О. С. Головня

Основи операційних систем

Навчальний посібник

Міністерство освіти і науки України
Державний університет «Житомирська політехніка»

О. С. Головня

ОСНОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Навчальний посібник

Житомир
2023

УДК 004.45:004.42

Г61

Рекомендовано до друку Вченою радою
Державного університету "Житомирська політехніка",
протокол № 12 від 01 вересня 2023 року.

Рецензенти:

Пількевич І. А. – доктор технічних наук, професор, професор кафедри комп'ютерних інформаційних технологій Житомирського військового інституту імені С. П. Корольова;

Молодецька К. В. – доктор технічних наук, професор, професор кафедри комп'ютерних технологій і моделювання систем Поліського національного університету;

Воротніков В. В. – доктор технічних наук, доцент, професор кафедри комп'ютерної інженерії та кібербезпеки Державного університету "Житомирська політехніка".

Головня О. С.

Г61 Основи операційних систем: навч. посібн. / О. С. Головня. – Електронні дані. – Житомир: «Житомирська політехніка», 2023. – 126 с.

ISBN 978-966-683-617-8

Навчальний посібник містить основні положення з теорії операційних систем. Розглянуто базові поняття операційних систем, шлях еволюції та розмаїття сучасних операційних систем. Викладено основи багатопотоковості, планування та диспетчеризації, керування пам'яттю, файлових систем, безпеки і захисту, міжпроцесової та міжпотокової взаємодії, а також віртуалізації операційних систем.

Навчальний посібник призначено для здобувачів вищої освіти, які навчаються за спеціальностями 123 Комп'ютерна інженерія та 125 Кібербезпека.

В оформленні обкладинки використано фото авторства Dylan Shaw та Jason Yuen

УДК 004.45:004.42

Навчальне видання

ГОЛОВНЯ Олена Сергіївна

Навчальний посібник

Електронне видання

Комп'ютерний дизайн та верстка: Головня О. С.

Державний університет «Житомирська політехніка»

вул. Чуднівська, 103, м. Житомир, 10005

ISBN 978-966-683-617-8

Зміст

Передмова

У навчальному посібнику “Основи операційних систем” розглянуто базові теоретичні питання, пов’язані з будовою та функціонуванням операційних систем.

У даному виданні здійснено огляд розмаїття операційних систем, описано базові принципи роботи операційних систем, розглянуто основи багатопотоковості, планування та диспетчеризації, керування пам’яттю в операційних системах, основи файлових систем, безпеки та захисту операційних систем, організації міжпроцесової та міжпотокової взаємодії, а також технології віртуалізації операційних систем.

Кожний розділ посібника супроводжується контрольними питаннями для перевірки та самоперевірки в здобувачів освіти рівня засвоєння матеріалу. Наприкінці розділів пропонується перелік рекомендованих джерел для всіх, хто потребує глибшого й детальнішого опрацювання відповідних тем.

Оскільки у даному виданні висвітлено передусім теоретичні питання, наголошуємо на важливості доповнення викладеного тут матеріалу серією лабораторних і/або практичних робіт, у межах яких здобувачі освіти мали б змогу поглибити своє розуміння теоретичних основ та здобути необхідні практичні навички з основ адміністрування операційних систем, написання скриптів і/або програм до них.

Посібник розроблено для підготовки бакалаврів за спеціальностями *123 Комп’ютерна інженерія* та *125 Кібербезпека.* За потреби, наведені у посібнику матеріали також можна використати для роботи зі здобувачами освіти спеціальностей *121 Інженерія програмного забезпечення*, *122 Комп’ютерні науки*, *126 Інформаційні системи та технології* й іншими спеціальностями, підготовка за якими передбачає вивчення особливостей будови та функціонування операційних систем.

Авторка радо розгляне всі зауваження та побажання за адресою olenaholovnia@gmail.com.

Розділ 1

Огляд операційних систем

1. Поняття операційної системи (ОС)

Існують різні способи означити поняття операційної системи (ОС). Скористаємося одним із найпоширеніших підходів, використаним зокрема в [3], згідно з яким операційна система розглядається з двох однаково важливих і взаємодоповнювальних поглядів.

З погляду користувача, **операційна система є розширеною машиною** (extended machine). Іншими словами, ОС є посередником між апаратним забезпеченням, з одного боку, та користувачем і користувацькими програмами, з іншого боку.

Взаємодіяти з апаратним забезпеченням напряму незручно. Так, деталі доступу до дискового накопичувача важливо знати системному програмісту, який займається розробкою драйверів до цього накопичувача чи іншого системного програмного забезпечення. Однак для щоденної взаємодії з цим накопичувачем значно простіше мати справу з файлами, папками та дисковими розділами. Втім, на низькому рівні цих понять не існує — це абстракції, що їх пропонує операційна система. Прикладні програми також здебільшого працюють з абстракціями.

Отже, операційна система надає прикладним програмам і користувачу зручний та стандартизований інтерфейс прикладного програмування (рис. 1.1).

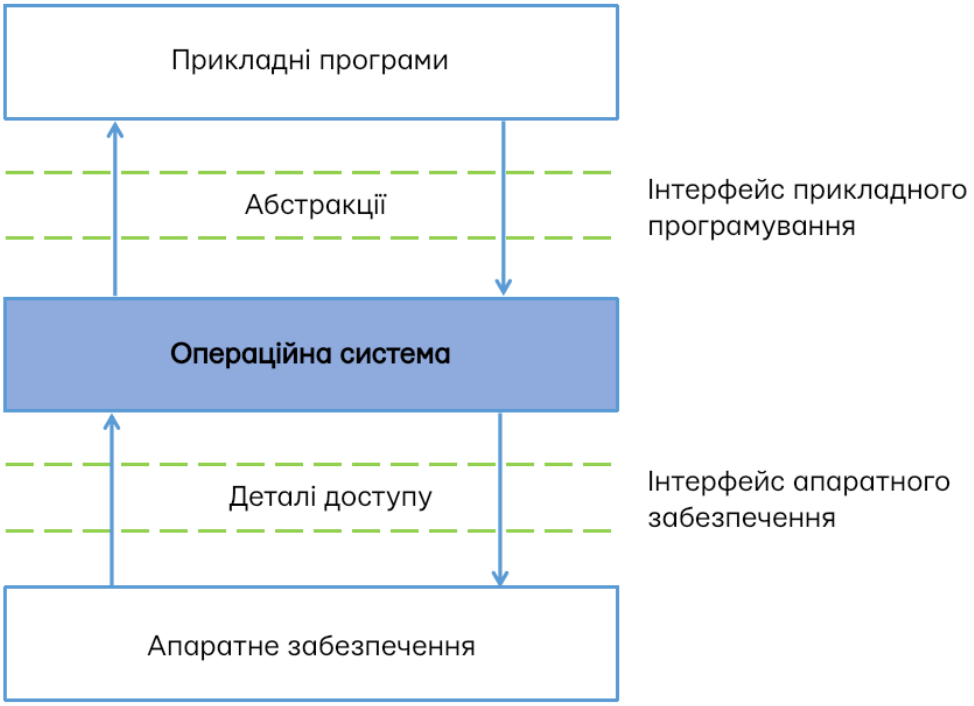

Рис. 1.1. Операційна система як розширена машина

З погляду комп'ютерної системи, **операційна система є менеджером ресурсів** (resource manager). Тобто ОС є спеціальним програмним забезпеченням, яке керує апаратним забезпеченням.

Передусім, важливо з'ясувати, про які саме ресурси йдеться. Це *системні ресурси* (system resources), серед яких можна виокремити, зокрема, наступні [1]:

- центральний процесор (ЦП / central processing unit, CPU)
- пам'ять (memory), передусім - оперативну пам'ять (main memory)
- пристрої введення-виведення (input/output devices, I/O devices)
- накопичувачі (storage)

У теорії операційних систем керування ресурсами поділяють на два види - керування у часі та керування у просторі [3]. Цей поділ є дещо умовним, проте допомагає одержати початкове розуміння управлінських функцій ОС.

Загальну ідею ***керування ресурсами у часі*** можна сформулювати як "спочатку це — потім це". Наприклад, якщо на друк на тому самому принтері було одночасно відправлено два документи, то спочатку буде надруковано документ А, а потім — документ В (а не, скажімо, спершу фрагмент документу А, потім, на тому ж аркуші, фрагмент документу В — це виглядало б безглуздо). Щоб друк відбувався коректно та впорядковано, в ОС наявний спеціальний компонент під назвою *спулер* (spooler), а той, у свою чергу, працює з *чергою друку*.

Інший приклад керування ресурсами у часі — виконання коду програм на ЦП. Цей приклад потребує дещо більш розлогих пояснень.

Одразу зазначимо, що коректніше тут буде говорити про *процеси* (процес — абстракція, що представляє програму під час виконання). Операційні системи, з якими більшість із нас працює щодня — це інтерактивні системи з графічним інтерфейсом і багатьма програмами (процесами), запущеними одночасно. Роботу одних процесів ми бачимо (на екрані є відповідне вікно), робота інших відбувається фоново (вікн*а* може і не бути). Одні з них ми запустили явно (наприклад, клацнувши по відповідному значку чи ввівши

команду у командному рядку), інші запустилися під час завантаження ОС чи пізніше, на вимогу іншого процесу. На рис. 1.2 показано групу процесів *spotify*, пов'язаних з вікном відповідної програми *Spotify* в ОС Ubuntu. Водночас, поряд відображаються імена багатьох інших процесів, які вікон наразі не мають (*goa-daemon*, *gsd-a11y-settings* та ін.).

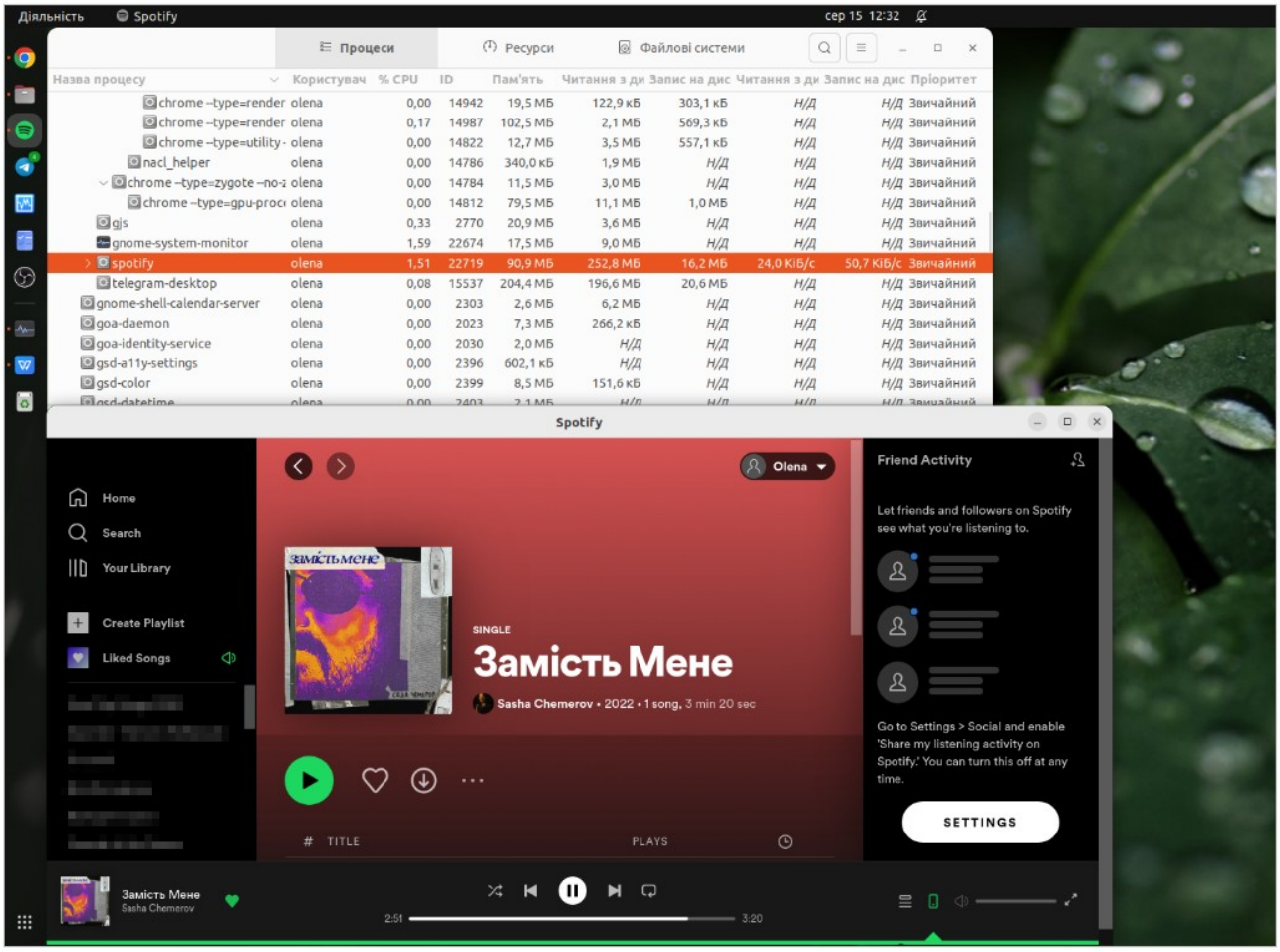

Рис. 1.2. Програма *Spotify* (внизу) та відповідна група процесів *spotify* серед інших процесів (вгорі) в ОС Ubuntu 22.04 Desktop

На практиці ми часто одночасно запускаємо значно більше однієї програми (процесу). І це не враховуючи тих процесів, які працюють фоново, а серед них — і тих, що опрацьовують наші натискання на клавіші клавіатури чи рухи мишкою. Тобто процесів, які претендують на виконання на ЦП у деякий момент часу, може бути більше, ніж процесорів / ядер у цій системі. І все ж ми можемо одночасно запускати досить велику кількість процесів і працювати з ними. Це можливо завдяки швидкому (приблизно через кожні кілька мілісекунд) перемиканню ЦП з виконання одного процесу на виконання іншого процесу. Через таке оперативне перемикання у користувача складається враження, що все виконується *одночасно*, хоч насправді це не зовсім так. Процеси та багатозадачність буде детальніше розглянуто у розділі 2, поки ж звернімо увагу на керування таким ресурсом, як *час ЦП.* Воно здійснюється у часі: *спочатку* виконується процес А, *потім* процес В і т.д.

Основна ідея ***керування ресурсами у просторі*** може бути сформульована як "тут буде це — а тут буде ось це". Наприклад, запущені процеси мають також одночасно зберігати свій програмний код і дані в *оперативній пам'яті*. Важливо

розділити код і дані одного процесу в пам'яті і код та дані іншого процесу. За це також відповідає операційна система.

Дисковий простір теж потрібно організувати з точки зору керування ресурсами у просторі. Один із прикладів — *дискове квотування*, яке часто запроваджується у корпоративних мережах з метою запобігання перевикористання місця на диску одними користувачами і нестачі місця для файлів інших користувачів. Дискова квота обмежує обсяг дискового простору, який може займати своїми файлами окремий користувач.

2. Основні функції та компоненти ОС

Операційні системи надзвичайно різноманітні. Вони працюють на комп'ютерних пристроях різних розмірів та різного призначення: від надпотужних серверів — до смартфонів, від пральних машин — до космічних кораблів, від промислових роботів — до безконтактних платіжних карток. Їх будова та функціонал значною мірою залежать від пристроїв, для яких створено ту чи іншу ОС та умов, у яких ця ОС використовуватиметься. Тож розмову про основні функції та компоненти ОС розпочнемо із застереження: тут буде розглянуто найбільш загальні моменти, спільні для багатьох систем.

За В. Столліґсом [2], до основних цілей розроблення ОС належать:

- зручність (систему зручно використовувати);
- ефективність (ефективне керування ресурсами);
- здатність до розвитку (систему можна вдосконалювати, додавати нові функції).

Будь-яка ОС так чи інакше мусить розвиватися, зокрема з наступних причин.

- ✓ *Розвиток апаратного забезпечення.* Вдосконалення наявних апаратних пристроїв та поява нових, потреби й можливості яких мають підтримувати нові ОС.
- ✓ *Нові сервіси.* Наприклад, тенденції до впровадження засобів легкого доступу потребували внесення відповідного функціоналу в інтерфейс ОС.
- ✓ *Виправлення помилок.* Помилки та недоліки згодом виявляються в усіх системах. Наступні оновлення та нові версії повинні містити їх виправлення.

Теза про різноманітність ОС ще більш справедлива, коли йдеться про склад цих систем. Простіше кажучи, є багато способів відповісти на питання, що є частиною ОС, а що — ні.

Однак є частина, яка завжди входить до складу ОС — це **ядро ОС** (OS kernel), яке містить найважливіші компоненти ОС. Втім, які це компоненти, знову ж таки залежить від конкретної архітектури (основні архітектури ядра ще буде розглянуто у розділі 2).

Часто разом з ОС постачаються також **системні програмні засоби**, потрібні для нормальної роботи ОС та її адміністрування (драйвери пристроїв, системні утиліти тощо). Разом з ОС можуть встановлюватися й **прикладні програми** (текстові та графічні редактори, медіапрогравачі та ін.), однак називати їх частиною ОС ми у даному посібнику все ж не будемо.

У [4] виокремлено наступні функціональні компоненти ОС:

- керування процесами і потоками;
- керування пам'яттю;
- керування введенням-виведенням;

- керування файлами;
- мережна підтримка;
- безпека;
- користувацький інтерфейс.

При цьому важливо враховувати, що на практиці більшість цих компонентів, імовірно, буде частиною ядра ОС, але не обов'язково. Так, у Windows графічний інтерфейс користувача реалізовано в ядрі, а в Linux — поза ним.

3. Історія розвитку ОС

Історія розвитку ОС тісно пов'язана з розвитком програмного забезпечення як такого, а воно, у свою чергу, пов'язане з розвитком апаратного забезпечення. Тому розвиток ОС ми розглядатимемо поряд з розвитком комп'ютерів. Основний наратив використаємо з книги Е. Таненбаума [3].

Передісторія. Появі перших комп'ютерів передували механічні обчислювальні машини. Звісно, програмного забезпечення і, тим більше, операційних систем для цих машин не було, як і операційних систем. Але певним винятком є аналітична машина Беббіджа (Babbage's Analytical Engine, Charles Babbage, 1871 рік). З одного боку, у той час машина (рис. 1.3) так і не була завершена — технології, доступні у XIX столітті, не давали змоги виготовити деталі механізму машини з необхідною точністю. З іншого боку, Беббідж усвідомив необхідність алгоритмів, за якими б виконувала обчислення його машина. Створенням цих алгоритмів займалася Ада Лавлейс (Ada Lovelace). Зрештою, одна з ранніх мов програмування була названа на її честь — Ada.

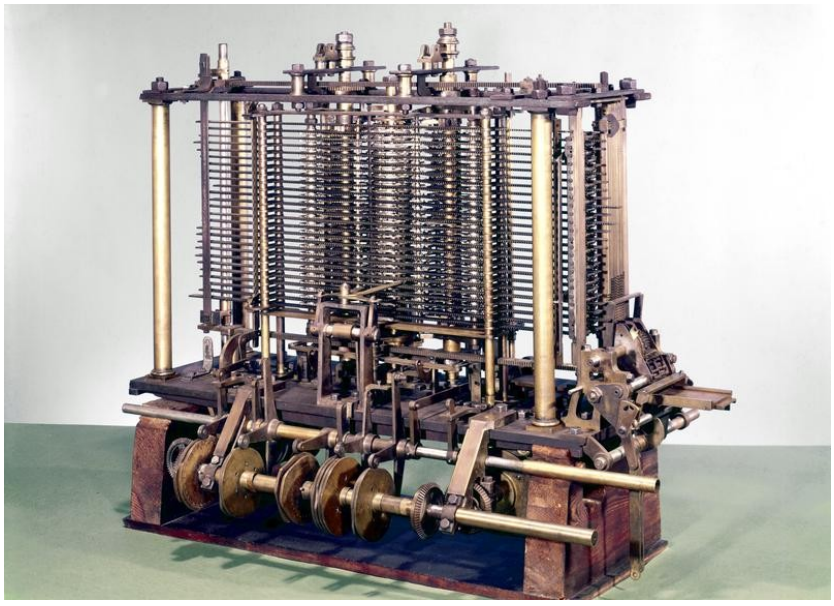

Рис. 1.3. Аналітична машина Бебіджа[1]

Комутаційні панелі. Електронні обчислювальні машини (ЕОМ), тобто комп'ютери у більш звичному розумінні, з'явилися у середині XX століття (1945-1955 рр.). ЕОМ першого покоління працювали на електронних лампах та використовувалися для науково-технічних розрахунків (обчислення значень

[1] Babbage's Analytical Engine, 1834-1871. (Trial model). - Science Museum Group Collection. - URL: https://collection.sciencemuseumgroup.org.uk/objects/co62245/babbages-analytical-engine-1834-1871-trial-model-analytical-engines

тригонометричних та логарифмічних функцій і т. п.). ЕОМ займали цілі зали (рис. 1.4).

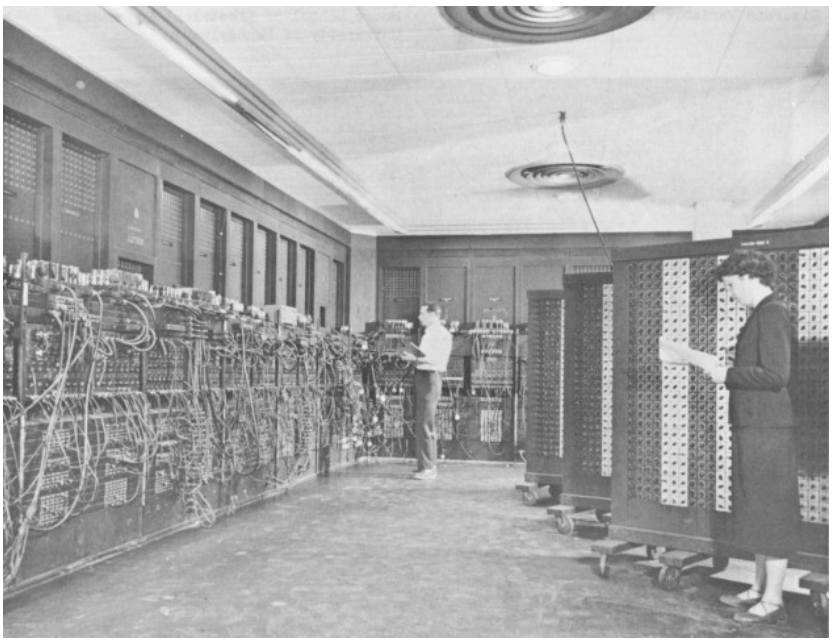

Рис. 1.4. ЕОМ першого покоління ENIAC, університет Пенсильванії[2]

Програмування для таких систем здійснювалося на основі машинної мови (нулів та одиниць), а управління — за допомогою комутаційної панелі (plugboard, рис. 1.5) або, пізніше, пульту (МЕОМ, рис. 1.6). Зазвичай тодішні ЕОМ проєктувалися, розроблялися та використовувалися одним і тим самим колективом людей. Мов програмування ще не існувало (навіть таких, як Assembler). Час для операційних систем ще не настав, тож тут є сенс говорити лише про інтерфейси суто *апаратні*, якими й були комутаційні панелі чи пульт.

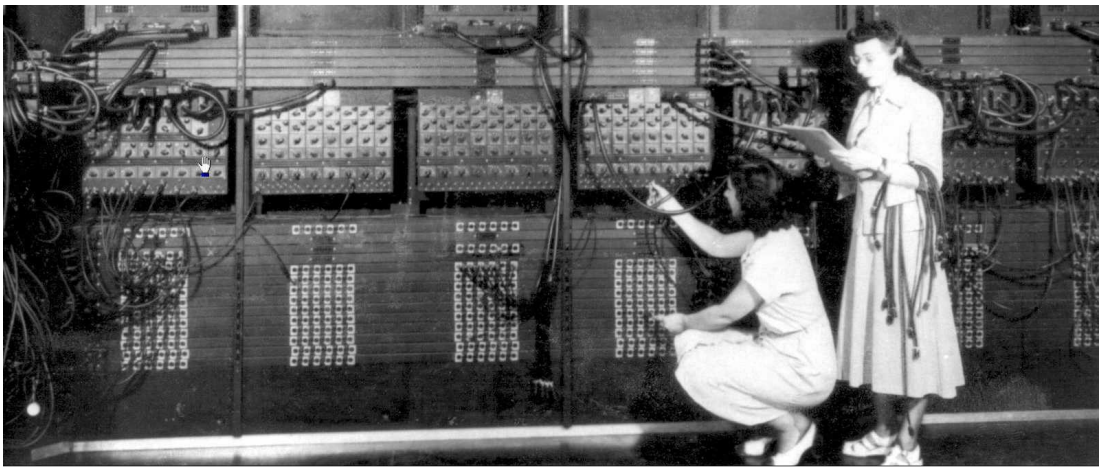

Рис. 1.5. Робота на ENIAC[3]

[2] Programming the ENIAC. - Columbia University Computing History. — URL: http://www.columbia.edu/cu/computinghistory/eniac.html

[3] Programming the ENIAC. - Columbia University Computing History. — URL: http://www.columbia.edu/cu/computinghistory/eniac.html

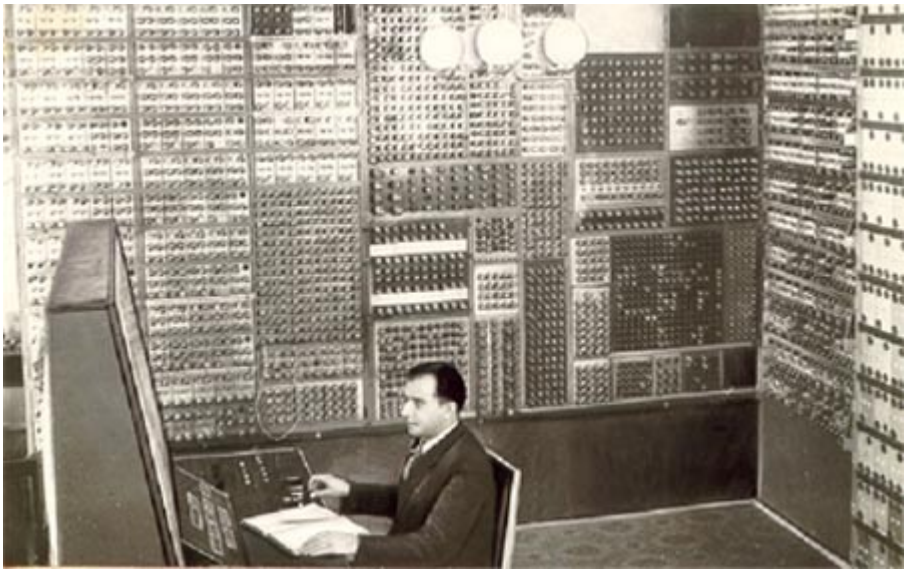

Рис. 1.6. ЕОМ першого покоління МЕОМ, Київ[4]

Системи пакетної обробки. Друге покоління ЕОМ (1955-1965 рр.) використовувало транзистори. Комп'ютери стали надійнішими і тепер підходили для складніших розрахунків (диференціальні рівняння, рівняння у частинних похідних тощо), хоч і досі лишалися громіздкими та дорогими (рис. 1.7).

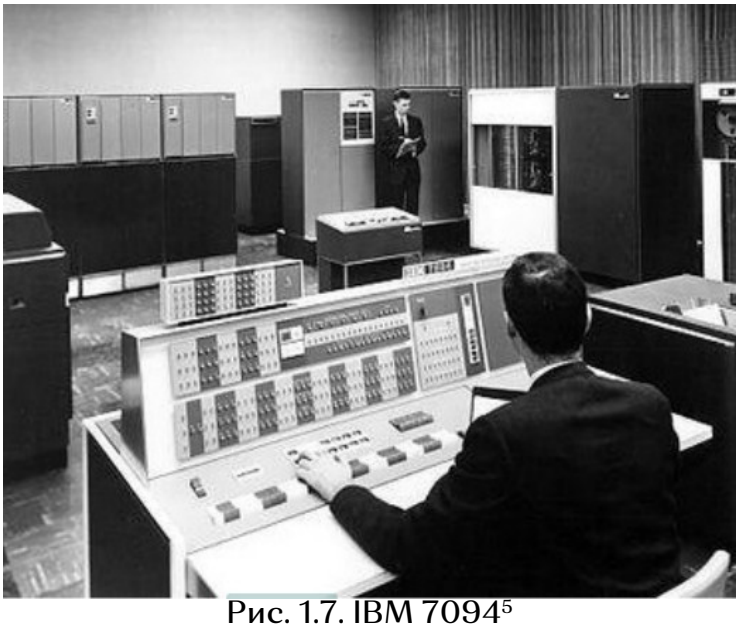

Рис. 1.7. IBM 7094[5]

Також у цей період набували поширення мови програмування (Assembler, Fortran та ін.). Виокремилася категорія працівників, які забезпечували щоденну роботу ЕОМ — оператори. Саме оператор приймав в програміста код програми (на перфокартах), за потреби підвантажував також компілятор відповідної мови програмування, запускав програму на виконання, а потім — передавав результати роботи програми (роздруковані) програмісту. Таким чином, оператор

[4] Перший комп'ютер "МЭСМ". — Музей історії комп'ютерної науки та техніки. — URL: http://www.icfcst.kiev.ua/MUSEUM/PHOTOS/MESM1_u.html

[5] 7094 Data Processing System. — IBM Archives. — URL: https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP7094.html

виконував багато однотипних повторюваних дій, які забирали час, а комп'ютерне обладнання вартістю в мільйони доларів у цей час простоювало. Як іронічно зазначено у [3], забагато часу марнувалося на саме лише ходіння операторів по машинному залу.

Описана ситуація призвела до бажання оптимізувати процес. Так з'явилися ***системи пакетної обробки*** (batch systems), що спиралися у своїй роботі на ідею збирання багатьох завдань разом (у *пакет*), а вже потім — їх виконання.

Загальний хід функціонування системи пакетної обробки проілюстровано на рис. 1.8. Колоду перфокарт з завданнями зчитували на відносно дешевшому комп'ютері (у прикладі на рис. 1.8 це IBM 1401). Зчитані завдання записували на магнітну стрічку, формуючи *пакет*. Магнітну стрічку зчитували на дорожчій ЕОМ (наприклад, IBM 7094). Оператор запускав на IBM 7094 спеціальну програму (*монітор*), яка зчитувала перше завдання з пакету і запускала його на виконання. Та сама програма-монітор за потреби підвантажувала компілятор, виводила результати виконання завдання на магнітну стрічку та переходила до виконання наступного завдання — і так до кінця пакету. Окреме завдання у пакеті містило ключові слова, які вказували, що робити далі (вантажити компілятор, запускати на виконання тощо).

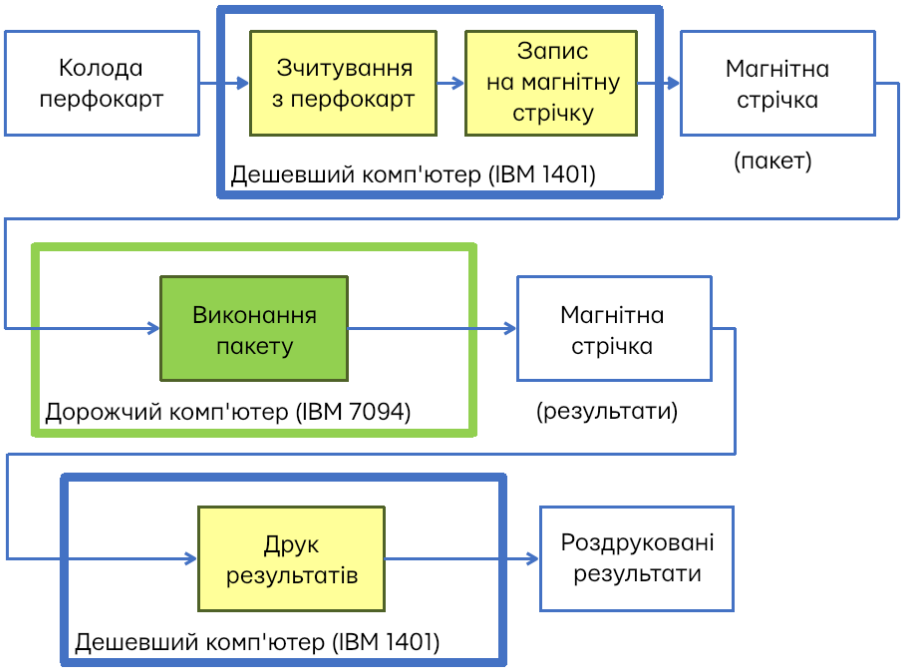

Рис. 1.8. Схема роботи системи пакетної обробки (на прикладі IBSYS)

Коли всі завдання з пакету було виконано, магнітну стрічку з результатами зчитували на IBM 1401 і виводили на друк. Таким чином, програма-монітор перебирала на себе значну частину рутинних обов'язків оператора, забезпечуючи автоматизацію та швидше виконання завдань. Іншими словами, вона робила взаємодію з ЕОМ зручнішою. Системи пакетної обробки називають ранніми операційними системами.

Серед прикладів систем пакетної обробки — FMS (Fortran Monitor System), IBSYS.

Варто зазначити, що некоректно говорити, ніби системи пакетної обробки зникли з появою наступних поколінь ЕОМ та інших різновидів ОС. Системи пакетної обробки і далі використовували в третьому поколінні. На сучасному ж етапі системи, що спираються на ідею опрацювання завдань пакетами, також застосовуються — зокрема, у поєднанні з ґрід-обчисленнями. З систем пакетної

обробки також походять сценарії командного рядка та планування завдань у сучасних ОС (наприклад, *cron* у Unix-подібних ОС чи *Планувальник завдань* у Windows).

Багатозадачність та системи поділу часу. Третє покоління ЕОМ (1965-1980 рр.) знаменується комп'ютерами на інтегральних схемах. Тогочасні комп'ютери все ще значно більші і дорожчі за ПК (котрі ще не винайдено). Водночас, тепер їх дедалі частіше можуть використовувати не лише науковці, а й бізнес. Це створює потребу в зовсім іншому класі систем: орієнтованих не так на обчислення у прямому сенсі цього слова, як на введення і виведення даних. Адже для бізнесу у більшості випадків не потрібні розв'язання диференціальних рівнянь, а більшість операцій пов'язані передусім із введенням та виведенням.

Серед основних нововведень третього покоління назвемо наступні.

Багатозадачність (multiprogramming). Відтепер у пам'яті могло одночасно перебувати декілька завдань — кожному з них виділявся свій розділ пам'яті. Відповідно, доки одне завдання чекало на введення-виведення, інше могло тим часом виконуватися. Це, звісно, відрізняється від сучасної багатозадачності інтерактивних систем, але базова ідея в основі та сама.

Приклад: OS/360.

Спулинг (spooling). Замість запису пакету завдань на магнітну стрічку, всі наявні завдання зчитуються на диск. Тоді, щойно якесь із запущених завдань завершиться, інше завдання підвантажується з диску у звільнений розділ.

Приклад: OS/360.

Режим поділу часу (timesharing). Різновид багатозадачності, за якого є потужна центральна ЕОМ і під'єднані до неї термінали. Термінали дозволяють здійснювати введення-виведення, а для обчислень використовується центральна ЕОМ. Її ЦП надається користувачам по черзі. Так, частина користувачів може бути зайнята обдумуванням своїх програм — ЦП у цей час надається тому, хто запустив програму на виконання.

Приклади: CTSS (Compatible Time Sharing System), MULTICS (MULTiplexed Information and Computing Service).

MULTICS (рис. 1.9) заслуговує на окрему увагу як проєкт, що значною мірою випередив свій час. Метою проєкту було надання доступу до центральної ЕОМ користувачам терміналів в околицях Бостона. Термінали під'єднувалися до центральної ЕОМ по мережі.

Рис. 1.9. Промофото MULTICS на обладнанні HIS, кінець 1970-х[6]

[6] Publicity Photos: 68/80 promo picture. - Multics. - URL: https://www.multicians.org/rr-publicity.html

MULTICS мала частковий успіх. Вона справді використовувалася низкою компаній (рис. 1.9). Однією з причин того, що MULTICS не набула того поширення, яке бачили собі її творці на початку, стала поява персональних комп'ютерів, завдяки чому велика кількість користувачів змогли придбати власні окремі комп'ютери, що їх не потрібно було ділити з іншими користувачами.

Втім, MULTICS заклала основу для низки сучасних технологій. Так, вона справила значний вплив на інші системи, передусім — Unix. Крім того, ідея доступу по мережі до великих обчислювальних потужностей зі слабших пристроїв згодом повернулася у вигляді *хмарних технологій* (cloud computing).

Персональні комп'ютери. Четверте покоління ЕОМ (1980 і дотепер) умовно можна розпочати з появи мікросхем та мікропроцесорів і, як наслідок, персональних комп'ютерів. Ці події надзвичайно сильно вплинули на комп'ютери, якими ми їх знаємо сьогодні. Водночас, сучасні ОС різноманітні за будовою і призначенням:

- вони працюють на дуже різних пристроях;
- одні орієнтовані на користувача, інші — на виконання завдань;
- одні мають графічний інтерфейс та підтримку мультимедіа — в інших наявний лише командний рядок або панель із кнопками;
- різні розробники ОС по-своєму бачать, якими ці ОС мають бути.

Виокремимо деякі важливі події, які стосуються даного етапу (хоч іноді й передують йому хронологічно) і які вплинули на сучасні ОС.

Графічний інтерфейс. Ще у 1960-ті Даґ Енгельбарт (Doug Engelbart) заклав теоретичні основи *графічного інтерфейсу користувача (GUI, Graphical User Interface)*. Тривалий час ідеї мали лише окремі реалізації (Xerox PARC, Lisa), а масового втілення набули саме у середині 1980-х в Apple Macintosh і невдовзі — у Windows.

CP/M. ОС CP/M розробив Ґері Кілдел (Gary Kildall) для процесора Intel 8080, створеного у 1974 році. CP/M було переписано для підтримки й інших наявних у ті часи архітектур, і у кінці 1970-х та на початку 1980-х CP/M була найвпливовішою і найпоширенішою системою для мікрокомп'ютерів (так тоді називали ПК).

Unix. Першу ОС Unix було розроблено у Bell Labs для комп'ютерів PDP-7 у 1970 році. Втім, найвідомішою версією стала Unix 7 (1976), що вплинула на сучасні Unix-подібні ОС (Linux, Mac OS, Android та багато інших) і лежить в основі цих систем. На базі програмного коду Unix виникла низка версій, серед яких BSD (Berkeley Software Distribution, Університет Каліфорнії, Берклі) та System V (компанія AT&T).

MS DOS. ОС MS DOS (Disk Operating System) стала першим продуктом новоствореної компанії Microsoft для комп'ютерів IBM PC. Microsoft створили свою ОС на основі придбаної ними ОС DOS (компанія Seattle Computer Products), переробивши її відповідно до вимог IBM.

Apple Macintosh Operating System. ОС для комп'ютерів Apple з'явилася 1984 року і є першою ОС з графічним інтерфейсом, яка набула масового поширення. Цю ОС побудовано на основі Unix, і дотепер вона лишається не лише Unix-подібною, а й однією з небагатьох сертифікованих Unix-систем (більше про це — у наступних розділах). З тих пір Apple неодноразово змінювали назву своєї ОС: спершу на Mac OS X, потім на OS X, згодом на macOS.

Windows. Компанія Microsoft випустила свою першу ОС з графічним інтерфейсом трохи пізніше за Apple — у 1985 році. Перші версії Windows фактично були надбудовою на MS DOS. Так тривало до появи Window 95, яка вийшла у 1995 році. Архітектура Windows з того часу зазнала кардинальних змін. Найзнаковіші зміни пов'язані з Windows NT (Windows NT 4.0, Windows 2000, Windows XP і решта пізніших версій). Для Windows NT команда під керівництвом

Девіда Катлера (David Cutler) повністю переписала ядро, після чого у Windows з'явилася низка принципових удосконалень, зокрема розмежування прав. Ось чому серверні версії Windows почали виходити саме після появи Windows NT.

Linux. ОС Linux починалася у 1991 році як проєкт Лінуса Торвальдса (Linus Torvalds), на той час — студента університету Гельсінкі (Фінляндія). Надихнувшись ОС Minix (Unix-подібною системою, розробленою Ендрю Таненбаумом, Andrey Tanenbaum), Торвальдс розробив власну невелику Unix-подібну систему з інтерфейсом командного рядка. Спершу Торвальдс не планував, що його ОС виросте у повноцінну ОС, що працюватиме на різних апаратних платформах. Однак проєкт зацікавив велику кількість ентузіастів, а згодом — організацію Free Software Foundation (FSF, засновник і, тривалий час, очільник — Річард Столлмен, Richard Stallmen). FSF займається просуванням ідеї вільно поширюваного програмного забезпечення, і на той момент їм саме бракувало підходящого ядра для власного проєкту вільно поширюваної ОС під назвою GNU (назва є рекурсивним акронімом і розшифровується як GNU's not Unix). Поєднання ядра, написаного Торвальдсом, і зусиль та відомості FSF призвели до подальшого розвитку і Linux, і GNU. Нині ядро Linux має відкритий код і поширюється під ліцензією GNU GPL 2.0, тож на його основі розроблено велику кількість варіантів (*дистрибутивів*, distributions) Linux: Red Hat Enterprise Linux, Cent OS, Stream OS, Fedora, Debian, Ubuntu, Mint, Arch, Gentoo, Slackware, openSUSE та ін. Значна частина дистрибутивів також поширюється вільно. Водночас, деякі є платними комерційними продуктами (Red Rat Enterprise Linux) чи надають платну професійну підтримку (Ubuntu)[7]. Більшість дистрибутивів Linux підтримують один чи декілька графічних інтерфейсів, однак зазвичай можуть використовуватися і без GUI, передусім на серверах і/або на комп'ютерах з малою кількістю ресурсів.

