрема: ініціалізувалися таблиці сторінок і всі дескриптори сторінок; остаточно ініціалізується таблиця переривань; ініціалізується розподільник пам'яті; встановлюються системні дата і час; виконується код ініціалізації драйверів пристроїв; файлову систему. Крім того, за допомогою функції kernel_thread () створюється потік ініціалізації («процес 1»), що виконує код функції init (). Даний потік створює інші потоки ядра і виконує програму / sbin / init, перетворюючись у перший в системі користувальницький процес init. При цьому системі вже доступна файлова система з найважливішими поділюваними бібліотеками (каталог / lib має бути на тому ж розділі, що й кореневий каталог /). На цьому ініціалізація ядра завершується, функція start_kernel () переходить в нескінченний цикл простою (idle loop). Подальша ініціалізація системи про-виходить при виконанні init.

Процес init є предком всіх інших процесів у системі. При запуску даний процес читає свій конфігураційний файл / etc / inittab і за-пускає процеси, визначені в ньому. Набір запускаються процесів залежить від постачання Linux.

Файл / etc / inittab при завантаженні системи Red Hat Linux визначає кілька рівнів роботи (runlevels), кожен з яких визначає особливу програмне конфігурацію системи, при якій може існувати тільки певна група процесів. Рівень роботи задає режим функціонування ОС (однокористувацький, багатокористувацький, перезавантаження і т. д). Стандартними рівнями роботи є рівні з 0 по 6:

0 - завершення роботи (shutdown);

1 - однокористувацький режим (single user mode) - в цьому режимі не дозволяється виконання низки фонових процесів, не можна входити через мережу і т. д.;

3 - стандартний багатокористувацький режим (зазвичай цей рівень задається за замовчуванням);

5 - багатокористувацький режим із завантаженням графічної системи X Window System;

6 - перезавантаження (reboot).

Для деяких рівнів задаються символьні синоніми, так, наприклад, для рівня 1 синонімом є рівень S. У файлі / etc / inittab визначені різні командні файли - скрипти, які повинні виконуватися для різних рівнів виконання.

Синтаксис рядка / etc / inittab - ідентифікатор: рівень - роботи: дія: програма.

Перший скрипт, який запускає init, визначено в рядку / etc / inittab з дією, заданим ключовим словом sysinit. Точне його ім'я залежить від поставки Linux, в Red Hat це / etc / rc.d / rc.sysinit. Його ще називають стартовим скриптом. Ось відповідний рядок / etc / inittab:

si::**sysinit**:/etc/rc.d/rc.sysinit

Стартовий скрипт налаштовує основні системні сервіси, в тому числі:

1. час від часу перевіряє диски на помилки командою fsck.

2. завантажує модулі ядра для драйверів пристроїв, які не повинні бути завантажені раніше;

3. ініціалізує область підкачки командою swapon;

4. монтує файлові системи для дисків, зазначених у файлі / etc / fstab.

Крім стартового скрипта, в каталозі / etc / rc.d знаходяться декілька інших скриптів:

- */etc/rc.d/rc* – викликається при зміні рівня виконання, де в якості параметра приймається номер рівня і запускаються всі сценарії, відповідні цьому рівню;
- */etc/rc.d/rc.local* – при завантаженні системи викликається останнім. Скрипт містить специфічні для даної машини команди, які не рекомендується поміщати в стартовий скрипт, оскільки при перевстановлення або оновленні системи цей файл затирається, а rc.local - ні.

Запуск системних фонових процесів і ініціалізація системних служб, таких, як мережеві інтерфейси, виконується на основі набору файлів у каталозі */etc/rc.d/init.d* і набора підкаталогів */etc/rc.d*, специфічних для рівня виконання (от *rc0.d* до *rc6.d*).

В каталозі */etc/rc.d/init.d* міститься набір індивідуальних скриптів, що відповідають за запуск і зупинку різних фонових процесів і служб. Наприклад, файл / etc / rc.d / init.d / httpd відповідає за управління web-сервером Apache. Кожен такий скрипт повинен приймати в якості параметрів принаймні ключові слова start і stop для запуску і зупинки. Такі скрипти можна запускати і безпосередньо:

```
# /etc/rc.d/init.d/httpd start
```

Зазвичай такі скрипти створюються при установці програмного забезпечення.

Скрипти в каталозі / etc / rc.d / init.d не запускаються init безпосередньо при за-вивантаження системи. Для організації такого запуску існує в / etc / rc.d набір каталогів від rc0.d до rc6.d, кожен з яких містить символічні зв'язку, що вказують на файли з / etc / rc.d / init.d. При переході на відповідний рівень виконання init запускає скрипт / etc / rc.d / rc, який переглядає всі зв'язки відповідного рівня каталогу й виконує дії у відповідності з їх іменами.

Кожна зв'язок має певне ім'я в форматі Knnімя або Snnімя, де nn-цифри, наприклад S70httpd.

1. Якщо зв'язок починається на S, це означає, що на даному рівні дана служба повинна бути запущена (і якщо її немає, то для неї потрібно запустити скрипт з / etc / rc.d / init.d з параметром start). Якщо зв'язок починається на K то на даному рівні дана служби не повинно бути запущена (якщо вона запущена, її потрібно зупинити, запустивши відповідний скрипт з параметром stop).

2. Число nn задає номер послідовності, що визначає порядок запуску або зупинки служб при переході на рівень (чим nn більше, тим пізніше виконається скрипт, важливо, щоб до цього часу вже були запущені всі служби, від яких залежить дана. Наприклад, ініціалізацію мережі потрібно виконувати якомога раніше, тому зв'язок, яка вказує на скрипт, в цьому випадку може бути такою: S20network).

3. Ім'я зв'язку завершується ім'ям скрипта в / etc / rc.d / init.d.

Коли init переходить на рівень виконання 3, всі зв'язки, що починаються на S в каталозі / etc/rc.d/rc3.d, будуть запущені послідовно в порядку їх номерів, при кожному запуску буде вказано параметр start. Після виконання всіх скриптів вимоги до рівня виконання 3 будуть задоволені.

В / etc / inittab повинен бути заданий рівень виконання за замовчуванням, для чого потрібно включити в файл рядок з ключовим словом initdefault. Система завершить свою завантаження після переходу на цей рівень. Зазвичай таким рівнем є рівень 3.

id:3:**initdefault**:

Після запуску всіх скриптів для переходу на рівень за замовчуванням, init завжди запускає спеціальну програму getty, яка відповідає за зв'язок з користувачем через консоль і термінали. Існують різні реалізації цієї програми, в Linux популярними є agetty і mingetty. Саме getty видає підказку «login:». Рядок в / etc / inittab, що задає запуск версії getty, виглядає так

1:2345: respawn :/ sbin / mingetty tty1 (ключове слово respawn означає, що процес буде перезапущений, якщо він припиниться).

Після того, як користувач ввів своє ім'я, getty викликає програму / bin / login, яка запитує пароль (видавши підказку «password:»), перевіряє його і ініціалізує сесію користувача. В більшості випадків це зводиться до запуску для користувача копії командного інтерпретатора (зазвичай bash) в його домашньому каталозі. В результаті користувач може почати сеанс роботи з системою.

Процес init залишається в пам'яті і після завантаження. Він відповідає за автомати-ний перезапуск процесів (для цього потрібно прописати програму в / etc / inittab с дією respawn, як getty), він також стає предком для всіх процесів, чий безпосередній предок припинив свою роботу.

При перезавантаженні або зупинці системи вона переходить на певні рівні виконання і при цьому виконуються скрипти із / etc / rc.d / init.d через зв'язки в каталогах для цих рівнів (rc0.d для зупинки, rc6.d для перезавантаження, такі зв'язки починаються на K). Для того, щоб почати перезавантаження або зупинити систему, використовується спеціальна програма / sbin / shutdown, яка доступна тільки супер користувачеві root. Крім цього консоль Linux підтримує перезавантаження від клавіатури шляхом натискання Ctrl Alt Del.

1.21.4 Загрузка Windows 2000

Завантаження Windows 2000 починається стандартним чином - з передачі управління коду завантажувального сектора активного розділу диска. Основна задача коду завантажувального сектора - визначити місцезнаходження файлу ntldr в кореневому каталозі цього розділу, завантажити його в пам'ять і передати управління на його точку входу. Код завантажувального сектора залежить від того, яка файлова система

встановлена для даного розділу - для FAT виконується один варіант, для NTFS - інший.

