МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ ДЕРЖАВНИЙ ЕКОЛОГІЧНИЙ УНІВЕРСИТЕТ

РОЛЬЩИКОВ В. Б.

ОПЕРАЦІЙНІ СИСТЕМИ

Конспект лекцій

Одеса
Одеський державний екологічний університет
2015

УДК 681.3
Р68

Рекомендовано методичною радою Одеського державного екологічного університету Міністерства освіти і науки України як конспект лекцій (протокол №7 від 30.04. 2015 р.)

Рольщиков В. Б.
Операційні системи: конспект лекцій. Одеса, Одеський державний екологічний університет, 2015. 151 с.

В конспекті лекцій з курсу «Операційні системи» розглянуті основні питання архітектури сучасних операційних систем як відкритого, підпорядкованого стандарту POSIX типу – Linux, так і закритого типу – Windows. На початку розглядаються загальні питання керування віртуальною пам’яттю, файловою системою, паралельною роботою із застосуванням процесів і потоків, потім ті самі питання пояснюються на прикладах відповідних сучасних ОС .

Конспект лекцій призначений для студентів третього курсу Одеського державного екологічного університету, які навчаються за кваліфікаційним рівнем «бакалавр» та спеціальністю 05.010101 «Інформаційні управляючі системи та технології».

ISBN 978-966-186-090-1

ЗМІСТ

Вступ.....5
1 Віртуальна пам’ять.....8
1.1 Сторінкова організація пам’яті.....10
1.1.1 Реалізація сторінкової організації пам’яті.....13
1.1.2 Виклик сторінок на вимогу і робоча безліч.....17
1.1.3 Політика заміщення сторінок.....19
1.1.4 Обробка сторінкового переривання.....23
1.1.5 Розмір сторінок і фрагментація.....24
1.2 Сегментація.....27
1.2.1 Способи й алгоритми реалізації сегментації.....32
1.2.2 Перша спроба створення сегментно-сторінкової організації пам’яті.....36
1.3 Віртуальна пам’ять у процесорі Pentium.....38
1.3.1 Рівні захисту пам’яті у процесорі Pentium.....44
1.4 Віртуальна пам’ять UltraSPARC.....45
1.5 Віртуальна пам’ять і кешування.....51
1.6 Питання для самоперевірки.....52
2 Віртуальні команди вводу-виводу.....59
2.1 Файли, їх використання і властивості.....60
2.2 Віртуальні команди вводу-виводу.....62
2.3 Реалізація віртуальних команд вводу-виводу.....64
2.4 Команди керування директоріями.....69
2.5 Питання для самоперевірки.....71
3 Віртуальні команди для паралельної обробки.....74
3.1 Формування процесу.....75
3.2 Стан гонок.....76
3.3 Синхронізація процесів з використанням семафорів.....82
3.4 Питання для самоперевірки.....87

4 Приклади операційних систем.....90
4.1 Коротка історія розвітку ОС UNIX та її узагальнена структура.....90
4.2 Розвиток ОС Windows та її структура.....96
4.3 Приклади віртуальної пам’яті.....105
4.3.1 Віртуальна пам’ять UNIX.....105
4.3.2 Віртуальна пам’ять Windows.....108
4.4 Приклади віртуального вводу-виводу.....111
4.4.1 Віртуальний ввід-вивід у системі UNIX.....111
4.4.2 Віртуальний ввід-вивід в Windows.....123
4.5 Приклади керування процесами.....131
4.5.1 Керування процесами в системі UNIX.....131
4.5.2 Керування процесами в Windows.....136
4.6 Питання для самоперевірки.....140
Висновки.....144
Перелік рисунків.....146
Перелік таблиць.....148

ВСТУП

Сучасний комп’ютер – досить складна система, архітектура якої складається з низки рівнів, і кожен з них додає додаткові функції до рівня, що перебуває під ним. Можна виділити цифровий логічний рівень, мікроархітектурний рівень і рівень команд і т.ін. У курсі операційних систем буде розглянуто функціонування комп’ютера на одному з цих рівнів, власне, на рівні операційної системи. Розгляд інших рівнів входить до задач інших курсів навчання, наприклад, до курсу архітектури і схемотехніки комп’ютерів.

Операційна система (далі скорочено ОС) – це програма, що додає ряд команд і особливостей до тих, які забезпечуються рівнем команд. Звичайно операційна система реалізується головним чином у програмному забезпеченні, але немає ніяких вагомих причин, з яких її не можна було б реалізувати в апаратному забезпеченні (як мікропрограми). Рівень операційної системи показаний на рис. 1.

Рисунок 1 – Розташування рівня операційної системи

Хоча й рівень операційної системи, і рівень команд абстрактні (у тому розумінні, що вони не є реальним апаратним забезпеченням), між ними є важливе розходження. Набір команд рівня операційної системи – це повний набір команд, що є загальнодоступними для прикладних програмістів. Він містить практично всі команди більш низького рівня, а також нові команди, які додає операційна систе-

ма. Ці нові команди називаються системними викликами. Вони звертаються до певної служби операційної системи, зокрема до однієї з її команд. Наприклад, звичайний системний виклик зчитує які-небудь дані з файла.

Рівень операційної системи завжди інтерпретується. Коли користувальницька програма виконує команду операційної системи, наприклад читання даних з файла, операційна система виконує цю команду крок за кроком, точно так само, як мікропрограма виконує, наприклад, команду додавання ADD. Однак, коли програма виконує команду рівня архітектури команд, ця команда виконується безпосередньо мікроархітектурним рівнем без участі операційної системи.

Сучасні ОС – принаймні, широко розповсюджені системи – багато в чому схожі одна на одну. Насамперед, це визначається вимогою можливості перенесення програмного забезпечення. Саме для забезпечення цієї можливості перенесення був прийнятий POSIX (Portable OS Interface based on uniX) – стандарт, що визначає мінімальні функції з керування файлами, міжпроцесної взаємодії й т.ін., які повинна вміти виконувати система.

За сучасними уявленнями, ОС повинна вміти робити таке:

1. Забезпечувати завантаження користувальницьких програм в оперативну пам’ять і їхнє виконання.
2. Забезпечувати роботу із пристроями довгострокової пам’яті, такими як магнітні диски, стрічки, оптичні диски й т.п. Як правило, ОС управляє вільним простором на цих носіях і структурує користувальницькі дані.
3. Надавати більш-менш стандартний доступ до різних пристроїв вводу/виводу, таких як термінали, модеми, друкувальні пристрої.
4. Надавати деякий користувальницький інтерфейс. Слово «деякий» тут застосовано не випадково – частина систем обмежується командним рядком, у той час як інші на 90% складаються із засобів інтерфейсу користувача.

Існують ОС, функції яких цим і вичерпуються. Більше розвинені ОС надають також такі можливості:

1) паралельне (точніше, псевдопаралельне, якщо машина має тільки один

процесор) виконання декількох завдань;

2) розподіл ресурсів комп’ютера між завданнями;
3) організація взаємодії завдань одного з одним;
4) взаємодія користувальницьких програм з нестандартними зовнішніми пристроями;
5) організація міжмашинної взаємодії й розподілення ресурсів;
6) захист системних ресурсів, даних і програм користувача, процесів, що виконуються, і себе самої від помилкових і злостивих дій користувачів і їхніх програм.

ОС є дуже складною програмною системою, розгляд реалізації виконання всіх названих функцій також досить трудомісткий і тривалий процес, тому ми насамперед зупинимося лише на тих основних питаннях, котрі вирішуються ОС. По-перше – це організація віртуальної пам’яті. Віртуальна пам’ять використову- ється багатьма операційними системами. Вона дозволяє створити враження, що в машині більше пам’яті, чим є насправді. По-друге – робота системи вводу-виводу за допомогою такого абстрактного поняття як файли. Це поняття більш високого рівня, чим команди вводу-виводу, які розглядаються на нижчому архітектурному рівні. По-третє – питання псевдопаралельної обробки (як кілька процесів можуть виконуватися, обмінюватися інформацією й синхронізуватися). Поняття процесу є дуже важливим, і ми докладно розглянемо його. Під процесом можна розуміти працюючу програму й всю інформацію про її стан (про пам’ять, регістри, лічильник команд, стан вводу-виводу й т.п.). Після обговорення цих основних характеристик ми покажемо, як вони реалізуються у операційних системах двох машин: з процесорами Pentium – Windows і UltraSPARC – UNIX.

Почнемо наш розгляд з функцій операційної системи, а, точніше, із забезпечення функції керування пам’яттю.

1 ВІРТУАЛЬНА ПАМ’ЯТЬ

Пам'ять являє собою важливий ресурс, який потребує ретельного керування. У перших комп’ютерах пам’ять була дуже дорога, а тому, обсяг її був дуже малим. Такі комп’ютери як IBM-650 або радянська БЭСМ мали усього 2000 слів пам’яті. Один з перших компіляторів 60-х років минулого століття – ALGOL, був написаний для комп’ютера з розміром пам’яті всього 1024 слова. Давня система з розподілом часу прекрасно працювала на комп’ютері PDP-1, загальний розмір пам’яті якого становив усього 4096 18-бітних слів разом для операційної системи й користувальницьких програм. У ті часи програмісти витрачали дуже багато часу на те, щоб умістити свої програми в малесеньку пам’ять. Часто доводилося використовувати алгоритм, що працює набагато повільніше за іншій алгоритм, оскільки останній, хоча й був кращим, але занадто великим за розміром й програма, у якій використовувався цей кращий алгоритм, могла не поміститися у пам’ять комп’ютера. Незважаючи на те, що в наші дні пам’ять середнього домашнього комп’ютера в тисячі разів перевищує ресурси машин, що існували на початку 60-х років, програми все ж таки збільшуються в розмірі швидше, ніж пам’ять. Фахівці з комп’ютерних наук, перефразовуючи закон Паркінсона жартують: «Програми розширюються, прагнучи заповнити весь об'єм пам'яті, доступний для їхньої підтримки». Зовсім не треба думати, що всі суперсучасні комп’ютери мають пам’ять, вимірювану сотнями або тисячами мебібайт. Комп’ютери, що працюють у системах реального часу, найчастіше мають «величезну» пам’ять розміром від 2-х до 64-х кібібайт. Більш того, навіть на сучасних комп’ютерах з досить великою пам’яттю, операційна система не завжди має змогу надати процесу стільки пам’яті, скільки він потребує, це пов’язано з тим, що на комп’ютері одночасно працює велика кількість процесів і деякому з них ресурсів може не вистачити.

Традиційним вирішенням проблеми малої пам’яті було використання допоміжної пам’яті (наприклад, простору диска). Програміст повинен був поділяти програму на кілька окремих частин, так званих «оверлеїв», кожен з яких поміщав-

ся у пам’ять, але не одночасно, а по черзі. Щоб виконати програму, спочатку необхідно було зчитати у пам’ять й запустити перший оверлей. Коли він завершувався, потрібно було зчитувати й запускати другий оверлей і т.д. Програміст відповідав за розбивку програми на оверлеї й вирішував, у якому місці допоміжної, і, навіть, основної пам’яті повинен був зберігатися кожен оверлей, контролював передачу оверлеїв між основною й допоміжною пам’яттю й взагалі управляв всім цим процесом без якої-небудь допомоги з боку комп’ютера.

Хоча ця технологія широко використовувалася протягом багатьох років, вона потребувала тривалої кропіткої роботи, пов’язаної з керуванням оверлеями, тому працювати з ними могли тільки дуже підготовлені програмісти, добре ознайомлені з архітектурою комп’ютерів. Для прикладних програмістів ця технологія була досить складною. В 1961 році група дослідників з Манчестера (Англія) запропонувала метод автоматичного виконання процесу накладення, при якому програміст міг взагалі не знати про цей процес. Цей метод, що зараз називається віртуальною пам’яттю, мав очевидну перевагу, оскільки звільняв програміста від величезної кількості нудотної роботи. Вперше цей метод було використано у ряді комп’ютерів, випущених в 60-ті роки. До початку 70-х років віртуальна пам’ять з’явилася в більшості комп’ютерів.

З того часу і дотепер пам’ять у комп’ютерах має ієрархічну структуру. Невелика частина її являє собою дуже швидку, дорогу, енергозалежну (тобто таку, що втрачає інформацію, при вимиканні живлення) кеш-пам’ять. Крім того, комп’ютери мають сотні, навіть тисячі, мебібайтів середньошвидкої, також енергозалежної оперативної пам’яті ОЗП (RAM), котра має й середню ціну, і сотні або тисячі гібібайтів повільного, порівняно дешевого, енергонезалежного простору на жорсткому диску. В наш час навіть персональні комп’ютери на процесорах UltraSPARC і Pentium (та й інших, що мають CISC архітектуру), містять дуже складні системи віртуальної пам’яті.

1.1 Сторінкова організація пам’яті

Однім із завдань операційної системи є координація використання всіх складових пам’яті. Як же створюється віртуальна пам’ять? Тут висувається ідея про відокремлення понять адресного простору й адрес пам’яті. Як приклад розглянемо гіпотетичний комп’ютер, команди якого мають 16-бітне поле адреси, а оперативний запам’ятовуючий пристрій містить лише 4096 слів пам’яті. Програма, що працює на такому комп’ютері, могла б звертатися до 65536 слів пам’яті (оскільки адреси 16-бітні, а 2^{16}=65536). Число слів пам’яті, що адресуються, залежить тільки від числа бітів адреси й ніяк не пов’язане із числом реально доступних слів. Адресний простір такого комп’ютера складається із чисел 0, 1, 2, …, 65535. Однак у дійсності комп’ютер має набагато менше слів у пам’яті.

До винаходу віртуальної пам’яті доводилося проводити тверде розмежування між адресами, меншими за 4096, і тими, які дорівнюють або більші за 4096. Ці дві частини можна розглядати як корисний адресний простір і марний адресний простір (адреси вищі за 4095 – марні, оскільки вони не відповідають реальним адресам пам’яті). Іншими словами, ніякого розходження між адресним простором і адресами пам’яті не робиться, оскільки між ними передбачається взаємооднозначна відповідність.

Ідея ж поділу понять адресного простору й адрес пам’яті полягає в наступному. У будь-який момент часу можна одержати прямий доступ до 4096 слів пам’яті, але з цього ні яким чином не виходить, що вони неодмінно повинні відповідати адресам пам’яті від 0 до 4095. Наприклад, ми могли б повідомити комп’ютер, що при звертанні до адреси 4096 повинно використовуватися слово з пам’яті з адресою 0, при звертанні до адреси 4097 – слово з пам’яті з адресою 1, при звертанні до адреси 8191 – слово з пам’яті з адресою 4095 і т.д. Інакше кажучи, ми визначаємо відображення з адресного простору в дійсні адреси пам’яті, таке відображення схематично представлене на рис. 2.

Виникає питання: а що відбудеться, якщо програма зробить перехід до однієї з адрес від 8192 до 12287? У машині без віртуальної пам’яті відбудеться помил-

ка, на екрані з’явиться фраза “Неіснуюча адреса пам’яті” і виконання програми зупиниться. У машині з віртуальною пам’яттю буде мати місце така послідовність кроків:

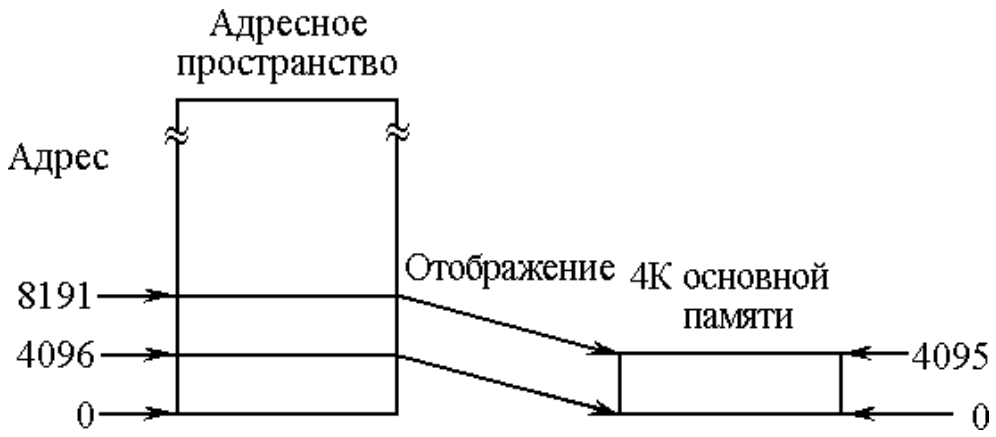

Рисунок 2 – Віртуальні адреси з 4096 до 8191 відображаються у адреси основної пам’яті від 0 до 4095.

- Слова з 4096 до 8191 будуть розміщені на диску.
- Слова з 8192 до 12287 будуть завантажені в основну пам’ять.
- Відображення адрес зміниться; тепер адреси з 8192 до 12287 відповідають коміркам пам’яті з 0 до 4095.
- Виконання програми буде тривати, начебто нічого жахливого не трапилося.

Така технологія автоматичного накладення називається ***сторінковою організацією пам’яті***, а частини програми, які зчитуються з диска, називаються ***сторінками***.

Можливі й інші, більш складні способи відображення адрес із адресного простору в реальні адреси пам’яті. Адреси, до яких програма може звертатися, називають ***віртуальним адресним простором***, а реальні адреси пам’яті в апаратному забезпеченні – ***фізичним адресним простором***. Для того, щоб правильно співвіднести віртуальні адреси з фізичними адресами, будується, так звана, схема

розподілу пам’яті або таблиця сторінок. Передбачається, що на диску досить місця для зберігання якщо не повного віртуального адресного простору то, принаймні, тієї його частини, що використовується в цей момент.

Програми пишуться так, начебто в основній пам’яті вистачить місця для розміщення всього віртуального адресного простору, навіть якщо це не відповідає дійсності. Програми можуть завантажувати слова з віртуального адресного простору або записувати слова у віртуальний адресний простір, незважаючи на те, що, насправді, фізичної пам’яті для цього не вистачає. Програміст може писати програми, навіть не усвідомлюючи, що віртуальна пам’ять існує. Просто створюється таке враження, що обсяг пам’яті певного комп’ютера досить великий.

Слід особливо підкреслити й звернути увагу на той факт, що сторінкова організація пам’яті створює повну ілюзію великої лінійної основної пам’яті такого ж розміру, як адресний простір. У дійсності основна пам’ять може бути менше (або більше), чим віртуальний адресний простір. Те, що пам’ять великого розміру просто моделюється за допомогою сторінкової організації пам’яті, не можна визначити за програмою (тільки за допомогою контролю синхронізації). При звертанні на будь-яку адресу завжди з’являються необхідні дані або потрібна команда. Оскільки програміст може писати програми й при цьому нічого не знати про існування сторінкової організації пам’яті, цей механізм називають прозорим.

Ситуація, коли програміст використовує який-небудь віртуальний механізм і навіть не знає, як він працює, не нова. До архітектури команд, наприклад, часто включається команда MUL (команда одержання добутку), при цьому зовсім необов’язково, щоб в апаратному забезпеченні був спеціальний пристрій для множення. Ілюзія, що машина може перемножувати числа, підтримується мікропрограмою. Точно так само операційна система може створювати ілюзію, що всі віртуальні адреси підтримуються реальною пам’яттю, навіть якщо це неправда. Тільки розроблювачам операційних систем і тим, хто вивчає операційні системи, потрібно знати, як створюється така ілюзія.

Існує й інша організація пам’яті, що є альтернативою до сторінкової. Це так звана ***організація пам’яті із сегментацією***, при якій програміст повинен знати про існування сегментів. Така організація пам’яті буде розглянута після докладного розгляду сторінкової пам’яті.

1.1.1 Реалізація сторінкової організації пам’яті

Парадигма віртуальної пам’яті передбачає необхідність застосування диска для зберігання повної програми й всіх її даних. Природно, що при зміні копії, котра знаходиться у фізичній пам’яті, повинен змінюватись й оригінал сторінки, що розташована на диску.

Віртуальний адресний простір розбивається на низку сторінок рівного розміру, звичайно від 512 байт до 64 КіБ, хоча іноді зустрічається й сторінки розміром в 4 МіБ. Розмір сторінки завжди *повинен бути степенем двійки*. Фізичний адресний простір теж розбивається на частини рівного розміру таким чином, щоб кожна така частина основної пам’яті вміщувала рівно одну сторінку. Ці частини основної пам’яті називаються ***сторінковими кадрами***. На рис. 2 основна пам’ять містить тільки один сторінковий кадр. На практиці звичайно є кілька тисяч сторінкових кадрів.

На рис. 3а, показано один з можливих варіантів поділу перших 64 КіБ віртуального адресного простору на сторінки по 4 КіБ (слід підкреслити, що зараз мова йде про 64 КіБ и 4 КіБ адрес, а не байтів). Адреса може відповідати байтам, але може бути відповідною до слів для комп’ютера, у якому послідовно розташовані слова мають послідовні адреси. Віртуальну пам’ять, зображену на рис. 3, можна реалізувати за допомогою таблиці сторінок, у якій кількість елементів дорівнює кількості сторінок у віртуальному адресному просторі. Тут для простоти показані тільки перші 16 елементів такої таблиці. Коли програма намагається звернутися до слова з перших 64 КіБ віртуальної пам’яті, щоб викликати команду або дані, або щоб зберегти дані, спочатку вона породжує віртуальну адресу від 0 до 65532 (передбачається, що адреси слів повинні ділитися на 4). Для породження цієї ад-

реси можуть використовуватися будь-які стандартні способи адресації, у тому числі індексування й непряма адресація.

Страница	Виртуальный адрес
15	61440 ÷ 65535
14	57344 ÷ 61439
13	53248 ÷ 57343
12	49152 ÷ 53247
11	45056 ÷ 49151
10	40960 ÷ 45055
9	36864 ÷ 40959
8	32768 ÷ 36863
7	28672 ÷ 32767
6	24576 ÷ 28671
5	20480 ÷ 24575
4	16384 ÷ 20479
3	12288 ÷ 16383
2	8192 ÷ 12287
1	4096 ÷ 8191
0	0 ÷ 4095

а

Нижние 32К адресов основной памяти

Странич-ный кадр	Физические адреса
7	28672 ÷ 32767
6	24576 ÷ 28671
5	20480 ÷ 24575
4	16384 ÷ 20479
3	12288 ÷ 16383
2	8192 ÷ 12287
1	4096 ÷ 8191
0	0 ÷ 4095

б

Рисунок 3 – а) перші 64 КіБ віртуального адресного простору поділені на 16 сторінок по 4 КіБ кожна; б) 32 КіБ основної пам’яті поділені на 8 сторінкових кадрів по 4 КіБ кожен

На рис. 3б зображена фізична пам’ять, що складається з восьми сторінкових кадрів по 4 КіБ. Цю пам’ять можна обмежити до 32 КіБ, оскільки:

- це вся пам’ять машини (для процесора, вбудованого в пральну машину або мікрохвильову піч, цього цілком достатньо);
- частина пам’яті, що залишилася не зайнятою іншими програмами.

Оскільки оперативний запам’ятовуючий пристрій сприймає тільки реальні адреси й ні в якому разі не сприймає віртуальні, то необхідно зробити відображення великих віртуальних адрес у обмежену адресну сітку фізичної пам’яті. Як приклад можна розглянути відображення 32-бітної віртуальної адреси у фізичну

15-бітну адресу основної пам’яті. Таке відображення у комп’ютерах з віртуальною пам’яттю здійснює спеціальний пристрій для відображення віртуальних адрес на фізичні. Цей пристрій називається контролером керування пам’яттю (MMU – Memory Management Unit). Він може перебувати як безпосередньо на мікросхемі процесора, так і на окремій мікросхемі десь поруч. У нашім прикладі контролер керування пам’яттю відображає 32-бітну віртуальну адресу в 15-бітну фізичну адресу, тому контролер керування пам’яттю повинен мати 32-бітний вхідний регістр і 15-бітний вихідний регістр адрес.

Для кращого розуміння того, як працює контролер керування пам’яттю, можна розглянути схематичний приклад такого контролера, що зображений на рис. 4. Коли в контролер керування пам’яттю надходить 32-бітна віртуальна адреса, він розділяє цю адресу на 20-бітний номер віртуальної сторінки й, оскільки сторінки в цьому прикладі мають розмір 4 КіБ, на 12-бітний зсув усередині сторінки. Отриманий номер віртуальної сторінки використовується як індекс у таблиці сторінок для знаходження потрібної сторінки. На рис. 4 номер віртуальної сторінки дорівнює трьом, тому з таблиці вибирається елемент з третьої комірки.

Оскільки в нас є 2^{20} віртуальних сторінок і лише 8 (тобто 2^3) сторінкових кадрів, не всі віртуальні сторінки можуть перебувати в пам’яті одночасно, тому спочатку контролер керування пам’яттю перевіряє, чи перебуває потрібна сторінка в теперішній момент у пам’яті. Для цього контролер керування пам’яттю перевіряє біт присутності в даному елементі таблиці сторінок. У нашім прикладі біт присутності дорівнює 1. Це означає, що сторінка в цей момент перебуває в пам’яті.

Далі з обраного елемента таблиці потрібно взяти значення сторінкового кадру (у нашім прикладі – 6) і це значення копіюється у старші три біти 15-бітного вихідного регістра. Потрібно саме тільки три біти, тому що у фізичній пам’яті міститься лише 8 сторінкових кадрів. Паралельно із цією операцією молодші 12 бітів віртуальної адреси (поле зсуву сторінки) копіюються до молодших 12 бітів вихідного регістра. Таким чином отримується 15-бітна фізична адреса, котра відправляється до кеш-пам’яті або до основної пам’яті для пошуку потрібного слова.

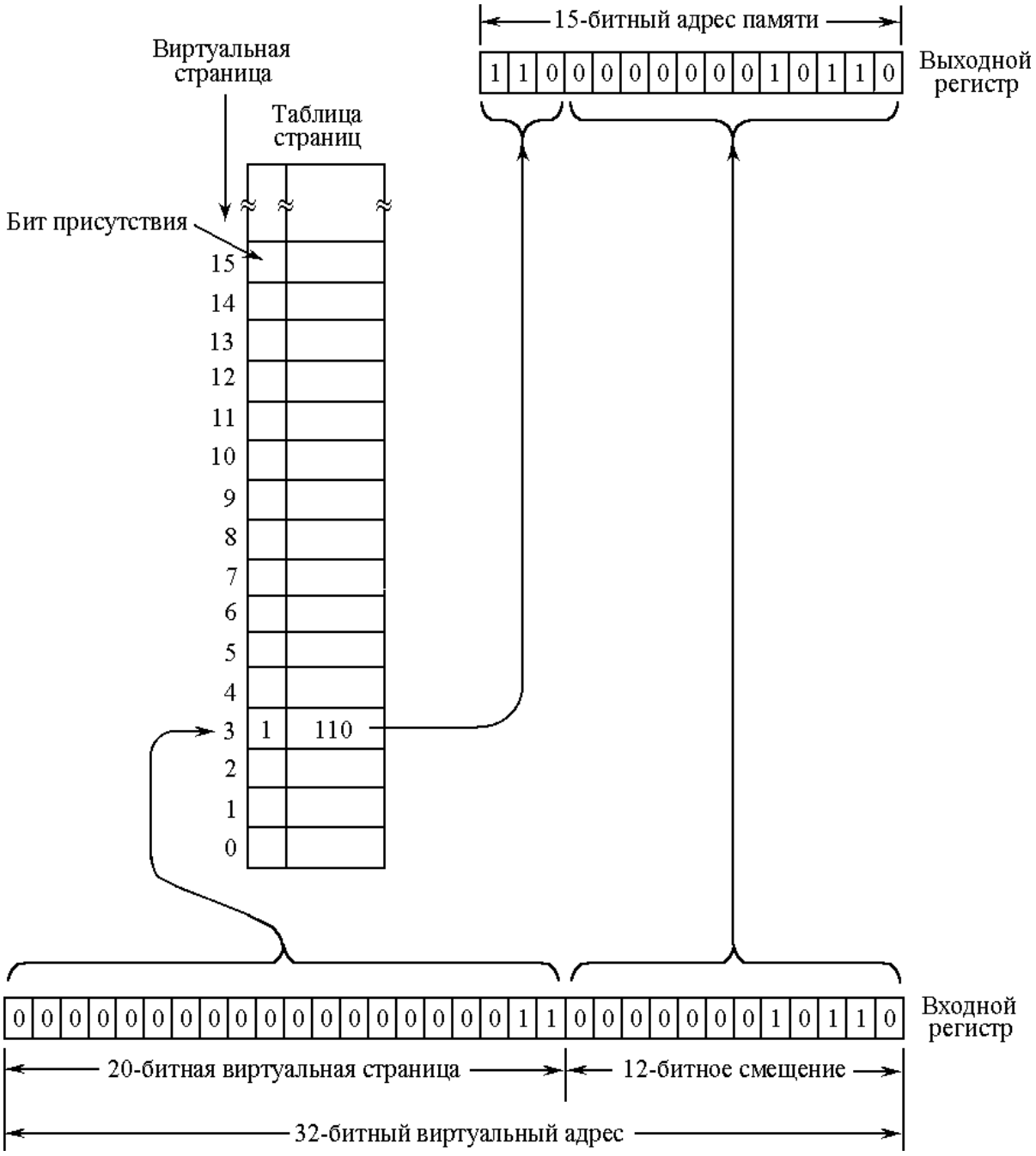

Рисунок 4 – Формування адреси основної пам’яті з адреси віртуальної пам’яті

Рис. 5 демонструє можливе відображення віртуальних сторінок у фізичні сторінкові кадри. Віртуальна сторінка 0 перебуває в сторінковому кадрі 1. Віртуальна сторінка 1 перебуває в сторінковому кадрі 0. Віртуальної сторінки 2 немає в основній пам’яті. Віртуальна сторінка 3 перебуває в сторінковому кадрі 2. Віртуа-

льної сторінки 4 немає в основній пам’яті. Віртуальна сторінка 5 перебуває в сторінковому кадрі 6 і т.д.

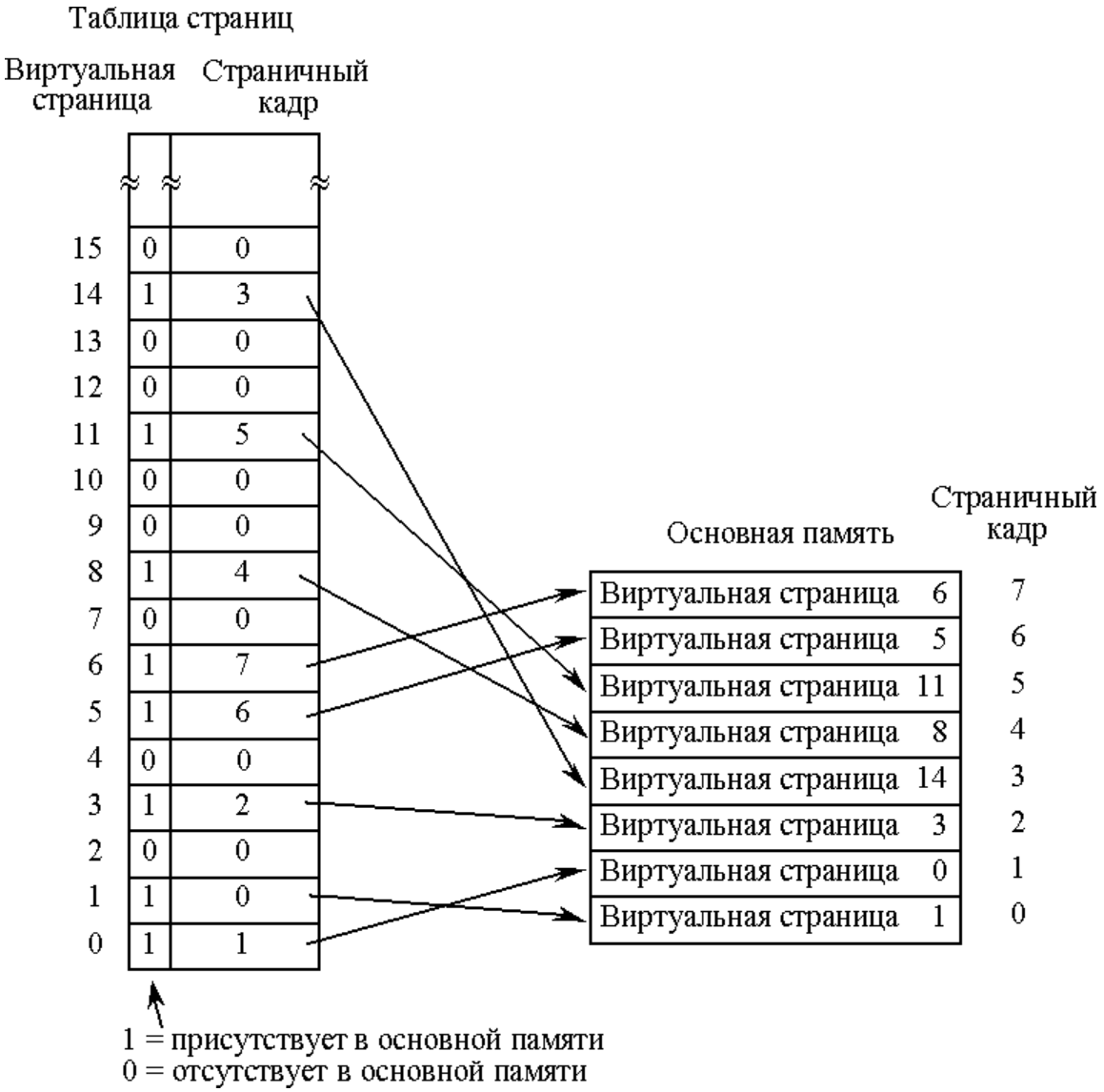

Рисунок 5 – Можливе відображення перших 16 віртуальних сторінок до основної пам’яті, що містить 8 сторінкових кадрів

1.1.2 Виклик сторінок на вимогу і робоча безліч

Раніше передбачалося, що віртуальна сторінка, до якої відбувається звернення, перебуває в основній пам’яті. Однак це припущення не завжди є вірним, оскільки в основній пам’яті недостатньо місця для всіх віртуальних сторінок. При

звертанні до адреси, що відповідає сторінці, якої немає в основній пам'яті, відбувається помилка через відсутність сторінки. У випадку такої помилки операційна система повинна зчитати потрібну сторінку з диска, ввести нову адресу фізичної пам'яті в таблицю сторінок, а потім повторити команду, що викликала помилку.

На машині з віртуальною пам'яттю можна запустити програму навіть у тому випадку, якщо ні однієї частини програми немає в основній пам'яті. Просто таблиця сторінок повинна вказувати, що абсолютно всі віртуальні сторінки перебувають у допоміжній пам'яті. Якщо центральний процесор намагається викликати першу команду, він відразу одержує помилку через відсутність сторінки, за результатом чого сторінка, що містить першу команду, завантажується у запам'ятовуючий пристрій й заноситься до таблиці сторінок. Після цього починається виконання першої команди. Якщо перша команда містить дві адреси й обидві ці адреси перебувають на різних сторінках, причому обидві сторінки не є сторінкою, у якій перебуває команда, то відбудеться ще дві помилки через відсутність сторінки й ще дві наступні сторінки будуть перенесені до основної пам'яті для того, щоб мати можливість завершити команду. Наступна команда може спричинити ще кілька помилок і т.д.

Такий метод роботи з віртуальною пам'яттю називається ***викликом сторінок на вимогу***. При виклику сторінок на вимогу сторінки переносяться в основну пам'ять з віртуальної тільки у разі якщо в тому виникає потреба, але не заздалегідь.

Питання про те, чи варто використовувати виклик сторінок на вимогу чи ні, має сенс тільки на самому початку при запуску програми, поки не буде заповнена вся можлива фізична пам'ять. Коли програма попрацює якийсь час, потрібні сторінки вже будуть зібрані в основній пам'яті. Якщо це комп'ютер з поділом часу й постійно, приблизно кожні 100 мілісекунд, виконується відкачка процесів у віртуальну пам'ять і назад, то кожна програма буде запускатися багато разів. Для кожної програми розподіл пам'яті унікальний й при перемиканні з однієї програми на іншу буде змінюватись, тому в системах з поділом часу підхід керування пам'яттю, заснований на виклику сторінок на вимогу, не годиться.

Альтернативний підхід заснований на тім спостереженні, що більшість команд звертаються до адресного простору не рівномірно. За звичаєм більшість звертань у пам’ять відносяться до невеликої кількості сторінок. При звертанні до пам’яті можна викликати команду, викликати дані або зберегти дані. У кожен момент часу **t** існує набір сторінок, які використовувалися при останніх **k** зверненнях. За визначенням Деннінга такий набір сторінок називають ***робочою безліччю*.**

Оскільки робоча безліч за звичай змінюється дуже повільно, можна, спираючись на набір сторінок, котрі входили в останню перед зупинкою програми робочу безліч, передбачити, які сторінки знадобляться при новому запуску програми. Ці сторінки, сподіваючись, що передбачення є досить точним, можна завантажити заздалегідь ще перед черговим запуском програми.

1.1.3 Політика заміщення сторінок

В ідеалі набір сторінок, які постійно використовуються програмою (так звана робоча безліч), можна зберігати в пам’яті, щоб скоротити кількість помилок через відсутність сторінок. Однак програмісти не мають змоги заздалегідь знати, які сторінки перебувають у робочій безлічі, тому операційна система періодично повинна переглядати цю безліч. Якщо програма звертається до сторінки, яка відсутня у основній пам’яті, то таку сторінку потрібно викликати з диска. Але щоб звільнити для неї місце у пам’яті, на диск потрібно відправити яку-небудь іншу сторінку. Отже, потрібним є алгоритм для визначення, яку саме сторінку необхідно забрати з пам’яті.

Вибирати таку сторінку просто навмання не можна. Якщо, наприклад, вибрати сторінку, котра містить команду, виконання якої викликало помилку, то при спробі викликати наступну команду відбудеться ще одна помилка через відсутність сторінки. Більшість операційних систем намагаються завбачувати, які зі сторінок у пам’яті найменш корисні у момент заміни в тому розумінні, що їхня відсутність не вплине сильно на хід виконання програми. Іншими словами, за-

мість того щоб видаляти сторінку, яка незабаром знадобиться, операційна система намагається вибрати таку сторінку, що не буде потрібна протягом довгого часу.

Є математично доведеним, що немає ідеального алгоритму для вирішення такої задачі. Тому існує безліч алгоритмів, що здійснюють більш-менш вдалий вибір непотрібних сторінок. Докладний розгляд всіх з них потребує доволі багато часу й простору у конспекті, тому коротко зупинимося тільки на двох з них.

За одним з алгоритмів видаляється та сторінка, котра використовувалася найбільш давно, оскільки ймовірність того, що вона буде в поточній робочій безлічі, дуже мала. Цей ***алгоритм називається LRU*** (Least Recently Used – алгоритм видалення елементів, що були використані найбільш давно). Хоча цей алгоритм працює досить добре, іноді виникають патологічні ситуації. Нижче описана одна з них.