Мережні та розподілені ОС. Більшість сучасних комп'ютерних пристроїв розраховані на роботу у дротових та бездротових комп'ютерних мережах. Втім, важливо розрізняти мережні ОС та розподілені ОС.

До ***мережних ОС*** (network opertaing systems) належать більшість сучасних ОС. Мережна ОС надає користувачу засоби для віддаленого доступу до ресурсів інших комп'ютерів мережі. Структура ОС від цього не зазнає суттєвих змін.

Натомість, ***розподілені ОС*** (distributed operating systems) є менш розповсюдженими. У розподіленій ОС програма користувача виконується на одному з багатьох процесорів мережі, а файл користувача зберігається на одному з багатьох накопичувачів мережі. Користувач не зобов'язаний знати, на якому саме процесорі, на якому саме накопичувачі тощо. Звісно, структура ОС від цього змінюється принципово. Зокрема, алгоритми планування виконання на ЦП мають враховувати передавання даних по мережі.

Приклади розподілених систем: Mach (університет Карнеґі-Меллона), Chorus (INRIA), Sprite (Університет Каліфорнії, Берклі).

Мобільні пристрої. Поруч з четвертим поколінням ЕОМ (персональні комп'ютери) виокремлюють також п'яте покоління ЕОМ (1990 рік - дотепер), пов'язане з появою і поширенням мобільних пристроїв. Втім, персональні комп'ютери та мобільні пристрої співіснують поруч, взаємодоповнюючи одне одного. ПК частіше використовуються для одних завдань, мобільні пристрої — для інших.

Тривалий час існувало дві найбільші й переважно відокремлені категорії мобільних пристроїв — телефони та КПК (кишеньковий персональний комп'ютер, personal digital assistant, PDA). У 1996 році компанія Nokia випустила N9000 (рис. 1.10), який поєднував можливості телефона і КПК. Сама назва "смартфон" з'явилася у 1997 році — компанія Ericsson використала її для свого телефону

[7] DistroWatch. - URL: https://distrowatch.com/

GS88 Penelope (рис. 1.11), який, втім, не було запущено у масове виробництво. І Nokia N9000, і Ericsson GS88 використовували ***GeOS***[8].

Рис. 1.10. Nokia N9000[9]

Рис. 1.11. Ericsson GS88 Penelope[10]

Також протягом першого десятиліття історії смартфонів популярності набула ***Symbian OS***, що широко використовувалася на пристроях Nokia, Samsung, Sony Ericsson, Motorola та ін. Згодом з'явилися ***Blackberry OS*** (компанія RIM, 1992), ***iOS*** (компанія Apple для iPhone, 2007), ***Android*** (компанія Google, 2008). Microsoft мала версію Windows для мобільних пристроїв (спершу ***Windows Mobile***, потім — ***Windows Phone***), підтримку яких було завершено у 2020 році.

---

[8] Nokia 9000 Communicator — Device Specs — PhoneDB. - URL: https://phonedb.net/index.php?m=device&id=879&c=nokia_9000_communicator

[9] The tale of Nokia's amazing 1996 smartphone. — Financial Times. — URL: https://www.ft.com/content/47156464-4cb2-3a19-9572-3a1766f42114

[10] Пенелопа Ericsson GS88. — RARE PHONES MUSEUM: a private collection from Ukraine. — URL: https://rare.pp.ua/ericssongs88/

4. Користувацькі інтерфейси ОС

Користувацький інтерфейс, або інтерфейс користувача (user interface), дає змогу користувачу взаємодіяти з комп'ютерною системою (дієслово *interfare* дослівно перекладається з англійської саме як *взаємодіяти*).

Серед основних типів користувацьких інтерфейсів назвемо:

- ***текстовий інтерфейс*** (text-based interface), або інтерфейс командного рядка (command line interface, CLI);
- ***графічний інтерфейс*** (graphical user interface, GUI);
- інші інтерфейси, які найчастіше використовуються у поєднанні з текстовим та графічним (***сенсорний інтерфейс*** — управління дотиком та рухами, ***голосовий інтерфейс*** — управління голосом тощо).

Інтерфейс командного рядка передбачає введення команд здебільшого з клавіатури та виведення результатів на екран переважно у текстовому вигляді. Інтерфейс командного рядка може бути єдиним інтерфейсом, доступним у системі, коли GUI відсутній — не доступний або не встановлений. Втім, у системах з GUI інтерфейс командного рядка часто доступний у межах GUI (як окреме вікно). Ще один варіант — інтерфейс командного рядка може бути доступний *поряд* з GUI. Ідеться про так звані текстові та графічні термінали у Linux. Зазвичай там є сім терміналів (не плутати з програмою *Термінал*), більшість з яких є текстовими, але кілька можуть бути і графічними. Перехід між цими терміналами найчастіше здійснюється комбінаціями клавіш Ctrl+Alt+одна з функціональних клавіш (F1, F2, ..., F7).

Те, які команди доступні і як саме вони працюватимуть, залежить від *командного інтерпретатора*, або *командної оболонки*.

У Unix-подібних ОС використовується низка загалом подібних між собою командних інтерпретаторів, найпопулярніші з яких *Bourne-Again Shell* (*Bash*), *C shell* (*Csh*), *Korn Shell* (*ksh*), *TENEX C shell* (*tcsh*).

У Windows найпоширенішими командними інтерпретаторами є cmd.exe (програма *Командний рядок*), значна частина команд якого походять ще з MS DOS, та *PowerShell*, який розроблений на заміну cmd.exe, використовує так звані *командлети*, побудовані за об'єкто орієнтованим підходом, і підтримує команди cmd.exe та деякі команди з Unix/Linux.

Графічний інтерфейс оперує графічними абстракціями (робочий стіл, значки, панелі, кнопки тощо).

Перший GUI, що набув масового поширення, був втілений у **macOS** (на той момент — Apple Macintosh Operating System).

GUI, що використовується у **Windows**, має назву *Shell*. Втім, ця назва використовується рідко, оскільки Windows без графічного інтерфейсу застосовується лише в окремих серверних версіях.

Для **Linux** існує багато графічних інтерфейсів (графічних оболонок), серед яких GNOME, KDE, Xfce, LXDE, MATE та ін. У версіях для ПК зазвичай за замовчуванням використовується якась одна оболонка, однак у багатьох випадках її можна й змінити на іншу.

Різні типи користувацьких інтерфейсів часто поєднуються між собою: GUI і CLI; GUI та сенсорний; GUI та голосовий; GUI, сенсорний та голосовий тощо.

Контрольні запитання

1) Поясніть, що таке операційна система з позиції користувача (прикладної програми) та з позиції комп'ютерної системи.

2) Назвіть кілька прикладів системних ресурсів.
3) Наведіть приклади того, як ОС може керувати системними ресурсами у часі та у просторі.
4) Які компоненти можуть входити до складу ОС?
5) Охарактеризуйте системи пакетної обробки. Назвіть приклади таких систем.
6) Дайте загальну характеристику системам поділу часу. Назвіть приклади таких систем.
7) Назвіть основні події у світі ОС на етапі персональних комп'ютерів. Коли та за яких обставин з'явилися ОС Unix, Mac OS, Windows, Linux?
8) Опишіть відокремлення ОС мобільних пристроїв в окремий клас ОС.
9) Поясніть відмінність між мережними ОС та розподіленими ОС.
10) Які типи користувацького інтерфейсу можуть використовуватися у сучасних операційних системах? Назвіть приклади кожного типу.
11) В якій ОС було вперше використано графічний інтерфейс користувача для масового вжитку?

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapters 1-2.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 2.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 1.
4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділ 1.

Розділ 2

Основні принципи роботи ОС

1. Класифікація ОС за сферою застосування

Операційні системи працюють на найрізноманітніших пристроях. Ці пристрої можуть мати різне призначення, різні умови використання, різний обсяг ресурсів, доступних системі — усе це означає, що операційні системи таких пристроїв теж суттєво різнитимуться. Тож операційні системи можуть бути подібними до тих, що працюють на наших ПК, ноутбуках та смартфонах — а можуть керувати зовсім іншими пристроями на зразок пральної машини, конвеєру, дрона чи банківської картки. Іноді йдеться про відомі операційні системи, а іноді — про системи, розроблені спеціально під певний пристрій (наприклад, на базі ядра Linux), і які можуть не мати окремої назви. У цьому посібнику ми розглянемо основні класи операційних систем, орієнтуючись на класифікацію, наведену в [3].

Операційні системи мейнфреймів (mainframe operating systems). Мейнфреймом раніше називали потужну центральну ЕОМ в обчислювальному центрі. Зараз мейнфрейм — це надпотужний сервер[11], орієнтований на опрацювання величезної кількості запитів одночасно. Прикладом мейнфремів є z-серія компанії IBM. На рис. 2.1 показано IBM System z14, а на рис. 2.2 — IBM System z9.

[11]Важливо не плутати мейнфрейм з кластером. Мейнфреймом прийнято називати більш цілісну обчислювальну одиницю (окрема спеціальна шафа, у яку встановлено все обладнання). Кластер може складатися з багатьох обчислювальних одиниць, кожна з яких не обов'язково мусить бути надпотужним комп'ютером. Обчислювальні одиниці кластера зазвичай з'єднуються між собою по мережі.

Рис. 2.1. IBM System z14 (вигляд зовні). *Фото — IBM España*

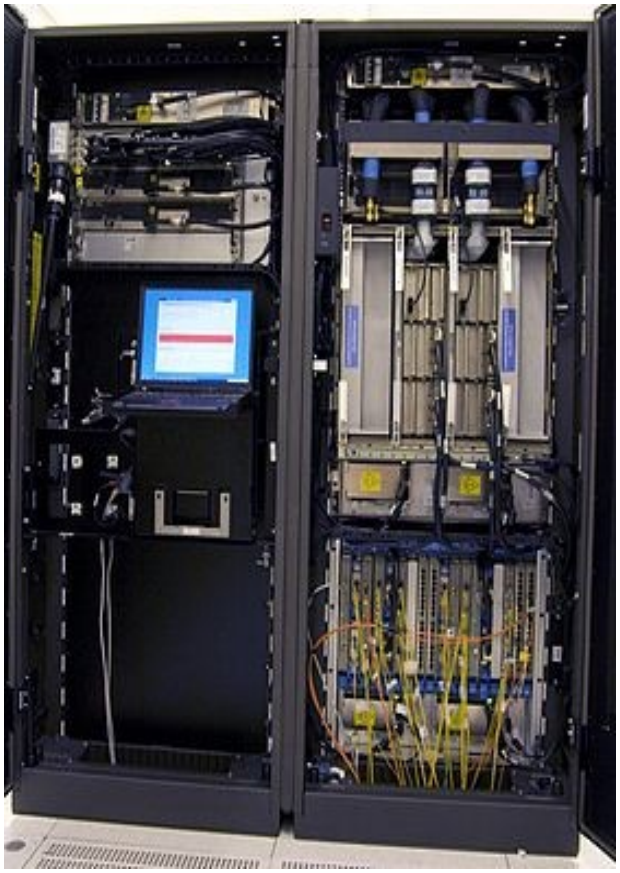

Рис. 2.2. IBM System z9 (вигляд всередині). Фото — *Ing. Richard Hilber*

Мейнфрейми використовуються в ролі серверів для критично важливих сервісів, яким притаманна велика кількість транзакцій.

Важливі акценти для ОС мейнфреймів:

- висока продуктивність введення-виведення;
- обробка великих обсягів даних.

До ОС мейнфреймів належать OS/390, z/OS (працює на Z-серії IBM). Також на мейнфреймах можуть працювати і Unix-подібні системи, нерідко — дистрибутиви Linux.

Серверні операційні системи (server operating systems). На відміну від ОС мейнфреймів, серверні ОС обслуговують сервери, що мають різну потужність (не обов'язково велику) та найрізноманітніше призначення (вебсервери, файлові сервери, сервери друку, сервери автентифікації, поштові сервери та ін.). Вони можуть працювати на досить потужному комп'ютері, окремо виділеному саме під сервер, можуть — на невеликому ноутбуці як студентський курсовий проєкт, а можуть — на віртуальній машині чи віртуальному контейнері, розміщених у хмарі.

Серверні ОС не завжди мають графічний інтерфейс (хоча можуть мати), натомість адаптовані під потреби адміністрування й оснащені відповідними програмними засобами, або такі засоби доступні для довстановлення.

Випуски багатьох сучасних ОС часто містять окремо серверну і настільну версію (Server Version, Desktop Version), однак у випадку одних ОС цей поділ чіткий (Windows), а у випадку інших ОС — досить умовний (Linux).

Важливі акценти для серверних ОС:

- надання спільного доступу до ресурсів;
- безпека.

На серверах використовуються ОС Linux, Windows 20xx, Solaris, FreeBSD та багато інших.

Настільні операційні системи (desktop operating systems) також називають *операційними системами персональних комп'ютерів,* втім, сучасні настільні системи давно вийшли за межі використання лише на ПК. Такі системи можна зустріти на домашніх та офісних ПК, ноутбуках, нетбуках, тонких клієнтах.

З усіх класів операційних систем настільні ОС, напевно, найменше потребують окремого представлення і, власне, саме таку систему більшість людей уявляє, коли чує термін "операційна система". Про настільні ОС передусім важливо розуміти, що вони орієнтовані на користувача (не завжди кваліфікованого) та його щоденні потреби.

Важливі акценти для настільних ОС:

- графічний інтерфейс;
- підтримка мультимедіа.

До настільних ОС належать Windows, MacOS, Linux, Chrome OS, FreeBSD, Syllable та інші. Перші чотири ОС в наведеному переліку формують своєрідний рейтинг популярності. Згідно з даними statistica.com[12], станом на літо 2022 року, частка настільних систем з Windows становила близько 76,3%, з MacOS — близько 14,6%, з Linux — близько 2,4%, з Chrome OS — близько 1,7% (також на основі ядра Linux).

Операційні системи мобільних пристроїв (mobile operating systems). Коли звучить назва цього класу ОС, то передусім спадають на думку ОС смартфонів, проте сюди також належать ОС, які працюють на планшетах, Е-рідерах (пристроях для читання електронних книг), смарт-годинниках та ін. (рис. 2.3).

[12]Global market share held by operating systems for desktop PCs, from January 2013 to June 2022. — URL: https://www.statista.com/statistics/218089/global-market-share-of-windows-7/

Такі пристрої зазвичай мають менший обсяг ресурсів, ніж ПК чи ноутбуки. З іншого боку, ця категорія комп'ютерних пристроїв стрімко розвивається, і сучасний смартфон має оперативну пам'ять, обсяг якої можна порівняти з обсягом, що іще відносно недавно був нормою для ПК чи ноутбука. Для більшості цих пристроїв дуже важливе мережне з'єднання (передусім бездротове), оскільки вони часто суттєво залежать від хмарних сервісів.

Важливі акценти для ОС мобільних пристроїв:

- робота з ресурсами *порівняно* невеликого обсягу;
- сенсорний інтерфейс (здебільшого).

Рис. 2.3. Різноманітність мобільних пристроїв та ОС для них

Серед ОС для мобільних пристроїв згадаємо Android, iOS, Harmony OS, Windows Mobile[13].

Вбудовані операційні системи (embedded operating systems) та **операційні системи реального часу** (real-time operating systems) мають багато спільного, тому розглянемо їх разом. Обидва класи ОС працюють на дуже різноманітних пристроях (побутова техніка, DVD з можливістю запису, військова техніка,

[13]ОС Windows Mobile (раніше — Windows Phone) не підтримується з 2020 року. Втім, вихід Windows на різноманітні платформи вплинув на будову сучасних версій Windows. Тому згадати про цю ОС варто.

системи космічних кораблів, конвеєри на виробництві, екрани для демонстрації рекламних відеороликів та багато інших). На відміну від ОС попередніх класів, ці системи навіть не завжди називають операційними. На таких пристроях можуть бути встановлені як ОС з досить відомими назвами, так і системи, розроблені спеціально під конкретний пристрій, і лише під нього. Однак усі подібні системи об'єднує принципова особливість: у них важливою характеристикою є надійність. Хоча таку вимогу традиційно висувають для всіх ОС, саме для вбудованих ОС та ОС реального часу надійність значно суттєвіша за інші характеристики, як то швидкодія, дружній до користувача інтерфейс тощо.

Втім, вбудовані ОС та ОС реального часу все ж часто розділяють на два окремі класи — передусім через ще одну важливу вимогу, цього разу притаманну передусім другому класу, ОС реального часу. Це виконання операцій у відведені часові проміжки (як під час управління конвеєром на виробництві, наприклад). Часові межі для виконання завдань у системах реального часу можуть бути жорсткіші або м'якші, але вони наявні.

Таким чином, важливий акцент, який об'єднує такі різноманітні вбудовані ОС, буде один:

- підвищені вимоги до надійності.

Натомість до акцентів ОС реального часу додається ще й часовий фактор:

- підвищені вимоги до надійності;
- операції мають виконуватися у задані проміжки часу.

Серед прикладів вбудованих ОС — QNX та VxWorks. Ті самі системи можна віднести і до ОС реального часу, разом з системою e-Cos.

У книзі [3] в окремий клас виносяться **операційні системи сенсорних вузлів** (sensor-node operating systems). Такі системи вирізняються тим, що працюють на пристроях, малих за розмірами і обсягом доступних ресурсів. Мережі сенсорних вузлів можуть використовуватися для охорони території, фіксування лісових пожеж, військової розвідки тощо).

Відповідно, акцентами для ОС сенсорних вузлів будуть:

- невеликий об'єм;
- економія ресурсів і, зокрема, енергії.

Прикладом ОС сенсорних вузлів є Tiny OS.

Операційні системи смарт-карт (smart card operating systems) працюють на чіпованих картках різноманітного призначення: картках для оплати проїзду в громадському транспорті, ідентифікації особи, банківських картках, SIM-картках у мобільних телефонах та ін. (рис. 2.4).

Так, SIM-картка у телефоні зберігає у пам'яті невелику кількість телефонних номерів та дозволяють здійснювати блокування чи розблокування залежно від введеного коду.

Зауважимо, що тут ідеться саме про картки з чіпом. Картки лише з магнітною смужкою, скажімо, просто містять дані, які можна зчитати за допомогою спеціального пристрою, але самі по собі жодними операціями керувати не можуть.

Акценти ОС смарт-карт:

- малий об'єм;
- можуть мати підтримку Java-аплетів.

Рис. 2.4. Приклади смарт-карт — чіпованих карток, які містять мінімальну ОС

Прикладами ОС смарт-карт є PayFlex, MicroPayFlex, JCOP (Java Card Operating Platform).

2. Процеси

У розділі 1 вже йшлося про багатозадачність в її сучасному розумінні: багато програм можуть бути запущені і виконуватися одночасно. Ці запущені програми називають процесами. Особливо багато таких процесів буває в інтерактивних ОС (тобто ОС, орієнтованих на активну взаємодію з користувачем). У таких системах користувач паралельно працює з багатьма прикладними програмами, і це не враховуючи системних процесів, потрібних для функціонування самої ОС. Звісно, деякі процеси можуть більшість часу просто очікувати, коли в них виникне потреба, а доти лише займати виділений їм обсяг пам’яті і на ЦП не виконуватися. Також один ЦП може мати декілька ядер, та й самих ЦП може бути декілька. Та все одно завдань, котрі можуть претендувати на виконання у деякий момент часу, часто буде більше, ніж ЦП / ядер ЦП. Тому так чи інакше все зводиться до того, що ЦП треба швидко перемикатися між виконанням різних процесів, аби створити ілюзію одночасного виконання цих процесів.

Багатозадачність (multitasking) — режим роботи, за якого декілька завдань одночасно перебувають в оперативній пам'яті та почергово виконуються на ЦП.

Така багатозадачність є псевдопаралелізмом. Справжній (апаратний) паралелізм має місце, коли ЦП і / або ядер ЦП більше, ніж 1. На практиці обидва види паралелізму поєднуються.

Тепер ми готові перейти до детальнішого розгляду процесів. У цьому посібнику спиратимемося на два означення процесу. Перше з них, по суті, вже було сформульоване на початку цього пункту:

Процес (process) — абстракція, що представляє програму під час виконання.

У такому разі програма є статичною (вона нічого не робить, доки її не запустили), а процес — динамічним. Сама по собі програма майже не використовує системних ресурсів, окрім місця на диску, де зберігаються її файли.

Споживачем системних ресурсів є саме процес. Це процес використовує час ЦП, регістри ЦП, оперативну пам’ять, дисковий простір, пристрої введення-виведення тощо.

Зупинімося на основних системних ресурсах, які використовує процес.

- **Час ЦП** (процесорний час, CPU time) - проміжок часу, відведений процесу для виконання на ЦП.
- **Регістри ЦП** (CPU registers) - пристрої надшвидкої пам'яті у ЦП, призначені для тимчасового зберігання команд, керуючої інформації, операндів, результатів.

Оперативна пам'ять (основна пам'ять, main memory) виділяється процесам через механізм адресного простору.

- **Адресний простір** (address space) — набір адрес, до яких процес має доступ.

Адресний простір зазвичай містить:

- скомпільований *код* програми;
- *дані* процесу:
  - глобальні змінні;
  - дані, розміщені у купі (heap) — ця пам'ять виділяється динамічно;
  - дані, розміщені у стеку (stack) — залучається під час викликів функцій.

Адресний простір також називають **захищеним адресним простором**, оскільки один процес за замовчуванням не має доступу до адресного простору іншого процесу. Коли є потреба в спільному доступу до даних, використовують міжпроцесову взаємодію (interprocess communication, IPC) та потоки (threads). Міжпроцесову взаємодію буде детальніше розглянуто у розділі 8. Потоки коротко розглянемо тут, а докладніше — у розділі 3.

У сучасних ОС процес складається з одного чи кількох *потоків*.

Потік (потік виконання, thread) — код програми, виконуваний на ЦП.

На перший погляд, це означення співпадає з означенням процесу, яке ми дали раніше (програма під час виконання). Але між *програмою* та *кодом* є суттєва різниця. У потоці зазвичай виконується окрема функція, яку так і називають — потоковою функцією. В окремі потоки виносять ті частини програми, які потрібно виконувати паралельно з іншими частинами тієї самої програми. Наприклад, як потік можна організувати фонове збереження документа у текстовому редакторі. При цьому такий потік повинен мати доступ до тих самих даних — до тексту та його форматування, — що й потік, відповідальний взаємодію з користувачем, який вводить текст. Якщо запрограмувати взаємодію з користувачем та фонове збереження як два процеси, то доведеться організовувати обмін даними між ними, бо за замовчуванням вони не матимуть доступ до адресних просторів одне одного. Це може повільніше працювати. Натомість усі потоки одного процесу використовують той самий адресний простір — адресний простір свого процесу. А отже, труднощів з передаванням даних між ними не буде.

Таким чином, можна дати ще одне означення процесу.

Процес (process) — один або декілька потоків виконання, а також сукупність пов’язаних з ними ресурсів.

Ієрархія процесів. Процес може створити інший процес. Тоді перший процес буде називатися *батьківський процесом* (parent process), а другий — *дочірнім процесом* (child process). У свою чергу, дочірній процес також може створювати процеси, для яких він уже вважатиметься батьківським. Таким чином утворюється дерево процесів, що може розгалужуватися новими й новими дочірніми процесами (рис. 2.5).

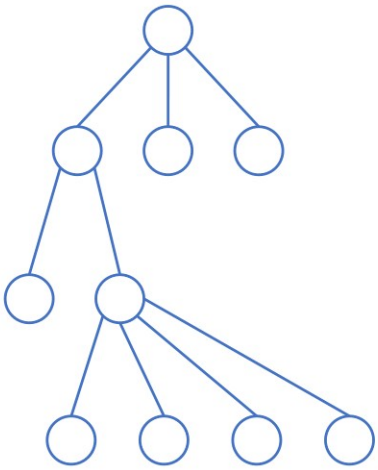

Рис. 2.5. Спрощена модель дерева процесів

Ієрархія процесів існує в усіх ОС, але в одних системах вона простежується чітко, а також суттєво впливає на особливості функціонування системи (Unix-подібні ОС), а в інших — значно менш помітна на практиці (Windows).

На рис. 2.6 показано фрагмент ієрархії процесів в дистрибутиві Linux Ubuntu. В Unix-подібних системах особлива роль відводиться процесу, який першим було створено під час завантаження ОС. Цей процес має ідентифікатор 1 і є батьківським для усіх інших процесів у системі. В різних системах ім'я цього процесу може не співпадати (init, systemd тощо). На рис. 2.6 це systemd.

```
olena@olena-HP-250-G5-Notebook-PC:~$ pstree
systemd─┬─ModemManager───2*[{ModemManager}]
        ├─NetworkManager───2*[{NetworkManager}]
        ├─accounts-daemon───2*[{accounts-daemon}]
        ├─acpid
        ├─avahi-daemon───avahi-daemon
        ├─bluetoothd
        ├─colord───2*[{colord}]
        ├─containerd───8*[{containerd}]
        ├─cron
        ├─cups-browsed───2*[{cups-browsed}]
        ├─cupsd
        ├─dbus-daemon
        ├─fwupd───4*[{fwupd}]
        ├─gdm3─┬─gdm-session-wor─┬─gdm-x-session─┬─Xorg───18*[{Xorg}]
        │      │                 │               ├─gnome-session-b───2*[{gnome-session-b}]
        │      │                 │               └─2*[{gdm-x-session}]
        │      │                 └─2*[{gdm-session-wor}]
        │      └─2*[{gdm3}]
        ├─gnome-keyring-d───3*[{gnome-keyring-d}]
        ├─irqbalance───{irqbalance}
        ├─2*[kerneloops]
        ├─networkd-dispat
        ├─nmbd
        ├─packagekitd───2*[{packagekitd}]
        ├─polkitd───2*[{polkitd}]
        ├─power-profiles-───2*[{power-profiles-}]
        ├─rsyslogd───3*[{rsyslogd}]
        ├─rtkit-daemon───2*[{rtkit-daemon}]
        ├─smbd─┬─cleanupd
        │      ├─samba-bgqd
        │      └─smbd-notifyd
        ├─snapd───11*[{snapd}]
        ├─switcheroo-cont───2*[{switcheroo-cont}]
```

Рис. 2.6. Приклад фрагменту дерева процесів з Ubuntu Linux (на базі systemd)

Стани процесів. Однією з важливих характеристик процесу є стан. Взагалі, стан може стосуватися і потоку — це залежить від конкретної реалізації. Заради зручності, надалі домовимося говорити про *стани процесів*, при цьому розуміючи, що може йтися і про *стани потоків*.

У теорії ОС розглядають *типові* стани процесів. У конкретних ОС кількість станів, їхні назви і суть відрізняються. Втім, аналогія з типовими станами завжди простежується. У цьому розділі ми розглянемо п'ятистанову модель процесів (five-state process model). Є й інші (двостанова, семистанова тощо).

Згідно з п'ятистановою моделлю процесів, процес може перебувати в одному з наступних станів:

- створення (new);
- виконання (running);
- готовність, або готовність до виконання (ready);
- очікування (waiting), або заблокований (blocked);
- завершення (terminated).

Основні переходи між станами показано на рис. 2.7.

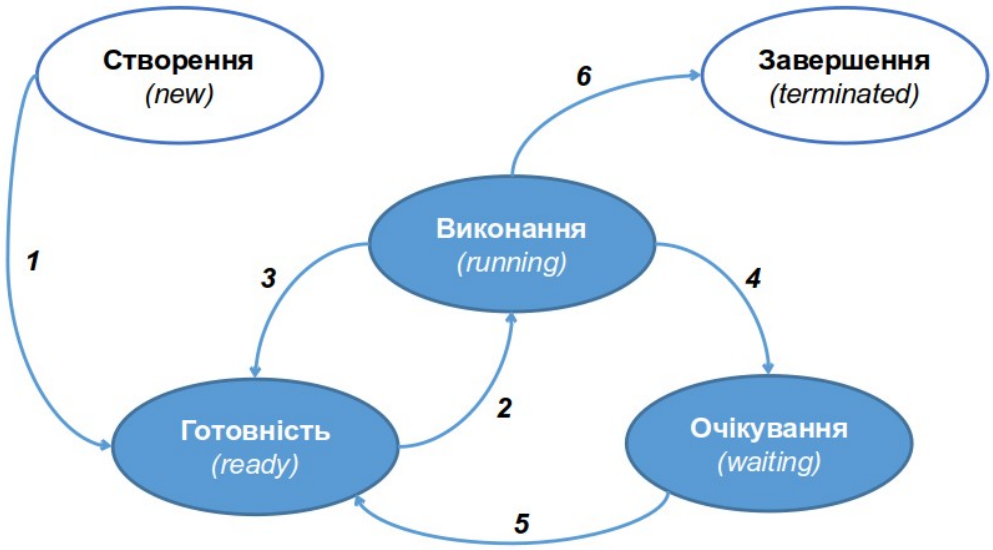

Рис. 2.7. Основні переходи між станами п'ятистанової моделі процесів

Рішення щодо переходу процесу з одного стану в інший приймає спеціальний компонент операційної системи — **планувальник** (scheduler). Переходи між станами відбуваються у наступних випадках.

Перехід (1) (**Створення → Готовність**) відбувається, коли процес було успішно створено, і тепер він чекає, доки його вибере планувальник.

Перехід (2) (**Готовність → Виконання**) здійснюється, коли планувальник вибирає саме цей процес, і процес виконується на ЦП.

Перехід (3) (**Виконання → Готовність**) відбувається, коли замість даного процесу планувальник вибирає інший процес, а стан даного процесу зберігається, щоб потім можна було продовжити його виконання (коли планувальник вибере цей процес знову).

Перехід (4) (**Виконання → Очікування**) застосовується, коли даний процес не може продовжити виконання, доки не відбудеться певна подія. Так, процес

може переходити в стан очікування, коли потрібний йому ресурс (наприклад, файл) наразі зайнятий. Процеси, орієнтовані на активну взаємодію з користувачем, переходять у стан очікування, коли очікують на введення користувачем даних.

Перехід (5) (**Очікування → Готовність**) здійснюється, коли подія, на яку чекав процес, відбулася (наприклад, потрібний файл вже доступний, користувач ввів потрібні дані тощо). Тоді процес знову претендує на виконання. Зверніть увагу: процес переходить у стан готовності, а не у стан виконання, оскільки рішення про продовження виконання процесу на ЦП приймає не сам процес, а планувальник.

Перехід (6) (**Виконання → Завершення**) відбувається, коли виконання процесу завершено. Втім, у багатьох системах можливий також перехід процесу у стан завершення й з інших станів. Тоді мають місце **перехід (7*)** (**Готовність → Завершення**) та **перехід (8*)** (**Очікування → Завершення**).

Важливий момент — припинення виконання процесу А і початок виконання процесу Б. Якщо процес А ще не завершено, то пізніше для продовження виконання йому знадобляться дані, які зберігаються у регістрах ЦП та у стеку. Якщо просто завантажити туди дані процесу Б, то ці дані процесу А буде втрачено (уявіть, що мусили б починати читати книгу чи виконувати лабораторну роботу з самого початку щоразу, як відволікалися б на телефонний дзвінок чи на повідомлення в месенджері). Щоб уникнути такої ситуації, існує перемикання контексту.

Перемикання контексту (context switching) — передача керування від одного процесу до іншого зі збереженням даних, пов'язаних з виконанням процесу на ЦП.

Під час перемикання контексту відбувається зберігання наступних даних:

- вмісту регістрів ЦП, зокрема лічильника команд (яку команду виконувати наступною);
- відомостей про пам'ять (наприклад, вказівника стеку).

Отже, коли процес наступного разу буде допущений до виконання на ЦП, його дані будуть наявні.

3. Режим користувача та режим ядра

Як уже згадувалося раніше, операційні системи можуть суттєво відрізнятися, зокрема й внутрішньою будовою. Втім, більшість із них мають принаймні два режими, у яких може працювати програма (процес), один привілейований, інший — ні. Перший зазвичай називають режимом ядра, другий — режимом користувача (рис. 2.8).

У **режимі ядра** (kernel mode, захищений режим, привілейований режим) процес може використати будь-яку інструкцію, яку здатне виконати дане обладнання. У режимі ядра працюють основні компоненти ОС (ядро), звідси й назва.

У **режимі користувача** (user mode) процес має доступ до обмеженої підмножини інструкцій даного комп'ютера. Більшість програм (тобто відповідних процесів) працюють у режимі користувача.

Отже, ми готові означити, що називатимемо ядром.

Ядром ОС називається сукупність базових компонентів ОС, які:
виконують її найважливіші функції,
працюють у привілейованому режимі
і зазвичай постійно перебувають у пам'яті.

Рис. 2.8. Узагальнене подання архітектури ОС

Наведене означення подекуди виглядає досить розмито. Скажімо, які саме функції вважати найважливішими? Що означає "*зазвичай* постійно перебувають у пам'яті"? Тобто можуть все ж перебувати в пам'яті не постійно?

Оскільки ОС відрізняються одна від одної, то і погляди на те, яким конкретно має бути ядро, та які компоненти ОС мають працювати в режимі ядра, а які ні. Поки важливо зрозуміти, що деякі компоненти ОС можуть працювати у режимі користувача (не входити до складу ядра).

Щоб краще розібратися, якими бувають ядра, спершу потрібно з'ясувати поняття системного виклику.

4. Системні виклики

Оскільки саме режим ядра передбачає виконання будь-яких доступних інструкцій, то виконання цих інструкцій можна розглядати не лише як право, а і як покладені на цей режим обов'язки. Адже процесам з режиму користувача також потрібно здійснювати операції, доступні лише в режимі ядра, причому дуже часто. У такому разі застосовується механізм системних викликів.

Коли програмі потрібно виконати дію, реалізовану у ядрі, вона використовує **системний виклик** (system call).

Системні виклики найчастіше написані мовами C, C++, Assembler. За допомогою системних викликів відбувається багато операцій, необхідних для роботи програм. Так, системні виклики використовуються для роботи з файлами (відкриття, закриття, читання, запис, створення, вилучення, читання/запис атрибутів тощо), для роботи з пристроями (приклади: вивести щось на екран, зчитати введену користувачем команду, ...), для роботи з процесами (створення, завершення, виділення/вивільнення пам'яті, читання/запис атрибутів тощо), обміну повідомленнями всередині системи і т. д. Отже, кількість виконаних системних викликів під час роботи ОС є значною.

З логічної точки зору робота системного виклику подібна до роботи звичайної підпрограми (наприклад, функції). Відбувається виклик функції і передавання необхідних параметрів, виконання тіла функції, потім — повернення результатів функції в точку виклику. Принципова відмінність системного виклику від виклику звичайної функції полягає у тому, що під час системного виклику здійснюється перемикання з режиму користувача (у ньому працювала прикладна програма) у режим ядра (у ньому працює код, який відповідає системному виклику), а потім знову у режим користувача (назад у прикладну програму).

Однак для взаємодії прикладної програми з операційною системою потрібний інтерфейс. Роль такого інтерфейсу виконує API операційної системи.

API. Загалом, API може стосуватися не лише операційної системи. Абревіатура API розшифровується як Application Programming Interface (інтерфейс прикладного програмування) і може бути як в ОС, так і в Інтернет-сервісів, баз даних тощо. Надалі, використовуючи термін API, матимемо на увазі API операційної системи.

API надає прикладному програмісту функції, типи та структури даних, константи тощо для організації взаємодії прикладної програми з ОС.

Приклади API для ОС: Windows API (Win32 API, Win64 API та деякі інші), POSIX API (у Unix-подібних ОС), Java API.

У багатьох випадках виклику API відповідає системний виклик. Але загалом бувають API-виклики, яким не відповідають системні виклики, і які повністю працюють у режимі користувача. Зокрема, інтерфейс Win32 API налічує тисячі викликів, і за частиною з них не стоять системні виклики [3, с. 61].

POSIX. API для Unix-подібних систем пов’язані передусім з POSIX (Portable Operating System Interface) — сімейством стандартів для забезпечення сумісності між різними ОС. Розробкою POSIX займається організація IEEE (Institute of Electrical and Electronics Engineers). POSIX містить вимоги щодо API, командних оболонок, інтерфейсів базових утиліт. Поточна версія — POSIX.1-2017.

Загалом POSIX асоціюється з Unix-подібними ОС. Однак на практиці описані у POSIX стандарти можуть стосуватися різноманітних систем (навіть не Unix-подібних), але різною мірою.

Окремі ОС належать до **POSIX-сертифікованих**. Станом на лютий 2023 р. це macOS, VxWorks, Integrity, z/OS та деякі інші. Належність ОС до POSIX-сертифікованих визначається за допомогою спеціальних автоматичних сертифікаційних тестів і може змінюватися, залежно від змін у наступних версіях ОС.

До **загалом POSIX-сумісних** належать Linux, Android, MINIX, OpenSolaris, FreeBSD, Syllable, NetBSD, OpenBSD та багато інших.

Деякі ОС забезпечують **часткову POSIX-сумісність** через додаткові механізми та середовища: Windows, eCos, Symbian, Plan 9, OpenVMS та ін. Це

здійснюється за допомогою засобів віртуалізації чи інших підсистем та надбудов.

У таблиці 2.1 наведено приклади деяких API-викликів у Unix-подібних ОС та у Windows. У випадку Windows вказано імена API-функцій, у випадку Unix — імена бібліотечних функцій.

Врахуйте, що у більшості наведених функцій є параметри, у таблиці їх пропущено. Крім того, деякі наведені функції можуть виконувати не лише ту дію, яку вказано у таблиці (так, `CloseHandle()` закриває не лише файл, а й будь-який інший дескриптор).

Одразу впадає в око різний підхід до написання імен. Водночас, важливо й те, що наведені для Windows назви є іменами API-функцій, назви ж відповідних системних викликів цілком відрізняються. Тим часом для Unix наведено імена системних викликів — вони часто співпадають або майже співпадають з назвами відповідних бібліотечних функцій.

Таблиця 2.1. *Відповідність деяких викликів Windows та викликів Unix*

Що робить виклик	Windows	Unix
Створює процес	`CreateProcess()`	`fork()`
Завершує процес	`ExitProcess()`	`exit()`
Чекає на завершення дочірнього процесу	`WaitForSingleObject()`	`wait()`
Створює файл	`CreateFile()`	`open()`
Зчитує з файлу	`ReadFile()`	`read()`
Записує у файл	`WriteFile()`	`write()`
Закриває файл	`CloseHandle()`	`close()`

Важливо розуміти, що якщо системні виклики здійснюються часто, то перемикання між режимами ядра і користувача також відбуватиметься часто. А ми вже з'ясували, що воно може бути досить інтенсивним.

Це підводить нас до наступної частини даної теми — архітектури ядра ОС.

5. Класифікація ОС за архітектурою

Загалом, питання щодо того, які компоненти ОС поміщати в ядро, а які мають лишатися працювати в режимі користувача, розробники різни ОС вирішують по-різному. Це залежить від пристроїв, на яких має працювати ОС, від поглядів головних розробників на те, що вважати оптимальною архітектурою ядра, від прикладів ОС, що беруться за основу, та ін..

Поділ ОС залежно від архітектури ядра є умовним, проте кожна з наявних систем тяжіє до певного класу.

1. ОС на основі монолітного ядра. Це найпоширеніший спосіб реалізації ОС, за якого усі базові функції ОС реалізуються в ядрі. Це зазвичай сприяє

продуктивності системи, адже системні виклики у такій ОС використовуються лише для роботи прикладних програм, ядро ж і так має доступ до усіх інструкцій наявного обладнання. Водночас, класичне монолітне ядро може мати проблеми з надійністю, адже помилка у компоненті, запущеному у режимі ядра, може призводити до значно серйозніших наслідків, ніж коли б цей компонент працював в режимі користувача.

Втім, переважна більшість ОС належать до монолітних або тяжіють до них.

2. ОС на основі мікроядра в чистому вигляді зустрічаються рідше. За такого підходу в ядро поміщають мінімальний набір функцій, а решта компонентів працює у режимі користувача. Це дозволяє підвищити надійність такої системи, але шляхом деяких втрат у продуктивності.

Прикладами ОС з мікроядерною архітектурою є Mach, MINIX 3, Symbian, QNX, Integrity, K42, L4, Pike OS та ін..

На практиці спостерігається певне **взаємопроникнення монолітної та мікроядерної архітектур** як спроба використати сильні сторони обох. Так, сучасні монолітні ОС зазвичай мають модульну структуру, й окремі модулі можуть за потреби підвантажуватися пізніше, без повного перезавантаження системи. До *монолітних ОС з модульним ядром* відносять Linux, FreeBSD, Solaris, Open VMS. Мікроядро також не завжди містить настільки мінімальний набір компонентів, як в своїй класичній реалізації. Тоді маємо ОС на базі мікроядра з рисами монолітного ядра, або гібридні ОС. До *гібридних ОС* тяжіють сучасні версії Windows, а також Mac OS X, Syllable, DragonFly BSD. Скажімо, мікроядро у Windows — це досить велике мікроядро. Втім, нагадаємо, що такий поділ є умовним, і ви можете натрапити й на інші підходи до класифікації наявних систем, зокрема й на дискусії на цю тему.

Виокремлюють іще одну архітектуру, яку на даний момент можна назвати радше історичною. Втім, як ми вже бачили раніше у цьому посібнику, часом деякі технології минулого згодом реінкарнують у цікаві й впливові сучасні технології. Тому незайве подивитися і на багаторівневі ОС.

