МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

В. Г. Зайцев, І. П. Дробязко

ОПЕРАЦІЙНІ СИСТЕМИ

ЛАБОРАТОРНИЙ ПРАКТИКУМ

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для студентів,
які навчаються за спеціальністю 123 «Комп'ютерна інженерія»

Київ
КПІ ім. Ігоря Сікорського
2018

Рецензенти: *Жабін В. П.*, д-р техн. наук, проф.
Забара С. С., д-р техн. наук, проф.

Відповідальний редактор *Тарасенко В. П.*, д-р техн. наук, проф.

Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 9 від 24.05.2018 р.) за поданням Вченої ради факультету прикладної математики (протокол № 9 від 21.05.2018 р.)

Електронне мережне навчальне видання

Зайцев Володимир Григорович, д-р техн. наук, проф.
Дробязко Ірина Павлівна, ст. викл.

ОПЕРАЦІЙНІ СИСТЕМИ
ЛАБОРАТОРНИЙ ПРАКТИКУМ

Операційні системи: Лабораторний практикум [Електронний ресурс]: навч. посіб. для студ. спеціальності 123 «Комп'ютерна інженерія» / В. Г. Зайцев, І. П. Дробязко; КПІ ім. Ігоря Сікорського. – Електронні текстові дані (1 файл: 1,2 Мбайт). – Київ: КПІ ім. Ігоря Сікорського, 2018. – 88 с.

Навчальний посібник розроблено для ознайомлення студентів з типовими підходами до побудови і особливостями функціонування основних компонентів операційних систем і набуття практичного досвіду з їх проектування та застосування. Навчальний посібник складається зі вступу та 4 розділів, кожен з яких присвячено певній лабораторній роботі. Для кожної лабораторної роботи наведено мету, завдання, теоретичні відомості та методичні вказівки щодо її виконання і результатів, також надано контрольні питання для самоперевірки та список рекомендованої літератури. Навчальне видання призначене для студентів, які навчаються за спеціальністю 123 «Комп'ютерна інженерія» факультету прикладної математики КПІ ім. Ігоря Сікорського.

ЗМІСТ

ВСТУП

Дисципліна «Інженерія програмного забезпечення» є спеціальною дисципліною циклу професійної та практичної підготовки бакалаврів за спеціальністю 123 «Комп'ютерна інженерія». Її кредитний модуль «Операційні системи» ознайомлює студентів з фундаментальними концепціями та принципами побудови операційних систем (ОС). Одним з важливих видів індивідуальних завдань, що передбачені навчальною програмою модуля та виконуються студентами при його вивченні, є лабораторні роботи.

Завданням циклу лабораторних робіт є закріплення отриманих студентами теоретичних знань щодо загальних підходів, особливостей побудови і функціонування основних компонентів ОС і набуття ними практичного досвіду з їх проектування та застосування при вирішенні типових інженерних задач.

Цикл лабораторних робіт складається з 4-х робіт, побудованих на єдиній методичній основі. Лабораторні роботи відображають різні аспекти побудови ОС та присвячені вивченню методів реалізації основних компонентів, а саме: засобів синхронізації процесів і потоків, планувальників процесора, управління пам'яттю, файлової системи.

Кожна робота виконується на 2-х заняттях. Студент виконує індивідуальне завдання згідно свого варіанта, що узгоджується з викладачем.

При виконанні лабораторних робіт студенти використовують сучасні технології, мови та середовища програмування, які обираються студентами самостійно з міркувань найбільш ефективної реалізації завдання.

План виконання лабораторних робіт

1. Ознайомлення з метою, завданням, теоретичними відомостями лабораторної роботи та матеріалами з відповідних розділів дисципліни.
2. Ознайомлення з конкретним варіантом завдання, особливостями та по-послідовністю виконання роботи. Номер варіанту завдання має відповідати номеру студента у списку групи. Якщо кількість варіантів завдання менша ніж кількість студентів у групі, номер завдання визначається за циклічним принципом.
3. Виконання завдання, використовуючи надані рекомендації.
4. Тестування програми та виправлення помилок за його результатами.
5. Підготовка відповідей на контрольні запитання.
6. Підготовка звіту з лабораторної роботи.
7. Демонстрація викладачеві результатів виконання роботи.

Загальні вимоги до оформлення звіту та захисту лабораторної роботи

Результати виконання лабораторної роботи подаються у вигляді працюючої програми (файла .exe) на диску і протоколу, що містить: титульний лист, загальне завдання до лабораторної роботи, завдання за варіантом, лістинг програми обраною мовою програмування та результуючі ілюстративні матеріали.