Якщо уявити собі програму, яка виконує величезний цикл, розміром на дев'ять віртуальних сторінок, а у фізичній пам'яті місце є тільки для восьми сторінок, то робота менеджера пам'яті за алгоритмом LRU буде виконуватись у такому порядку. Коли за результатами виконання програма перейде до передостанньої сторінки уявного циклу – 7, в основній пам'яті будуть перебувати сторінки з 0 по 7 (рис. 6 а). Потім відбувається спроба викликати команду з віртуальної сторінки 8, це повинно спричинити помилку через відсутність сторінки у пам'яті. Потрібно прийняти рішення, яку сторінку забрати. За алгоритмом LRU буде обрана віртуальна сторінка 0, оскільки вона використовувалася раніше за всіх. Віртуальна сторінка 0 видаляється, а потрібна віртуальна сторінка завантажується на її місце (рис. 6 б).

Після виконання команд із віртуальної сторінки 8 програма повинна вернутися до початку циклу, тобто до віртуальної сторінки 0. Цей крок спичиняє ще одну помилку через відсутність сторінки. Тільки що викинуту віртуальну сторінку 0 доводиться викликати назад. За алгоритмом LRU на цей раз видаляється сторінка 1 (рис. 6 в). Через якийсь час програма намагається викликати команду з віртуальної сторінки 1, що знову спричиняє помилку. При цьому викликається сторінка 1 і видаляється сторінка 2 і т.д.

Віртуальна сторінка	7	Віртуальна сторінка	7	Віртуальна сторінка	7
Віртуальна сторінка	6	Віртуальна сторінка	6	Віртуальна сторінка	6
Віртуальна сторінка	5	Віртуальна сторінка	5	Віртуальна сторінка	5
Віртуальна сторінка	4	Віртуальна сторінка	4	Віртуальна сторінка	4
Віртуальна сторінка	3	Віртуальна сторінка	3	Віртуальна сторінка	3
Віртуальна сторінка	2	Віртуальна сторінка	2	Віртуальна сторінка	2
Віртуальна сторінка	1	Віртуальна сторінка	1	Віртуальна сторінка	0
Віртуальна сторінка	0	Віртуальна сторінка	8	Віртуальна сторінка	8
а)		б)		в)	

Рисунок 6 – Ситуація, у якій алгоритм LRU не діє

Очевидно, що в цій ситуації алгоритм LRU зовсім не працює (заради справедливості необхідно сказати, що й інші алгоритми при подібних обставинах теж не працюють). Однак, якщо розширити розмір робочої безлічі, число помилок через відсутність сторінок стане мінімальним.

Існує ще один досить нескладній алгоритм, який з тим же успіхом можна застосувати у цій ситуації – ***алгоритм FIFO*** (First-in First-out – першим прийшов – першим пішов). FIFO видаляє ту сторінку, що раніше всіх завантажувалася, незалежно від того, коли востаннє вироблялося звертання до цієї сторінки.

Програмну та й апаратну реалізацію обох алгоритмів можна зробити, зв’язавши з кожним сторінковим кадром окремий лічильник, який відповідає за

підрахування терміну життя сторінки у фізичній пам’яті. Спочатку всі такі лічильники встановлюються у 0.

Тоді за алгоритмом FIFO після кожної помилки через відсутність сторінок лічильники всіх сторінок, що наразі перебувають в пам’яті, збільшуються на 1, а лічильник щойно викликаної сторінки набуває значення 0. За алгоритмом LRU навпаки, у 0 скидається лічильник сторінки, до якої виконувалось звернення, незалежно від того довантажувалася вона до пам’яті чи ні.

Коли потрібно вибрати сторінку для видалення за тим чи іншим алгоритмом, то вибирається сторінка із найбільшим значенням лічильника життя. Оскільки вона завантажувалася першою, або до неї дуже давно не було звернення. В цьому випадку існує велика ймовірність того, що сторінка найближчим часом не знадобиться.

Якщо розмір робочої безлічі більший за число доступних сторінкових кадрів, жоден алгоритм не дає хороших результатів, і помилки через відсутність сторінок будуть відбуватися часто. Якщо програма постійно спричиняє подібні помилки, то говорять, що спостерігається ***пробуксовка програми*** (thrashing). Не потрібно пояснювати, що пробуксовка дуже небажана. Й навпаки, якщо програма використовує великий віртуальний адресний простір, але має невелику робочу безліч, яка змінюється досить повільно і таку, що цілком вміщується до основній пам’яті, нічого страшного, звісно, відбуватися не буде. Це твердження має силу, навіть якщо програма використовує в сто разів більше слів віртуальної пам’яті, чим їх утримується у фізичній основній пам’яті.

Якщо сторінка, яку потрібно видалити, не змінилася з тих пір, як її зчитали з диска до операційного запам’ятовуючого пристрою (а це цілком ймовірно, якщо сторінка містить програму, а не дані), то необов’язково записувати її назад на диск, оскільки точна копія там уже існує. Однак, якщо ця сторінка змінилася, то копія на диску вже їй не відповідає, і її потрібно оновити.

Якщо навчитися визначати, чи змінювалася сторінка, чи не змінювалася, то можна цілком уникнути непотрібних переписувань на диск і тим самим заощадити багато часу. На багатьох машинах у контролері керування пам’яттю для кожної

сторінки міститься один додатковий біт, що дорівнює 0 при завантаженні сторінки й набирає значення 1, коли мікропрограма або апаратне забезпечення змінюють цю сторінку. За допомогою цього біта операційна система визначає, чи змінювалася ця сторінка, чи ні, й потрібно її перезаписувати на диск або ні.

1.1.4 Обробка сторінкового переривання

Підсумовуючи все вище сказане, можна більш детально описати, що ж відбувається при сторінковому перериванні в разі коли адреса виходить за межі поточної сторінки. Послідовність дій така:

1. Апаратне забезпечення перемикає систему в режим ядра ОС, зберігаючи лічильник команд у стеку. На більшості машин у спеціальних регістрах процесора зберігається деяка інформація про стан поточної інструкції.
2. Запускається написана на асемблері програма, що зберігає основні регістри й іншу інформацію, котра може змінитися, захищаючи її від руйнування операційною системою. Ця програма викликає операційну систему як процедуру.
3. Операційна система виявляє, що відбулося сторінкове переривання, і намагається знайти необхідну віртуальну сторінку. Часто необхідна інформація міститься в одному з апаратних регістрів. Якщо ні, операційна система дістає зі стека лічильник команд, вибирає інструкцію й програмно аналізує її з метою визначити, що вона робила в той момент, коли сталася помилка звернення до сторінки.
4. Як тільки стає відома віртуальна адреса, що спричинила переривання, система перевіряє, чи не є ця адреса помилковою і чи узгоджується рівень захисту з доступом. Якщо ні, то процесу посилається сигнал або процес знищується. Якщо адреса дійсна й не відбулося помилки захисту, система перевіряє наявність вільних сторінкових кадрів. Якщо вільних кадрів немає, запускається алгоритм заміщення сторінок, що вибирає ту котра буде усунена з ОЗП.
5. Якщо обраний сторінковий кадр «брудний», тобто змінювався з часу за-

вантажування до пам’яті, він заноситься в графік запису на диск і відбувається перемикання контексту, що припиняє процес, який спричинив переривання, і це дозволяє працювати іншому процесу доти, поки не буде виконано перенесення сторінки на диск. У кожному разі кадр відзначається як зайнятий, щоб запобігти його використанню в інших цілях.

6. Як тільки сторінковий кадр записується на диск («очищається»), операційна система шукає адресу на диску, де розташована необхідна сторінка, і планує дискову операцію для її переносу у пам’ять. Під час завантаження сторінки процес, що спричинив переривання, усе ще знаходиться у припиненому стані і виконується інший користувальницький процес, якщо такий доступний.
7. Коли дискове переривання відзначає, що сторінка розміщена у пам’яті, оновлюється таблиця сторінок, відображаючи її позицію, а кадр позначається, як такий, що перебуває в нормальному стані.
8. Перервана команда вертається до того стану, з якого вона починалася, і значення лічильника команд припиненого процесу (у стеці або в системній комірці пам’яті) коректується так, щоб указувати на цю команду.
9. Перерваний процес вноситься в графік, і операційна система повертає керування у асемблерну процедуру, яка звернулась до неї.
10. Ця процедура перезавантажує регістри й іншу інформацію про стан процесу і повертає керування в користувальницький простір для продовження виконання користувальницької програми так, якби ніякого переривання не відбулося.

1.1.5 Розмір сторінок і фрагментація

Якщо користувальницька програма й дані час від часу заповнюють рівно цілу кількість сторінок, то коли вони перебувають у пам’яті, вільного місця там не залишається. З іншого боку, якщо вони не заповнюють рівно цілу кількість сторінок, тобто спільний розмір програми і даних не є кратним до розміру сторінки, то на останній сторінці завжди залишиться невикористаний простір. Наприклад, як-

що програма й дані займають 26000 байтів на машині з 4096 байтами на сторінку, то перші 6 сторінок будуть заповнені повністю, що в сумі дасть 6×4096=24576 байтів, а остання сторінка буде містити 26000-24576=1424 байта. Оскільки в кожній сторінці є простір для 4096 байтів, 2672 байта залишаться вільними. Щораз, коли сьома сторінка присутня в пам'яті, ці надлишкові байти будуть займати місце в основній пам'яті, але при цьому не будуть виконувати ніякої функції. Ця проблема називається ***внутрішньою фрагментацією*** (оскільки невикористаний простір є внутрішнім відносно якоїсь сторінки).

Якщо розмір сторінки дорівнює **n** байтів, то середній невикористаний простір в останній сторінці програми буде **n**/2 байтів. Цілком очевидно, що потрібно використовувати сторінки як найменшого розміру, щоб звести до мінімуму кількість невикористаного простору. З іншого боку, якщо застосовуються сторінки маленького розміру, то буде потрібно багато сторінок і набагато більша таблиця сторінок. Якщо таблиця сторінок зберігається в апаратному забезпеченні, то для зберігання такої великої таблиці сторінок потрібно багато регістрів, що підвищує вартість комп'ютера. Крім того, при запуску й зупинці програми на завантаження й збереження цих регістрів буде втрачатися більше часу.

Не складно математично проаналізувати витрати пам'яті, пов'язані з застосуванням маленьких віртуальних сторінок. Якщо припустити, що середній розмір процесу дорівнює **s** байт, а сторінки, як вже говорилось, – **n** байт. Крім того, вважати, що запис у таблиці сторінок потребує **b** байтів для кожної сторінки. Тоді приблизна кількість сторінок, необхідна для процесу, дорівнює **s/n** і потребує **sb/n** байтів для таблиці сторінок. Таким чином, загальні накладні витрати внаслідок підтримки таблиці сторінок і втрати від внутрішньої фрагментації дорівнюють сумі цих двох складових:

$$\mathbf{outlay = sb/n + n/2.}$$

Перший доданок (розмір таблиці сторінок) збільшується при зменшенні розміру сторінки. Другий доданок (внутрішня фрагментація) зростає при збільшенні розміру сторінки. Оптимальний варіант повинен перебувати десь посередині. Якщо взяти першу похідну по змінній **n** і дорівняти її нулю, ми одержимо рівність:

$$-sb/n^2 + 1/2 = 0.$$

Ця рівність дає оптимальний розмір сторінок (беручи до уваги тільки втрати пам’яті на фрагментацію й розмір таблиці сторінок).

$$n = (2sb)^{1/2}$$

Для середнього розміру процесу **s = 1** МіБ і розміру запису в таблиці сторінці – 8 байтів оптимальний розмір сторінки буде дорівнювати 4 КіБ. У комп'ютерах, що випускаються серійно, розмір сторінок може змінюватись у діапазоні від 512 байтів до 64 КіБ. Раніше звичайно вживалася величина 1 КіБ, але тепер частіше зустрічаються сторінки розміром 4 або 8 КіБ. Тому що пам’яті стає більше, то й розмір сторінок також має тенденцію до зростання (але залежність не лінійна, збільшення вчетверо розміру оперативної пам’яті рідко подвоює розмір сторінки).

Ще одним недоліком маленьких сторінок є те, що маленькі сторінки знижують ефективність пропускної здатності диска. Оскільки перед початком передачі даних з диска доводиться чекати приблизно 10 мс (час, що затрачується на пошук + час обертання диска), вигідніше робити великі передачі. Так простий розрахунок показує, що при швидкості передачі даних 10 МіБ за секунду, пошук та передача 8 КіБ даних однім блоком потребує 10 + 0.8 = 10.8 мс. Тобто у порівнянні з передачею 1 КіБ, час необхідний для передачі збільшується лише на 0,7 мс, але натомість час пошуку збільшується у вісім разів, і загальний час пошуку і передачі 8 КіБ даних блоками по 1 КіБ становить 80.8 мс.

Однак у маленьких сторінок є й свої переваги. Якщо робоча безліч складається з великої кількості маленьких відділених одна від одної областей віртуального адресного простору, то при маленькому розмірі сторінки буде рідше виникати пробуксовка (режим інтенсивного підкачування), чим при великому. Прикладом такої робочої безлічі може слугувати декотра матриця **A** розміром 10000×10000 елементів, що зберігаються в послідовних словах, складених з 8 байтів (**A[1,1], A[2,1], A[3,1]** і т.д.). При такому записі елементи першого рядка матриці (**A[1,1], A[1,2], A[1,3]** і т.д.) будуть починатися на відстані 80000 байтів один від одного. Програма, що виконує обчислення над елементами цього рядка, буде використовувати 10000 областей, кожна з яких відділена від сусідніх наступними 79992 байтами. Якби розмір сторінки становив 8 КіБ, то для зберігання всіх сторінок у оперативному запам'ятовуючому пристрої знадобилося б 80 МіБ.

Навпаки, при розмірі сторінки в 1 КіБ для зберігання всіх сторінок буде потрібно вже не 80, а всього 10 МіБ ОЗП. Цілком зрозуміло, що при розмірі пам'яті в 32 МіБ у першому випадку (при розмірі сторінки в 8 КіБ) програма ввійде в режим інтенсивного підкачування, а при розмірі сторінки в 1 КіБ цього не трапиться.

1.2 Сегментація

Дотепер мова йшла про одновимірну віртуальну пам'ять, у якій віртуальні адреси йдуть одна за одною від 0 до якоїсь максимальної адреси. З багатьох причин набагато зручніше використовувати мінімум два й навіть більше окремих віртуальних адресних просторів. Наприклад, компілятор може мати декілька таблиць, які створюються в процесі компіляції:

- таблицю символів, що містить імена й атрибути змінних;
- вихідний текст, збереження якого потрібно для роздруківки;
- таблицю, що містить всі цілочисельні константи й константи із плаваючою точкою, які використовуються у програмі;

– дерево розбору висловлювань, котре містить синтаксичний аналіз програми;

– стек, що використовується для виклику процедур у компіляторі.

В одновимірній моделі пам'яті ці п'ять таблиць довелося б розмістити у віртуальному адресному просторі у вигляді суміжних областей, як показано на рис. 7. В такому одновимірному адресному просторі одна з таблиць може «врізатися» в іншу. У прикладі – кожна з перших чотирьох таблиць постійно зростає в процесі компіляції. Остання таблиця зростає й зменшується зовсім непередбачено.

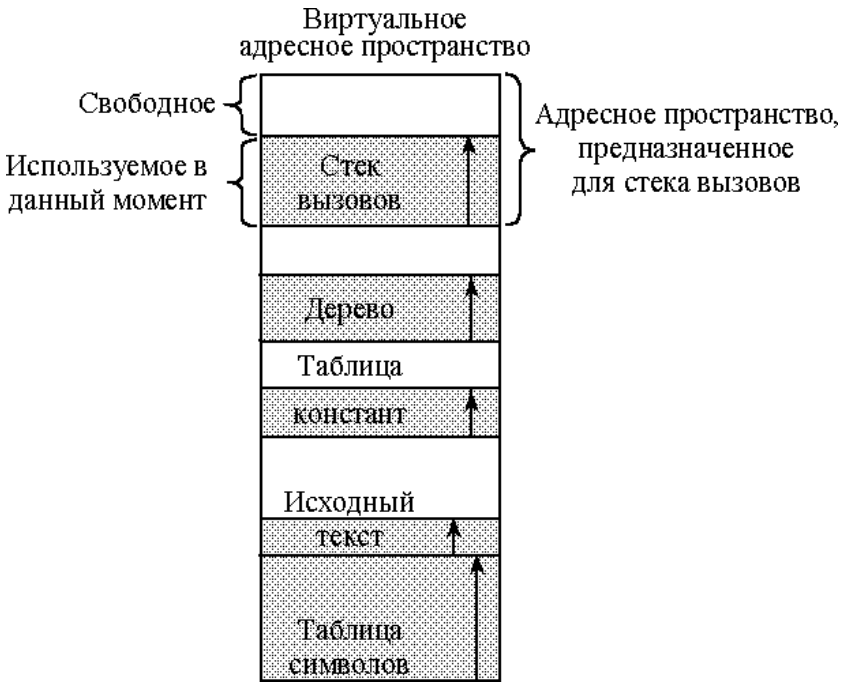

Рисунок 7 – Схематичне представлення одновимірного адресного простору, який містить таблиці, що постійно змінюються

Якщо програма містить дуже велике число змінних, то цілком ймовірно, що може відбутися така ситуація. Область адресного простору, призначена для таблиці символів, може переповнитися, навіть якщо в інших таблицях повно вільного місця. Компілятор, звичайно, може повідомити, що він не здатен продовжувати роботу через велику кількість змінних, але можна без цього й обійтися, оскільки в інших таблицях багато вільного місця.

Компілятор може забирати вільний простір у одних таблиць і передавати його іншим таблицям, але це схоже на керування оверлеями вручну – деяка незручність у найкращому разі, а в гіршому – довга нудна робота.

Насправді потрібно просто звільнити програміста від розширення й скорочення таблиць, подібно тому, як віртуальна пам’ять виключає необхідність стежити за розбивкою програми на оверлеї.

Для цього достатньо створити багато абсолютно ***незалежних адресних просторів***, які називаються ***сегментами***. На рис. 8 зображена така сегментована пам’ять.

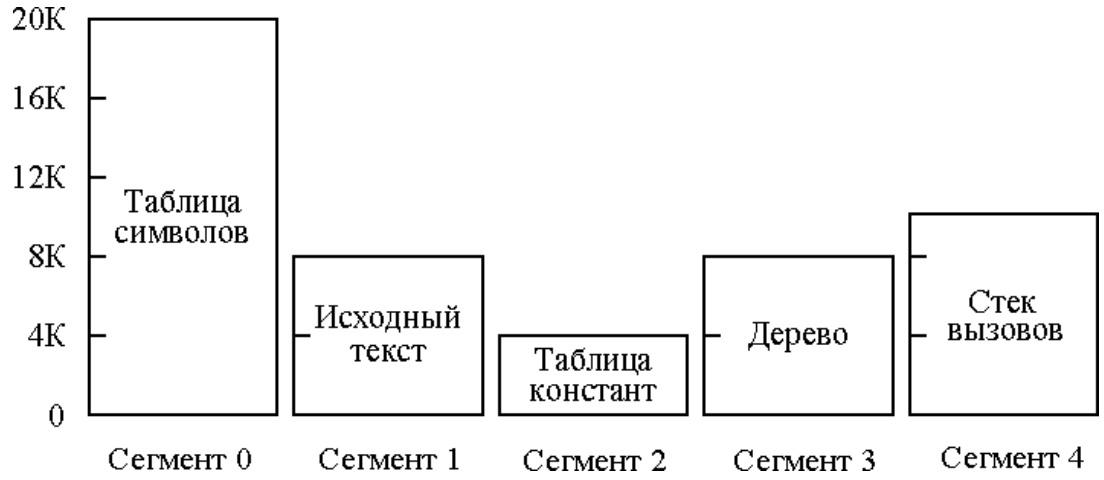

Рисунок 8 – Сегментна модель пам’яті для приклада з компілятором

Сегментована пам’ять позволяє збільшувати й зменшувати кожен сегмент (у даному разі – кожну таблицю) незалежно від інших сегментів (таблиць). Кожен сегмент складається з лінійної послідовності адрес від 0 до якого-небудь максимуму. Таким чином довжина кожного сегмента може бути у межах від 0 до деякого припустимого максимального значення. Різні сегменти, в свою чергу, можуть мати різну довжину (зазвичай так і буває). Більше того, довжина сегмента може змінюватися під час виконання програми. Довжина, наприклад, стекового сегмента може збільшуватися щоразу, коли що-небудь поміщається на стек, і зменшуватися, коли що-небудь виштовхується зі стека.

Через те що кожен сегмент засновує окремий адресний простір, різні сегменти можуть збільшуватися або зменшуватися незалежно один від одного й не впливаючи один на одного. Якщо стеку в певному сегменті знадобиться більше адресного простору (щоб стек міг збільшитися), він може одержати його, оскільки в його адресному просторі більше нема в що «урізатися». Природно, сегмент може переповнитися, але це відбувається рідко, оскільки сегменти дуже великі. Щоб визначити адресу у двовимірній пам'яті, програма повинна назначати номер сегмента й вказувати адресу всередині сегмента.

Обов'язково слід зазначити, що ***сегмент є логічним елементом, про який програміст знає і який він використовує***. Сегмент може містити процедуру, масив, стек або ряд скалярних змінних, але ***як правило, до сегмента входить тільки один тип даних.***

Сегментована пам'ять крім спрощення роботи зі структурами даних, які постійно зменшуються або збільшуються, має й інші переваги. Якщо кожна процедура займає окремий сегмент, у якому перша адреса – це адреса 0, то зв'язування процедур, які компілюються окремо, сильно спрощується. Коли всі процедури програми скомпільовані й зв'язані, при виклику процедури із сегмента **`n`** для звертання до слова 0 буде використовуватися адреса (**`n, 0`**).

Якщо процедура в сегменті згодом буде змінена й перекомпільована, то інші процедури змінювати не потрібно (оскільки жодна початкова адреса не буде змінена), навіть якщо нова версія більша за розміром у порівнянні зі старою. В одновимірній пам'яті процедури звичайно розташовуються одна за одною, і між ними немає ніякого вільного адресного простору. Отже, зміна розміру однієї процедури може вплинути на початкову адресу інших процедур. Це, у свою чергу, потребує зміни всіх процедур, які звертаються до кожної із цих процедур, щоб попадати в нові початкові адреси. Якщо в програму включено багато процедур, цей процес буде занадто втратним.

Сегментація полегшує поділ загальних процедур або даних між декількома програмами. Якщо комп'ютер містить кілька програм, що працюють паралельно

(це може бути реальна або змодельована паралельна робота), і якщо всі ці програми використовують деяку кількість одних й тих же процедур, постачати кожну програму окремою копією всіх цих процедур є не досить розумно та й марнотратно для пам’яті. А якщо зробити кожну процедуру окремим сегментом, їх легко можна буде розділяти між декількома програмами, що цілком виключить необхідність створювати фізичні копії кожної поділюваної процедури. У результаті у системі буде заощаджена певна кількість пам’яті.

Різні сегменти можуть мати різні види захисту. Наприклад, сегмент із процедурою можна визначити як «тільки для виконання», заборонивши тим самим зчитування з нього й запис до нього. Для масиву, який містить лише дані із плаваючою точкою дозволяється тільки читання й запис, але не виконання й т.д. Такий захист часто допомагає виявити помилки в програмі.

Необхідно розуміти, чому захист має сенс тільки для сегментованої пам’яті й не має ніякого сенсу для одновимірній (лінійної) пам’яті. Зазвичаєм в сегменті одночасно не можуть утримуватися й процедура й стек (тільки що-небудь одне). Тому оскільки кожен сегмент містить у собі об’єкт тільки одного типу, він може використовувати захист, що підходить для цього типу. Але для лінійної віртуальної пам’яті, де немає різниці між програмою і даними, створити подібну систему захисту сторінок досить складно, та й найчастіше непотрібно і навіть неможливо.

У табл. 1 наведено порівняння властивостей сторінкової організації пам’яті і пам’яті, організованої за допомогою сегментації.

Вміст сторінки в деякому розумінні є випадковим. Програміст може нічого не знати про сторінкову організацію пам’яті. В принципі можна помістити декілька бітів у кожен елемент таблиці сторінок для доступу до нього, але щоб використовувати цю особливість, програмістові доведеться стежити за границями сторінки в адресному просторі. Справа в тому, що сторінкова розбивка пам’яті і була придумана якраз для усунення подібних труднощів.

Таблиця 1 – Порівняння сторінкової організації пам’яті й сегментації

Властивості	Сторінкова організація пам’яті	Сегментація
Чи повинен програміст мати якусь уяву про організацію пам’яті?	Ні	Так
Скільки лінійних адресних просторів є?	Один	Багато
Чи може віртуальний адресний простір збільшувати розмір пам’яті?	Так	Так
Чи легко управляти таблицями з розмірами, що змінюються?	Ні	Так
Мета створення технології?	Моделювання пам’яті великого розміру	Забезпечення кількох адресних просторів

З іншого боку, при використанні сегментної організації в користувача створюється повна ілюзія, що всі сегменти постійно перебувають в основній пам’яті і до них можна звернутися у будь-який момент часу. Тому програмістові, на відміну від оверлеїв, немає необхідності стежити за наявністю і розташуванням сегментів у пам’яті.

1.2.1 Способи й алгоритми реалізації сегментації

Існує два способи якими можна реалізувати сегментацію пам’яті. По-перше, це підкачування і по-друге, розбивка на сторінки. При першому підході у кожен окремий відрізок часу в пам’яті перебуває деякий обмежений набір сегментів. Якщо відбувається звертання до сегмента, якого немає в цей момент у пам’яті, то викликаний сегмент повністю переноситься з довгострокової пам’яті в оперативну. Якщо для нього не вистачає місця в пам’яті, тоді спочатку потрібно один або

кілька сегментів записати на диск. Це робиться у разі, якщо там вже не перебуває їх відповідна копія. Але у випадку її наявності копія, що вже розташована у оперативній пам'яті, просто видаляється з неї. У деякому розумінні, підкачування сегментів дуже схоже на виклик сторінок на вимогу: сегменти завантажуються й видаляються лише в тому разі, коли виникає така потреба.

Однак сегментація суттєво відрізняється від розбивки на сторінки саме в тім, що розмір сторінок фіксований, а розмір сегментів – ні. Рис. 9 пояснює застосування сегментної пам'яті.

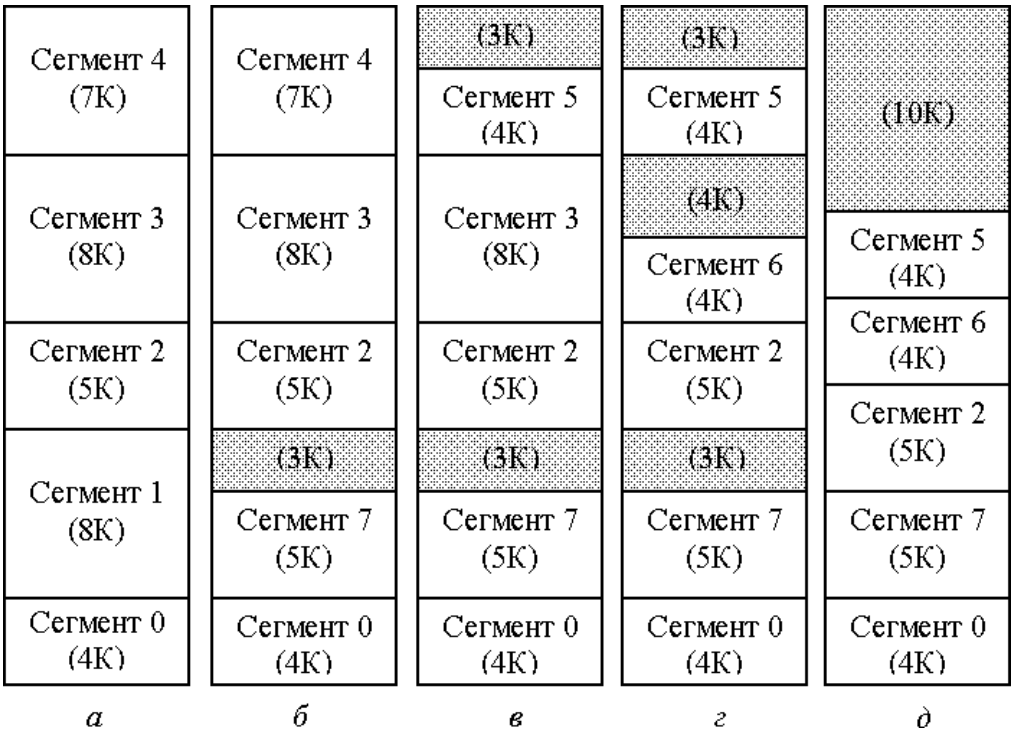

Рисунок 9 – Динаміка зовнішньої фрагментації (а, б, в, г);
видалення зовнішньої сегментації шляхом ущільнення (д)

На рис. 9а показано приклад фізичної пам'яті, у якій спочатку розташовані 5 сегментів. Подивимося, що відбувається, якщо сегмент 1 видаляється, а сегмент 7, що менше його за розміром, поміщається на його місце. У результаті вийде конфігурація, що зображена на рис. 9б. Між сегментом 7 і сегментом 2 з'являється невикористана область («дірка»). Нехай потім сегмент 4 замінюється сегментом 5

(рис. 9в), а сегмент 3 заміщається сегментом 6 (рис. 9г). Через якийсь час оперативна пам'ять розділиться на ряд областей, одні з яких будуть містити сегменти, а інші – невикористані області. Це називається ***зовнішньою фрагментацією*** (оскільки невикористаний простір попадає не в сегменти, а в «дірки» між ними, тобто процес відбувається поза сегментами). Іноді зовнішню фрагментацію називають ***поклітинною розбивкою***.

Подивимося, що відбудеться, якщо програма знов звернеться до сегмента 3 у той момент, коли у пам'яті вже утворилася значна зовнішня фрагментація (рис. 9г). Для завантаження сегмента 3 потрібно 8 КіБ пам'яті, а загальний простір «дірок» становить 10 КіБ, і це більше, ніж потрібно для сегмента 3. Але тому що цей простір розбитий на маленькі шматочки, сегмент 3 завантажити не можна. Замість цього доводиться спочатку видаляти інший сегмент.

Щоб уникнути такої ситуації, можна зробити наступне. Щораз, коли з'являється «дірка», слід переміщати сегменти, що виникають за «діркою», ближче до адреси 0, усуваючи в такий спосіб цю «дірку» і залишаючи більшу «дірку» наприкінці. Але робити ущільнення після завантаження кожного сегмента і після появи кожної дірки є не досить раціональним з погляду часу виконання програми. Тому частіше роблять в інший спосіб. За цим способом чекають, поки зовнішня фрагментація не сягне деякого критичного стану (коли на частку «дірок» припадає більше певного відсотка від усього обсягу пам'яті), і тільки після цього робиться ущільнення. На рис. 9д показано, як буде виглядати пам'ять після ущільнення. Мета ущільнення пам'яті – зібрати всі маленькі «дірки» до однієї великої «дірки», у яку можна помістити один або кілька нових сегментів. Недолік ущільнення полягає в тому, що на цей процес витрачається деяка кількість часу.

Якщо на ущільнення пам'яті потрібно занадто багато часу, потрібен спеціальний алгоритм для визначення, яку саме «дірку» краще використовувати для сегмента, що завантажується. Для цього в ОС створюється список адрес і розмірів всіх «дірок». Пошук потрібної «дірки» у такому списку може вестися двома способами.

За популярним алгоритмом ***оптимального припасування*** або *best fit* вибирають найменьшу «дірку», у яку може помістится потрібний сегмент. Мета цього алгоритму – співвіднести «дірки» і сегменти, щоб уникнути «обламування» шматка великої «дірки», котра може знадобитися пізніше для великого сегмента.

Інший популярний алгоритм передбачає перегляд списку «дірок» по колу і вибирає ***першу придатну*** за розміром для даного сегмента «дірку», це так званий алгоритм *first fit*. Природно, робота за цим алгоритмом займає менше часу, чим перевірка всього списку, задля знаходження оптимальної «дірки». Тому не дивно, що останній алгоритм набагато кращий, ніж алгоритм оптимального припасування, з погляду загальної продуктивності, оскільки оптимальне припасування породжує дуже багато маленьких невикористаних дірок. Дійсно, якщо ми знайшли блок з розміром більшим від заданого, ми повинні відокремити «хвіст» і позначити його як новий вільний блок. Зрозуміло, що у випадку *best fit* середній розмір цього хвоста буде маленьким, і ми в підсумку одержимо велику кількість дрібних блоків, які неможливо об’єднати, тому що простір між ними зайнятий.

Загалом кажучи, обидва алгоритми скорочують середній розмір «дірки». Щораз, коли сегмент поміщається в «дірку», яка більша, ніж цей сегмент, а це буває практично завжди (точні влучення дуже поодинокі), «дірка» поділяється на дві частини. Одну частину займає сегмент, а друга частина – це нова «дірка». Нова «дірка» завжди менша, ніж стара.

При використанні алгоритму *first fit* з лінійним двоспрямованим списком виникає інша специфічна проблема. Якщо щораз переглядати список з того самого місця, то більші блоки, розташовані ближче до початку списку, будуть частіше вилучатися. Відповідно, дрібні блоки будуть мати тенденцію до накопичування на початку списку, що збільшує середній час пошуку. Простий спосіб боротьби із цим явищем полягає в тому, щоб переглядати список то в одному напрямку, то в іншому. Більш радикальний і ще більш простий метод полягає в тому, що список робиться кільцевим, і кожен раз пошук починається з того місця, де ми зупинилися минулого разу. У це ж місце додаються блоки, що звільнилися. У результаті список дуже ефективно перемішується й ніякого «антисортування» не виникає.

Але у ситуаціях, коли потребується розташування блоків декількох фіксованих розмірів, алгоритм *best fit* виявляється кращим.

Без відтворення великих «дірок» з маленьких, тобто без ущільнення, обидва алгоритми у кінцевому підсумку призводять до наповнювання пам'яті маленькими невикористаними «дірками». Крім ущільнення існують і інші процеси об'єднання «дірок». Наприклад, щораз, коли сегмент видаляється з пам'яті, а одна або обидві сусідні області цього сегмента являють собою «дірки», а не сегменти, то суміжні невикористані простори можна злити в одну велику «дірку». Так, якщо у ситуації, що зображена на рис. 9г, видалити сегмент 5, то дві сусідні «дірки» і 4 КіБ, які використовувалися цим сегментом, будуть злиті до однієї «дірки» в 11 КіБ.

Раніше вже говорилось, що реалізувати сегментацію можна двома способами: підкачуванням і розбивкою на сторінки. Дотепер мова йшла про підкачування. При такому підході за необхідності між пам'яттю й диском переміщаються цілі сегменти.

При великому розмірі сегментів може бути незручно або навіть неможливо зберігати їх в оперативній пам'яті цілком. Це приводить до ідеї їхньої сторінкової організації, щоб поблизу знаходилися тільки ті сторінки, які насправді потрібні.

У цьому випадку одні сторінки сегмента можуть перебувати в пам'яті, а інші – на диску. Щоб розбити сегмент на сторінки, для кожного сегмента потрібна окрема таблиця сторінок. Оскільки сегмент – це лінійний адресний простір, всі засоби розбивки на сторінки, які ми дотепер розглядали, можна застосовувати до будь-якого сегмента. Єдина різниця полягає в тому, що кожен сегмент одержує окрему таблицю сторінок.

1.2.2 Перша спроба створення сегментно-сторінкової організації пам'яті

Вперше організація пам'яті, подібна до описаної вище, була застосована в операційній системі MULTICS (Multiplexed Information and Computing Service – служба загальної інформації й обчислень) – це дуже стара операційна система (розроблена у 50-і роки минулого сторіччя), що сполучала сегментацію з розбив-

кою на сторінки. Вона була розроблена у Массачусетському технологічному інституті разом з компаніями Bell Labs і General Electric. Адреси в ОС MULTICS складалися із двох частин: номера сегмента й адреси всередині сегмента. Для кожного процесу існував ще додатковий сегмент дескриптора, котрий містив дескриптори для кожного сегмента процесу. Бітова структура цих дескрипторів наведена на рис. 10.

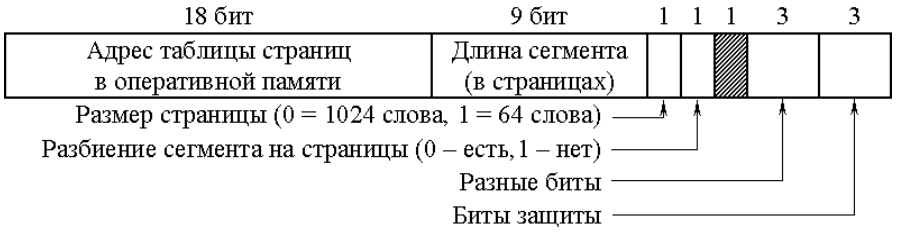

Рисунок 10 – Структура дескриптора системи MULTICS

Коли апаратне забезпечення одержувало віртуальну адресу, номер сегмента використовувався як індекс у сегменті дескрипторів для знаходження дескриптора потрібного сегмента (рис. 11).

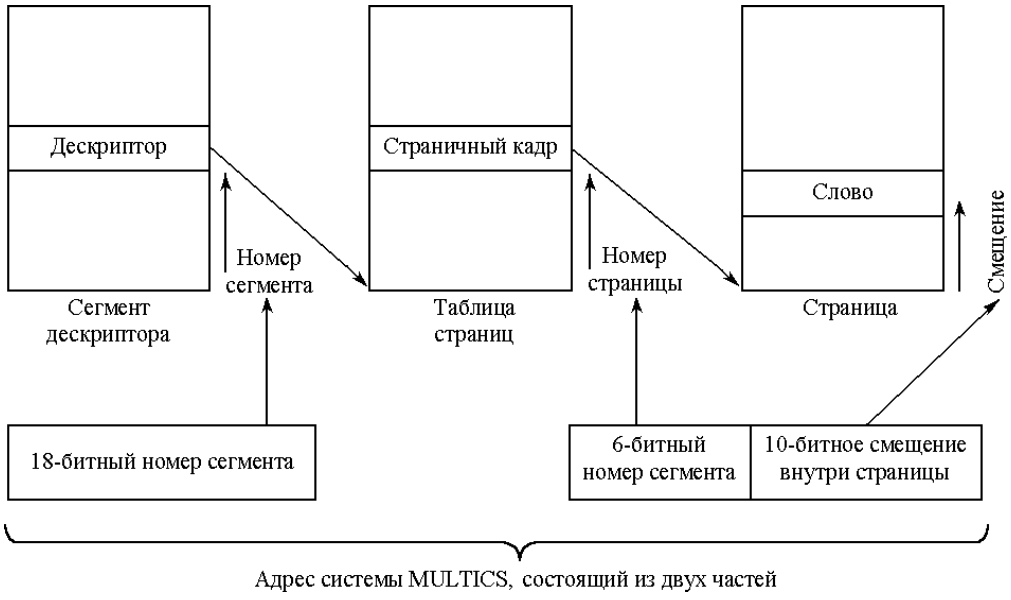

Рисунок 11 – Перетворення у системі MULTICS віртуальної адреси, яка складається з двох частин, на адресу фізичної пам’яті

В свою чергу, дескриптор указував на початок таблиці сторінок, що дозволяло розбивати кожен сегмент на сторінки звичайним способом. Для підвищення продуктивності декілька комбінацій сегмента й сторінки, які використовувались недавно, уміщувалися до асоціативної пам'яті, що була здатна утримувати до шістнадцяти елементів.