3. Багаторівневі ОС. Замість двох режимів (ядра і користувача) у таких ОС виокремлювалося багато рівнів. Наприклад, в ОС THE Едсгера Дейкстри (Edsger W. Dijkstra) таких рівнів було п’ять. Кожен наступний рівень мав менше привілеїв доступу і спирався на функції попереднього рівня (як режим ядра спирається на функції режиму користувача). Однак там, де у дворівневій архітектури під час системного виклику маємо перемикання між двома режимами (режим користувача — режим ядра — режим користувача), у багаторівневій таких перемикань могло бути значно більше (рис. 2.9).

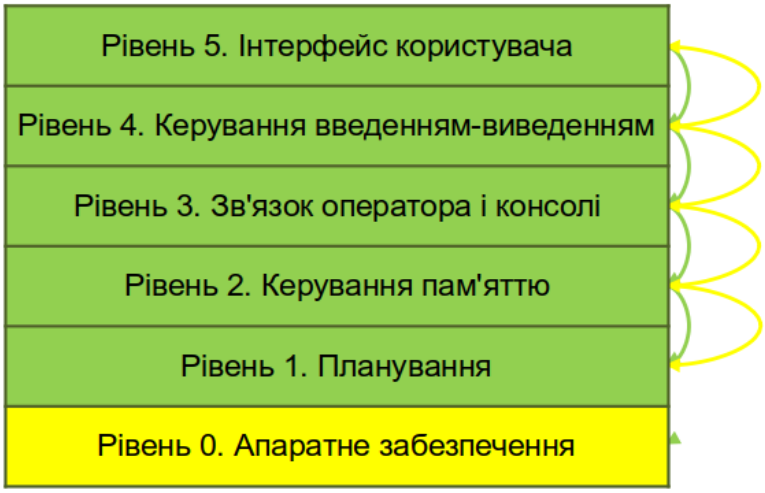

Рис. 2.9. Схематичне подання архітектури ОС THE

Попри очевидну з сучасного погляду втрату продуктивності у такому випадку, така архітектура все ж мала й переваги. Зокрема, такі системи було легше реалізовувати на практиці. Створюючи свій компонент ОС, програміст не мусив знати деталі роботи використовуваних компонентів нижнього рівня — досить було знати, як правильно до них звернутися.

Прикладами багаторівневих ОС були системи THE та MULTICS.

6. Блок керування процесом

Для того, аби керувати процесом, в ОС має бути інформація про цей процес: де процес розміщений (здебільшого в оперативній пам'яті, але частина може бути вивантажена на диск), а також які атрибути має цей процес (щоб відрізнити цей процес з-поміж інших процесів; щоб знати, що йому можна, а що ні; щоб знати, чи цей процес важливіший за інші; щоб стежити, скільки ресурсів споживає цей процес і т.д.).

У різних ОС ці відомості зберігаються по-різному. В теорії ОС зазвичай говорять про блок керування процесом (process control block, PCB).

Блок керування процесом (*process control block*, ***PCB***) — сукупність атрибутів процесу, потрібних ОС для керування цим процесом.

На практиці назви та організація таких сукупностей може відрізнятися. Наприклад, у Linux йдеться про структуру даних, а у Windows — про об'єкти [4]. Втім, атрибути будуть подібні. У таблиці 2.2 наведено найбільш типові з них [1].

Таблиця 2.2. *Типові атрибути блоку керування процесом*

Призначення атрибутів	Атрибути
Основні ідентифікатори	• ідентифікатор процесу (Process IDentifier — PID), унікальний у межах системи; • ідентифікатор батьківського процесу (parent PID, PPID)
Інформація стану процесу (для перемикання контексту)	• вміст регістрів ЦП, зокрема операнди, результат останньої команди, лічильник команд (яку команду виконувати наступною) тощо; • вказівники вершин стеків
Безпекові атрибути	• ідентифікатор користувача, від імені якого запущено процес; • ідентифікатор групи, від імені якої запущено процес; • інші безпекові атрибути

Відомості для планувальника	• стан процесу; • пріоритет (пріоритети); • вказівники на черги, у яких перебуває процес; • відомості про подію, на яку очікує процес.
Відомості для керування пам'яттю	залежать від системи керування пам'яттю в ОС (базовий і межовий регістри, таблиця сегментів, таблиця сторінок тощо)
Облік споживання процесом ресурсів	• використаний час ЦП і загальний час роботи процесу; • часові обмеження тощо
Відомості про введення-виведення	• перелік пристроїв введення-виведення, виділених процесу; • перелік відкритих файлів тощо

Разом програмний код процесу, дані процесу, стек процесу та PCB (атрибути процесу) в теорії ОС називають **образом процесу** (process image). Проте треба бути обережними із застосуванням цього терміну на практиці, оскільки іноді образом процесу позначають ім’я виконуваного файлу, запуск якого призвів до створення цього процесу.

7. Інтерактивні і фонові процеси. Створення та завершення процесів

Інтерактивні та фонові процеси. З огляду на характер взаємодії з користувачем, виокремлюють інтерактивні та фонові процеси.

Інтерактивні процеси (interactive processes, foreground processes) орієнтовані на безпосередню взаємодію з користувачем. Зазвичай користувач помічає передусім ці процеси. В GUI інтерактивні процеси можуть мати вікно, хоча інтерактивним вважатиметься і процес, що працює у командному рядку, проте очікує певних дій користувача й реагує на них. Наприклад, текстові редактори *nano* чи *vim* у Linux працюють у командному рядку, проте належать до інтерактивних.

Фонові процеси (backdround processes) не мають безпосередньої взаємодії з користувачем. Якщо таку взаємодію й можна простежити, то опосередковано. Фоновим може бути й інтерактивний процес, запущений як фоновий або переведений у фоновий режим пізніше (наприклад, якщо *nano* запустити саме як фоновий процес). Та частіше фоновими є процеси, які виконують службову роль у системі, зокрема й ті, які є її частиною. Такі процеси реагують на запити інших процесів, решту ж часу вони можуть просто перебувати у стані очікування.

У Windows фонові процеси називаються службами (*services*), а у Unix-подібних ОС й зокрема у Linux — англомовним терміном *daemons*, який часто перекладають як “демони”. Варто зазначити, що цей переклад не є цілком коректним. В англійській мові є два схожих слова — *demons* (демони, значення має дещо негативний відтінок) та *daemons* (духи, значення має більш

нейтральний відтінок, бо в міфології та релігіях духи не обов'язково є добрими чи злими, вони часто уособлюють сили, які керують світом, а також можуть допомагати людям — за певних обставин). Друге значення справді пасує фоновим процесам, тож фонові процеси у Linux мали б перекладатися як "духи". Втім, термін "демони" вже суттєво закріпився у наявних перекладах, тож, аби зберегти чіткість викладу, надалі у цьому посібнику ми називатимемо такі процеси у Linux словосполученням "фонові процеси".

Створення та завершення процесів. Створення процесу може відбуватися під час ініціалізації системи (наприклад, у багатьох сучасних дистрибутивах Linux перший створюється процес *systemd*, і йому присвоюється ідентифікатор 1). Та переважна більшість процесів створюються під час виконання інших процесів.

Завершення процесу залежить від завдань, поставлених перед відповідною програмою, та від інших обставин. Так, процес може завершуватися сам, коли він виконав свою роботу до кінця (наприклад, команда ping здійснила чотири заплановані тестові обміни пакетами з віддаленим хостом, після чого вивела на екран результати і завершила свою роботу). Але процес може бути завершується ззовні — наприклад, коли процес більше не потрібний (відповідне завдання скасоване або вся система завершує роботу), коли процес перевикористав дозволений йому обсяг ресурсів. Буває, що має місце так зване каскадне завершення (cascading termination): коли завершено батьківський процес, його дочірні процеси також мають бути завершені.

Цікавий стан процесу пов'язаний із завершенням процесів у Unix/Linux — це **стан зомбі**. Такий процес фактично завершений, але його блок керування ще не вилучено. Ці дані треба спершу передати батьківському процесу, після чого батьківський процес здійснює остаточне завершення дочірнього процесу у стані зомбі за допомогою системного виклику `waitpid()`. Загалом, даний системний виклик також може переводити процес у стан очікування, однак у випадку з зомбі-процесом він видаляє з системи структуру даних цього процесу.

8. Переривання

ЦП має коректно реагувати на події, які відбуваються з апаратним забезпеченням і програмними компонентами. Якщо читання або запис даних на диск завершено, якщо сталася помилка доступу до пам'яті, якщо користувач натиснув клавішу на клавіатурі, якщо під'єднано ще один пристрій, якщо виконуваний процес має зачекати на введення-виведення — це все події, які можуть потребувати реакції на них. Переривання використовуються операційною системою для багатозадачності, планування, диспетчеризації, керування пам'яттю, а також для синхронізації процесів.

Щоб ЦП міг реагувати на подібні події, потрібно, щоб, по-перше, ЦП якось дізнавався про ці події, і по-друге, існували спеціальні процедури, котрі ЦП мав би виконати, якщо подія настала.

Для таких потреб використовується механізм **переривань** (*interrupts*). Термін відповідає основній суті механізму: ЦП має тимчасово *перервати* опрацювання поточного завдання, щоб відреагувати на певну подію. Реалізація переривань у різних архітектурах може відрізнятися, але загалом є стандартні переривання й стандартне призначення. Реакція ЦП на переривання описується у вигляді згаданих вище спеціальних процедур у коді ОС — **IRC** (interrupt service routines).

Спрощену часову діаграму обробки переривань показано на рис. 2.10.

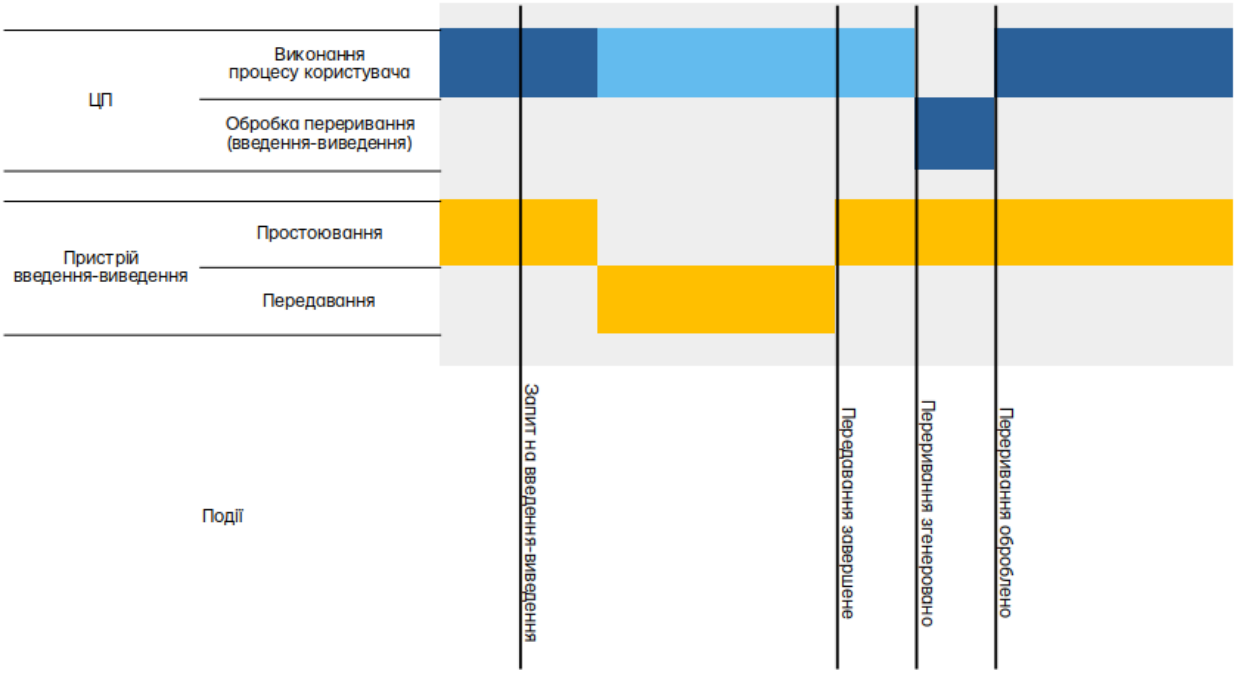

Рис. 2.10. Часова діаграма переривань для випадку одного процесу, який здійснює введення-виведення даних

На рис. 2.10 спершу користувацький процес виконується на ЦП, а пристрій введення-виведення простоює. Коли процесу потрібно зробити введення-виведення, здійснюється відповідне переривання, а пристрій введення-виведення переходить зі стану простоювання у стан передавання. Процес у цей час буде перебувати у стані очікування (заштрихована область на діаграмі). Якщо система багатозадачна, то у цей час можливість виконуватися на ЦП може бути надано іншому готовому до виконання процесу[14]. Коли введення-виведення завершиться, пристрій введення знову переходить у стан простоювання, а процес користувача продовжує виконання.

Зазвичай виокремлюють *немасковані переривання* (nonmaskable interrupts) та *масковані переривання* (maskable interrupts). Немасковані переривання стосуються найважливіших подій (наприклад, критичних помилок під час роботи з пам'яттю). Натомість масковані переривання використовуються для звичайних подій (наприклад, запитів від контролерів пристроїв). Коли ЦП виконує критичну послідовність інструкцій, масковані переривання можуть тимчасово блокуватися (masked), звідси й назва.

Список адрес, за якими зберігаються обробники переривань, називається вектором переривань (interrupt vector). Приклад вектора переривань для процесорів Intel наведено у [1, с. 11].

Контрольні запитання

1) Які класи ОС залежно від сфери застосування (від пристроїв, на яких працює ОС) вам відомі? Коротко охарактеризуйте кожний з них. Наведіть приклади ОС.

[14] Роботу з іншими процесами не відображено на діаграмі, оскільки це б суттєво її ускладнило. Роботу з кількома процесами ще буде розглянуто у темі "Планування та диспетчеризація".

2) Що таке багатозадачність? В чому полягає різниця між справжнім паралелізмом та псевдопаралелізмом?
3) Дайте означення процесу (як однієї з абстракцій, які надає ОС).
4) Які основні системні ресурси використовує процес?
5) Що містить адресний простір процесу? Чому адресний простір процесу називають захищеним адресним простором?
6) Дайте означення потоку (потоку виконання). Дайте означення процесу (через поняття потоку).
7) Поясніть концепцію ієрархії процесів. Які процеси називають батьківським і дочірнім?
8) Охарактеризуйте п'ятистанову модель процесів. Щодо кожного стану дайте відповідь на питання: що означає перебування процесу у цьому стані? у який стан (стани) може здійснюватися перехід із даного стану?
9) Що таке перемикання контексту? Які дані щодо виконання процесу важливо зберегти під час перемикання контексту?
10) Поясніть ключові відмінності між режимом ядра та режимом користувача.
11) Дайте означення ядра ОС.
12) Поясніть призначення системних викликів та охарактеризуйте загальний принцип їхньої роботи.
13) Охарактеризуйте роль API в операційній системі. Порівняйте Win32 API та POSIX API. Наведіть приклади викликів з обох інтерфейсів.
14) Що означає абревіатура POSIX? Яка версія POSIX використовується на даним момент? Яким чином POSIX враховується у реальних ОС? (сертифіковані POSIX-системи, POSIX-сумісність)
15) Які класи ОС виокремлюють залежно від архітектури ядра? Коротко охарактеризуйте кожний з них. Наведіть приклади ОС.
16) Що таке блок керування процесом? Перерахуйте основні типові атрибути, які може містити блок керування процесом.
17) Поясніть відмінності між інтерактивними і фоновими процесами. За яких обставин інтерактивний процес може бути фоновим? Які традиційні назви фонові процеси мають у Windows-системах та Linux-системах?
18) За яких обставин може створюватися та завершуватися процес? У якому випадку в ОС Linux процес переходить у стан зомбі? Що відбувається з таким процесом далі?
19) Навіщо в ОС потрібні переривання? Коротко опишіть роль переривань у введенні-виведенні даних для процесу.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. - Chapter 1 (1.2.1), Chapter 3 (3.1).
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. - Chapters 3-4.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. Chapter 2 (2.1), Chapter 1 (1.4).
4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. - Розділи 2-3.

Розділ 3

Багатопотоковість

1. Потоки: призначення, переваги, виклики

У розділі 3 було розглянуто процеси та потоки. *Потоком* ми назвали код програми, виконуваний на ЦП, а *процесом* — один або декілька потоків та сукупність пов'язаних з ними ресурсів.

У сучасних ОС процес складається з одного чи кількох потоків. Тобто сучасні ОС зазвичай підтримують **багатопотоковість**. На рис. 3.1 показано приклади співвідношення кількості процесів та потоків у багатопотоковій системі.

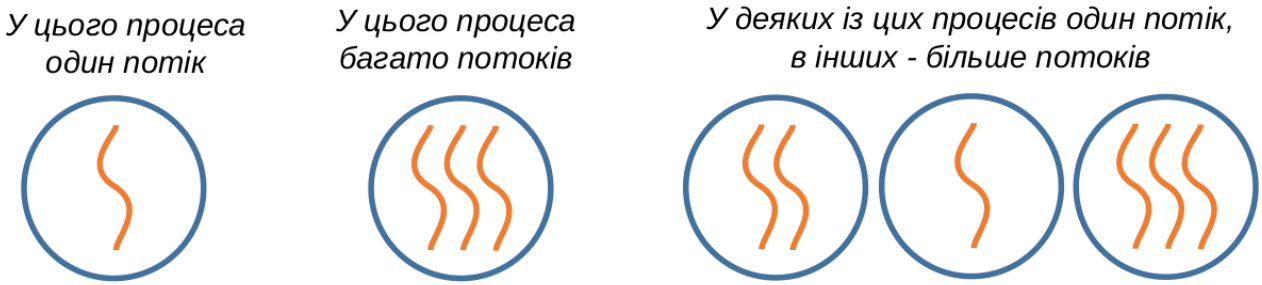

Рис. 3.1. Схематичне подання одного процесу з одним потоком (зліва), одного процесу з багатьма потоками (по центру) та трьох процесів з різною кількістю потоків (справа)

Найчастіше з потоком пов'язане те, що стосується безпосередньо виконання на ЦП: *значення регістрів ЦП*, *лічильник команд* (program counter, зберігається в окремому регістрі ЦП) та *стек* (використовується під час виклику функції). Натомість, усе, що стосується всіх потоків того самого процесу, зазвичай пов'язане з процесом: *програмний код*, *дані* (окрім тих, що у стеку, та тих, що у регістрах ЦП) та *інші ресурси* (сигнали, відкриті файли тощо).

Важливо розуміти, що багатопотоковість потребує значних організаційних зусиль на рівні операційної системи, бібліотек, окремих багатопотокових застосунків. Втім, за умови правильної реалізації багатопотоковість може забезпечити низку ***переваг***, зокрема наступних.

- **Легше і швидше спільне використання ресурсів.** У потоків одного процесу адресний простір спільний, тому для обміну даними між потоками не потрібно використовувати додаткових механізмів.
- **Оперативність реакції програм** (app responsiveness). Якщо зробити окремий потік, відповідальний за взаємодію з користувачем, то користувач помічатиме менше затримок у роботі застосунку, і загалом його користувацький досвід покращиться.
- **Економія часу і пам'яті.** Створення нового потоку відбувається швидше і потребує менше пам'яті, ніж створення нового процесу. Перемикання контексту між потоками теж здійснюється швидше, принаймні потенційно.
- **Масштабованість.** Багатопотокова програма може адаптуватися під різні апаратні можливості. Так, на більшій кількості ядер доцільно створювати більшу кількість потоків (вони зможуть виконуватися справді паралельно), а на меншій кількості ядер — кількість потоків краще обмежити (усі вони зможуть виконуватися одночасно лише за рахунок псевдопаралелізму, і продуктивність може не лише не зрости, а й знизитися).

Пригадаймо: терміни "багатозадачність", "паралелізм", "псевдопаралелізм" вже використовувалися у розділі 2. Уточнимо їх, враховуючи те, що дізналися про багатопотоковість.

Багатозадачність (multitasking, concurrency) можна реалізувати або *лише як псевдопаралелізм* (коли апаратного паралелізму немає взагалі, наприклад, на одному ЦП з одним ядром), або *з використанням апаратного паралелізму* (тобто і апаратного паралелізму, і псевдопаралелізму).

Термінологічна колізія тут полягає у тому, що *паралелізм* (*parallelism*) без префікса "псевдо" одними авторами тлумачиться як справжній, апаратний, паралелізм (як на багатоядерних процесорах та в багатопроцесорних системах), іншими ж авторами — як будь-який паралелізм, в тому числі й псевдопаралелізм. Тому використовувати термін "паралелізм" треба, розуміючи, ці нюанси.

Окремо підкреслимо: багатопотоковість розкриває свій потенціал, коли є не лише псевдопаралелізм, а й апаратний паралелізм.

Та наявність відповідного обладнання — не єдина передумова розкриття переваг багатопотоковості. *Багатопотоковість ставить* ***виклики*** і перед розробниками ОС, і перед прикладними програмістами. Розробники ОС мають врахувати багатопотоковість у плануванні виконання потоків на ЦП. Прикладні програмісти, які розробляють багатопотокові застосунки, також зіштовхуються з низкою проблем, зокрема наступними:

- як виокремити всередині програми паралельні завдання? (кожне завдання має бути варте виокремлення)
- як узгодити роботу паралельних завдань зі спільними даними? (завдання не повинні заважати одне одному)
- як налагодити та протестувати роботу багатопотокової програми? (послідовність виконання потоків може відрізнятися під час кожного запуску)

Планування процесів та потоків буде детальніше розглянуто у розділі 4, а проблеми узгодження роботи потоків у багатопотоковому застосунку — у розділі 8.

2. Основні моделі та стратегії багатопотоковості

Існують різні підходи до виокремлення моделей багатопотоковості. У даному посібнику розглянемо два — *моделі процесів та потоків* (залежно від їхньої кількості) та власне *моделі багатопотоковості*. Друга використовується у теорії ОС частіше, натомість перша дає загальне уявлення про те, як системи, що могли виконувати задачі лише послідовно одна за одною, трансформувалися у сучасні багатопотокові системи.

Моделі процесів і потоків (залежно від їхньої кількості) схематично подано на рис. 3.2.

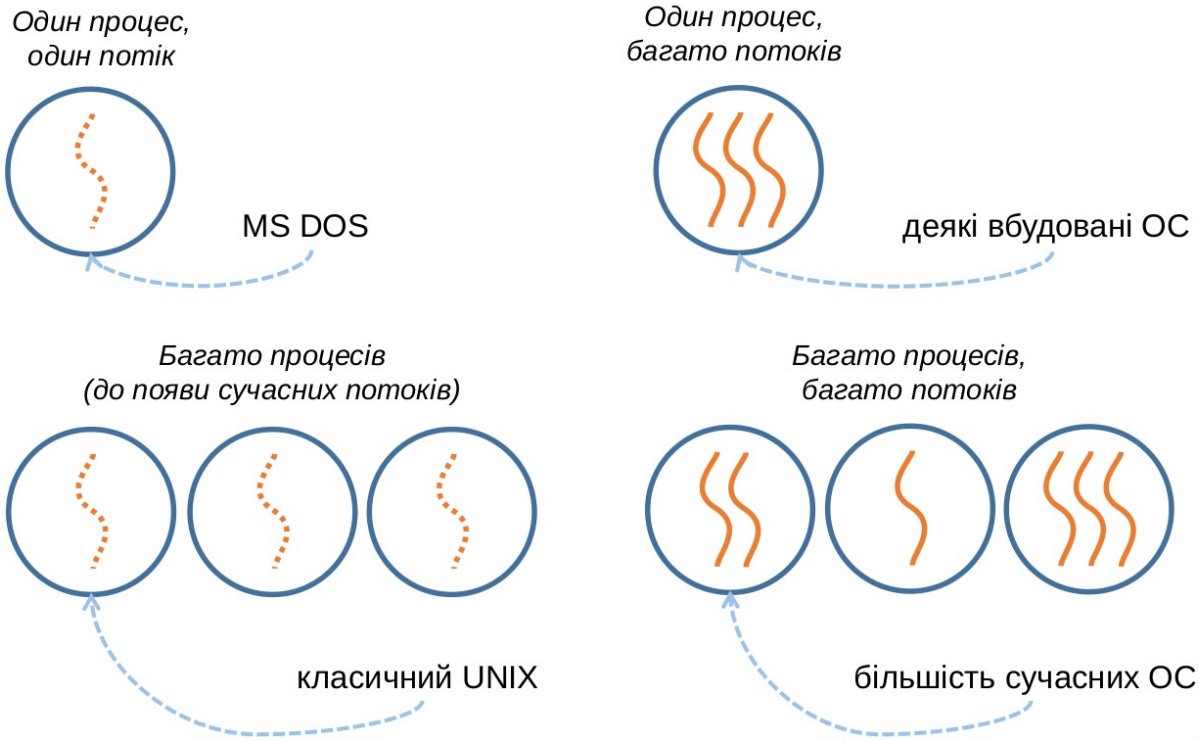

Рис. 3.2. Схематичне подання моделей процесів і потоків (залежно від їхньої кількості)

У старих однозадачних системах на зразок MS DOS фактично працював лише один процес. Про потоки тут фактично не йдеться, але можна вважати, що цей процес має єдиний потік.

Дещо екзотичний варіант можна зустріти у деяких вбудованих системах: багато потоків працюють у межах єдиного процесу. Це спрощує обмін даними між потоками.

У класичних Unix-системах могло паралельно працювати багато процесів, але багатопотоковості ще не було. Знову ж таки, як і у випадку з MS DOS, можна виокремити по одному потоку всередині кожного процесу, але лише умовно.

Натомість у більшості сучасних ОС підтримується одночасне існування багатьох процесів, а в межах кожного з них — одного чи кількох потоків.

Основні моделі багатопотоковості включають модель “багато до одного” (many-to-one), модель “один до одного” (one-to-one), модель “багато до багатьох” (many-to-many) та ін.

Модель “багато до одного” (many-to-one) передбачає реалізацію потоків у просторі користувача. На рис. 3.3 чотирьом потокам у просторі користувача відповідає один потік у просторі ядра. У такому разі ядро нічого не знає про потоки — воно фактично працює з однопотоковими процесами. Недолік такої моделі очевидний: коли котрийсь із потоків процесу виконується, решта не може.

Прикладами моделі “багато до одного” є рання ОС Solaris та рання реалізація багатопотоковості на Java.

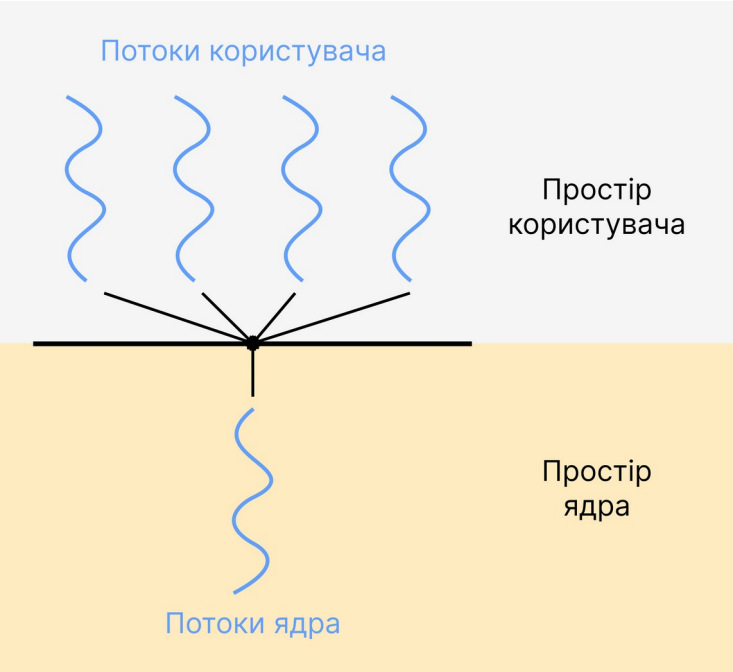

Рис. 3.3. Модель багатопотоковості “багато до одного” (many-to-one)[15]

Модель “один до одного” (one-to-one) кожному потоку у просторі користувача ставить у відповідність один потік у просторі ядра (рис. 3.4). Тоді, якщо процесор багатоядерний, то потоки одного процесу зможуть виконуватися паралельно. Втім, і в такої моделі є недолік: якщо потоків користувача (а отже, і потоків ядра) стане забагато — це може не лише не дати додаткового виграшу у продуктивності, а й знизити її.

[15] Рис. 3.3, рис. 3.4, рис. 3.5 та рис. 3.6 адаптована за ілюстраціями з підручника [1, с. 166-168].

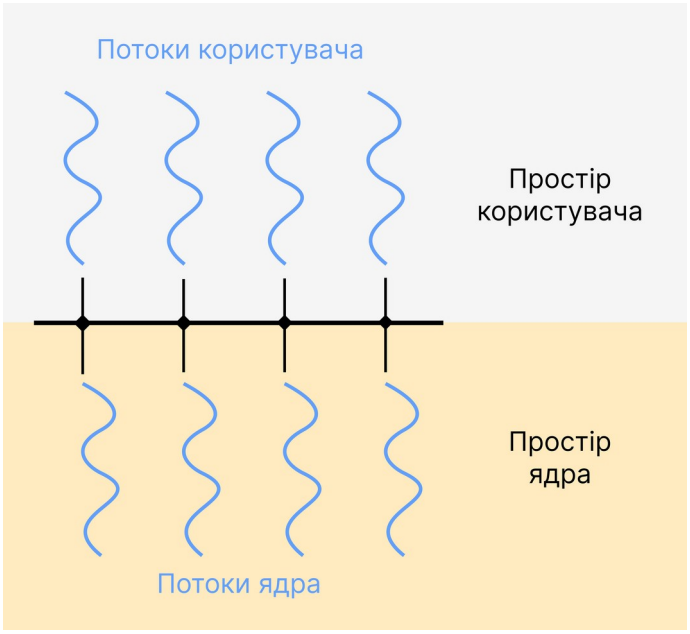

Рис. 3.4. Модель багатопотоковості “один до одного” (one-to-one)

Модель “один до одного” реалізовано у більшості сучасних ОС, зокрема й сучасних версія Windows та Linux.

Модель “багато до багатьох” (many-to-many) передбачає наявність окремих потоків у просторі користувача й окремих потоків у просторі ядра (рис. 3.5). При цьому кількість потоків простору ядра має бути меншою або рівною кількості потоків простору користувача. Це дозволяє обмежити кількість потоків, порівняно з моделлю “один до одного”. Водночас, описану модель складно втілити на практиці.

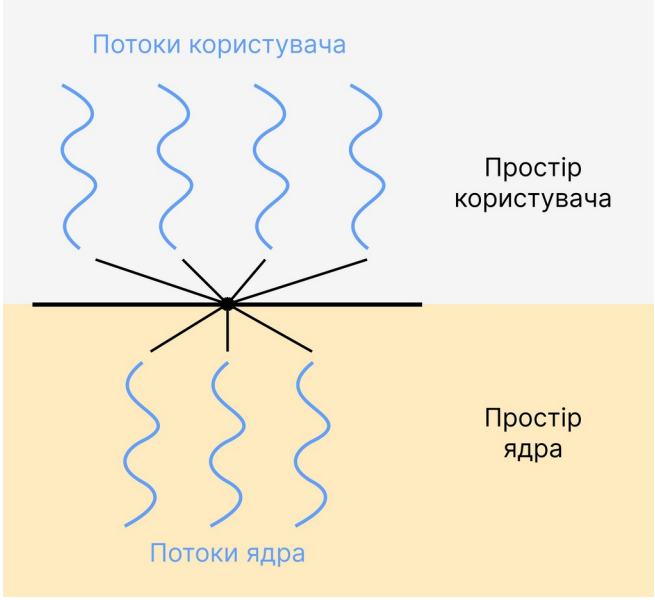

Рис. 3.5. Модель багатозадачності “багато до багатьох” (many-to-many)

Прикладами моделі “багато до багатьох” є реалізації на рівні бібліотек.

Серед інших моделей багатопотоковості на окрему увагу заслуговує ***дворівнева модель (two-level model)***. За дворівневої моделі багатопотоковості частина потоків організовується так, як у моделі “багато до багатьох”, а частина — яку у моделі “один до одного” (рис. 3.6).

Реалізацію дворівневої моделі можна зустріти, наприклад, в ОС HP-UX.

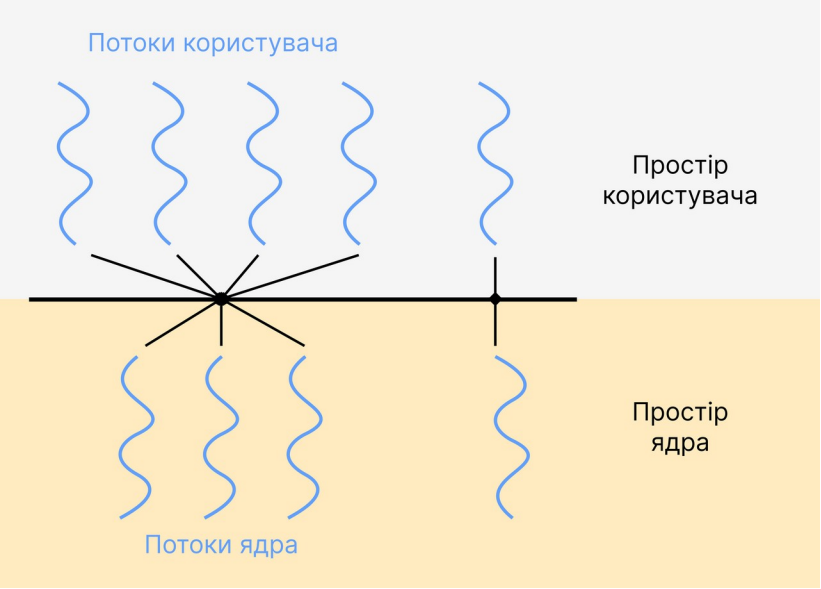

Рис. 3.6. Дворівнева модель багатопотоковості (two-level model)

Основні стратегії багатопотоковості. Залежно від поділу завдань, у багатопотоковому застосунку може бути використано стратегію багатопотоковості завдань та стратегію багатопотоковості даних. У разі ***багатопотоковості завдань*** здійснюється поділ даних між ядрами. При цьому над кожним набором даних виконуються ті самі операції, просто це роблять різні потоки. ***Багатопотоковість даних*** передбачає, що кожний потік виконує різні операції. На практиці обидві стратегії можуть поєднуватися.

Також стратегії багатопотоковості різняться залежно від синхронності виконання. Після створення батьківським потоком дочірнього потоку батьківський потік може призупинитися й зачекати на завершення дочірніх потоків — тоді йтиметься про ***синхронну багатопотоковість*** (synchronious threading). Подібна стратегія застосовується, наприклад, у паралельних обчисленнях. Натомість ***асинхронна багатопотоковість*** (asynchronious threading) передбачає, що батьківський процес також продовжує виконання паралельно з дочірніми. Така стратегія доцільна, наприклад, для користувацьких інтерфейсів з оперативною реакцією (responsive apps).

3. Бібліотеки для роботи з потоками

Бібліотеки для роботи з потоками надають API для створення потоків та маніпуляцій з ними. Потоки може бути реалізовано як на рівні ядра, так і на рівні користувача. Коли потоки реалізовано на рівні ядра, то під час роботи з ними залучаються системні виклики. Коли ж реалізація потоків виконана на рівні користувача, то потоки створюються без системних викликів.

Наприклад, у бібліотеці POSIX Pthreads потоки реалізовано на рівні користувача або на рівні ядра, у бібліотеці Windows Threads — на рівні ядра, а у

випадку Java Threads рівень, на якому працюють потоки, залежить від реалізації потоків в основній ОС.

Розглянемо приклади простої багатопотокової програми — спершу написаної для Linux (із використанням бібліотеки Pthreads), а тоді — для Windows (на базі бібліотеки Windows Threads).

Суть програми однакова в обох випадках. Зчитується значення аргументу *x*, після чого обчислюється значення функцій *F(x)* та *G(x)*, причому кожна функція працює в окремому потоці. Коли результати обох функцій будуть готові, обраховується сума одержаних значень — і виводиться на екран.

Описана програма переважно використовує стратегію синхронної багатопотоковості: після створення двох потоків (для обчислення функції *F* та функції *G*), головна функція `main()` мусить дочекатися завершення виконання цих потоків, оскільки для обчислення суми їй знадобляться їхні результати.

У поданих далі прикладах використано дуже прості потокові функції:

$$F(x) = x^2$$

$$G(x) = x^3$$

У реальній програмі, звісно, було б доцільно винести в окремі потоки складніші перетворення.

У лістингу 3.1 показано програму для Linux.

Лістинг 3.1. Проста багатопотокова програма на основі Pthreads

```
#include <iostream>
#include <pthread.h>
#include <math.h>
#include <stdlib.h>

using namespace std;

struct DATA_
{
  double x;
};

typedef struct DATA_ DATA;
double res1, res2;

void * F (void * arg);
void * G (void * arg);

int main()
{
  pthread_t threads[2]; // масив вказівників на потоки
  DATA arg;
  cout<<"Введіть значення аргументу x: "; cin>>arg.x;
  int er;
  er = pthread_create(&threads[0], NULL, F, (void *) & arg);
```

```
    if (er > 0)
    {
      cout<<"Не вдалося створити потік 1"<<endl;
      exit (EXIT_FAILURE);
    }

    er = pthread_create(&threads[1], NULL, G, (void *) & arg);
    if (er > 0)
    {
      cout<<"Не вдалося створити потік 2"<<endl;
      exit (EXIT_FAILURE);
    }

    for (int i = 0; i < 2; i++)
    {
      pthread_join (threads[i], NULL); // приєднуємо потоки
    }
    cout<<"F(x) + G(x) = "<<res1 + res2<<endl;
    return 0;
  }

  void * F (void * arg) // перша потокова функція
  {
    DATA* a = (DATA*) arg;
    res1 = pow (a->x, 2);
    return 0;
  }

  void * G (void * arg) // друга потокова функція
  {
    DATA* a = (DATA*) arg;
    res2 = pow (a->x, 3);
    return 0;
  }
```

Для роботи з потоками на початку програми включаємо файл заголовків `pthread.h`. Глобальні змінні `res1` та `res2` використаємо для зберігання результатів потокових функцій. Потокові функції назвемо відповідно `F()` та `G()` — їх оголошено на початку програми та описано наприкінці, після функції `main ()`.

Функція `F()` формально використовує аргументи і результат типу `void *`. На практиці у ролі аргументу вона використовує поле `x` структури `DATA_` (переданої через вказівник). Можна використати й складнішу структуру з більшою кількістю полів, однак для даної програми вистачить і структури з одним полем. У тілі функції `F()` значення `x` підноситься до квадрату, а результат обчислень зберігається у глобальній змінній `res1`. Функція `G()` працює аналогічно.

Основний хід роботи програми відображено у функції `main()`. Після необхідних оголошень на введення з клавіатури значення `x` створюється перший потік. Це робиться за допомогою функції `pthread_create()`, а виконуватиметься у цьому потоці функція `F()`. Також у першого потоку є вказівник, що зберігатиметься у масиві вказівників `threads`, що має тип `pthread_t`. Це одновимірний масив на два елементи — за кількістю потоків, передбачених у програмі. Першому потоку відповідатиме вказівник `threads[0]`. Якщо створення потоку пройшло успішно, функція `pthread_create()` поверне значення 0. В іншому випадку виводимо повідомлення про помилку і достроково завершуємо роботу програми. Аналогічно працює створення другого потоку (вказівник `threads[1]`, потокова функція `G()`).

Далі, завдяки поміщеній у цикл функції `pthread_join()`, головна функція очікує на завершення обох потоків, перш ніж виконуватися далі. Очікування відбувається саме ті два потоки, вказівники на які зберігаються у масиві pthreads, що видно з відповідного параметра функції `pthread_join()` та запису циклу `for`. У Linux такий підхід називається **приєднанням потоків** (*thread joining*).

Щойно обидві потокові функції завершать обчислення, функція `main()` продовжує роботу: знаходить суму результатів потокових функцій та виводить його на екран.

У лістингу 3.2 продемонстровано програму для Windows.

Лістинг 3.2. Проста багатопотокова програма на основі Windows Threads

```
#include <iostream>
#include <windows.h>
#include <math.h>

using namespace std;
float res1, res2;
float x;

HANDLE hThreads[2]; // масив дескрипторів потоків

void F ();
void G ();

int main()
{
  cout<<"Введіть значення аргументу x: "; cin>>x;
  DWORD IDThread1, IDThread2; // змінні для ідентифікаторів потоків

  hThreads[0] = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)F,
                              NULL, 0, &IDThread1);
  if (hThreads[0] == NULL)
  {
    cout<<"Ne vdalosya stvoryty 1-i potik!"<<endl;
    return 1;
  }
```

```
  hThreads[1] = CreateThread (NULL, 0, (LPTHREAD_START_ROUTINE)G,
                              NULL, 0, &IDThread2);
  if (hThreads[1] == NULL)
  {
    cout<<"Ne vdalosya stvoryty 2-j potik!"<<endl;
    return 2;
  }
  // Чекаємо на завершення потоків так довго, як знадобиться
  WaitForMultipleObjects (2, hThreads, TRUE, INFINITE);

  CloseHandle (hThreads[0]);
  CloseHandle (hThreads[1]);

  cout<<"f(x) + g(x) = "<<res1 + res2<<endl;
  return 0;
}

// Перша потокова функція
void F ()
{
  res1 = pow (x,2);
}
// Друга потокова функція
void G ()
{
  res2 = pow (x,3);
}
```

У Windows для роботи з потоками потрібно включити файл заголовків `windows.h` (це досить універсальний файл заголовків, який дуже часто використовується у програмуванні для Windows, зокрема, він містить і стандартні типи даних Windows). Для зберігання результатів потокових функцій знову використаємо глобальні змінні `res1` та `res2`. Самі потокові функції названо аналогічно — `F()` та `G()`.