При захисті лабораторної роботи студент повинен бути готовим відповісти на будь-яке запитання щодо виконання роботи, змісту програми та на контрольні запитання. Студент повинен:

- знати основні теоретичні положення та вміти їх формулювати;
- продемонструвати працездатність створених програм на ПЕОМ;
- довести раціональність кодування та за допомогою тестів коректність розробленого алгоритму алгоритму;
- продемонструвати здатність аналізувати отримані результати та робити висновки.

ЛАБОРАТОРНА РОБОТА №1

Планування процесів

Метою лабораторної роботи є ознайомлення з основними алгоритмами планування процесів в операційних системах і моделювання роботи планувальника (побудова програмної моделі) за певним, конкретно заданим алгоритмом.

Основні теоретичні відомості

Планування процесів

Ефективність роботи обчислювальної системи залежить безпосередньо не тільки від ефективності алгоритмів обробки інформації і технічних характеристик обчислювальної системи, але і від прийнятих в системі правил виконання робіт, прийому і обробки заявок користувачів на виконанння певної роботи (програми).

При вивченні дисциплін обслуговування заявок передбачається, що процедури введення і обслуговування заявок є незалежними. Заявка, яка надходить до системи, починає обслуговуватися негайно, якщо у цей момент часу ресурс для її обслуговування вільний. Якщо ж ресурс зайнятий обслуговуванням попередніх заявок, тоді приймається рішення або перервати виконувану в цей час програму і обслуговувати нову заявку, або остання має очікувати своєї черги. Рішення залежить від типу заявки, яка надійшла, і прийнятого в системі правила (дисципліни) обслуговування, У разі переривання виконання заявки, передбачається повернення її в чергу з подальшим очікуванням на продовження перерваного обслуговування. Таким чином, тривалість перебування кожної заявки у обчислювальній системі складається з *часу очікування заявки* і *часу її обслуговування* (виконання), разом – *час обігу* або *повний час виконання*.

Розподіл ресурсів між заявками, що мають виконуватись і які є в наявності, носить назву *планування*. Найважливішим ресурсом є процесор (*процесорний час*).

Планування процесів (process scheduling) означає визначення процесу, якому далі потрібно надати процесорний час. Зокрема, це означає ухвалення рішення, коли і який процес повинен переходити із стану *Готовий* в стан *Виконуваний*. У системі з одним процесором тільки один процес може виконуватися в кожний момент часу. Решта процесів повинна чекати надання їм процесора у *черзі* (чергах).

Є ОС, в яких визначені два типи роботи. Більша одиниця – *процес*, що вимагає для своєї реалізації дещо дрібніших робіт, і ця дрібніша одиниця називається *потоком*. При реалізації потоків з'являється можливість організації паралельних обчислень у рамках процесу. Справа в тому, що програми, виконувані в рамках одного процесу, можуть мати внутрішній паралелізм, що у принципі може прискорити час виконання процесу. Із цього витікає, що в ОС поряд з механізмами керування процессами, був би потрібний інший механізм розпаралелювання обчислень, що враховував би тісні зв'язки між окремими гілками обчислень того ж самого додатка. Для цього у ряді сучасних ОС використовується механізм *багатопоточної обробки*. При цьому вводиться нова одиниця роботи – *потік виконання*, а поняття «процес» до деякої міри змінює свій зміст.

Поняттю «потік» відповідає послідовний перехід процесора від однієї команди програми до іншої. ОС розподіляє процесорний час між потоками, а процесу ОС призначає адресний простір і набір ресурсів, які спільно використовуються всіма його потоками. У подальшому будемо ототожнювати поняття процесу і потоку, тобто будемо вважати, що кожен процес має у своєму складі тільки один потік.

Існує багато типів черг. Нові заявки знаходяться у *вхідній черзі*, що часто зветься *чергою робіт – завдань* (*job queue*), та очікують звільнення ресурсу. Вибір процесу, який буде активізований (тобто, який повинен виконуватися

наступним), здійснюється відповідним системним процесом – *планувальником* **(***scheduler***)** відповідно до прийнятих *алгоритмів планування* (*дисциплін обслуговування черг*). Процес, який реалізує рішення планувальника – *диспетчер* (*dispatcher*) – передає процесор обраному планувальником процесу, тобто перемикає процесор на рішення іншої задачі.