Операційна система MULTICS уже давно не застосовується, але віртуальна пам'ять всіх процесорів Intel, починаючи з 386-ї моделі, дуже схожа на цю систему.

1.3 Віртуальна пам'ять у процесорі Pentium

Процесор Pentium має складну систему віртуальної пам'яті, що підтримує виклик сторінок на вимогу, чисту сегментацію й сегментацію з розбивкою на сторінки. Віртуальна пам'ять описується двома таблицями: LDT (Local Descriptor Table – локальна таблиця дескрипторів) і GDT (Global Descriptor Table – глобальна таблиця дескрипторів). Кожна програма має свою власну локальну таблицю ***дескрипторів***, а єдина глобальна таблиця дескрипторів розділяється всіма програмами комп'ютера. Локальна таблиця дескрипторів LDT описує локальні сегменти кожної програми (її код, дані, стек і т.ін.), а глобальна таблиця дескрипторів GDT описує системні сегменти, у тому числі саму операційну систему.

Щоб одержати доступ до сегмента, процесор Pentium спочатку завантажує в один із сегментних регістрів так званий ***селектор сегмента***. Під час виконання програми 16-бітний сегментний регістр CS містить селектор для сегмента коду, а такий же сегментний регістр DS містить селектор для сегмента даних і т.д. Кожен селектор являє собою 16-бітне число (рис. 12).

Один з бітів селектора показує, є сегмент локальним або глобальним, тобто в якій із двох таблиць він перебуває: у локальній таблиці дескрипторів або в глобальній таблиці дескрипторів. Ще 13 бітів визначають номер комірки в локальній або глобальній таблиці дескрипторів, в якій міститься потрібній дескриптор. Тому обсяг кожної із цих таблиць обмежений у 8 Кібі (2^{13}) сегментних дескрипторів,

тобто у 64 КіБ. Два біти, що залишилися, пов’язані із захистом сегмента. Їх опис буде надано пізніше.

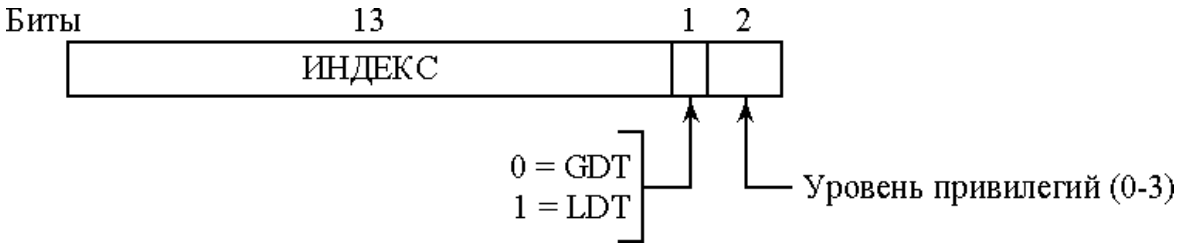

Рисунок 12 – Бітова структура селектора процесора Pentium

Нульова комірка таблиці дескрипторів відповідає недійсному дескриптору. Але у сегментний регістр можна завантажити значення, що вказує на цю комірку. Це робиться задля того, щоб показати, що регістр сегмента в цей момент є недоступним. Однак, якщо спробувати використати дескриптор 0, то така дія спричинить спрацювання пастки.

Коли селектор завантажується в сегментний регістр, то відповідний дескриптор викликається з локальної таблиці дескрипторів або із глобальної таблиці дескрипторів і зберігається у внутрішніх регістрах контролера керування пам’яттю, тому до нього можна швидко одержати доступ. Дескриптор складається з 8 байтів. В ньому містяться всі дані про сегмент – базова адреса сегмента, його розмір і інша інформація (рис. 13).

Розробниками процесора формат селектора був обраний таким чином, щоб спростити знаходження дескриптора. Спочатку на основі біта 2 у селекторі вибирається локальна таблиця дескрипторів LDT або глобальна таблиця дескрипторів GDT. Потім селектор копіюється в тимчасовий регістр контролера керування пам’яттю, а три молодших біти набирають значення 0, у результаті 13-бітне число селектора помножується на 8. Нарешті, до цього додається адреса локальної таблиці дескрипторів або глобальної таблиці дескрипторів (яка зберігається у внутрішніх регістрах контролера керування пам’яттю), і в результаті одержується покажчик на дескриптор. Наприклад, селектор із значенням 72 зверта-

ється до 9-го елемента у глобальній таблиці дескрипторів, що перебуває в пам'яті за адресою GDT+72.

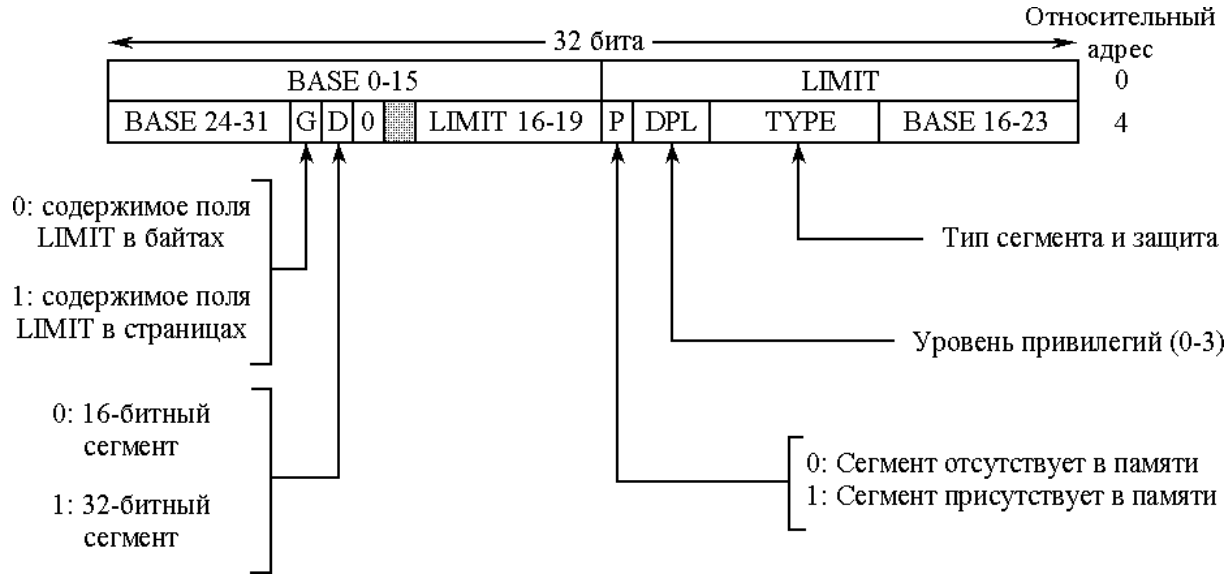

Рисунок 13 – Бітова структура дескриптора сегмента в процесорі Pentium

Певний інтерес викликає розгляд питання того, яким чином пара (селектор, зміщення) перетворюється на фізичну адресу. Як тільки апаратне забезпечення визначає, який саме сегментний регістр використовується, воно може знайти той дескриптор, що відповідає даному селектору у внутрішніх регістрах. Якщо такого сегмента не існує (селектор 0) або в цей момент він не перебуває в пам'яті (біт P=0), викликається системне переривання (пастка). У першому випадку – це помилка програмування и ОС відповідним чином реагує на цю ситуацію. Другий випадок вимагає, щоб операційна система довантажувала потрібний сегмент з диска.

Потім апаратне забезпечення перевіряє, чи не виходить зміщення за межі сегмента. Якщо виходить, то знову відбувається пастка. За логікою речей у дескрипторі повинно бути 32-бітне поле для визначення розміру сегмента, але там у наявності є всього 20 бітів, тому в цьому випадку використовується зовсім інша схема. Якщо поле G дескриптора (Granularity – ступінь деталізації) дорівнює 0, то

поле LIMIT дає точний розмір сегмента (максимальна межа – 1 МіБ). Якщо поле G дорівнює 1, то поле LIMIT указує розмір сегмента не в байтах, а в сторінках. Розмір сторінки в комп'ютері Pentium ніколи не буває менше за 4 КіБ, тому 20 бітів цілком достатньо для визначення сегментів розміром до 2^{32} байтів.

Якщо сегмент перебуває в пам'яті, а зміщення не виходить за межі сегмента, процесор додає значіння 32-бітного поля BASE дескриптора до зміщення, у результаті чого одержується лінійна адреса (рис. 14). Поле BASE розбивається на три частини й розноситься по дескриптору, щоб забезпечити сумісність із стареньким процесором моделі 80286, у якому розмір BASE становить усього 24 біта. Тому кожен сегмент може починатися з довільного місця в 32-бітному адресному просторі.

Рисунок 14 – Перетворення пари (селектор, зміщення) на лінійну адресу

Якщо розбивка на сторінки блокована (це визначається станом біта в регістрі глобального керування), лінійна адреса безпосередньо інтерпретується як фізична адреса й відправляється у пам'ять для читання або запису. Таким чином, при блокуванні розбивки на сторінки ми маємо чисту схему сегментації, де кожна базова адреса сегмента задається в його дескрипторі. Допускається перекриття сегментів, оскільки було б занадто утомливо витрачати багато часу на перевірку, щоб всі сегменти були непересічними.

З іншого боку, якщо розбивка на сторінки є дозволеною, лінійна адреса інтерпретується як віртуальна адреса й відображається на фізичну адресу з викори-

станням таблиць сторінок, майже як у прикладах, розглянутих вище. Єдина складність полягає в тому, що при 32-бітній віртуальній адресі й сторінках розміром 4 КіБ сегмент може містити 1 мільйон сторінок. Тому, щоб скоротити розмір таблиці сторінок для маленьких сегментів, застосовується дворівневе відображення.

Кожна працююча програма має спеціальну таблицю сторінок, що складається з 1024 32-бітних елементів. Її адреса вказується глобальним регістром. Кожен елемент у цій таблиці вказує на таблицю сторінок, що також містить 1024 32-бітних елемента. Елементи таблиці сторінок указують на сторінкові кадри. Така схема відображення зображена на рис. 15.

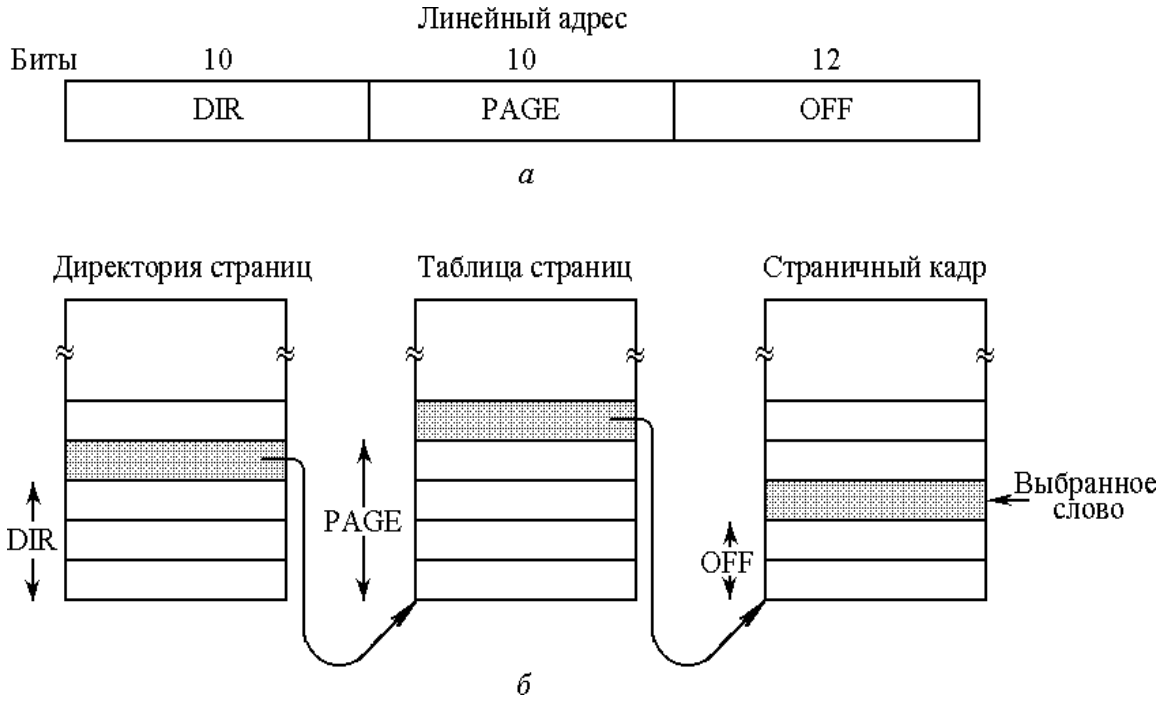

Рисунок 15 – Відображення лінійної адреси на фізичну у процесорі Pentium

На схемі видно, що лінійна адреса розбита на три поля: DIR, PAGE і OFF. Поле DIR використовується як індекс у директорії сторінок для знаходження покажчика на потрібну таблицю сторінок. Поле PAGE використовується як індекс у таблиці сторінок для знаходження фізичної адреси сторінкового кадру. Нарешті, поле OFF додається до адреси сторінкового кадру, і, таким чином, одержується фізична адреса потрібного байта або слова у пам’яті комп’ютера.

Розмір кожного елемента таблиці сторінок – 32 біта, 20 з яких містять номер сторінкового кадру. Біти, що залишилися, є біти доступу до сторінки і біт її змінення. Ці біти встановлюються апаратним забезпеченням для допомоги операційній системі. Тут також розташовані біти захисту сторінки та деякі інші біти.

У кожній таблиці сторінок містяться елементи для 1024 сторінкових кадрів по 4 КіБ кожен, тому одна таблиця сторінок працює з 4 МіБ пам'яті. Сегмент коротше 4 МіБ буде мати директорію сторінок з одним елементом (покажчиком на його єдину таблицю сторінок). Таким чином, непродуктивні витрати для коротких сегментів становлять усього дві сторінки, а не мільйон сторінок, як було б в одноровневій таблиці сторінок.

Щоб уникнути повторних звертань до пам'яті, пристрій керування пам'яттю у процесорі Pentium містить спеціальну апаратну підтримку для пошуку комбінацій DIR-PAGE, що використовувалися недавно, і відображення їх на фізичну адресу відповідного сторінкового кадру. Кроки, котрі показані на рис. 15, виконуються тільки в тому випадку, коли поточна комбінація не використовувалася досить тривалий час.

Є цілком очевидним, при застосуванні розбивки на сторінки значення поля BASE у дескрипторі може дорівнюватись 0. Єдине, для чого потрібно поле BASE, – це мотивувати невелике зміщення і використовувати елемент у середині директорії сторінок, а не на початку. Поле BASE є включеним до дескриптора тільки для здійснення чистої сегментації (без розбивки на сторінки), а також для зворотної сумісності зі старим процесором 80286, у якому не було розбивки на сторінки.

Слід відзначити, що коли конкретний додаток не має потреби в сегментації й задовольняється єдиним 32-бітним адресним простором зі сторінковою організацією, цього легко досягти. В такому разі всі сегментні регістри можуть бути заповнені тим самим селектором, дескриптор якого містить поле BASE, що дорівнює 0, і поле LIMIT, в яке занесене максимальне значення. Зміщення команди буде тоді лінійною адресою з єдиним адресним простором – по суті, традиційна розбивка на сторінки.

1.3.1 Рівні захисту пам’яті у процесорі Pentium

На завершення опису організації віртуальної пам’яті процесора Pentium варто сказати кілька слів про її захист, оскільки це має безпосереднє відношення до організації і до роботи віртуальної пам’яті. Pentium підтримує 4 рівні захисту, де рівень 0 – найбільш привілейований, а рівень 3 – найменш привілейований. Схематично ці рівні можуть представлятись у вигляді деякої умовної сфери, як це зображено на рис. 16. У кожен момент працююча програма перебуває на певному рівні, що вказується 2-бітним полем PSW (Program Status Word – слово стану програми) у спеціальному регістрі апаратного забезпечення, котрий містить коди умов і інші біти стану. Більш за те, кожен сегмент програми у системі також належить до певного рівня.

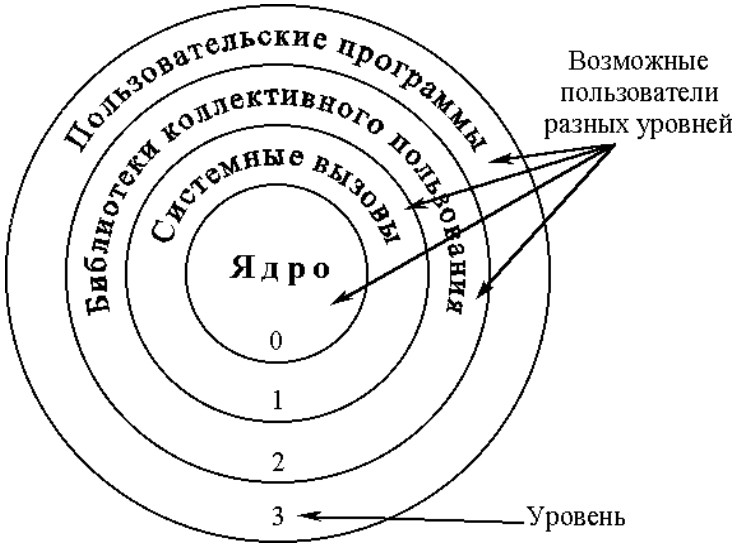

Рисунок 16 – Рівні захисту пам’яті у процесорі Pentium

Поки програма використовує сегменти тільки свого власного рівня, все йде нормально. Доступ до даних більш низького рівня дозволяється без ніяких перешкод. Доступ до даних більш високого рівня повністю забороняється. У випадку звертання програми зі свого до вищого рівня, відбувається системне переривання (пастка). Але є повністю допустимим виклик процедур як більш низького, так і більш високого рівня, але при цьому потрібно здійснювати дуже строгий конт-

роль. Для виклику процедури з іншого рівня команда CALL замість адреси звернення повинна містити селектор сегмента. Цей селектор обумовлює дескриптор, котрий вже вказує адресу потрібної процедури. Таким чином, це унеможливлює виконання переходу до середини довільного сегмента на іншому рівні. Можуть використовуватися тільки офіційно встановлені точки входу.

З рис. 16 видно, що на рівні 0 розташовується ядро операційної системи, яке контролює процеси вводу-виводу, роботу пам'яті і т.ін. На рівні 1 перебуває оброблювач системних викликів. Користувальницькі програми можуть викликати процедури із цього рівня, але тільки строго наперед визначені процедури. Рівень 2 містить бібліотечні процедури, які можуть розділятися декількома працюючими програмами. Користувальницькі програми можуть викликати ці процедури, але не можуть змінювати їх. Нарешті, на рівні 3 працюють користувальницькі програми, які мають найнижчий ступінь захисту. Система захисту Pentium, як і схема керування пам'яттю, у цілому заснована на ідеях системи MULTICS.

Пастки і переривання також використовують механізм, подібний до описаного вище. Вони теж звертаються до дескрипторів, а не до абсолютних адрес, а ці дескриптори вказують на процедури, які потрібно виконати. Поле TYPE на рис. 13 служить для встановлення відмінностей між сегментами коду, сегментами даних і різних типів логічних елементів.

1.4 Віртуальна пам'ять UltraSPARC

Традиційні таблиці сторінок, тип яких був розглянутий вище, вимагають по одному запису на кожну віртуальну сторінку, тому що вони індексуються за номером цієї сторінки. Якщо адресний простір складається з 2^{32} байтів і сторінки мають розмір 4096 байтів, тоді в таблиці сторінок повинне бути більше мільйона записів. При цьому таблиця сторінок буде займати мінімум 4 МіБ пам'яті. У досить великих системах це, очевидно, можна й здійснити.

Однак все частіше зустрічаються 64-розрядні комп'ютери й ситуація радикально змінюється. Тепер при адресному просторі 2^{64} байтів і при розмірі сторінки 4 КіБ, потрібна таблиця сторінок з 2^{52} записами. Якщо при цьому кожен запис

дорівнює 8 байтам, то таблиця займе більше 30 ТіБ. Виділення такої кількості пам'яті тільки для таблиці сторінок нереально сьогодні і не буде реальним коли-небудь в майбутньому. Отже, для 64-розрядного віртуального простору необхідно інше рішення.

Розроблювачі комп'ютерів багато років міркували над цією проблемою й нарешті дійшли нового вирішення проблеми. Воно засновано на спостереженні, що більшість програм схильна робити величезну кількість звертань до невеликої кількості сторінок. Таким чином, у таблиці сторінок тільки мала частка записів читається інтенсивно, інша частина навряд чи взагалі використовується.

У результаті було ухвалено таке рішення – забезпечити комп'ютер невеликим апаратним пристроєм, що слугує для відображення віртуальних адрес у фізичні без проходу по таблиці сторінок. Цей пристрій, називається ***буфером швидкого перетворення адреси*** (TLB – Translation Lookaside Buffer) або іноді ***асоціативною пам'яттю***. Він за звичаєм перебуває всередині диспетчера пам'яті й складається з декількох записів. Фактично записів рідко буває більше 64. Кожен запис, як і раніше, містить інформацію про одну сторінку.

Функціонування буфера швидкого перетворення адреси (TLB) відбувається в такий спосіб. Коли віртуальна адреса надходить у диспетчер пам'яті для відображення, апаратура TLB спочатку шляхом порівняння адреси з усіма записами одночасно (тобто паралельно) переконується в тім, що номер віртуальної сторінки, котра відповідає адресі, присутній в буфері. Якщо збіг знайдений і звертання не порушує біти захисту, номер сторінкового кадру береться просто-таки з буфера TLB, без переходу до таблиці сторінок. Якщо номер віртуальної сторінки присутній в буфері TLB, але інструкція намагається записати щось на сторінку, доступну тільки для читання, формується помилка захисту точно так само, як це відбувалося б і зі звичайною таблицею сторінок.

В іншому випадку, якщо номер віртуальної сторінки не знаходиться в буфері швидкого перетворення адреси. Диспетчер пам'яті виявляє відсутність сторінки й виконує звичайний пошук у таблиці сторінок. Потім він видаляє один із записів з буфера TLB і заміняє його тільки що знайденим записом з таблиці сторінок. Та-

ким чином, якщо ця сторінка знову незабаром буде викликана, то другого разу пошук вже виявиться успішним. Коли запис видаляється з буфера швидкого перетворення адреси, біт зміни копіюється в запис таблиці сторінок у пам'яті. Інші величини вже перебувають там. Коли буфер TLB завантажується з таблиці сторінок, всі поля беруться з пам'яті.

Подібна система віртуальної пам'яті з буфером швидкого перетворення застосована у комп'ютерах UltraSPARC фірми Sun. UltraSPARC – це 64-розрядна машина, що підтримує віртуальну пам'ять зі сторінковою організацією і з 64-бітними адресами. Проте, з різних причин програми не можуть використовувати повний 64-бітний віртуальний адресний простір. Підтримується тільки 44 біта, тому розмір програми не може перевищувати 1.8×10^{13} байтів. Припустима віртуальна пам'ять поділяється на 2 зони по 2^{43} байтів кожна, одна з яких перебуває у верхній частині віртуального простору, а інша – у нижній. Між ними перебуває «дірка», що містить адреси, які не використовуються. Спроба використати їх викликає помилку через відсутність сторінки.

Максимальна фізична пам'ять комп'ютера UltraSPARC становить 2^{41} байтів (2200 ГіБ). Підтримується 4 різних розмірів сторінок: 8 КіБ, 64 КіБ, 512 КіБ і 4 МіБ. Відображення цих чотирьох розмірів показані на рис. 17.

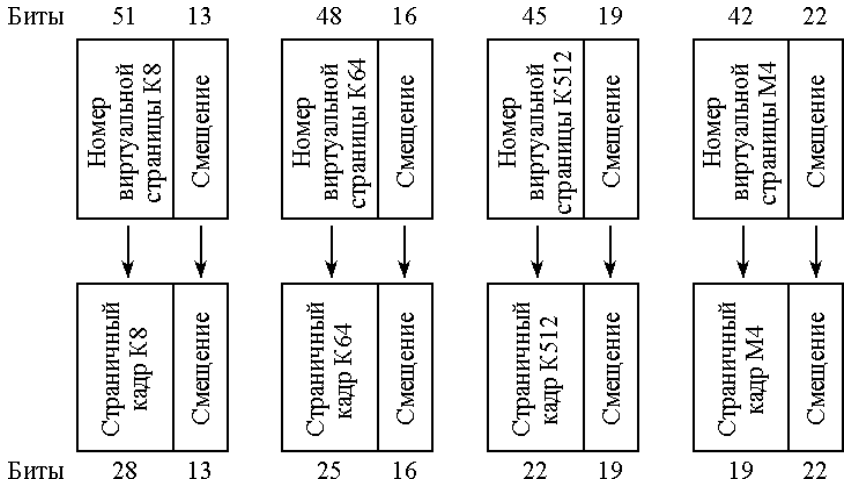

Рисунок 17 – Відображення віртуальних адрес на фізичні в комп'ютері UltraSPARC

Як вже вказувалось, через величезний віртуальний адресний простір звичайна таблиця сторінок (як в Pentium) не буде практичною. Тому в UltraSPARC пристрій керування пам'яттю містить таблицю TLB. Ця таблиця відображає номери віртуальних сторінок у номери фізичних сторінкових кадрів. Для сторінок розміром 8 КіБ існує 2^{31} номерів віртуальних сторінок, тобто більше двох мільярдів. Природно, не всі вони можуть бути відображені.

Тому TLB містить тільки номери найостанніших використовуваних віртуальних сторінок. Сторінки команд і сторінки даних розглядаються окремо (Ґарвардська RISC-архітектура процесора). Для кожної з таких категорій до TLB включаються номери 64 останніх віртуальних сторінок. Кожен елемент цього буфера включає номер віртуальної сторінки й відповідний номер фізичного сторінкового кадру. Коли процес звертається до свого контексту, віртуальна адреса в цього контексту передається у контролер керування пам'яттю, а той за допомогою спеціальної схеми порівнює номер віртуальної сторінки з усіма елементами буфера швидкого перетворення адреси TLB для даного контексту одночасно. Якщо виявляється збіг, номер сторінкового кадру в цьому елементі буфера з'єднується зі зміщенням, узятим з віртуальної адреси, щоб одержати 41-бітну фізичну адресу та обробляються деякі прапори (наприклад, біти захисту). Умовна структура буфера швидкого перетворення адреси TLB зображена на рис. 18*а*.

Якщо збіг не виявився, тобто відбувся промах в TLB, це викликає пастку в операційній системі. Обробляти помилку повинна сама операційна система. Слід відзначити, що даний промах відрізняється від помилки через відсутність сторінки. Промах буфера TLB може відбутися, навіть якщо потрібна сторінка присутня в пам'яті. Теоретично операційна система може сама завантажити новий елемент цього буфера для потрібної віртуальної сторінки. Однак для прискорення даної операції до цієї роботи підключається апаратне забезпечення, якщо програмне забезпечення взаємодіє з ним.

Але є цілком зрозумілим, що буфер пошуку лише на шістдесят чотири сторінки, котрі завантажувались у найближчій час, є дуже малим, тому у комп'ютері

UltraSPARC передбачений ще один рівень таблиці для зберігання віртуальних сторінок.

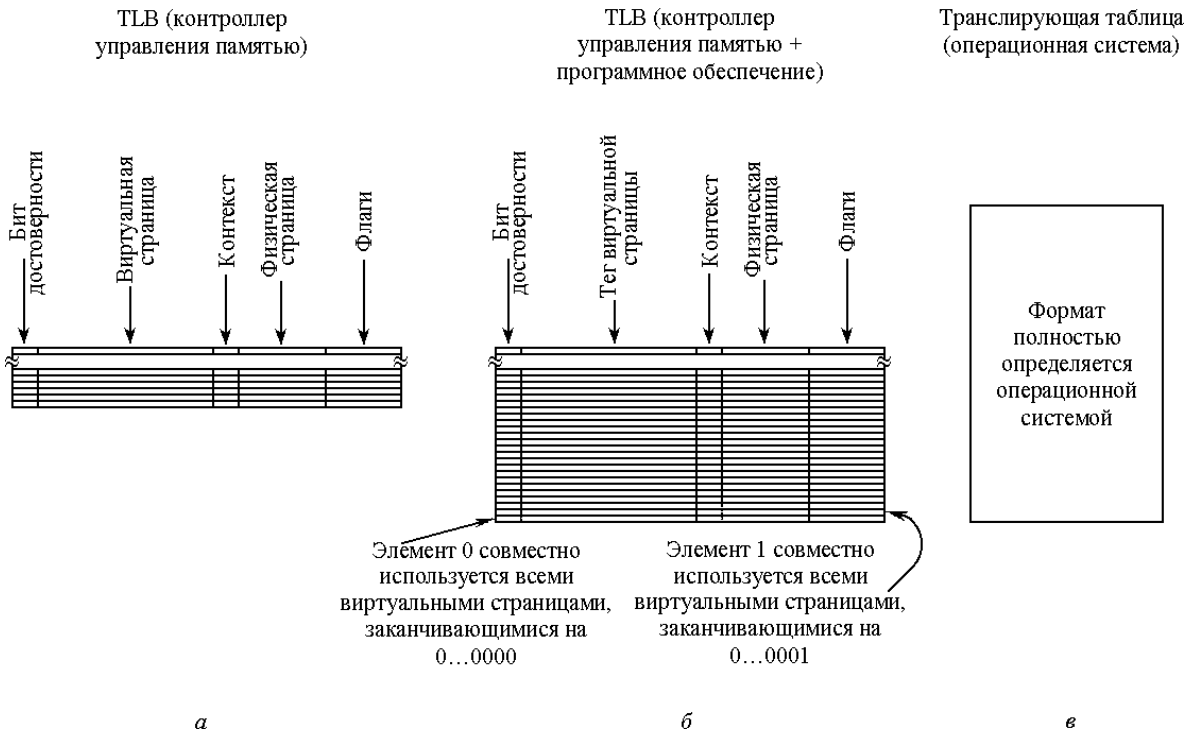

Рисунок 18 – Структури даних, що застосовуються для трансляції віртуальної адреси у комп’ютері UltraSPARC II: буфер швидкого перетворення адреси TLB (а); буфер зберігання перетворень TSB (б); таблиця трансляції (в)

Операційна система повинна зберігати найбільш часто використовувані елементи буфера TLB у таблиці за назвою ***буфер зберігання перетворень*** (TSB – Translation Storage Buffer). Ця таблиця побудована як кеш-пам’ять прямого відображення віртуальних сторінок. Кожен 16-байтний елемент цієї таблиці, як і таблиці TLB, вказує на одну віртуальну сторінку й містить біт вірогідності, номер контексту, тег віртуальної адреси, номер фізичної сторінки й декілька бітів прапорів. Якщо розмір кеш-пам’яті становить, скажемо, 8192 елемента, тоді всі віртуальні сторінки, у яких молодші 13 бітів адреси відображаються в 0000000000000, будуть претендувати на нульовий елемент у цій таблиці. Точно

так само всі віртуальні сторінки, у яких молодші біти адреси відображаються в 0000000000001, претендують на перший елемент у цій таблиці, як це показано на рис. 18*б*. Розмір таблиці визначається програмним забезпеченням і передається в контролер керування пам'яттю через спеціальні регістри, доступні тільки для операційної системи.

При промаху буфера збереження перетворень операційна система перевіряє, чи містить відповідний елемент буфера TLB потрібну віртуальну сторінку. Контролер керування пам'яттю обчислює адресу цього елемента і поміщає його у свій внутрішній регістр, який є доступним для операційної системи. Якщо потрібний елемент є в таблиці збереження перетворень, то який-небудь елемент видаляється з буфера TLB, а відповідний елемент буфера збереження перетворень копіюється туди. Апаратне забезпечення за допомогою алгоритму LRU вибирає, який саме елемент потрібно відкинути.

Якщо потрібної віртуальної сторінки немає в кеш-пам'яті, операційна система використовує іншу таблицю для знаходження інформації про сторінку, що може перебувати або не перебувати в основній пам'яті. Таблиця, що застосовується для цього пошуку, називається ***таблицею трансляції***. Оскільки тут апаратне забезпечення не бере участі у пошуку елементів, операційна система може використовувати будь-який формат. Наприклад, вона може хешірувати номер віртуальної сторінки, розділивши його на яке-небудь число **`p`**, і використовувати залишок для індексування таблиці покажчиків, кожен з яких указує на зв'язний список віртуальних сторінок, розділених на **`p`**. Слід сказати, що ці елементи – власне не є сторінками, а відповідають елементам таблиці TSB. Якщо пошук сторінки в таблиці трансляції привів до знаходження потрібної сторінки в пам'яті, то елемент TSB у кеш-пам'яті оновлюється. Якщо за результатами пошуку виявилося, що потрібної сторінки немає в пам'яті, то відбувається стандартна помилка відсутності сторінки у пам'яті.

Порівнюючи схеми розбивки на сторінки в Pentium і UltraSPARC можна зазначити наступне. Pentium підтримує чисту сегментацію, чисту розбивку на сторінки й сегментацію в сполученні з розбивкою на сторінки. UltraSPARC підтри-

мує тільки розбивку на сторінки. Pentium використовує апаратне забезпечення для перезавантаження елемента буфера TLB у випадку промаху TLB. UltraSPARC у випадку такого промаху просто передає керування операційній системі.

Причина цього розходження полягає в тому, що Pentium використовує 32-бітні сегменти, а такі маленькі сегменти (всього тільки 1 млн. сторінок) можуть оброблятися за допомогою лише сторінкових таблиць. Теоретично в Pentium могли б виникнути проблеми, якби програма використовувала тисячі сегментів, але тому що жодна з версій Windows або UNIX та й інших ОС не підтримує більше декількох сегментів на процес, ніяких проблем не виникає. UltraSPARC – 64-бітна машина. Вона може містити до 2 млрд. сторінок, тому технологія простих таблиць сторінок в цьому разі не працює. У найближчому часі всі машини будуть мати 64-бітні віртуальні адресні простори, і схема UltraSPARC, мабуть, стане найбільш поширеною системою керування віртуальною пам’яттю.

1.5 Віртуальна пам’ять і кешування

На перший погляд може здатися, що віртуальна пам’ять і кешування ніяк не зв’язані між собою, але насправді вони дуже подібні. При наявності віртуальної пам’яті вся програма зберігається на диску й розбивається на сторінки фіксованого розміру. Деяка підмножина цих сторінок перебуває в основній пам’яті. Якщо програма головним чином використовує сторінки з основної пам’яті, то помилки через відсутність сторінки будуть зустрічатися рідко й програма буде працювати швидко. При кешуванні вся програма зберігається в основній пам’яті й розбивається на блоки фіксованого розміру. Деяка підмножина цих блоків перебуває в кеш-пам’яті. Якщо програма головним чином використовує блоки з кеш-пам’яті, то промахи кеш-пам’яті будуть відбуватися рідко й програма буде працювати швидко. Як бачимо, віртуальна пам’ять і кеш-пам’ять ідентичні, тільки працюють вони на різних рівнях ієрархії архітектури комп’ютера.

Цілком природно, що віртуальна пам’ять і кеш-пам’ять мають деякі розходження. Так, наприклад, промахи кеш-пам’яті обробляються апаратним забезпеченням, а помилки через відсутність сторінок обробляються операційною систе-

мою. Крім того, блоки кеш-пам’яті звичайно набагато менше сторінок (наприклад, 64 байта у кеш-пам’яті і сторінки у 8 КіБ). Крім того, таблиці сторінок індексуються по старших битках віртуальної адреси, а кеш-пам’ять індексується по молодших битках адреси пам’яті. Проте, дуже важливо розуміти те, що розходження тут тільки в реалізації, а зовсім не у принципах і не у меті, з якою використовуються обидва ці механізми керування пам’яттю.

1.6 Питання для самоперевірки

1. Системи підкачування усувають вільні ділянки за допомогою ущільнення. Припустимо, що безліч вільних ділянок і безліч сегментів даних розподілені випадково, а час для читання 32-розрядного слова в пам'яті або запису туди дорівнює 10 нс. Скільки приблизно часу займе ущільнення 128 МіБ у цьому випадку? Для простоти вважаємо, що слово 0 – це частина незайнятої області й що найстарше слово пам’яті містить дійсні дані.
2. Зрівняєте кількість місця, необхідного для обліку вільної пам'яті за допомогою бітового масиву й за допомогою зв'язного списку. Пам’ять розміром 128 МіБ надається блоками по n байт. Для зв'язного списку передбачається, що пам'ять складається з послідовності, що чергується, сегментів і вільних областей, кожна по 64 КіБ. Також уважаємо, що для кожного вузла у зв'язному списку необхідна 32-розрядна адреса в пам’яті, 16 розрядів для довжини й 16 розрядів для поля посилання на наступний вузол. Скільки буде потрібно байтів для зберігання структур у кожному випадку? Який метод краще?
3. Розглянемо систему звичайного підкачування, у пам'яті якої втримуються вільні ділянки таких розмірів і в такому порядку: 10 КіБ, 4 КіБ, 20 КіБ, 18 КіБ, 7 КіБ, 9 КіБ, 12 КіБ і 15 КіБ. Який з них буде обрано для успішного задоволення запиту сегмента розміром

   а) 12 КіБ,

   б) 10 КіБ,

   в) 9 КіБ

по алгоритму «перший підходящий»? Відповідайте на те ж саме питання для алгоритмів «самий підходящий».

4. У чому різниця між фізичною адресою й віртуальною?
5. Для кожної з наступних десяткових віртуальних адрес: 20 000, 32 768, 60 000 обчислите номер віртуальної сторінки й зсув, якщо розмір сторінки дорівнює 4 КіБ або 8 КіБ.
6. Використовуючи таблицю сторінок на рис. 5, визначте фізичну адресу, що відповідає кожному з наступних віртуальних адрес:

   а) 20,

   б) 4100,

   в) 8300.
7. Процесор Intel 8086 не підтримує віртуальну пам'ять. Проте деякі компанії раніше продавали системи, що містять незмінений процесор 8086 і виконують сторінкове підкачування. Запропонуйте гіпотезу того, як вони це робили. Підказка: подумайте про логічне розташування диспетчера пам'яті (MMU).
8. Обсяг простору на диску, який повинен бути доступним для зберігання сторінок, пов'язаний з максимальною кількістю процесів **n**, кількістю байтів у віртуальному адресному просторі **v** і числом байтів в оперативній пам'яті **r**. Виведіть формулу вимог на дисковий простір у найгіршому разі. Наскільки ця величина є реалістичною?
9. Комп'ютер з 32-розрядною адресою використовує дворівневу таблицю сторінок. Віртуальні адреси розщеплюються на 9-розрядне поле верхнього рівня таблиці, 11-розрядне поле другого рівня таблиці сторінок і зсув. Чому дорівнює розмір сторінок і скільки їх в адресному просторі?
10. Припустимо, що 32-розрядна віртуальна адреса розбивається на чотири поля: **a**, **b**, **c** и **d**. Перші три використовуються для трирівневої системи таблиць сторінок. Четверте поле – це зсув. Чи залежить кількість сторінок від розміру всіх чотирьох полів? Якщо ні, то які з полів мають значення, а які ні?
11. Комп'ютер підтримує 32-розрядні віртуальні адреси й сторінки розміром 4

КіБ. Програма й дані разом уміщаються в наймолодшу сторінку (0-4095). Стек розміщується в найстаршій сторінці. Скільки записів у таблиці сторінок необхідно для цього процесу, якщо використовується традиційна (одно-рівнева) сторінкова структура? Скільки записів у таблиці сторінок потрібно при дворівневій сторінковій структурі, де кожна частина –10-розрядна?