Функція `F()` у ролі аргументу, як і у попередньому прикладі, використовуватиме глобальну змінну, але цього разу це буде змінна `x` типу `float`. У тілі функції `F()` буде обчислюватися значення x^2, а результат потраплятиме у глобальну змінну `res1`. Функція `G()` працює подібним чином.

У функції `main()` вводиться значення `x` та оголошуються необхідні змінні. Далі створюється перший потік за допомогою функції `CreateThread()`. Серед параметрів функції звернімо увагу на параметр `(LPTHREAD_START_ROUTINE)F` — у такий спосіб у даному потоці виконуватиметься функція F, причому з самого початку. Параметр `&IDThread1` дасть змогу повернути ідентифікатор новоствореного потоку типу `DWORD`. Функція `CreateThread()` повертає дескриптор новоствореного потоку типу `HANDLE` (один зі стандартних типів Windows). Дескриптор буде виконувати роль, подібну до тієї, яку виконували вказівники на потоки у програмі для Linux. Загалом дескрипторів потоків буде два, для

кожного потоку — свій, і вони зберігатимуться у масиві `hThreads`. Якщо ж створення потоку не було успішним, функція `CreateThread()` повертає результат NULL — тоді виводимо повідомлення про помилку і завершуємо роботу програми. Аналогічно відбувається створення другого потоку (вказівник `hThreads[1]`, потокова функція `G()`).

Для очікування на завершення потоків використовуємо функцію `WaitForMultipleObjects()`. У параметрах функції вказано кількість потоків, на які треба очікувати (`2`), масив з дескрипторами цих потоків (`hThreads`). Також очікування триватиме стільки, скільки знадобиться (`INFINITE`).

Після завершення потоків у Windows важливо також закрити їхні дескриптори за допомогою функції `CloseHandle()`.

Результати роботи потокових функцій сумуються, значення суми виводиться на екран.

Як бачимо, основні прийоми для роботи з потоками у POSIX Pthreads та Windows Threads загалом подібні. Відмінності стосуються деталей використання і переважно обумовлені особливостями програмування під ці досить різні платформи.

Опосередкована багатопотоковість.

Багатопотоковість, безумовно, є важливим трендом сучасних ОС, а тому засоби для створення багатопотокових застосунків активно розвиваються. Так, ідея *опосередкованої багатопотоковості* полягає в делегуванні відповідальності за створення потоків та керування ними від розробників програм до бібліотек часу виконання (run-time libraries). Таким чином, паралельні завдання відокремлюються від реалізації багатопотоковості. При цьому розробник визначає паралельні завдання, а бібліотеки тим часом відповідають за виконання цих завдань як потоків.

Серед прикладів реалізації опосередкованої багатопотоковості згадаємо **пули потоків** (thread polls, використовується в Android, Java), **відгалуження-об'єднання** (fork-join, використовується в Unix-подібних ОС, Java), **OpenMP** (Open Multi-processing, на основі ідеї паралельних областей), **Grand Central Dispatch** (розробка Apple, використовується у macOS, iOS, на основі ідеї динамічного підлаштування кількості потоків у пулі), **Intel Thread Building Blocks** та ін. [1, с. 176-188].

Контрольні запитання

1) Що із переліченого найчастіше стосується потоку, а що — процесу, у межах якого цей потік виконується разом з іншими потоками?
   лічильник команд; значення регістрів інших ЦП; стек; код програми; дані (які зберігаються не в регістрах і не у стеку); сигнали; відкриті файли.
2) Які потенційні переваги має багатопотоковість?
3) Як співвідносяться терміни *паралелізм, псевдопаралелізм, апаратний паралелізм, багатозадачність*?
4) Назвіть приклади викликів, які багатопотоковість ставить перед розробниками ОС та перед прикладними програмістами, які розробляють багатопотокові застосунки.
5) Які моделі процесів та потоків можна виокремити залежно від кількості процесів/потоків? У яких ОС використовуються ці моделі?

6) Охарактеризуйте моделі багатопотоковості “багато до одного” (many-to-one), “один до одного” (one-to-one), “багато до багатьох” (many-to-many), дворівнева модель (two-level model). У яких ОС використовуються ці моделі?
7) Чим багатопотоковість завдань відрізняється від багатопотоковості даних? У яких випадках застосовується кожна з цих стратегій?
8) Чим синхронна багатопотоковість відрізняється від асинхронної багатопотоковості? У яких випадках дотримуються кожної з цих стратегій?
9) Назвіть приклади бібліотек багатопотоковості. Яка бібліотека багатопотоковості зазвичай використовується у Linux-системах? у Windows-системах?
10) Назвіть основні кроки створення простої багатопотокової програми, у якій спочатку створюється два окремих потоки, а після їх завершення результати передаються у головну функцію. Які функції знадобляться з написання такої програми засобами бібліотеки POSIX Pthreads? Windows Threads?
11) * Нехай потрібно написати просту багатопотокову програму. Програма обчислює значення функції `A()` в одному потоці і функції `B()` — у другому потоці. Потім результати обчислення передаються в головну функцію, після чого знаходиться їхній добуток і виводиться на екран.

    Функції `A()` та `B()` задано наступним чином:

    $$A(x) = x^2 + 12;$$
    $$B(x) = x^3 - x + 1.$$

    Напишіть та протестуйте просту багатопотокову програму:

    а) на базі бібліотеки POSIX Pthreads;

    б) на базі бібліотеки Windows Threads.
12) Поясніть, у чому полягає ідея опосередкованої багатопотоковості. Назвіть приклади реалізації опосередкованої багатопотоковості.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 4.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 4.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 2 (2.2).
4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділ 3.

Розділ 4

Планування та диспетчеризація

1. Основні поняття планування

Планування ЦП чи планування введення-виведення. Передусім нам необхідно визначитися, про яке саме планування йтиметься у даному розділі. Серед розглянутих раніше системних ресурсів планування потребують ті, керування якими здійснюється у часі (згадаймо: *спочатку це — потім це*). Тобто у теорії ОС йдеться або про **планування ЦП** (планування ЦП, CPU scheduling), або про **планування введення-виведення** (I/O sheduling). Це два окремі напрями, і планування ЦП має свої особливості, а планування введення-виведення — свої.

У даному посібнику ми зосередимося на плануванні використання ЦП. Планування введення-виведення детальніше розглядається, наприклад, у підручнику В. Столлінґса [2, ch. 11].

Планування процесів чи планування потоків. У розділі 3 ішлося про те, що у багатопотокових системах потоки може бути реалізовано на рівні ядра або на рівні користувача, і коли потоки реалізовано на рівні ярда, то ядро працює безпосередньо з потоками, коли ж потоки реалізовано на рівні користувача, то ядро у кожний момент часу "бачить" лише один з потоків даного процесу й, відповідно, працює з процесами. Це безпосередньо впливає на те, що є об'єктом планування в ОС — потік чи процес. Якщо ОС підтримує потоки на рівні ядра — то й планування ЦП здійснюється для потоків, якщо ж підтримка потоків в ОС працює у просторі користувача — планування ЦП відбувається для процесів.

Надалі у цьому розділі ми для спрощення говоритимемо про планування процесів. Там, де це принципово, уточнюватимемо, про яке саме планування йдеться — процесів чи потоків. Також ми переважно розглядатимемо один ЦП з

одним ядром. Планування у разі наявності кількох ядер / кількох ЦП буде спиратися на планування для випадку одного ядра (детальніше у [2, ch. 10]).

Враховуючи описані вище домовленості, можемо означити планування наступним чином:

Планування — це встановлення послідовності виконання процесів на ЦП.

Планування здійснює відповідний компонент ОС — **планувальник** (scheduler). Саме планувальник вибирає процес, який має виконуватися наступним. Планувальник діє згідно з **алгоритмом планування**, часто — за кількома алгоритмами. Зазвичай планування передбачає наявність черги (черг) процесів. У чергу можуть потрапляти процеси, готові до виконання, але ще не вибрані планувальником. Також окремі черги можуть існувати й для інших процесів — наприклад, для процесів у стані очікування чи процесів, тимчасово вивантажених з оперативної пам'яті на диск.

Диспетчер (dispatcher) дає змогу процесу, вибраному планувальником, почати виконуватися на ЦП шляхом перемикання контексту.

Образно кажучи, планувальник приймає рішення (згідно з закладеними в нього алгоритмами), а диспетчер це рішення виконує, зокрема диспетчер:

- перемикає контекст з одного процесу на інший;
- перемикається у режим користувача;
- переходить на те місце у програмному коді, звідки треба почати (продовжити) виконання процесу.

Перемикання контексту може бути добровільним або примусовим.

Добровільне перемикання контексту передбачає, що процес перейшов у стан очікування (сам), або самостійно завершився.

Примусове перемикання контексту відбувається тоді, коли процес міг би виконуватися далі, проте його перервано системою. Це трапляється, наприклад, коли на виконання претендує інший, важливіший, процес, коли процесом вичерпано відведений квант часу тощо.

Залежно від характеру перемикання контексту, виокремлюють планування без витіснення та планування з витісненням.

Планування без витіснення (non-preemptive scheduling) передбачає, що процесу надається можливість виконуватися поки процес не завершиться сам або поки процес не перейде у стан очікування. Іншими словами, у випадку планування без витіснення має місце *лише добровільне перемикання контексту*.

Планування з витісненням (preemptive scheduling) допускає переривання виконання процесу з передаванням права на виконання іншому процесу. Тобто у разі планування з витісненням застосовується *не лише добровільне, а й примусове перемикання контексту*.

Вимоги до планувальника та диспетчера. Розпочнімо з диспетчера, оскільки, у зв'язку з його суто виконавчою роллю, вимоги до нього сформулювати простіше. *Основна вимога до диспетчера* — працювати дуже швидко, оскільки перемикання контексту може відбуватися часто.

Натомість вимоги до планувальника менш однозначні, оскільки значною мірою залежать від призначення ОС та умов, у яких вони працюють. Серед типових критеріїв планування — відсоток використання ЦП, пропускна здатність, час відгуку, час очікування, час обороту.

Використання ЦП (CPU utilization) показує, який відсоток часу ЦП зайнятий. Зазвичай висувається вимога, щоб використання ЦП коливалося у межах від 40 до 90% [1, с. 204].

Пропускна здатність (throughput) дорівнює кількості процесів, завершених за одиницю часу.

Час відгуку (response time) — це час від надходження запиту до початку відповіді системи. Йдеться саме про *початок* відповіді — після того, як користувач натиснув клавішу на клавіатурі, клацнув по значку на екрані тощо.

Критерій особливо актуальний для інтерактивних систем, оскільки саме за швидкістю реакції системи на запити користувач суб'єктивно оцінює продуктивність цієї системи. Простіше кажучи, якщо час відгуку буде великим, користувач матиме враження, що система працює повільно — навіть якщо, нарешті розпочавши виконуватися, завдання завершаться швидко.

Час очікування (waiting time) важливо не плутати з часом відгуку. Час очікування включає час, протягом якого очікує не користувач, а процес. Це час, який процес проводить у чергах. При цьому йдеться про весь час в усіх чергах. Так, процес може потрапляти у чергу багато разів, а тоді знову повертатися до виконання. Також процес може перебувати спершу в одній черзі, а потім в іншій — важливо врахувати усі ці періоди.

Час обороту (turnaround time) обчислюється як весь час, витрачений на виконання процесу. Час обороту включає як час, протягом якого процес виконувався на ЦП, так і час очікування на введення-виведення та час очікування в чергах. Тобто фактично час обороту може значно перевищувати час безпосереднього виконання процесу на ЦП.

Використання ЦП та пропускну здатність зазвичай прагнуть максимізувати. Час відгуку, час очікування та час обороту, навпаки, переважно мінімізують. Часто мінімізують чи максимізують середнє значення критерію або його максимальне / мінімальне значення. А, наприклад, в інтерактивних системах рекомендовано зменшувати не максимальний чи середній час відгуку, а його дисперсію (тобто те, наскільки окремі значення часу очікування відрізняються від середнього). Так система поводиться передбачуваніше для користувача, а робота з нею є комфортнішою [1, с. 205].

Отже, підсумуємо сформульовані вище вимоги до планувальника.

- ЦП не має простоювати, коли є невиконані завдання.
- Система має бути готова оперативно реагувати на нові запити, якщо вони нагальні.
- В одиницю часу потрібно виконувати якомога більше процесів.
- Процес потрібно виконувати швидко.
- Не варто тримати процес у черзі занадто довго.

Прогляньте перелік вимог іще раз і спробуйте уявити, що планувальник — це ви, а процеси — це люди, які звертаються до вас з різними заявками, нерідко одночасно. Якщо вам знадобиться обслужити людину з терміновою заявкою поза чергою, то інша людина муситиме довше простояти у черзі. Якщо при цьому вам також доведеться щокілька хвилин відповідати на телефонні дзвінки чи повідомлення у месенджері, то ваша "операційна система" справить гарне враження на віддалених клієнтів, але погане враження — на тих хто тим часом чекає поряд із вами в одному приміщенні. Можливо, роботу з віддаленими клієнтами можна доручити іншому працівнику, та всіх проблем це не вирішить.

Описана метафора, звісно, не передає нюансів роботи справжнього планувальника, але допомагає зрозуміти: на практиці, у конкретній системі, всі сформульовані вимоги не можуть бути однаково пріоритетними, чимось

доведеться пожертвувати. Тому, скажімо, у системах пакетної обробки та їхніх сучасних аналогах найвагомішими будуть критерії, які допомагають оцінити результат роботи (наприклад, пропускна здатність), в інтерактивній системі — особливо важливим буде час відгуку, в системах реального часу — додасться вимога, щоб процеси виконувалися у визначені строки згідно з певним розкладом або до визначеного дедлайну. Також, які б алгоритми планування не застосовувалися, важливо, аби затрати часу на виконання процесів з таким плануванням не перевищували затрати часу на виконання тих самих процесів без планування.

Ми ще повернемося до особливостей планування у різних типах ОС у п. 2, а поки розглянемо інші теоретичні аспекти планування в ОС.

Інтервали використання ЦП та інтервали введення-виведення. Виконання більшості процесів є чергуванням двох видів активності — інтервалів використання ЦП (CPU bust) та інтервалів введення-виведення (I/O burst). Процес виконується, потім йому потрібно зробити введення-виведення, потім він знову зможе повернутися до виконання, згодом йому знову знадобиться здійснити введення-виведення і т.д.

Залежно від виду активності, що переважає, виокремлюють процеси, обмежені швидкістю обчислень, та процеси, обмеженні швидкістю введення-виведення.

У **процесів, обмежених швидкістю обчислень** (**процесів, прив'язаних до швидкості обчислень**, CPU bound processes) переважає використання ЦП, а введення-виведення відбувається рідше (рис. 4.1). Тому вони й *обмежені* швидкістю обчислень: якби ЦП працював ще швидше, це б прискорило виконання таких процесів.

Рис. 4.1. Схематичне подання активності процесу, обмеженого швидкістю обчислень (CPU bound process)[16]

Процеси, обмежені швидкістю введення-виведення (**процеси, прив'язані до введення-виведення**, I/O bound processes), навпаки, більше використовують пристрої введення-виведення, а виконують свої інструкції на ЦП менше (рис. 4.2). Ось чому вони *обмежені* швидкістю введення-виведення: пришвидшення роботи пристроїв введення-виведення суттєво скоротило б час виконання цих процесів.

Рис. 4.2. Схематичне подання активності процесу, обмеженого швидкістю введення-виведення (I/O bound process)

[16] Схеми на рис. 4.1 та 4.2 адаптовано за джерелом: Tanenbaum, Bos. Modern Operating Systems, 2014

На практиці сучасні ЦП працюють значно швидше за пристрої введення-виведення. Що швидше працює ЦП, то більше процесів, обмежених швидкістю обчислень, переходять у категорію процесів, обмежених швидкістю введення-виведення (прямокутники на рис. 4.1 ставатимуть коротші, й нагадуватимуть прямокутники на рис. 4.2).

Розподіл періодів обчислень та очікування на введення-виведення може впливати на вибір алгоритмів планування. Це питання буде детальніше розглянуто у п. 2.

Семистанова модель процесів. У своєму щоденному житті ми тримаємо в голові не всі справи, а лише найактуальніші на даний момент. Водночас, ми часто маємо тимчасово відкладені справи (за які візьмемося, щойно з'явиться можливість), а також довгострокові плани. Тут планування виконання процесів на ЦП в ОС виглядає дещо схоже. Для того, щоб краще з цим розібратися, перейдімо від п'ятистанової моделі процесів (розглянутої у розділі 2) до **семистанової моделі процесів** (seven-state process model).

Розглянемо ще два типових стани процесів:

- Очікування / Відкладено (Blocked / Suspended)
- Готовий до виконання / Відкладено (Ready / Suspended)

Відкладання процесів пов'язане з їх вивантаженням з оперативної пам'яті на диск у межах механізму **підкачування** (swapping) — зокрема задля економії оперативної пам'яті. Згодом процес повертається в оперативну пам'ять, і його знову може бути обрано для виконання.

Підкачування тісно пов'язане з керуванням пам'яті в ОС, яке детальніше розглянуто в розділі 5. Однак підкачування впливає і на роботу планувальника, оскільки для відкладених процесів знадобляться окремі черги, які відповідатимуть різним видам планування.

За тривалістю, виокремлюють три основні види планування: довгострокове, середньострокове та короткострокове. Разом вони утворюють семистанову модель процесів (рис. 4.3).

Довгострокове планування (long-term scheduling) передбачає вибір процесу з черги завдань (job queue) і поміщення його у чергу процесів, готових до виконання (ready queue). У межах довгострокового планування планувальник також приймає рішення, чи створювати новий процес узагалі. Загалом, тут можливі обмеження — наприклад, у системах реального часу, де фактор часу дуже важливий, процес може не бути створено, коли виконуване ним завдання не встигає виконатися у задані строки.

У межах **середньострокового планування** (medium-term scheduling) планувальник керує чергами відкладених завдань. Власне, саме з середньостроковим плануванням і пов'язане введення двох додаткових станів у семистановій моделі.

Короткострокове планування (short-term scheduling) передбачає вибір процесу з черги процесів, готових до виконання (ready queue). Саме цьому процесу передає керування диспетчер.

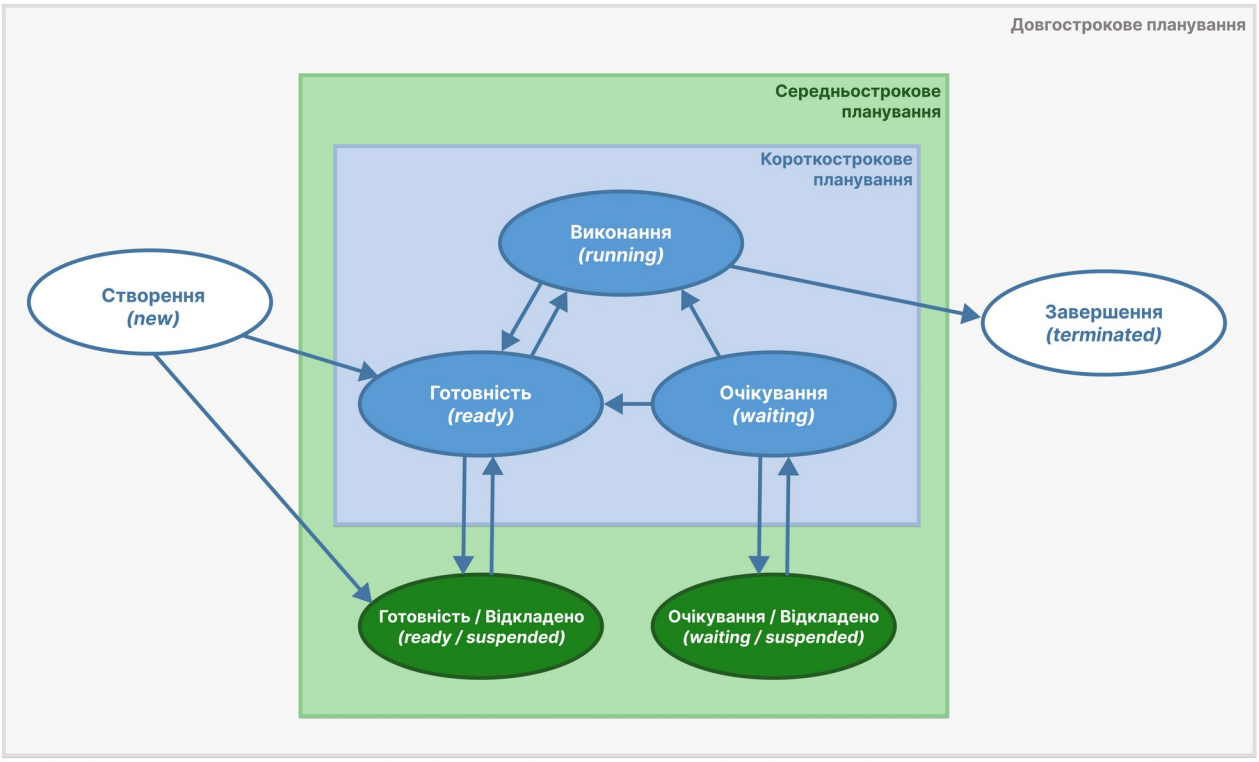

Рис. 4.3. Семистанова модель процесів

Досі ми переважно говорили про перебування у черзі процесів, готових до виконання. Однак процеси в інших станах також можуть потребувати черг. У теорії ОС розглядають чергу процесів, готових до виконання (ready queue), чергу відкладених процесів, готових до виконання (ready, suspend queue), чергу процесів у стані очікування (blocked queue / waiting queue), чергу відкладених процесів у стані очікування (blocked, suspend queue / waiting, suspend queue) [2, п. 9.1].

Розглянемо переходи між станами на рис. 4.3 детальніше. У межах довгострокового планування відбувається створення процесу. Далі новостворений процес може перейти у стан “Готовність” або у стан “Готовність / Відкладено”. В обох випадках він опиниться у черзі процесів — або готових до виконання, або відкладених і готових до виконання. Також довгостроковому плануванню відповідає завершення процесу — це перехід зі стану “Виконання” у стан “Завершення.

Середньострокове планування пов’язане зі станами “Готовність / Відкладено” та “Очікування / Відкладено”. Процеси у цих станах не просто перебувають у відповідних чергах — вони не можуть одразу перейти у стан “Виконання”. Процес у стані “Готовність / Відкладено” спершу має перейти у стан “Готовність” (тобто його має бути повернуто з диска в оперативну пам’ять), й лише після цього він може претендувати на виконання на ЦП. Аналогічно, процес у стані “Очікування / Відкладено” спочатку мусить перейти у стан “Очікування”, тоді у стан “Готовність”, а вже після цього претендуватиме на виконання.

Короткострокове планування містить базові стани, які було детально розглянуто раніше у межах п’ятистанової моделі (розділ 2). Зазначимо лише, що процес у стані “Готовність” може перейти у стан “Готовність / Відкладено”, а процес у стані “Очікування” — у стан “Очікування / Відкладено”. За яких саме обставин це відбувається, залежить від конкретної ОС, але загалом метою

такого переходу є звільнення місця в оперативній пам'яті від коду та даних, які не потрібні безпосередньо зараз і, ймовірно, не знадобляться найближчим часом. Скажімо, якщо у Windows вікно програми лишається згорнутим більше 5 секунд, то відповідний процес вважається таким, що його можна вивантажувати на диск [5].

2. Особливості планування у різних типах ОС

У попередніх темах ми вже розглядали класифікацію ОС за сферою застосування та за архітектурою ядра. У цьому розділі, у контексті планування нам знадобиться ще одна класифікація — класифікація ОС залежно від характеру отримуваних ними завдань.

Залежно від характеру отримуваних завдань виокремлюють три основні класи ОС: системи пакетної обробки, інтерактивні системи та системи реального часу.

Системи пакетної обробки асоціюються зі старими системами, які існували ще у добу ЕОМ на базі транзисторів на зразок FMS чи IBSYS. Та насправді подібні системи та їх елементи зустрічаються і в сучасному світі. Так, подібні системи знайшли застосування у ґрід-обчисленнях. Також за подібним принципом функціонують сценарії командного рядка, без яких важко уявити сучасне адміністрування ОС. Інший приклад — засоби для запланованого запуску завдань на зразок cron у Unix-подібних ОС чи Планувальника завдань у Windows. І насамкінець, саме у системах пакетної обробки використовуються найбазовіші алгоритми планування (наприклад, FIFO).

У системах пакетної обробки завдання надходять пакетами, і їх потрібно виконати якомога швидше. Активної взаємодії з користувачем у таких завдань немає, тому завданням зазвичай дають змогу виконуватися якомога довше, не перериваючи їх без вагомої причини. Відповідно, тут найкраще підходять алгоритми без витіснення або алгоритми з витісненням, але з якнайдовшим періодом безперервного виконання.

Інтерактивні системи, на противагу системам пакетної обробки, передусім орієнтовані на активну взаємодію з користувачем. Дії користувача часто непередбачувані, водночас на них необхідно постійно реагувати. Більшість ОС, які спадають на думку, коли ми чуємо "операційна система" — це інтерактивні системи.

Для врахування фактору користувача, який потребує постійної уваги й опрацювання своїх запитів, в інтерактивних системах переважають алгоритми з витісненням. Застосування алгоритмів без витіснення можливе, але все ж має бути передбачено переривання тривалого виконання процесів, адже коли котрийсь із процесів неперервно виконуватиметься надто довго, користувач це помітить — для нього виглядатиме, ніби система зависла.

Системи реального часу мають підвищені вимоги до часу виконання — завдання мають виконуватися у визначені строки. У таких системах витіснення має місце передусім для коректної й оперативної реакції на зовнішні події. Але відсутність потреби постійно реагувати на потреби користувача дається взнаки, адже у системах реального часу:

- робота процесів спрямована на досягнення спільної мети;
- процеси поводяться передбачувано;
- процеси запускаються ненадовго, виконують свою частину роботи і переходять у стан очікування;
- відсутні сторонні програми.

У системах реального часу використовуються *статичні* та *динамічні* алгоритми планування, відмінність між якими полягає у тому, що в статичних алгоритмах планування здійснюється до запуску системи, а в динамічних алгоритмах — вже під час роботи системи.

У наступному пункті буде детальніше розглянуто приклади алгоритмів планування, які можуть використовуватися у наведених класах ОС.

3. Приклади алгоритмів планування

Алгоритм FIFO (First In — First Out, дослівно: першим прийшов — першим вийшов, інша назва — FCFS, First Came — First Served, дослідно: першим прийшов — першим обслужили) є найпростішим базовим алгоритмом.

FIFO є алгоритмом без витіснення, тобто має місце лише добровільне переривання контексту (процес не переривають, але він сам може перейти у стан очікування, щоб зробити введення-виведення). Черга процесів, готових до виконання, у FIFO побудована подібно до звичайної черги. Новоприбулі процеси поміщаються у хвіст черги. Якщо процес А переходить зі стану виконання у стан очікування, право на виконання надається процесу В (голові черги). Коли процес А знову перейде у стан готовності, його буде поміщено у хвіст черги.

Безумовною перевагою алгоритму FIFO є простота реалізації. Та, оскільки цей алгоритм узагалі не передбачає витіснення, він не підходить для інтерактивних ОС. Також FIFO притаманний так званий “ефект конвою”: процеси, орієнтовані на введення-виведення, будуть дискримінуватися процесами, орієнтованими на обчислення. Якщо процес часто здійснює введення-виведення, то при цьому він весь час потраплятиме в хвіст черги і в результаті щоразу буде наново проходити шлях до голови черги. Так подібні процеси виконуватимуться значно довше за ті, що переважно виконуються на ЦП.

Візуалізацію прикладу роботи алгоритму FIFO у формі діаграми Ґанта наведено на рис. 4.4.

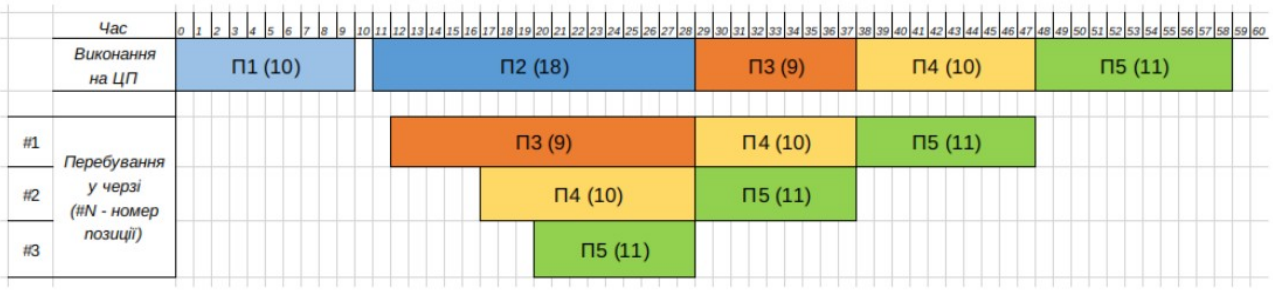

Рис. 4.4. Приклад діаграми Ґанта для алгоритму FIFO[17]

Алгоритм SJF (Shortest Job First, дослівно: спершу найкоротше завдання) на практиці трохи не відповідає своїй назві, оскільки йдеться про надання переваги не взагалі найкоротшому завданню, а завданню з *найкоротшим наступним періодом обчислювальної активності* (CPU burst).

Серед готових до виконання процесів вибирається той, у якого орієнтовний розмір наступного періоду обчислювальної активності найменший.

Існує варіант цього алгоритму без витіснення та з витісненням. Варіант без витіснення — це класичний SJF, варіант з витісненням має назву SRT (Shortest Remaining Time, дослівно: найменший час до кінця виконання). Якщо

[17] Задля спрощення у цьому та наступних прикладах вважатимемо ЦП одноядерним. Також у цих прикладах кожний процес має лише один період обчислювальної активності (CPU burst), не враховується здійснення процесами введення-виведення. Витрати часу на перемикання контексту прийнято за такі, що ними можна знехтувати

використовується алгоритм SRT, то виконуваний процес може бути витіснений новоприбулим процесом з меншим орієнтовним часом наступного періоду обчислювальної активності. У класичному SRT процес із меншим орієнтовним часом наступного періоду обчислювальної активності лише посуває у черзі ті процеси, в яких аналогічний показник більший.

Перевагою алгоритму SJF є можливість оперативно реагувати на короткі швидкі запити і, як наслідок, уникнення "ефекту конвою". Потенційним недоліком є складність оцінювання орієнтовного часу наступного періоду обчислювальної активності процесу, оскільки заздалегідь невідомо, скільки часу виконуватиметься процес до наступного введення-виведення. Для подібного оцінювання застосовуються наближені методи, зокрема метод екпоненціального згладжування (exponential average, exponential smoothing [1, с. 208], [2, с. 868]), спираючись на тривалість попередніх періодів обчислювальної активності відповідних процесів.

Візуалізацію прикладу роботи алгоритму SJF продемонстровано на рис. 4.5. У дужках для кожного процесу наведено два значення. Перше значення є орієнтовним часом наступного періоду обчислювальної активності процесу (одержане наближеними методами), а друге значення — фактична тривалість цього періоду. Заштриховані області відповідають періодам, коли процес мав би виконуватися, згідно з орієнтовним значенням. Але насправді процес виконувався той період, який залито відповідним кольором. Іноді це більше (наприклад, процес П2), іноді — менше (наприклад, процес П3). Однак орієнтовний час вплинув на функціонування черги. Процес П3, який, згідно з попереднім оцінюванням, мав би виконуватися 15 мс, у черзі постійно посувався іншими процесами і, зрештою, виконався останнім. Це могло бути виправдано, але у даному прикладі вийшло інакше — всупереч оцінкам, процесу П3 знадобилося для виконання лише 9 мс.

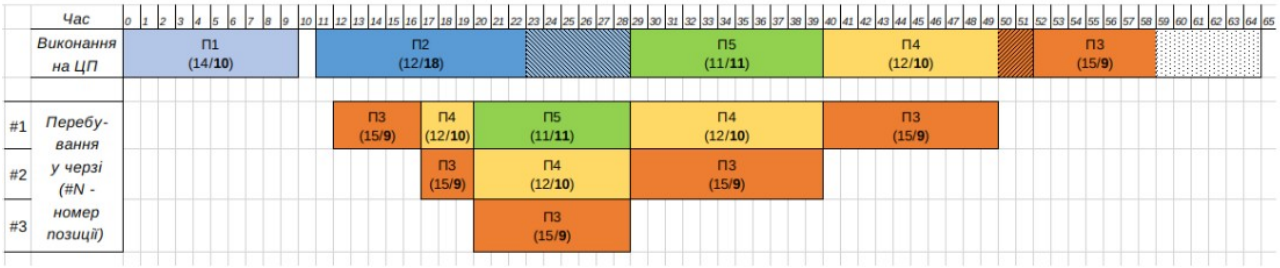

Рис. 4.5. Приклад діаграми Ґанта для алгоритму SJF

Алгоритм RR (Round Robin, циклічне планування) спирається на припущення, що всі процеси рівноважливі.

В алгоритмі RR кожному процесу виділяється для виконання певний період часу — квант часу (time slice, time quantum). Процес може завершитися сам, до того, як буде вичерпано квант. Якщо ж квант вичерпано, а процес ще не завершився, то процес може бути витіснено наступним у черзі процесом. Черга для RR організовується за алгоритмом FIFO, але є циклічною. RR поєднується й з іншими алгоритмами. Зокрема, чергу можна організувати не за FIFO.

На рис. 4.6 та 4.7 показано приклади роботи алгоритму RR з двома різними квантами.

Розміри кванту зазвичай коливаються у проміжку 10-100 мілісекунд. Оптимальний розмір кванту залежить від особливостей використання системи, характеру виконуваних завдань та інших факторів. Надто короткий квант означатиме надто часті перемикання контексту (це добре видно, якщо порівняти рис. 4.6 і рис. 4.7). Натомість, надто довгий квант призведе до деградації RR у бік звичайного FIFO, що не підходить для інтерактивних систем.

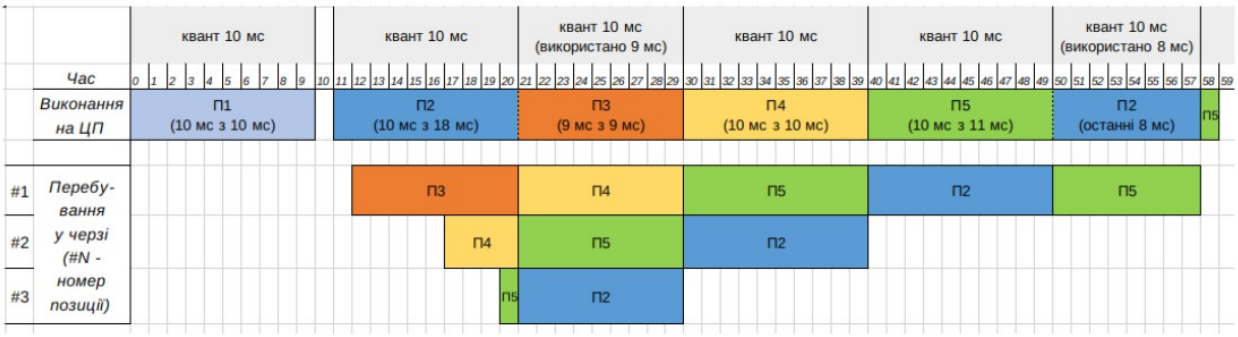

Рис. 4.6. Приклад діаграми Ґанта для алгоритму RR з квантом $q = 10$ *мс*

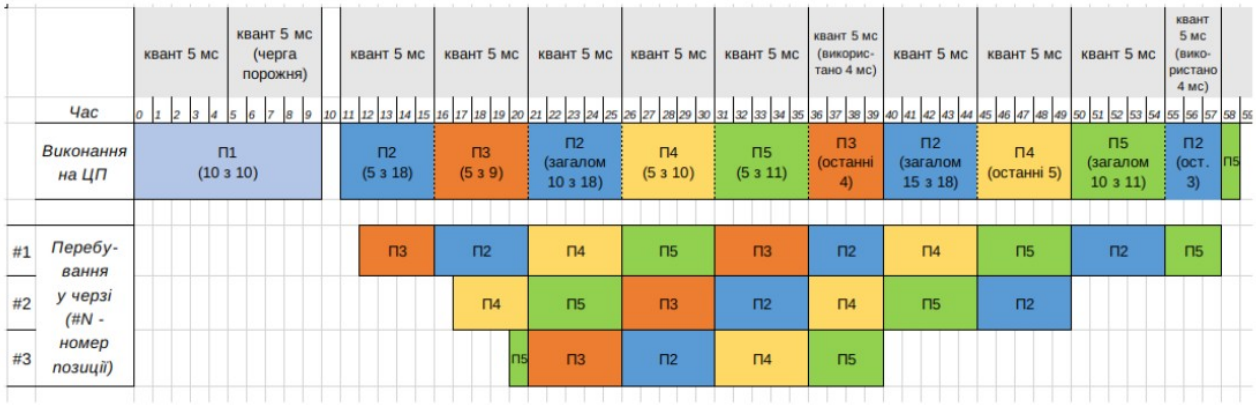

Рис. 4.7. Приклад діаграми Ґанта для алгоритму RR з квантом $q = 5$ *мс*

Гарна практика щодо визначення розміру кванту полягає у наступному: 80% періодів обчислювальної активності процесів мають бути коротші за квант [1, с. 211]. Тобто більшість процесів встигатиме завершити свій черговий період обчислювальної активності за виділений їм квант, а рідкісні випадки надто тривалого виконання будуть тимчасово перериватися на користь інших процесів.

Алгоритм пріоритетного планування (Priority Scheduling) спирається на припущення, що одні процеси важливіші за інші.

В алгоритмі пріоритетного планування кожному процесу присвоюється пріоритет. Процеси з вищим пріоритетом мають перевагу під час вибору планувальником. Часто процеси групують у класи з різним пріоритетом — тоді у межах одного класу процеси можуть вважатися рівнозначними, і до них буде застосовуватися RR чи інший алгоритм.

Системи пріоритетів у різних ОС можуть мати суттєві відмінності. Наприклад, у Linux вищому пріоритету буде відповідати менше числове значення, а нижчому пріоритету — більше числове значення. У Windows, навпаки, більше числове значення буде означати вищий пріоритет, а менше — нижчий.

Один процес може мати декілька пріоритетів, крім того, пріоритет процесу може змінюватися у ході його виконання. Переглядати пріоритети важливо ще й для уникнення **голодування** (starvation) — ситуації, за якої процеси з низькими пріоритетами не можуть отримати доступ до ЦП, скільки їх постійно посувають у черзі процеси з вищими пріоритетами.

Розглянемо ще декілька цікавих алгоритмів.

Гарантоване планування (Guaranteed Scheduling) спрямоване на те, щоб надати всім процесам рівні можливості щодо виконання на ЦП.

Гарантується, що кожний з n процесів отримає 1/n частину часу ЦП. Відстежується, скільки часу ЦП процес уже використав, і використаний час ділиться на розмір частки цього процесу (1/n). Наприклад, результат ділення 0,3 свідчить про те, що процес використав близько третини часу ЦП, а результат 2,7 — що процес перевикористав час ЦП на 1,7, і його має бути витіснено, а на його місце обрано процес з меншим відношенням використаного часу до 1/n.

Лотерейне планування (Lottery Scheduling) передбачає роздачу всім процесам лотерейних квитків. Доступ до ресурсів розігрується, і процес, якому належить квиток-переможець, одержує цей доступ. За потреби забезпечити процесу більшу пріоритетність, такому процесу дається більша кількість лотерейних квитків, що підвищує його шанси на перемогу.

Лотерейне планування може застосовуватися там, де складно використати більш традиційні алгоритми (наприклад, у роботі стримінгових сервісів [3, с. 163]).

Справедливе планування (Fair-Share Scheduling) дещо нагадує гарантоване планування, але у справедливому плануванні частка часу ЦП надається не процесам, а користувачам. Частка часу ЦП, виділена користувачу, може бути використана процесами цього користувача. Кількість процесів користувача не має значення, головне — не вийти за межі частки.

Справедливе планування дозволяє уникнути ситуацій, коли хтось із користувачів одержує більшість часу ЦП завдяки тому, що запустив велику кількість процесів, а користувачі з малою кількістю запущених процесів, навпаки, дискримінуються.

Планування у системах реального часу суттєво відрізняється від плануання в системах пакетної обробки та інтерактивних системах. У таких системах часові межі для виконання процесів чіткіші й більш передбачувані, ніж у системах загального призначення. Всі завдання або якомога більше завдань мають завершитися (або розпочатися) до деякого дедлайну. Для систем реального часу характерне суворіше використання пріоритетів та витіснення. Тому для таких систем не підходять ні алгоритми на зразок FIFO (немає витіснення), ні пріоритетне планування у чистому вигляді (не дозволяє організувати дотримання строків). Натомість може бути застосовано поєднання пріоритетів з перериваннями на основі таймера, а за потреби — негайне переривання.

Негайне переривання (immediate preemtion) передбачає, що ОС відповідає на переривання якомога швидше (майже негайно) за винятком випадків, коли ОС виконує критично важливий код, що має бути виконаний до кінця. Негайне переривання використовується для оперативної реакції на події зовнішнього середовища.