Ntldr може розглядатися як завантажувач другого етапу. Він починає своє виконання в 16-бітному реальному режимі процесора, перше, що він робить - це переводить процесор в захищений режим і включає підтримку сторінкової організації пам'яті. Після цього він зчитує з кореневого каталогу файл boot.ini і виробляє його синтаксичний розбір. Ось фрагмент файлу *boot.ini*:

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000"
C:\="Windows 98"
```

Після тега [boot loader] заданий варіант завантаження за замовчуванням і час, після закінчення якого, система буде автоматично завантажуватися відповідно з цим варіантом, після [operating systems] заданий список можливих варіантів завантаження. Для кожного варіанта завантаження може бути заданий один з декількох адрес завантаження:

- розділ з кореневим каталогом WINNT (для загрузки Windows 2000);
- буквене позначення розділу, на якому знаходитися інша ОС;
- ім’я файлу з покажчиком розділу.

У разі зазначення літерного імені розділу ntldr знаходить на диску файл bootsec.dos (в якому при установці Windows 2000 зберігається завантажувальний сектор DOS або Consumer Windows, коли поверх нього записується завантажувальний сектор Windows 2000), перемикає процесор в реальний режим і починає виконувати код цього завантажувального сектора.

Якщо задано ім'я файлу, ntldr буде завантажувати файл з таким ім'ям, таким чином, якщо у файлі зберегти завантажувальний сектор іншої ОС, наприклад, Linux, ntldr зможе завантажити і його, для цього варіант завантаження має виглядати так:

C:\bootsec.lnx="Linux"

При завантаженні Windows 2000 розділ з установкою Windows 2000 в boot.ini не зобов'язаний збігатися з розділом, з якого відбувається завантаження, тому що таких розділів може бути кілька.

Якщо у нас один варіант завантаження, система відразу починає завантажуватися, якщо їх більше одного - відображається меню завантаження. Після вибору варіанта з меню ntldr запускає програму ntdetect.com, яка в реальному режимі визначає базову конфігурацію комп'ютера (аналогічно, як це робила функція setup () для Linux). Зібрана інформація зберігається в системі і пізніше збережеться в реєстрі. Внизу екрана з'являється індикатор прогресу. При бажанні можна натиснути F8 і перейти в меню додаткових можливостей завантаження (в безпечному режимі і т. д.).

Далі ntldr завантажує в пам'ять ntoskrnl.exe (містить ядро і виконавчу підсистему Windows 2000), bootvid.dll (відеодрайвер за замовчуванням, який відповідає за відображення інформації в ході завантаження), hal.dll (рівень абстрагування від устаткування) та основні файли реєстру. Після цього він визначає з реєстру, які драйвери встановлені в режимі запуску при завантаженні (це, наприклад, драйвер жорсткого диска) і завантажує їх (без ініціалізації). Завантажується також драйвер кореневої файлової системи. На цьому роль ntldr у завантаженні завершується і він викликає головну функцію в ntoskrnl.exe для продовження завантаження.

Ініціалізація ntoskrnl.exe складається з двох етапів: фази 0 і фази 1. Багато підсистеми виконавчої системи беруть параметр, що задає, в якій фазі ініціалізації зараз знаходиться система.

При виконанні фази 0 переривання заборонені, на екрані нічого не відображається. Основною метою даного етапу є підготовка початкових структур даних, необхідних для розширеної ініціалізації під час виконання фази 1. Відзначимо, що менеджер процесів на даному етапі ініціалізується майже повністю, з його допомогою створюється початковий об'ектпроцесс

під назвою Idle, процес System і системний потік для виконання ініціалізації фази 1.

Після завершення фази 0 дозволяються переривання і починає виконання системний потік. В ході виконання фази 1 управління екраном здійснюється відеодрайвером bootvid.dll, який відображає завантажувальний екран і графічний індикатор прогресу на ньому, який буде змінюватися протягом усього даної фази. Відбувається остаточна ініціалізація різних підсистем виконавчої системи (менеджера об'єктів, планувальника, служби безпеки, менеджера віртуальної пам'яті, менеджера кеша і т. д.). В ході ініціалізації підсистеми вводу-виводу (яка займає до 50% часу даної фази) відбувається підготовка необхідних структур даних, ініціалізація драйверів з запуском при завантаженні (boot-start), завантаження і ініціалізація драйверів з системним запуском (system-start). Фаза 1 завершується запуском менеджера сесій (smss.exe).

Подальша завантаження проводиться трьома системними процесами, які ми розглянули в главі 1 (пункт 1.8.2.3) - менеджером сесій smss.exe, процесом реєстрації в системі winlogon.exe і менеджером управління сервісами (SCM, services.exe). Основним завданням менеджера сесій є завантаження і ініціалізація всіх компонентів підсистеми Win32 (як користувацького режиму, так і режиму ядра), а також остаточна ініціалізація реєстру та запуск winlogon.exe.

Процес реєстрації в системі запускає менеджер управління сервісами, менеджер аутентифікації (lsass.exe), і відображає вікно входу в систему (за відображення вікна відповідає окремий компонент системи msgina.dll). Після успішного входу користувача в систему запускається додаток, що в реєстрі зареєстровано як оболонка (за умовчанням це explorer.exe).

Менеджер сервісів (SCM) завантажує і ініціалізує сервіси користувацького режиму, які встановлені в режимі автоматичного завантаження, цей процес може тривати вже після початку інтерактивної роботи користувачів. Після ініціалізації сервісів завантаження вважається успішною.

1.22. ОСНОВИ ФУНКЦІОНУВАННЯ БАГАТОПРОГРАМНИХ ОПЕРАЦІЙНИХ СИСТЕМ (СТРУКТУРА, СКЛАД, ВЗАЄМОДІЯ)

Подальший розгляд ядра операційної системи буде засновано на деякій деталізації операційних систем UNIX і OS / 2.

1.22.1. Принципи функціонування UNIX

Ядро UNIX і взаємодія завдань

На мал. 1.34 зображена архітектура верхнього рівня системи UNIX. Технічні засоби, показані в центрі діаграми, виконують функції, що забезпечують функціонування операційної системи.

Операційна система взаємодіє з апаратурою безпосередньо, забезпечуючи обслуговування програм та їх незалежність від деталей апаратної конфігурації. Якщо уявити систему складається з пластів, в ній можна виділити ядро системи, ізольоване від користувацьких програм. Оскільки програми не залежать від апаратури, їх легко переносити з однієї системи UNIX в іншу, що функціонує на іншому комплексі технічних засобів, якщо тільки в цих програмах не мається на увазі робота з конкретним обладнанням, якого немає на новій машині. Наприклад, програми, розраховані на певний розмір машинного слова, набагато важче перекладати на інші машини в порівнянні з програмами, що не вимагають подібних параметрів.