12. Машина підтримує 48-розрядні віртуальні адреси й 32-розрядні фізичні адреси. Розмір сторінки дорівнює 8 КіБ. Скільки потрібно записів у таблиці сторінок?
13. Студент курсу конструювання компіляторів запропонував професорові проект написання компілятора, який одержує список сторінкових звертань, що може використовуватися для реалізації оптимального алгоритму заміщення сторінок. Це можливо? Чому «так» або чому «ні»? Чи існує який-небудь спосіб, що міг би підвищити ефективність сторінкового підкачування під час роботи?
14. Якщо використовується алгоритм заміщення сторінок FIFO у системі із чотирма сторінковими кадрами й вісьма сторінками, скільки сторінкових переривань відбудеться для послідовності звернень 0 1 7 2 3 2 7 1 0 3 за умови, що чотири сторінкових блоки споконвічно порожні? Тепер вирішіть це завдання для алгоритму LRU.
15. Скільки часу займе завантаження з диска програми розміром 64 КіБ, якщо його середній час пошуку дорівнює 10 мс, час обертання – 10 мс, кожна доріжка містить 32 КіБ

    а) для розміру сторінки 2 КіБ?

    б) для розміру сторінки 4 КіБ?

    Сторінки розкидані по диску випадково, і кількість циліндрів така велика, що можна ігнорувати варіант, при якому дві сторінки опиняються на тому самому циліндрі.
16. Комп’ютер забезпечує кожен процес 65536 байтами адресного простору, розділеного на сторінки по 4096 байт. Якась програма має розмір тексту 32 768 байт, розмір даних 16 386 байт і розмір стека 15 870 байт. Чи помістить-

ся ця програма в адресному просторі? А якби розмір сторінки був 512 байт, вона помістилася б? Пам’ятаєте, що сторінка не може вміщати частини двох різних сегментів.

17. Чи може сторінка виявитися у двох робочих наборах одночасно? Поясніть.
18. Поясніть різницю між внутрішньою й зовнішньою фрагментацією. Яка з них відбувається в сторінкових системах? А яка має місце в системах, що використовують чисту сегментацію?
19. Коли підтримуються й сегментація, і сторінкова організація пам'яті, як у системі MULTICS, спочатку слід знайти дескриптор сегмента, потім ідентифікатор сторінки. Чи може при такому дворівневому пошуку працювати також буфер швидкого перетворення адреси (TLB)?
20. Машина містить 32-бітний віртуальний адресний простір з побайтовою адресацією. Розмір сторінки становить 8 КіБ. Скільки існує сторінок віртуального адресного простору?
21. Віртуальна пам’ять містить 8 віртуальних сторінок і 4 фізичних сторінкових кадри. Розмір сторінки становить 1024 слова. Нижче наведена таблиця сторінок:

Віртуальна сторінка	Сторінковий кадр
0	3
1	1
2	Немає в основній пам’яті
3	Немає в основній пам’яті
4	2
5	Немає в основній пам’яті
6	0
7	Немає в основній пам’яті

Створіть список віртуальних адрес, звертання до яких буде викликати помилку через відсутність сторінки. Які фізичні адреси для 0, 3728, 1023, 1024, 1025, 7800 і 4096?

22. Комп'ютер має 16 сторінок віртуального адресного простору й тільки 4 сторінкових кадри. Спочатку пам'ять порожня. Програма звертається до віртуальних сторінок у наступному порядку: 0, 7, 2, 7, 5, 8, 9, 2, 4

    а. Які зі звернень викличуть помилку за алгоритмом LRU?

    б. Які зі звернень викличуть помилку з алгоритмом FIFO?

23. У розділі був запропонований алгоритм заміщення сторінок FIFO. Розробіть більш ефективний алгоритм. Підказка: можна обновляти лічильник у сторінці, що завантажується знову, залишаючи всі інші.

24. У системах зі сторінковою організацією пам'яті, які ми обговорювали в цій главі, оброблювач помилок, які відбуваються через відсутність сторінок, був частиною рівня архітектури команд і, отже, не був присутній в адресному просторі операційної системи. На практиці ж цей оброблювач займає деякі сторінки й може бути вилучений при певних обставинах (наприклад, відповідно до політики заміщення сторінок). Що б трапилося, якби оброблювача помилок не було в наявності в той момент, коли відбулася помилка? Як можна було б розв'язати цю проблему?

25. Не всі комп'ютери містять спеціальний біт, що автоматично встановлюється, коли виробляється запис у сторінку. Однак потрібно якимсь образом стежити, які сторінки були змінені, щоб не довелося записувати всі сторінки назад на диск після їхнього використання. Якщо припустити, що кожна сторінка має спеціальні біти для дозволу читання, записи й виконання, то як операційна система може стежити, які сторінки змінювалися, а які – ні?

26. Сегментована пам'ять містить сторінкові сегменти. Кожна віртуальна адреса містить 2-бітний номер сегмента, 2-бітний номер сторінки й n-бітний зсув усередині сторінки. Основна пам'ять містить 32Кбайт, які розділені на сторінки по 2Кбайт. Кожен сегмент дозволяється або тільки читати, або читати й виконувати, або читати й записувати, або читати, записувати й виконувати. Таблиці сторінок із вказівкою на захист наведені нижче:

Сегмент 0		Сегмент 1		Сегмент 2	Сегмент 3	
Тільки для читання		Читання/виконання		Читання/запис/ виконання	Читання/запис	
Віртуальна сторінка	Сторінковий кадр	Віртуальна сторінка	Сторінковий кадр		Віртуальна сторінка	Сторінковий кадр
0	9	0	На диску	Таблиці	0	14
1	3	1	0	сторінок	1	1
2	На диску	2	15	немає	2	6
3	12	3	8	в основній пам’яті	3	На диску

Обчисліть фізичну адресу для кожного з нижчеперелічених доступів до віртуальної пам’яті. Якщо відбувається помилка, вкажіть, якого вона типу.

Доступ	Сегмент	Сторінка	Зсув усередині сторінки
Виклик даних	0	1	1
Виклик даних	1	1	10
Виклик даних	3	3	2047
Збереження даних	0	1	4
Збереження даних	3	1	2
Збереження даних	3	0	14
Перехід	1	3	100
Виклик даних	0	2	50
Виклик даних	2	0	5
Перехід	3	0	60

27. Деякі комп’ютери дозволяють здійснювати ввід-вивід безпосередньо в ко-

ристувальницький простір. Наприклад, програма може почати передачу даних з диска в буфер усередині користувальницького процесу. Чи спричинить це які-небудь проблеми, якщо для реалізації віртуальної пам'яті використовується ущільнення? Аргументуйте.

28. Операційні системи, у яких допускаються файли, що відображуються на пам'ять, завжди вимагають, щоб файли були відображені в границях сторінок. Наприклад, якщо в нас є сторінки по 4 КіБ, файл може бути відображений, починаючи з віртуальної адреси 4096, але не з віртуальної адреси 5000. Навіщо це потрібно?
29. При завантаженні сегментного регістра в Pentium викликається відповідний дескриптор, що завантажується в невидиму частину сегментного регістра. Як ви думаєте, чому розроблювачі Intel вирішили це зробити?
30. Програма в комп'ютері Pentium звертається до локального сегмента 10 зі зсувом 8000. Поле BASE сегмента 10 у локальній таблиці дескрипторів містить число 10000. Який елемент таблиці сторінок використовує Pentium? Який номер сторінки? Яке зсув?
31. Розгляньте можливі алгоритми для видалення сегментів у сегментированной пам'яті без сторінкової організації.
32. Зрівняєте внутрішню фрагментацію із зовнішньою фрагментацією. Що можна зробити, щоб поліпшити кожну з них?
33. Супермаркети часто зіштовхуються із проблемою, подібною із заміщенням сторінок у системах з віртуальною пам'яттю. У супермаркетах є фіксована площа простору на полках, куди потрібно поміщати усе більше й більше різних товарів. Якщо надійшов новий важливий продукт, на приклад харчування для собак дуже високої якості, який-небудь інший продукт потрібно забрати, щоб звільнити місце для нового продукту. Ми знаємо два алгоритми: LRU і FIFO. Який з них ви б зволіли?
34. Чому блоки кеш-пам'яті завжди набагато менші, ніж сторінки у віртуальній пам'яті (буває навіть, що в 100 разів менші)?

2 ВІРТУАЛЬНІ КОМАНДИ ВВОДУ-ВИВОДУ

Наступною задачею, що повинна вирішити ОС – це забезпечення роботи із пристроями довгострокової пам’яті. Як правило, ОС управляє вільним простором на цих носіях і структурує користувальницькі дані. Навіть, як витікає з попереднього, обслуговування віртуальної пам’яті також цілком і повністю залежить від здатності системи працювати з пристроями вводу-виводу.

У архітектурі комп’ютерів вирізняються три рівня команд (див. рис. 1). Причому набір команд рівня архітектури команд повністю відрізняється від набору команд мікроархітектурного рівня. І самі операції, і формати команд на цих двох рівнях різні. Наявність декількох однакових команд випадкова.

Навпаки, набір команд рівня операційної системи містить більшу частину команд з рівня архітектури команд, а також кілька нових дуже важливих команд. В свою чергу деякі непотрібні команди не включаються до рівня операційної системи. Але ввід-вивід – це одна з областей, у яких ці два рівні різняться дуже сильно. Причина такого розходження проста. По-перше, користувач, здатний виконувати команди вводу-виводу рівня архітектури команд, легко зможе одержати доступ до конфіденційної інформації, що зберігається де-небудь у системі, і взагалі такий користувач буде загрозою для самої системи. По-друге, звичайні прикладні програмісти не мають бажання здійснювати ввід-вивід за допомогою команд, що належать до рівня архітектури команд, оскільки це є занадто складним й утомливим. Замість простих команд звернення до системи вводу-виводу, на цьому рівні обов’язково потрібно встановити певні поля й біти в низці регістрів процесора, а також, у регістрах пристроїв, потім почекати, поки операція закінчиться, і перевірити, що відбулося. Дисководи звичайно встановлюють біти регістрів пристроїв для виявлення таких помилок.

1) Апаратура диска не змогла виконати позиціонування.
2) Як буфер вводу-виводу визначений неіснуючий елемент пам’яті.
3) Процес вводу-виводу з диска (на диск) почався до того, як закінчився попередній.

4) Помилка синхронізації при зчитуванні.
5) Звертання до неіснуючого диска.
6) Звертання до неіснуючого циліндра пристрою.
7) Звертання до неіснуючого сектора.
8) Помилка перевірки запису після операції запису.

І це ще зовсім не вичерпний перелік. При наявності однієї із цих помилок установлюється відповідний біт у регістрі пристрою. Звісно, що звичайному (не системному) програмісту, як вже вказувалось, не надає ніякої радості перевіряти всі регістри пристроїв і окремі біти, що встановлені в них, після кожної, мабуть невеликої, операції вводу-виводу. Тому всі сучасні операційні системи пропонують користувачам дещо інший механізм.

2.1 Файли, їх використання і властивості

Один зі способів організації віртуального вводу-виводу – використання ***абстракції за назвою файл***. Файли відносяться до абстрактного механізму. Вони надають спосіб зберігати інформацію на диску й зчитувати її знову пізніше. При цьому від користувача повинні приховуватися такі деталі, як спосіб і місце зберігання інформації, а також деталі роботи дисків. Файли можуть бути структуровані кількома різними способами. Файл може бути неструктурованою послідовністю байтів. У цьому випадку операційна система не цікавиться вмістом файла. Усе, що вона бачить – це байти. Значення цим байтам надається програмами рівня користувача. Такий підхід використовується як у системах типу UNIX, так і в Windows. Розгляд операційною системою файлів як простої послідовності байтів забезпечує максимальну гнучкість. Програми користувача можуть поміщати у файли все що завгодно й іменувати їх будь-яким зручним для них способом. Операційна система не втручається в цей процес, що може бути особливо цінним для користувачів, які збираються зробити що-небудь незвичне.

Іноді файл – це послідовність записів фіксованої довжини, кожна зі своєю внутрішньою структурою. Для файлів, що складаються із записів, важливим є те, що операція читання повертає один запис, а операція запису перезаписує або до-

повнює один запис. Такий спосіб представлення файлів є досить застарілим і практично жодна сучасна ОС його не застосовує.

Є ще третій варіант внутрішньої структури файлів – дерево. При такій організації файл являє собою дерево записів, не обов'язково однієї й тієї ж довжини. Кожен запис у фіксованій позиції містить поле ключа. Дерево є сортованим по ключовому полю, що забезпечує швидкий пошук заданого ключа. Основною файловою операцією тут є не одержання наступного запису, хоча це також можливо, а одержання запису із зазначеним значенням ключа. При додаванні нових записів операційна система, а не користувач, повинна вирішувати, куди їх помістити. Такий тип файлів принципово відрізняється від неструктурованих потоків байтів, котрі застосовуються в UNIX і Windows, але вони мають широке застосування на великих мейнфреймах, які ще використовуються для комерційної обробки даних.

Якщо пристрій вводу-виводу є пристроєм зберігання інформації (наприклад, диск), то файл можна записати, а потім можна знов зчитати назад. Якщо пристрій не є пристроєм зберігання інформації (наприклад, це принтер), то файл звідти зчитати не можна. На диску може зберігатися багато файлів, у кожному з яких утримуються дані певного типу, наприклад картинка, великоформатна таблиця або текст. Файли мають різну довжину й мають різні властивості. Ця абстракція дозволяє легко організувати віртуальний ввід-вивід.

Операційні системи підтримують різні типи файлів. Наприклад, у системах типу UNIX і у Windows є певна неоднаковість між ***регулярними*** (звичайними) ***файлами*** й ***каталогами***. У системі UNIX також різняться ***символьні*** й ***блокові спеціальні файли***. До регулярних файлів належать всі файли, що містять інформацію користувача. Каталоги – це системні файли, що забезпечують підтримку структури файлової системи. Вони будуть розглянуті докладніше нижче. Символьні спеціальні файли мають відношення до вводу-виводу й використовуються для моделювання послідовних пристроїв вводу-виводу, таких як термінали, принтери й мережі. Блокові спеціальні файли використовуються для моделювання дисків.

У першу чергу розгляду підлягають регулярні файли. Регулярні файли в основному є або ASCII-файлами, або двійковими файлами. ***ASCII-файли*** складаються з текстових рядків. У деяких системах кожен рядок завершується символом повернення каретки. В інших (наприклад, UNIX) використовується символ переведення рядка. У деяких системах (наприклад, Windows) використовуються обидва символи. Рядки не зобов'язані мати ту ж саму однакову довжину.

Великою перевагою ASCII-файлів є те, що вони можуть відображатися на екрані й виводитися на друк так, як є, без якого-небудь перетворення, і можуть редагуватися практично будь-яким текстовим редактором. Більш того, якщо велика кількість програм використовує ASCII-файли для входу й виходу, то виявляється нескладним з'єднати вхід однієї програми з виходом іншої, як це робиться в конвеєрах командної оболонки (обмін даними між процесами при цьому не стає простіше, але інтерпретація інформації полегшується, якщо для її вираження застосовується стандарт, такий як ASCII). Інші файли ***називаються двійковими***, тобто вони не є ASCII-файлами. При виведенні їх на принтер виходить незрозумілий набір символів, що нагадує випадкове сміття. Звичайно в них є якась внутрішня структура, відома програмі, що використовує їх.

2.2 Віртуальні команди вводу-виводу

Для операційної системи файл є просто послідовністю байтів, як описано вище. Ввід-вивід файла здійснюється за допомогою системних викликів для відкриття, читання, запису й закриття файлів. Перед тим як зчитувати файл, його потрібно відкрити. Процес відкриття файла дозволяє операційній системі знайти файл на диску й передати у пам'ять інформацію, необхідну для доступу до цього файла.

Після відкриття файла його можна зчитувати. Системний виклик для зчитування повинен мати як мінімум такі параметри:

- покажчик, який саме відкритий файл потрібно зчитувати;
- покажчик на буфер у пам'яті, у який потрібно помістити дані;
- кількість зчитуваних байтів.

Даний системний виклик поміщає потрібні дані у буфер. За звичаєм він повертає число зчитаних байтів. Це число може бути менше від того числа, що запитувалося спочатку (наприклад, не можна прочитати 2000 байтів з файла, що має розмір лише 1000 байт).

З кожним відкритим ***файлом зв'язується покажчик***, який повідомляє, до якого байта у файлі буде виконуватись звернення на наступній операції. Після команди **`read`** покажчик інкрементується числом зчитаних байтів, тому послідовні команди **`read`** зчитують послідовні блоки даних з файла. Щоб програми могли одержувати безпосередній доступ до будь-якої частини файла, у ОС є засоби, які дозволяють встановити цьому покажчику довільне значення. Коли програма закінчила зчитування файла, вона може закрити його й повідомити операційній системі, що вона більше не буде використовувати цей файл. Операційна система зможе звільнити простір у таблиці, у якій зберігалася інформація про цей файл.

В операційних системах для універсальних обчислювальних машин файл є більш складною структурою. Тут файл може бути послідовністю логічних записів, кожна з яких має строго певну структуру. Наприклад, логічний запис може бути структурою даних, що складається з деякої кількості елементів (наприклад, п'яти елементів: двох рядків символів, "Ім'я" і "Начальник", двох цілих чисел "Відділ" і "Кімната", і одного логічного числа "Стать"). Деякі операційні системи розрізняють файли, у яких всі елементи мають однакову структуру, і файли, що містять різні типи даних.

Основна віртуальна команда вводу – **`read`** зчитує наступний запис із потрібного файла й поміщає його в основну пам'ять, починаючи з певної адреси, як показано на рис. 19. Щоб виконати цю операцію, віртуальна команда повинна одержати відомості про те, який файл зчитувати й куди в пам'яті помістити запис. Часто існують параметри для читання деякого запису, які визначаються або за його місцем у файлі, або за його ключем.

Основна віртуальна команда виводу записує логічний запис із пам'яті до файла. Послідовні команди **`write`** роблять послідовні логічні записи до файла.

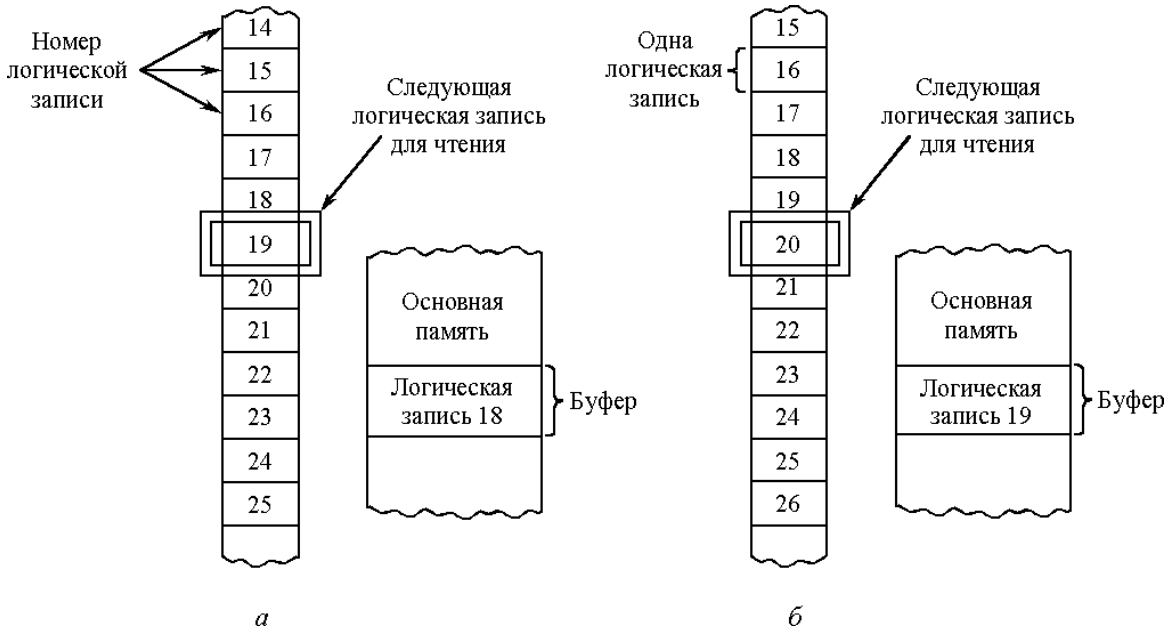

Рисунок 19 – Читання файла, який складається з логічних записів: до читання запису а) й після читання запису б)

2.3 Реалізація віртуальних команд вводу-виводу

Щоб зрозуміти, як реалізуються віртуальні команди вводу-виводу, потрібно знати, як файли організуються й зберігаються. Основне питання тут – розподіл дискового простору для файлів. Файли зберігаються у ***дискових блоках***. Одиничним блоком може бути просто сектор на диску, але частіше блок охоплює декілька секторів.

Ще одна фундаментальна властивість реалізації системи файлів – це питання про те, чи зберігається файл у послідовних блоках диска чи ні. На рис. 20 зображений простий диск із одною поверхнею, що складається з 5 доріжок по 12 секторів кожна. На рис. 20 *а* файл складається з послідовних секторів. Послідовне розташування п’яти блоків звичайно застосовується на компакт-дисках. На рис. 20 *б* файл займає непослідовні сектори. Така схема є нормою для жорстких дисків.

Сприйняття файла прикладним програмістом сильно відрізняється від сприйняття файла операційною системою. Програміст сприймає файл як лінійну

послідовність байтів або логічних записів. Операційна система сприймає файл як упорядковану, хоча необов'язково послідовну, сукупність одиничних блоків на диску.

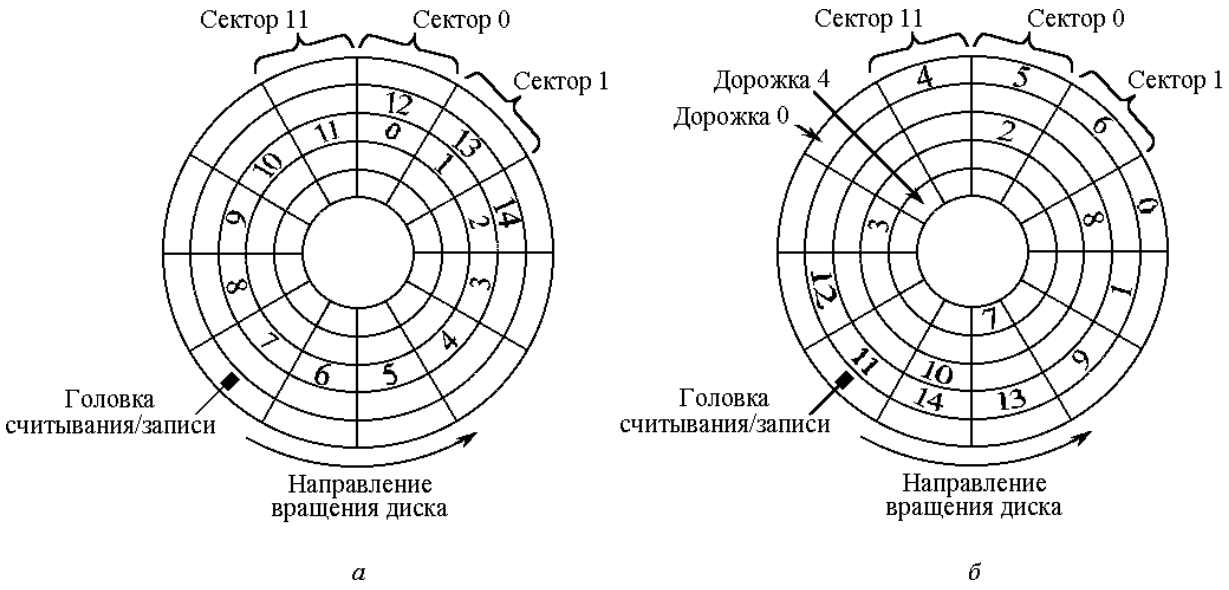

Рисунок 20 – Варіанти розташування файла на диску:
а – файл займає послідовні сектори; б – непослідовні сектори

Для того, щоб операційна система могла доставити на вимогу байт або логічний запис **n** з якогось файла, вона повинна користуватися яким-небудь методом для визначення місцезнаходження даних. Якщо файл розташований послідовно, операційна система повинна знати тільки місце початку файла, щоб обчислити позицію потрібного байта або логічного запису.

Якщо файл розташований на диску непослідовно, то неможливо тільки за початковою позицією файла обчислити позицію довільного байта або логічного запису в цьому файлі. Щоб знайти довільний байт або логічний запис, потрібна таблиця (так званий індекс файла), що визначає для файла номери одиничних блоків і їхні фізичні адреси на диску. Індекс файла можна організувати або у вигляді списку адрес блоків (така схема використовується в UNIX), або у вигляді списку логічних записів, для кожного з яких дається адреса блоку на диску й зміщення запису відносно початку блока. Іноді кожен логічний запис має ключ, і

програми можуть звертатися до запису за цим ключем, а не за номером логічного запису. В останньому випадку кожен елемент таблиці повинен містити не тільки інформацію про місцезнаходження запису на диску, але і його ключ. Така структура звичайно застосовується в ОС універсальних обчислювальних машин.

Альтернативний метод знаходження блоків файла – організувати файл у вигляді зв'язного списку. Кожен одиничний блок містить адресу наступного одиничного блоку. Для реалізації цієї схеми потрібно в основній пам'яті мати таблицю з усіма наступними адресами. Наприклад, для диска з 64 кібі-блоків операційна система може мати в пам'яті таблицю з 64 кібі-елементів, у кожному з яких задається індекс наступного одиничного блоку. Таким чином, якщо файл займає блоки 4, 52 і 19, то елемент 4 у таблиці буде містити число 52, елемент 52 буде містити число 19, а елемент 19 буде містити спеціальний код (наприклад, 0 або -1), що вказує на кінець файла. Так працюють системи файлів в MS DOS, Windows 95 і Windows 98. Windows NT підтримує цю систему файлів, але крім цього має свою власну систему файлів, що більше схожа на файлову систему UNIX.

Дотепер ми обговорювали як послідовно розташовані, так і непослідовно розташовані на диску файли, але ми ще не пояснили, навіщо потрібні ці два типи розташування. Послідовно розташованими файлами легко управляти, але якщо максимальний розмір файла не відомий заздалегідь, цю технологію не можна використовувати. Якщо файл починається із сектора **j** і розростається в послідовні сектори, він може наткнутися на інший файл у секторі **k**, і йому не вистачить місця для розширення. Якщо файл розташовується непослідовно, то таких проблем не виникає, оскільки наступні блоки можна помістити в інше місце на диску. Якщо диск містить низку файлів, що збільшуються, і кінцеві розміри яких невідомі, записати їх у послідовні блоки на диску практично неможливо. Іноді можна перемістити існуючий файл, але це є дуже накладним з точці зору затрачуваного часу.

З іншого боку, якщо максимальний розмір всіх файлів відомий заздалегідь (наприклад, це має місце, коли створюється компакт-диск), програма може заздалегідь визначити серії секторів, точно рівних за довжиною кожному файла. Якщо файли по 1200, 700, 2000 і 900 секторів потрібно помістити на компакт-диск, вони

просто можуть починатися з секторів 0, 1200, 1900 і 3900 відповідно (вміст тут не враховується). Знайти будь-яку частину будь-якого файла легко, оскільки перший сектор файла завжди є відомим.

Це те, що стосується пошуку потрібного запису на диску. З іншого боку, щоб розподілити простір на диску для файла, операційна система повинна відстежувати, які блоки доступні, а які вже зайняті іншими файлами. Коли створюється запис на компакт-диск, обчислення розташування файлів виконується один раз і назавжди, а на жорсткому диску файли постійно записуються й видаляються, тому розташування вільних блоків теж змінюється. Один зі способів відстеження доступних блоків – збереження списку всіх «дірок» (невикористаних просторів), де «дірка» – це будь-яке число суміжних одиничних блоків. Цей список називається ***списком вільної пам'яті.*** На рис. 21 *а* зображений список вільної пам'яті для диска з рис. 20 *б.*

Дорожка	Сектор	Количество секторов в дырке
0	0	5
0	6	6
1	0	10
1	11	1
2	1	1
2	3	3
2	7	5
3	0	3
3	9	3
4	3	8

а

	Сектор											
Дорожка	0	1	0	1	0	1	0	1	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	1	0
2	1	0	1	0	0	0	1	0	0	0	0	0
3	0	0	0	1	1	1	1	1	1	0	0	0
4	1	1	1	0	0	0	0	0	0	0	0	1

б

Рисунок 21 – Два способи відстеження вільних секторів:
список вільної пам'яті (а) і бітове відображення (б)

Альтернативний підхід – зберегти бітове відображення, у форматі один біт на одиничний блок, як показано на рис. 21*б*. Біт зі значенням 1 показує, що даний блок уже зайнятий, а біт зі значенням 0 показує, що даний блок вільний.

Перший підхід дозволяє легко знаходити дірку потрібної довжини. Однак у цього методу є досить суттєвий недолік: в міру створення й знищення файлів довжина списку буде змінюватися, а це небажано. Перевага таблиці бітів полягає в тому, що вона має постійний розмір. Крім того, для зміни статусу одиничного блоку з вільного на зайнятий потрібно поміняти значення всього одного біта. Однак при такому підході важко знайти блок потрібного розміру. Обидва методи потребують, щоб при записі файла на диск або видаленні файла з диска список розміщення або таблиця обновлялися.

На завершення обговорення питання щодо реалізації системи файлів, потрібно сказати ще декілька слів про розмір одиничного блоку. При визначенні його розміру відіграють роль кілька факторів. По-перше, слід відзначити, що час пошуку потрібного блоку й час, затрачуваний на обертання диска, загальмовують доступ до диска. Якщо на знаходження початку блоку витрачається 10 мс, то набагато вигідніше зчитати 8 КіБ за одним разом (це займе приблизно 1 мс), чим вісім раз зчитувати блоки розміром 1 КіБ (це займе приблизно той же час в 1 мс, тобто 8*0,125 мілісекунди), але при зчитувані 8 КіБ як 8 блоків по 1 КіБ, потрібно буде здійснювати пошук 8 разів. В зв’язку з цім бажано для підвищення продуктивності застосовувати як можна більші одиничні блоки.

По-друге, чим менший розмір одиничного блоку, тим більше повинно бути їх на диску того ж самого розміру. Велика кількість одиничних блоків, у свою чергу, спричиняє довгі індекси файлів і більші таблиці в пам’яті. Системі MS DOS довелося перейти на багатосекторні блоки з тієї причини, що в цій системі дискові адреси зберігалися у вигляді 16-бітних чисел. Коли розмір дисків став перевищувати 2^{16} секторів, представити їх можна було, тільки використовуючи одиничні блоки більшого розміру, для того щоб число таких блоків не перевищувало 2^{16}. Перші випуски системи Windows мали таку ж саму проблему, однак в наступних випусках вже використовувалися 32-бітні числа. Зараз системи Windows підтримують обидва розміри.

З іншого боку маленькі блоки теж мають свої переваги. Справа в тому, що файли дуже рідко (практично ніколи) займають рівно ціле число одиничних бло-

ків. Отже, практично в кожному файлі в останньому одиничному блоці залишиться невикористаний простір. Якщо розмір файла сильно перевищує розмір одиничного блоку, то в середньому невикористаний простір буде становити половину блоку. Чим більше блок, тим більше залишається невикористаного простору. Якщо середній розмір файла набагато менший від розміру одиничного блоку, більша частина простору на диску буде невикористаною. Наприклад, у системі MS DOS або в перших версіях Windows з розділом диска в 2 ГіБ розмір одиничного блоку становив 32 КіБ, тому при записі на диск файла в 100 символів 32668 байтів дискового простору пропадали даремно. Таким чином з погляду розподілу дискового простору маленькі одиничні блоки мають переваги над більш великими. У цей час найважливішим фактором уважається швидкість передачі даних, тому розмір блоків постійно збільшується.

2.4 Команди керування директоріями

Багато років тому (у середині минулого сторіччя) програми й дані зберігалися на перфокартах. Оскільки розмір програм збільшувався, а даних ставало усе більше, така форма зберігання стала незручною. Тоді виникла ідея замість перфокарт використовувати для зберігання програм і даних допоміжну пам’ять (наприклад, диск). Інформація, доступна для комп’ютера без втручання людини, називається ***неавтономною***. ***Автономна*** інформація, навпаки, потребує втручання людини (наприклад, потрібно вставити компакт-диск).

Неавтономна інформація зберігається у файлах. Програми можуть одержати доступ до неї через програми вводу-виводу. Щоб стежити за інформацією, записаною неавтономно, групувати її в зручні блоки й захищати від незаконного використання, потрібні додаткові команди.

Звичайно операційна система групує неавтономні файли в директорії. На рис. 22 проілюстровано приклад такої організації. При виконання подібного угруповання ОС забезпечують низку системних викликів, принаймні, для таких функцій:

- створення файла й включення його до директорії;
- стирання файла з директорії;
- перейменування файла;
- зміна статусу захисту файла.

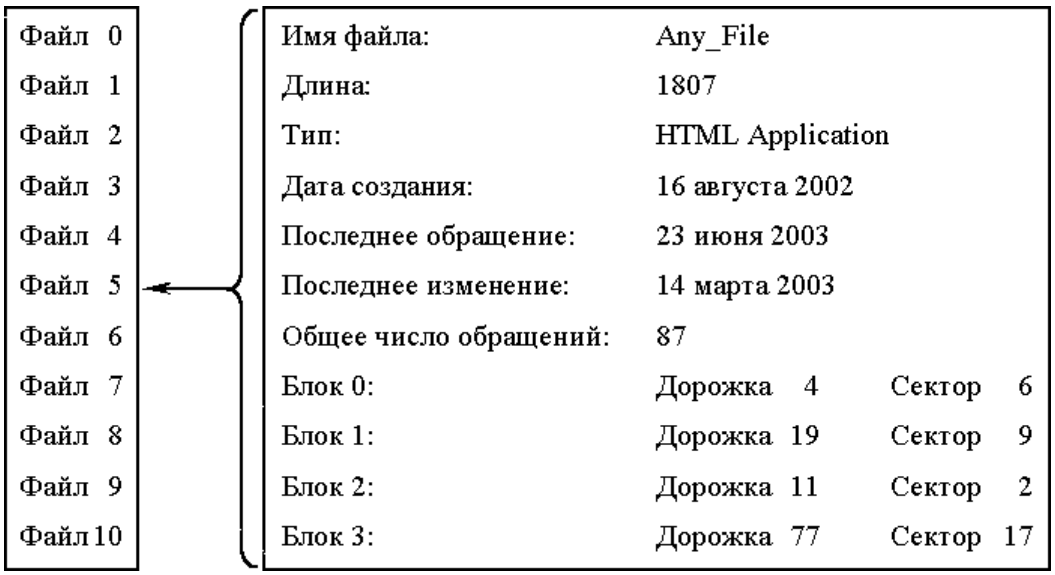

Рисунок 22 – Користувальницька директорія (а); можливий вміст одного з елементів директорії (б)

Застосовуються різні схеми захисту. Наприклад, власник файлів може ввести секретний пароль для кожного файла. При спробі доступу до файла програма повинна видавати пароль, що перевіряється операційною системою. Доступ до файла дозволяється тільки в тому випадку, якщо пароль правильний. Для кожного файла можна створити список людей, програми яких можуть одержувати доступ до такого файла.

Всі сучасні операційні системи дозволяють зберігати більше однієї директорії. Кожна директорія звичайно сама є файлом і як така може вставлятись в іншу директорію, у результаті чого можна одержати дерево директорій. Велика кількість директорій особливо потрібна програмістам, які працюють над декількома проектами. Вони можуть згрупувати в одну директорію всі файли, пов’язані з од-

ним проектом, а у другу – з іншим. Директорії – це зручний спосіб поділяти файли з членами своїх робочих груп.

2.5 Питання для самоперевірки

1. У деяких операційних системах надається системний виклик **`rename`**, що дозволяє змінити ім'я файла. Чи є різниця між використанням цього системного виклику й копіюванням файла з новим ім'ям з наступним видаленням старого файла?
2. Проста операційна система підтримує тільки один каталог, але дозволяє зберігати в ньому довільну кількість файлів з іменами довільної довжини. Чи можна на такій системі симулювати ієрархічну файлову систему? Як?
3. Як було вказано, робота з безперервними файлами приводить до фрагментації диска, тому що губиться деяка частина дискового простору в останніх блоках файлів, чия довжина не кратна цілому числу блоків. Ця фрагментація є внутрішньою або зовнішньою? Порівняйте з фрагментацією пам'яті.
4. Один зі способів використовувати безперервні файли на диску й не страждати від «дірок» полягає в ущільненні диска при кожному вилученому файлі. Оскільки всі файли є безперервними, для копіювання файла потрібен певний час на пошук циліндра й обертання диска при зчитуванні файла, після якого відбувається перенесення даних на повній швидкості. При записі файла на диск потрібні аналогічні операції. При часі пошуку циліндра, рівному 5 мс, затримці обертання в 4 мс, швидкості передачі даних 8 Міб/с і середньому розмірі файла 8 Кіб, скільки знадобиться часу для того, щоб прочитати файл в оперативну пам'ять, а потім записати його назад на нове місце на диску? При тих же параметрах, скільки буде потрібно часу для ущільнення половини 16-гібібайтного диска?
5. У світлі відповіді на попереднє питання, чи є взагалі смисл в ущільненні диска?
6. Облік вільного дискового простору може здійснюватися за допомогою

зв'язних списків або бітових масивів. Дискові адреси складаються з **D** біт. При якій умові для диска з **B** блоків, **F** з яких вільні, список займе менше місця, чим бітовий масив? Виразіть вашу відповідь у відсотках від обсягу диска для **D** = 16.