У статичних алгоритмах планування здійснюється до запуску системи, у динамічних — вже у ході її роботи. Основні підходи до планування у системах реального часу є наступними.

Статичний підхід на основі таблиці (static table-driven approach) передбачає статичний аналіз підходящого процесу. Статично визначається, коли саме процес має розпочатися.

Статичний підхід з пріоритетами і витісненням (static priority-driven preemptive) також застосовує статичний аналіз підходящого процесу, але статично визначаються пріоритети процесів.

Динамічний підхід на основі планування (dynamic planning-based approach) передбачає вибір підходящого процесу переважно у ході роботи ОС. За динамічного підходу на основі планування новоприбуле завдання приймається до виконання тільки коли воно відповідає часовим обмеженням, тобто коли це завдання можна поставити у розклад.

Динамічний підхід негарантованого виконання (dynamic best effort approach), на відміну від динамічного підходу на основі планування, не передбачає аналізу того, чи процес підходящий, і чи його вдасться виконати взагалі. Головне у динамічному підході негарантованого планування — дотриматися дедлайнів, і якщо процес не встигає до дедлайну, його буде завершено. Крім того, кожному процесу присвоюється пріоритет, виходячи з його характеристик.

На практиці підходи можуть поєднуватися. Статичні підходи добре підходять для періодичних завдань, а для менш прогнозованих завдань може бути обрано динамічні підходи.

Контрольні запитання

1) Яких двох системних ресурсів найчастіше стосується планування в ОС?
2) Планування ЦП в ОС здійснюється для процесів чи для потоків? Від чого це залежить?
3) Чим відрізняється роль диспетчера і планувальника в ОС?
4) Поясніть різницю між добровільним і примусовим перемиканням контексту.
5) Яке планування називається плануванням без витіснення? плануванням з витісненням?
6) Які вимоги висуваються до диспетчера?
7) Які основні вимоги висуваються до планувальника? За якими критеріями може оцінюватися ефективність планування в ОС? Від чого залежить, якими саме критеріями керуватимуться передусім?
8) Назвіть два основні види (інтервали) активності процесу в ОС. Поясніть терміни “процеси, обмежені швидкістю обчислень” та “процеси, обмежені швидкістю введення-виведення”.
9) Які два додаткові стани відрізняють семистанову модель процесів від п’ятистанової? З якою метою їх додано?
10) Які стани семистанової моделі відповідають довгосторокового, середньострокового, короткострокового планування? За яких умов відбуваються переходи між цими станами?
11) Перерахуйте основні особливості планування у системах пакетної обробки, інтерактивних системах та системах реального часу.
12) Опишіть роботу алгоритму планування FIFO. Які переваги та недоліки має цей алгоритм?
13) Опишіть роботу алгоритму планування SJF. Які переваги та недоліки має цей алгоритм?
14) Опишіть роботу алгоритму планування RR. Яким чином підбирається розмір кванту у цьому алгоритмі?
15) Опишіть роботу алгоритму пріоритетного планування. Які проблеми можуть виникнути, якщо не переглядати пріоритетів?

16) Порівняйте роботу алгоритмів гарантованого, лотерейного та справедливого планування. У яких ситуаціях підходить кожен із цих алгоритмів?
17) Що означає термін “негайне переривання” (immediate preemtion) стосовно планування у системах реального часу?
18) Опишіть та порівняйте основні статичні та динамічні підходи до планування у системах реального часу.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 5.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 9-10.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 2 (2.4).
4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділ 4.
5. P. Yosifovich, A. Ionesku, M. E. Russinovich, D. A. Solomon. Windows internals. Part1: System architecture, processes, threads, memory management, and more. 7-th edition. Microsoft Press, 2017.

Розділ 5

Пам’ять

1. Фізичні та віртуальні адреси

Керування пам’яттю — ще одна важлива функція, яку виконує операційна система. На рис. 5.1 наведено основні різновиди пам’яті у комп’ютерній системі та відображено важливі тенденції, які їх пов’язують. Так, регістри ЦП є надшвидкою пам’яттю, а дискові накопичувачі — найповільнішим різновидом пам’яті серед перелічених тут. Натомість, обсяг цих видів пам’яті у більшості систем має зворотну тенденцію: обсяг регістрів ЦП дуже обмежений, а сучасні дискові накопичувачі мають величезні обсяги. Вартість різних видів пам’яті у перерахунку на 1 байт буде найвищою в регістрів ЦП й найнижчою — у дискових накопичувачів.

Операційні система так чи інакше причетна до всіх наведених на рис. 5.1 видів пам’яті. Проте у цьому розділі ми зосередимося передусім на основній, або оперативній, пам’яті, а також певною мірою, торкнемося дискової пам’яті (для підкачування). Водночас, про дискову пам’ять детальніше йтиметься у розділів 6 “Файлові системи”.

Надалі, вживаючи термін “пам’ять”, зазвичай матимемо на увазі оперативну пам’ять, якщо не вказано інше.

Оскільки переважна більшість сучасних операційних систем є багатозадачними, то в оперативній пам’яті можуть одночасно перебувати більше багато процесів. У такому разі основне питання полягає у тому, як взаємно розташувати адресні простори процесів в оперативній пам'яті.

Рис. 5.1. Тенденції щодо характеристик основних видів пам’яті[18]

Найпростіше рішення виглядає приблизно так: адресні простори процесів послідовно розміщуються у пам'яті один за одним (рис. 5.2). Адреси на рис. 5.2 і далі у цьому розділі переважно наведено у шістнадцятковій системі числення, але таким чином, аби за потреби було зручно виконувати операції над значеннями адрес.

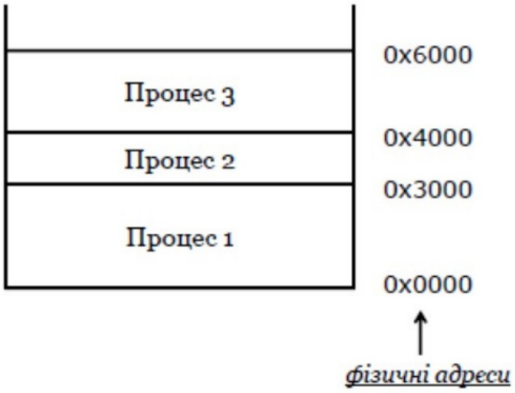

Рис. 5.2. Адресні простори процесів у разі роботи лише з фізичними адресами

На рис. 5.2 всі зображені адреси — це фізичні адреси.

Фізична адреса — це реальна адреса, за якою у пам'яті зберігаються дані. Визначається на рівні мікросхеми пам'яті.

У випадку застосування лише фізичної адресації, адресні простори процесів розташовуються безпосередньо один за одним. Кожному процесу відповідає один неперервний діапазон адрес.

Проте у сучасних системах поверх фізичних адрес додається ще один рівень адрес — логічні, або віртуальні, адреси.

Віртуальні адреси набувають актуальності, наприклад, тоді, коли процесу знадобилася додаткова пам'ять на додачу до тієї, яку вже, було виділено йому під час створення. За умови такої адресації, як на рис. 5.1, виділити процесу пам’ять деінде поза межами його початкового адресного простору не вийде. І навпаки, виділеної пам'яті може виявитися забагато для даного процесу — а іншим процесам може тим часом не вистачати пам'яті. Змінити таку ситуацію за використання виключно фізичних адрес також немає змоги.

[18] Схему адаптовано з книги Tanenbaum, Bos. Modern Operating Systems, 2014

Застосування лише фізичних адрес також не підійде, коли котромусь із процесів потрібний обсяг пам'яті, що перевищує обсяг основної пам'яті на даному комп'ютері. У деяких випадках таку проблему можна було б вирішити шляхом залучення дискової пам’яті (підкачування), але як прив’язати ділянки на диску до адресного простору процесу в оперативній пам’яті з самими лише фізичними адресами також є проблемою.

Іншою проблемою залучення лише фізичних адрес є відстежування випадків, коли процес помилково звернувся до “чужої” адреси. Не вдасться реалізувати й коректне використання окремих блоків пам’яті кількома процесами — а це було б незайве, наприклад, для завантаження в такі блоки бібліотек.

Наостанок, у розділі 2 було розглянуто адресний простір процесу, в якому код зберігається окремо від даних. Однак якщо у нашому розпорядженні є лише фізичні адреси, то зробити такий поділ не вийде.

Таким чином і виникає потреба додати ще один рівень адрес поверх фізичних. Причому операційна система знову виступає як посередник (рис. 5.3) — цього разу між процесом (з його потребами десь зберігати свої код та дані, і якому неважливі деталі реалізації) та апаратним забезпеченням (з його комірками пам’яті у складі інтегрованої мікросхеми, під’єднаної до материнської плати).

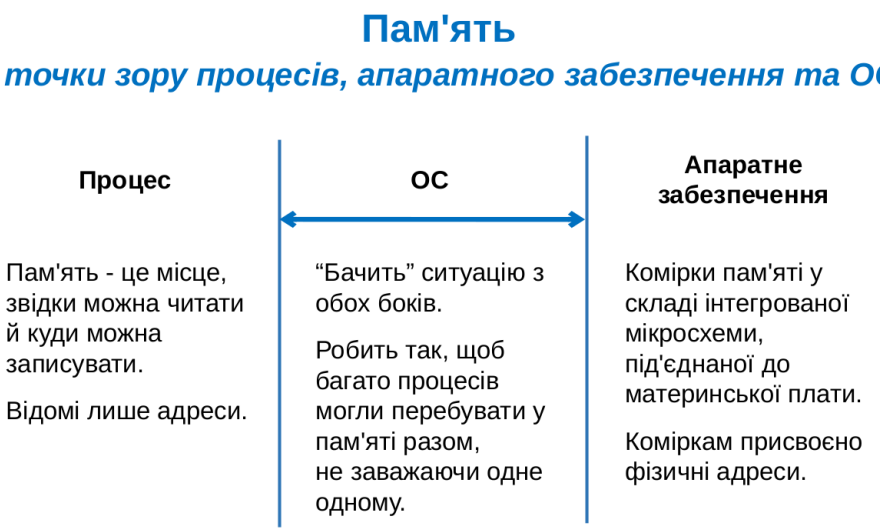

Рис. 5.3. Операційні система як посередник між процесом та фізичною пам’яттю

Суть віртуальної пам'яті полягає у наступному: фізичним адресам ставлять у відповідність віртуальні (логічні) адреси.

Віртуальна (логічна) адреса — умовна адреса у пам'яті, що у процесі роботи перетворюється у фізичну адресу.

Тобто процес звертається до віртуальних адрес, а ОС має надати засоби для перетворення цих адрес у фізичні — ті адреси, за якими насправді зберігаються дані на рівні мікросхеми пам’яті.

Перетворення логічних адрес у фізичні здійснює **MMU** (memory management unit, блок керування пам'яттю, пристрій керування пам'яттю). MMU є апаратним компонентом, який найчастіше входить до складу ЦП або чипсета.

Сукупність усіх віртуальних адрес називають **віртуальним адресним простором**, а сукупність усіх фізичних адрес — **фізичним адресним простором**.

Зверніть увагу, що вище ми паралельно використовуємо два терміни — “віртуальна адреса” і “логічна адреса”. У більшості випадків ці терміни можна використати як синоніми. Втім, у деяких випадках це різні поняття (див. *Сторінково-сегментна організація пам’яті*).

Далі у даному розділі йтиметься про різні підходи до організації пам'яті.

2. Підхід базового і межового регістрів

Підхід базового і межового регістрів є найпростішим підходом до організації віртуальної пам'яті. Підхід вже не є популярним, проте лежить в основі інших підходів, тож зрозумівши його, ви згодом легше сприйматимете сучасніші підходи.

У підході базового і межового регістрів кожному процесу відповідають два значення:

- база (фізична адреса, з якої починається адресний простір процесу);
- межа (кількість адрес, відведених процесу).

Для зберігання бази і межі відводяться два спеціальні регістри ЦП — *базовий і межовий регістри* (звідси й назва підходу).

Маючи логічну адресу і значення бази, можна обчислити фізичну адресу:

```
Фізична адреса = Логічна адреса + База
```

На рис. 5.4 наведено схематичне пояснення того, як відбувається визначення адреси у підході базового та межового регістрів.

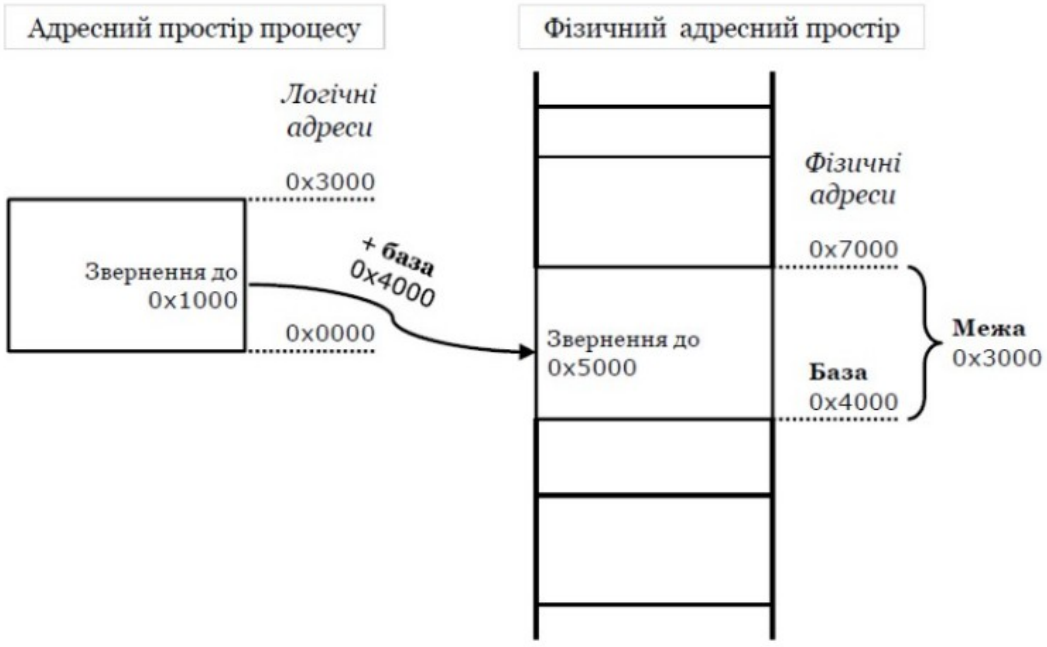

Рис. 5.4. Визначення адреси у підході базового і межового регістрів[19]

[19] Схеми на рис. 5.4-5.6, 5.8, 5,9, 5.11, 5.12 адаптовано за підручником В. А. Шеховцов “Операційні системи” (2005)

Як бачимо на рис. 5.4, процес працює з адресним простором від 0x0000 до 0x3000. Це логічні адреси, а на фізичному рівню їм відповідають адреси з 0x4000 по 0x7000. Коли процес звернувся до адреси 0x1000, треба перейти до фізичних адрес. Для цього до значення логічної адреси додаємо значення бази (0x4000). Одержуємо 0x5000 — це і є фізична адреса, до якої звернувся процес.

Найважливішою перевагою підходу базового і межового регістрів є простота його реалізації. Проте цей підхід не надає можливості розширення адресного простору процесу під час його роботи, можливості спільного використання адресного простору кількома процесами, а програмний код все ще не відокремлений від даних.

3. Сегментна організація пам’яті

Сегментна організація пам’яті також наразі нечасто використовується у чистому вигляді. Водночас, цей підхід надає значно більше можливостей, ніж підхід базового та межового регістрів.

Згідно з підходом сегментної організації пам'яті, віртуальний адресний простір ділять на сегменти. Сегменти нумеруються і можуть мати різний розмір.

Логічна адреса у випадку сегментної організації пам’яті задається:

- номером сегмента
- зсувом усередині сегмента

Відомості про всі сегменти процесу зберігаються у **таблиці сегментів**. У кожного процесу — своя таблиця сегментів. Також може існувати глобальна таблиця, яка зберігає інформацію про всі сегменти всіх процесів.

Основні відомості про сегмент у таблиці є наступними:

- номер сегмента;
- база (фізична адреса, з якої починається сегмент);
- межа (кількість адрес, відведених під сегмент);
- права доступу до сегменту.

На рис. 5.5 показано визначення адреси у випадку сегментної організації пам’яті.

У прикладі на рис. 5.5 процес звернувся до сегмента номер 3, а зсув всередині цього сегмента становить 256. Знаходимо відповідний сегмент у таблиці сегментів і з’ясовуємо що на рівні фізичних адрес він починається з бази 0x1000 і завершується адресою 0x1000 + 512. Також цей сегмент має права доступу rw, тобто процесу дозволено читати та записувати даний сегмент.

Перевіряємо, чи заданий зсув (256) не перевищує значення межі (512), а також чи запитуваний доступ до сегменту відповідає правам доступу до нього для даного процесу. І там, і там усе гаразд, тому далі відбувається перетворення логічної адреси у фізичну.

До значення бази для цього сегмента (0x1000) додається зсув (256). Одержуємо фізичну адресу 0x1000 + 256.

Зверніть увагу, що тут і далі додавання зсуву в окремих випадках ми не здійснювали. Це зроблено з метою збереження наочності, оскільки таке додавання потребувало б зведення доданків до однієї системи числення (перший доданок у шістнадцятковій системі, а другий — у десятковій), і одержана сума ілюструвала б хід перетворення адрес менш ефективно. Втім, за бажання можна здійснити необхідні перетворення в цьому та наступних прикладах — це буде гарним повторенням систем числення.

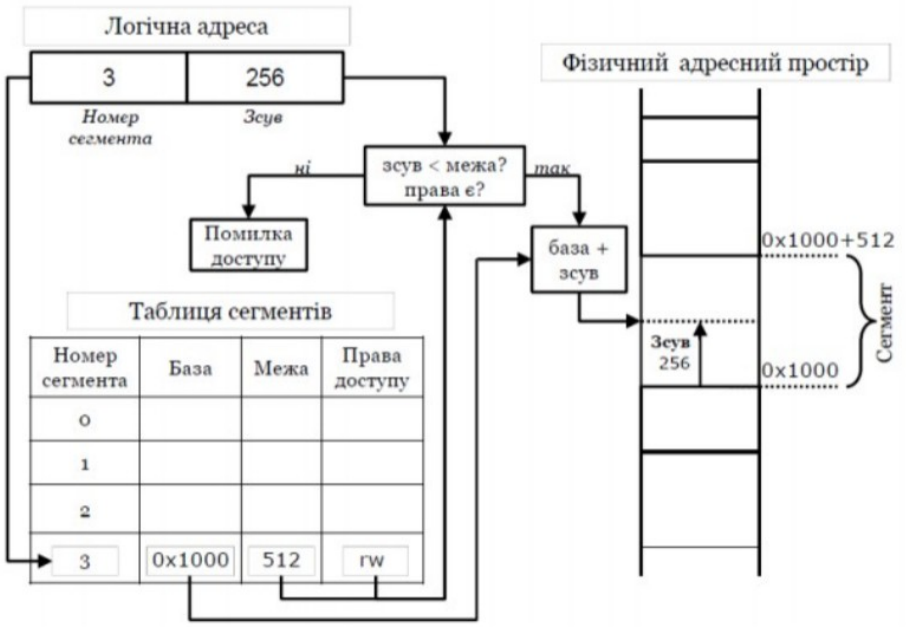

Рис. 5.5. Визначення адреси у випадку сегментної організації пам'яті

На рис. 5.6 продемонстровано, як співвідносяться віртуальний та фізичний адресний простір процесу у випадку сегментної організації пам'яті. Як бачимо, у фізичній пам'яті сегменти не обов'язково розташовані послідовно. Також між сегментами даного процесу можуть міститися інші блоки пам'яті, у яких розташовано сегменти інших процесів, або які поки вільні.

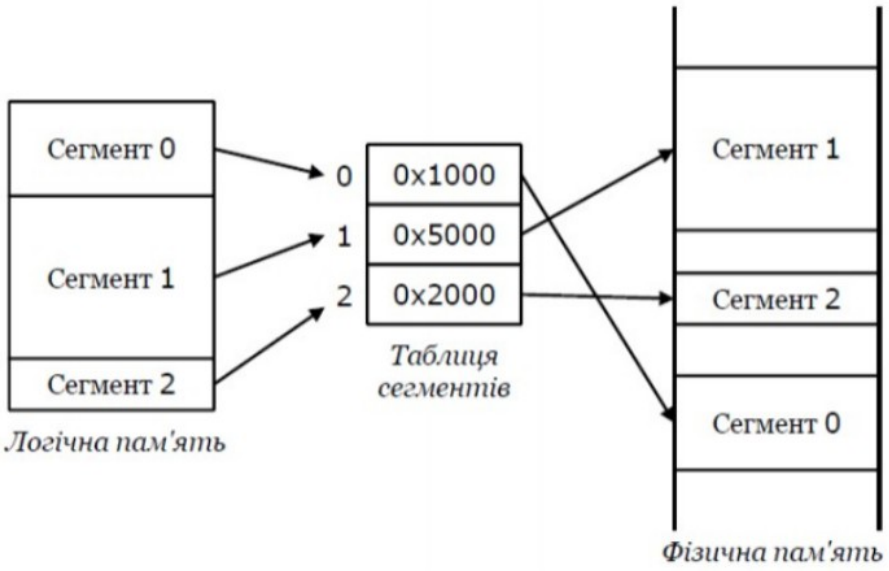

Рис. 5.6. Співвідношення віртуального і фізичного адресного простору (сегментна організація пам'яті)

Переваги сегментної організації пам'яті, порівняно з підходом базового і межового регістрів є помітними. У випадку сегментної організації пам'яті кілька процесів можуть спільно використовувати деякий сегмент, причому за потреби — з різними правами доступу до цього сегмента. Також з'являється можливість зберігати програмний код і дані процесу в окремих сегментах, чого

не було у підході базового та межового регістрів. І нарешті, частину сегментів можна тимчасово вивантажувати на диск, чим забезпечити підкачування.

Водночас, сегментній організації пам’яті притаманна низка недоліків. Зокрема, обмінюватися сегментами між диском та основною пам'яттю у межах механізму підкачування насправді не дуже зручно через різний розмір сегментів. Крім того, сегментній організації притаманна проблема під назвою *зовнішня фрагментація*.

Фрагментація в ОС трапляється не лише для основної пам’яті. Вона, наприклад, також може бути притаманна файловій системі, розташованій на дисковому накопичувачі. Проте у цьому розділів зосередьмося на фрагментації основної пам’яті.

Фрагментація — це явище, за якого у пам'яті з'являється велика кількість коротких несуміжних блоків, внаслідок чого система не в змозі виділити довгий блок, навіть якщо загальний обсяг пам'ять для цього достатній.

Загалом, фрагментація може бути зовнішня та внутрішня. Внутрішню фрагментацію буде розглянуто у п. 4, а зараз сфокусуємо увагу на зовнішній фрагментації, оскільки для сегментної організації пам’яті характерна саме вона.

Зовнішня фрагментація притаманна підходам до організації пам'яті з поділом адресного простору на частини змінної довжини — наприклад, сегменти. За зовнішньої фрагментації короткі несуміжні блоки лежать за межами адресних просторів процесів (тобто зовні — звідси й походить назва).

Зовнішня фрагментація виникає, коли деякий невеликий блок пам'яті вже звільнено, а наступний за ним блок — ще ні. На місці звільненого блоку виникає ***діра*** (рис. 5.7).

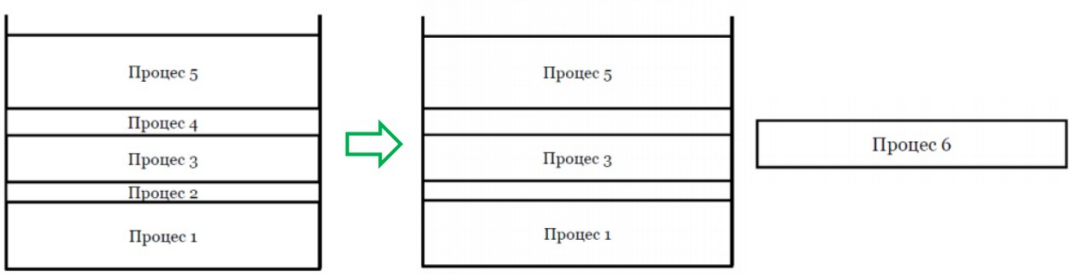

Рис. 5.7. Схематичне подання зовнішньої фрагментації
(сегментна організація пам’яті)

У випадку, зображеному на рис. 5.7, видно сегменти, які належать п’ятьом різним процесам — процесам 1-5. Імовірно, в цих процесів можуть бути іще сегменти в інших діапазонах адрес, не зображених тут, але для спрощення розглянемо лише ті, які бачимо.

На певному етапі процес 2 та процес 4 завершуються, і, відповідно, блоки пам’яті, де зберігалися їхні сегменти, звільняються. Новоствореному процесу 6 саме потрібний обсяг пам’яті, який дорівнює сумі обсягу двох звільнених блоків. Однак це буде не неперервний діапазон адрес, а два окремих сегменти. Таким чином, кількість коротких несуміжних блоків у пам’яті поступово зростає, а доступ до пам’яті для процесів ускладнюється.

4. Сторінкова організація пам’яті

На перший погляд, сторінкова організація пам’яті досить подібна до сегментної, проте сторінкова організація пам’яті має ключову відмінність, яка суттєво змінює подальше функціонування підходу. Якщо сегментна організація пам’яті працює з блоками змінного розміру (сегментами), то сторінкова організація пам’яті передбачає поділ адресного простору на блоки однакового розміру.

За сторінкової організації пам'яті віртуальний адресний простір ділять на ***сторінки***, а фізичний адресний простір на ***фрейми***.

І сторінки, і фрейми мають однаковий розмір, фіксований у межах системи. Розмір сторінки зазвичай коливається у межах від 512 б до 1 ГіБ та виражається степенем числа 2. Розмір сторінки в ОС залежить від загального обсягу оперативної пам’яті у цій системі, він споживання пам’яті типовими застосунками, які може бути встановлено у даній ОС, та від інших факторів — ми ще повернемося до питання розміру сторінки згодом у цьому розділі.

Логічна адреса у випадку сторінкової організації пам’яті задається:

- номером сторінки;
- зсувом усередині сторінки.

Відомості про сторінки процесу зберігаються у ***таблиці сторінок*** (подібно до того, як у випадку сегментної організації пам’яті відомості про сегменти процесу містяться у таблиці сегментів). Відомості про фрейми (зокрема, які фрейми зайняті, а які вільні) зберігаються у ***таблиці фреймів***.

Основні відомості про сторінку у таблиці є наступними:

- номер сторінки;
- номер відповідного фрейму;
- права доступу до сторінки.

Як бачимо, порівняно з сегментним підходом, серед перелічених відомостей відсутній розмір сторінки, адже він стандартний, і його немає сенсу зберігати для кожного процесу.

На рис. 5.8 схематично продемонстровано, як логічна адреса перетворюється у фізичну у випадку сторінкової організації пам’яті.

Як було і в ситуації з сегментною організацією пам’яті, процес звертається до адреси, використовуючи номер (цього разу номер сторінки) 3 та зсув 256. Відповідну сторінку знаходимо у таблиці сторінок і з’ясовуємо, що їй відповідає фрейм номер 2, а процесу дозволено читання та запис цієї сторінки.

Після перевірки прав доступу (вони наявні), логічна адреса перетворюється у фізичну. Номер фрейму (2) множимо на розмір сторінки (2^n) і додаємо зсув (256). Тобто на фізичному рівні сторінці номер 2 відповідає діапазон адрес від $2*2^n$ до $3*2^n$, а шукана фізична адреса буде $2^n + 256$.

Співвідношення віртуального та фізичного адресного простору процесу у разі використання сторінкової організації пам’яті може виглядати так, як зображено на рис. 5.9. Як було і з сегментами, сторінки теж розміщені у фізичній пам’яті не послідовно. І так само, між сторінками даного процесу можуть розташовуватися сторінки інших процесів або вільні сторінки.

Важлива відмінність полягає в однакових розмірах вільних блоків, що усуває проблему зовнішньої фрагментації.

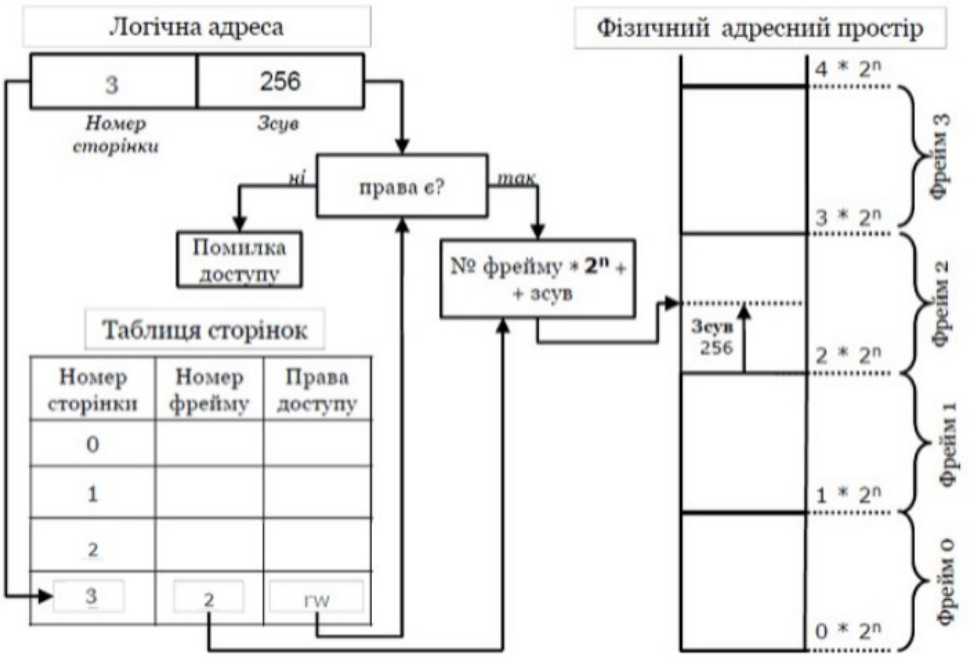

Рис. 5.8. Визначення адреси у випадку сторінкової організації пам’яті

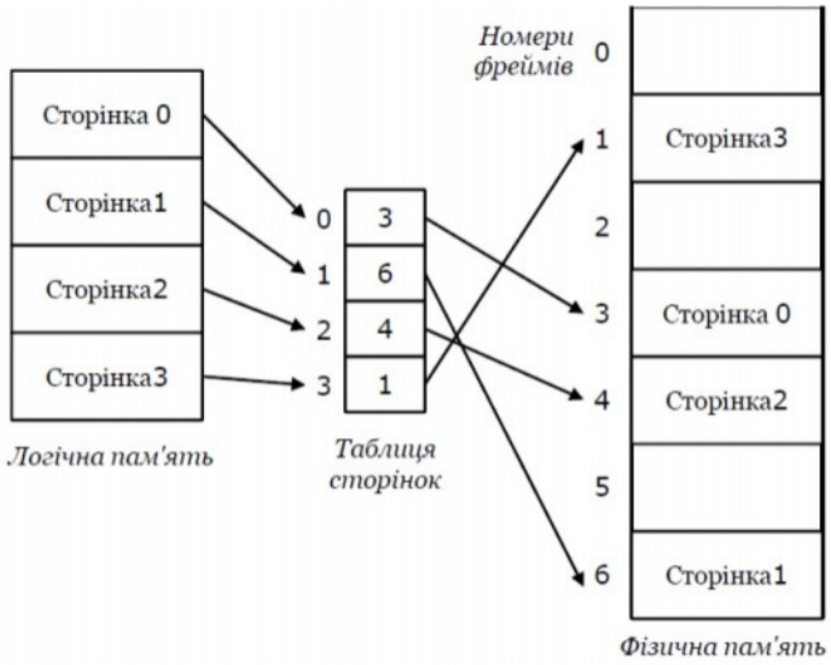

Рис. 5.9. Співвідношення віртуального і фізичного адресного простору (сторінкова організація пам’яті)

Таким чином, порівняно з сегментною організацією пам'яті, сторінкова організація пам’яті має переваги. Зокрема, зовнішня фрагментація як проблема усувається завдяки однаковим розмірам сторінок. Також сторінку процесу можна передати для використання з іншою метою (у випадку сегментів це було б незручно через змінний розмір сегментів). Через однаковий розмір сторінок спрощується й обмін даними з диском у межах підкачування.

Однак і сторінкова організація пам’яті має недоліки. Основним є складність підбору оптимального розміру сторінки, адже замалий чи завеликий розмір сторінки може призводити до проблем.

Якщо сторінки мають малий розмір, то таблиці сторінок у такій системі, навпаки, можуть розростатися до значних розмірів. Щоб краще зрозуміти це, уявіть собі, що зігнули кілька аркушів паперу формату А4 удвоє і склали їх стосиком. Одержаний стосик вийде не надто високим. Водночас, якщо ті самі аркуші скласти вчетверо і знову поскладати їх один на одний, то цього разу стосик вийде вже вищий — і т.д., що більше складань робитимемо, то вищий стосик отримуватимемо, адже маленьких аркушів у ньому ставатиме дедалі більше.

Водночас, завеликий розмір сторінки призводить до *внутрішньої фрагментації*.

Внутрішня фрагментація характерна для підходів до організації пам'яті з поділом адресного простору на частини фіксованої довжини — зазвичай сторінки. У випадку внутрішньої фрагментації короткі несуміжні блоки лежать всередині адресних просторів процесів (виникає *внутрішньо* — звідси й назва).

Внутрішня фрагментація виникає, коли через фіксований розмір сторінок неможливо виділити черговий блок пам'ять менший за розмір сторінки (рис. 5.10). Може статися, що чергова сторінка використовується процесом лише частково (процесу стільки не потрібно). Водночас, інші процеси теж не зможуть скористатися вільним місцем, адже це не їхня сторінка.

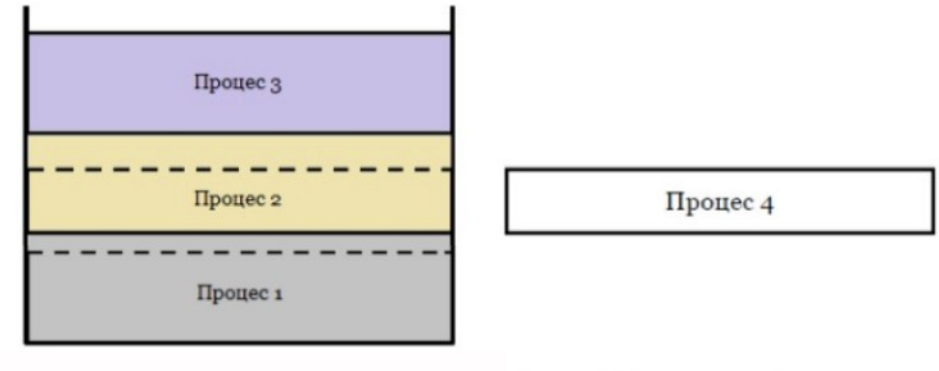

Рис. 5.10. Схематичне подання зовнішньої фрагментації (сторінкова організація пам’яті)

На рис. 5.10 показано сторінки трьох процесів. Як і у прикладі зовнішньої фрагментації, це не обов’язково єдині їхні сторінки, але для демонстрації їх буде достатньо. Процес 1 і процес 2 використовують свої сторінки лише частково — простір угорі сторінки, відділений пунктирною лінією, лишається незадіяним. Новоствореному процесу 4 потрібно стільки ж пам’яті, скільки не використовують у межах своїх сторінок процес 1 і процес 2 в сумі. Але ми не маємо права віддати цю пам’ять процесу 4, оскільки сторінка є неподільною.

У сучасних ОС сторінкова організація пам’яті використовується найчастіше, проте вказані недоліки важливо враховувати. Маємо наступну ситуацію: надто великі сторінки призводять до внутрішньої фрагментації, а надто малі сторінки — до великих таблиць сторінок і, як наслідок, до повільного пошуку потрібної сторінки.

Для того, щоб впоратися з описаними проблемами, розробники ОС вживають наступних заходів:

- намагаються підібрати оптимальний розмір сторінки залежно від умов роботи розроблюваної системи;
- створюють кеш з відомостями про сторінки, які використовуються найчастіше;
- створюють ієрархію сторінок для зручнішого доступу (на зразок змісту книги);
- поєднують сторінки з сегментами тощо.

Часто перелічені вище ідеї комбінують між собою. У цьому пункті буде розглянуто перших три ідеї, а четверту — у п. 5.

Оптимальний розмір сторінки залежить від того, скільки загалом оперативної пам'яті встановлено на даному комп'ютерному пристрої. Звісно, недоцільно вводити великі сторінки, скажімо, на невеликому пристрої з 512 Mb оперативної пам'яті. І навпаки, на потужному сервері з кількома десятками гігабайтів оперативної пам'яті та процесами, які типово споживають великі обсяги пам'яті, варто використовувати більші сторінки.

Багаторівневі таблиці сторінок. Якщо адресний простір процесу великий, то кількість сторінок теж вийде великою і може сягати мільйонів елементів, що значно уповільнить пошук потрібної сторінки. Одним із рішень можуть бути багаторівневі таблиці сторінок.

У випадку багаторівневих таблиць сторінок є декілька рівнів сторінок (небагато, зазвичай 2, іноді 3). Таблиці сторінок нижнього рівня розбиваються на сторінки, а інформація про сторінки нижнього рівня міститься у таблиці сторінок верхнього рівня (наприклад, в архітектурі IA-32 — у каталозі сторінок). Приклад багаторівневих таблиць сторінок буде продемонстровано у п. 5 у поєднанні зі сторінково-сегментною організацією пам'яті.

Асоціативна пам'ять. Зазвичай до одних сторінок процес звертається частіше, ніж до інших. Якщо вибрати сторінки, до яких процес звертається частіше, і зберігати відомості про них окремо, то пошук потрібної сторінки може стати швидшим.

Такі сторінки зберігають у спеціальний кеш — **кеш трансляції** (TLB, translation lookaside buffer), або ж **асоціативну пам'ять**.

Основні відомості про сторінку у кеші TLB є наступними:

- номер сторінки;
- номер відповідного фрейму.

На рис. 5.11 показано, як визначається адреса у випадку поєднання сторінкової організації пам'яті та кешу трансляції.

Спершу сторінку номер 3 шукаємо у кеші. Якщо сторінка у кеші є, то відбувається *влучання кешу трансляції*. Тоді за одержаним із кешу номером відповідного фрейму можна одразу з'ясовувати фізичну адресу, яка відповідає заданим сторінці та зміщенню. Коли ж сторінки у кеші не виявиться, то матиме місце *промах кешу трансляції*, і далі пошук потрібної сторінки здійснюватиметься у таблиці сторінок. Зрештою сторінку все одно має бути знайдено, але перед перетворенням адреси у фізичну також потрібно буде перевірити права доступу.

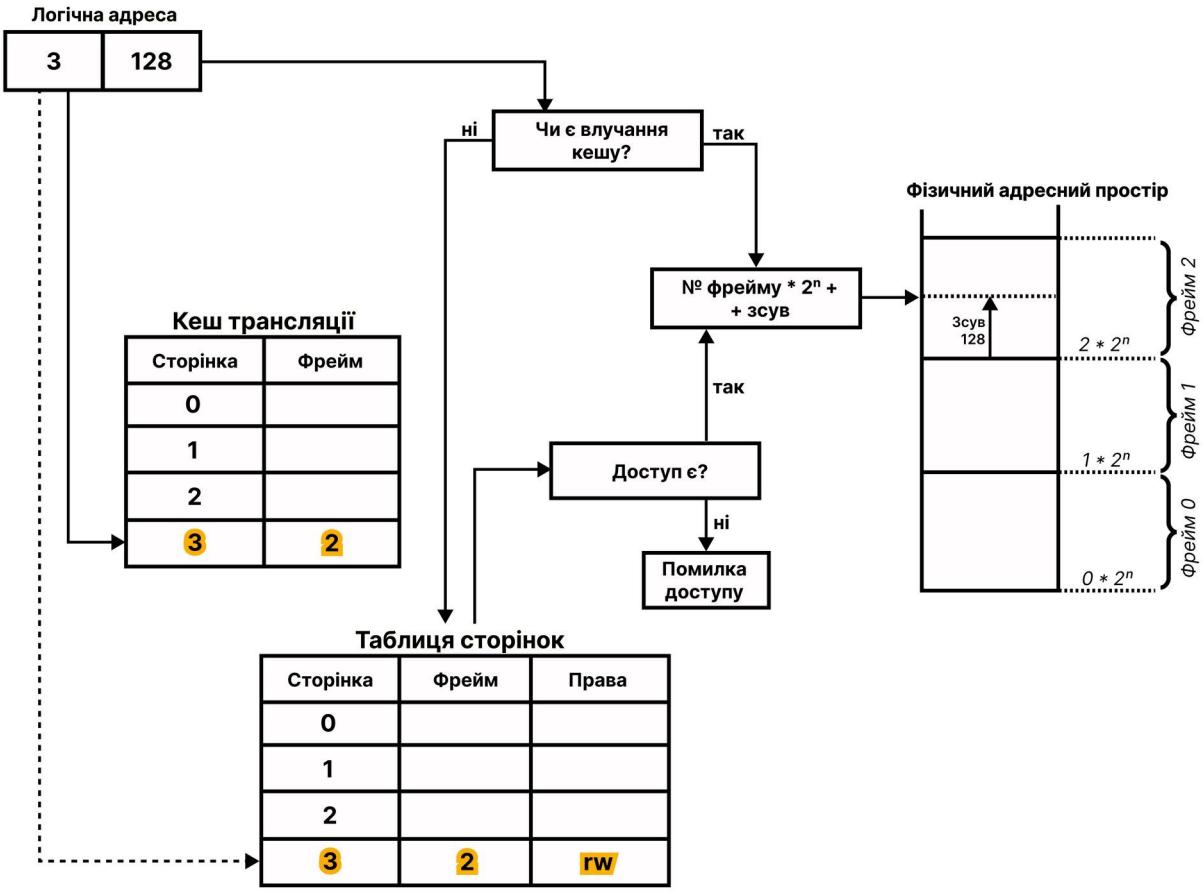

Рис. 5.11. Визначення адреси за участю кешу трансляції (сторінкова організація пам’яті)

Термін “асоціативна пам’ять” (як синонім кешу трансляції) може на перший погляд видаватися нелогічним. Однак він випливає з особливостей пошуку у кеші трансляції. Загалом, можна шукати за індексом (наприклад, у таблиці сторінок номери — це індекси), а можна шукати за вмістом деякого поля (у кеші TLB номери сторінок не є індексом, бо у кеш потрапляють не всі сторінки, а отже, нумерація не наскрізна). Пошук за вмістом, а не за індексом, і є асоціативним.