Мал. 1.34. Архітектура системи UNIX

Програми, подібні командному процесору shell і редакторам (ed та vi), показані на зовнішньому по відношенню до ядра, шарі, взаємодіють з ядром за допомогою набору звернень до операційної системи. Звернення до операційної системі примушують ядро до виконання різних операцій, яких вимагає викликає програма, і обміну даними між ядром і програмою. Деякі з програм, наведених на малюнку, в стандартних конфігураціях системи відомі як команди, однак, на одному рівні з ними можуть знаходитися і доступні користувачеві програми, такі як програма a.out, стан-дротяні ім'я для виконуваного файлу, створеного компілятором з мови Сі. Інші прикладні програми розташовуються вище зазначених програм на верхньому рівні, як це показано на малюнку. Наприклад, стандартний компіляції тор з мови Сі, cc, розташовується на самому зовнішньому шарі: він викликає препроцесор для Сі, асемблер і завантажувач (компонувальник), тобто окремі програми попереднього рівня. Хоча на малюнку приведена дворівнева ієрархія прикладних програм, користувач може розширити ієрархічну структуру на стільки рівнів, скільки необхідно. Стиль програмування, прийнятий в систем UNIX, допускає розробку комбінації програм, що виконують одну і ту ж загальну задачу.

Багато прикладні підсистеми і програми, що становлять верхній рівень системи, такі як командний процесор shell, редактори, SCCS (система обробки вихідних текстів програм) та пакети програм підготовки документації, поступово стають синонімом поняття "система UNIX". Однак, всі вони користуються послугами програм нижніх рівнів і в кінцевому рахунку, ядра з допомогою набору звернень до операційної системи. Набір звернень до операційної системи разом з реалізують їх внутрішніми алгоритмами складають "тіло" ядра. Коротше кажучи, ядро реалізує функції, на яких основується виконання всіх прикладних програм в системі UNIX, і ним же визначаються ці функції.

На мал. 1.33. рівень ядра операційної системи зображений безпосередньо під рівнем прикладних програм користувача. Виконуючи різні елементарні операції по запитах користувацьких процесів, ядро забезпечує функціонування користувальницького інтерфейсу, описаного вище.

Серед функцій ядра можна відзначити:

• Управління виконанням процесів за допомогою їх створення, завершення або призупинення та організації взаємодії між ними.

• Планування черговості надання виконуваним процесам часу центрального процесора (диспетчеризація). Процеси працюють з центральним процесором в режимі поділу часу. Центральний процесор виконує процес, а по завершенні відлічуваного ядром кванта часу процес припиняється і ядро активізує виконання іншого процесу. Пізніше ядро запускає призупинений процес.

• Виділення виконуваному процесу оперативної пам'яті. Ядро операційної системи дає процесам можливість спільно використовувати ділянки адресного простору на певних умовах, захищаючи при цьому адресний простір, виділений процесу, від втручання ззовні. Якщо системі потрібна вільна пам'ять, ядро звільняє пам'ять, тимчасово вивантажуючи процес на зовнішні запам'ятовуючі пристрої, які називають пристроями вивантаження. Якщо ядро вивантажує процеси на пристрої вивантаження це-

ликом, така реалізація системи UNIX називається системою із свопінгом (підкачкою), якщо ж на пристрій вивантаження виводяться сторінки пам'яті, така система називається системою із заміщенням сторінок.

• Виділення зовнішньої пам'яті з метою забезпечення ефективного збереження інформації і вибірка даних користувача. Саме в процесі реалізації цієї функції створюється файлова система. Ядро виділяє зовнішню пам'ять під користувальницькі файли, фіксує невживану пам'ять, структурує файлову систему в формі, доступній для розуміння, і захищає файли користувача від несанкціонованого доступу.

• Управління доступом процесів до периферійних пристроїв, таким як термінали, стрічкові пристрої, дисководи та мережеве обладнання.

Ядро виконує свої функції, приховуючи від користувача багато нюансів, зв’язаних з особливостями апаратури та архітектури системи. Наприклад, воно дізнається, що даний файл є звичайним файлом або пристроєм, але приховує це розходження від користувацьких процесів. Так само воно, форматуючи ін-формацію файлу для внутрішнього зберігання, захищає внутрішній формат від користувацьких процесів, повертаючи їм не відформатований потік бай-тов. Нарешті, ядро реалізує ряд необхідних функцій по забезпеченню виконання процесів користувацького рівня, за винятком функцій, кото-які можуть бути реалізовані на самому користувацькому рівні. Наприклад, ядро виконує дії, необхідні shell'у як інтерпретатору команд: воно дозволяє процесору shell читати вводяться з терміналу дані, динамічно породжувати процеси, синхронізувати виконання процесів, відкривати канали і переадресовувати ввід / вивід. Користувачі можуть розробляти свої версії командного процесора shell з тим, щоб привести робоче середовище у відповідність зі своїми вимогами, не зачіпаючи інших користувачів. Такі програми користуються тими ж послугами ядра, що й стандартний процесор shell.

Система UNIX дозволяє таким пристроям, як зовнішні пристрої введення/вивода і системні годинник, асинхронно переривати роботу

центрального процесора. Після отримання сигналу переривання ядро операційної системи зберігає свій поточний контекст (застиглий образ виконуваного процесу), уста-встановлюються причину переривання і обробляє його. Після того, як переривання буде оброблено ядром, перерваний контекст відновлюється і робота про-продовжували так, як ніби нічого не сталося. Пристроям зазвичай приписуються пріоритети відповідно до черговості обробки переривань. В процесі обробки переривань ядро враховує ці пріоритети і блокує обслуговування переривання з низьким пріоритетом, на час обробки переривання з більш високим пріоритетом.

Особливі ситуації пов'язані з виникненням незапланованих подій, що вікликані процесом, таких як неприпустима адресація, завдання привілейованих команд, розподіл на нуль і т.д. Вони відрізняються від переривань, які викликаються подіями, зовнішніми по відношенню до процесу. Особливі ситуації метушні кают безпосередньо при виконанні команди, і система, обробивши особливу ситуацію, намагається перезавантажити команду, вважається, що переривання виникають між виконанням двох команд, при цьому система після обробки переривання продовжує виконання процесу, вже починаючи з наступної команди фахівців. Для обробки переривань і особливих ситуацій в системі UNIX виконується один і той же механізм.