7. Зрівняєте застосування бітового відображення й списку невикористаних просторів для спостереження за вільним простором на диску. Диск складається з 800 циліндрів, на кожному з яких розташовані 5 доріжок по 32 сектори. Скільки знадобиться «дірок», щоб список «дірок» (список вільної пам'яті) став більшим, ніж бітове відображення? Передбачається, що одиничний блок – це сектор, і що для «дірки» потрібен 32-бітний елемент таблиці.
8. Після першого форматування дискового розділу початок бітового масиву обліку вільних блоків виглядає так: 1000 0000 0000 0000 (перший блок використовується для кореневого каталогу). Система завжди шукає вільні блоки від початку розділу, тому після запису файла A, що займає 6 блоків, бітовий масив набирає такого вигляду: 1111 1110 0000 0000. Покажіть, як буде виглядати бітовий масив після кожної з наступних дій: а) записується файл B розміром в 5 блоків; б) видаляється файл A; в) записується файл B розміром в 8 блоків; г) видаляється файл B.
9. Щоб зробити деякі прогнози щодо продуктивності диска, потрібно мати модель розподілу пам'яті. Припустимо, що диск розглядається як лінійний адресний простір з **N >> 1** секторів. Тут спочатку йде послідовність блоків даних, потім невикористаний простір, потім інша послідовність блоків даних і т.д. Емпіричні виміри показують, що ймовірнісні розподіли для довжин даних і невикористаних просторів однакові, причому для кожного з них імовірність бути **i** секторів становить 2^{-i}. Яке при цьому очікуване число «дірок» на диску?
10. На певній машині програма може створювати стільки файлів, скільки їй потрібно, і всі файли можуть збільшуватися в розмірах під час виконання про-

грами, причому операційна система не одержує ніяких додаткових даних про їхній кінцевий розмір. Як ви думаєте, чи зберігаються файли в послідовних секторах? Поясніть.

3 ВІРТУАЛЬНІ КОМАНДИ ДЛЯ ПАРАЛЕЛЬНОЇ ОБРОБКИ

Деякі обчислення зручно робити за допомогою двох і більше паралельних процесів (тобто начебто б на різних процесорах). Інші обчислення можна поділити на частини, які потім виконуються паралельно, що скорочує загальний час обчислення. Щоб кілька процесів могли відбуватися паралельно, потрібні спеціальні віртуальні команди. Такі команди є темою обговорювання в наступних курсах і будуть розглянуті пізніше.

Інтерес до паралельної обробки зумовлюється деякими фундаментальними законами фізики. Згідно теорії відносності Ейнштейна, швидкість передачі електричних сигналів не може перевищувати швидкість світла, яка дорівнює приблизно 30 см/нс у вакуумі, а в мідному провіднику або оптичному волокні – ще менше. Важливо враховувати цю межу при розробці комп'ютерів. Наприклад, якщо процесору потрібні дані з основної пам'яті, блоки якої перебувають на відстані 30 см від нього, то буде потрібно принаймні 1 нс, щоб запит дійшов до пам'яті, і ще 1 нс, щоб відповідь повернулася до центрального процесора. Отже, щоб комп'ютери могли передавати сигнали швидше, вони (комп'ютери) повинні бути зовсім малюсінькими. Альтернативний спосіб збільшення швидкодії комп'ютера – створення машини з декількома процесорами. Комп'ютер, що містить 1000 процесорів із часом циклу в 1 нс, буде мати приблизно таку ж потужність, як процесор із часом циклу 0,001 нс, але перше здійснити набагато простіше й дешевше.

У комп'ютері з декількома процесорами кожен з декількох взаємодіючих процесів можна приписати до одного певного процесора, щоб виконувати кілька дій одночасно. Якщо в комп'ютері є тільки один процесор, ефект паралельної обробки вже моделюється засобами операційної системи. При цьому процеси виконуються по черзі один за одним, кожен у наперед визначений невеликій період часу. Іншими словами, процесор в таких системах поділяється між декількома процесами.

На рис. 23 схематично показана різниця між реальною паралельною обробкою, коли в системі присутні кілька фізичних процесорів, і змодельованою пара-

лельною обробкою, коли є всього один фізичний процесор. Навіть якщо паралельна обробка моделюється, зручно вважати, що кожному процесу приписується його власний віртуальний процесор. При модельованій паралельній обробці виникають ті ж проблеми, що й при реальній паралельній обробці.

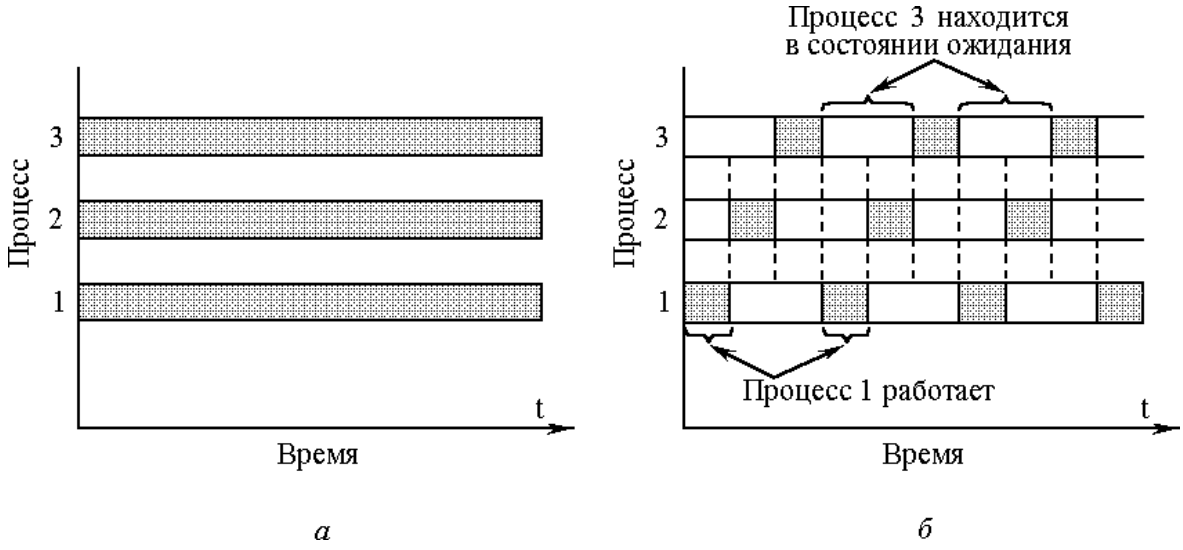

Рисунок 23 – Паралельна обробка з декількома процесорами (а);
моделювання паралельної обробки шляхом перемикання одного процесора
з одного процесу до іншого (в даному разі лише три процеси) (б)

3.1 Формування процесу

Кожна програма в системі повинна працювати як частина якого-небудь процесу. Цей процес, як і всі інші процеси, характеризується станом і адресним простором, через звернення до якого можна одержати доступ до програм і даних. Стан процесу як мінімум містить:

- лічильник команд;
- слово стану програми;
- покажчик стека;
- регістри загального призначення.

Більшість сучасних операційних систем дозволяють формувати й переривати процеси динамічно. Для формування нового процесу потрібен системний виклик. Цей системний виклик може просто створити клон програми, яка звернулась до нього або дозволити вихідному процесу зумовити початковий стан нового процесу, тобто його програму, дані й початкову адресу.

Є дві стратегії створення породжених процесів. В одних випадках вихідний процес зберігає частковий або навіть повний контроль над породженим процесом. Віртуальні команди дозволяють вихідному процесу зупиняти й знову запускати, перевіряти й завершувати підлеглі процеси. В інших випадках вихідний процес ніяк не контролює породжений процес: після того як новий процес сформований, вихідний процес не може його зупинити, запустити заново, перевірити або завершити. Таким чином, ці два процеси працюють незалежно один від одного.

3.2 Стан гонок

У багатьох випадках паралельні процеси повинні взаємодіяти, і їхню роботу потрібно синхронізувати. Спочатку є сенс розглянути синхронізацію процесів і деякі труднощі, пов’язані з нею. Способи вирішення цих труднощів будуть розглянуті трохи пізніше.

Нехай у системі працюють два незалежних процеси, процес 1 і процес 2, які взаємодіють через загальний буфер в основній пам’яті. Для простоти можна назвати процес 1 виробником (producer), а процес 2 – споживачем (consumer). Виробник генерує прості числа й поміщає їх у буфер по одному. Споживач видаляє їх з буфера по одному й друкує.

Ці два процеси працюють паралельно, але з різною швидкістю. Якщо виробник виявляє, що буфер заповнений, він повинен перейти у режим очікування, тобто тимчасово припинити генерацію нових чисел й очікувати сигналу від споживача. Коли споживач видаляє число з буфера, він посилає сигнал виробникові, щоб той відновив роботу. Якщо споживач виявить, що буфер порожній, то тепер вже він повинен припинити роботу. Коли виробник поміщає число в порожній буфер, він посилає відповідний сигнал споживачеві.

Цілком зрозуміло, що буфер не може бути безрозмірним, тому зазвичаєм, такі буфери організують і використовують як кільцеву структуру. При цьому з буфером пов'язуються два покажчики *in* і *out*, які використовуються в такий спосіб: *in* указує на наступне вільне слово (куди виробник помістить наступне просте число), а *out* указує на наступне число, яке повинен видалити споживач.

Якщо буфер порожній, то покажчики дорівнюють один до одного *in=out*, як це показано на рис. 24 *а*. На рис. 24 *б* показана ситуація після того, як виробник породив кілька простих чисел. На рис. 24 *в* зображений буфер після того, як споживач видалив з нього кілька простих чисел для друку. На рис. 24 *г – е* представлені проміжні стадії роботи буфера. Як вже вказувалось, буфер заповнюється по колу. Тобто коли покажчик *in* дійде до верхньої межі буфера, його значення знов встановлюється на початок буфера.

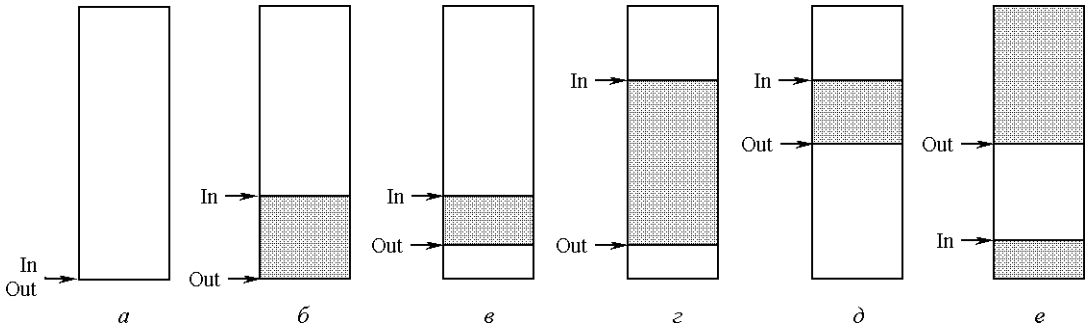

Рисунок 24 – Пояснення роботи кільцевого буфера

Оскільки виробник працює швидше за споживача, то з плином часу може статися ситуація, коли буфер повністю заповниться. Це станеться тоді коли значення показника *in* буде на одиницю менше за *out* (наприклад, *in*=97, а *out*=98), тобто під *out* є тільки одне вільне слово. Останнє слово не може використовуватись; якби воно використовувалося, то не було б можливості повідомити, що саме значить рівність *in=out* – повний буфер або порожній буфер.

Вирішення завдання з роботи виробника і споживача можна описати таким кодом мовою Java (рис. 25).

```
public class Main {
  final public static int BUF_SIZE  = 100;       //буфер від 0 до 99
  final  public static long MAX_PRIME = 100000000000L; //зупинитися
                                                        //тут
  public static int in = 0, out  = 0;            //покажчики на дані
  public static long buffer[] = new  long[BUF_SIZE];
  public static producer p;                        //ім'я виробника
  public static  consumer c;                       //ім'я споживача
  public static void main(String args[]){
    p = new producer();                          //створення  виробника
    c = new consumer();                          //створення  споживача
    p.run();                                     //запуск  виробника
    c.run();                                     //запуск  споживача
  }
//Це метод для циклічного змінення in і out
  public static int next(int k)
    {if (k < BUF_SIZE - 1) return(k+1); else return 0;}
}

class producer extends  Thread {                   //клас виробника
  public void  run()  {                            //код виробника
    long prime = 2;                                //тимчасова змінна
    while (prime < Main.MAX_PRIME) {
      prime = next_prime(prime);                                 //крок  P1
      if  (Main.next(Main.in)  ==  Main.out)  suspend();  //крок  P2
      Main.buffer[Main.in] = prime;                              //крок  P3
      Main.in = Main.next(Main.in);                              //крок  P4
      if(Main.next(Main.out)==Main.in) Main.c.resume();   //крок  P5
      }
    }
  private long  next_prime(long prime){...}   //функція, що обчислює
                                              //наступне число
}
class consumer extends  Thread {                  //клас споживача
  public void  run()  {                           // код споживача
    long emirp = 2;                               // тимчасова змінна
    while (emirp < Main.MAX_PRIME) {
      if (Main.in ==  Main.out) suspend();                       //крок C1
      emirp = Main.buffer[Main.out];                             //крок C2
      Main.out = Main.next(Main.out);                            //крок C3
      if (Main.out == Main.next(Main.next(Main.in)))
                                    Main.p.resume();     //крок C4
      System.out.println(emirp);                                 //крок C5
    }
  }
}
```

Рисунок 25 – Код паралельної обробки зі станом гонок

Тут використовуються три класи: *Main, producer* і *consumer*. Клас *Main* містить деякі константи, покажчики буфера *in* і *out* і сам буфер, котрий у цьому при-

кладі вміщає 100 простих чисел (від *buffer*[0] до *buffer*[99]).

Для моделювання паралельних процесів у цьому випадку використовуються потоки (*threads*). У прикладі є клас *producer* і клас *consumer*, яким приписуються значення змінних *p* і *c* відповідно. Кожен з цих класів успадковується від базового класу *Thread* і перевизначає процедуру *run*, котра містить код для *thread*. Коли викликається метод *run* для об'єкта, успадкованого від *Thread*, запускається новий потік.

Кожен потік схожий на процес. Єдиною різницею є те, що всі потоки в межах однієї програми мовою Java працюють в одному адресному просторі. Це дозволяє їм розділяти один загальний буфер. Якщо в комп'ютері є два й більше процесорів, кожен потік може виконуватися на іншому процесорі, тому в цьому випадку має місце реальний паралелізм. Якщо комп'ютер містить тільки один процесор, потоки розділяються в часі на одному процесорі. Мова Java підтримує тільки паралельні потоки, а не реальні паралельні процеси, але в прикладі, що розглядається, можна продовжувати називати виробника й споживача процесами, оскільки в цьому випадку різниця між процесами і потоками не є принциповою.

Метод *next* дозволяє збільшувати значення *in* і *out* з виконанням перевірки умови циклічного повернення. Якщо параметр в *next* дорівнює 98 або вказує на більше низьке значення, то вертається наступне одне по одному ціле число. Якщо параметр дорівнює 99, це значить, що виробник або споживач досягнув кінця буфера, тому метод вертає 0.

Для здійснення синхронної роботи процесів, повинен бути спосіб «присипляти» кожен з процесів у випадку, коли він не може продовжувати свою роботу. Для цього розроблювачі Java включили до класу *Thread* спеціальні методи керування потоками – *suspend* (відключення) і *resume* (поновлення). Їхнє використання видно у коді з рис. 25.

Як же працюють і як взаємодіють один з одним виробник і споживач у цій програмі? Спочатку виробник на кроці P1 породжує нове просте число. Є цілком очевидним, що префікс *Main* у виразі *Main.MAX_PRIME* указує на той факт, що

мається на увазі параметр *MAX_PRIME*, який є визначеним у класі *Main.* З тієї ж причини цей префікс потрібен для *in*, *out*, *buffer* і *next.*

Потім на кроці P2 виробник перевіряє, чи не перебуває покажчик *in* нижче за *out*. І якщо так (наприклад, *in*=62 і *out*=63), то буфер заповнений і виробник на цьому кроці викликає метод *suspend*. Якщо буфер не заповнений, то на кроці P3 у буфер уміщується нове просте число і значення *in* збільшується на одиницю – крок P4. Якщо нове значення *in* на 1 більше значення *out* (крок P5) (наприклад, *in*=17, *out*=16), це означає, що *in* і *out* були рівні перед тим, як збільшилося значення *in*. В такий ситуації виробник робить висновок, що буфер був порожній і що споживач не функціонував (перебував у режимі очікування) і, звісно, що в цей момент теж не функціонує. Тому виробник викликає метод *resume*, щоб відновити роботу споживача (крок P5). Нарешті, виробник починає генерувати наступне просте число.

Програма споживача за структурою дуже проста. Спочатку робиться перевірка (крок C1), щоб довідатися, чи порожній буфер. Якщо він порожній, то споживачеві нічого не потрібно робити, тому він відключається. Якщо буфер не порожній, то споживач на кроці C2 видаляє з буфера наступне число для друку і збільшує значення *out*. Якщо після цього покажчик *out* став на дві позиції вище за *in*, то виходить, що перед цим *out* був на одну позицію вище за *in*. А тому що рівність *in*=*out*-1 – це умова заповнювання буфера, то споживач робить висновок, що виробник не працює й він (споживач) повинен викликати метод *resume*. Після цього число виводиться на друк, і весь цикл повторюється знову.

На жаль, така програма містить помилку, наявність якої пояснюється за допомогою рис. 26. Тут слід пам’ятати, що ці два процеси працюють асинхронно й з різними швидкостями, які, до того ж, можуть змінюватися. У такій системі цілком може статися випадок, коли в буфері залишилося тільки одне число в елементі 21, і *in*=22, а *out*=21 (див. рис. 26 *а*). Виробник на кроці P1 шукає просте число, а споживач на кроці C5 друкує число з позиції 20. Споживач закінчує друкувати число, робить перевірку на кроці C1 і забирає останнє число з буфера на кроці C2. Потім він збільшує значення *out*. У цей момент і *in* і *out* рівні 22. Споживач друкує чис-

ло, а потім переходить до кроку C1, на якому він викликає *in* і *out* з пам'яті, щоб зрівняти їх (рис. 26 *б*).

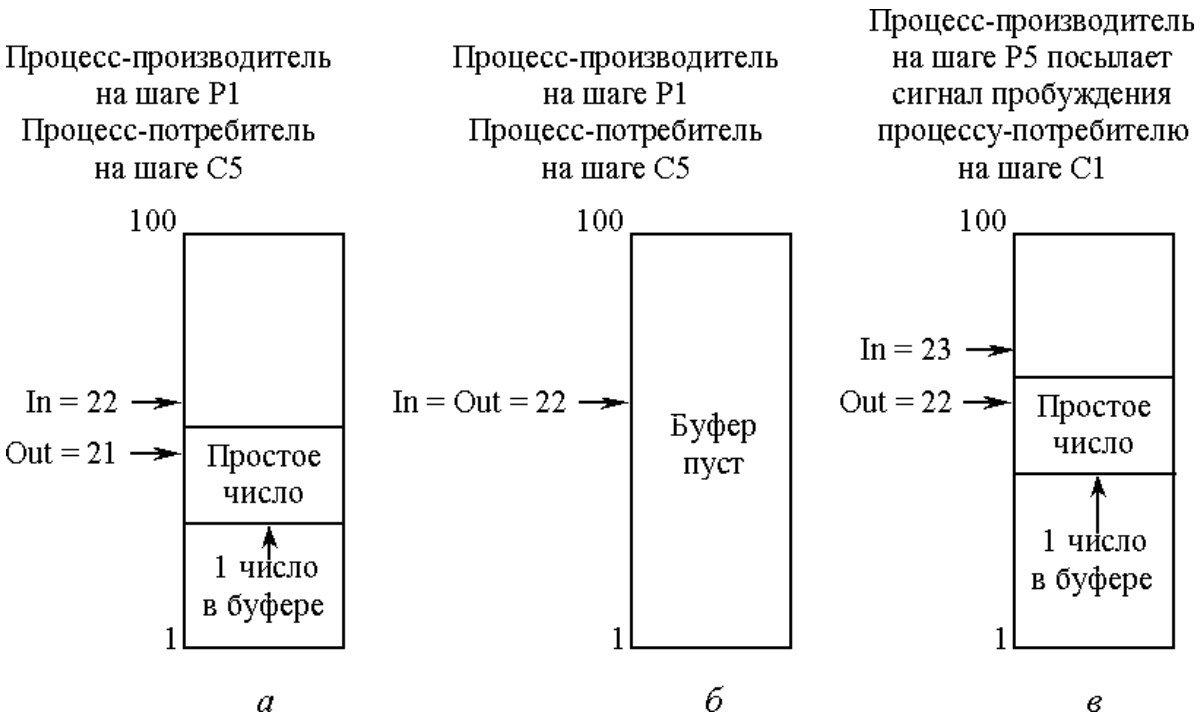

Рисунок 26 – Ситуація, при якій механізм взаємодії виробника и споживача не працює

У цей момент, після того як споживач викликав *in* і *out*, але ще трохи раніш ніж він зрівняв їх, виробник знаходить наступне просте число. Він поміщає це просте число в буфер на кроці P3 і збільшує *in* на кроці P4. Тепер *in*=23, а *out*=22. На кроці P5 виробник виявляє, що *in*=*next*(*out*). Іншими словами, *in* на одиницю більше *out*, а це значить, що в буфері в цей момент перебуває один елемент.

Виходячи із цього, виробник робить невірний висновок, що споживач відключений, і викликає процедуру *resume* (рис. 26 *в*). Насправді споживач весь цей час продовжував працювати, тому виклик процедури *resume* виявився помилковим. Потім виробник починає шукати наступне просте число.

У цей момент споживач продовжує працювати. Він уже викликав *in* і *out* з пам'яті, перед тим як виробник помістив останнє число в буфер. Тому що *in*=22 і

out=22 (буфер пустий), споживач відключається. До цього моменту виробник знаходить наступне просте число. Він перевіряє покажчики й виявляє, що *in*=24, а *out*=22. Із цього він робить висновок, що в буфері перебуває 2 числа (що відповідає дійсності) і що споживач функціонує (що невірно).

Виробник продовжує цикл. Зрештою, він заповнює буфер і відключається. Тепер обидва процеси відключені й будуть перебувати в такому стані до кінця століть.

Складність тут у тім, що між моментом, коли споживач викликає *in* і *out*, і моментом, коли він відключається, виробник, виявивши, що *in*=*out*+1, і припустивши, що споживач відключений (хоча насправді він ще продовжує функціонувати), викликає процедуру *resume*, чого не потрібно робити, оскільки споживач функціонує. Така ситуація називається станом гонок, оскільки успіх процедури залежить від того, хто виграє гонку по перевірці *in* і *out*, після того як значення *out* збільшилося.

Проблема стану гонок добре відома. Вона була настільки серйозна, що через кілька років після появи Java компанія Sun змінила клас *Thread* і забрала виклики процедур *suspend* і *resume*, оскільки вони дуже часто приводили до стану гонок. Запропоноване рішення було засновано на зміні мови, але оскільки у курсі вивчаються операційні системи, то потрібно обговорити інше рішення, що використовується в багатьох операційних системах, у тому числі UNIX і Windows.

3.3 Синхронізація процесів з використанням семафорів

Проблему стану гонок можна вирішити, принаймні, двома способами. Перший спосіб – забезпечивши кожен процес спеціальним ***бітом очікування пробудження***. Якщо процес, що функціонує в цей момент, одержує сигнал «пробудження», то цей біт встановлюється у одиничній стан. Якщо процес відключається в той момент, коли цей біт установлений, він негайно перезапускається, а біт скидається. Цей біт служить для збереження сигналу пробудження з метою його майбутнього використання.

Цей метод вирішує проблему стану гонок тільки в тому випадку, якщо в нас є лише 2 процеси. У загальному випадку при наявності *n* процесів він не працює. Звичайно, кожному процесу можна приписати *n*-1 таких бітів очікування пробудження, але це незручно.

Один з провідних розробників операційної системи UNIX – Дийкстра запропонував інше вирішення цієї проблеми. Своє рішення він запозичив з досвіду функціонування залізниць. Десь у пам'яті перебувають дві змінні, які можуть містити ненегативні цілі числа. Ці змінні називаються ***семафорами***. Операційна система надає два системних виклики, *up* і *down*, які оперують семафорами. *Up* додає 1 до семафора, а *down* віднімає 1 від семафора. Якщо операція *down* відбувається над семафором, значення якого більше 0, цей семафор зменшується на 1 і процес триває. Якщо значення семафора дорівнює 0, то операція *down* не може завершитися. Тоді даний процес відключається доти, поки інший процес не виконає операцію *up* над цим семафором.

Системний виклік *up* перевіряє, чи не дорівнює семафор нулю. Якщо він дорівнює 0 і інший процес перебуває в режимі очікування, то семафор збільшується на 1. Після цього процес, що «спить», може завершити операцію *down*, установивши семафор на 0. Тепер обидва процеси можуть продовжувати роботу. Якщо семафор не дорівнює 0, команда *up* просто збільшує його на 1. За результатом такої роботи, семафор дозволяє зберігати сигнали пробудження, так що вони не пропадуть даремно. У семафорних команд є одна важлива властивість: якщо один із процесів почав виконувати команду над семафором, то інший процес не може одержати доступ до цього семафора доти, поки перший не завершить виконання команди або не буде припинений при спробі виконати команду *down* над 0. У табл. 2 викладені важливі властивості системних викликів *up* і *down*.

Як вже було сказане вище, у мові Java передбачене своє вирішення проблеми стану гонок, але оскільки зараз обговорюються операційні системи, потрібно яким-небудь чином виразити використання семафорів у мові Java. Можна допустити, що були написані два *native*-метода, *up* і *down*, які роблять системні виклики

up і *down* відповідно. Використовуючи як параметри звичайні цілі числа, можна виразити застосування семафорів у програмах мовою Java.

Таблиця 2 – Результати операцій над семафором

Команда	Семафор = 0	Семафор > 0
Up	Семафор = семафор + 1; якщо інший процес намагається зробити команду down над цим семафором, тепер він зможе це зробити й продовжити свою роботу	Семафор = семафор +1
Down	Процес зупиняється доти, поки інший процес не виконає операцію up над цим семафором	Семафор = семафор - 1

В коді на рис. 27 показано, як можна усунути стан гонок за допомогою семафорів. До класу *Main* додаються два семафори: *available*, котрий ініціюється значенням 100 (це розмір буфера), і *filled*, котрий спочатку дорівнює 0. Виробник починає роботу з кроку P1, а споживач – з кроку C1. Виконання процедури *down* над семафором *filled* відразу ж припиняє роботу споживача. Коли виробник знаходить перше просте число, він викликає процедуру *down* з *available* як параметр, установлюючи *available* на 99. На кроці P5 він викликає процедуру *up* з параметром *filled*, установлюючи *filled* на 1. Ця дія звільняє споживача, що тепер може завершити виклик процедури *down*. Після цього *filled* приймає значення 0, і обидва процеси продовжують роботу.

А тепер має сенс, як і раніш, розглянути стан гонок. У певний момент *in*=22, а *out*=21, виробник перебуває на кроці P1, а споживач – на кроці C5. Споживач завершує дію й переходить до кроку C1, що викликає процедуру *down*, щоб виконати її над семафором *filled*, котрий до виклику мав значення 1, а після виклику набув значення 0. Потім споживач бере останнє число з буфера й виконує процедуру *up* над *available*, після чого *available* набуває значення 100. Споживач друкує узяте число й переходить до кроку C1.

```
public class Main {
  final public static int BUF_SIZE  = 100;       //буфер від 0 до 99
  final  public static long MAX_PRIME = 100000000000L; //зупинитися
                                                        //тут
  public static int in = 0, out  = 0;            //покажчики на дані
  public static long buffer[] = new  long[BUF_SIZE];
  public static  producer p;                          //ім'я виробника
  public static  consumer c;                          //ім'я споживача
  public static int filled = 0, available  = 100;  //семафори
  public static void  main(String args[]){     //основний клас
    p = new producer();                         //створення виробника
    c = new consumer();                         //створення споживача
    p.run();                                    //запуск виробника
    c.run();                                    //запуск споживача
  }
//Це утиліта для циклічного збільшення in і out
  public static int next(int k)
    {if (k < BUF_SIZE - 1) return(k+1); else return(0);} }

class producer extends  Thread {                    //клас виробника
  native void up(int s); native void  down(int s);  //процедури над
                                                     //семафорами
  public void  run() {                             //код виробника
    long prime = 2;                                 //тимчасова змінна
    while (prime < Main.MAX_PRIME) {
      prime = next_prime(prime);                              //крок P1
      down(Main.available);                                   //крок P2
      Main.buffer[Main.in] = prime;                           //крок P3
      Main.in = Main.next(Main.in);                           //крок P4
      up(Main.filled);                                        //крок P5
      } }
  private long  next_prime(long prime){...}   //функція, що обчислює
                                              //наступне число
}

class consumer extends  Thread {               //клас споживача
  native void up(int s); native void  down(int s);  //процедури над
                                                     //семафорами
  public void  run() {                             // код споживача
    long emirp = 2;                                 // тимчасова змінна
    while (emirp < Main.MAX_PRIME) {
      down(Main.filled);                                      //крок C1
      emirp  = Main.buffer[Main.out];                         //крок C2
      Main.out = Main.next(Main.out);                         //крок C3
      up(Main.available);                                     //крок C4
      System.out.println(emirp);                              //крок C5
    }
  }
}
```

Рисунок 27 – Паралельна обробка з використанням семафорів

Саме перед тим, як споживач може викликати процедуру *down*, виробник знаходить наступне просте число й швидко виконує кроки P2, P3 і P4.

У цей момент *filled*=0. І виробник, і споживач збираються виконати над ним команду *up*. Якщо споживач виконає команду першим, то він перейде у припинений стан доти, поки виробник не звільнить його (викликавши процедуру *up*). Якщо ж першим на семафорі буде виробник, то семафор набере значення 1 і споживач взагалі не буде припинений. В обох випадках сигнал пробудження не пропаде. Саме для цього до програми й введені семафори.

Операції над семафорами неподільні. Якщо операція над семафором уже почалася, то ніякий інший процес не може використовувати цей семафор доти, поки перший процес не завершить операцію або поки він не буде припинений. Більше того, при наявності семафорів сигнали пробудження не пропадають. А от оператори *if* з рис. 25 є діленими. Між перевіркою умови й виконанням потрібної дії інший процес може послати сигнал пробудження.

По суті, проблема синхронізації була усунута шляхом введення неподільних системних переривань **up** і **down**. Щоб ці операції були неподільними, операційна система повинна заборонити двом і більше процесам використовувати один семафор одночасно. Якщо був зроблений системний виклик **up** або **down**, жоден інший код користувача не буде запущений, поки даний виклик не завершиться. Для цього звичайно під час виконання операцій над семафорами вводиться заборона на переривання.

Технологія з використанням семафорів працює для довільної кількості процесів. Кілька процесів можуть «спати», не завершивши системний виклик **down** на тому самому семафорі. Коли який-небудь інший процес виконає процедуру **up** на тім же семафорі, один з очікуючих процесів може завершити виклик **down** і продовжити роботу. Семафор зберігає значення 0, і інші процеси продовжують чекати.

У одній з класичних книжок з архітектури комп’ютера, для пояснення дії семафорів автор наводить такий приклад.

«Уявіть собі 20 баскетбольних команд. Вони грають 10 партій (процесів). Кожна гра відбувається на окремому полі. Є великій кошик (семафор) для баскетбольних м'ячів. На жаль, є тільки 7 м'ячів. У кожен момент у кошику перебуває від 0 до 7 м'ячів (семафор набуває значення від 0 до 7). Поміщенню м'яча до кошика – відповідає операція **up**, оскільки вона збільшує значення семафора. Добування м'яча з кошика – це, навпаки, операція **down**, оскільки вона зменшує значення семафора.

На самому початку один гравець від кожного поля посилається до кошика за м'ячем. Сімом з них вдається одержати м'яч (завершити операцію **down**); ті троє, що залишилися, змушені чекати м'яч. Їхні ігри тимчасово припинені. Зрештою, одна з партій закінчується, і м'яч вертається до кошика (виконується операція **up**). Ця операція дозволяє одному із трьох гравців, що залишилися, одержати м'яч (закінчити незавершену операцію **down**) і продовжити гру. Дві партії, що залишилися, залишаються припиненими доти, поки ще два м'ячі не повернуться в кошик. Коли ці два м'ячі покладуть у кошик (тобто, буде виконано ще дві операції **up**), можна буде продовжити останні дві партії.»

3.4 Питання для самоперевірки

1. Розглянемо один метод реалізації команд для роботи із семафорами. Щораз, коли центральний процесор збирається зробити команду **up** або **down** над семафором (семафор – це цілочисельна змінна в пам'яті), спочатку він установлює пріоритет центрального процесора таким чином, щоб блокувати всі переривання. Потім він викликає з пам'яті семафор, змінює його й відповідно до цього робить перехід. Після цього він знову знімає заборону з переривань. Чи буде цей метод працювати, якщо:
    а. Існує один центральний процесор, що перемикається між процесами кожні 100 мілісекунд?
    б. Два центральних процесори розділяють загальну пам'ять, у якій розташований семафор?

2. Компанія, що розробляє операційні системи, одержує скарги від своїх клієнтів із приводу останньої розробки, що підтримує операції із семафорами. Клієнти вирішили, що аморально з боку процесів припиняти свою роботу (тобто спати на роботі). Щоб догодити своїм клієнтам, компанія вирішила додати третю операцію, **peek**. Ця операція просто перевіряє семафор, але не змінює його й не блокує процес. Таким чином, програми спочатку перевіряють, чи можна робити над семафором операцію **down**. Чи буде ця ідея працювати, якщо семафор використовують три й більше процеси? А якщо два процеси?
3. Складіть таблицю, у якій у вигляді функції від часу від 0 до 1000 мілісекунд показано, які із трьох процесів PI, P2 і P3 працюють, а які блоковані. Всі три процеси виконують команди **up** і **down** над тим самим семафором. Якщо два процеси блоковані й відбувається команда **up**, то запускається процес із меншим номером, тобто P1 має перевагу над P2 і P3 і т.д. Спочатку всі три процеси працюють, а значення семафора дорівнює 1.

   При t=100 P1 робить **down**.

   При t=200 P1 робить **down**.

   При t=300 P2 робить **up**.

   При t=400 P3 робить **down**.

   При t=500 P1 робить **down**.

   При t=600 P2 робить **up**.

   При t=700 P2 робить **down**.

   При t=800 P1 робить **up**.

   При t=900 P1 робить **up**.
4. У системі бронювання квитків на авіарейси необхідно бути впевненим у тім, що поки один процес використовує файл, ніякий інший процес не може використовувати цей же файл. У іншому випадку два різних процеси, які працюють на два різних агентства із продажу квитків, можуть продати останнє місце, що залишилося, двом пасажирам. Розробіть метод синхронізації з ви-

користанням семафорів, щоб точно знати, що тільки один процес у конкретний момент часу може одержувати доступ до файла (передбачається, що процеси підлягають правилам).

5. Щоб уможливити реалізацію семафорів на комп'ютері з декількома процесорами, які розділяють загальну пам'ять, розроблювачі включають у машину команду для перевірки й блокування. Команда TSL X перевіряє комірку X. Якщо її зміст дорівнює 0, семафори встановлюються на 1 за один неподільний цикл пам'яті, а наступна команда пропускається. Якщо зміст комірки не дорівнює 0, TSL працює як порожня операція. Використовуючи TSL, можна написати процедури **lock** і **unlock** з такими властивостями: **lock(x)** перевіряє, чи замкнений **x**. Якщо ні, ця процедура замикає **x** і повертає керування; **unlock** скасовує існуюче блокування. Якщо **x** уже замкнений, процедура просто чекає, поки він не звільниться, і тільки після цього замикає **x** і повертає керування. Якщо всі процеси замикають таблицю семафорів перед її використанням, то в певний момент часу тільки один процес може робити операції зі змінними й покажчиками, що запобігає стану гонок. Напишіть процедури **lock** і **unlock** на асемблері.
6. Яке буде значення **in** і **out** для кільцевого буфера довжиною в 65 слів після кожної з наступних операцій? Спочатку значення **in** і **out** дорівнюють 0.

    а. 22 слова містяться в буфер;

    б. 9 слів видаляються з буфера;

    в. 40 слів містяться в буфер;

    г. 17 слів видаляються з буфера;

    д. 12 слів містяться в буфер;

    е. 45 слів видаляються з буфера;

    ж. 8 слів містяться в буфер;

    и. 11 слів видаляються з буфера.

4 ПРИКЛАДИ ОПЕРАЦІЙНИХ СИСТЕМ

Після того як розглянутими є загальні принципи функціонування операційних систем, слід звернути увагу на те, як застосовуються ці принципи на прикладі двох операційних систем загального призначення Windows і UNIX. Розгляд мабуть варто почати з UNIX, оскільки ця система набагато простіша за Windows. Крім того, ОС UNIX була розроблена раніше й сильно вплинула на розвиток інших систем, в тому числі і Windows, тому такий порядок викладу буде більш осмисленим. Трохи зупинившись на історії створення і розвитку систем, особлива увага буде звернута на структуру ОС й системні виклики для роботи з віртуальною пам’яттю, файловою системою та процесами і потоками у системі.

4.1 Коротка історія розвітку ОС UNIX та її узагальнена структура

Операційна система UNIX була розроблена в компанії Bell Labs на початку 70-х років минулого сторіччя. Перша версія була написана Кеном Томпсоном (Ken Thompson) на асемблері для міні-комп’ютера PDP-7. Потім була написана друга версія для комп’ютера PDP-11, уже мовою С. Її автором був Денніс Рітчі (Dennis Ritchie). В 1974 році Рітчі й Томпсон опублікували дуже важливу роботу про систему UNIX. За цю роботу вони були нагороджені престижною премією Тьюрінга Асоціації обчислювальної техніки (Ritchie, 1984; Thompson, 1984). Після публікації цієї роботи багато університетів попросили в Bell Labs копію UNIX. Оскільки материнська компанія Bell Labs, AT&T була в той момент регульованою монополією і їй не дозволялося брати участь у комп’ютерному бізнесі, університети змогли придбати операційну систему UNIX за невелику плату.

PDP-11 використовувалися практично у всіх комп’ютерних наукових відділах університетів, і операційні системи, які прийшли туди разом з PDP-11, не подобалися ні професорам, ні студентам. UNIX швидко заповнив цю нішу. Ця операційна система постачалася з вихідними текстами, тому люди могли нескінченно виправляти її.

Одним з перших університетів, які придбали систему UNIX, був Каліфорнійський університет у Берклі. Оскільки була в наявності повна вихідна програма, у Берклі зуміли істотно перетворити цю систему. Серед змін було портірованіє цієї системи на міні-комп'ютер VAX, створення віртуальної пам'яті зі сторінковою організацією, розширення імен файлів з 14 символів до 255, а також включення мережного протоколу TCP/IP, що зараз використовується в Інтернеті (багато в чому завдяки тому факту, що він був у системі Berkeley UNIX).

Поки в Берклі робилися всі ці зміни, компанія AT&T самостійно продовжувала розробку UNIX, у результаті чого в 1982 році з'явилася System III, а в 1984 – System V. Наприкінці 80-х років широко використовувалися дві різні й зовсім несумісні версії UNIX – Berkeley UNIX і System V. Таке положення, та ще й відсутність стандартів на формати програм у двійковому коді сильно перешкоджало комерційному успіху системи UNIX. Постачальники програмного забезпечення не могли писати програми для UNIX, адже не було ніякої гарантії, що ці програми будуть працювати на будь-якій версії UNIX (як було зроблено з MS DOS, котра з початку розроблялась як UNIX-сумісна ОС). Після довгих суперечок комісія стандартів в Інституті інженерів з електротехніки й електроніки випустила стандарт POSIX (у розшифровці цієї абревіатури в літературних джерелах немає спільної думки тут є і Portable Operating System-IX – інтерфейс переносної операційної системи, і Portable Operating System Interface for Computer Environments – незалежний від платформи системний інтерфейс, і, навіть, Portable OS Interface based on uniX). POSIX також відомий як стандарт P1003. Пізніше він став міжнародним стандартом.