5. Сторінково-сегментна організація пам’яті

Якщо поєднати сегментну і сторінкову організацію пам'яті, вийде наступне.

Матимемо дві віртуальні адреси: одна з сегментного підходу, інша — зі сторінкового. Назви будуть відрізнятися. Наприклад, в IA-32 це:

- логічна адреса (відповідно до сегментного підходу);
- лінійна адреса (відповідно до сторінкового підходу).

Перетворення адреси здійснюється у наступній послідовності:

1. ЦП звертається до логічної адреси.
2. Модуль сегментації перетворює логічну адресу на лінійну адресу.
3. Сторінковий модуль перетворює лінійну адресу на фізичну адресу.

На рис. 5.12 наведено приклад перетворення адреси у випадку сторінково-сегментної організації пам’яті.

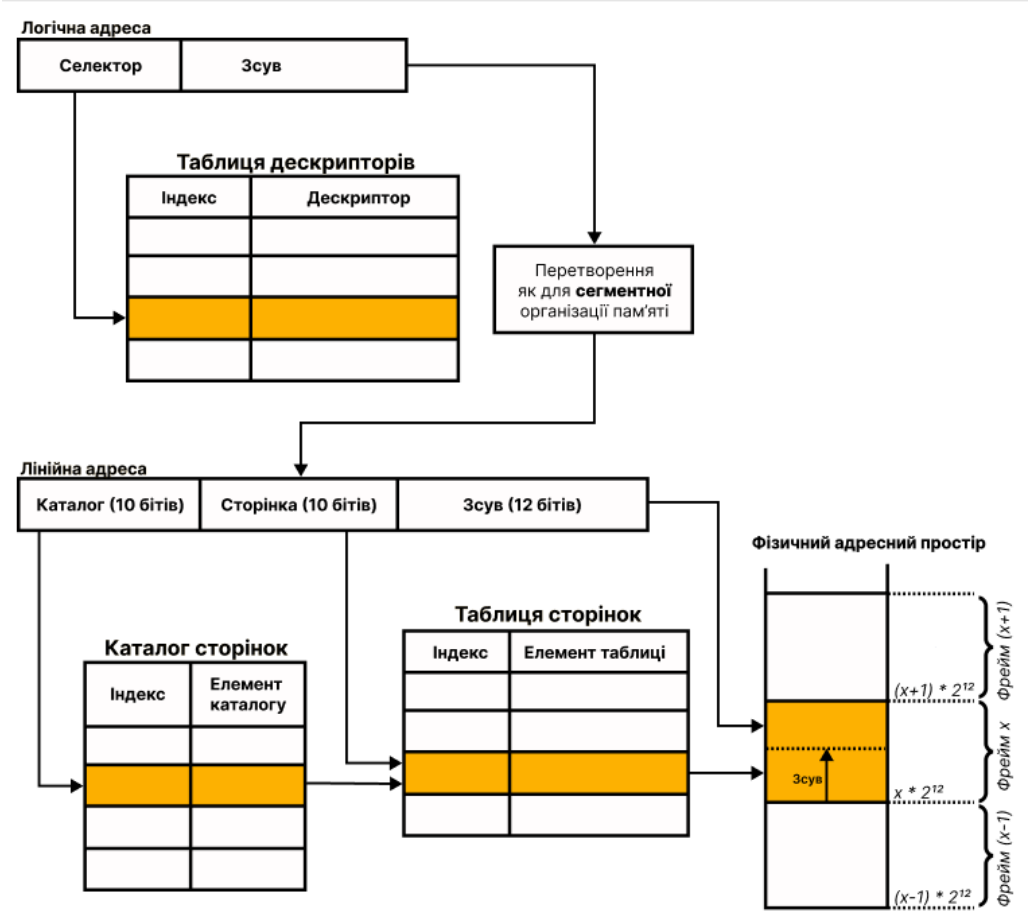

Рис. 5.12. Визначення адреси у випадку сторінково-сегментної організації пам’яті (на прикладі IA-32)

Спершу відбувається перетворення логічної адреси у лінійну. Це проходить за правилами сегментної організації пам’яті, причому залучається таблиця дескрипторів. У сучасних системах цей етап може бути умовним і зберігатися задля сумісності з попередніми системами [4].

Головне перетворення стосується переходу від лінійної адреси до фізичної адреси, яке здійснюється за сторінковим підходом. Тут також використано дворівневі таблиці сторінок: є каталог сторінок, а є таблиці сторінок. Тому до складу лінійної адреси входить індекс потрібного каталогу, індекс сторінки у межах цього каталогу, а також зсув всередині сторінки. Поділ фізичного простору на фрейми виглядає подібно до такого поділу за суто сторінкового підходу.

6. Підкачування. Заміщення сторінок

Підкачування (swapping, свопінг) — технологія передачі сторінки з основної пам'яті на диск і навпаки.

Підкачування можливе і для сегментів, але найчастіше реалізоване для сторінок. Тому надалі говоримо про сторінки.

Якщо ви спробуєте дослідити обсяги підкачування у власній системі, то можете одержати досить різноманітні результати, які залежатимуть і від особливостей підкачування ОС даного сімейства, і від його налаштувань саме у вашій ОС, і від завантаженості системи у момент спостереження. Тому за різних обставин ви можете спостерігати і нульовий розмір підкачування, і підкачування у значних обсягах.

Чого можна досягнути завдяки підкачуванню. За умови збалансованого використання, підкачування здатне забезпечити чимало переваг у сучасних ОС. Приміром, було б добре мати більше процесів, запущених одночасно, але так, щоб вони не займали оперативну пам'ять без потреби — це надало б більшу пропускну здатність та більший % використання ЦП. Також важливо зазначити, що значна частина сторінок процесу використовується дуже рідко, тож недоцільно тримати їх усі в оперативній пам'яті. Оперативна пам'ять енергозалежна, зберігання у ній зайвих даних означає більші витрати не лише місця, а й енергії.

У межах механізму підкачування в основній пам'яті тримається лише частина сторінок (***резидентна множина***). В ідеалі до резидентної множини включають найчастіше затребувані сторінки. Решта сторінок зберігається на диску.

Якщо під час виконання процес звертається до сторінки, що на даний момент зберігається на диску, то виконується наступна послідовність кроків:

1) відбувається сторінкове переривання (помилка відсутності сторінки, сторінкова відмова);

2) виконання процесу призупиняється;

3) потрібна сторінка завантажується з диску в основну пам'ять;

4) виконання процесу продовжується.

Щоб звільнити місце в основній пам'яті для нової сторінки, якусь іншу сторінку може знадобитися вилучити з основної пам'яті (вивантажити на диск). Є різні алгоритми вибору такої сторінки, їх називають ***алгоритмами заміщення сторінок***. У цьому посібнику здійснено короткий огляд базових алгоритмів. Більше про них можна дізнатися з джерел, наведених у кінці розділу.

☆ **Оптимальний алгоритм.** Кожна сторінка має спеціальну позначку — кількість команд, що лишилися до моменту, коли процес звернеться до цієї сторінки. Тоді вилучати треба ту сторінку, звернення до якої відбудеться найпізніше.

У теорії ОС цей алгоритм приймається за еталон ефективності, але на практиці його неможливо реалізувати, оскільки в ОС відсутня інформація для створення необхідних позначок. Тому оптимальний алгоритм використовується для оцінювання ефективності інших алгоритмів — інші алгоритми порівнюють з оптимальним.

☆ **Алгоритм FIFO (First In, First Out).** Вилучається сторінка, яку було завантажено в основну пам'ять першою. Реалізувати алгоритм заміщення сторінок FIFO нескладно, але цей алгоритм може призвести до вилучення сторінки, що часто використовується.

Алгоритм FIFO має невисоку ефективність і у чистому вигляді застосовується рідко.

☆ **Алгоритм “другий шанс” (second chance algorithm).** Сторінки зберігаються у списку, відсортованому за часом надходження сторінки в

основну пам'ять. Якщо до першої сторінки у списку нещодавно зверталися, її не вилучаються, а переміщують у кінець списку. Замість неї перевіряють сторінку, яка була у списку другою, і т. д.

Алгоритм “другий шанс” мінімізує ризик вилучити активно використовувану сторінку. Водночас, цей алгоритм дуже повільний через постійні переміщення сторінок у списку.

☆ **Алгоритм “годинник” (clock algorithm)** подібний до алгоритму “другий шанс”, але використовує циклічний список. Найстаріша сторінка у списку (кандидат на вилучення) позначена вказівником. Якщо сторінці дають другий шанс, вказівник пересувається на наступну сторінку.

Завдяки відсутності переміщень елементів у списку, алгоритм “годинник” значно ефективніший за звичайний “другий шанс”.

☆ **Алгоритм “покращений другий шанс” (enhanced second-chance algorithm; NRU algorithm — Not Recently Used).** Резидентна множина сторінок ділиться на 4 класи. Класи є наступними:

1) Клас (0, 0). Сторінки, до яких останнім часом не було звернень і які не модифікувалися.

2) Клас (0, 1). Сторінки, до яких останнім часом не було звернень, але їх було модифіковано (раніше).

3) Клас (1, 0). Сторінки, до яких останнім часом були звернення, але які не модифіковувалися (взагалі).

4) Клас (1, 1). Сторінки, до яких останнім часом були звернення і які було модифіковано.

Вилучається довільна сторінка з найнижчого непорожнього класу.

Алгоритм “покращений другий шанс” особливо ефективний у поєднанні з ”годинником”.

☆ **Алгоритм LRU (Least Recent Used algorithm).** Вилучається сторінка, що використовувалася найдавніше.

За ефективністю алгоритм LRU близький до оптимального, але, як і оптимальний, є дуже складним у реалізації. Однак є наближення до алгоритму LRU (наприклад, алгоритм старіння) [3].

☆ **Алгоритм “робочий набір” (work set algorithm).** До робочого набору включають ті сторінки, які останнім часом використовувалися найчастіше. Даний алгоритм складний у реалізації і ресурсомісткий.

☆ **Алгоритм WSClock** є поєднанням алгоритмів “робочий набір” та “годинник”. Список у алгоритмів WSClock використовується такий, як у годиннику, тобто циклічний з рухом вказівника. Водночас, замість сортування за часом завантаження сторінку у пам'ять список сортується за часом останнього використання сторінки. Таке поєднання алгоритмів забезпечує простішу реалізацію, ніж у “робочому наборі” й вищу ефективність, ніж у звичайному “годиннику”.

Контрольні запитання

1) Яку адресу в оперативній пам’яті називають фізичною?
2) Обґрунтуйте потребу введення логічних адрес поверх фізичних.
3) Опишіть роль операційної системи як роль посередника між процесом та фізичною пам’яттю.
4) Яку адресу в оперативній пам’яті називають логічною (віртуальною)?
5) Поясніть роль пристрою керування пам’яттю (MMU) в організації пам’яті.
6) Що таке віртуальний адресний простір? фізичний адресний простір?

7) Дайте загальну характеристику підходу базового та межового регістрів. Яким чином здійснюється перетворення логічної адреси у фізичну? Які переваги та недоліки має цей підхід?
8) Дайте загальну характеристику сегментної організації пам’яті. Яким чином здійснюється перетворення логічної адреси у фізичну? Які переваги та недоліки має цей підхід?
9) Що таке фрагментація (у випадку оперативної пам’яті)? Поясніть суть явища зовнішньої фрагментації.
10) Дайте загальну характеристику сторінкової організації пам’яті. Яким чином здійснюється перетворення логічної адреси у фізичну? Які переваги та недоліки має цей підхід?
11) Поясніть суть явища внутрішньої фрагментації.
12) Які ідеї використовують розробники ОС для вирішення проблем сторінкової організації пам’яті?
13) Поясніть, як здійснюється перетворення адрес у випадку використання сторінкової організації пам’яті з кешем трансляції.
14) Дайте загальну характеристику сторінково-сегментної організації пам’яті. Яким чином здійснюється перетворення логічної адреси у фізичну?
15) Що таке підкачування (swapping)? Які потенційні переваги має підкачування?
16) Перерахуйте та охарактеризуйте основні алгоритми заміщення сторінок у межах підкачування.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 9.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 7-8.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 3.
4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділи 8-9

Розділ 6

Файлові системи

1. Роль та рівні організації файлової системи

У розділі 1 вже йшлося про те, що ОС забезпечує зручні абстракції, якими прикладні програми і користувачі можуть послуговуватися для взаємодії з апаратними складовими комп'ютерними системами. Тоді у ролі першого прикладу такої абстракції було наведено файл та інші абстракції, які надає файлова система.

Згодом ми розглянули й інші абстракції. На рис. 6.1 наведено найбільш базові абстракції, на які спирається у своїй роботі ОС: процеси та потоки, адресні простори та файли.

Рис. 6.1. Базові абстракції в операційній системі

Абстракції на рис. 6.1 подано у вигляді трьох китів, і, на відміну від архаїчних уявлень стародавніх людей про Землю, наші кити — це просто образ для розуміння важливості цих абстракцій, оскільки на них зрештою так чи інакше спираються усі інші механізми в ОС.

Процеси і потоки було детально розглянуто у розділах 1-4, адресні простори частково досліджено у розділі 2 та більш повно — у розділі 5. Настав час з'ясувати, як організовано файлові системи.

Файлова система виконує наступні базові функції:

- організовує зберігання даних на дисках та інших носіях;
- надає доступ до цих даних через абстракцію — файл.

Файлові системи суттєво різняться за будовою та функціонуванням. Але загалом у файлових системах можна виокремити два рівні:

- **фізичний рівень** (має справу з блоками даних на носіях);
- **логічний рівень** (має справу з файлами).

Межі цих рівнів та їх компонентний склад відрізняються залежно від конкретної реалізації. Та незалежно від деталей реалізації, зв'язок між рівнями пролягає там, де розрізнені блоки інтерпретуються як один файл. У цьому розділі більше йтиметься про логічний рівень. Більше про фізичний рівень організації файлової системи можна з наведених наприкінці розділу джерел.

На практиці дані файлу не обов'язково зберігаються на диску у вигляді цілісного блоку. Часто це розрізнені блоки, і в один файл їх об'єднуються спеціальні структури даних.

Файлову систему можна уявити як об'єднання трьох великих складових, пов'язаних між собою:

- файли;
- службові структури даних для зберігання відомостей про файли;
- системні програмні засоби для виконання операцій над файлами (система керування файлами — file management system).

Тобто частиною файлової системи є й самі файли разом з даними в них; структури даних, що визначають, яким саме чином у системі зберігатимуться атрибути та дані файлів; система керування файлами, що забезпечує можливість виконувати операції над файлами (створювати, видаляти, копіювати, переміщувати тощо).

2. Основні поняття файлової системи

Існують різні означення поняття "файл". Передусім визначимося, що нас цікавлять лише ті файли, які зберігаються у комп'ютерних системах — скажімо, файли як справи працівників у паперовій формі не є предметом розгляду в теорії операційних систем. Файл зазвичай містить якісь дані, і можливість мати їх є невід'ємною для файлу — навіть коли наразі файл порожній. Залежно від операційної та файлової системи, файл може мати різноманітні атрибути (властивості), але один атрибут є обов'язковим — це ім'я файлу. І наостанок, для файлу має значення, у якій послідовності мають читатися його дані. Текстовий файл, у якому два символи поміняли місцями — це вже інший файл. Тобто файл

також є впорядкованим (дані мають фіксований порядок, не плутати з відсортованим).

Враховуючи сказане вище, сформулюємо робоче означення файлу, на яке спиратимемося далі у цьому розділі.

Файлом називають впорядкований набір даних, що зберігається у комп'ютерній системі під спільним ім'ям.

У підручнику Столлінґса [2] викоремлено ***очікувані характеристики файлів***, які є наступними.

- ***Довготермінове існування.*** Файли лишаються на носії після відключення живлення.
- ***Спільне використання кількома процесами.*** Щоб різні процеси могли звернутися — потрібне ім'я. Щоб доступ був безпечний, потрібне розмежування доступу.
- ***Структурованість.*** Файл може мати спеціальну внутрішню структуру залежно від типу. Файли можуть організовуватися у різноманітні структури, які показують зв'язки між цими файлами (наприклад, ієрархічна структура).

Синонімом до згаданого вище файлового атрибута є *властивість* (згідно з термінологією теорії систем, *суттєва* властивість). Загалом, терміни "файловий атрибут" та "властивість файлу" можна використовувати як взаємозамінні.

Серед типових атрибутів файлів є, зокрема, наступні:

- ім'я;
- тип;
- розмір;
- атрибути безпеки (власник / власники, повноваження тощо);
- часові атрибути (дата і час створення, дата і час останньої модифікації, дата і час останнього доступу тощо);
- інші атрибути.

Вимоги до імен файлів відрізняються залежно від системи. Відмінності виявляються зокрема у чутливості до регістру символів, ролі розширення файлів, максимально допустимій довжині імені файлу.

Чутливість до регістру символів в іменах файлів, простіше кажучи, виявляється в тому, чи розрізняється велика та мала літери в іменах файлів. Наприклад, у файловій системі Ext4 (типова для багатьох Linux-систем) можна створити в одній папці два файли, імена яких будуть відрізнятися лише регістром символів (наприклад, `file1.txt` та `File1.txt`). Це буде два окремих файли. Натомість у файлові системі NTFS (типова для Windows) аналогічна спроба створення двох файлів `file1.txt` та `File1.txt` в одній папці завершиться помилкою, оскільки імена сприйматимуться як однакові. Зовні може здаватися, що NTFS також розрізняє регістр символів, оскільки користувач може використовувати великі літери в іменах файлів. Однак на нижчому рівні регістр символів в NTFS таки не розрізняється.

Роль розширення файлу також може відрізнятися залежно від системи. Загалом, розширення покликане допомогти розрізняти файли різних типів. Водночас, важливо розуміти, що в жодній системі розширення не є єдиним критерієм, за яким формується тип файлу. Адже, певна річ, не можна зробити з

текстового файлу виконуваний файл Windows, просто змінивши розширення на exe, адже внутрішня структура файлу від цього не зміниться.

Водночас, розширення важливіше в одних системах й менш важливе в інших. Наприклад, традиційно, у Windows розширення відіграє помітнішу роль, ніж у Linux. Хоч у сучасних настільних версіях Linux розширення і впливає на низку "користувацьких" властивостей файлу. Зокрема, розширення визначає типові програми, які потрібно використовувати для відкриття файлу, на значок, що відображається біля файлу тощо. Водночас, багато файлів у Linux і далі не має жодного розширення. Особливо це стосується низки конфігураційних файлів та файлів, робота з якими здійснюється переважно з командного рядка.

Розширень може б.ути декілька. Наприклад, у Linux системах поширеною є ситуація, коли файли архівів мають подвійне розширення (`.tar.gz`, `.tar.xz` тощо). Такі розширення в архівів допомагають зберігати інформацію про те, яку кількість файлів було збережено в архіві, чи було застосовано до них стиснення, і якщо так, то яке.

Максимально допустима довжина імені файлу у сучасних системах зазвичай складає 255 символів (або 256 — залежно від того, як рахувати символ кінця рядка). Натомість для старіших систем використовувався так званий стандарт 8.3 (три — максимальна довжина розширення). У сучасних системах такий стандарт також може підтримуватися задля сумісності. Стандарт 8.3 досі можна зустріти у деяких вбудованих системах (так, авторка цього посібника свого часу користувалася MP3-плеєром Canyon з підтримкою лише коротких імен файлів, тимчасом як у настільних ОС вже давно використовувалися довгі імена файлів).

Каталоги. Лише дуже прості файлові системи не використовують каталогів. Це означає, що всі файли у такій системі будуть зберігатися на одному ієрархічному рівні (уявіть, що не можете створити жодної папки на флешці).

В англомовних джерелах **каталогу** зазвичай відповідає термін "**directory**", тому українською його іноді також перекладають як **директорія** (цей термін ви часто можете зустріти у комп'ютерній літературі 1990-х, наприклад). Інший популярний термін — **папка** (англійською — **folder**, ще один україномовний варіант — **тека**). Існує певна традиційна прив'язка каталогів до Unix/Linux-систем, а папок — до Windows-систем, але на практиці в сучасних ОС можна зустріти і папки/теки в Linux, і каталоги у Windows. Тому усі ці терміни (окрім, можливо, директорії, яка трохи збиває з пантелику) варто використовувати як синоніми.

На каталог можна дивитися з двох позицій:

1) каталог — це група файлів

2) каталог — це особливий файл, що зберігає відомості про деяку групу файлів.

З урахуванням того, що насправді каталог — такий своєрідний "файл про файли", добре знайома логічна структура каталогів у вигляді дерева починає видаватися ілюзією. Але тут варто дотримуватися погляду "я можу з цим працювати — отже, це існує". По суті, логічна структура каталогів є прикладом віртуалізації в широкому сенсі. Так, віртуальної машини, наприклад, теж не існує (адже немає комп'ютера саме з такими системними параметрами), але на цю віртуальну машину можна встановити операційну систему, і більшість програм працюватимуть на ній без змін.

Таким чином, обидві позиції, з яких можна дивитися на каталог, правильні, і застосування першої чи другої залежить від ситуації. З точки зору вивчення деталей роботи файлової системи, корисніша друга позиція (каталог — це

особливий файл), а з точки зору використання структури каталогів для щоденної роботи з ними — перша позицію (каталог — це група файлів).

Також відтепер у межах цього посібника ми іноді *називатимемо файлом те, що насправді може бути як файлом, так і каталогом* — у випадках, коли не важливо, про що насправді йдеться.

Абсолютні та відносні імена. *Абсолютне ім'я файлу* ще називають повним. Воно починається з дискового тому (Windows) або зі знаку кореневого каталогу / (Linux) і завершується ім'ям кінцевого файлу або каталогу, який мається на увазі.

Приклади абсолютних імен файлу:

`C:\users\usr1\dir1\file5.txt` файлова систем **NTFS** (ОС Windows)

`/home/usr1/dir1/file5.txt` файлова система **ext4** (ОС Linux)

Зверніть увагу: обидва файли називаються `file5.txt`, обидва містяться у так званих домашніх каталогах (домашніх папках) користувача `usr1`. Натомість, відрізняється напрям слешів та стандартна структура каталогів.

Відносне ім'я файлу також називається скороченим. Втім, це не обов'язково лише ім'я файлу разом з розширенням. Відносне ім'я файлу також може містити частину шляху до файлу — просто не весь шлях. Тут у пригоді стає саме назва "*відносне* ім'я файлу", бо таке ім'я визначається *відносно* певного каталогу, нерідко — поточного каталогу.

Наприклад, якщо у попередньому прикладі поточним є каталог `/home/usr1/dir1`, то матимемо наступну ситуацію:

абсолютне ім'я файлу: **`/home/usr1/dir1/`**`file5.txt`

відносне ім'я файлу: `file5.txt`

Коли ж поточним буде каталог `/home/usr1`, відносне ім'я файлу можна буде записати вже відносно нього, і воно відрізнятиметься:

абсолютне ім'я файлу: **`/home/usr1/`**`dir1/file5.txt`

відносне ім'я файлу: `dir1/file5.txt`

Розуміння абсолютних і відносних імен файлів в ОС надзвичайно важливе для роботи у командному рядку, створення скриптів та програмних засобів для цієї ОС.

Журналювання. Сучасні файлові системи часто ***журнальовані*** (journaling file systems). У журнальованій файловій системі зміни, що вносяться у файлову систему, розглядаються як транзакції. Транзакції мають бути виконані до кінця або не виконані зовсім, оскільки транзакція, виконана лише частково, може порушити цілісність файлової системи.

Таким чином, якщо під час транзакції трапився збій, то відповідна операція потім:

а) поновлюється,

б) скасовується (відкочується),

але не виконується частково (бо це б порушило цілісність файлової системи).

Однак, можливість поновлення або відкочування потребує фіксування певних даних про виконувану операції з файлом. Причому це фіксування має відбутися до того, як розпочнеться спроба виконати операцію, адже в ході виконання може статися будь-що.

Для фіксування операцій над файловою системою перед тим, як виконувати ці операції, використовується ***журнал***.

Важливо розуміти, що в ОС можуть бути й *інші журнали* — вони необхідні для забезпечення безпеки ОС та для підтримки й відновлення працездатності

ОС та окремих її програм. Однак про інші журнали йтиметься у розділі 7, коли розглядатиметься ***системний аудит***.

Дискові томи. Загалом, **том** (volume) — це набір секторів на накопичувачі, використовуваний операційною системою для зберігання даних.

Однак з наведеного означення неможливо сказати, чи на одному носії розміщено ці сектори. І справді, у сучасних ОС існує поняття віртуальних томів — вони можуть фізично міститися на різних накопичувачах. Це, зокрема, використовують на серверах для збільшення дискового простору, аби уникнути перевстановлення системи та повторного налаштування програм та критично важливих сервісів.

Отже, можливі, зокрема, наступні ситуації:

- один диск — це один том;
- один диск — це кілька томів (кожний розділ є окремим томом);
- кілька дисків — це один том.

Дискові томи також називають *дисковими розділами*. Втім, розділи частіше асоціюються з частинами одного накопичувача, тимчасом як томи є більш універсальним поняттям.

3. Приклади файлових систем

У даному посібнику ми не ставимо мети розглянути чи принаймні перерахувати всі файлові, оскільки файлових систем та їхніх модифікацій існує величезна кількість. Та для того, аби мати загальну картину, у ролі прикладів файлових систем назвемо вже історичну файлову систему **UFS** (Unix File System), яку покладено в основу сучасних файлових систем для Unix-подібних ОС, зокрема Linux, macOS, Android; сімейство файлових систем для Linux під назвою Ext (найновіша версія — Ext4); файлові системи для macOS в різних її інкарнаціях (HFS, HFS+ / Hierarchical File System, APFS / Apple File System); сімейство файлових систем FAT для старіших Windows-систем (FAT12, FAT16, FAT32); шифрована файлова система від Microsoft EFS (Encrypting File System); файлова система від Sun Microsystems для Unix-подібних ОС ZFS (Zettabyte File System); кластеризована файлова система від VMware для зберігання файлів віртуальних дискових накопичувачів VMFS (Virtual Machine File System); стандарти записи для оптичних дисків, які не завжди відносять до повноцінних файлових систем, ISO 9660, UDF (Universal Disk Format). Цей перелік можна продовжувати.

Розгляньмо кілька прикладів відомих і впливових файлових систем.

FAT — сімейство файлових систем для старіших версій Windows та ОС MS DOS. Основними представниками сімейства є FAT12, FAT16, FAT32, а також є низка пізніших модифікацій.

FAT розшифровується як File Allocation Table (таблиця розміщення файлів), і саме ця таблиця є основою всієї файлової системи.

У FAT окремі файлові блоки, з яких складаються файли, називаються кластери. Розмір кластера коливається від 4 Кб до 256 Кб, залежить від розміру сектора на диску, а також може відрізнятися залежно від характеру використання системи. З кластерами працює логіка, подібна до тієї, яка застосовується до розміру сторінок у сторінковій організації пам'яті. Тобто великий розмір кластера може призвести до внутрішньої фрагментації (подібно до фрагментації сторінок в оперативній пам'яті, але замість сторінок кластери, а замість адресного простору процесів — файли). Малий розмір кластера, знову ж

таки, за аналогією зі сторінковою організацією пам'яті, призведе до зростання кількості кластерів і, як наслідок, до уповільнення операцій з файлами.

Файли у FAT є ланцюжками кластерів. У кожному кластері ланцюжка зберігається фрагмент файлу, щоб прочитати весь файл, потрібно послідовно зчитати всі кластери ланцюжка.

Кожному кластеру відповідає індексний вказівник у таблиці FAT.

Відомості про кластер в індексному вказівнику є наступними:

- кластер вільний чи зайнятий;
- чи є кластер збійним;
- чи є кластер зарезервованим;
- якщо кластер зайнятий не останнім фрагментом файлу — номер кластера з продовженням цього файлу;
- якщо кластер містить останній фрагмент файлу — мітка кінця файлу, EOC (End of Clusterchain - кінець ланцюжка кластерів).

На рис. 6.2 схематично показано приклад розміщення двох файлів у FAT. Для того, щоб, до прикладу, прочитати ***файл A***, потрібно розпочати з кластеру 4, дізнатися, що продовження міститься у кластері 7, перейти до кластеру 7, з'ясувати, що далі треба буде зчитувати з кластеру 1, а зчитавши його, далі вирушити до кластеру 2, тоді до кластеру 10, а вже звідти — до кластеру 12, який виявиться останнім кластером у ланцюжку, бо містить мітку EOC (на схемі позначену як -1).

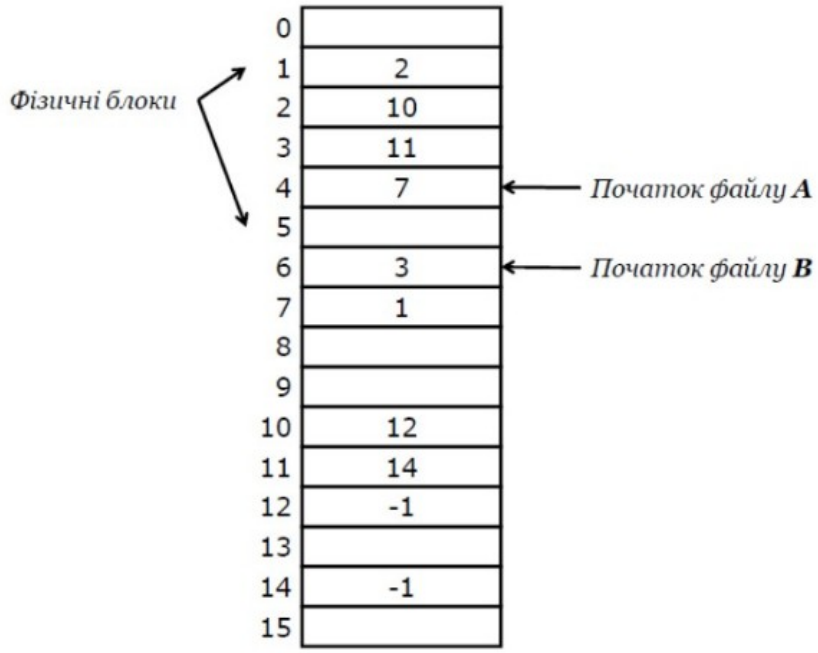

Рис. 6.2. Розміщення файлів у FAT[20]

Запис, який відповідає файлу у каталозі, у файловій системі FAT містить наступні поля:

- ім'я;
- розширення;

[20] Схему адаптовано за книгою: Tanenbaum, Bos. Modern Operating Systems, 2014

- атрибути;
- зарезервоване поле (10 байтів);
- час модифікації (створення);
- дата модифікації (створення);
- номер першого блоку;
- розмір.

Довжина імені файлу у FAT12, FAT16 ще відповідала формату 8.3, а в FAT32 вже становить 255 (256) символів.

Файлова система FAT використовувалася у системі MS DOS та Winodows (до платформи NT). Її також можна, наприклад, зустріти на змінних носіях. Так, FAT32 досі активно використовується на флеш-накопичувачах.

Водночас, FAT має низку принципових обмежень.

Передусім, у FAT не передбачено розмежування доступу до файлів. На практиці це фактично означає, що будь-хто може одержати доступ до будь-чиїх файлів. Така ситуація неприпустима на основних томах під управлінням сучасних ОС як для індивідуального використання, так і для серверів (особливо для серверів!).

Також FAT має обмеження на максимальний розмір файлу — 4 ГіБ, чого замало для багатьох сучасних потреб. Водночас, є модифікації FAT, які підтримують і більші файли.

Журналювання у FAT відсутнє, а надійність забезпечується переважно наявністю резервної копії таблиці розміщення файлів.

FAT має суттєву схильність до фрагментації (про що вже йшлося вище). Також уся таблиця розміщення файлів має постійно перебувати в пам'яті, поки система працює, що почало створювати проблеми зі зростанням обсягів сучасних дискових накопичувачів.

NTFS (New Technology File System) — файлова система для Windows, розроблена для платформи NT й на заміну FAT.

NTFS має розмежування доступу на основі дескрипторів захисту та списків керування доступом (ACL).

Що не менш важливо, NTFS є журнальованою файловою системою, у якій ведеться журнал та зберігаються резервні копії критично важливих системних даних. Також NTFS підтримує більші файли та томи (порівняно з FAT), працює з кластерами.

В NTFS будь-який елемент всередині тому розглядається як **файл**. Кожний файл має атрибути, причому (це відрізняє NTFS від Ext, наприклад) *вміст файлу також є атрибутом*.

Том у NTFS має наступні складники.

Завантажувальний сектор (partition boot sector). Містить відомості про розбиття тому, службові структури файлової системи, дані для завантаження файлової системи.

MFT (master file table). Головна файлова таблиця, про яку ще йтиметься далі.

Системні файли (дзеркальна копія MFT, log-файл з переліком транзакцій, відомості про зайняті кластери, відомості про підтримувані на цьому томі файлові атрибути тощо).

Власне файли, які є наповненням цього тому.

Головна файлова таблиця (MFT) теж є файлом. А з логічної точки зору, головну файлову таблиця MFT можна розглядати саме як таблицю. Рядки MFT містять перелік всіх файлів та їхніх атрибутів у межах тому.

Кожному файлу в NTFS відповідає принаймні один запис у MFT. Якщо файл великий, записів може бути більше.

Розмір одного запису MFT складає 1 Кб.

Запис у MFT містить:

- атрибут вмісту (якщо файл маленький — сам вміст, якщо файл більший - частина вмісту й адреси кластерів, що містять файл);
- решту атрибутів.

Файлові атрибути в MFT можуть відрізнятися залежно від тому, але типові атрибути є наступними:

- стандартна інформація (доступ лише для читання, читання/запис, часові атрибути тощо);
- список атрибутів (якщо всі атрибути не вміщуються в один запис MFT);
- ім'я файлу;
- дескриптор захисту (англ. security descriptor; власник, ACL тощо);
- дані файлу (один атрибут даних без імені за замовчуванням і додаткові пойменовані атрибути даних за потреби);
- спеціальні атрибути для папок та томів;
- бітове поле (показує використовувані записи в MFT чи папці).

Перші шістнадцять записів MFT зарезервовані для службової інформації. Цим записам теж відповідають файли — **метафайли** (їхні імена починаються з символу $). Метафайли зберігають інформацію, яка дає змогу відновлювати коректний стан файлової системи після збою.

Деякі метафайли наведено у таблиці 6.1.

Таблиця 6.1. Приклади деяких метафайлів у NTFS

Номер	Ім’я відповідного метафайлу	Призначення
0	`$MFT`	Головна файлова таблиця
1	`$MFTmirr`	Дзеркальна копія MFT (її перших 16 записів)
2	`$LogFile`	Журнал
4	`$AttrDef`	Стандартні атрибути файлів

Ext (від extended — розширена) — сімейство файлових систем для Linux. Ці файлові системи походять від файлової системи UFS (Unix Filesystem).

Представниками сімейства Ext є перша файлова система Ext, яка не мала номера (1992, Remy Card), її наступниця, файлова система Ext2 (1993, Remy Card), файлова система, в якій було впроваджено журналювання, Ext3 (2001, Stephen Tweedie) й сучасна версія Ext4 (2008, Theodore Tso), яка, окрім журналювання, має механізм екстентів, про який ітиметься трохи нижче.

Базові файлові структури в Ext запозичені з UFS. Зокрема, саме з UFS походить файловий об’єкт, що асоціюється з окремим файлом — **айнод** (inode, від index node — індексний вузол).

Айнод містить наступну інформацію.

• Атрибути файлу:

- тип файлу (звичайний файл, каталог, файл пристрою тощо);
- розмір файлу;
- дата та час останньої модифікації файлу;
- ідентифікатор користувача-власника, групи-власниці;
- атрибути доступу (читання, запис, виконання для базових ролей).

• Вказівники на ті блоки диску, в яких зберігаються фрагменти цього файлу.

В UFS та в файлових системах Ext до Ext3 включно використовувалася базова структура айнода, схематично зображена на рис. 6.3. Для невеликих файлів за такого підходу вистачить і основних адрес (від 0 до 7), але більший файл може складатися з більшої кількості блоків, тому останній вказівник зберігає адресу ще одного блоку зі вказівниками, додаткового. З розвитком технологій і зростанням обсягів файлів це стало проблемою, оскільки великі файли означали більше адрес вказівників, й доступ до таких файлів виявився повільним.

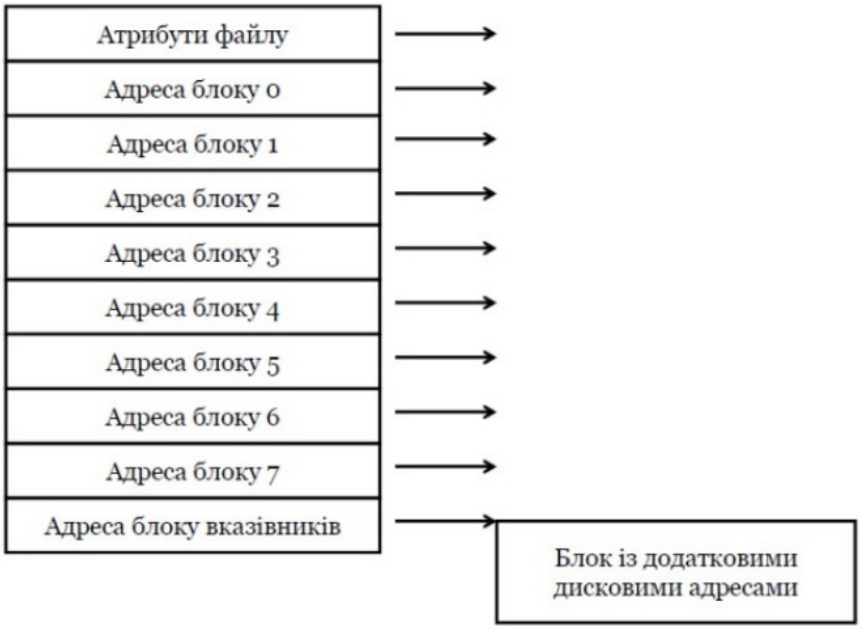

Рис. 6.3. Загальна структура айнода в UFS та ранніх Ext[21]

В Ext4 для адресації використовується інший підхід — екстенти.

Екстент (extent) — діапазон суміжних фізичних блоків, відведений під зберігання файлу.

Екстент визначає:

- адресу початкового блоку;
- кількість блоків в екстенті.

Врахуйте, що сказане вище не означає, що весь файл мусить зберігатися в неперервному блоці на диску. Але метою підходу екстентів є збільшення розміру таких блоків і, як наслідок, уникнення надмірної фрагментації файлу.

[21] Схему адаптовано за книгою: Tanenbaum, Bos. Modern Operating Systems, 2014

Також екстенти забезпечують економію пам'яті, оскільки для одного екстента потрібно зберігати лише два значення.

Водночас, якщо файл все ж виявиться фрагментованим, то йому відповідатиме багато екстентів. Для боротьби з фрагментацією використовується *відстрочене виділення пам'яті* (allocating-on-flush), основна ідея якого полягає у певному зволіканні перед записом чергового блоку, оскільки так зростає ймовірність, що з'явиться вільне місце під більший неперервний блок.

Окремим важливим питанням є потреба працювати з декількома файловими системами одночасно, яка часто виникає на практиці. Тобто, певна річ, є файлова система того розділу, куди встановлено ОС, але що, як інші дискові томи мають інші файлові системи, або буде підключено флеш-носій з іншою файловою системою? або до комп'ютера буде під'єднано додатковий дисковий накопичувач? Перелік подібних ситуацій можна продовжувати.

Механізми для забезпечення роботи з нерідними файловими системами наявні у різних ОС. У даному посібнику буде коротко розглянуто VFS у Linux як приклад фундаментального підходу до роботи з файловими системами.

VFS (Virtual File System) використовується у Linux-системах. З концептуальної точки зору, VFS — це загальна модель властивостей і поведінки довільної допустимої файлової системи. Під допустимою файловою системою мається на увазі файлова система, роботу з якою дана Linux-система підтримує і забезпечує.

Файли у VFS розглядаються як об'єкти у пам'яті накопичувача. Спеціальний модуль — **модуль відображення** (mapping module) — перетворює характеристики деякої реальної файлової системи у характеристики, очікувані файловою системою VFS.

Загальна послідовність роботи VFS є наступною.