Ядро зобов'язане попереджати систему про виникнення переривань з критичними діями, що можуть у разі обробки переривання зіпсувати інформацію. Наприклад, під час обробки списку з покажчиками виникнення переривання від диска для ядра небажано, тому що при обробці переривання можна зіпсувати покажчики. Звичайно є ряд привілейованих команд, що встановлюють рівень переривання процесора в слові стану процесора. Установка рівня переривання на певне значення відсікає переривання цього і більш низьких рівнів, дозволяючи обробку тільки переривань з бо-леї високим пріоритетом. На рис. 1.35. показана послідовність рівнів переривання. Якщо ядро ігнорує переривання від диска, то в цьому випадку

ігноруються і всі інші переривання нижчого рівня, крім переривань від годинника і машинних збоїв.

Мал. 1.35. Стандартні рівні переривань

Ядро постійно розташовується в оперативній пам'яті, поряд з виконуються в даний момент процесом або з частиною його. В процесі компіляції програма-компілятор генерує послідовність адрес, які є адресами змінних і інформаційних структур, а також адресами інструкцій і функцій. Компілятор генерує адреси для віртуальної машини так, немов на фізичній машині не буде виконуватись паралельно з транслюється жодна інша програма.

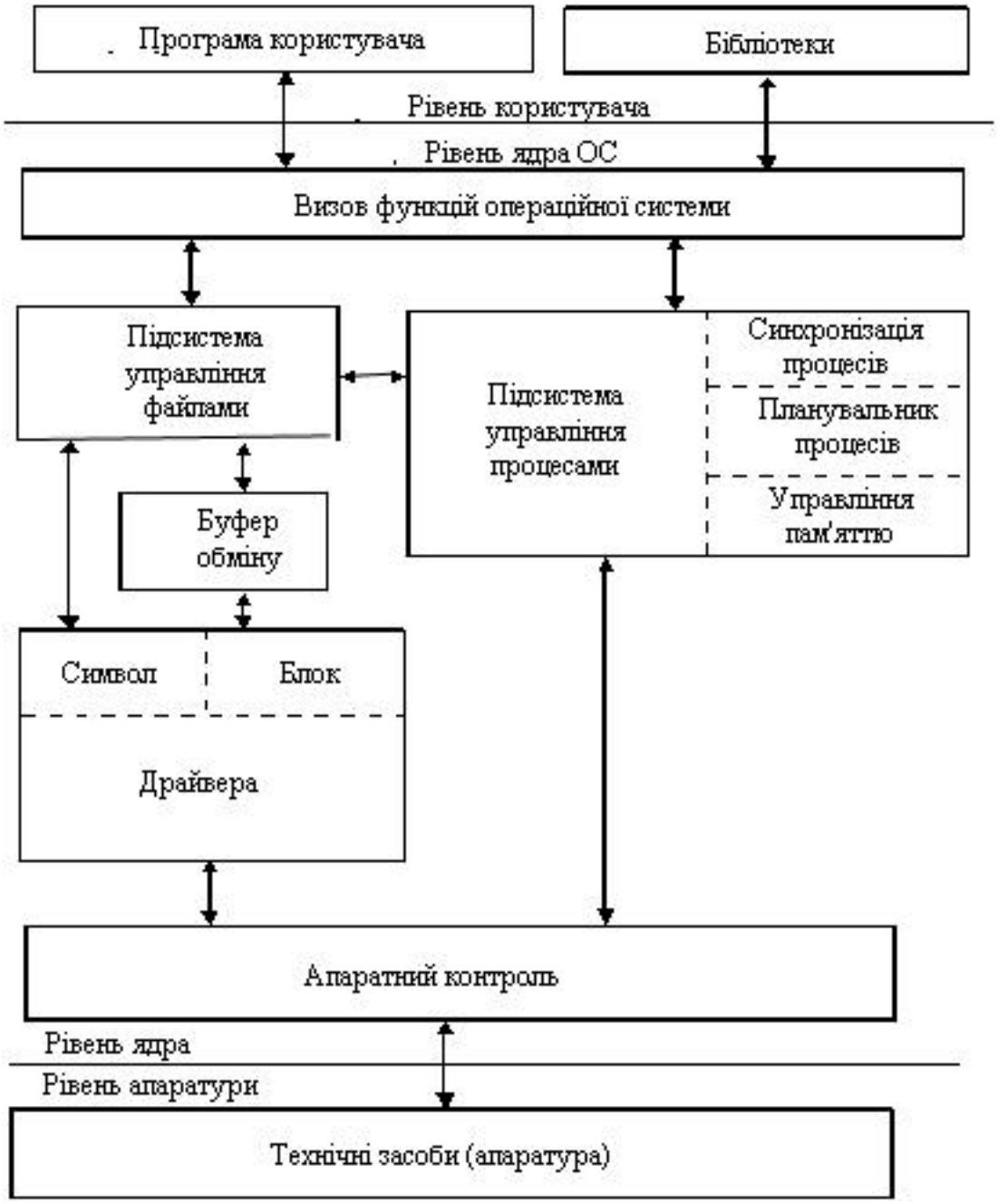
Коли програма запускається на виконання, ядро виділяє для неї місце в оперативній пам'яті, при цьому збіг віртуальних адрес, згенерованих компілятором, з фізичними адресами зовсім необов'язково. Ядро, взаємодіючи з апаратними засобами, транслює віртуальні адреси у фізичні, тобто відображає адреси, згенровані компілятором, у фізичні, машинні адреси. Таке відображення спирається на можливості апаратних засобів, тому компоненти системи UNIX, що займаються їм, є машинно-залежними. Наприклад, окремі обчислювальні машини мають спеціальне обладнання для підкачки вивантажених сторінок пам'яті.

Звернення до операційної системи дозволяють процесам виконувати операції, які вони самі прямим зверненням до ресурсів виконати не можуть.

На додаток до обробки подібних звернень ядро операційної системи здійснює загальні облікові операції, управляє плануванням процесів, розподілом пам'яті та захистом процесів в оперативній пам'яті, обслуговує переривання, управляє файлами та пристроями і обробляє особливі ситуації, що виникають в системі. У функції ядра системи UNIX навмисно не включені багато функцій, які є частиною інших операційних систем, оскільки набір звернень до системи дозволяє процесам виконувати всі необхідні операції на рівні користувача.

Файли і процеси, є центральними поняттями в операційній системі UNIX. На мал. 1.36. представлена структура ядра системи, що відображає склад основних модулів, з яких складається ядро, і їх взаємозв'язку один з одним. Зокрема, на ній зліва зображена файлова підсистема, а праворуч - підсистема управління процесами, дві головні компоненти ядра. Ця схема дає загальне логічне уявлення про ядрі ОС, хоча в дійсності в структурі ядра є відхилення від цієї схеми, оскільки окремі модулі в реальному схемою мають більш складну взаємодію з іншими модулями.

Схема на мал. 1.36. має три рівні: рівень користувача, рівень ядра і рівень апаратури. Звернення до операційної системи і бібліотекам визначають кордон між користувацькими програмами і ядром, проведену на мал. 2.33. Звернення до операційної системи виглядають так само, як звичайні виклики функцій в програмах на мові Сі, а бібліотеки встановлюють відповідність між цими викликами функцій і елементарними системними операціями. При цьому програми на асемблері можуть звертатися до операційної системи безпосередньо, без використання бібліотеки системних викликів. Програми часто звертаються і до інших бібліотек, таким як бібліотека стандартних підпрограм вводу / виводу. Для цього під час компіляції бібліотеки зв'язуються з програмами, які частково включаються в програми користувачів.



Мал. 1.36. Блок - схема ядра операційної системи

Сукупність звернень до операційної системи розділена на ті звертання, які взаємодіють з підсистемою управління файлами і ті, котрі взаємодіють з підсистемою управління процесами. Файлова підсистема управляє файлами, розміщує записи файлів, управляє вільним простором, доступом до файлів і пошуком даних для користувачів. Процеси взаємодіють з підсистемою управління файлами, використовуючи при цьому сукупність спеціальних звернень до операційної системи, таких як open (для того, щоб відкрити файл

на читання або запис), close, read, write, stat (запросить атрибути файлу), chown (змінити запис з інформацією про власника файлу) і chmod (змінити права доступу до файлу).

Підсистема управління файлами звертається до даних, які зберігаються в файлі, використовуючи буферний механізм, керуючий потоком даних між отрута-ром і пристроями зовнішньої пам'яті. Буферний механізм, взаємодіючи з драйверами блокових пристроїв введення / виводу, ініціює передачу даних до ядра і назад. Драйвери пристроїв є такими модулями в складі ядра, які керують роботою периферійних пристроїв. Блокові пристрої введення / виводу відносяться програмами користувача до типу запам'ятовуючих пристроїв з довільною вибіркою та їх драйвери побудовані таким чином, що всі інші компоненти системи сприймають ці пристрої як запам'ятовувальні пристрої з довільною вибіркою. Наприклад, драйвер запам'ятовуючого пристрою на магнітній стрічці дозволяє ядру системи сприймати цей пристрій як накопичувач з довільною вибіркою. Підсистема управління файлами також безпосередньо взаємодіє з драйверами пристроїв "неструктурованого" введення / виводу без втручання буферного механізму. До пристроїв неструктурованого введення / виводу, іноді іменованим пристроями посимвольного введення / виводу (текстовими), відносяться пристрої, відмінні від блокових пристроїв введення / виведення.

Підсистема управління процесами відповідає за синхронізацію процесів, взаємодія процесів, розподіл пам'яті і планування виконання процесів. Підсистема управління файлами і підсистема управління процесами взаємодіють між собою, коли файл завантажується в пам'ять на виконання: підсистема управління процесами читає в пам'ять виконувані файли перед тим, як їх виконати.