Розрізняють два типи дисциплін обслуговування: *безпріоритетні та пріоритетні.*

У разі дисциплін *без пріоритетів* заявки різних типів не мають наперед встановлених пріоритетів для обслуговування і вважаються при вході в систему *рівнопріоритетними*. При реалізації *пріоритетних* дисциплін окремим процесам надається привілейоване право переходу до стану виконання. Пріоритет, який присвоєно процесу, може мати стале значення або змінюватись в процесі його виконання. У деяких системах визначаються класи пріоритетів. Це звичайно робиться за рахунок організації окремих черг для кожного класу. Ресурс буде наданий у першу чергу тим заявкам, котрі знаходяться у черзі (класі) з найбільш високим пріоритетом.

Алгоритм обслуговування називають *невитісняльним* (*non-preemptive*), якщо планувальник не призупиняє процеси, що виконуються, і активний процес виконується доти, поки сам не віддасть управління диспетчеру задач для вибору із черги іншого процесу, який готовий до виконання. Тобто рішення приймаються тільки тоді, коли виконуваний процес переходить із стану виконання в стан очікування, готовності або коли процес закінчився.

Алгоритм обслуговування називають *витісняльним* (*preemptive*), якщо планувальник може перервати процес, що виконується. Рішення про перемикання процесора з виконання одного процесу на виконання іншого приймається планувальником, а не активним процесом.

Механізм диспетчеризації задач цілком зосереджено в операційній системі і програміст може писати своє програмне забезпечення, не турбуючись про його виконання разом з іншими процесами.

Нижче розглядаються найуживаніші алгоритми планування.

Алгоритми планування

FCFS (First Come First Serve) – першим прийшов – першим обслуговується

Це – невитісняльна стратегія. Процеси виконуються у порядку їх надходження до системи, тобто і-й процес може бути запущений на виконання тільки після завершення (і-1)-го процесу. Якщо в момент надходження і-го процесу (і-1)-й процес ще не закінчився, тоді і-й процес переходить у стан очікування до моменту вивільнення процесора.

LCFS (Last Come First Serve) або LIFO – першим обслуговується останній, який надійшов

Це – невитісняльна стратегія. Процес, що надійшов, буде виконуватися, якщо процесор незайнятий. Інакше він надходить до черги і очікує там моменту вивільнення процесора. Якщо в черзі є декілька процесів, тоді при вивільненні процесора на виконання обирається той процес з черги, що надійшов останнім.

SJF (Shortest Job First) – найкоротша робота виконується першою

Це – стратегія, в якій рішення планувальника залежать від очікуваного часу виконання кожного із процесів, які знаходяться у черзі готових процесів. Розглянемо невитісняльну версію цієї стратегії. Коли процес залишає процесор, планувальник вибирає з черги готових процесів процес із найкоротшим очікуваним часом виконання (приклад наведено в табл. 1.1).

Таблиця 1.1 – Черга готових процесів

Процес	Час надходження	Очікуваний час виконання
P1	0	8
P2	2	5
P3	4	1
P4	5	2

Припустимо, що ми маємо тільки один готовий процес *P1* у момент 0, і він зараз виконується. У момент часу 8 (коли *P1* залишає процесор) всі інші процеси вже готові, тому планувальник вибирає процес з найкоротшим очікуваним часом виконання, яким є *P3*. Коли *P3* закінчує, планувальник вибирає *P4*, а потім *P2* (табл. 1.2).

Таблиця 1.2 – Послідовність виконання (алгоритм SJF)

Обраний процес	*P1*	*P3*	*P4*	*P2*
Час виконання	8	1	2	5

Процес *P1* чекав 0 одиниць часу, *P2* чекав 9, *P3* чекав 4, і *P4* чекав теж 4. Отже, середній час очікування в черзі становить (0 + 9 + 4 + 4)/4 = 4.25 одиниць часу.

***SRTF* (*S*hortest *R*emaining *T*ime *F*irst) –** *першим виконується процес з найкоротшим часом виконання, що залишився)*

Так іноді називають версію SJF з витісненням. Першою ініціюється процес, який має найкоротший час, що залишився для його виконання. Звичайно, в цій стратегії рішення планувальника ухвалюються в час завершення процесу, що займав процесор. Однак, це також здійснюється, коли надходить новий процес. Тому це – витісняльна стратегія. Коли під час виконання процесу з'являється інший процес з меншим очікуваним часом виконання, що залишився, поточний виконуваний процес витісняється, і замість нього обирається на виконання процес, що надійшов.