Стандарт POSIX поділено на кілька частин, кожна з яких покриває окрему область системи UNIX. Перша частина P1003.1 визначає системні виклики; друга частина P1003.2 визначає основні обслуговуючі програми й т.д. Стандарт P1003.1 визначає близько 60 системних викликів, які повинні підтримуватися всіма відповідними системами. Це виклики для читання й запису файлів, створення нових процесів і т.д. Зараз практично всі системи UNIX підтримують системні виклики P1003.1. Однак багато систем UNIX підтримують і додаткові системні виклики,

зокрема ті, які визначені в System V або в Berkeley UNIX. Іноді до набору POSIX додається до 100 системних викликів. Операційна система для машини UltraSPARC заснована на System V. Вона називається Solaris. Вона підтримує й багато викликів із системи Berkeley.

У табл. 3 наведені деякі категорії системних викликів. Системні виклики керування файлами й директоріями – це, мабуть, найбільші й найважливіші категорії у системі викликів. Більшість із них визначаються і відносяться до стандарту P1003.1. Інші походять із системи System V.

Таблиця 3 – Системні виклики UNIX

Категорія	Приклади
Керування файлами	Відкриття, читання, запис, закриття й блокування файлів
Керування директоріями	Створення й видалення директорій; переміщення файлів по директоріях
Керування процесами	Породження, завершення, відстеження процесів і передача сигналів
Керування пам’яттю	Поділ загальної пам’яті між процесами; захист сторінок
Виклик/установка параметрів	Ідентифікація користувача, групи, процесу; установка пріоритетів
Дати й періоди часу	Вказівка на час доступу до файлів; використання датчика часових інтервалів; робочий профіль програми
Робота в мережі	Установка/прийняття з’єднання; відправлення/одержання повідомлення
Інше	Облік використання ресурсів; обмеження на доступний обсяг пам’яті; перезавантаження системи

Сфера використання мереж більшою мірою визначається у Berkeley UNIX, а не у System V. У Берклі було уведено поняття ***сокет*** (кінцевий пункт мережного зв'язку). Чотирипровідні стінні розетки, до яких можна приєднувати телефони, послужили як модель цього поняття. Процес у системі UNIX може створити сокет, приєднатися до нього й встановити зв'язок із сокетом на віддаленому комп'ютері. По цьому зв'язку можна пересилати дані в обох напрямках, за звичай з використанням протоколу TCP/IP. Оскільки технологія мережного зв'язку десятиліттями застосовувалася в системі UNIX, значне число серверів в Інтернеті використовують саме UNIX.

Існує багато різних варіантів системи UNIX, і кожна з них чимсь відрізняється від всіх інших, тому структуру даної операційної системи описати важко. Але схема, котра зображена на рис. 28, є застосовною до більшості з них. ОС ґрунтується на апаратному забезпеченні комп'ютера і має декілька рівнів. На найнижчому рівні знаходяться ***драйвери пристроїв,*** які захищають систему файлів від апаратного забезпечення. Спочатку розробки системи, кожен драйвер пристрою писався окремо від всіх інших і був незалежною одиницею. Це призвело до численних дублювань, оскільки практично всі драйвери повинні мати справу з керуванням потоками, виправленням помилок, пріоритетами, відділенням даних від команд і т.ін. Із цієї причини Денніс Рітчі винайшов ***структуру за назвою потік*** для написання драйверів у модулях. При наявності потоку можна встановити двосторонній зв'язок між користувальницьким процесом і пристроєм апаратного забезпечення й вставити між ними один або кілька модулів. Користувальницький процес передає дані в потік, а потім ці дані обробляються або передаються далі кожним модулем доти, поки вони не дійдуть до апаратного забезпечення. При передачі даних від апаратного забезпечення відбувається зворотний процес.

На рис. 29 показана структура системи вводу-виводу, з усіма рівнями й основними функціями кожного рівня. Знизу вгору ці рівні являють собою апаратуру, оброблювачі переривань, незалежне від пристроїв програмне забезпечення й, нарешті, процеси користувача.

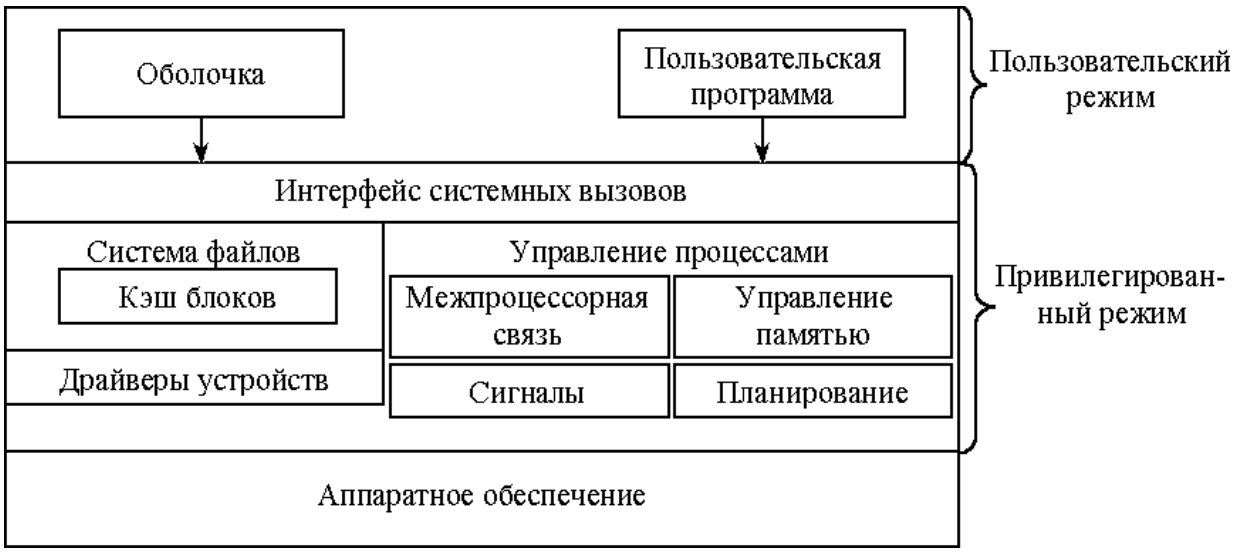

Рисунок 28 – Структура типової системи UNIX

Рисунок 29 – Рівні й основні функції системи вводу-виводу

Стрілки на рис. 29 зображують потік керування. Наприклад, коли програма користувача намагається прочитати блок з файла, для обробки виклику запускається операційна система. Незалежне від пристроїв програмне забезпечення шу-

кає цей блок у кеші. Якщо необхідного блоку там немає, воно викликає драйвер пристрою, щоб звернутися до апаратури й одержати цей блок з диска. При цьому процес блокується доти, поки не завершиться дискова операція. Коли диск завершує операцію, апаратура ініціює переривання. Оброблювач переривань запускається, щоб визначити, що трапилося, тобто який пристрій вимагає уваги. Потім він витягає статус пристрою й активізує «сплячий» процес, щоб завершити запит вводу-виводу й надати користувальницькому процесу можливість продовжуватися.

Над драйверами пристроїв перебуває ***система керування файлами***. Вона управляє іменами файлів, директоріями, розташуванням блоків на диску, захистом, керуванням віртуальною пам'яттю і виконує багато інших функцій. У системі файлів є так званий ***кеш*** блоків для зберігання недавно зчитаних з диска блоків, на випадок якщо вони знадобляться ще раз. Деякі системи файлів використовувалися протягом багатьох років. Серед них можна назвати швидку файлову систему Berkeley і файлові системи, які підтримують ***журналірування***.

Ще одна частина ядра системи UNIX – ***структура керування процесами***. Вона виконує різні функції, у тому числі управляє міжпроцесним зв'язком, що дозволяє різним процесам взаємодіяти один з одним і синхронізує роботу процесів, щоб уникнути стану гонок. Для цього існує ряд механізмів. Код керування процесами також управляє плануванням роботи процесів, цей механізм заснований на пріоритетах. Крім того, тут обробляються сигнали переривань. Нарешті, тут же сумісно з системою файлів відбувається керування пам'яттю. Більшість систем UNIX підтримують віртуальну пам'ять із підкачуванням сторінок на вимогу, іноді з деякими додатковими особливостями (наприклад, кілька процесів можуть розділяти загальні області адресного простору).

Від створення, UNIX повинен був бути маленькою системою, щоб досягти збільшення надійності й високої продуктивності. Інтерфейс користувача у перших версіях UNIX був повністю текстовим й в системі використовувалися термінали, які могли відображати 24 або 25 рядків по 80 символів ASCII-кодів. Користувальницьким інтерфейсом управляла програма, так звана ***оболонка***, що надавала ін-

терфейс командного рядка. Оскільки оболонка не була частиною ядра, було легко додавати нові оболонки в UNIX, і з часом було придумано декілька надзвичайно складних оболонок.

Пізніше, коли з'явилися графічні термінали, у Масачусетскому технологічному інституті для UNIX була розроблена система XWindows. Ще пізніше повністю дороблений графічний інтерфейс користувача за назвою Motif був установлений поверх XWindows. Оскільки було потрібно зберегти маленьке ядро, практично весь код системи XWindows і Motif працює у користувальницькому режимі поза ядром.

4.2 Розвиток ОС Windows та її структура

Перша машина IBM PC, випущена в 1981 році, була оснащена 16-бітною операційною системою індивідуального користування, котра працювала в реальному режимі, з інтерфейсом командного рядка. Вона називалася MS-DOS 1.0. Ця операційна система складалася із програми, що перебувала в пам'яті і займала 8 КіБ. Через два роки з'явилася потужніша система на 24 КіБ – MS-DOS 2.0. Вона містила процесор командного рядка (оболонку) з рядом особливостей, запозичених із системи UNIX. В 1984 році компанія IBM випустила машину PC/AT з операційною системою MS-DOS 3.0, розмір якої на той момент становив вже 36 КіБ. З роками в системи MS-DOS з'являлися все нові й нові особливості, але вона при цьому залишалася системою з командним рядком.

Натхненна успіхом Apple Macintosh, компанія Microsoft вирішила створити графічний користувальницький інтерфейс, котрий був названий Windows. Перші три версії Windows, включаючи систему Windows 3.x, були не справжніми операційними системами, а графічними користувальницькими інтерфейсами на базі MS-DOS. Всі програми працювали в тому самому адресному просторі, і помилка в кожній з них могла призвести до зупинки всієї системи.

В 1995 році з'явилася система Windows 95, але це не усунуло MS-DOS, хоча MS-DOS уже являла собою нову версію 7.0. Windows 95 і MS-DOS 7.0 у сукупності містили в собі особливості розвиненої операційної системи, у тому числі вір-

туальну пам’ять, керування процесами й мультипрограмування. Однак операційна система Windows 95 не була повністю 32-бітною програмою. Вона містила великі частини старого 16-бітного коду й усе ще використовувала файлову систему MS-DOS з усіма обмеженнями. Єдиною зміною в системі файлів було додавання довгих імен файлів (раніше в MS-DOS довжина імен файлів була не більше за 8+3 символи).

Навіть у 1998 році при випуску Windows 98 система MS-DOS усе ще була присутня (цього разу версія 7.1) і включала 16-бітний код. Система Windows 98 не дуже відрізнялася від Windows 95, хоча частина функцій перейшла від MS-DOS до Windows, а формат дисків, що краще підходив для дисків більшого розміру, став стандартним. Основним розходженням був користувальницький інтерфейс, що об’єднав робочий стіл, Інтернет, телебачення й зробив систему більш закритою. Саме це й привернуло увагу судового департаменту США, що тоді подав на компанію Microsoft у суд, обвинувативши її в незаконному монополізмі.

Під час всіх цих перетворень компанія Microsoft розробляла зовсім нову 32-бітну операційну систему, що була написана заново з нуля. Ця нова система називалася Windows New Technology (нова технологія) або Windows NT. При розробці передбачалося, що система замінить всі інші операційні системи для комп’ютерів на базі процесорів Intel, але вона дуже повільно поширювалася й пізніше була переорієнтована на більш коштовні комп’ютери. Поступово вона стала користуватися популярністю й в інших колах.

ОС сімейства Windows NT, а це всі системи починаючи з NT 4 й до останніх таких як Windows 2012, випускаються у двох варіантах: для серверів і для робочих станцій. Ці дві версії практично ідентичні й вироблені з одного вихідного коду. Перша версія призначена для локальних файлових серверів і серверів друку й має більш складні особливості керування, чим версія для робочих станцій, що призначена для настільних обчислень одного користувача. Існують особливі варіанти версій для серверів, призначених для великих сайтів. Різні версії настроюються по-різному, і кожна з них є оптимізованою для очікуваного оточення. У всьому іншому такі версії подібні. Практично всі виконувані файли ідентичні для

всіх версій. Система Windows сама визначає свою версію за спеціальною змінною у внутрішній структурі даних (системний реєстр). Користувачам заборонено змінювати цю змінну й у такий спосіб перетворювати дешеву версію для робочої станції на більш коштовну версію для сервера або у версію для підприємства. Усі ці розходження не є досить принциповими, тому при подальшому розгляданні, особливої уваги на них звертатися не буде.

MS-DOS і всі попередні версії Windows були розраховані на одного користувача. Зараз Windows підтримує мультипрограмування, тому на одній і тій же машині в той саме час можуть працювати кілька користувачів. Тут слід відмітити, що, на відміну від UNIX, Windows не дозволяє декільком користувачам одночасно працювати з комп'ютером, оскільки це однотермінальна система, тоді як UNIX – це мультитермінальна операційна система. Однак у комп'ютерній мережі з нею можуть одночасно взаємодіяти декілька користувачів, що працюють на своїх власних комп'ютерах. Наприклад, мережний сервер дозволяє декільком користувачам входити в систему по мережі одночасно, причому кожен з них одержує доступ до своїх власних файлів.

Системи сімейства NT являють собою реальні 32-бітні операційні системи з мультипрограмуванням. Вони підтримують кілька користувальницьких процесів, кожен з яких має у своєму розпорядженні повний 32-бітний віртуальний адресний простір з підкачуванням сторінок на вимогу. Крім того, самі системи написані як 32-бітний, а зараз вже і 64-бітний код.

ОС NT, на відміну від Windows 95, мають модульну структуру. Вона складається з невеликого ядра, що працює в привілейованому режимі, і декількох обслуговуючих процесів, що працюють у користувальницькому режимі. Користувальницькі процеси взаємодіють із обслуговуючими процесами із застосуванням моделі клієнт-сервер. Клієнт надсилає запит серверу, а сервер виконує роботу й відправляє результат клієнтові. Модульна структура дозволяє переносити системи сімейства NT на деякі комп'ютери не з сімейства Intel (DEC Alpha, IBM Power PC і SGI MIPS). Однак з міркувань підвищення продуктивності більша частина системи була перенесена назад у ядро.

Структура ОС Windows показана на рис. 30. Вона складається з ряду модулів, розташованих по рівнях. Їхня спільна робота реалізує операційну систему. Кожен модуль виконує певну функцію й має певний інтерфейс із іншими модулями. Практично всі модулі написані мовою С, хоча частина графічного інтерфейсу написана на С++, а дещо із самих нижніх рівнів – на асемблері.

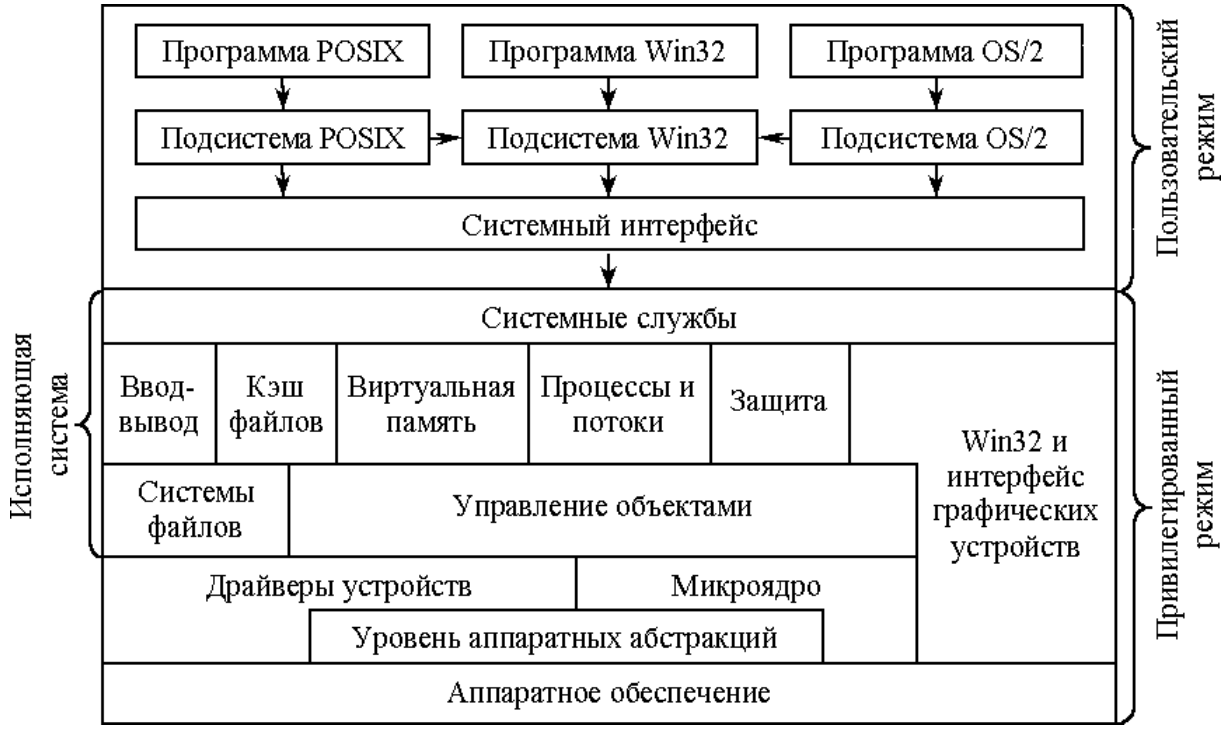

Рисунок 30 – Структура ОС Windows

Як і має бути, у самому низу системи розташоване апаратне забезпечення, а безпосередньо над ним міститься ***рівень апаратних абстракцій***. Він повинен постачати операційну систему абстрактними апаратними пристроями, позбавленими всіх недоліків, яких у реального апаратного забезпечення в надлишку. До пристроїв, що моделюються відносяться: кеш-пам’ять поза кристалом, тактові генератори, шини вводу-виводу, контролери переривань і контролери прямого доступу до пам’яті. Якщо ці пристрої представити перед операційною системою в ідеалізованому виді, то це спростить перенос Windows на інші апаратні платформи, оскільки більша частина необхідних змін концентрується в одному місці, як раз у

царині доступу до апаратного забезпечення комп'ютера.

Над рівнем апаратних абстракцій розташований рівень, що містить ***мікроядро*** й ***драйвери пристроїв***. Мікроядро й всі драйвери пристроїв окрім рівня апаратних абстракцій мають й прямий доступ до апаратного забезпечення, оскільки вони містять залежний від апаратного забезпечення код.

Мікроядро підтримує примітивні об'єкти ядра, обробку переривань, пасток, виключень, синхронізацію процесів, синхронізацію роботи процесорів у багатопроцесорних системах і керування часом. Основне завдання цього рівня – зробити іншу частину операційної системи повністю незалежною від апаратного забезпечення й, отже, високомобильною. Мікроядро постійно перебуває в основній пам'яті й нікуди не передається, хоча воно тимчасово може передати керування перериванням вводу-виводу.

Кожен драйвер пристроїв може управляти одним або декількома пристроями вводу-виводу. Крім того, драйвер пристроїв може виконувати якісь функції, не пов'язані з конкретним пристроєм, наприклад шифровку потоку даних або навіть забезпечення доступу до структур даних ядра. З причини того, що користувачі мають можливість установлювати нові драйвери пристроїв, вони можуть вплинути на ядро й зіпсувати всю систему, тому й драйвери потрібно писати з особливою обережністю.

Над мікроядром і драйверами пристроїв перебуває ***виконуюча система***. Виконуюча система – незалежна архітектура, тому її можна переносити на інші машини. Вона складається із трьох рівнів.

Найнижчий рівень містить ***файлові системи*** й ***диспетчер об'єктів***. Файлові системи управляють використанням файлів і директорій. Диспетчер об'єктів управляє об'єктами, відомими ядру (процесами, потоками, директоріями, семафорами, пристроями вводу-виводу, тактовими генераторами й т.ін.). Ця програма також управляє ***простором імен***, куди можна поміщати нові об'єкти, щоб звертатися до них пізніше якщо буде потреба.

Наступний рівень складається з шести основних частин, як показано на рис. 30. ***Диспетчер вводу-виводу*** забезпечує структуру для керування пристроями

вводу-виводу, а також загальними службами вводу-виводу. Диспетчер вводу-виводу використовує служби файлової системи, що, у свою чергу, використовує драйвери пристроїв, а також служби диспетчера об'єктів.

Диспетчер кеш-пам'яті зберігає в пам'яті блоки з диска, які недавно використовувалися, щоб підвищити швидкість доступу до них, якщо вони знадобляться знову. Диспетчер кеш-пам'яті повинен обчислювати, які блоки можуть знадобитися знову, а які – ні. Можна конфігурувати Windows з декількома системами файлів. У цьому випадку диспетчер кеш-пам'яті працює на всі системи файлів, тому окремий диспетчер для кожної з них не потрібний. Якщо є потреба одержати який-небудь блок з диска, диспетчерові кеш-пам'яті посилається сигнал видати цей блок. Якщо даного блоку у кеші немає, диспетчер викликає відповідну систему файлів, щоб одержати цей блок. Оскільки файли можуть бути відображені на адресні простори процесів, диспетчер кеш-пам'яті повинен взаємодіяти з модулем керування віртуальною пам'яттю, щоб забезпечити необхідну погодженість.

Модуль керування віртуальною пам'яттю реалізує архітектуру віртуальної пам'яті з підкачуванням сторінок на вимогу. Він управляє відображенням віртуальних сторінок на фізичні сторінкові кадри. Він також уводить додаткові правила захисту, які обмежують доступ кожного процесу тільки до тих сторінок, які належать його адресному простору. Крім цього він обробляє деякі системні виклики, які пов'язані з віртуальною пам'яттю,

Диспетчер процесів і потоків управляє процесами й потоками, у тому числі їхнім створенням і видаленням.

Диспетчер безпеки надає механізми безпеки Windows, які задовольняють вимогам Помаранчевої книги департаменту захисту США. У Помаранчевій книзі визначається величезна кількість правил, які повинна задовольняти система, починаючи з пароля й закінчуючи тим, що віртуальні сторінки повинні онулятися перед повторним використанням.

Інтерфейс графічних пристроїв управляє зображенням на моніторі й принтерами. Він забезпечує системні виклики, які дозволяють користувальниць-

ким програмам записувати інформацію на монітор або принтери незалежно від пристроїв. Він також містить ***диспетчер вікон*** і драйвери апаратних пристроїв. У версіях до NT 4.0 він перебував у користувальницькому просторі, але продуктивність при цьому залишала бажати кращого, тому компанія Microsoft перенесла його в ядро. Модуль Win32 також управляє багатьма системними викликами. Спочатку він теж розташовувався в користувальницькому просторі, але пізніше був переміщений у ядро з метою підвищення продуктивності.

Самий верхній рівень виконуючої системи – ***системні служби***. Цей рівень забезпечує інтерфейс із виконуючою системою. Він приймає системні виклики Windows і викликає інші частини виконуючої системи для виконання.

Поза ядром перебувають користувальницькі програми й підсистеми оточення. Необхідність підсистем оточення пояснюється тим, що користувальницькі програми не здатні безпосередньо здійснювати системні виклики. Тому кожна така підсистема експортує певний набір викликів функцій, які можуть використовувати користувальницькі програми. На рис. 30 показані 3 підсистеми оточення: Win32 (Windows), POSIX (для програм UNIX) і OS/2 для програм OS/2. Існує омана, що Windows може виконувати різні програми, розроблені для OS/2. Однак насправді ця підсистема NT дозволяє виконувати тільки ті деякі 16-бітні програми OS/2, які працюють винятково в текстовому режимі. 32-бітні програми OS/2 система Windows виконувати не може.

Додатки Windows використовують функції підсистеми Win32 і взаємодіють із підсистемою Win32, щоб робити системні виклики. Підсистема Win32 приймає виклики функцій Win32 і використовує модуль системного інтерфейсу, щоб системні виклики Windows могли виконувати їх.

Підсистема POSIX повинна забезпечувати підтримку для додатків UNIX. Вона підтримує тільки стандарт P 1003.1. Це закрита підсистема. Її додатки не можуть використовувати можливості підсистеми Win32, що сильно обмежує її придатність. На практиці перенос будь-якої програми UNIX на NT з використанням цієї підсистеми майже неможливий. Її включили до Windows тільки тому, що уряд США зажадав, щоб операційні системи на комп’ютерах урядових організа-

ції відповідали стандарту P 1003.1. Ця підсистема не є самодостатньою, тому хоча для своєї роботи вона і використовує підсистему Win32, але при цьому не передає повний інтерфейс Win32 своїм користувальницьким програмам.

Функції підсистеми OS/2 теж обмежені. Вона також використовує підсистему Win32. Існує й підсистема MS DOS (вона не показана на рис.). Така обмеженість призвела до того, що сьогодні ці системи практично не підтримуються і поступово виключаються зі структури ОС Windows.

Якщо обговорювати систему служб, які пропонує операційна система Windows, то слід відзначити, що її інтерфейс – це основний засіб зв'язку програміста із системою. На жаль, компанія Microsoft не опублікувала повний список системних викликів Windows, крім того, вона змінює їх від випуску до випуску. При таких обставинах практично неможливе написання програм, які безпосередньо роблять системні виклики.

Зате компанія Microsoft визначила набір викликів Win32 API (Application Programming Interface – прикладний програмний інтерфейс). Це бібліотечні процедури, які або роблять системні виклики, щоб виконати певні дії, або в деяких випадках виконують окремі дії прямо в бібліотечній процедурі користувальницького простору або підсистеми Win32. Виклики Win32 API не змінюються при створенні нових версій.

Однак, крім цього, існують виклики Windows API, які можуть змінюватися в нових версіях. Тому що виклики Win32 API задокументовані й більш стабільні, автори практично всіх книг, присвячених програмуванню в ОС Windows, зосереджують свою увагу саме на них, а не на системних викликах.

У системах Win32 API і UNIX застосовуються зовсім різні підходи до системного програмування. В UNIX всі системні виклики загальновідомі й формують мінімальний інтерфейс: видалення хоча б одного з них змінить функціонування операційної системи. Підсистема Win32 навпаки забезпечує дуже повний інтерфейс. Тут часто ту саму дію можна виконати трьома або чотирма різними спосо-

бами. Крім того, Win32 містить у собі багато функцій, які не є системними викликами (наприклад, копіювання цілого файла).

Багато викликів Win32 API створюють об'єкти ядра того або іншого типу (файли, процеси, потоки, канали й т.ін.). Кожен виклик, що створює об'єкт ядра, повертає програмі, котра його викликала, результат, що називається ***ідентифікатором об'єкта*** (handle). Цей ідентифікатор згодом може використовуватися для виконання операцій над об'єктом. Для кожного процесу існує свій ідентифікатор. Він не може передаватися іншому процесу й використовуватися там (***дескриптори*** файла в UNIX теж не можна передавати іншому процесу). Однак при певних обставинах можна дублювати ідентифікатор, передати його іншим процесам і дозволити їм доступ до об'єктів, які належать іншим процесам. Кожен об'єкт має пов'язаний з ним ***дескриптор захисту***, що повідомляє, кому дозволено або заборонено робити ті або інші операції над об'єктом.

Операційну систему Windows іноді називають об'єктно-орієнтованою, оскільки оперувати з об'єктами ядра можна тільки за допомогою виклику процедур (функцій API) по їхніх ідентифікаторах. З іншого боку, вона не має таких основних властивостей об'єктно-орієнтованої системи, як спадкування і поліморфізм.

Інтерфейс Win32 API був наявним й у системі Windows 95/98, щоправда, з деякими виключеннями. Наприклад, Windows 95/98 не має захисту, тому ті виклики API, які пов'язані із захистом, просто повертають код помилки. Крім того, для імен файлів в зараз використовується набір символів Unicode, якого немає в Windows 95/98. Існують розходження в параметрах для деяких викликів API. У сучасних системах, наприклад, всі координати екрана є 32-бітними числами, а в системі Windows 95/98 використовуються тільки молодші 16 бітів (для сумісності з Windows 3.1). Існування набору викликів Win32 API на декількох різних операційних системах спрощує перенос програм між ними, але при цьому дещо видаляється з основної системи викликів. Розходження між Windows 95/98 і сучасними системами Windows викладені в табл. 4.

Таблиця 4 – Деякі розходження між версіями Windows

Характеристика	Windows 95/98	Windows NT 5.X
Win32 API	Так	Так
Повністю 32-бітна система	Ні	Так
Захист	Ні	Так
Відображення захищених файлів	Ні	Так
Окремий адресний простір для кожної програми MS-DOS	Ні	Так
Plug and Play	Так	Так
Підтримка Unicode	Ні	Так
Процесор	Тільки Intel 80x86	Всі Intel
Багатопроцесорна підтримка	Ні	Так
Реентерабельна програма (припускає повторне входження) усередині операційної системи	Ні	Так
Користувач може сам написати деякі важливі частини операційної системи	Так	Ні

4.3 Приклади віртуальної пам’яті

Після розглядання загальної структури обох систем є сенс більш докладно поговорити про віртуальну пам’ять у системах UNIX і Windows. З погляду програміста, якщо абстрагуватись від деталей, вони багато в чому подібні одна до одної.

4.3.1 Віртуальна пам’ять UNIX

Модель пам’яті в системі UNIX досить проста. Кожен процес має три сегменти: код, дані й стек, як показано на рис. 31. У машині з лінійним адресним простором код звичайно розташовується в нижній частині пам’яті, а за ним ідуть дані. Стек міститься у верхній частині пам’яті. Розмір коду фіксований, а дані й

стек можуть збільшуватися або зменшуватися. Таку модель легко реалізувати практично на будь-якій машині. Наприклад, вона використовується й в операційній системі Solaris на процесорі UltraSPARC.

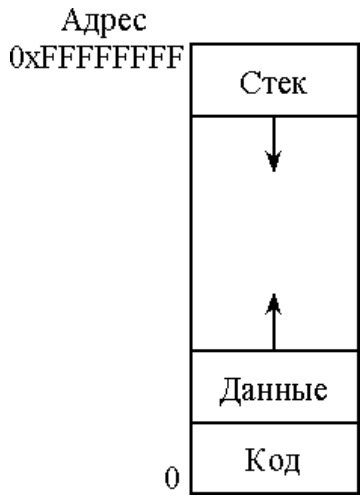

Рисунок 31 – Адресний простір одного процесу UNIX

Більше того, якщо машина має сторінкову пам’ять, то й весь адресний простір може бути розбитим на сторінки, а користувальницькі програми цього не помітять. Єдине, що їм буде відомо – те, що розмір програми може перевищувати розмір фізичної пам’яті машини. Системи UNIX, у яких немає сторінкової організації пам’яті, звичайно перекачують цілі процеси між пам’яттю й диском, щоб надати можливість як завгодно великій кількості процесів працювати в режимі поділу часу.

Опис, даний вище (віртуальна пам’ять із підкачуванням сторінок на вимогу), у цілому підходить для Berkeley UNIX. Однак System V (і Solaris) має деякі особливості, що дозволяють користувачам управляти віртуальною пам’яттю. Найважливішої є здатність процесу відображати файл або частину файла на частину власного адресного простору. Наприклад, якщо файл в 12 КіБ відображається на віртуальну адресу 144 кібі, то у комірці пам’яті з адресою 144 кіби буде міститися перше слово цього файла. Таким чином, можна здійснювати ввід-вивід даних у файл без застосування системних викликів. Оскільки розмір деяких файлів

може перевищувати розмір віртуального адресного простору, можна відображати не весь файл, а тільки його частину. Щоб здійснити відображення, спочатку потрібно відкрити файл і одержати ***дескриптор файла*** **fd** (file descriptor). Дескриптор використовується для ідентифікації файла, який потрібно відобразити. Безпосередньо для відображення файла процес повинен звернутися до системного виклику **mmap**, наприклад таким чином:

```
paddr = mmap(virtual_address, length, protection, flags, fd, file_offset);
```

який відображає **length** байтів, починаючи з **file_offset** у файлі, на віртуальний адресний простір, починаючи з **virtual_address**. Параметр **flags** вимагає, щоб система вибрала віртуальну адресу, котра потім вертається через **paddr.** Відображувана область повинна містити ціле число сторінок і повинна бути вирівняна в границях сторінки. Параметр **protection** визначає дозвіл на читання, запис і виконання (у будь-якій комбінації). Відображення можна надалі видалити за допомогою системного виклику **munmap**, з наступним прототипом:

```
int munmap(void *addr, size_t length);.
```

У той самий файл можна одночасно відображати кілька процесів. Є два варіанти поділу загальних сторінок. У першому випадку розділяються всі сторінки, тому записи, вироблені одним процесом, видні всім іншим процесам. Ця можливість забезпечує між процесами тракт із високою пропускною здатністю. У другому випадку сторінки розділяються всіма процесами доти, поки який-небудь процес не змінить їх. Як тільки один із процесів намагається зробити запис у сторінку, він одержує помилку захисту, у результаті чого операційна система видає йому копію цієї сторінки, у яку можна робити запис. Така схема використовується в тому випадку, коли для кожного з декількох процесів потрібно створити ілюзію, що тільки він відображається у файл.

4.3.2 Віртуальна пам’ять Windows

В Windows кожен користувальницький процес має свій власний віртуальний адресний простір. Довжина віртуальної адреси становить 32(64) біта, тому кожен процес має 4 ГіБ віртуального адресного простору. Нижні 2 ГіБ призначені для коду й даних процесу; верхні 2 ГіБ дозволяють обмежений доступ до пам’яті ядра. Виключення становлять версії для підприємств, у яких поділ пам’яті може бути іншим: 3 ГіБ – для користувача й 1 ГіБ – для ядра. Віртуальний адресний простір дозволяє підкачування сторінок на вимогу і має сторінки фіксованого розміру (4 КіБ на машині Pentium).

Кожна віртуальна сторінка може перебувати в одному із трьох станів: вона може бути ***вільною*** (**`free`**), ***зарезервованою*** (**`reserved`**) і ***виділеною*** (**`committed`**). Знаходження сторінки у вільному стані означає, що вона терміново не використовується, а звертання до неї викликає помилку через відсутність сторінки. Коли процес починається, всі його сторінки перебувають у вільному стані доти, поки програма і її початкові дані не будуть відображені у свій адресний простір. Якщо код або дані відображені в сторінку, то така сторінка стає виділеною. Звертання до виділеної сторінки буде успішним, якщо сторінка перебуває в основній пам’яті. Якщо сторінка відсутня в основній пам’яті, то, як завжди, відбудеться помилка і операційна система буде змушена викликати потрібну сторінку з диска. Віртуальна сторінка може перебувати й у зарезервованому стані. Це значить, що ця сторінка недоступна для відображення доти, поки резервування не буде скасовано. Крім атрибутів стану сторінки мають і інші атрибути (наприклад, таки, що вказують на можливість читання, записи й виконання). Верхні 64 КіБ і нижні 64 КіБ пам’яті завжди вільні, щоб можна було відшукувати помилки вказівників (неініційовані вказівники часто дорівнюють 0 або -1).

Кожна виділена сторінка має ***тіньову сторінку*** на диску, де вона зберігається при відсутності її в основній пам’яті. Вільні і зарезервовані сторінки не мають тіньових сторінок, тому звертання до них спричиняють помилки через відсутність сторінки (система не може викликати сторінку з диска, якщо цієї сторінки

немає на диску). Тіньові сторінки на диску згруповані в один або кілька сторінкових файлів. Операційна система постійно стежить, у яку частину якого сторінкового файла відображається кожна віртуальна сторінка. Файли з текстами програм мають власні тіньові сторінки, для сторінок даних використовуються спеціальні сторінкові файли.

Windows, як і System V, дозволяє відображати файли безпосередньо до області віртуального адресного простору. Якщо файл був відображений в адресний простір, його можна зчитувати або записувати шляхом звичайних звертань до пам'яті.

Відображувані на пам'ять файли реалізуються так само, як інші виділені сторінки, тільки тіньові сторінки можуть перебувати у файлі на диску, а не в сторінковому файлі. У результаті, коли файл відображається, версія в пам'яті може не збігатися з версією на диску (через останні записи у віртуальний адресний простір). Однак коли відображення файла видаляється, версія на диску обновляється.

Windows також дозволяє відображати два й більше процесів в одному файлі одночасно, можливо, у різних віртуальних адресах. Шляхом зчитування слів з пам'яті й запису слів у пам'ять процеси можуть взаємодіяти один з одним і передавати дані в обох напрямках з досить високою швидкістю, оскільки ніякого копіювання не потрібно. Різні процеси можуть мати різні дозволи на доступ. Всі процеси, що використовують відображений файл, розділяють ті самі сторінки, тому зміни, зроблені одним з процесів, видні всім іншим процесам, навіть якщо файл на диску ще не був оновлений. Win32 API містить ряд функцій, які дозволяють процесу відкрито управляти віртуальною пам'яттю. Найважливіші із цих функцій наведені в табл. 5. Всі вони працюють в області, що складається або з однієї сторінки, або із двох, або більше сторінок, послідовно розташованих у віртуальному адресному просторі.

Перші чотири функції очевидні. Наступні дві функції дозволяють процесу робити скільки-небудь область пам'яті розміром до 30 сторінок і скасовувати цю дію. Ця якість може знадобитися програмам, що працюють у режимі реального часу. Операційна система встановлює певну межу, щоб процеси не ставали занад-

то поглинаючими. У системі NT також є функції API, які дозволяють процесу одержувати доступ до віртуальної пам'яті іншого процесу (вони не зазначені в табл. 8).