1. Процес ініціює системний виклик через інтерфейс VFS.
2. Ядро викликає відповідну функцію VFS.
3. Функція VFS ініціює виклик функції цільової файлової системи (File System X).
4. Код виклику функції цільової файлової системи передається через функцію модуля відображення. Функція модуля відображення конвертує виклик у виклик цільової файлової системи.

Скажімо, щоб ви могли працювати з розділами на NTFS з-під Linux, у Linux має бути встановлено модуль відображення для NTFS. Часто такі модулі вже встановлено для більшості популярних файлових систем, іноді виникає потреба їх довстановити.

Розділи з іншими файловими системами у Linux монтуються. Каталог, з якого буде забезпечуватися доступ до змонтованої файлової системи, називається точкою монтування. У настільних версіях Linux монтування часто відбувається автоматично, й автоматично створюється спеціальний каталог, який стає точкою монтування. Але в деяких випадках може знадобитися виконати монтування вручну — для цього використовується команда `mount`.

Контрольні запитання

1) Які дві базові функції виконує файлова система?
2) Які два основні рівні можна виокремити у межах файлової системи?

3) Які складові частини, окрім власне файлів, разом утворюють файлову систему?
4) Що називається файлом?
5) Перелічіть три очікувані характеристики файлів за В. Столлінґсом.
6) Перерахуйте типові атрибути файлів.
7) Як можуть різнитися вимоги до іменування файлів з точки зору чутливості до регістру символів, ролі розширення, максимально допустимої довжини імені?
8) Дайте два означення каталогу — з точки зору внутрішньої будови файлової системи і з точки зору щоденного користувачання.
9) Чим абсолютне ім'я файлу відрізняється від відносного? Наведіть приклади.
10) Яка файлова система називається журнальованою? Опишіть суть механізму журналювання у файлових системах.
11) Як пов'язана кількість дискових томів та фізичних накопичувачів, які їм відповідають? Чи завжди один том займає увесь накопичувач? Чи може том виходити за межі накопичувача?
12) Наведіть приклади файлових систем для різних ОС.
13) Охарактеризуйте файлову систему FAT. Які недоліки притаманні цій файловій системі? Які способи мінімізації дії цих недоліків застосовуються?
14) Охарактеризуйте файлову систему NTFS. Які недоліки притаманні цій файловій системі? Які способи мінімізації дії цих недоліків застосовуються?
15) Охарактеризуйте файлові системи Ext. Що вирізняє файлову систему Ext4 з-поміж решта сімейства Ext? Які недоліки притаманні цій файловій системі? Які способи мінімізації дії цих недоліків застосовуються?
16) На прикладі VFS поясніть, як працюють механізми взаємодії з нерідними файловими системами в ОС.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 10-11.

2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 12 (paragr. 12.7-12.10).

3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 4.

4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділ 11, 12 (парагр. 12.4), 13.

Розділ 7

Безпека і захист

1. Базові поняття і принципи безпеки ОС

У назві цього розділу терміни “безпека” і “захист” фігурують разом, втім, їх не завжди розділяють. У цьому посібнику ми переважно дотримуватимемося підходу, запропонованому у підручнику [1], автори якого означають безпеку і захист ОС наступним чином.

Безпека ОС (OS security) — міра впевненості, що цілісність системи і даних всередині неї буде збережено.

Захист ОС (OS protection) — механізм контролю доступу програм, процесів або користувачів до ресурсів комп'ютерної системи.

Тобто за такого підходу безпека ОС розглядається як результат, а захист ОС — як комплекс засобів для досягнення цього результату.

Механізм контролю, що надається у межах захисту, має надавати:

- засоби для визначення заходів контролю;
- засоби для примусового застосування цих заходів.

Іншими словами, потрібні гарні безпекові правила, і ці правила повинні працювати.

Порушення безпеки може мати як умисний, так і неумисний характер, зокрема бути спричиненим людською чи технічною помилкою, форс-мажорними обставинами).

На рис. 7.1 ресурси ОС категоризовано залежно від притаманних їм вразливостей.

Рис. 7.1. Ресурси ОС та їхні основні вразливості

Апаратне забезпечення є сукупністю фізичних пристроїв та комплектуючих до них, тому воно може бути втрачене фізично (загублене, крадене, знищене) або зазнати пошкодження (внаслідок зношування, виробничого браку, неправильного використання, умисного псування, форс-мажорних обставин).

Програмне забезпечення залежить від апаратного у тому сенсі, що потребує апаратної платформи, щоб працювати на ній (навіть якщо опосередковано, через хмарні технології), а програмний код та інсталяційні файли технічно є даними, котрі потрібно десь зберігати. Але саме по собі програмне забезпечення може мати такі вразливості, як зміна поведінки нешкідливого ПЗ (через помилку у програмі, через її неправильне використання, через умисне використання вразливостей або заздалегідь створених лазівок) та ПЗ, яке є власне шкідливим — це звичайне ПЗ, код якого змінено (як у випадку троянів), або це ПЗ початково розроблялося як шкідливе (як віруси, черви, руткіти).

Дані у комп'ютерній системі сьогодні є надважливим ресурсом. Дані може бути втрачено, пошкоджено, підмінено. Також може статися витік даних, і конфіденційні або чутливі дані потраплять до рук зловмисників. Усе це може трапитися внаслідок програмного збою, пошкодження чи втрати фізичного носія з даними, людської помилки, дій зловмисників тощо.

Комунікації — це передусім електричні та комп'ютерні мережі, які є джерелом енергії та зв'язку відповідно. Живлення необхідне для більшості комп'ютерних пристроїв, і за відсутності резервних джерел енергії відсутність доступу до електромережі унеможливлює роботу системи. За умов нестабільної роботи електромережі та інші джерела живлення також бувають джерелом проблем, оскільки комп'ютерні пристрої можуть пошкодитися. Комп'ютерні мережі є невід'ємною частиною сучасної IT-інфраструктури, без них не

працюватиме значна частина робочих процесів, не функціонуватиме звичним чином дозвілля та критично важливі системи, тож однією з вразливостей комп'ютерних мереж є втрата ними своєї працездатності. Водночас, комп'ютерні мережі часто виступають каналом, через який на комп'ютер потрапляє шкідливе ПЗ, про яке йшлося вище, а зловмисники здійснюють різноманітні атаки.

Як бачимо, всі перелічені ресурси мають вразливості, відповідні загрози різноманітні, і захист від них має бути комплексним. То яку ж роль відіграє у цьому ОС? Чи може ОС містити у собі засоби, які будуть ефективні для захисту всіх перелічених вразливостей? Коротка відповідь на друге питання може бути сформульована наступним чином: всі — не може, але може достатньо багато.

Безумовно, безпека — це комплексне завдання, і її забезпечення передбачає заходи на різних рівнях. Умовно виділимо *чотири рівні впровадження заходів з безпеки*.

1. Фізичний рівень. Місце фізичного розміщення комп'ютерних систем має бути фізично захищене від збройного й таємного проникнення зловмисників.

2. Людський рівень. Людей, які використовують комп'ютерні системи, часто називають слабкою ланкою в системі безпеки. Варто спрямовувати зусилля на те, аби так не було. Людський рівень заходів з безпеки передусім передбачає протидію передачі авторизованими користувачами своїх реквізитів для доступу до системи стороннім особам, свідомого чи несвідомого.

3. Рівень операційної системи. ОС має захищати себе від порушень системи безпеки (неумисних та умисних). Приклади розглянемо нижче.

4. Мережний рівень. Забезпечення працездатності мережних з'єднань, захист даних, які передаються по мережі, та систем, під'єднаних до мережі.

Розробники, адміністратори та користувачі ОС мають дуже обмежену можливість впливати на безпеку на фізичному, людському та мережному рівні. Але їхні дії (і бездіяльність) безпосередньо впливають на те, що стосується самої ОС. *Перелік заходів з безпеки, доступних на рівні ОС*, містить, зокрема, наступні пункти:

- зупинити несанкціонований доступ (автентифікація, розмежування доступу, шифрування);
- вести аудит подій, які відбуваються у системі (для виявлення порушень безпеки, для вчасного реагування, для проведення розслідування);
- запобігти втраті даних (резервне копіювання);
- попереджати про потенційно небезпечне ПЗ (накладання обмежень на встановлення ПЗ з ненадійних джерел чи встановлення ПЗ користувачами без відповідних повноважень, пропонування надійних і зручних джерел програмного забезпечення);
- поважати приватність користувача (дотримання законодавства щодо збирання та використання особистих даних);
- протидіяти шкідливому ПЗ (антивірусне ПЗ, дієві безпекові механізми та коректні безпекові налаштування).

У теорії ОС часто використовуються терміни "механізми безпеки" та "політики безпеки". Важливо з'ясувати, чим вони відрізняються.

Політики безпеки задають, *що* саме буде зроблено. Політики можуть відрізнятися залежно від обставин і змінюватися з часом.

Механізми безпеки визначають, *за допомогою чого* буде зроблено те, що задане політиками. Механізми в основі політики мають лишатися незмінними.

Політики безпеки можуть бути закладені на етапі проєктування ОС, встановлені у процесі адміністрування ОС, а також визначені окремими користувачами для захисту їхніх власних файлів та програм. Політики, закладені у ході проєктування ОС, відображаються у вигляді налаштувань за замовчуванням, які зазвичай можна змінити. В ідеалі налаштування за замовчуванням мають забезпечувати принаймні базовий рівень безпеки. Так, у під час встановлення сучасних настільних ОС стандартним чином зазвичай не вдається створити нового користувача зі слабким паролем, що сприяє вибору користувачем надійніших паролів. Адміністратори змінюють стандартні налаштування відповідно до наявних потреб, і ці політики також мають бути виваженими. Звичайному непривілейованому користувачу зазвичай доступно небагато безпекових налаштувань, але і такий користувач може встановлювати власні політики.

Безпечна ОС і надійна ОС. Поняття безпеки і надійності, безумовно, мають спільні аспекти, але загалом описують різні характеристики системи. *Безпечною* називають ОС, захищену від загроз, тимчасом як *надійною* називають ОС, яка коректно працює протягом передбачуваного періоду часу.

Варто розуміти, що і поняття безпеки, і поняття надійності умовне, і, як і оптимальний алгоритм заміщення сторінок, є ідеальним станом, на який мають бути спрямовані зусилля. Високий рівень безпеки чи надійності може бути підтверджений великою кількістю неупереджених тестів та даними, зібраними у ході тривалого використання ОС. Та розвиток IT загалом й ОС зокрема передбачає постійні зміни, тож і нові виклики з’являються постійно, й цілком обґрунтовані твердження про високу безпеку чи надійність за деякий час втрачають актуальність.

Базових механізмів безпеки ОС є три, і вони є наступними:

- **автентифікація** (authentication) — кожна дія виконується певним суб'єктом, ідентичність якого встановлено системою;
- **авторизація** (authorization) — система регулює, які дії дозволені тим чи іншим суб'єктам;
- **аудит** (accounting): система документує події, пов'язані з безпекою.

Перелічені механізми застосовуються одночасно і разом утворюють абревіатуру **AAA** (англ. **triple A**). Подібні механізми також використовуються у комп’ютерних мережах, звідки й походить абревіатура.

Важливо також з’ясувати **загальні вимоги до систем безпеки та захисту**, яких важливо дотримуватися і під час розробки ОС, і під час їх адміністрування. Наведемо у дещо адаптованому вигляді перелік таких вимог, сформульований у роботі авторів Saltzer, Schroeder ще у 1975 році, й пізніше переосмислений і доповнений [5]. Ці вимоги або втілені у сучасних системах, або лишаються предметом дискусій, все ж маючи часткове втілення.

1. Механізми безпеки мають бути настільки простими і невеликими, наскільки це можливо.

2. Орієнтація передусім на дозволи, а не на заборони.

3. Кожний доступ до кожного об'єкта має перевірятися на предмет наявності повноважень.

4. Будова механізмів безпеки має бути відкритою, а не засекреченою.

5. Там, де це доцільно, для посилення механізми захисту варто використовувати ті, де передбачено доступ за двома ключами.

6. *Принцип мінімальних повноважень* (principle of least privilege): кожній програмі чи користувачу у системі має бути надано мінімальний набір повноважень, необхідний для виконання поставлених перед ними завдань.

7. Психологічна доступність: механізми захисту повинні мати зручний для використання людиною інтерфейс, щоб користувачі могли регулярно й коректно використовувати ці механізми.

8. Порівнювати вартість обходу захисних механізмів з наявними в потенційного зловмисника ресурсами.

9. Надійні записи інформації про випадки, коли систему безпеки було скомпрометовано, можуть допомогти у виробленні надійніших механізмів.

2. Автентифікація та авторизація

Передусім, уточнимо ролі об’єкта і суб’єкта у системі безпеки: суб’єкти виконують дії над об’єктами. **Суб’єктом** найчастіше є користувач, група, процес, потік. **Об’єктами** є різноманітні ресурси, наявні у системі: файли, папки, принтери, вебкамери тощо. На даному етапі буде зручно уявляти суб’єкт як певного користувача, а об’єкт — як файл.

Суб’єкт проходить автентифікацію, щоб підтвердити своє право увійти у систему. Вже всередині системи суб’єкт уже може запитувати доступ до конкретних об’єктів. За допомогою авторизації перевіряється, чи має цей суб’єкт право на доступ до даного об’єкта. Якщо так, доступ буде надано.

Розрізняють два основні *типи автентифікації*:

- **локальна** (перевірка легітимності входу здійснюється на комп'ютері, за яким працює користувач);
- **мережна** (перевірка легітимності входу виконується на віддаленому комп'ютері, дані передаються по мережі).

Локальна автентифікація вважається менш безпечною (дані зберігаються локально) та складнішою для централізованого керування. Водночас, у разі ускладненого доступу до мережі може знадобитися саме цей тип автентифікації.

Мережна автентифікація потребує наявності мережного з’єднання і певних організаційних зусиль на етапі налаштування та пізніше, у ході обслуговування. Водночас, мережна автентифікація виправдовує себе у більших мережах — нею зручніше керувати централізовано, а організувати безпечне передавання реквізитів входу по мережі та зберігання даних для перевірки легітимності входу на сервері зазвичай усе ж простіше, ніж гарантувати безпеку локальної автентифікації.

Також існують різні форми автентифікації, найпоширенішими з яких є наступні:

- автентифікація на основі паролю;
- автентифікація на основі фізичного об'єкту (наприклад, апаратного ключа, банківської картки, персонального мобільного пристрою на зразок смартфона);
- автентифікація на основі біометричних параметрів (розпізнавання відбитків пальців, обличчя, голосу, сітківки ока тощо).

Часто застосовується двофакторна автентифікація або мультифакторна автентифікація, яка поєднує два чи більше способів автентифікації відповідно. Нерідко тут ідеться про поєднання різних форм. Наприклад, автентифікація за допомогою банківської картки (фізичного об’єкта) і PIN-коду (тобто паролю).

Інший приклад — використання смартфона для підтвердження автентифікації в користувацькому профілі на настільному ПК чи ноутбуці (також пароль плюс фізичний об'єкт).

Користувачі та групи. Основними суб'єктами розмежування доступу в ОС є користувачі та групи. Процеси та потоки діють від імені тих чи інших облікових записів користувачів та груп, можливо, системних. Наприклад, якщо ви запустили програму з-під облікового запису користувача, то новостворений процес також діятиме від імені цього користувача й матиме відповідні повноваження.

Серед типових відомостей облікового запису користувача варто згадати зокрема наступні:

- логін користувача;
- ідентифікатор користувача (у Linux — UID / User Identifier, у Windows — SID / Security Identifier);
- відомості про пароль (сам пароль у зашифрованому вигляді або спеціальні відомості, які дозволяють перевірити правильність введеного паролю, наприклад, хеш);
- відомості про належність користувача до груп;
- обмеження на вхід для даного користувача (термін дії облікового запису, паролю, дні чи години, в які користувачу дозволено входити до системи);
- шлях до домашнього каталогу користувача;
- шлях до типового для користувача командного інтерпретатора та ін.

Під *користувачем* може матися на увазі як людина-користувач, так і обліковий запис, створений для цієї людини (одна людина може мати багато облікових записів). Крім того, облікові записи можуть і не бути асоційовані з реальною людиною, як-то системні облікові записи — наприклад, *root, daemon* у Linux чи *System* у Windows.

Групи застосовуються, щоб керувати доступом до ресурсів за певним зразком. Існують стандартні групи (наприклад, *Administrators*, *Users* у Windows, *sudo* у Linux), але адміністратор може створювати власні групі відповідно до наявних потреб.

Групи, які і облікові записи користувачів, на рівні ОС розрізняються за ідентифікаторами. У Linux це GID (Group Identifier), у Windows — SID (Security Identifier), як і у користувачів.

До паролів висувається низка вимог, які можуть частково залежати від безпекових політик всередині мережі чи організації, але загалом відповідають наступним *правилам*.

1. Пароль має бути довгим (зазвичай щонайменше вісім символів), але не занадто довгим (користувачі схильні записувати довгі паролі, і це складно контролювати).
2. Пароль повинен містити літери різного регістру, цифри та інші символи.
3. Пароль не має бути словниковим словом, повторювати логін або містити інформацію про користувача, наявну у відкритому доступі (дата народження, улюблений музичний гурт тощо).
4. За можливості рекомендовано використовувати двофакторну автентифікацію.
5. Пароль мусить мати обмежений термін існування.

Наразі останній пункт наведеного вище переліку стосовно обмеженого терміну існування часто піддається сумнівам. Безумовно, важливо контролювати облікові записи, які використовуються рідко, або облікові записи, які вже не повинні використовуватися (наприклад, відключати та видаляти облікові записи працівників, які більше не працюють в організації). Водночас, якщо користувач має пароль, що відповідає вимогам складності, і застосовує двофакторну автентифікацію, то у частій регулярній зміні паролю може не бути потреби.

Основні підходи до розмежування доступу. Розмежування доступу передбачає існування певних правил, які пов'язували б суб'єкта, що пройшов автентифікацію, об'єкт (тобто ресурс), дії, виконувані з об'єктом (ресурсом), й однозначно давали б відповідь на питання: чи дозволено виконувати деяку дію деякому суб'єкту щодо деякого ресурсу.

Основні *підходи до розмежування доступу* включають наступні:

- матриця доступу;
- списки контролю доступу;
- списки можливостей.

Матриця доступу (access matrix) описує загальний випадок розмежування доступу. Логічну структуру матриці доступу можна уявити як таблицю, яка задає однозначну відповідність між усіма наявними у системі суб’єктами та всіма доступними ресурсами.

Приклад матриці доступу наведено у таблиці 7.1.

Таблиця 7.1. Приклад матриці доступу для малої кількості суб’єктів та ресурсів

	Файл1	Файл2	Файл3	Каталог1	Каталог2	Принтер1
Користувач1	Читання	Читання, виконання			Читання	
Користувач2			Читання, запис	Запис		Друк
Користувач3						Друк, зміна налаштувань
Група1	Читання					
Група2		Читання, запис, виконання			Читання	

Дуже важливо враховувати, що в реальності зберігати та використовувати подібні дані як матрицю не надто зручно. У справжній системі перелік суб’єктів буде значно більшим, а перелік об’єктів (ресурсів) — більший у рази, і матриця матиме величезні розміри. Тому на практиці відомості про розмежування доступу прив’язують або до суб'єктів, або до об'єктів.

Списки керування доступом (ACL — Access Control List) передбачають, що для кожного ресурсу задано список суб'єктів, яким дозволено використовувати цей ресурс, та зазначено, як саме їм дозволено це робити.

Такий підхід використовується у більшості ОС. Наприклад, у Windows *списки керування доступом* називаються так само, і також можуть містити заборони (якими, втім, рекомендовано там не зловживати). У Linux використовується скорочений варіант ACL — *рядки повноважень*, або *повноваження*.

Переліки можливостей (Capabilities List, C-list) задають для кожного суб'єкта перелік ресурсів, котрі йому дозволено використовувати.

У чистому вигляді переліки можливостей використовуються переважно у проєктах дослідницького спрямування на зразок Mach, Hydra, Cambridge CAP System, однак елементи переліків можливостей є, приміром, у Windows (привілеї та права доступу облікових записів).

Детальніше розгляньмо, як працює розмежування доступу у Linux та у Windows.

Розмежування доступу у Linux базується на трьох основних ролях та одній додатковій: `u` — користувач-власник (user owner), `g` — група-власниця (group owner), `o` — решта користувачів (others), `a` — всі користувачі (all, додаткова роль, застосовується для зміни налаштувань).

Для кожної ролі задається комбінація дозволів: `r` — читання (**r**ead), `w` — запис (**w**rite), `x` — виконання (e**x**ecute). Коли котрийсь із цих дозволів відсутній, на його місці ставиться дефіс (-). Інтерпретація цих дозволів для каталогів буде дещо іншою. Наприклад, `r` означатиме можливість переглядати вміст каталогу, `w` — зміну вмісту каталогу (включно зі створенням та видаленням елементів у цьому каталозі), а `x` — надаватиме змогу проходити крізь цей каталог для доступу до вкладених елементів, роблячи його поточним або вказуючи як частину шляху (наприклад, можна не мати можливості перегляду самого каталогу, але мати право переглядати вміст підкаталогу).

На рис. 7.2 показано рядок повноважень для файлу `file1`, власниками якого є користувач `olena` та група `olena`.

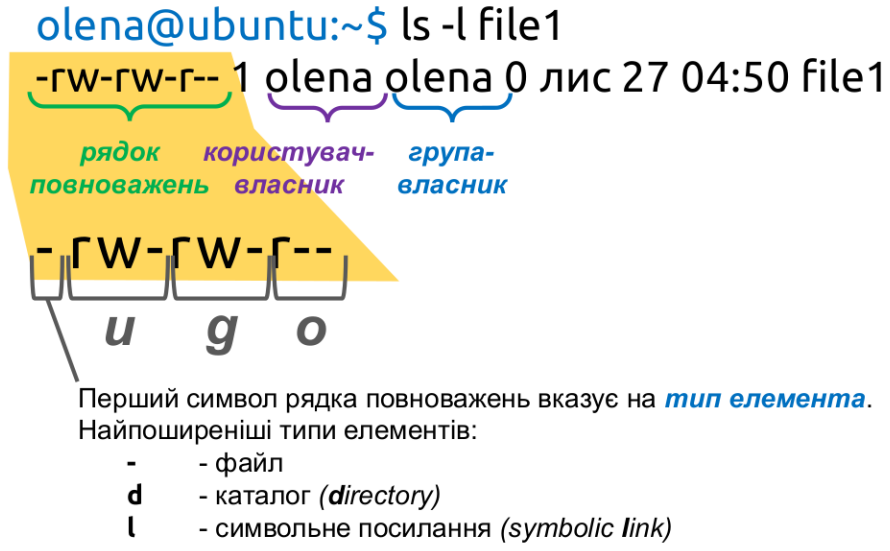

Рис. 7.2. Приклад інтерпретації рядка повноважень у Linux

Користувачу-власнику (**u**) дозволено читати та записувати цей файл, але не дозволене виконання. Втім, судячи з усього, це звичайний текстовий файл, до

того ж, порожній (розмір дорівнює 0), тому він і не має виконуватися. Члени групи-власниці (**g**) мають такі самі повноваження. Решті користувачів (**o**) дозволено лише читати файл.

Перший символ рядка повноважень відповідає за тип елемента. У випадку, зображеному на рис. 7.2 елемент є файлом (-).

Механізм *sudo* у Linux. Традиційний обліковий запис адміністратора у Unix/Linux має ім'я *root*. Обліковка root має необмежений доступ, а решта звичайних користувацьких облікових записів традиційно має суттєво вужчі повноваження.

Водночас, у сучасних Linux часто використовується механізм *sudo*: усі користувачі входять під обмеженими обліковими записами, але частина цих користувачів має право використовувати команду `sudo` чи її аналоги. Обліковий запис *root* при цьому може бути відключений, але наявний у системі і продовжує володіти багатьма файлами та каталогами, зокрема й системними. Це потрібно для того, щоб змінювати їх могли лише користувачі з правом *sudo*.

Користувачі, яким дозволяється використовувати *sudo*, зазвичай є членами однойменної групи *sudo*. Враховуючи сказане вище, надзвичайно важливою практикою є вкрай виважене додавання користувачів до цієї групи. Інші суб'єкти, яким також можна застосовувати `sudo`, вказуються у файлі `/etc/sudoers`.

Розмежування доступу у Windows. .В ОС Windows кожному об'єкту відповідає *дескриптор захисту* (security descriptor).

Ось деякі елементи дескриптора захисту:

- SID власника об'єкта;
- список керування доступом (ACL);
- ACE (Access Control Entry):
  - тип ACE (дозвільний чи забороннний);
  - SID користувача або групи;
  - набір прав доступу, які надаються чи забираються

Набір прав доступу відповідає повноваженням в NTFS, або дозволам в NTFS. Другий термін у випадку Windows може часом виглядати контроверсійно, оскільки такі *дозволи* можуть бути *забороннними* за типом ACE (тобто фактично дозвіл не надається, а прибирається). Заборона передбачає, наприклад, скасування дозволу для користувача на доступ до певного ресурсу, коли за інших обставин користувач мав би цей доступ як член групи.

Розрізняють *основні* та *розширені* повноваження (дозволи) в NTFS. Перелік дозволів NTFS та їхня інтерпретація може відрізнятися залежно від того, встановлено їх для папки чи для файлу.

Основні повноваження (дозволи) в NTFS є наступними.

Читання (Read). Користувачам дозволяється читати вміст файлу та його атрибутів. Користувачам дозволяється переглядати вміст папки та підпапок.

Читання та виконання (Read and Execute). Користувачам дозволяється переглядати та запускати виконувані файли, у тому числі скрипти.

Запис (Write). Дозволяє користувачам додавати файли та підпапки всередині папки; записувати у файл.

Зміна (Modify). Користувачам дозволяється читати, записувати файли і підпапки. Дозволяється також видаляти папку.

Виводити вміст папки (List Folder Contents). Діє лише для папок. Усередині папки користувачам дозволяється переглядати перелік файлів та підпапок.

Повний доступ (Full Control). Користувачам дозволяється читати, перезаписувати, змінювати вміст та видаляти файли та підпапок. Також користувачам дозволяється змінювати налаштування доступу для всіх файлів та підпапок.

Розширені повноваження в NTFS надають можливість точнішого налаштування доступу. Фактично, кожний із основних дозволів NTFS складається з певної комбінації розширених дозволів.

Огляд папки / Виконання файлу (Traverse Folder / Execute File). *Огляд папки:* дозволяє рух углиб папки, доступ до якої обмежений, аби дістатися до файлів та папок на нижчих рівнях ієрархії папок. Ці повноваження не обов'язково означають, що користувачу так само буде дозволено й запуск виконуваних файлів. *Виконання файлу:* дозволяє запуск виконуваного файлу.

Вміст папки / Читання даних (List Folder / Read Data). *Вміст папки:* дозволяє переглядати імена файлів та підпапок всередині даної папки. Налаштування не впливає на можливість перегляду відомостей про саму папку. *Читання даних:* дозволяє перегляд даних у файлах.

Читання атрибутів (Read Attributes). Дозволяє перегляд основних атрибутів файлу чи папки (лише для читання, прихований тощо).

Читання розширених атрибутів (Read Extended Attributes). Дозволяє перегляд розширених атрибутів файлу чи папки. Це атрибути, які визначаються програмами і залежать від конкретної програми.

Створення файлів / Запис даних (Create Files / Write Data). *Створення файлів:* дозволяє створення файлів всередині папки. *Запис даних:* дозволяє внесення змін у файл, включаючи перезапис наявного вмісту.

Створення папок / Дозапис даних (Create Folders / Append Data). *Створення папок:* дозволяє створювати підпапки всередині даної папки. *Дозапис даних:* дозволяє дописувати дані після вже наявного вмісту файлу. Змінювати, видаляти, перезаписувати наявні у файлі дані не можна.

Запис атрибутів (Write Attributes). Дозволяє змінювати атрибути файлу чи папки. Не стосується атрибутів новостворених файлів та підпапок.

Запис розширених атрибутів (Write Extended Attributes). Дозволяє змінювати розширені атрибути. Не стосується розширених атрибутів новостворених файлів та підпапок.

Видалення підпапок та файлів (Delete Subfolders and Files). Дозволяється видалення підпапок та файлів всередині даної папки, навіть якщо для цього файлу чи підпапки це не дозволено.

Видалення (Delete). Дозволяється видалення файлів та папок.

Читання дозволів (Read Permissions). Дозволяється читати налаштування повноважень щодо файлу чи папки.

Зміна дозволів (Change Permissions). Дозволяється змінювати налаштування повноважень щодо файлу чи папки.

Зміна власника (Take Ownership). Дозволяється зміна власника папки чи файлу. Власник файлу чи папки завжди може змінювати повноваження цих файлу чи папки, незалежно від інших налаштувань.

Синхронізація (Synchronize). Різним потокам під час роботи з цією папкою (цим файлом) дозволяється очікувати на інші потоки. Стосується програм з багатьма процесами та багатьома потоками.

3. Аудит

Аудит передбачає фіксування важливих подій, які відбуваються у системі. У цьому розділів нас передусім цікавить аудит безпекових подій, але загалом стосуватися безпеки можуть дуже різноманітні події. Наприклад, нестандартна поведінка давно встановленої програми також може становити інтерес для фахівців з кібербезпеки.

Системним аудитом називається документована перевірка роботи системи.

Політика аудиту — це перелік подій, повідомлення про які підлягають документуванню. Такі повідомлення називаються ***повідомленнями аудиту***.

Повідомлення найчастіше фіксуються у **журналах аудиту** (logs). За це відповідає спеціальний фоновий процес. Журнали аудиту зберігають тривалий час та аналізують як на регулярній основі (з метою профілактики), так і задля пошуку причин проблем чи розслідування порушення безпеки.

Процеси (особливо системні) можуть використовувати для аудиту стандартні системні процеси, але також можуть здійснювати власний аудит. Відповідно, log-файли також можуть бути стандартні або індивідуальні, створені окремими програмами і, відповідно, з власним форматом.

У Linux журнал аудиту має назву **системний журнал** (system log). Стандартні процеси аудиту у Linux можуть відрізнятися і бути наступними.

Процеси **syslogd** та **klogd**. Використовуються у старіших системах, *syslogd* відповідає за аудит процесів, які працюють у режимі користувача, а *klogd* — за аудит процесів, які працюють у режимі ядра)

Процес **rsyslog.** Використовується у новіших системах.

Процес **journald**. Працює у найновіших системах, працює не лише з текстовими, й з з двійковими файлами журналів.

Стандартний каталог аудиту у Linux має назву `/var/log`, однак можуть бути й інші.

У Windows журнал аудиту називається **журналом подій** (event log). Те, які події фіксуватимуться у журналі, визначається **локальною політикою безпеки**. Журнали у Windows можна переглядати через спеціальне оснащення з графічним інтерфейсом, а також за допомогою командного рядка.

Служба, відповідальна за ведення журналу, у Windows називається **eventlog**.

Журнали подій у Windows зазвичай зберігаються у папці `C:\WINDOWS\system32\config\.`

Контрольні запитання

1) Поясніть відмінність між термінами “безпека” та “захист”.
2) Опишіть основні вразливості таких ресурсів ОС, як апаратне забезпечення, програмне забезпечення, дані та комунікації.

3) Коротко опишіть заходи безпеки, які можуть впроваджуватися на фізичному рівні, людському рівні та мережному рівні.
4) Опишіть основні заходи безпеки, які може бути застосовано на рівні операційної системи.
5) У чому полягає відмінність між механізмами безпеки та політиками безпеки в ОС?
6) Чим безпечна ОС відрізняється від надійної ОС? Наскільки строгим можна вважати твердження, що деяка ОС є безпечною чи надійною?
7) Як розшифровується абревіатура AAA у безпеці ОС та комп'ютерних мереж?
8) Поясніть суть базових механізмів безпеки ОС (автентифікація, авторизація, аудит).
9) * Проаналізуйте наведені у п. 1 загальні вимоги до систем безпеки та захисту з праці Saltzer, Schroeder (1975) з пізнішими доповненнями. Які з вимог зараз є стандартами й гарними практиками, а які мають статус дискусійних?
10) Наведіть приклади суб'єктів та об'єктів системи безпеки ОС.
11) У чому полягають відмінності між локальною та мережною автентифікацією? Які переваги та недоліки мають ці типи автентифікації?
12) Назвіть відомі вам форми автентифікації. Назвіть приклади використання цих форм на практиці.
13) Які типові відомості про обліковий запис користувача зазвичай наявні в ОС?
14) Для чого використовуються групи в ОС?
15) Перелічіть та обґрунтуйте основні вимоги до паролів.
16) Назвіть три основні підходи до розмежування доступу в ОС. Якщо підхід використовується у реальних ОС, наведіть приклади.
17) Опишіть основні особливості розмежування доступу у Linux.
18) Опишіть основні особливості розмежування доступу у Windows.
19) Що таке системний аудит? Що називається політикою аудиту? Порівняйте основні особливості системного аудиту у Linux та Windows.

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 14-15.

2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 15.

3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 9.

4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділ 18.

5. R. E. Smith. Smith, A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles. IEEE Security Privacy. 2012. 10(6). P. 20–25. URL: http://web.mit.edu/Saltzer/www/publications/protection/Basic.html

Розділ 8

Міжпроцесова та міжпотокова взаємодія

1. Проблеми міжпроцесової та міжпотокової взаємодії

Процесам і потокам потрібно взаємодіяти — для виконання спільних завдань та доступу до спільних ресурсів. У розділі 3 було розглянуто основні положення багатопотоковость, але питання узгодження взаємодії потоків у даному посібнику поки що детально не висвітлювалося. Цей розділ присвячено саме таким питанням.

Водночас, варто зазначити, що у даному розділі здійснено досить загальний огляд питань міжпроцесової та міжпотокової взаємодії з метою передусім забезпечити базове розуміння основних понять, проблематики та застосовуваних технологій. Якщо ви цікавитеся системним програмуванням та розробкою багатопотокових застосунків, рекомендуємо звернутися до списку джерел, наведених у кінці розділу.

Міжпроцесова взаємодія (interprocess communication, IPC) передбачає використання додаткових засобів для обміну даними між процесами та забезпечення їх коректної взаємодії. Нагадуємо, що за замовчуванням процеси мають захищені адресні простори, а тому не мають доступу до адресних просторів одне одного.

Можна виокремити *три ключові аспекти* проблеми організації міжпроцесової взаємодії.

1) як передати дані від одного процесу до іншого?

2) як уникнути зіткнення двох процесів у критичних ситуаціях?

3) як узгодити послідовність дії процесів?

Аспекти (2) і (3) також стосуються й потоків, точніше взаємодії двох і більше потоків у межах одного процесу. Аспект (1) стосується передусім процесів, оскільки потоки одного процесу мають спільний адресний простір, і обмін даними між ними можна організувати без додаткових засобів.

Критичний ресурс та критична область. Ресурс, до якого можуть одночасно звертатися декілька процесів (потоків), називається спільно використовуваним ресурсом.

Область програми, яка передбачає звернення до спільно використовуваного ресурсу, називається **критичною областю** (critical region, критична секція, critical section). Такий спільно використовуваний ресурс також називають **критичним ресурсом**, що підкреслює важливість цього ресурсу одразу для кількох процесів (потоків) та потенційні проблеми у зв'язку з цим.

Для того, аби проілюструвати згадані вище проблеми з критичним ресурсом, наведемо два приклади. Перший буде використовувати потоки, другий — процеси. Реальні ситуації у прикладах максимально спрощено.

Приклад 1 (про банк) [4]

Нехай для обслуговування кожного клієнта виділено окремий потік. Розглянемо ситуацію: двоє клієнтів хочуть одночасно перерахувати кошти на один рахунок.

Введемо наступні позначення:

- K1 — потік першого клієнта, K2 — потік другого клієнта;
- `zah_suma` — загальна сума на рахунку (глобальна змінна, до якої мають доступ усі потоки);
- `nove_nad` — нове надходження.

Операція зміни розмірів загальної суми на рахунку виглядатиме наступним чином:

```
zah_suma = zah_suma + nove_nad
```

Насправді ця операція складається з таких дій:

- потік обслуговування користувача X зчитує поточне значення загальної суми на рахунку із глобальної змінної `zah_suma`;
- потік обслуговування користувача X збільшує зчитане значення загальної суми на розмір нового надходження `nove_nad`, після чого оновлює значення глобальної змінної `zah_suma`.

Нехай:

- сума на рахунку становить 2000 грн;
- потік K1 має покласти на рахунок 1000 грн, а потік K2 — 500 грн.

Можливі різні варіанти розвитку подій. Спершу наведемо приклад сприятливого розвитку подій.

1) Потік K1 зчитує значення `zah_suma` (2000), збільшує його на 1000, виходить 3000. Потік K1 оновлює глобальну змінну: `zah_suma = 3000`

2) Потік K2 зчитує значення `zah_suma` (3000), збільшує його на 500, виходить 3500. Потік K2 оновлює глобальну змінну: `zah_suma = 3500`

Цього разу все правильно. Сума на рахунку є такою, якою й мала стати, тобто 3500 грн.

Однак можливі й менш сприятливі ситуації. Скажімо, така.

1) Потік К1 зчитує значення `zah_suma` (2000).

2) Потік К2 зчитує значення `zah_suma` (2000).

3) Потік К1 збільшує зчитане на кроці (1) значення `zah_suma` (2000) на 1000. Виходить 3000. Потік К1 оновлює глобальну змінну: `zah_suma = 3000`.

4) Потік К2 збільшує зчитане на кроці (2) значення `zah_suma` (2000) на 500. Виходить 2500. Потік К2 оновлює глобальну змінну: `zah_suma` = 2500.

Тепер результат неправильний: загублено 1000 грн.

Проблема полягає у тому, що передбачити, як саме розвиватимуться події, - неможливо, а тестувати всі варіанти послідовності виконання потоків дуже складно у реальних програмах.

Розгляньмо ще один приклад.

Приклад 2 (про принтер) [3]

Спрощено розглянемо роботу спулера (spooler) — фонового процесу, відповідального за чергу друку. Орієнтовна послідовність роботи спулера є наступною.

- Коли процесу потрібно роздрукувати файл, процес поміщає ім'я цього файлу у каталог спулера.
- Спулер періодично перевіряє каталог спулера на наявність файлів, відправлених на друк. Якщо каталог непорожній, спулер вибирає звідти файли (по черзі) і друкує їх.
- Ім'я надрукованого файлу вилучається з каталогу спулера.

Нехай процесу А та процесу Б майже одночасно знадобилося надрукувати кожному свій файл. Цього разу одразу змоделюємо ситуацію, коли розвиток подій був несприятливим (рис. 8.1).

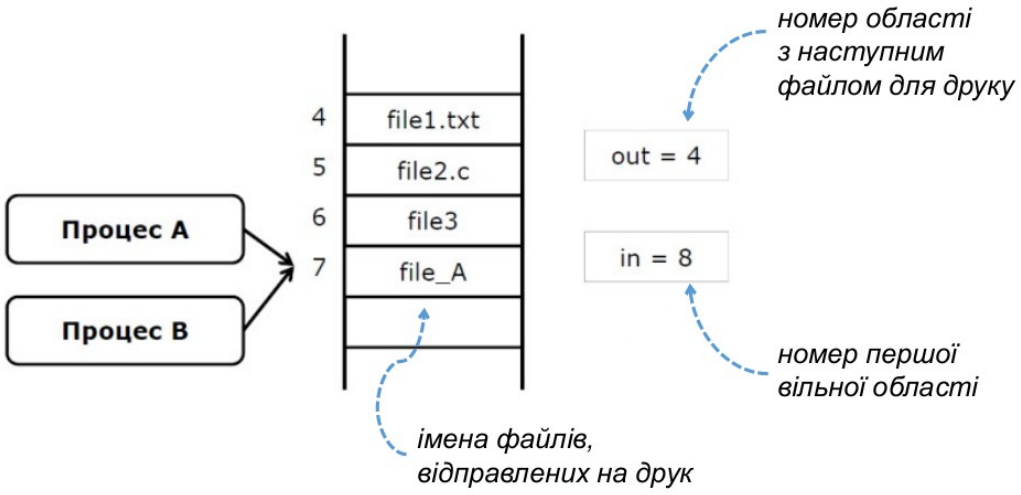

Рис. 8.1. Моделювання конфлікту між двома процесами під час взаємодії з каталогом спулера[22]

[22] Схему адаптовано за книгою: Tanenbaum, Bos. Modern Operating Systems, 2014

1) Процес А зчитує значення змінної `in` (7) і зберігає його у локальній змінній `next_free_slot_a` (наступна вільна область).

2) Відбувається переривання. Керування передається процесу Б.

3) Процес Б також зчитує значення змінної `in` (7) і зберігає його у своїй локальній змінній `next_free_slot_b`.

4) Процес Б зберігає ім'я свого призначеного для друку файлу в області номер 7. Змінній `in` присвоюється значення 7+1=8.

5) Відновлюється виконання процесу А. За даними процесу А, перша вільна область — досі область номер 7. Процес А записує свій відправлений на друк файл в область номер 7. При цьому він:

- стирає (перезаписує) ім'я файлу, раніше записане процесом Б;
- повторно присвоює змінній `in` значення 8.

Результатом описаної ситуації буде друк лише файлу процесу А. Водночас, із точки зору спулера все гаразд: нумерацію не порушено, друк відбувається.

Ситуація, за якої два чи більше процесів зчитують або записують деякі спільні дані, а кінцевий результат залежить від того, який процес і коли саме виконується, називається **змаганням**, або **перегонами** (race).

Задача синхронізації полягає у тому, аби забезпечити, щоб спільне використання різними процесами (потоками) критичного ресурсу відбувалося взаємно несуперечливо.