Прикладами звернень до операційної системи, що використовуються при управлінні процесами, можуть служити: fork - створення нового процесу, exec - накладення образу програми на виконуваний процес, exit - завершення виконання процесу, wait - синхронізація продовження виконання основного

процесу з моментом виходу з породженого процесу, brk - управління розміром пам'яті, виділеної процесу і signal - управління реакцією процесу на виникнення екстраординарних подій.

Модуль розподілу пам'яті контролює виділення пам'яті процесам. Якщо в якийсь момент система відчуває нестачу у фізичній пам'яті для запуску всіх активізованих процесів, ядро пересилає процеси між основною і зовнішньою пам'яттю з тим, щоб всі процеси мали можливість виконуватися. Існує два способи управління розподілом пам'яті: вивантаження (підкачка) і заміщення сторінок. Програму підкачки іноді називають планувальником, т.я. вона "планує" виділення пам'яті процесам і впливає на роботу планувальника центрального процесора.

Модуль "планувальник" розподіляє між процесами час центрального процесора. Він планує черговість виконання процесів до тих пір, поки вони добровільно не звільнять центральний процесор, дочекавшись виділення ресурсу, або до тих пір, поки ядро системи не вивантажить їх після того, як їхній час виконання перевищить заздалегідь певний квант часу. Планувальник вибирає на виконання готовий до запуску процес з найвищим пріоритетом; виконання попереднього процесу (припиненого) буде поновлено тоді, коли його пріоритет буде найвищим серед пріоритетів усіх готових до запуску процесів. Існує кілька форм взаємодії процесів між собою: від асинхронного обміну сигналами про події до синхронного обміну повідомленнями.

Нарешті, система апаратного контролю відповідає за обробку переривань і за зв'язок з машиною. Такі пристрої, як диски та термінали, можуть переривати роботу центрального процесора під час виконання процесу. При цьому ядро системи після обробки переривання може відновити виконання перерваного процесу. Переривання обробляються не самими процесами, а спеціальними функціями ядра системи, перерахованими в контексті виконуваного процесу.

Ядро системи ідентифікує кожен процес по його номеру, який називається ідентифікатором процесу (PID). Нульовий процес є особливим процесом, який створюється "вручну" в результаті завантаження системи; після породження нового процесу (процес 1) нульовий процес стає процесом підкачки. Процес 1, відомий під ім'ям init, є предком якого іншого процесу в системі і пов'язаний з кожним процесом особливим чином.

Користувач, транслюючи вихідний текст програми, створює виконуваний файл, який складається з кількох частин:

• набору "заголовків", що описують атрибути файлу,

• тексту програми,

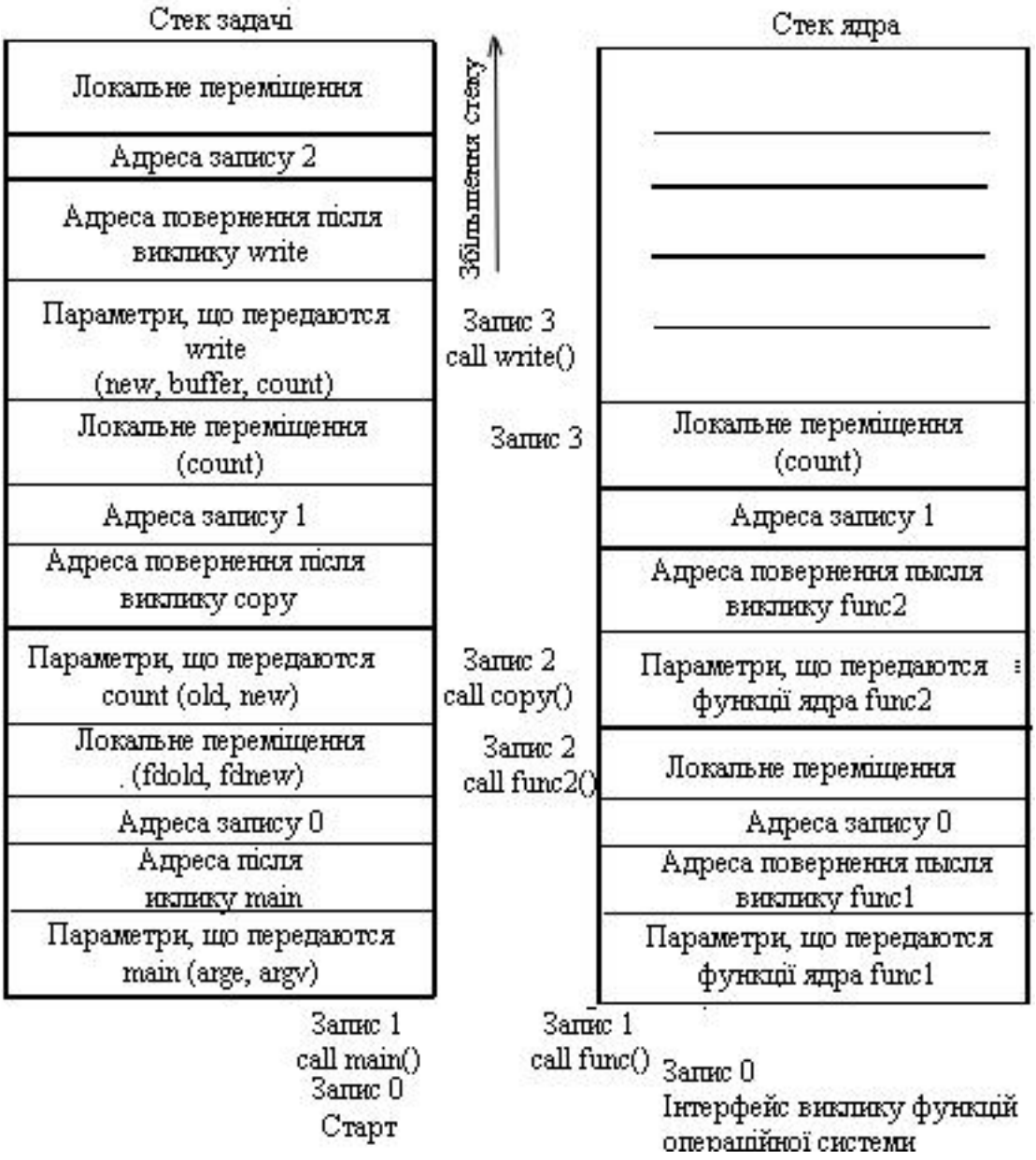
• подання на машинній мові даних, що мають початкові значення при запуску програми на виконання, і вказівки на те, скільки простору пам'яті ядро системи виділить під неініціалізовані дані, так звані bss (*) (ядро встановлює їх в 0 в момент запуску),

• інших секцій, таких як символічні таблиці.

Ядро завантажує виконуваний файл в пам'ять при виконанні системної операції exec, при цьому завантажений процес складається, щонайменше з трьох частин, так званих областей: тексту, даних і стека. Області тексту і даних кореспондуються з секціями тексту і bss-даних виконуваного файлу, а область стека створюється автоматично і її розмір динамічно встановлюється ядром системи під час виконання. Стек складається з логічних записів активації, які розміщені в стек при виклику функції і виштовхує з стека при поверненні управління в викликала процедуру; спеціальний регістр, іменований покажчиком вершини стека, показує поточну глибину стека. Запис активації включає параметри передаються функції, її локальні змінні, а також дані, необхідні для відновлення попереднього запису активації, у тому числі значення лічильника команд і покажчика вершини стека в момент виклику функції.

Текст програми включає послідовності команд, що керують збільшенням стека, а ядро системи виділяє, якщо потрібно, місце під стек.

Оскільки процес в системі UNIX може виконуватися в двох режимах, режимі ядра або режимі завдання, він користується в кожному з цих режимів окремим стеком. Стек завдання містить аргументи, локальні змінні та іншу інформацію щодо функцій, які виконуються в режимі завдання. На малюнку 2.37. ліворуч показаний стек завдання для процесу, пов'язаного з виконанням системної операції write в програмі copy. Початок процесу (включена в бібліотеку) звернулася до функції main з передачею їй двох параметрів, помістивши в стек завдання запис 1; в запису 1 є місце для двох локальних змінних функції main. Функція main потім викликає функцію copy з передачою їй двох параметрів, old і new, і поміщає в стек завдання запис 2; у записі 2 є місце для локальної змінної count. Нарешті, процес активізує системну операцію write, викликавши бібліотечну функцію з тим же ім'ям. Кожній системної операції відповідає точка входу в бібліотеці системних операцій. Бібліотека системних операцій написана на мові Асемблера і включає спеціальні команди переривання, які, виконуючи, породжують "переривання", що викликає перемикання операційної системи в режим ядра. Процес шукає в бібліотеці точку входу, відповідну окремої системної операції, подібно до того, як він викликає будь-яку з функцій, створюючи при цьому для бібліотечної функції запис активації. Коли процес виконує спеціальну інструкцію і перемикається в режим ядра, виконує операції ядра він використовує стек ядра.



Мал. 1.37. Стеки завдання і ядра для програми копіювання.

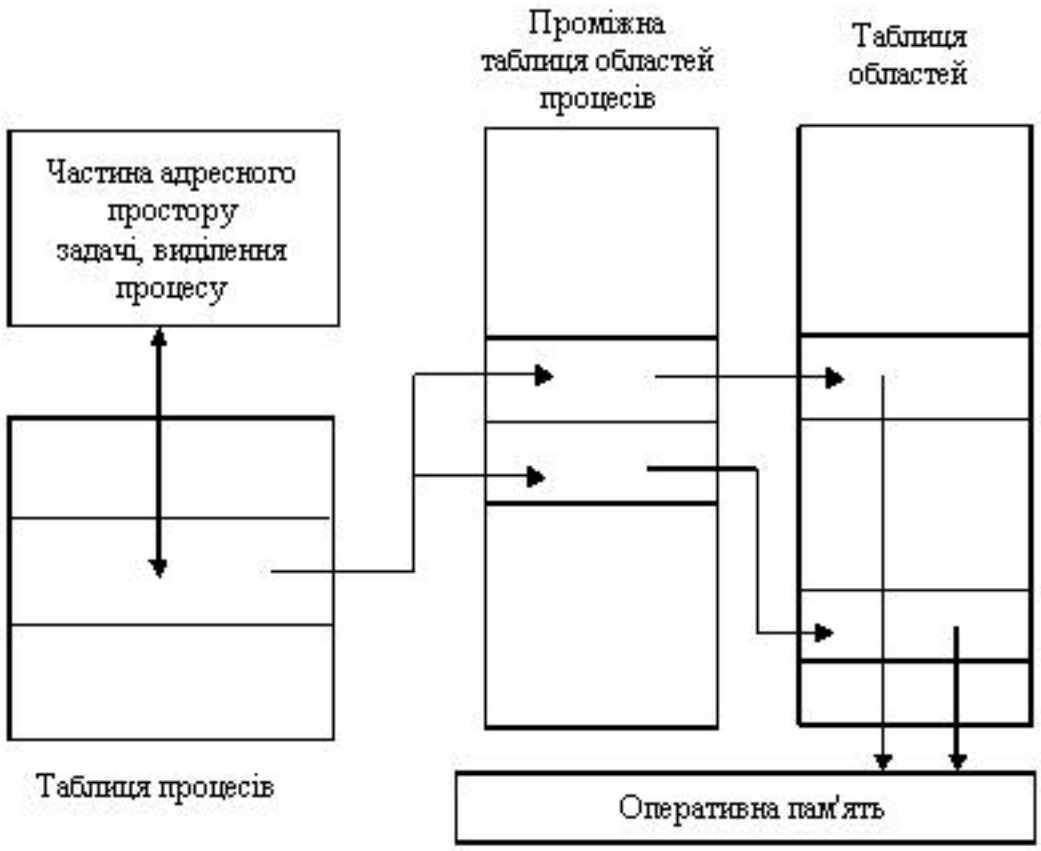
Стек ядра містить записи активації для функцій, що виконуються в режимі ядра. Елементи функцій і даних в стеку ядра відповідають функціям і даним, які належать до ядра, але не до програми користувача, однак, конструкція стека ядра подібна конструкції стека завдання. Стек ядра для процесу порожній, якщо процес виконується в режимі завдання. Праворуч на Малюнку 2.37 представлений стек ядра для процесу виконання системної операції write в програмі copy.