Таблиця 5 – Основні функції API для керування віртуальною пам'яттю в системі Windows NT

Функція API	Значення
VlrtualAlloc	Резервація або виділення області
VirtualFree	Звільнення області або зняття виділення
Virtual Protect	Зміна типу захисту на читання/запис/виконання
VlrtualQuery	Запит про стан області
VlrtualLock	Робить область пам'яті резидентною (тобто забороняється розбивка на сторінки в ній)
VlrtualUnlock	Знімає заборону на розбивку на сторінки
CreateRleMappIng	Створює об'єкт відображення файла й іноді приписує йому ім'я
MapVlewOfFile	Відображає файл або частину файла в адресний простір
UnmapVlewOfFlle	Видаляє відображений файл із адресного простору
OpenFlleMappIng	Відкриває раніше створений об'єкт відображення файла

Останні 4 функції API призначені для керування відображуваними на пам'ять файлами. Щоб відобразити файл, спочатку потрібно створити об'єкт відображення файла за допомогою функції **CreateFileMapping**. Ця функція повертає ідентифікатор (handle) об'єкту відображення файла й іноді ще й уводить у систему файлів ім'я для нього, щоб інший процес міг використовувати об'єкт. Дві функції відображають файли й видаляють відображення відповідно. Наступна функція потрібна для того, щоб відобразити файл, що у цей момент відображений

іншим процесом. Таким чином, два й більше процесів можуть розділяти області своїх адресних просторів.

Ці функції API є основними. На них будується вся інша система керування пам'яттю. Наприклад, існують функції API для розміщення й звільнення структур даних в одній або декількох ***купах***. Купи використовуються для зберігання структур даних, які створюються й руйнуються динамічно. Купи не займаються «***збиранням сміття***», тому користувальницьке програмне забезпечення самостійно повинне звільняти блоки віртуальної пам'яті, які вже не потрібні. («Збирання сміття» – це автоматичне видалення структур даних, котрі вже не використовуються). Створення купи в Windows подібне до виділення пам'яті функцією **`malloc`** у системах UNIX, але у Windows, на відміну від UNIX, може бути кілька незалежних куп.

4.4 Приклади віртуального вводу-виводу

Як вже відмічалося вище основним завданням будь-якої операційної системи є надання служб для користувальницьких програм. Головним чином, це служби вводу-виводу для читання й запису файлів. І UNIX, і Windows пропонують широкий спектр служб вводу-виводу. Для більшості системних викликів UNIX в Windows є еквівалентний виклик, але зворотне не вірно, оскільки Windows містить набагато більше викликів і кожен з них є набагато складнішим до відповідного виклику в UNIX.

4.4.1 Віртуальний ввід-вивід у системі UNIX

Система UNIX користувалася великою популярністю багато в чому завдяки своїй простоті, що, у свою чергу, є прямим результатом організації системи файлів. Звичайний файл являє собою лінійну послідовність 8-бітних байтів від 0 до максимум 2^{32}-1 байтів. Сама операційна система ніколи не повідомляє користувальницькі програми про структуру записів у файлах, хоча багато користувальницьких програм розглядають, наприклад, текстові файли в коді ASCII як послідовності рядків, кожен з яких завершується переводом рядка (редактори текстів).

З кожним відкритим файлом зв'язується покажчик на наступний байт, який потрібно зчитати або записати. Системні виклики **read** і **write** зчитують і записують дані, починаючи з позиції, котру визначає покажчик. Після операції обидва виклики переміщають покажчик в іншу позицію, пересуваючи його рівно на стільки байтів, скільки було зчитано або записано. Можливий і випадковий доступ до файлів, коли покажчик файла встановлюється на певне вказане значення.

Крім звичайних файлів, система підтримує спеціальні файли, які використовуються для доступу до пристроїв вводу-виводу. З кожним пристроєм вводу-виводу звичайно зв'язується один або декілька спеціальних файлів. Зчитуючи інформацію із цих файлів і записуючи інформацію в ці файли, програма може зчитувати інформацію із пристрою вводу-виводу й записувати інформацію на пристрій вводу-виводу. Так відбувається робота з дисками, принтерами, терміналами й багатьма іншими пристроями.

Основні системні виклики для файлів в UNIX наведені в табл. 6. Виклик **creat** (без *е* на кінці) використовується для створення нового файла. У цей час він не є обов'язковим, оскільки виклик **open** теж може створювати новий файл. Виклик **unlink** видаляє файл (передбачається, що файл перебуває тільки в одній директорії).

Виклик **open** використовується для відкриття існуючих файлів, а також для створення нових. Прапор **mode** повідомляє, як його відкрити (для читання, для запису й т.ін.). Виклик повертає невелике ціле число, що називається ***дескриптором файла***. Дескриптор файла визначає файл у наступних викликах. Сам процес вводу-виводу здійснюється за допомогою процедур **read** і **write**, кожна з яких містить дескриптор файла (він указує, який файл використовувати), буфер для даних і кількість даних у байтах, яку потрібно передати. Виклик **lseek** використовується для переміщення покажчика файла, що уможливлює випадковий доступ до файлів.

stat вертає інформацію про файл (розмір, час останнього доступу, ім'я власника й т.п.). **chmod** змінює режим захисту файла (наприклад, дозволяє або,

навпаки, забороняє яким-небудь користувачам читати його). Нарешті, **fend** виконує різні дії над файлами, наприклад блокування й розблокування.

Таблиця 6 – Основні системні виклики UNIX

Системний виклик	Значення
creat(name, mode)	Створює файл; **mode** визначає тип захисту
unlink(name)	Видаляє файл (передбачається, що є тільки 1 зв'язок)
open(name, mode)	Відкриває або створює файл і повертає дескриптор файла
close(fd)	Закриває файл
read (fd, buffer, count)	Зчитує байти в кількості **count** в **buffer**
write(fd, buffer, count)	Записує у файл **count** байтів з **buffer**
lseek(fd, offset, w)	Переміщає покажчик файла на **offset** і **w**
stat(name, buffer)	Повертає інформацію про файл
chmod(name, mode)	Змінює тип захисту файла
fcntl(fd, cmd, j)	Робить різні операції керування (наприклад, блокує файл або його частину)

На рис. 32 показано, як відбувається процес вводу-виводу. Ця програма мінімальна й не містить у собі перевірку помилок. Перед тим як увійти в цикл, програма відкриває існуючий файл **data** і створює новий файл **newf.** Кожен виклик повертає дескриптор файла **infd** і **outfd** відповідно. Наступний параметр в обох викликах – біти захисту, які визначають, що файли потрібно вважати й записати відповідно. Обидва виклики повертають дескриптор файла. Якщо не вдалося зробити **open** або **creat**, то вертається негативний дескриптор файла, що повідомляє, що виклик не вдався.

Цей фрагмент написаний мовою С, оскільки в мові Java не відсутня можливість звертання до системних викликів низького рівня, але в системному програмування вони використовуються часто і потрібно їх знати.

```
/* Відкриття дескрипторів файла */
  infd = open("data", 0);
  outfd = creat("newf", ProtectionBits):
/* Цикл копіювання */
  do{
      count = read(infd, buffer, bytes);
      if (count > 0) write(outfd, buffer,  count);
    }
  while (count > 0);
/* Закриття файлів */
  close(infd);
  close(outfd);
```

Рисунок 32 – Фрагмент програми для копіювання файла з використанням системних викликів UNIX

Виклик **read** має три параметри: дескриптор файла, буфер і число байтів. Даний виклик повинен зчитати потрібне число байтів із зазначеного файла в буфер. Число зчитаних байтів поміщається в **count. Count** може бути менше, ніж **bytes**, якщо файл був занадто коротким або зчитаним є залишок файла. Виклик **write** копіює зчитані байти у вихідний файл. Цикл триває доти, поки вхідний файл не буде прочитаний повністю. Тоді цикл завершується, а обидва файли закриваються.

Дескриптори файлів у системі UNIX – це невеликі цілі числа (звичайно до 20). Дескриптори файлів 0, 1 і 2 відповідають стандартному вводу, стандартному виводу й стандартному пристрою повідомлення про помилку відповідно. Звичайно перший з них звертається до клавіатури, а другий і третій – до дисплея, але користувач може перенаправляти їх до інших файлів. Багато програм UNIX одержують вхідні дані зі стандартного пристрою вводу й записують вихідні дані в стандартний пристрій виводу. Такі програми називаються фільтрами. Такий підхід дозволяє застосовувати механізм передачі даних безпосередньо від однієї програми до іншої.

Із системою файлів тісно зв’язана система директорій. Кожен користувач може мати будь-яку бажану кількість директорій, а кожна директорія може містити файли й піддиректорії. Система UNIX звичайно конфігурується з головною ди-

ректорією, так званим ***кореневим каталогом***, що містить піддиректорії **bin** (для часто використовуваних програм), **dev** (для спеціальних файлів пристроїв вводу-виводу), **lib** (для бібліотек) і **usr** (для користувальницьких директорій, як показано на рис. 33). У цьому прикладі директорія **usr** містить піддиректорії **ast** і **jim**. Директорія **ast**, у свою чергу, містить у собі два файли (**data** і **foo.c**) і піддиректорію **bin**, у яку входять чотири гри.

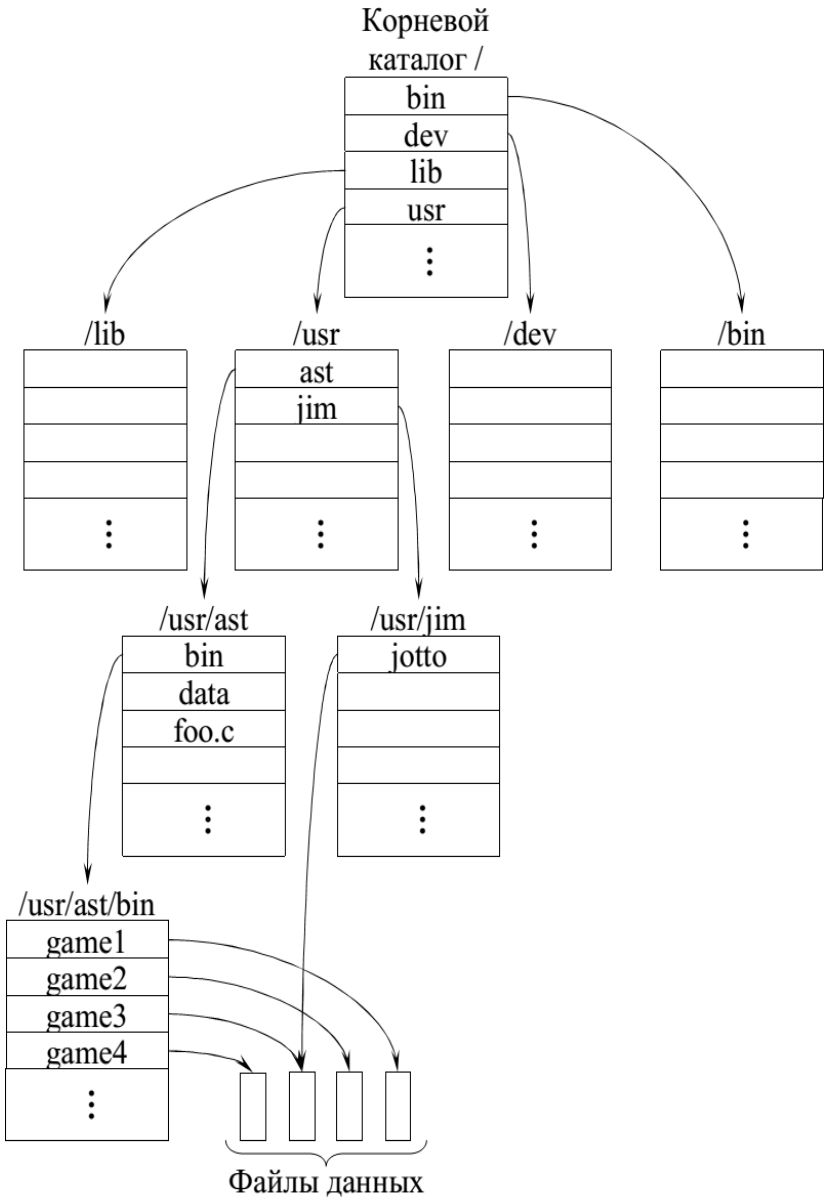

Рисунок 33 – Умовне позначення частини системи директорій в операційній системі UNIX

Щоб назвати файл, потрібно вказати його шлях від кореневого каталогу. Шлях містить список всіх директорій від кореневого каталогу до файла, для поділу директорій використовується слеш. Наприклад, шлях до файла **game2** буде таким: **/usr/ast/bin/game2**. Шлях, що починається з кореневого каталогу, називається ***абсолютним шляхом***.

У кожен момент часу кожна працююча програма має ***поточний каталог***. Шлях може бути пов'язаний з поточним каталогом. У цьому випадку на початку шляху слеш не ставиться (щоб відрізнити такий шлях від абсолютного шляху). Такий шлях називається ***відносним шляхом***. Якщо **/usr/ast** – поточний каталог, то можна одержати доступ до файла **game3**, використовуючи шлях **bin/game3**. Користувач може створити зв'язок із чужим файлом, використовуючи для цього системний виклик **link**. У нашім прикладі шляхи **/usr/ast/bin/game3** і **/usr/jim/jotto** приведуть до одного й того ж файла. Не дозволяється застосовувати зв'язування для директорій, щоб запобігти циклам у системі директорій. Виклики **open** і **creat** використовують як абсолютні, так і відносні шляхи.

Основні виклики для операцій з директоріями в системі UNIX наведені в табл. 7. **mkdir** створює нову директорію, а **rmdir** видаляє існуючу *порожню* директорію. Наступні три виклики застосовуються для читання елементів директорій. Перший відкриває директорію, другий зчитує елементи з її, а третій закриває директорію. **chdir** змінює поточну директорію.

link створює елемент директорії, що вказує на вже існуючий файл. Наприклад, елемент **/usr/jim/jotto** можна створити за допомогою виклику

```
link("/usr/ast/bin/game3", "/usr/jim/jotto");
```

або за допомогою еквівалентного виклику, використовуючи відносні шляхи, які залежать від поточної директорії. **unlink** видаляє елемент директорії. Якщо файл має тільки один зв'язок, він видаляється. Якщо він має два й більше зв'язків,

то він не видаляється. Не має ніякого значення, чи був вилучений зв'язок спокон-вічно створений, або це копія. Виклик

```
unlink("/usr/ast/bin/game3");
```

робить файл **game3** доступним тільки через шлях **/usr/jim/jotto**. Виклики **link** і **unlink**, будучи викликаними послідовно один за одним, можуть використовуватися для переміщення файлів з однієї директорії в іншу.

Таблиця 7 – Основні виклики для роботи з директоріями в системі UNIX

Системний виклик	Значення
mkdir(name, mode)	Створює нову директорію
rmdir(name)	Видаляє порожню директорію
opendir(name)	Відкриває директорію для читання
readdir(dirpointer)	Читає наступний елемент директорії
closedir(dirpointer)	Закриває директорію
chdir(dirname)	Змінює поточний каталог на **dirname**
link(name1, name2)	Створює елемент директорії **name2**, котрий вказує на **name1**
unlink(name)	Видаляє **name** з директорії

Запис у директорії для кожного її елемента має структуру, котра наведена у табл. 8.

Особливу увагу у цьому запису слід звернути на перше поле по зміщенню 0 і довжиною 4 байти. В системі UNIX кожен файл (і кожна директорія, оскільки директорія – це теж файл) характеризується окремим дуже важливим блоком інформації розмиром 64 байта. Цей блок називається ***індексним дескриптором*** – ***i-node***. I-node повідомляє, хто володіє файлом, що дозволено робити з файлом, де знайти дані й т.ін. Індексні дескриптори для файлів розташовані або послідовно

на початку диска, або, якщо диск розділений на групи циліндрів, – на початку циліндра (рис. 34). Індексні дескриптори мають послідовні номери, котрі і вміщуються до запису про єлемент. Таким чином, система UNIX може знайти потрібний i-node просто шляхом обчислення його адреси на диску просто помноживши його номер на 64.

Таблиця 8 – Структура запису для елемента директорії

Зміщення поля	Розмір поля	Ім’я поля	Опис
0	4	**i-node**	посилання (номер) на індексний дескриптор
4	2	**rec_len**	довжина даного запису
6	1	**name_len**	довжина імені файла
7	1	**file_type**	тип файла
8	...	**name**	ім’я файла

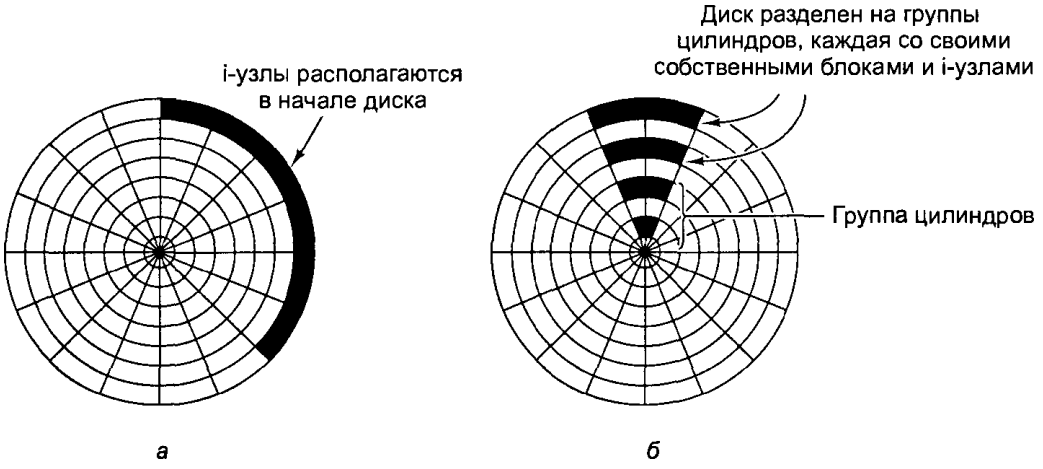

Рисунок 34 – i-node, що розміщені на початку диска (а); диск, поділений на групи циліндрів, кожна зі своїми власними блоками та i-node (б)

Фактично можна вважати, що елемент запису про файл у директорії (як це в дійсності має місце у деяких різновидах системи UNIX) містить два основних по-

ля – i-node і ім’я файла. Операційна система маючи таку інформацію в змозі за допомогою простого алгоритму знайти на диску будь-який файл.

Наприклад, коли програма виконує команду:

```
open("foo.c", 0);,
```

система шукає поточний каталог для файла «foo.c», щоб знайти номер індексного дескриптора для цього файла. Зчитавши номер індексного дескриптора, програма може одержати всю інформацію про цей файл.

При більшій довжині шляху до файла основні кроки, викладені вище, повторюються кілька разів, поки не буде пройдений весь шлях. Наприклад, щоб знайти номер індексного дескриптора для шляху **/usr/ast/data**, система спочатку шукає кореневий каталог для елемента **usr**. Виявивши індексний дескриптор для **usr**, вона може прочитати цей файл (директорія в системі UNIX – це теж файл). У цьому файлі вона шукає елемент **ast** і знаходить номер індексного дескриптора для файла **/usr/ast**. Зчитавши інформацію про місцезнаходження директорії **/usr/ast**, система може виявити елемент для **data** і, отже, номер індексного дескриптора для **/usr/ast/data**. Знайшовши номер індексного дескриптора для цього файла, система може вже довідатися все й про цей файл.

Відносні шляхи файлів обробляються так само, як і абсолютні, з тією різницею, що алгоритм починає роботу не з кореневого, а з робочого каталогу. У кожному каталозі є елементи «**.**» і «**..**», що поміщаються в каталог у момент його створення. Елемент «**.**» містить номер і-вузла поточного каталогу, а елемент «**..**» – номер i-node батьківського каталогу. Таким чином, процедура, що шукає файл **../ast/foo.c**, просто знаходить «**..**» у робочому каталозі (у даному разі **/usr** на рис. 33), розшукує в ньому номер i-node батьківського каталогу, у якому шукає дескриптор каталогу **ast**. Для обробки цих імен не потрібно спеціального механізму. Імена робочого й батьківського каталогів – це звичайні ASCII-рядки, що не відрізняються від будь-яких інших імен.

Формат, зміст і розміщення індексних дескрипторів трохи різняться в різних системах (особливо коли мова йде про мережу), але наступні характеристики властиві дескрипторам практично всіх UNIX-систем:

1. Тип файла.
2. 9 бітів захисту RWX і деякі інші біти.
3. Число зв'язків з файлом (число елементів директорій).
4. Ідентифікатор власника.
5. Група власника.
6. Довжина файла в байтах.
7. Тринадцять адрес на диску.
8. Час, коли файл читали востаннє.
9. Час, коли останній раз вироблявся запис у файл.
10. Час, коли востаннє мінявся індексний дескриптор.

Типи файлів бувають наступні: звичайні файли, директорії й два види особливих файлів для пристроїв вводу-виводу із блоковою структурою й неструктурованими пристроями вводу-виводу відповідно.

Дев'яти-бітне поле повідомляє, кому дозволений доступ до цього файла. Це поле поділене на три окремих поля RWX (Read, Write, eXecute – читання, запис, виконання). Перше з них контролює дозвіл на читання, запис і виконання файлів для їхнього власника, друге – для інших користувачів із групи власника, а третє – для будь-яких користувачів. Наприклад, наступне відображення полів у вигляді **RWX R-X --X** означає, що власник файла може читати цей файл, записувати щонебудь у нього й виконувати його (очевидно, файл є виконуваною програмою, інакше не було б дозволу на його виконання), інші члени групи можуть читати й виконувати його, а сторонні люди – тільки виконувати. Таким чином, сторонні люди можуть використовувати цю програму, але не можуть з неї украсти (скопіювати), оскільки їм заборонено читання. Віднесення користувачів до тих або інших груп здійснюється системним адміністратором, якого звичайно називають привілейованим користувачем. Привілейований користувач має право діяти всупереч механізму захисту й зчитувати, записувати й виконувати будь-який файл.

Число зв’язків показує кілька разів для файла застосовувався системний виклик **`link`**. Ідентифікатор власника і його групи надається кожному користувачу системи при його реєстрації в системі і перевіряється кожен раз коли користувач входить у систему.

Довжина файла виражається 32-бітним цілим числом, що показує найстарший байт файла. Цілком можливо створити файл, потім системним викликом **`lseek`** перенести покажчик на позицію 1000000 і записати 1 байт. У результаті вийде файл довжиною 1 000 001. Проте цей файл не потребує збереження всіх відсутніх байтів.

Сьома позиція в переліку полів являє особливий інтерес, оскільки у цьому полі міститься повна інформація про розміщення на диску окремих блоків, що належать до файла. Поле складається з тринадцяти 32-бітних адрес. Перші 10 адрес безпосередньо вказують на блоки даних на диску – ***пряма адресація***. Якщо розмір блоку становить 1024 байта, то можна працювати з файлами розмір яких не перевищує 10240 байтів. Зрозуміло, що цього замало. Тому одинадцята адреса у полі указує на блок ***непрямої адресації***, що містить 256 адрес блоків на диску. Тепер вже є можливість збільшити розмір і працювати з файлами розміром до 10240+256*1024=272384 байта. Але й цього ще занадто мало. Для адресації блоків файлів ще більшого розміру слугує дванадцята адреса у полі. Вона вже вказує на 256 блоків непрямої адресації – ***блоки подвійної непрямої адресації***. При використанні цієї адреси припустимий розмір файлів знов збільшується і вже становить 272384+256*256*1024=67381248 байтів. Якщо й цій схеми блоку подвійної непрямої адресації занадто мало (деякі файли, особливо у базах даних, значно перевищують цей розмір), то використовується тринадцята адреса. Вона указує на блок ***потрійної непрямої адресації***, що містить адреси 256 блоків подвійної непрямої адресації. Така багатоступенева система адресації блоків на диску схематично наведена на рис. 35.

Використовуючи пряму, непряму, подвійну непряму й потрійну непряму адресацію, можна звертатися до 16843018 блоків. Це значить, що максимально можливий розмір файла становить 17247250432 байта. Оскільки розмір покажчи-

ків файлів обмежений до 32 битов, реальна верхня межа на розмір файла становить 4294967295 байтів.

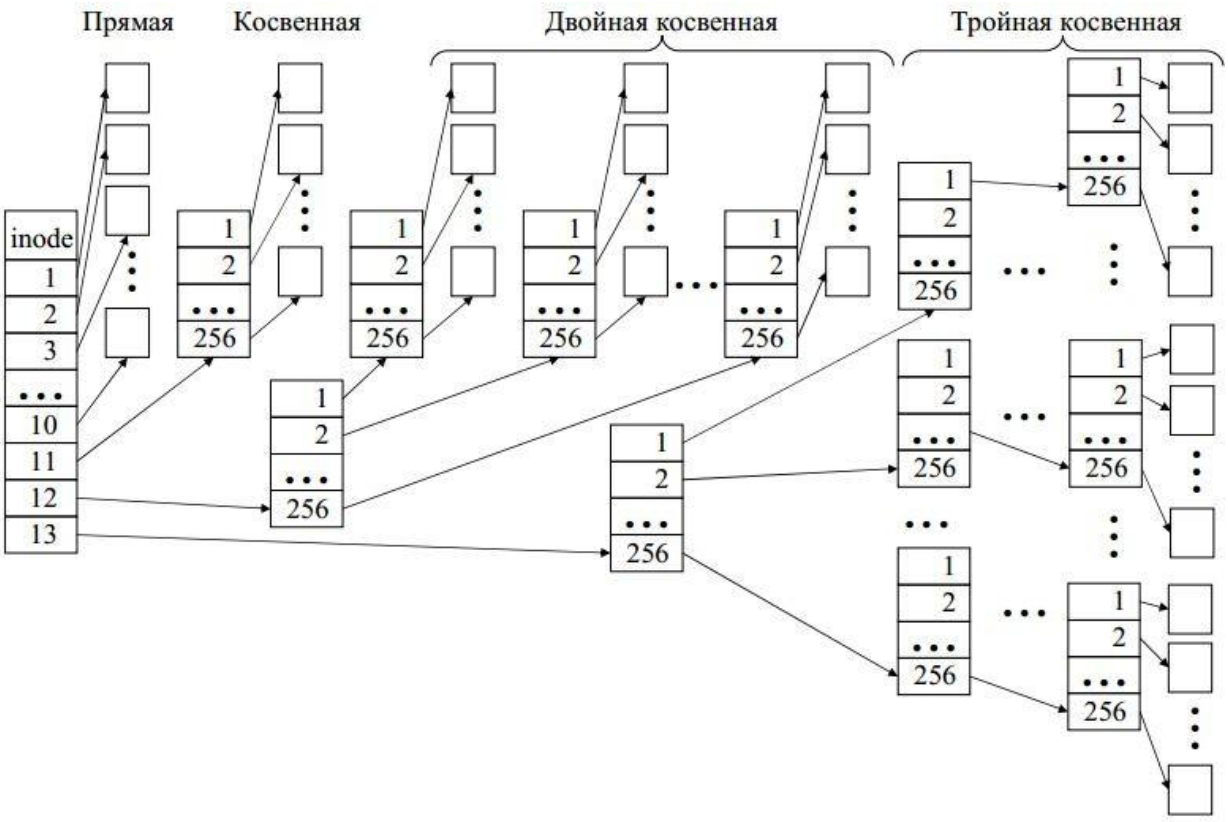

Рисунок 35 – Схема використання поля i-node для адресації блоків на диску

Вільні блоки диска зберігаються у зв'язному списку. Якщо потрібний новий блок, він береться зі списку. У результаті виходить, що блоки кожного файла безладно розкидані по всім диску.

Щоб підвищити швидкість вводу-виводу з диска, у ОС застосовуються наступні механізми. Після відкриття файла його індексний дескриптор копіюється в таблицю в основній пам'яті й зберігається там, поки файл залишається відкритим. Крім того, у пам'яті перебуває набір блоків, до яких недавно вироблялося звернення. Тому що більшість файлів зчитується послідовно, часто при звертанні до файла потребується той же блок, що й при попередньому зверненні. Щоб збільшити швидкість, система зчитує наступний блок у файл ще до того, як до нього зроблене звернення. Всі ці моменти сховані від користувача. Коли користувач ро-

бить виклик **read**, робота програми припиняється, поки необхідні дані не з'являться в буфері.

Знаючи все це, можно розглянути, як відбувається процес вводу-виводу. Системний виклик **open** змушує систему шукати директорії по певному шляху. Якщо пошук успішний, то індексний дескриптор зчитується у внутрішню таблицю. В свою чергу виклики **read** і **write** вимагають, щоб система обчислила номер блоку з поточної позиції файла. Адреси перших 10 блоків диска завжди перебувають в основній пам'яті (в індексному дескрипторі); для інших блоків спочатку потрібно зчитати один або кілька блоків непрямої адресації. Виклик позиціонування – **lseek** просто змінює поточну позицію покажчика й не робить ніякого вводу-виводу.

Коли виконується приєднання файла до каталогу – **link**, то виклик дивиться на свій перший аргумент, щоб виявити номер індексного дескриптора. Потім він створює елемент директорії для другого аргументу й поміщає номер індексного дескриптора першого файла в цей елемент директорії. Нарешті, він збільшує число зв'язків в індексному дескрипторі на 1. З іншого боку виклик **unlink** видаляє елемент директорії й зменшує число зв'язків в індексному дескрипторі на одиницю. Якщо це число дорівнює 0, файл видаляється й всі блоки, що належали файла, переміщуються до списку вільних блоків.

4.4.2 Віртуальний ввід-вивід в Windows

Windows підтримує кілька файлових систем, найважливіші з яких – NTFS (New Technology File Sysytem – файлова система Windows NT) і FAT (File Allocation Table – таблиця розміщення файлів). Перша була розроблена спеціально для NT. Друга є старою файловою системою для MS-DOS, що також використовується в Windows 95/98 (хоча й з довгими іменами файлів). Її різновид FAT32 – модернізована файлова система для роботи з дисками великого розміру спочатку теж використовувалася в NT 5.0 одночасно з NTFS. Вона підтримувалася й у

більше пізніх версіях Windows 95 і Windows 98. Оскільки система FAT вже морально застаріла, має сенс розглядати лише файлову систему NTFS.

У файловій системі NT довжина імені файла може сягати 255 символів. Імена файлів у директоріях зберігаються в коді Unicode, завдяки чому люди в різних країнах, де не використовується латинський алфавіт, можуть писати імена файлів на їхній рідній мові. У файловій системі NT великі й малі літери в іменах файлів вважаються різними (тобто foo відрізняється від FOO). На жаль, у системі Win32 API великі й малі літери в іменах файлів і директорій не різняться, тому ця перевага губиться для програм, які використовують Win32.

Як і в системі UNIX, файл являє собою лінійну послідовність байтів. Максимальна довжина файла становить 2^{64}-1 байтів. Покажчики теж існують, але їхня довжина не 32, а 64 біта, щоб можна було підтримувати максимальну довжину файла. Виклики функцій в Win32 API для маніпуляцій з директоріями й файлами в цілому схожі з викликами функцій у системі UNIX, але більшість із них мають більше параметрів, і модель захисту інша. При відкритті файла вертається ***ідентифікатор*** (handle), що потім використовується для читання й запису файла. На відміну від системи UNIX, ідентифікатори не є маленькими цілими числами, а стандартний пристрій вводу, стандартний пристрій виводу і стандартний пристрій повідомлення про помилку заздалегідь не визначаються як 0, 1 і 2 (виключення становить термінальний режим роботи). Основні функції Win32 API для керування файлами наведені в табл. 9.

Виклик **`CreateFile`** використовується для створення нового файла й повертає ідентифікатор (handle) для нього. Ця функція застосовується й для відкриття вже існуючого файла, оскільки в системі API немає функції **`open`**. В цьому конспекті немає можливості приводити всі параметри всіх функцій API, оскільки їх дуже багато. Наприклад, **`CreateFile`**, на відміну від **`open`**, має наступних сім параметрів.

1. Покажчик на ім’я файла, який потрібно створити або відкрити.
2. Прапори, які повідомляють, які дії дозволено робити з файлом: читати, записувати або й те й інше.

3. Прапори, які повідомляють, чи можуть кілька процесів відкривати файл одночасно.

Таблиця 9 – Основні функції Win32 API для вводу-виводу файлів (у другому стовпчику дається еквівалент із UNIX)

Функція API	UNIX	Значення
CreateFile	**open**	Створює файл або відкриває існуючий файл; повертає ідентифікатор
DeleteFlle	**unlink**	Видаляє існуючий файл
CloseHandle	**close**	Закриває файл
ReadFile	**read**	Зчитує дані з файла
WriteFile	**write**	Записує дані у файл
SetFilePointer	**lseek**	Установлює покажчик файла на певне місце у файлі
SetFileAttributes	**stat**	Повертає властивості файла
LockFile	**fcntl**	Блокує область файла, щоб забезпечити взаємне виключення доступу
UnlockFile	**fcntl**	Знімає блокування з раніше заблокованої області файла

4. Покажчик на ***дескриптор безпеки***, що повідомляє, хто має доступ до файла.
5. Прапори, які повідомляють, що потрібно робити, якщо файл існує або не існує.
6. Прапори, пов’язані з атрибутами архівації, компресії й т.ін.
7. Ідентифікатор файла (handle), атрибути якого потрібно клонувати для нового файла.

Більш докладно параметри функцій API і самі функції наведені у методичних вказівках до лабораторних робіт з курсу.

Наступні шість функцій API подібні до відповідних функцій в системі UNIX. Останні дві дозволяють блокувати й розблоковувати область файла, щоб забезпечити взаємне виключення доступу.

Використовуючи ці функції API, можна написати процедуру для копіювання файла, котра аналогічна процедурі з рис. 32. Така процедура (без перевірки помилок) наведена на рис. 36. На практиці програму для копіювання файла писати не потрібно, оскільки у API існує спеціальна функція **CopyFile**.

```
/* Відкриття файлів для вводу й виводу */
  inhandle = CreateFile("data", GENERIC_READ, 0,  NULL,
                        OPEN EXISTING, 0, NULL);
  outhandle = CreateFile("newf", GENERIC_WRITE, 0,  NULL,
                         CREATE_ALWAYS,
                         FILE_ATTRIBUTE_NORMAL,  NULL);
/* Копіювання файла */
  do{
      s = ReadF1le(inhandle,buffer, BUF_SIZE,&count, NULL);
      if (s > 0 && count > 0)
         WriteFile(outhandle, buffer, count, &ocnt,  NULL);
    }
  while (s > 0 && count > 0);
/* Закриття файлів */
  CloseHandle(inhandle);
  CloseHandle(outhandle);
```

Рисунок 36 – Фрагмент програми для копіювання файла із застосуванням функцій Win32 API

Фрагмент, знов-таки як і попередній, написано мовою C, оскільки мова Java немає безпосереднього доступу до викликів низького рівня.

Windows підтримує ієрархічну систему файлів, подібну до системи файлів UNIX. Однак як роздільник тут використовується не **/**, а **** (запозичене з MS-DOS). Тут теж існує поняття поточного каталогу, а шляхи можуть бути абсолютними й відносними. Однак між Windows і UNIX є одне істотне розходження. UNIX дозволяє монтувати в одне дерево системи файлів з різних дисків і машин, приховуючи в такий спосіб структуру диска від програмного забезпечення. Ранні версії Windows не мають можливості робити таке монтування, тому абсолютні

імена файлів в ОС Windows повинні починатися з букви, що вказує на диск (наприклад, **C:\windows\system\foo.dll**). Властивість монтування систем файлів з'явилася у Windows лише з появою версії NTFS 5.0. Ранні версії також не підтримують зв'язки файлів. На рівні графічного робочого стола підтримуються клавішні комбінації швидкого виклику, але ці структури не мали відповідностей у самій системі файлів. Прикладна програма не могла ввійти у файл з другої директорії, не скопіювавши до неї весь файл. Починаючи з NTFS 5.0 до системи файлів були додані файлові зв'язки.

Основні функції для роботи з директоріями наведені в табл. 10. Мабуть сенсу розкривати їхнє значення немає в зв'язку з їхніми доволі зрозумілими іменами. У другому стовпці таблиці наведені еквіваленти з UNIX, якщо вони існують.

Таблиця 10 – Основні функції Win32 API для роботи з директоріями

Функція API	UNIX	Значення
CreateDirectlry	**mkdir**	Створює нову директорію
RemoveDirectory	**rmdir**	Видаляє порожню директорію
FindFirstFile	**opendir**	Ініціалізує читання елементів директорії
FindNextFile	**readdir**	Читає наступний елемент директорії
MoveFile		Переміщає файл із однієї директорії в іншу
SetCurrentDirectory	**chdir**	Змінює поточну директорію

Windows має більш складний механізм захисту, чим в UNIX. Коли користувач входить у систему, його процес одержує ***маркер доступу*** від операційної системи. Маркер доступу містить ***ідентифікатор безпеки*** (SID – Security ID), список груп, до яких належить користувач, наявні привілеї й деяку іншу інформацію. Маркер доступу концентрує всі відомості про захист в одному легко доступному місці. Всі процеси, створені цим процесом, успадковують цей же маркер доступу.

Дескриптор захисту – це один з параметрів, що привласнюється будь-якому об'єкту при його створенні. Дескриптор захисту містить список елементів, котрий називається ***списком контролю доступу*** (ACL – Access Control List). Кожен елемент дозволяє або забороняє робити певний набір операцій над об'єктом якій-небудь окремій людині або групі. Наприклад, файл може містити дескриптор захисту, який визначає, що Іванов не має доступу до файла взагалі, Петров може читати файл, Сидоров може читати й записувати файл, а всі члени групи XYZ можуть прочитати тільки розмір файла.

Якщо процес намагається виконати яку-небудь операцію над об'єктом з використанням ідентифікатора (handle), що він одержав при відкритті об'єкта, диспетчер безпеки одержує маркер доступу даного процесу й починає перебирати елементи списку контролю доступу один по одному. Як тільки він знаходить елемент, що відповідає власнику процесу або однієї із груп, інформація про дозвіл або заборону доступу, знайдена там, береться як задана. Із цієї причини елементи, що забороняють доступ, звичайно уміщуються у список контролю доступу перед елементами, що дозволяють доступ (щоб користувач, у якого немає доступу, не зміг одержати його незаконно, будучи членом однієї із груп, який доступ є дозволеним). Дескриптор захисту також містить інформацію, котра використовується для аудита доступів до об'єкта.

Слід також розглянути й питання того, як файли й директорії реалізуються в Windows. Кожен диск поділяється на томи, такі ж як розділи диска в UNIX. Кожен том містить файли, бітові відображення директорій і інші структури даних. Кожен такий том організований у вигляді лінійної послідовності кластерів. Розмір кластера фіксований для кожного тому. Він може бути від 512 байтів до 64 КіБ, залежно від розміру тому. Звертання до кластера здійснюється по зсуві від початку тому. При цьому використовуються 64-бітні числа.

Основною структурою даних на кожному томі є MFT (Master File Table – головна файлова таблиця), у якій утримується елемент для кожного файла й директорії в томі. Ці елементи аналогічні елементам індексного дескриптора (i-node) у системі UNIX. Головна файлова таблиця сама є файлом і може поміщатись в

будь-яке місце в межах тому. Це усуває проблему, що виникає при наявності зіпсованих блоків на диску в зоні розташування індексних дескрипторів.

Умовне зображення структури головної файлової таблиці показане на рис. 37. MFT починається із заголовка, у якому дається інформація про том (покажчики на кореневий каталог, файл завантаження, список осіб, що користуються вільним доступом і т.п.). Потім іде по одному елементу на кожен файл або директорію. Розмір такого елемента – 1 КіБ за винятком тих випадків, коли розмір кластера становить 2 КіБ і більше. Кожен елемент містить всі метадані (адміністративну інформацію) про файл або директорію. Допускається кілька форматів, один із яких як раз і зображений на рис. 37.