Це досить складна і багатогранна задача, тож доцільно поділити її на підзадачі, виконати які простіше.

Підзадачі задачі синхронізації:

1) у кожний момент часу у критичній області має перебувати обмежена кількість процесів (потоків);

2) неважливо, який процес (потік) працює швидше;

3) що б не сталося з процесом (потоком) за межами критичної області, це не повинно вплинути на подальшу роботу інших процесів (потоків) у критичній області;

4) процес має звільнити критичний ресурс за скінченний проміжок часу.

Найпростіший спосіб вирішити задачу синхронізації — не переривати процес, доки той сам не вийде з критичної області.

Водночас, цей спосіб нівелює потенційні переваги паралелізму. До того ж, один процес може зайняти ЦП надовго (це не підходить для інтерактивних ОС, наприклад). Якщо процес, крім того, зазнає краху у критичній області, то це призведе до краху всієї системи.

Враховуючи сказане вище, зазвичай приймають рішення переривати процес, коли є така потреба, а для уникнення проблем з доступом до критичного ресурсу застосовувати різноманітні синхронізаційні механізми, зокрема семафори, м'ютекси, умовні змінні, монітори, бар'єри, блокування читання-запису тощо.

У даному посібнику здійснено загальний огляд частини синхронізаційних механізмів. Детальніше ці питання висвітлено у джерелах, наведених у кінці розділу.

2. Синхронізація

Розгляд прикладів синхронізаційних механізмів розпочнемо з семафорів. Тут і далі пояснення будуть за потреби супроводжуватися прикладами із використанням псевдокоду.

Семафори. Алгоритм семафорів запропонував Едсгер Дейкстра (Edsger Wybe Dijkstra, нідерландський вчений) у 1965 році.

Ідея семафорів полягає у наступному: семафор дозволяє увійти до критичної області не більш як `n` потокам. Якщо до критичної області намагається потрапити `(n+1)`-й потік, він блокується, і його виконання може бути поновлене, коли семафор знову дозволятиме вхід.

Сам семафор є лічильником, який може набувати значень від 0 до `n`. Семафор підтримує три неперервні операції:

1) `init` – ініціалізація семафору

```
sem = n;
```

2) `enter` – вхід (зменшення семафору)

```
if (sem > 0) sem--;
else sleep(); // переводимо потік у стан очікування (сну)
// кажуть: "потік заблокований на семафорі"
```

3) `leave` – вихід (збільшення семафору)

```
sem++;
if (waiting_threads()) wakeup (some_thread);
```

Семафори є достатньо простим синхронізаційним механізмом. Водночас, вони мають низку недоліків. Зокрема, програму із використанням семафорів складно документувати, оскільки для кожного семафору потрібно прописувати, за що саме той відповідає. Потрібно також стежити за вчасним звільненням семафорів, оскільки можна виконати `enter` і забути про `leave`. Ще одним недоліком є небезпека виникнення *взаємного блокування потоків*.

Приклад 3. Про тотальну безвихідь

Нехай маємо код із використанням двох семафорів та двома потоками. Обидва потоки в певний момент взаємодіють і з першим, і з другим семафором.

Далі наведемо кілька фрагментів псевдокоду.

```
// Фрагмент псевдокоду 1: Ініціалізація семафорів
semaphore1.init(1);
semaphore2.init(1);
```

Як бачимо, обидва семафори допускають перебування лише одного потоку в один момент часу. Можливо, це пов'язано з особливістю критичних ресурсів, доступ до яких захищають ці семафори.

```
// Фрагмент псевдокоду 2: Частина функції, що виконується у потоці 1
```

```
semaphore1.enter();
semaphore2.enter();
//...
semaphore2.leave();
semaphore1.leave();
```

Зверніть увагу: потік 1 входить на другий семафор, не звільнивши першого. Запам'ятайте цей момент.

```
// Фрагмент псевдокоду 3: Частина функції, що виконується у потоці 2
semaphore2.enter();
semaphore1.enter();
//...
semaphore1.leave();
semaphore2.leave();
```

Потік 2, подібно до потоку 1 також входить на семафор, не звільнивши іншого семафору, але цього разу йдеться про вхід на перший семафор, коли другий семафор досі не звільнено.

Якщо не пощастить, виконання наведених вище фрагментів коду може відбуватися у наступній послідовності.

1) Обидва семафори проініціалізовано. Лічильники дорівнюють 1. Вхід на семафори дозволено.

2) Потік 1 увійшов на семафор 1. Семафор 1 закритий на вхід (0).

3) Потік 2 увійшов на семафор 2. Семафор 2 закритий на вхід (0).

4) Потік 1 має намір увійти на семафор 2, але семафор 2 зайнятий. Потік 1 блокується на семафорі 2.

5) Потік 2 має намір увійти на семафор 1, але семафор 1 зайнятий. Потік 2 блокується на семафорі 1.

Таким чином, складається наступна ситуація. Потік 1 чекає на звільнення семафору 2, а тому не виконується далі і, як наслідок, не звільняє семафор 1. З іншого блоку, потік 2 чекає на звільнення семафору 1 (зайнятого потоком 1), а тому теж не може виконуватися далі, а тому не звільняє семафор 2 (потрібний потоку 1).

Це приклад взаємного блокування, яке можливе і на вході, і на виході з семафорів.

Взаємне блокування (deadlock – букв. глухий кут, безвихідь) – ситуація, за якої для подальшого виконання одного процесу (потоку) необхідні ресурси, захоплені іншим процесом (потоком), котрому, в свою чергу, потрібні для виконання інші ресурси, теж захоплені іншим процесом (потоком).

Взаємне блокування можливе для двох, а можливе й для більшої кількості потоків. Відповідальність за уникнення взаємного блокування на семафорах покладається на програмістів, які створюють багатопотоковий код.

М'ютекси. М'ютекс є різновидом семафора. Слово "mutex" є штучно утвореним з двох англійських слів — **mut**ual (взаємне) та **ex**clusion (виключення).

М’ютекс подібний до семафора, проініціалізованого значенням 1: доступ до критичного ресурсу, захищеного м’ютексом, у конкретний момент часу має лише один потік, доступ інших потоків до ресурсу на цей момент виключається.

М'ютекс може перебувати у двох станах:

- заблокований (ціле додатне число);
- незаблокований (нуль).

Якщо м'ютекс у незаблокованому стані, він пропускає потік у критичну область, інакше – ні.

Виокремлюють також ***рекурсивний м'ютекс*** – це різновид м'ютекса, який може бути повторно зайнятий тим самим потоком. Якщо потік, що намагається зайняти рекурсивний м’ютекс, це потік-власник (потік, що створив м’ютекс), то йому буде дозволено доступ, якщо інший потік — його буде заблоковано на м’ютексі, доки м’ютекс не звільниться.

М'ютекси часто використовують разом з *умовними змінними*: м’ютекс буде блокувати потік доти, доки не буде виконано певну умову, за виконання чи невиконання якої й відповідає умовна змінна.

Монітори. Механізм моніторів запропонували Б. Хансен (1973) та Ч. Хоар (1974). Монітори належать до високорівневих синхронізаційних примітивів, тому вони підтримуються не всіма мовами програмування. Втім, саму ідею монітору можна реалізувати й іншими мовами.

До мов програмування, які підтримують монітори, належать, зокрема, наступні мови: Ada, C# та інші мови на базі технології .NET, Concurrent Pascal, D, Java (ключове слово synchronized), Mesa, Modula-3, Ruby Squeak Smalltalk, uC++ (інша назва – μC++).

Розгляньмо невеликий приклад монітору на псевдокоді.

```
monitor mon
integer i;
condition c;
procedure pr1();
…
end;
procedure pr2();
…
end;
end monitor;
```

У цьому моніторі описано дві процедури — `pr1()` та `pr2()`. Також у моніторі використовується умовна змінна c.

Ідея монітору полягає в тому, що у кожний момент часу в моніторі може виконуватися лише одна процедура монітору.

Коли якийсь процес *P* викликає одну з процедур монітору, спершу здійснюється перевірка, чи не виконується вже у цей момент якась процедура цього ж монітору. Якщо процедура виконується, процес P буде призупинено, доки процедура не завершиться. Якщо жодна процедура монітору не виконується - процес P зможе виконати свою процедуру (увійти до монітору).

Монітори застосовуються у поєднанні з умовними змінними. Умовна змінна дозволяє процедурі, яка не може продовжити свою роботу, сигналізувати про це, щоб могли виконуватися інші процедури.

Бар'єри. Бар'єри використовують у програмах, у яких роботу можна розподілити на послідовні етапи, яким притаманні наступні характеристики:

- кожний етап виконує окрема група потоків;
- черговий етап не може початися, доки не буде повністю завершено попередній.

Бар'єр блокує виконання групи потоків до тих пір, поки кількість потоків, які очікують на бар'єрі не досягне заданого значення.

Більше про ці та інші механізми синхронізації — у джерелах наприкінці цього розділу.

Переходимо до обговорення проблеми, яка стосується лише процесів — проблеми передачі даних між процесами.

3. Передача даних між процесами

Нагадуємо, що питання передачі даних стосується процесів і не є актуальним для потоків, які і так мають спільний адресний простір. Втім, обмін даними між *потоками різних процесів* таки стосується цього пункту й фактично зводиться до передачі даних між процесами, у межах яких виконуються ці потоки.

До основних категорій засобів обміну даними між процесами належать:

- сигнали;
- спільно використовувана пам'ять;
- передача повідомлень;
- відображувана пам'ять.

Розгляньмо кожну із наведених категорій.

Сигнали (signals). У випадку сигналів між процесами передається мінімальна кількість інформації. Сигнали використовуються, як правило, для повідомлення процесу про настання деякої події.

Як приклади сигналів, наведемо сигнали, які передаються у Linux процесу, який ви потрібно завершити. Сигнал `SIGTERM` (код `15`) передбачає звичайне завершення процесу (так зване “ввічливе завершення”). Таку пропозицію завершитися процес може проігнорувати — наприклад тоді, коли виконується в фоні. Натомість сигнал `SIGKILL (9)` відповідає примусовому завершення процесу (своєрідне “завершуйся негайно!”). Цей сигнал за нормального функціонування системи процес проігнорувати не може.

Спільно використовувана пам'ять (shared memory). Такі засоби цілком відповідають назві: два чи більше процеси спільно використовують деяку ділянку пам'яті, яка й називається спільною пам’яттю.

Процеси попередньо приєднують цю ділянку до свого адресного простору, для чого мусять мати відповідні права. Синхронізацію доступу до даних у спільно використовуваній пам'яті програміст реалізує самостійно.

Передача повідомлень (message passing). Засоби з цієї категорії дещо схожі на сигнали, але передбачають передавання більшого обсягу інформації, ніж у випадку сигналів. Процеси (чи потоки різних процесів), які беруть участь у

передачі повідомлень, можуть виконуватися на одному комп'ютері або на різних комп'ютерах мережі.

Повідомлення бувають фіксованої або змінної довжини, а спільно використовуваних даних як таких немає, оскільки відправник перестає взаємодіяти з вмістом повідомлення після чого відправлення, а одержувач починає взаємодіяти з цим вмістом лише після його одержання.

Обмінюватися повідомленнями можна з конкретним процесом, але можливо також відправляти повідомлення, прийняти які може будь-який процес. У другому випадку процеси, що обмінюються повідомленнями, можуть навіть не знати про існування одне одного.

Відображувана пам'ять (mapped memory) є досить цікавим способом організувати міжпроцесову взаємодію, що найчастіше реалізується через файли, відображувані у пам'ять (memory-mapped files).

У випадку файлу, відображуваного у пам’ять, деяку частину адресного простору процесу однозначно пов'язують зі спеціальним файлом. Будь-які зміни в адресному просторі викликають аналогічні зміни у файлі. Інші процеси також можуть використовувати дані цього файлу через аналогічний механізм, таким чином отримуючи доступ і до частини адресного простору відповідного процесу.

Контрольні запитання

1) Порівняйте аспекти міжпроцесової та міжпотокової взаємодії? Чи є серед них аспекти, які стосуються міжпроцесової взаємодії, але не стосуються міжпотокової? З чим це пов’язано?
2) Що таке критичний ресурс? критична область?
3) Що таке ситуація змагання (перегонів)? Опишіть проблеми, які може породжувати ця ситуація, на прикладі.
4) Сформулюйте задачу синхронізації та відповідні підзадачі. Які основні два підходи до вирішення задачі синхронізації можна застосувати?
5) Дайте загальну характеристику роботи семафорів.
6) Поясніть суть явища взаємного блокування. Наведіть приклад.
7) Дайте загальну характеристику роботи м’ютексів. Який м’ютекс називають рекурсивним?
8) Дайте загальну характеристику роботи моніторів. Як пов’язані монітори та умовні змінні? Наведіть приклади мов програмування, які підтримують монітори.
9) Дайте загальну характеристику роботи бар’єрів.
10) Опишіть та порівняйте основні категорії засобів обміну даними між процесами (сигнали, спільно використовувана пам’ять, передача повідомлень, відображувана пам’ять).

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 6-8.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 5-6.

3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 2 (2-3), chapter 6.

4. В. А. Шеховцов, Операційні системи: Підручник. К.: Видавнича група BHV, 2005. — Розділи 5-6.

Розділ 9

Віртуалізація

1. Поняття про віртуалізацію

Багатозначність терміну. Терміни "віртуалізація", "віртуальний" вживаються у багатьох областях знань, зокрема у філософії, політології, психології, соціології, економіці, науках про освіту тощо — і звісно, у галузі інформаційних технлогій. Відповідно, тлумачення цих термінів відрізнятиметься залежно від обраної області.

З філософської точки зору **віртуалізацію** можна розглядати як процес переходу від фактичної (актуальної) реальності до віртуальної (потенційної) реальності через певну діяльність людини (енергію) [4].

У галузі інформаційних технологій віртуалізацію тлумачать *у ширшому та вужчому розумінні*. Віртуалізація у ширшому розумінні передбачає створення абстракцій для реальних обчислювальних ресурсів, які отримують у своє розпорядження користувачі замість приховуваних таким чином реальних ресурсів. Працювати з такими абстракціями користувачу та прикладному програмісту зручніше, ніж взаємодіяти з обчислювальними ресурсами напряму, адже, маючи справу з абстракцією певного ресурсу, користувач не обов'язково мусить знати деталі доступу до цього ресурсу. Такого роду абстракції надає операційна система загалом (процеси, потоки, адресні простори, файли і т.п.), і про це йшлося у розділі 1 даного посібника.

Однак часто віртуалізацію у галузі інформаційних технологій тлумачать вужче, і саме вужче розуміння віртуалізації найкраще відповідає технологіям, які дозволяють віртуалізувати цілу операційну систему чи окремі її частини. Саме вужче розуміння й візьмемо за основу робочого означення віртуалізації.

Віртуалізація – поняття, що об'єднує технології, засоби, методи тощо, яким притаманні три головні риси (рис. 9.1):

(1) поділ ресурсів одного фізичного комп'ютера на декілька взаємно незалежних віртуальних середовищ або об'єднання ресурсів кількох фізичних комп'ютерів в одне віртуальне середовище;

(2) оперативність переходу з одного віртуального середовища в інше;

(3) приховування реальних фізичних ресурсів та заміна їх абстракціями.

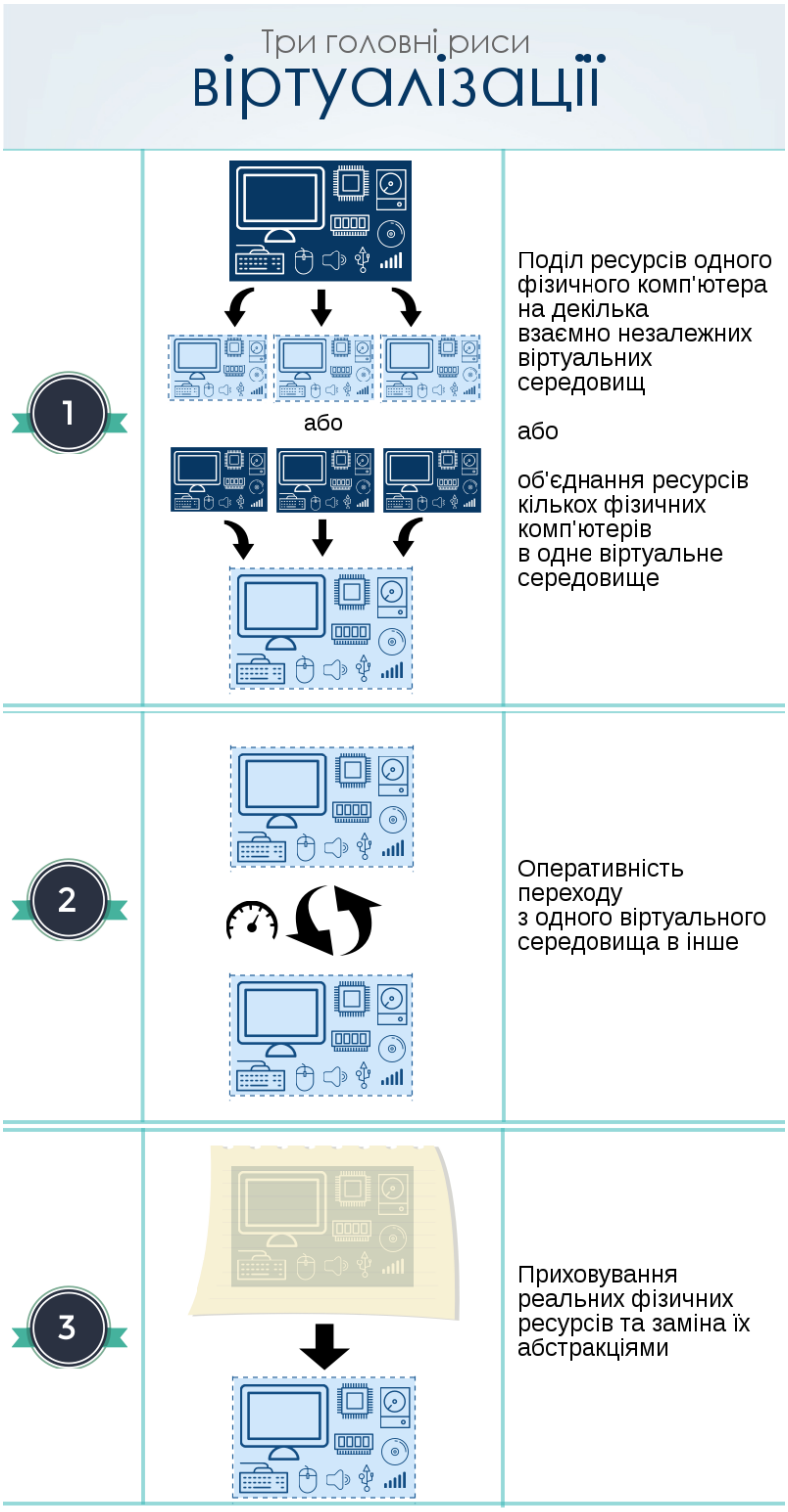

Рис. 9.1. Три головні риси віртуалізації (згідно з робочим означенням)

Різні **програмні засоби віртуалізації** належать до різних **технологій віртуалізації**. Однією з найпопулярніших технологій віртуалізації є віртуальні машини.

Віртуальна машина / ВМ (virtual machine / VM) — змітований у межах реального комп'ютера віртуальний комп'ютер, на який може бути встановлено окрему ОС з окремим ядром.

Гіпервізор (hypervisor) — спеціальне програмне забезпечення, яке розподіляє ресурси між ВМ, організовує взаємодію ВМ з фізичним обладнанням, надає користувачу інтерфейс для керування ВМ.

Важливо розрізняти віртуальні машини (результат віртуалізації) та гіпервізори (засоби віртуалізації). Говорити “віртуальна машина VirtualBox” некоректно.

2. З історії віртуалізації

60-ті роки XX ст.: віртуалізація задля багатозадачності. Терміни "віртуалізація" та "віртуальна машина" застосовувалися ще у 60-х роках XX ст. і були пов'язані з концепцією багатозадачності. Під багатозадачністю (або псевдопаралелізмом) розуміють режим, за якого декілька задач одночасно перебувають в оперативній пам'яті та почергово виконуються на центральному процесорі. Багатозадачність доби інтегральних схем полягала у тому, що кожній задачі відводився свій розділ пам'яті, і поки одна задача очікувала на завершення роботи пристроїв введення-виведення (наприклад, зчитування даних), інша могла тим часом виконуватися. Така багатозадачність реалізовувалася шляхом віртуалізації.

Віртуалізація у VM/370. Розглянемо механізм віртуалізації у VM/370, операційної системи для комп'ютерів IBM System 370 (рис. 9.2). Кожний користувач системи працював з окремим образом всього апаратного забезпечення фізичної машини – віртуальною машиною. Віртуальні машини різних користувачів могли виконуватись паралельно, а користувачі — взаємодіяти з комп'ютером одночасно (режим поділу часу). Така паралельність виконання мала апаратну підтримку, тобто забезпечувалася відповідними технологіями на рівні процесора (важливий факт, до якого ми ще повернемось). Запуск віртуальних машин та керування ними здійснювала окрема програма у складі VM/370 – CP 67, монітор віртуальних машин. Монітор віртуальних машин працював на рівні 1 і реалізовував багатозадачність. На рівні 2 виконувалися окремі віртуальні машини, на яких вже було запущено користувацькі операційні системи (системи пакетної обробки, орієнтовані на опрацювання даних, або інтерактивні системи, орієнтовані на взаємодію з користувачем).

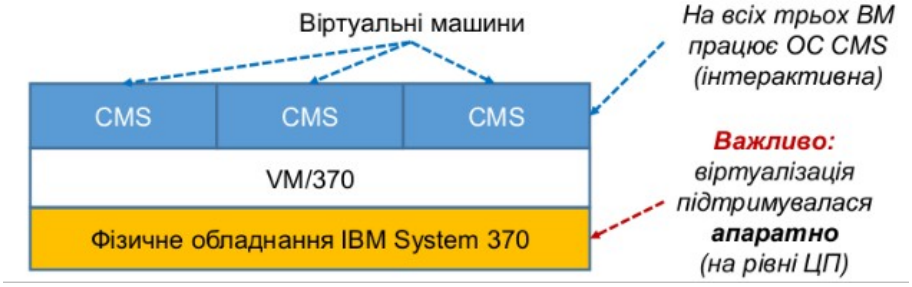

Рис. 9.2. Схематичне подання віртуалізації у VM/370

Скажімо, ми маємо справу з однокористувацькою інтерактивною системою CMS, і вона, як уже йшлося, працює на рівні 2 на окремій віртуальній машині. Коли під час роботи CMS виникає системний виклик, вона перехоплює його сама і сама дає відповідні команди обладнанню. Проте при цьому вона має справу не з фізичним, а з віртуальним обладнанням, земульованим для неї монітором віртуальних машин. Для CMS створюється ілюзія, ніби це обладнання — реальне, тимчасом як насправді з фізичним обладнанням взаємодіє система VM/370, перехоплюючи команди до віртуального обладнання й виконуючи їх на обладнанні реальному.

x86: кому потрібна ця віртуалізація на ПК? Оскільки у добу інтегральних схем комп'ютери розміщувалися в обчислювальних центрах, і про окремий комп'ютер для кожного користувача не йшлося, то описаний механізм був покликаний розподілити обчислювальні потужності однієї фізичної машини між багатьма користувачами. Із появою та поширенням персональних комп'ютерів потреба у розвитку такої технології для цих комп'ютерів, на перший погляд, відпала, адже тепер дедалі більше людей та організацій могли придбати власні комп'ютери, а не ділити їх.

Згодом ідея віртуалізації набула нової актуальності. Адже вона могла б дозволити організувати одразу декілька віртуальних комп'ютерів всередині одного фізичного комп'ютера — тепер уже персонального. Однак тепер для цього з'явилася суттєва перепона, для розуміння якої треба спершу з'ясувати суть поняття службової інструкції.

Службова інструкція (службова команда, sensitive instruction) — команда ЦП, необхідна для роботи ОС й розрахована на виконання лише на найвищому рівні привілегій.

Якщо службова інструкція надходить від програми, що працює на нижчому рівні привілеїв (від ОС, запущеної не безпосередньо, а на ВМ), то таку інструкцію буде виконано інакше або не буде виконано зовсім.

Прикладами службових команд: команди введення-виведення, команди зміни налаштувань MMU тощо. Тобто йдеться про команди, необхідні для роботи ОС.

У VM/370 службові інструкції перехоплював ЦП. Він же робив відповідні виклики монітора віртуальних машин (це апаратна віртуалізація — hardware-assisted virtualization). Але в x86 цього передбачено не було.

System/360, System/390, z-серії: тим часом у світі мейнфреймів. Важливим та дещо іронічним є те, що в ОС System/360, а згодом в System/390 та z-серіях IBM апаратна підтримка віртуалізації лишалися і наявна дотепер.

DISCO, VMware, паравіртуалізація, віртуальні контейнери, WINE: герої йдуть в обхід. Були спроби обійти апаратну неспроможність і реалізувати віртуалізацію повністю програмно.

Проблема віртуалізації архітектури x86 досліджувалася тривалий час. Вимоги до комп'ютерної архітектури для уможливлення віртуалізації на цій архітектурі сформульовано ще у 1974 році у відомій статті Джеральда Попека (Gerald Popek) та Роберта Ґолдберга (Robert Goldberg) "Формальні вимоги для віртуалізації архітектур третього покоління" ("Formal Requirements for Virtualizable Third Generation Architectures") [4]. Архітектура x86 цим вимогам не відповідала. Те саме стосується майже всіх архітектур (за невеликим винятком, про який іще буде сказано трохи далі) аж до 2005 року. У такій ситуації логічними є спроби знайти програмне рішення, що дозволило б обійти означену

апаратну неспроможність. Програмна технологія, яка давала задовільну продуктивність, виникла лише у 90-х роках XX ст. у межах проекту Disco (Стенфордський університет). Однак значно відоміша інша назва, пов'язана з цією. Дослідники зі Стенфорда, які займалися проєктом Disco, стали засновниками широко відомої нині компанії Vmware, а у 1999 році ця компанія випустила своє перше програмне рішення з віртуалізації.

Ідея Disco та Vmware полягала у тому, щоб перехоплювати проблемні команди й замінювати їх безпечними послідовностями коду. Це одержало назву технології **динамічної трансляції**, або **бінарної трансляції** (dynamic translation, binary translation). Змін до джерельних кодів гостьової ОС при цьому не вноситься.

Інший метод — метод **паравіртуалізації** (paravirtualization) передбачає заміну службових інструкцій безпосередньо у коді гостьової ОС (потрібний доступ до джерельних кодів гостьової ОС).

Повні емулятори (pure emulators) імітують усе апаратне забезпечення комп'ютера, включно з процесором. Технологія може нижчу продуктивність, але дозволяє зімітувати машину однієї архітектури на машині іншої архітектури.

Емуляція системних бібліотек (system library emulation) — низка дуже різних за реалізацією технологій, які віртуалізують не всю ОС, а її системні бібліотеки. Наприклад, коли програму для Windows запущено у Linux за допомогою WINE, то кожний API-виклик цієї програми перехоплюється, й імітується таке його опрацювання, ніби програму й справді запущено у Windows.

Віртуальні контейнери (virtual containers), або віртуалізація рівня ОС (OS level virtualization) - технологія, що передбачає створення і паралельну роботу в межах одного фізичного комп'ютера окремих віртуальних середовищ (контейнерів). Контейнери відносно незалежні, але використовують одне ядро.

Головна перевага віртуальних контейнерів полягає у тому, що вони швидкі та потребують менше ресурсів, ніж повноцінні віртуальні машини.

Віртуальні контейнери розраховані на Unix-подібні ОС. У деяких випадках можуть запускатися на Windows як на основній ОС (наприклад, Docker), але для цього основна ОС має надати контейнерам віртуальну машину.

Наведений тут перелік технологій віртуалізації неповний. Більше технологій віртуалізації наведено у п. 3 даного посібника.

2005: повернення апаратної віртуалізації для ПК. У 2005 році компанії Intel та AMD представили свої перші процесори з підтримкою віртуалізації (Intel VT-x та AMD-V). Обоє спираються на досвід IBM VM/370 із деякими індивідуальними особливостями.

3. Огляд технологій віртуалізації

Важливо розуміти, що єдина загальноприйнята систематизація технологій віртуалізації відсутня. До того ж, конкретні засоби віртуалізації часто використовують поєднання кількох технологій. Технології віртуалізації продовжують розвиватися, і нові технології не завжди вдається одразу ж прив'язати до наявних систематизацій. У цьому посібнику наведено одну з можливих систематизацій технологій віртуалізації за напрямом та за методом

За напрямом віртуалізації виокремимо: віртуалізацію серверів, віртуалізацію настільних ОС, віртуалізацію прикладних програм та віртуалізацію робочого столу (рис. 9.3).

Рис. 9.3. Систематизація технологій віртуалізації за напрямом

Віртуалізація серверів (server virtualization). Назва напряму цілком промовиста: віртуалізується сервер (веб-сервер, файловий сервер, поштовий сервер, сервер для запуску деякого спеціалізованого ПЗ тощо). При цьому необхідність використання реальної машини, де фізично розміщені такі сервери, як настільного ПК може бути як наявною (комп'ютер стоїть у вас вдома, і ви періодично запускаєте з нього віртуальний сервер), так і відсутньою (комп'ютер виділено для функціонування лише як сервер або як декілька серверів). Від цього нюансу, а також від ряду інших, буде залежати вибір конкретної технології віртуалізації.

Загалом, віртуалізація серверів здійснюється на основі віртуальних машин, віртуальних контейнерів чи їх деякого поєднання.

Технологія **віртуальних машин** (virtual machines) передбачає імітацію у межах реального комп'ютера одного або кількох віртуальних комп'ютерів, на кожний з яких може бути встановлено окрему ОС з окремим ядром. Кожна віртуальна машина використовує ресурси реальної машини (основну пам'ять, час центрального процесора, дисковий простір, мережні пристрої і т.д.). У тому числі, віртуальній машині виділяється деякий обсяг основної пам'яті від загального обсягу основної пам'яті фізичного комп'ютера, віртуальний жорсткий диск, який є файлом спеціального формату на реальному жорсткому диску.

Розподілом ресурсів між віртуальними машинами та організацією взаємодії віртуальних машин з обладнанням реальної машини займається *гіпервізор* (hypervisor). Він же надає користувачеві інтерфейс для створення віртуальних машин та керування ними.

Розрізняють гіпервізори I типу, гіпервізори II типу та гібридні гіпервізори.

Гіпервізори I типу (type 1 hypervisors) також називають гіпервізорами, "виконуваними на голому залізі" (bare-metal hypervisors). Назва відповідає суті: такий гіпервізор не має під собою жодної ОС і, власне, сам є мінімальною ОС.

До гіпервізорів I типу можна віднести VMware ESX.

Гіпервізори II типу (type 2 hypervisors) працюють як ПЗ у межах уже встановленої на комп'ютері ОС (основної ОС). Таким чином, засобами гіпервізора II типу всередині основної ОС можна запускати одну чи декілька віртуальних машин, кожну – з власною ОС (гостьовою ОС).

Прикладами гіпервізорів II типу є Oracle VirtualBox, VMware Player, VMware Workstation, KVM тощо.

Виділяють також **гібридні гіпервізори** (hybrid hypervisors), які є поєднанням гіпервізорів I та II типів. Дана технологія передбачає наявність, з одного боку, гіпервізора I типу, що працює безпосередньо на апаратному забезпеченні, а з іншого – спеціальної сервісної ОС, котра функціонує під управлінням гіпервізора I типу. На практиці гібридні гіпервізори ближчі до гіпервізорів I типу, тому часто їх не виокремлюють у повноцінний клас, а відносять до гіпервізорів I типу. Гібридними можна вважати, зокрема, гіпервізори Microsoft Hyper-V, Citrix Xen Server, Oracle VM Server.

Втім, незалежно від типу гіпервізора, віртуальні машини об'єднує згадана вище можливість встановити на кожну віртуальну машину окрему ОС із окремим ядром. Це означає, що на одній віртуальній машині може бути, наприклад, Linux, на другій – Windows, на третій – FreeBSD і т.д., і всі ці віртуальні машини функціонуватимуть у межах одного фізичного комп'ютера, у тому числі паралельно. Така особливість відрізняє технологію віртуальних машин від технології віртуальних контейнерів.

Технологія **віртуальних контейнерів** (virtual containers), або віртуалізація рівня ОС (operating system level virtualization), передбачає створення і паралельну роботу в межах одного фізичного комп'ютера окремих віртуальних середовищ (контейнерів). Контейнери відносно незалежні, проте спільно використовують єдине ядро ОС. Це робить віртуальні контейнери загалом швидшими за віртуальні машини. Водночас, звідси випливає й головне обмеження цієї технології: неможливо мати в одному контейнері одну ОС, а в другому – цілком іншу.

Прикладами реалізації технології віртуальних контейнерів є OpenVZ, Parallels Virtuozzo, Linux-VServer, FreeBSD-jail, LXC, Canonical LXD, Docker та ін.

Зі сказаного вище зрозуміло, що значно природнішим вибором для віртуалізації серверів є віртуальні машини на базі гіпервізорів I типу та гібридних гіпервізорів або віртуальні контейнери. Гіпервізори II типу, які передбачають встановлення поверх основної ОС, мають у своєму розпорядженні менше ресурсів, оскільки значна частина фізично доступних ресурсів витрачається на функціонування основної ОС. Втім, якщо віртуальний сервер розрахований на пробну роботу, тимчасову роботу, створюється з навчальною метою і т.д., гіпервізори II типу, навпаки, можуть стати доцільнішим вибором, бо потребують відносно незначних організаційних зусиль (з цієї точки зору вони програють лише віртуальним серверам у хмарі, про які йтиметься у наступному розділі).

Віртуалізація настільних ОС (local desktop virtualization) переважно передбачає застосування гіпервізорів II типу (основна ОС + гіпервізор + віртуальні машини з гостьовими ОС всередині, детальніше – див. вище) або віртуальних контейнерів. Така віртуалізація може мати на меті цілий ряд різноманітних застосувань: розробка та налагодження ПЗ під різні ОС (різні дистрибутиви/версії/випуски однієї ОС); запуск застосунків однієї ОС на комп'ютері з іншою ОС; навчання ОС (у тому числі, у межах університетського курсу з ОС). Цей перелік не є вичерпним, і його може бути продовжено. З нього

виключається лише розглянута раніше організація віртуального сервера. Втім, межа між віртуалізацією серверів та віртуалізацією настільних ОС не є чіткою.

Зауваження. Англомовний відповідник "local desktop virtualizatoin" потребує обережного використання, оскільки словосполучення "desktop virtualization" вживається на позначення іншого напряму віртуалізації – віртуалізації робочого столу (див. далі). Віртуалізація настільних ОС у більшості випадків спирається на гіпервізори II типу, вже представлені у межах напряму віртуалізації серверів, тому виділяється лише деякими дослідниками.

У ході **віртуалізації прикладних програм** (application virtualization) відбувається створення віртуального середовища для кожного екземпляра програми. Це дає змогу застосунку працювати у середовищах, які цей за стосунок початково не підтримують. Віртуалізації програмних застосунків суттєво полегшує міграцію застосунків (портативні версії програм). При цьому віртуалізація програмних застосунків потребує менше ресурсів, ніж віртуалізація всієї операційної системи.

Ця технологія застосовується у Citrix XenApp, VMware ThinApp, Microsoft App-V та ін.

Віртуалізація робочого столу (desktop virtualization) передбачає віртуалізацію середовища робочого столу, включаючи застосунки користувача. Віртуалізація робочого столу часто (але не обов'язково) передбачає, що користувач взаємодіє з обчислювальними потужностями віддалено, і його процеси виконуються не на пристрої користувача (ПК, ноутбуці, тонкому клієнті, планшеті, смартфоні тощо), на сервері.

Віртуалізація робочого столу здійснюється на основі двох базових підходів: сервіси віддаленого робочого столу та інфраструктура віддаленого робочого столу.

Сервіси віддаленого робочого столу (remote desktop services, RDS) ґрунтуються на наданні спільного доступу до ОС серверу (одного екземпляру цієї ОС для всіх користувачів).

Прикладами реалізації технології сервісів віддаленого робочого столу є Microsoft Remote Desktop Services, xrdp.

Інфраструктура віддаленого робочого столу (virtual desktop infrastructure, VDI) забезпечує кожному користувачу доступ до окремого екземпляру ОС.

Технологію VDI втілено, зокрема, у Citrix XenDesktop, VMware View.

За методом віртуалізації виділимо, передусім, програмну та апаратну віртуалізацію (рис. 9.4).

Апаратна віртуалізація (hardware assisted virtualization, hardware based virtualization) виконується на базі двох основних технологій: Intel VT-x та AMD-V. Обидві технології, попри відмінності у реалізації, ґрунтуються на спільній ідеї — створення контейнерів для віртуальних машин на апаратному рівні.

Програмну віртуалізацію (software virtualization) можна поділити на повну емуляцію, емуляцію системних бібліотек та квазіемуляцію. Квазіемуляція, у свою чергу, поділяться на динамічну трансляцію та паравіруталізацію.

Суть та приклади перелічних програмних методів віртуалізації було описано у п. 2 цього посібника.

Варто відзначити, що всупереч можливим очікуванням, на практиці технології апаратної віртуалізації за продуктивністю не обов'язково переважають технології програмної віртуалізації [6].

Рис. 9.4. Систематизація технологій віртуалізації за методом

Загалом, у сучасних засобах віртуалізації різні напрями і методи віртуалізації часто поєднуються. Так, гіпервізор II типу KVM застосовує технологію динамічної трансляції для віртуалізації ОС із закритим кодом і технологію паравіртуалізації для віртуалізації ОС із відкритим кодом. Гібридний гіпервізор Oracle VM Server для x86 поєднує технологію паравіртуалізації (для ОС із відкритим кодом) із апаратною віртуалізацією (для ОС із закритим кодом). Програмний засіб Virtuozzo, що початково спирається на технологію віртуальних контейнерів, згодом також додав до свого функціоналу адаптований гіпервізор KVM. Програмний засіб Proxmox VE використовує віртуальні машини на базі KVM (динамічна трансляція і паравіртуалізація) та віртуальні контейнери на основі Open VZ. Цей перелік не є вичерпним, і з розвитком технологій віртуалізації він лише зростає.

Цікавою течією у розвитку технологій віртуалізації є вкладена віртуалізація.

Вкладена віртуалізація (nested virtualization) — термін, що використовується на позначення можливості запуску одного віртуалізованого середовища всередині іншого.

Вкладена віртуалізація — порівняно новий напрям віртуалізації. Відповідний функціонал надається лише деякими засобами віртуалізації та віртуалізаційними платформами, і часто потребує процесора, що підтримує апаратну віртуалізацію. Вкладену віртуалізацію реалізовано зокрема у гіпервізорах KVM, Xen, VMware EXSi, Microsoft Hyper-V, хмарній платформі Oracle Ravello; VirtualBox (раніше — лише для AMD, тепер і для Intel).

Технології віртуалізації є важливою частиною сучасних інформаційних технологій і, зокрема, основною для хмарних технологій.

Контрольні запитання

1) Назвіть та поясніть три основні риси віртуалізації.
2) Коротко опишіть розвиток технологій віртуалізації від системи VM/370 до сучасних технологій. Наведіть приклад технології, яка тривалий час перестала підтримуватися, хоча потреба в ній лишалася, а згодом повернулася на сучаснішій апаратній платформі.
3) Які основні напрями віртуалізації можна виокремити? Які ключові відмінності цих напрямів? Наведіть приклади засобів, які відповідають кожному з них.
4) Що таке віртуальна машина? гіпервізор?
5) У чому ключові відмінності гіпервізорів I та II типу? Які гіпервізори можуть також розглядатися як гібридні гіпервізори. Наведіть приклади гіпервізорів різних типів.
6) Охарактеризуйте технологію віртуальних контейнерів. Яка основна перевага віртуальних контейнерів, порівняно з віртуальними машинами?
7) Які основні методи віртуалізації можна виокремити? Які ключові відмінності цих методів? Наведіть приклади засобів, які відповідають кожному з них.
8) Що таке вкладена віртуалізація?
9) Наведіть приклади поєднання різних технологій віртуалізації в одному засобі.
10) Яке відношення технології віртуалізації мають до хмарних технологій?

Джерела та посилання

1. A. Silberschatz, P. Galvin and G. Gagne, Operating system concepts, 10th ed., Wiley, 2018. — Chapter 18.
2. W. Stallings, Operating Systems Internals and Design Principles, 9th ed., Pearson, 2017. — Chapter 14.
3. A. S. Tanenbaum, H. Bos, Modern operating systems, 4th ed., Pearson, 2014. — Chapter 7.
4. G. J. Popek, R. P. Goldberg. Formal Requirements for Virtualization Third Generation Architectures. Communications of the ACM. 1974. №7(17). P. 412-421.
5. В. Брязкун. До уточнення поняття "віртуалізація" як філософської категорії. Наукові записки Національного університету "Острозька академія". Сер.: Філософія. 2011. Вип. 8. С. 93-103.
6. K. Adams, O. Agesen. A Comparison of Software and Hardware Techniques for x86 Virtualization. Proceedings of ASPLOS'06, 2006.

Навчальне видання

ОСНОВИ ОПЕРАЦІЙНИХ СИСТЕМ

Навчальний посібник

Підготувала

Головня Олена Сергіївна

Формат 60×84/16.
Ум. друк. арк. 7,32.
Безкоштовно

Редактура та комп’ютерне верстання авторські.

Гарнітура Orchidea Pro Md, Mariupol, Midpoint Pro Regular, Ubuntu Mono