Подивіться приклад попереднього алгоритму. Коли процес *P2* надходить в 2, час виконання *P1*, що залишається, дорівнює 6, а очікуваний час для *P2* – відповідно 5. Тому процес *P1* залишає процесор, і виконуваним процесом стає *P2*. Також, коли з'являється процес *P3*, процес *P2* витісняється. У момент часу 5, коли *P3* закінчився, надходить *P4* з очікуваним часом виконання 2. Таким чином, він, а не *P1* або *P2*, стає черговим виконуваним процесом. Після завершення *P4*, почергово оброблюються *P2* і *P1* (табл.1.3).

Таблиця 1.3 – Послідовність виконання (алгоритм SRTF)

Час виконання	**2**	**2**	**1**	**2**	**3**	**6**
Обраний процес	*P1*	*P2*	*P3*	*P4*	*P2*	*P1*
Час, що залишився	6	3	0	0	0	0

Час очікування складав: для *P1* – 0 + 6, для *P2* – 0 + 3, для *P3* – 0 і для *P4* також 0. Отже, середній час очікування дорівнює: (6 + 3 + 0 + 0)/4 = 2.25.

RR (Round Robin) *– кругове (карусельне) планування*

За цією стратегією, кожен процес почергово отримує у використання невеликий проміжок *(квант)* процесорного часу. Звичайно, він складає близько 100 мс. Таким чином, процеси не повинні чекати на свій квант довше, ніж довжина кванту, що помножена на кількість наявних у системі процесів. Оскільки для виконання може бути не досить одного кванту, далі процеси почергово знову отримуватимуть по кванту і т.д. до їх завершення.

Розглянемо кругове планування на прикладі, що використовувався при обговоренні SJF. Припустимо, що квант дорівнює 2 одиницям часу. Нижче (табл. 1.4) показана послідовність виконання процесів (готові процеси виконуються в порядку FIFO).

Таблиця 1.4 – Послідовність виконання (алгоритм RR)

Час виконання	2	2	2	1	2	2	2	1	2
Процес	*P1*	*P2*	*P1*	*P3*	*P2*	*P4*	*P1*	*P2*	*P1*
Час, що залишився	6	3	4	0	1	2	0	0	0

Періоди очікування становили: для *P1* – 2 + 5 + 1, для *P2* – 3 + 4, для *P3* – 2 і для *P4* – 4. Звідси, середній час очікування дорівнює (8 + 7 + 2 + 4)/4 = 5.25.

Багаторівневі черги *(2 рівні)*

Усі процеси можна обробляти однаково, але це недоцільно. Якщо ми відрізняємо в системі, наприклад, *фонові* (*пакетні*) процеси і процеси *переднього плану* (*інтерактивні процеси*), тоді можна для кожної з цих груп використовувати свою стратегію планування, що відповідає саме цій групі. Для роботи у фоновому режимі, кращими є стратегії SJF і SRTF, а для інтерактивної роботи – RR.

Фонові процеси обслуговуються згідно стратегії SJF, оскільки при роботі у пакетному режимі ми піклуємося про мінімізацію середнього часу очікування. Процеси переднього плану обслуговуються згідно RR стратегії, оскільки при інтерактивній роботі потрібно мінімізувати час відгуку.

Залишається знайти співвідношення між обома групами процесів. Звичайно, процесам переднього плану потрібно дати привілеї (тоді ми мінімізуємо час відгуку). Проте, наскільки? Можна припустити, що ми завжди виконуємо процеси переднього плану перед пакетними у надії, що завжди існують періоди неактивності користувача. Якщо, проте, ми побоюємося голодування фонових процесів, можна розподілити час між чергою фонових процесів і процесів переднього плану. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу.

Черги із зворотним зв'язком

Черг такого типу може бути більш ніж дві, і процеси можуть переміщуватися між ними. Перехід між чергами називають зворотнім зв'язком. Визначення стратегії планування з багатьма чергами і зворотним зв'язком повинне включати:

- кількість черг;
- стратегію планування для кожної черги;
- спосіб визначення процесу, що переміщується в іншу чергу;
- спосіб визначення черги, в яку повинен бути поміщений новий процес (у завданні будемо вважати, що черга вже визначена!).