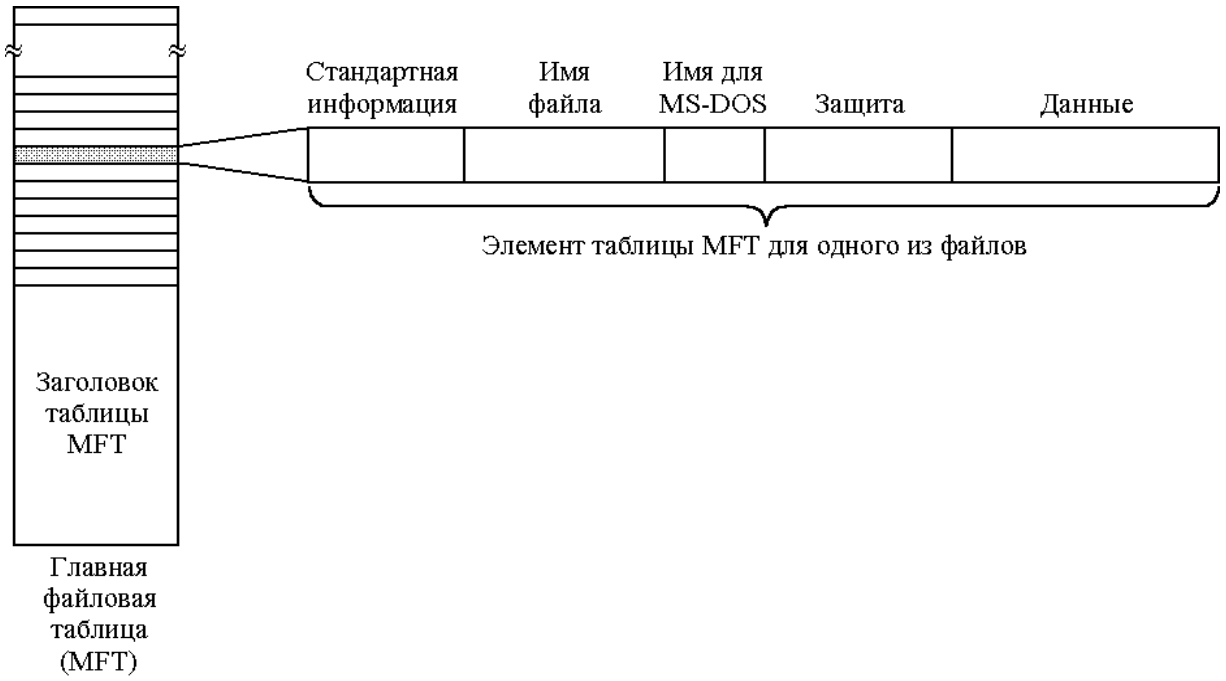

Рисунок 37 – Структура головної файлової таблиці в системі Windows

Поле стандартної інформації містить інформацію про позначки часу, наявність яких потребує стандарт POSIX, про число зв’язків файла, про біти «тільки для читання», бітах архівування й т.ін. Це поле має фіксовану довжину і є обов’язковим. Ім’я файла може мати будь-яку довжину до 255 символів Unicode. Щоб такі файли стали доступні для старих 16-бітних програм, вони можуть забез-

печуватися додатковим ім'ям MS-DOS, що складається максимум з 8 символів, за якими слідує точка й розширення з 3 символів. Якщо дійсне ім'я файла відповідає правилу найменування в MS-DOS (8+3), то друге ім'я для MS-DOS не використовується.

Далі слідує інформація про захист. У всіх версіях, аж до NTFS 4.0, у полі захисту втримувався дескриптор захисту. Починаючи з NTFS 5.0, вся інформація про захист уміщується до одного окремого файла, а поле захисту просто вказує на відповідну частину цього файла.

Для файлів невеликого розміру самі дані цих файлів утримуються в елементі головної файлової таблиці, що спрощує їхній виклик, – для цього не потрібно звертатися до диска. Дана ідея одержала назву ***безпосередній файл*** (immediate file). Для файлів великого розміру це поле містить покажчики на кластери, у яких утримуються дані або (що більш поширено) блоки послідовних кластерів, так що номер кластера і його довжина можуть представляти довільну кількість даних. Якщо елемент головної файлової таблиці недостатньо великий для зберігання потрібної інформації, до нього можна приєднати один або кілька додаткових елементів.

Максимальний розмір файла становить 2^{64} байтів. Щоб наочно уявити, що собою представляє файл у 2^{64} байтів можно скористуватись таким прикладом. Якщо представити, що файл був написаний у двійковій системі, а кожен 0 або 1 займає 1 мм простору. Довжина ланцюжка нулів і одиниць на 2^{64} мм склала б 15 світлових років! Цього вистачило б для того, щоб вийти за межі Сонячної системи, досягти Альфа Центавра й повернутися назад.

Файлова система NTFS має багато інших цікавих особливостей, у тому числі можливість компресії даних і відмовостійкість із застосуванням атомарних транзакцій. У теперішній час ця файлова система є однією з найнадійніших систем організації файлів у ОС.

4.5 Приклади керування процесами

Системи Windows і UNIX дозволяють поділити роботу на кілька процесів, які виконуються паралельно й взаємодіють один з одним, як у прикладі з виробником і споживачем, що обговорювався раніше. Зараз настала черга поговорити про те, як відбувається керування процесами в обох системах. Обидві системи підтримують не тільки паралелізм процесів, але й паралелізм у межах одного процесу з використанням потоків.

4.5.1 Керування процесами в системі UNIX

У будь-який момент процес у системі UNIX може створити підпроцес, що є його точною копією. Для цього виконується системний виклик **`fork`**. Первинний процес називається ***породжуючим процесом***, а новий – ***породженим процесом***. Два процеси, що отримуються в результаті виконання операції **`fork`**, абсолютно ідентичні й навіть розділяють ті самі файлові дескриптори. Але кожен із цих двох процесів виконує свою роботу незалежно від іншого.

Часто породжений процес певним чином дезорієнтує дескриптори файлів, а потім виконує системний виклик **`exec`**, що заміщає його програму й дані програмою й даними з деякого виконуваного файла, ім’я якого передане як параметр у виклик **`exec`**. Наприклад, якщо користувач друкує команду «xyz», то інтерпретатор команд (оболонка системи) виконує операцію **`fork`**, створюючи, таким чином, породжений процес (свою копію). А цей процес вже виконує процедуру **`exec`**, щоб запустити програму **`xyz`**.

Ці два процеси працюють паралельно, але іноді, породжуючий процес, повинен за якимись причинами чекати, щоб породжений процес завершив свою роботу, і тільки після цього продовжувати виконання тих або інших дій. У цьому випадку процес, що породжує, виконує системний виклик **`wait`** або **`waitpid`**, у результаті чого він тимчасово припиняється й чекає, поки породжений процес не виконає системний виклик **`exit`**.

Процеси можуть виконувати процедуру **fork** як завгодно часто, у результаті чого виходить ціле дерево процесів – рис. 38. Тут процес А виконав процедуру **fork** двічі й породив два нових процеси. Потім один з породжених процесів теж виконав процедуру **fork** двічі, а другий породжений процес виконав її один раз. Таким чином, вийшло дерево із шести процесів.

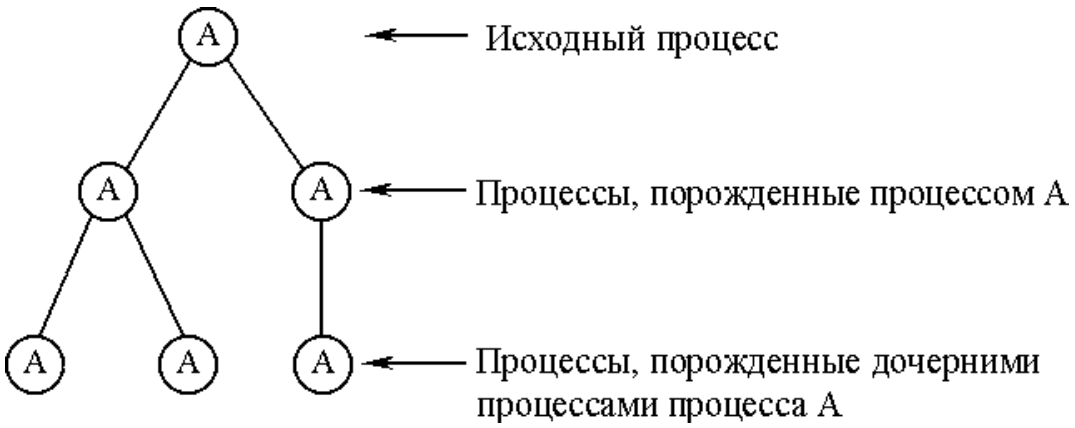

Рисунок 38 – Дерево процесів в системі UNIX

Процеси в системі UNIX можуть взаємодіяти один з одним через спеціальну інформаційну структуру, що називається ***каналом***. Канал являє собою вид буфера, у який один процес записує потік даних, а інший процес забирає ці дані звідти. Байти завжди вертаються з каналу в тім порядку, у якому вони були записані. Випадковий доступ неможливий. Канали не зберігають межі між відрізками даних, тому якщо один процес записав у канал 4 відрізки по 128 байтів, а інший процес зчитує дані по 512 байтів, то цей другий процес одержить всі дані відразу без вказівки на те, що вони були записані в кілька заходів.

У системах System V і Solaris застосовується ще й інший метод взаємодії процесів. Тут використовуються так звані ***черги повідомлень***. Процес може створити нову чергу повідомлень або відкрити вже існуючу за допомогою виклику **msgget**. Для відправлення повідомлень використовується виклик **msgsnd**, а для одержання – **msgrecv**. Повідомлення, відправлені таким способом, відрізняються від даних, що поміщаються в канал. По-перше, границі повідомлень зберіга-

ються, а в канал передається просто потік даних. По-друге, повідомлення мають пріоритети, тому термінові повідомлення йдуть перед всіма іншими. По-третє, повідомлення типізовані, і виклик **`msgrecv`** може визначати їхній тип, якщо це необхідно.

Два й більше процеси можуть розділяти загальну область своїх адресних просторів. UNIX управляє цією розділеною пам'яттю шляхом відображення тих самих сторінок у віртуальний адресний простір всіх розділених процесів. У результаті запис у загальну область, котра зроблена одним із процесів, буде видна всім іншим процесам. Цей механізм забезпечує дуже високу пропускну здатність між процесами.

Ще одна особливість System V і Solaris – наявність семафорів. Принципи їхньої роботи вже описані вище, коли мова йшла про виробника й споживача.

Системи UNIX можуть підтримувати кілька ***потоків керування*** в межах одного процесу. Ці потоки керування за звичаєм називають просто потоками. Вони схожі на процеси, які поділяють загальний адресний простір і всі об'єкти, що пов'язані із цим адресним простором (дескриптори файлів, ***змінні оточення*** й т.п.). Однак кожен потік має свій власний лічильник команд, свої власні регістри й свій власний стек. Якщо один з потоків припиняється (наприклад, поки не завершиться процес вводу-виводу), інші потоки в тім же процесі можуть продовжувати роботу. Два потоки в одному процесі, які діють як процес-виробник і процес-споживач, подібні до двох процесів з одним потоком, які розділяють сегмент пам'яті, що містить буфер, хоча і не ідентичні ім. У іншому випадку кожен потік має свої власні дескриптори файлів і т.ін., тоді як у першому випадку всі ці елементи є загальними.

Необхідність застосування потоків замість процесів можна проілюструвати на такому прикладі. Якщо до початку розробки сервера World Wide Web уявити собі цей сервер і алгоритм його роботи, то можна зробити висновок, що для прискорення роботи такий сервер повинен зберігати в основній пам'яті кеш часто використовуваних web-сторінок. Якщо потрібна сторінка перебуває в кеш-пам'яті, то вона може бути виданою клієнтові негайно. Але якщо web-сторінки в кеші не-

має, то вона буде викликана з диска. На жаль, на це потрібно досить тривалий час (за звичай не менше чим 20 мс), і на цей час процес блокується й не може обслуговувати нові запити, котрі надходять до нього, навіть якщо ці web-сторінки перебувають у кеш-пам'яті.

Із цієї причини було б слушно ухвалити рішення зробити кілька потоків в одному процесі, які розділяють загальну кеш-пам'ять web-сторінок. Якщо один з потоків блокується, нові запити можуть оброблятися іншими потоками. Запобігти блокуванню процесів можна й без використання потоків. Для цього буде потрібно мати кілька процесів, але тоді потрібно буде продублювати кеш, а це трохи марнотратно, оскільки розмір пам'яті обмежений.

Стандарт системи UNIX для потоків називається **`pthreads`**. Він визначається стандартом POSIX (P1003.1C) і містить системні виклики для керування потоками і їхньої синхронізації. У стандарті не визначено, управляються потоки ядром ОС, або вони повністю перебувають у користувальницькому просторі, тому різні ОС роблять це по-різному. Однак, частіш за все, потоки створюються у користувальницькому просторі. Найпоширеніші виклики для роботи з потоками наведені в табл. 11.

Мабуть є сенс розглянути ці виклики трохи докладніше. Перший виклик, **`pthread_create`**, створює новий потік. Після виконання цієї процедури в адресному просторі з'являється на один потік більше. Потік, що виконав свою роботу, повинен викликати **`pthread_exit`**. Якщо потоку потрібно почекати, поки інший потік не закінчить свою роботу, то це робиться за допомогою виклику **`pthread_join`**. Якщо очікуваний потік уже закінчив свою роботу, виклик **`pthread_join`** завершується негайно.

Потоки можна синхронізувати за допомогою спеціальних об'єктів, які називаються ***м'ютексами***. Ці об'єкти одержали назву м'ютекси (mutexes), оскільки вони використовуються для забезпечення взаємного виключення доступу до якого-небудь з ресурсів (англійською MUTual EXclusion означає «взаємне виключення»). Звичайно м'ютекс управляє яким-небудь ресурсом (наприклад, буфером, розділеним між двома потоками). Для того щоб у конкретний момент часу тільки

один потік міг одержувати доступ до загального ресурсу, потоки повинні замикати м'ютекс перед використанням ресурсу й відмикати його після завершення роботи з ним. Таким чином, можна уникнути стану гонок, оскільки цьому протоколу підлягають всі потоки. М'ютекси схожі на бінарні семафори (тобто семафори, які можуть мати тільки два значення: 0 або 1).

Таблиця 11 – Основні виклики стандарту POSIX для керування потоками

Виклик потоку	Значення
pthread_create	Створює новий потік в адресному просторі процедури, що зробила виклик
pthread_exit	Завершує потік
pthread_join	Чекає завершення потоку
pthread_mutex_init	Створює новий мьютекс
pthread_mutex_destroy	Видаляє мьютекс
pthread_mutex_lock	Блокує мьютекс
pthread_mutex_unlock	Знімає блокування з мьютекса
pthread_cond_init	Створює змінну умови
pthread_cond_destroy	Видаляє змінну умови
pthread_cond_wait	Чекає змінну умови
pthread_cond_signal	Знімає блокування з одного з потоків, що чекає змінну умови

М'ютекси можна створювати й руйнувати за допомогою викликів **pthread_mutex_init** і **pthread_mutex_destroy** відповідно. М'ютекс може перебувати в одному із двох станів: блокованому й неблокованому. Якщо потоку потрібно встановити блокування на незамкнений м'ютекс, він використовує **pthread_mutex_lock**, а потім продовжує роботу. Однак якщо потік намагається замкнути м'ютекс, що вже є замкненим, він блокується. Коли потік, що у цей момент використовує загальний ресурс, завершить роботу із цим ресурсом,

він повинен розблокувати відповідний м'ютекс за допомогою **pthread_mutex_unlock**.

М'ютекси призначені для короткочасного блокування (наприклад, для захисту загальної змінної). Вони не призначені для тривалої синхронізації (наприклад, для очікування, поки звільниться накопичувач на магнітній стрічці, або принтер). Для тривалої синхронізації існують ***змінні умови*** (condition variables). Ці змінні створюються й видаляються за допомогою викликів **pthread_ cond_init** і **pthread_cond_destroy** відповідно.

Якщо, наприклад, потік виявив, що накопичувач на магнітній стрічці, що йому потрібний, у цей момент зайнятий, цей потік робить системний виклик **pthread_cond_wait** над змінною умови. Коли потік, що використовує накопичувач на магнітній стрічці, завершує свою роботу із цим пристроєм (а це може відбутися через кілька годин), він посилає сигнал **pthread_cond_signal**, щоб розблокувати рівно один потік, що очікує цю змінну умови. Якщо жоден потік не очікує цю змінну, сигнал пропадає. У змінних умови (condition variables) немає лічильника, як у семафорів. Слід пам'ятати, що над потоками, м'ютексами й змінними умови можна виконувати ще й деякі інші операції.

4.5.2 Керування процесами в Windows

Windows підтримує кілька процесів, які можуть взаємодіяти й синхронізуватися. Кожен процес, на відміну від UNIX, вже при створенні містить, принаймні, один потік, котрий, у свою чергу, містить, принаймні, одну ***нитку*** (fiber) (це ***легковагий потік***). Процеси, потоки й нитки в сукупності забезпечують набір засобів для підтримки паралелізму й у машинах з одним процесором, і в багатопроцесорних системах.

Нові процеси створюються за допомогою функції API **CreateProcess**. Ця функція має 10 параметрів, кожен з яких має безліч опцій. Ясно, що така розробка набагато складніша за відповідну схему в UNIX, де **fork** взагалі не має параметрів, а **exec** має всього три параметри: покажчики на ім'я файла, який потрібно

виконати, на масив параметрів командного рядка й на рядки опису конфігурації. Нижче викладені 10 параметрів функції **CreateProcess**.

- Покажчик на ім'я виконуваного файла.
- Сам командний рядок.
- Покажчик на дескриптор захисту для даного процесу.
- Покажчик на дескриптор захисту для внутрішнього потоку.
- Біт, що повідомляє чи успадковує новий процес ідентифікатори (handles) вихідного процесу.
- Різні прапори (наприклад, помилка, пріоритет, налагодження, консоль).
- Покажчик на рядки опису конфігурації.
- Покажчик на ім'я поточного каталогу нового процесу.
- Покажчик на структуру, що описує вихідне вікно екрана.
- Покажчик на структуру, що повертає 18 значень процедурі, що зробила виклик.

В Windows, знову ж на відміну від UNIX, немає ніякої ієрархії типу «той, що породжує-породжений». Всі процеси створюються рівними. Але оскільки одним з 18 параметрів, що повертаються вихідному процесу, є ідентифікатор (handle) для нового процесу (а це дає можливість контролювати новий процес), тут існує внутрішня ієрархія з погляду того, що певні процеси володіють ідентифікаторами інших процесів. Ці ідентифікатори не можна просто безпосередньо передавати іншим процесам, але процес може зробити певний ідентифікатор доступним для іншого процесу, а потім передати йому цей ідентифікатор, так що внутрішня ієрархія процесів не може зберігатися довго.

Кожен процес в Windows створюється з одним потоком, але пізніше цей процес може створити ще кілька потоків. Створення потоку простіше, ніж створення процесу, оскільки **CreateThread** має всього 6 параметрів замість 10:

- дескриптор захисту;
- розмір стека;
- початкову адресу;
- обумовлений користувачем параметр;

− початковий стан потоку (готовий до роботи або блокований);

− ідентифікатор потоку.

Створення потоків робиться ядром (тобто потоки реалізуються не в користувальницькому просторі, як у деяких інших системах).

Коли ядро робить розподіл ресурсів, воно викликає не тільки процес, що повинен запускатися наступним, але й потік у цьому процесі. Це значить, що ядро завжди знає, які потоки блоковані, а які – ні. Тому що потоки є об'єктами ядра, вони, як і всі інші об'єкти ОС, мають дескриптори захисту й ідентифікатори. Оскільки ідентифікатор дозволяється передавати іншому процесу, можна зробити так, щоб один процес управляв потоками в іншому процесі. Ця особливість може знадобитися для програм налагодження.

Створювати потоки в Windows досить марнотратно, оскільки для створення потоку потрібно ввійти в ядро, а потім вийти з нього. Щоб уникнути цього, в Windows передбачені нитi (fibers), які схожі на потоки, але розподіляються в користувальницькому просторі програмою, що їх створює. Кожен потік може мати кілька ниток, точно так само як процес може мати кілька потоків, тільки в цьому випадку, коли нитка блокується, вона встає в чергу заблокованих ниток і вибирає іншу нитку для роботи у своєму потоці. Ядро не знає про цей перехід, оскільки потік однаково продовжує працювати, навіть якщо спочатку діяла одна нитка, а потім інша. Ядро управляє процесами й потоками, але не управляє нитками. Нитки можуть бути корисними, наприклад, у тому випадку, коли програми, які управляють своїми власними потоками, переносяться на платформу Windows.

Процеси можуть взаємодіяти один з одним різними способами: через ***канали***, ***іменовані канали***, ***поштові скриньки***, ***сокети*** (sockets), ***віддалені виклики*** процедур і ***загальні файли***. Канали бувають двох видів: ***байтові канали*** й ***канали повідомлень***. Тип вибирається під час створення. Байтові канали працюють так само, як у системі UNIX. Канали повідомлень зберігають границі повідомлень, тому чотири записи по 128 байтів будуть прочитані як чотири повідомлення по 128 байтів (а не як одне повідомлення на 512 байтів, як у випадку з байтовими каналами). Крім того, існують іменовані канали, які теж бувають двох видів. Іменовані канали можуть використовуватися в мережі, а звичайні канали – ні.

Поштові скриньки є тільки в Windows (у системі UNIX їх немає). Вони багато в чому схожі на канали, хоча не у всім. По-перше, вони однобічні, а канали – двосторонні. Їх можна використовувати в мережі, але вони не гарантують доставку. Нарешті, вони дозволяють відправляти повідомлення декільком одержувачам, а не тільки одному.

Сокети схожі на канали, але вони зазвичай зв'язують процеси на різних машинах. Однак їх можна застосовувати й для зв'язку процесів на одній машині. Загалом кажучи, зв'язок через сокет є ненабагато вигіднішим за зв'язок через звичайний або іменований канал.

Віддалені виклики процедур дозволяють процесу А наказати процесу В зробити виклик процедури в адресному просторі В від імені А и повернути результат процесу А. Тут існують різні обмеження на параметри. Наприклад, не має ніякого смислу передача покажчика іншому процесу.

Нарешті, процеси можуть розділяти загальну пам'ять шляхом відображення одночасно в той самий файл. Тоді всі записи, зроблені одним процесом, з'являються в адресному просторі інших процесів. Застосовуючи такий механізм, можна легко реалізувати розділений буфер, що був описаний вище в прикладі із процесом-виробником і процесом-споживачем.

Windows надає безліч механізмів синхронізації – семафори, мьютек-сы, ***критичні секції*** й ***події***. Всі ці механізми працюють з потоками, але не з процесами, тому, коли потік блокується на семафорі, це ніяк не впливає на інші потоки в цьому процесі – вони просто продовжують працювати.

Семафор створюється за допомогою функції API **`CreateSemaphore`**, що може встановити його на певне значення й визначити його максимальне значення. Семафори у Windows є об'єктами ядра, тому вони мають дескриптори захисту й ідентифікатори (handles). Ідентифікатор для семафора може дублюватися за допомогою **`DuplicateHandle`** і передаватися іншому процесу, тому на одному семафорі можна синхронізувати кілька процесів. Присутні функції **`up`** і **`down`**, хоча вони мають особливі назви: **`ReleaseSemaphore`** (up) і **`WaitForSingleObject`** (down). для функції **`WaitForSingleObject`** можна

визначити межу на час простою, і тоді потік, котрий звернувся до функції, зрештою може бути розблокований, навіть якщо семафор зберігає значення 0.

М'ютекси теж є об'єктами ядра, але вони простіше за семафори, оскільки в них немає лічильників. Вони, по суті, є об'єктами з функціями API для блокування (**WaitForSingleObject**) і розблокування (**ReleaseMutex**). Ідентифікатори м'ютексів, як і ідентифікатори семафорів, можна дублювати й передавати іншим процесам, так що потоки з різних процесів можуть мати доступ до того ж самого м'ютексу.

Третій механізм синхронізації заснований на критичних секціях, які подібні до м'ютексів, за винятком локальності стосовно адресного простору вихідного потоку. Оскільки критичні секції не є об'єктами ядра, у них немає ідентифікаторів (handles) і дескрипторів захисту і їх не можна передавати іншим процесам. Блокування й розблокування здійснюються за допомогою **EnterCriticalSection** і **LeaveCriticalSection** відповідно. Тому що ці функції API виконуються цілком у користувальницькому просторі, вони працюють набагато швидше за м'ютекси.

Останній механізм – пов'язаний з використанням об'єктів ядра, які називаються подіями. Якщо потоку потрібно дочекатися якоїсь події, він викликає **WaitForSingleObject**. Можна розблокувати один потік, що очікує, за допомогою **SetEvent** або всі потоки, що очікують, за допомогою **PulseEvent**. Існує кілька видів подій, які мають кілька опцій.

Події, м'ютекси й семафори можна назвати певним чином і зберігати в системі файлів як іменовані канали. Можна синхронізувати роботу двох і більше процесів шляхом відкриття той самої події, м'ютекса або семафора. У цьому випадку їм не потрібно створювати об'єкт, а потім дублювати ідентифікатори, хоча такий підхід теж можливий.

4.6 Питання для самоперевірки

1. Чому багато систем файлів вимагають, щоб файл перед прочитанням явно відкривався за допомогою системного виклику **open**?

2. Припустимо, що одна з версій UNIX використовує 2 кібі блоків на диску й зберігає 512 адрес диска на кожен блок непрямої адресації (звичайної непрямої адресації, подвійний і потрійний). Який буде максимальний розмір файла? Передбачається, що розмір покажчиків файла становить 64 біта.
3. Припустимо, що системний виклик UNIX

   `unlink(«/usr/ast/bin/game3»)`

   виконано у контексті рис. 33. Опишіть докладно, які зміни відбудуться в системі директорій.
4. Уявіть, що вам потрібно реалізувати систему UNIX на мікрокомп'ютері, де основної пам'яті недостатньо. Після довгої роботи вона усе ще цілком не вміщується у пам'ять, і ви вибираєте системний виклик навмання, щоб пожертвувати їм для загального добра. Ви вибрали системний виклик **`pipe`**, що створює канали для передачі потоків байтів від одного процесу до іншого. Чи можливо після цього якось змінити ввід-вивід? Що ви можете сказати про конвеєри? Розгляньте проблеми й можливі рішення.
5. Комісія із захисту дескрипторів файлів висунула протест проти системи UNIX, тому що коли ця система повертає дескриптор файла, вона завжди повертає найменший номер, який у цей момент не використовується. Отже, навряд чи коли-небудь будуть використовуватися дескриптори файлів з більшими номерами. Комісія наполягає на тому, щоб система повертала дескриптор із самим маленьким номером з тих, які ще не використовувалися програмою, а не з тих, які не використовуються в цей момент. Комісія стверджує, що цю ідею легко реалізувати, це не вплине на існуючі програми й, крім того, це буде набагато справедливіше стосовно дескрипторів. А що ви думаєте із цього приводу?
6. У системі Windows можна скласти список керування доступом таким чином, щоб один користувач не мав доступу до жодного з файлів, а всі інші мали повний доступ до них. Як це можна реалізувати?
7. Опишіть два способи програмування роботи процесора-виробника й процесора-споживача з використанням загальних буферів і семафорів в Windows. Подумайте про те, як можна реалізувати розділений буфер у кожному із

двох випадків.

8. Створіть п’ять різних шляхів до файла **/etc/passwd**. Підказка: використовуйте елементи каталогу «.» і «..».
9. Чи є необхідним системний виклик **open** у системі UNIX? Якими будуть наслідки його відсутності?
10. Деякі системи дозволяють відображати частину файла на пам’ять. Які обмеження повинні накладатись на таку систему? Як реалізується таке часткове відображення файла на пам’ять?
11. У системах UNIX і Windows довільний доступ до файла здійснюється за допомогою спеціального системного виклику, що переміщає покажчик поточної позиції у файлі на нове місце. Запропонуйте альтернативний метод реалізації довільного доступу без використання цього системного виклику.
12. Розгляньте дерево каталогів на рис. 33. Якщо **/usr/jim** є робочим каталогом, як буде виглядати абсолютний шлях для файла з відносним шляхом **../ast/x**?
13. Деяким цифровим споживчим пристроям потрібно зберігати дані, наприклад, у вигляді файлів. Назвіть сучасний пристрій, для якого зберігання даних у вигляді безперервних файлів є прекрасною ідеєю.
14. Якщо **i-node** містить 10 дискових адрес, по 4 байт кожен, а всі дискові блоки мають розмір 1024 байт, чому дорівнює максимальний розмір файла?
15. Було запропоновано збільшити продуктивність і ефективність використання дискового простору за допомогою зберігання коротких файлів прямо в **i-node**. Скільки даних може зберігатися усередині **i-node**?
16. Дві студентки з факультету кібернетики, Кэролин і Элинор, обговорюють **i-node**. Кэролин стверджує, що пам'ять стала такою великою й дешевою, що при відкритті файла простіше й швидше зчитати нову копію **i-node** у таблицю **i-node**, чим шукати цей **i-node** по всій таблиці. Элинор не згодна. Хто має рацію?
17. Що відбудеться, якщо бітовий масив або список вільних блоків виявиться повністю загублений у результаті збою? Чи є спосіб відновлення від такого

збою або з диском можна попрощатися? Обґрунтуйте свою відповідь окремо для файлових систем UNIX і FAT-16.

18. Було запропоновано зберігати першу частину кожного файла системи UNIX у тім же дисковому блоці, що і його **i-node**. Які переваги такого підходу?
19. Скільки знадобиться дискових операцій для зчитування **i-node** файла **/usr/ast/courses/os/handout.t**? Припустимо, що **i-node** кореневого каталогу перебуває в оперативній пам'яті, але більше нічого, що відноситься до цього шляху, у пам'яті немає. Крім того, припустіть, що всі каталоги займають по одному блоку диска.
20. У багатьох версіях системи UNIX **i-node** зберігаються на початку диска. Альтернативний дизайн полягає у виділенні **i-node** блоку в момент створення файла й вміщенні цього блоку на початку першого блоку файла. Обговорите переваги й недоліки обох підходів.
21. Напишіть програму, що змінює порядок байтів у файлі так, що останній байт стає першим і навпаки. Програма повинна працювати з файлами довільної довжини, однак постарайтеся зробити програму доволі ефективною.
22. Напишіть програму, яка починає роботу в заданому каталозі й спускається по дереву каталогів, записуючи на шляху розміри всіх файлів, що зустрілися їй. Закінчивши сканування каталогу, програма повинна роздрукувати гістограму розмірів файлів, використовуючи крок гистограми як параметр (наприклад, при кроці 1024, файли розміром від 0 до 1023 байт попадають в один інтервал, від 1024 до 2047 байт – у наступний інтервал і т.д.).
23. Напишіть програму, яка сканує всі каталоги файлової системи UNIX і знаходить всі **i-node** із двома й більше твердими зв’язками. Для кожного такого файла програма повинна друкувати список всіх імен файлів, що вказують на цей файл.
24. Напишіть нову версію програми **ls** для системи UNIX. Ця версія повинна брати за як параметр одне або кілька імен каталогів і для кожного каталогу видавати список всіх файлів, що втримуються в цьому каталозі, по одному рядку на файл. Кожне поле повинне форматуватися відповідно до його типу. Вкажіть у списку тільки першу дискову адресу або не вказуйте ніяких.

ВИСНОВКИ

Операційна система є інтерпретатором певних особливостей комп'ютера, як системи, яких немає на рівні команд процесора. Головними серед них є віртуальна пам'ять, віртуальні команди вводу-виводу й засоби паралельної обробки.

Віртуальна пам'ять потрібна для того, щоб дозволити програмам використовувати більше адресного простору, чим є в машини насправді, або щоб забезпечити зручний механізм для захисту й поділу пам'яті. Віртуальну пам'ять можна реалізувати у вигляді чистої розбивки на сторінки, чистої сегментації або того й іншого разом. При сторінковій організації пам'яті адресний простір розбивається на рівні за розміром віртуальні сторінки. Одні з них відображаються на фізичні сторінкові кадри, інші – ні. Відсилання до відображеної сторінки перетворюється контролером керування пам'яттю на правильну фізичну адресу. Звертання до невідображеної сторінки спричиняє помилку через відсутність сторінки. Процесори Pentium і UltraSPARC мають складні контролери керування пам'яттю, які підтримують віртуальну пам'ять і сторінкову організацію.

Найважливішою абстракцією вводу-виводу на рівні операційної системи є файл. Файл складається з послідовності байтів або логічних записів, які можна читати й записувати, не знаючи при цьому про те, як працюють диски й інші пристрої вводу-виводу. Доступ до файлів може здійснюватися послідовно, непослідовно за номером запису й непослідовно за ключем. Для угруповання файлів використовуються директорії. Файли можуть зберігатися в послідовних секторах, а можуть бути розкидані по диску. В останньому випадку потрібні спеціальні структури даних для знаходження всіх блоків файла. Щоб стежити за вільним простором на диску, можна використовувати список або бітове відображення.

У системах часто підтримується паралельна обробка. Для цього шляхом поділу часу в одному процесорі моделюється кілька процесів. Неконтрольована взаємодія різних процесів може призвести до стану гонок. Щоб уникнути їх, уводяться спеціальні засоби синхронізації. Найпростішими з них є семафори.

UNIX і Windows є складними операційними системами. Обидві системи підтримують сторінкову організацію пам’яті й відображувані на пам’ять файли. Крім того, вони підтримують ієрархічні системи файлів, у яких файли складаються з послідовності байтів. Нарешті, обидві системи підтримують процеси й потоки і забезпечують механізми їхньої синхронізації.

ПЕРЕЛІК РИСУНКІВ

Рисунок 1 – Розташування рівня операційної системи 5
Рисунок 2 – Віртуальні адреси з 4096 до 8191 відображаються у адреси основної пам'яті від 0 до 4095 11
Рисунок 3 – а) перші 64 КіБ віртуального адресного простору поділені на 16 сторінок по 4 КіБ кожна; б) 32 КіБ основної пам'яті поділені на 8 сторінкових кадрів по 4 КіБ кожен 14
Рисунок 4 – Формування адреси основної пам'яті з адреси віртуальної пам'яті....16
Рисунок 5 – Можливе відображення перших 16 віртуальних сторінок до основної пам'яті, що містить 8 сторінкових кадрів.....17
Рисунок 6 – Ситуація, у якій алгоритм LRU не діє.....21
Рисунок 7 – Схематичне представлення одновимірного адресного простору, який містить таблиці, що постійно змінюються 28
Рисунок 8 – Сегментна модель пам'яті для приклада з компілятором.....29
Рисунок 9 – Динаміка зовнішньої фрагментації (а, б, в, г); видалення зовнішньої сегментації шляхом ущільнення (д).....33
Рисунок 10 – Структура дескриптора системи MULTICS 37
Рисунок 11 – Перетворення у системі MULTICS віртуальної адреси, яка складається з двох частин, на адресу фізичної пам'яті.....37
Рисунок 12 – Бітова структура селектора процесора Pentium 39
Рисунок 13 – Бітова структура дескриптора сегмента в процесорі Pentium 40
Рисунок 14 – Перетворення пари (селектор, зміщення) на лінійну адресу.....41
Рисунок 15 – Відображення лінійної адреси на фізичну у процесорі Pentium 42
Рисунок 16 – Рівні захисту пам'яті у процесорі Pentium 44
Рисунок 17 – Відображення віртуальних адрес на фізичні в комп'ютері UltraSPARC 47
Рисунок 18 – Структури даних, що застосовуються для трансляції віртуальної адреси у комп'ютері UltraSPARC II: буфер швидкого перетворення адреси TLB (а); буфер зберігання перетворень TSB (б); таблиця трансляції (в).....49

Рисунок 19 – Читання файла, який складається з логічних записів: до читання запису а) й після читання запису б) 64
Рисунок 20 – Варіанти розташування файла на диску: а – файл займає послідовні сектори; б – непослідовні сектори 65
Рисунок 21 – Два способи відстеження вільних секторів: список вільної пам'яті (а) і бітове відображення (б) 67
Рисунок 22 – Користувальницька директорія (а); можливий вміст одного з елементів директорії (б) 70
Рисунок 23 – Паралельна обробка з декількома процесорами (а); моделювання паралельної обробки шляхом перемикання одного процесора з одного процесу до іншого (в даному разі лише три процеси) (б) 75
Рисунок 24 – Пояснення роботи кільцевого буфера 77
Рисунок 25 – Код паралельної обробки зі станом гонок 78
Рисунок 26 – Ситуація, при якій механізм взаємодії виробника и споживача не працює 81
Рисунок 27 – Паралельна обробка з використанням семафорів 85
Рисунок 28 – Структура типової системи UNIX 94
Рисунок 29 – Рівні й основні функції системи вводу-виводу 94
Рисунок 30 – Структура ОС Windows 99
Рисунок 31 – Адресний простір одного процесу UNIX 106
Рисунок 32 – Фрагмент програми для копіювання файла з використанням системних викликів UNIX 114
Рисунок 33 – Умовне позначення частини системи директорій в операційній системі UNIX 115
Рисунок 34 – i-node, що розміщені на початку диска (а); диск, поділений на групи циліндрів, кожна зі своїми власними блоками та i-node (б) 118
Рисунок 35 – Схема використання поля i-node для адресації блоків на диску 122
Рисунок 36 – Фрагмент програми для копіювання файла із застосуванням функцій Win32 API 126
Рисунок 37 – Структура головної файлової таблиці в системі Windows 129
Рисунок 38 – Дерево процесів в системі UNIX 132

ПЕРЕЛІК ТАБЛИЦЬ

Таблиця 1 – Порівняння сторінкової організації пам’яті й сегментації 32
Таблиця 2 – Результати операцій над семафором 84
Таблиця 3 – Системні виклики UNIX 92
Таблиця 4 – Деякі розходження між версіями Windows 105
Таблиця 5 – Основні функції API для керування віртуальною пам’яттю в системі Windows NT 110
Таблиця 6 – Основні системні виклики UNIX 113
Таблиця 7 – Основні виклики для роботи з директоріями в системі UNIX 117
Таблиця 8 – Структура запису для елемента директорії 118
Таблиця 9 – Основні функції Win32 API для вводу-виводу файлів (у другому стовпчику дається еквівалент із UNIX) 125
Таблиця 10 – Основні функції Win32 API для роботи з директоріями 127
Таблиця 11 – Основні виклики стандарту POSIX для керування потоками 135

Навчальне електронне видання

РОЛЬЩИКОВ ВАДИМ БОРИСОВИЧ

ОПЕРАЦІЙНІ СИСТЕМИ

Конспект лекцій

Видавець і виготовлювач
Одеський державний екологічний університет
вул. Львівська, 15, м. Одеса, 65016
тел./факс: (0482) 32-67-35
E-mail: info@odeku.edu.ua
Свідоцтво суб'єкта видавничої справи
ДК № 5242 від 08.11.2016