Кожному процесу відповідає точка входу в таблиці процесів ядра, крім того, кожному процесу виділяється частина оперативної пам'яті, відведена під завдання користувача. Таблиця процесів включає в себе покажчики на проміжну таблицю областей процесів, точки входу, які служать в якості покажчиків на власне таблицю областей. Областю називається безперервна зона адресного простору, що виділяється процесу для розміщення тексту, даних і стека. Точки входу в таблицю областей описують атрибути області, як, наприклад, зберігається чи в області текст програми або дані, закрита ця область або ж спільно використовувана, і де конкретно в пам'яті розміщується вміст області. Зовнішній рівень непрямої адресації (через проміжну таблицю областей, використовуваних процесами, до власне таблиці областей) дозволяє незалежним процесам спільно використовувати області. Коли процес запускає системну операцію exec, ядро системи виділяє області під її текст, дані і стек, звільняючи старі області, які використовувалися процесом. Якщо процес запускає операцію fork, ядро подвоює розмір адресного простору старого процесу, дозволяючи процесам спільно використовувати області, коли це можливо, і, з іншого боку, виробляючи фізичне копіювання вмісту цих областей. Якщо процес запускає операцію exit, ядро звільняє області, які використовувалися процесом. На малий. 2.38 зображені інформаційні структури, пов'язані з запуском процесу. Таблиця процесів посилається на проміжну таблицю областей, використовуваних процесом, в якій містяться покажчики на записи у власне таблиці областей, відповідні областям для тексту, даних і стека процесу.

Запис в таблиці процесів і частина адресного простору завдання, виділена процесу, містять керуючу інформацію і дані про стан процесу. Це адресний простір є розширенням відповідного запису в таблиці процесів. В якості полів у таблиці процесів виступають:

- поле стану,

• ідентифікатори, які характеризують користувача, що є власником процесу (код користувача або UID), значення дескриптора події, коли процес призупинено (знаходиться в стані "сну").



Мал. 1.38. Інформаційні структури для процесів

Адресний простір завдання, виділене процесу, містить описує процес інформацію, доступ до якої повинен забезпечуватися тільки під час виконання процесу.

Важливими полями є:

• покажчик на позицію в таблиці процесів, відповідну поточному про-цес,

• параметри поточної системної операції, повернені значення і коди помилок,

• дескриптори файлу для всіх відкритих файлів,

• внутрішні параметри вводу / виводу,

• поточний каталог і поточний корінь,

• кордону файлів і процесу.

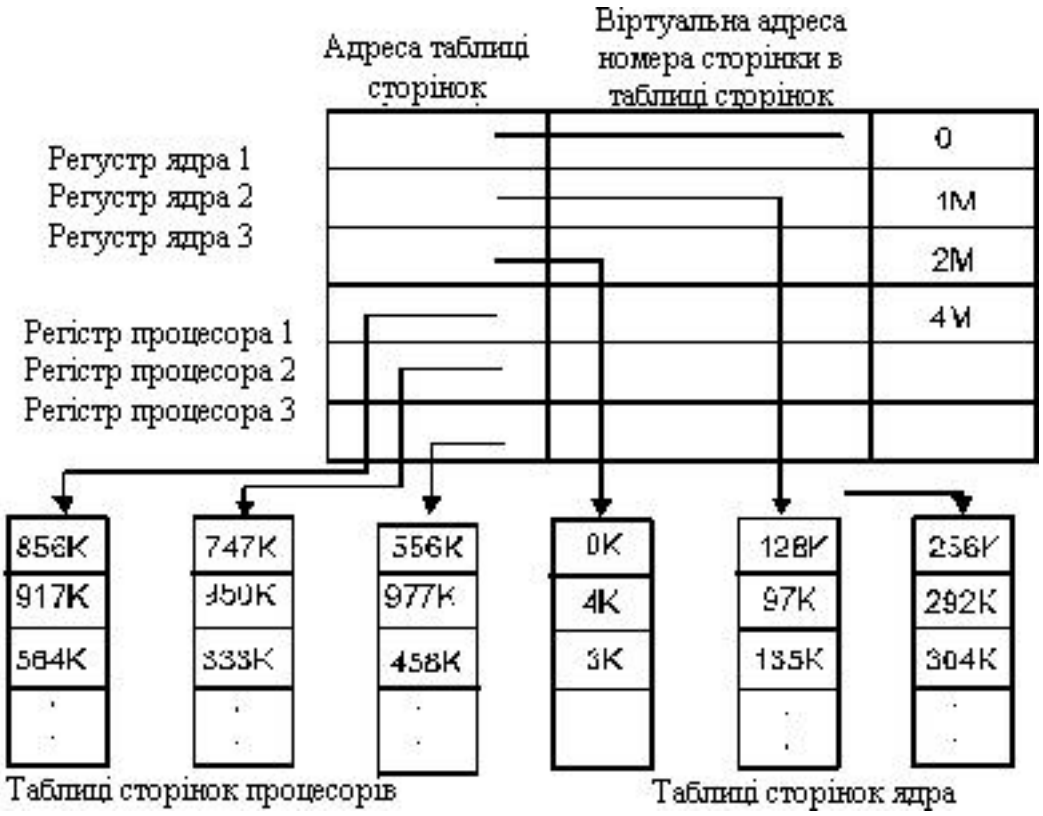
Ядро системи має безпосередній доступ до полів адресного простору завдання, виділеного виконуваному процесу, але не має доступ до відповідних полях інших процесів. З точки зору внутрішнього алгоритму, при зверненні до адресного простору завдання, виділеному виконуваному процесу, ядро посилається на структурну змінну u, і, коли на виконання запускається інший процес, ядро переналаштовує віртуальні адреси таким чином, щоб структурна змінна u вказувала б на адресний простір завдання для нового процесу. У системній реалізації ОС передбачено полегшення ідентифікації поточного процесу, зважаючи на наявність покажчика на відповідний запис у таблиці процесів з адресного простору завдання.

Більшість інформаційних структур ядра розміщується в таблицях фіксованого розміру, а не в динамічно виділеної пам'яті. Перевага такого підходу полягає в тому, що програма ядра проста, але в ній обмежується число елементів інформаційної структури до значення, заздалегідь визначеного при генерації системи. Якщо під час функціонування системи число елементів інформаційної структури ядра вийде за вказане значення, ядро не зможе динамічно виділити місце для нових елементів і повинна повідомити про помилку користувачеві, який зробив запит. Якщо, з іншого боку, ядро згенеровано таким чином, що вихід за межі табличного простору буде маловірогідний, додаткове табличний простір може не знадобитися, оскільки воно не може бути використане для інших цілей. Як би там не було, простота алгоритмів ядра представляється більш важливою характеристикою, ніж ефективне використання оперативної пам'яті. Зазвичай в алгоритмах для пошуку вільних місць в таблицях використовуються нескладні цикли і цей метод більш зрозумілий і іноді більш ефективний у порівнянні з більш ефективними, але складними схемами виділення пам'яті.

Незважаючи на те, що ядро працює в контексті процесу, відображення віртуальних адрес, пов'язаних з ядром, здійснюється незалежно від усіх процесів. Програми та структури даних ядра є резидентними і спільно

використовуються всіма процесами. При запуску системи відбувається завантаження програм ядра в пам'ять з установкою відповідних таблиць і регістрів для відображення віртуальних адрес ядра у фізичні. Таблиці сторінок для ядра мають структуру, аналогічну структурі таблиці сторінок, пов'язаної з процесом, а механізми відображення віртуальних адрес ядра схожі на механізми, що використовуються для відображення користувальницьких адрес. На багатьох машинах віртуальний адресний простір процесу розбивається на декілька класів, у тому числі системний та користувацький, і кожен клас має свої власні таблиці сторінок. При роботі в режимі ядра система дозволяє доступ до адрес ядра, при роботі ж в режимі завдання доступ до них заборонений. Тому, коли в результаті переривання або виконання системної функції відбувається перехід з режиму завдання в режим ядра, операційна система по дозволяє посилання на адреси ядра, а при поверненні в режим ядра ці посилання заборонені. В інших машинах можна міняти перетворення віртуальних адрес, завантажуючи спеціальні регістри під час роботи в режимі ядра.

На мал. 1.39 наведено приклад, в якому віртуальні адреси від 0 до 4М-1 належать ядру, а починаючи з 4М - процесу. Є дві групи регістрів управління пам'яттю, одна для адрес ядра й інша для адрес процесу, причому кожній групі відповідає таблиця сторінок, що зберігає номера фізичної-ческих сторінок з посиланням на адреси віртуальних сторінок. Адресні посилання з використанням групи регістрів ядра допускаються системою тільки в режимі ядра, отже, для переходу між режимом ядра та режимом завдання потрібно тільки, щоб система дозволила або заборонила адресні посилання з використанням групи регістрів ядра.



Мал. 1.39. Переключення режиму роботи з непривілейованого (режиму завдання) на привілейований (режим ядра)

У деяких системах ядро завантажується в пам'ять таким чином, що більша частина віртуальних адрес ядра збігається з фізичними адресами, і функція перетворення віртуальних адрес у фізичні перетворюється на функцію тотожності. Робота з простором процесу, тим не менш, вимагає, щоб перетворення віртуальних адрес у фізичні вироблялося ядром.

Запис в таблиці процесів і частина адресного простору завдання, виділена процесу, містять керуючу інформацію і дані про стан процесу. Це адресний простір є розширенням відповідного запису в таблиці процесів. В якості полів у таблиці процесів виступають:

- поле стану,
- ідентифікатори, які характеризують користувача, що є власником процесу (код користувача або UID),
- значення дескриптора події, коли процес призупинено (знаходиться у стані "сну")..

2.22.2. Мережева операційна система Windows NT

Windows NT (далі іменована NT) - багатозадачна середу для мережевих систем обробки інформації, що функціонує як на високопродуктивних хост - ЕОМ, так і на окремих робочих станціях і володіє вбудованими засобами для роботи з мережами. Будь-яка з робочих станцій у мережі під управлінням NT може одночасно виступати в ролі сервера файлів або друку, бути членом кластера процесорів, що виконують спільну інформаційну завдання, працювати в режимі однокористувацького робочого місця.

В NT реалізовані багато досить відомі концепції, які були до сих пір специфікою лише великих ЕОМ, і при цьому використані новітні інформаційні технології, що відображено в назві системи. "Великі можливості для дешевої ЕОМ" - так можна сформулювати девіз ОС NT.

** Основні особливості системи*

При запуску програми на виконання ОС породжує процес. Процесом називається одиниця управління і споживання ресурсів системи. Як уже згадувалося, NT є багатозадачною ОС. Це означає, що процес не може монопольно захопити обчислювальні ресурси комп'ютера - їх розподіл є функцією ОС. Наприклад, ОС централізовано виділяє кожному з процесів пам'ять, вийти за межі якої вони самостійно не можуть. Цим виключається можливість накладення або перекриття процесів.

У пам'яті комп'ютера можуть одночасно перебувати кілька процесів, багатозадачна ОС повинна передбачати процедуру вибору одного з них і запуску його на виконання. На відміну від перших версій Windows, де перемикання процесів відбувається з ініціативи користувача, в NT їх зміна здійснюється примусово через певні проміжки часу.

** Компоненти NT*

Всі процеси в NT поділяються на системні і користувальницькі. Системні процеси мають безпосередній доступ до апаратних ресурсів і діляться на два класи:

• виконавчий модуль, що забезпечує базові операції ОС, необхідні для роботи інших її компонентів (процесів);

• процеси розширення привілейованого режиму, тісно взаємодіють з апаратно залежними компонентами ОС.

Користувальницькі процеси, що вимагають підтримки засобів системного рівня, також поділяються на два класи:

• захищені підсистеми різних користувацьких інтерфейсів;

• прикладні програми користувачів.

Поділ процесів на типи і класи тісно пов'язане з апаратними можливостями мікропроцесора, званими кільцями захисту.

** Кільця захисту*

Користувачам UNIX відомі системний і користувальницький режими виконання процесу. У режимі користувача виповнюється обмежений набір апаратних інструкцій мікропроцесора, тоді як в системному можуть бути використані будь-які інструкції. Перехід в системний режим відбувається в результаті виконання системного виклику.

Сучасні мікропроцесори мають більше двох рівнів привілеїв виконання інструкцій (наприклад, в мікропроцесорі Intel 80x86 їх чотири). Ці рівні називаються кільцями захисту. Розробники NT розмістили в нульовому кільці захисту тільки найважливішу частину коду ОС - виконавчий модуль. Інші компоненти ОС мають менші привілеями. Це відрізняє NT, скажімо, від NetWare 3.x, де створюються користувачем файли модулі (NLM) можуть виконуватися навіть у нульовому кільці, порушуючи при цьому захист даних.

** Виконавчий модуль*

Виконавчий модуль (Executive) здійснює базові операції нижнього рівня: управління процесами і пам'яттю, диспетчеризацію ресурсів системи, обслуговування переривань і забезпечення міжпроцесорного зв'язку для інших компонентів ОС і для користувача завдань. Кожному процесу в ОС присвоюється певний пріоритет. Завданням виконавчого модуля є вибір процесу з найвищим пріоритетом і його запуск.

** Мікроядро*

Одним з достоїнств UNIX традиційно вважається компактне ядро системи, яке спрощує процес переносу на нову комп'ютерну платформу. У багатьох сучасних ОС ядро виросло настільки, що треба було виділити в ньому мікроядро, що містить апаратно - залежну частину коду.

Одне з технологічних нововведень NT - об'єктно - орієнтована структура мікроядра, об'єктами якого є ресурси. Це означає, що мікроядро взаємодіє з будь-якими ресурсами за допомогою єдиного набору низькорівневих викликів. Для порівняння: в традиційних системах ядро при управлінні драйверами використовувало специфічні для кожного з них команди.

У новій ОС ядро і програмні модулі, що управляють окремими ресурсами, мають загальний канал обміну інформацією, який використовує механізм повідомлень. Отримавши повідомлення, кожен модуль управління ресурсом виконує специфічні дії, наприклад, виведення на термінал або у файл.

В результаті вдалося досягти високої ефективності роботи, централізації управління та гнучкості функціонування. Для кожного модуля використовується документований програмний інтерфейс, що спрощує розробку програмних засобів. В більшості випадків розробники прагнули забезпечити максимальну сумісність цих інтерфейсів з уже існуючими.

** Розширення привілейованого режиму*

Розширення привілейованого режиму забезпечують високорівнева сервіс для інших компонентів системи. Процеси цього класу управляють ресурсами ОС і включають менеджер пам'яті, планувальник процесів, монітор безпеки, драйвери пристроїв, файлову систему і мережні служби.

Реалізація ОС у вигляді сукупності процесів полегшує не лише модифікацію існуючих модулів, а й подальше розширення функцій ОС. Наприклад, управління мережевими службами вбудовано як розширення привілейованого режиму. У UNIX для цієї мети зазвичай використовуються

так звані "процеси - демони", що виконуються в режимі користувача і, отже, менш ефективні і захищені.

** Потокові драйвери*

Динамічно завантажувані драйвери зовнішніх пристроїв мають дворівневу структуру. Драйвери верхнього рівня є апаратно-незалежними, в той час як нижній рівень, організований у вигляді спеціальних бібліотек, враховує особливості конкретних пристроїв.

Для написання нового драйвера зазвичай необхідно лише внести невеликі зміни в модуль нижнього рівня, причому драйвери пишуться не на мові Асемблера, а на Сі. Драйвери пристроїв можуть динамічно завантажуватися в процесі роботи системи.

Драйвери реалізовані як потокові (Streams). Це дозволяє направити інформацію з виходу одного драйвера на вхід іншого, зв'язавши їх у ланцюжки і налагодивши спільну і послідовну роботу.

Можна організувати програмні мультиплексори, направивши інформацію з виходу одного драйвера на вхід декількох драйверів іншого рівня. Потокові драйвери широко застосовуються і в UNIX для організації роботи з мережами і терміналами.

** Файлова система*

Файлова система побудована за принципом драйвера високого рівня, що дозволяє в рамках однієї ОС підтримувати різні способи організації файлів на зовнішніх пристроях, наприклад, CD-ROM; DOS-сумісну таблицю розташування файлів FAT; сумісну з OS / 2 високошвидкісну файлову систему HPFS і власну файлову систему NTFS.

Файлова система NTFS дозволяє працювати зі надвеликими інформаційними томами і має кошти виправлення помилок і заміни дефектних секторів. Спеціальний механізм відстежує і фіксує всі дії, що виконуються з накопичувачами, тому в разі порушення цілісності інформації вона відновлюється автоматично.

** Модель клієнт - сервер*

Принципово важливим у NT є те, що в основу взаємодії прикладної програми і ОС закладена модель клієнт - сервер.

Усі звернення користувальницької програми (клієнта) до ОС переадресовуються ядром, для обробки, спеціальною програмою (серверу), званої захищеної підсистемою. При цьому використовується механізм, аналогічний викликом віддаленої процедури (Remote Procedure Call - RPC), що дозволяє легко перейти від взаємодії між процесами в межах однієї машини до паралельної системі в локальній мережі.

Розробляючи свою захищену підсистему, програміст має можливість використовувати стандартні компілятори і потужні відладники. Повністю налагоджена програма запускається в захищеному режимі.

** Захищені підсистеми*

Важливим компонентом NT є захищені користувальницькі підсистеми, що виконуються в непривілейованому режимі користувача. У вигляді таких підсистем реалізовані інтерфейси прикладних програм (Application Programming Interface - API), причому одночасно підтримуються три типи інтерфейсів: власний 32-розрядний, сумісний з 16-розрядним інтерфейсом DOS і Windows 3.x; інтерфейс для програм OS / 2; POSIX - інтерфейс UNIX - програм.

Завдяки наявності декількох інтерфейсів в NT можна запускати програми, розроблені для різних ОС. Інтерфейс в стандарті POSIX забезпечує сумісність з багатьма програмами на рівні вихідних текстів, що працюють під UNIX. Аналогічна захищена підсистема дозволяє запускати прикладні програми, створені для OS / 2, наприклад, SQL Server.

Новими в порівнянні з Windows, засобами управління ресурсами в NT є семафори, колективна пам'ять, іменовані програмні канали (named pipes). Серед додаткових графічних можливостей - підтримка кривих Безьє і різних

геометричних перетворень, наявність засобів апаратно незалежного управління кольором.

** Організація пам'яті*

Модель пам'яті в NT використовує 32-розрядну лінійну адресацію (без сегментів), поділ та захист як для користувача, так і системних процесів. Застосування сторінкової підкачки файлів (paging) дозволяє завантажити на виконання файл, розмір якого перевершує обсяг ОЗУ комп'ютера. Процеси, що знаходяться в ОЗП, але не виконуються в даний момент, можуть бути тимчасово переміщені на диск в область підкачки, в якості якої використовується спеціальний файл PAGEFILE.SYS. Нововведенням є можливість розміщення цього файлу на декількох дисках.

Один з недоліків традиційних ОС - необхідність надлишкового копіювання інформаційних масивів в різних буферах операційної системи. В NT виконавчий модуль, файлова система і драйвери пристроїв використовують розділяється пул програмних буферів, що дозволяє обійтися без додаткової пересилання даних в процесі вводу / виводу.

** Захист інформації*

Вбудований в ядро NT монітор захисту (Security Monitor) забезпечує єдиний механізм перевірки прав доступу до будь-яких ресурсів системи. Тим самим виключається можливість несанкціонованого доступу до інформації та забезпечується високий ступінь її захисту.

NT не тільки контролює доступ до будь-яких ресурсів, а й протоколює в різних журналах всі дії адміністратора і користувачів, спрямовані на порушення захисту.

** Група користувачів*

В NT є облікова база, що містить імена користувачів і робочих груп (до яких належить той чи інший користувач), а також паролі. База ведеться спеціальним адміністратором. Перед початком роботи системі необхідно повідомити своє ім'я і пароль, які реєструються в обліковій базі, в іншому випадку доступ в систему не надається.

Дуже важлива для NT концепція домена. Домен це об'єднання декількох комп'ютерів, із загальною обліковою базою користувачів і єдиною стратегією забезпечення безпеки. Користувач може зареєструватися на будь-якому з комп'ютерів домену.

Всі користувачі розбиті на групи, що розрізняються привілеями доступу до системи. Найбільш широкими повноваженнями володіють групи адміністраторів. Однак, приналежність до цієї групи не дає права доступу до будь-якої інформації на диску, як в UNIX. Якщо користувач зняв право доступу адміністратора до своїх файлів, то останній не зможе їх використовувати.

Облікова база користувачів знаходиться у віданні адміністратора домену.

В UNIX право користувача на запуск тієї чи іншої програми визначається атрибутами доступу до відповідного виконуваного файлу. В NT ця концепція розширена за рахунок розділення програм на персональні (personal) та загального користування (common). Персональна програма з'являється на екрані у вигляді піктограми тільки у тих користувачів, які мають до неї доступ, для інших вона залишається невидимою.

** Мережеві засоби*

Складовою частиною продукту Windows NT Advanced Server є відомий користувачам Mickosoft Lan Manager. NT дозволяє за допомогою різноманітних мережевих транспортних протоколів що з'єднує з додатками, що працюють в середовищах DOS, Windows, OS / 2 і Mac, а через LAN Manager / X - з UNIX.

** Вимоги до апаратури*

ОС випускається в двох версіях (власне Windows NT і Windows Advanced Server для серверів доменів) і може працювати на комп'ютерах з мікропроцесорами Intel 80x86, RISK - процесорами R4000 і R4400, і в будь-якому випадку, ОС необхідна машина зі значною конфігурацією апаратних засобів. Необхідно мати 12-16 Мбайт оперативної пам'яті і 80-100 Мбайт дискового простору: 30 Мбайт займає власне ОС, 20 Мбайт рекомендується

зарезервувати для файлу підкачки сторінок (його розмір повинен в 1,5 рази перевищувати ємність ОЗУ), засоби розробки й налагодження програм (SDK) займає 35 Мбайт. У RISK - комп'ютерів ці потреби в півтора рази вище. Крім того, система поставляється тільки на CD-ROM, так що необхідний відповідний накопичувач.