Уявимо собі, наприклад, стратегію з трьома чергами Q0, Q1 і Q2. Умови наступні:

1) Черга Q0 має вищий пріоритет у порівнянні з іншими (тобто, якщо процес знаходиться в Q0, тоді процеси з решти черг не беруться до уваги).
2) Черга Q1 має вищий пріоритет, ніж Q2.
3) Черга Q0 обслуговується згідно RR алгоритму з квантом 50 мс.
4) Черга Q1 обслуговується згідно RR алгоритму з квантом 100 мс.
5) Черга Q2 обслуговується згідно FCFS алгоритму.
6) Новий процес надходить в Q0.
7) Якщо процес з черги Q0 не завершується після надання йому кванта часу (50 мс), він витісняється і передається в чергу Q1.
8) Якщо процес з черги Q1 не завершується після надання йому кванта часу (100 мс), він витісняється і передається в чергу Q2 і звідти передається на виконання.
9) Не можна повернутися в чергу з вищим пріоритетом.

Пріоритетне планування

Кожному процесу може бути наданий певний пріоритет, і планувальник завжди повинен серед процесів обрати процес з найвищим пріоритетом. Будемо вважати, що є кілька черг готових до виконання процесів. Черги впорядковані за спаданням пріоритетів (в межах черги – один пріоритет).

При виборі процесу планувальник починає з черги Q0, що має найвищий пріоритет. Якщо черга не порожня, процес вибирають згідно деякої стратегії, наприклад, FCFS. Якщо ж черга порожня, тоді розглядається черга Q1 і т.д.

Зниження пріоритетів

Одним із шляхів надання переваг коротким процесам є застосування штрафних санкцій до процесів, які довго виконуються. Ось як цього можна досягнути. Виконується витісняльне (за квантами часу) планування з використанням динамічного механізму. При надходженні процес потрапляє у

чергу Q0. Після першого виконання (коли квант закінчився, і процес повертається в стан готовності) він надходить вже у чергу Q1. Далі при кожному витісненні він надходить у чергу із все меншим пріоритетом. Таким чином, довгі процеси поступово втрачають свій початковий пріоритет, а короткі – не дуже далеко заходять в ієрархію процесів і отримують перевагу над довгими. У кожній черзі можна, наприклад, використовувати алгоритм FCFS. При досягненні найнижчої черги процес вже не залишає її до свого завершення, знову і знову потрапляючи до неї після використання наданого йому кванта (тобто тут фактично працює алгоритм RR).

Підвищення пріоритетів

Недолік досить простої попередньої схеми зниження пріоритетів полягає у розтягуванні часу виконання довгих процесів. Як результат, це може призвести до голодування, навіть при збільшенні часу виконання для кожної нижньої черги. Для запобігання цьому можна передбачити переміщення процесу в чергу з вищим пріоритетом, якщо за деякий встановлений пороговий період часу його обслуговування не було завершено.

Завдання на виконання роботи

1. Написати програму, що моделює послідовність обслуговування процесів, які знаходяться у черзі готових до виконання, за певним алгоритмом планування (перелік варіантів представлений нижче).

Вхідні дані (студент задає самостійно з урахуванням особливостей заданого алгоритму):

- послідовність процесів, які надходять до черги готових процесів;
- час надходження процесів до черги;
- очікуваний час виконання;
- пріоритет процесів та/або інщі додаткові дані, що необхідні для реалізації алгоритму планування згідно завдання.

Програма повинна обчислювати:

- час початку виконання для кожного процесу;
- час завершення для кожного процесу;
- час затримки (очікування) в черзі для кожного процесу;
- повний час виконання (з урахуванням затримки) для кожного процесу;
- середній час очікування процесів у черзі;
- середній повний час виконання процесів (з урахуванням затримки);
- інші результати, необхідні для оцінки алгоритму планування.

2. Зробити візуалізацію роботи програми у часі (на різних наборах вхідних даних, що ілюструють особливості алгоритму), а також представити кінцеві результати роботи на екрані у вигляді таблиці (як, наприклад, у табл. 1.5), діаграми, графіку, малюнку тощо.

Таблиця 1.5 – Таблиця результатів роботи програми

№ процесу	Час надходження t1	Очікуваний час виконання t2	Час початку виконання t3	Час завершення t4	Час затримки t5	Повний час виконання t6
1	0	6	?	?	?	?
2	3	2	?	?	?	?
3	10	1	?	?	?	?
4	10	3	?	?	?	?
5	12	4	?	?	?	?

3. Проаналізувати та пояснити отримані результати. Надати висновки щодо алгоритму планування за результатами роботи.

Варіанти завдання

1. Розробити окремі програми для кожного з нижчезазначених алгоритмів та порівняти результати їх роботи:

1) алгоритм **FCFS**

2) алгоритм **LCFS**

2. Розробити окремі програми для кожного з нижчезазначених алгоритмів та порівняти результати їх роботи:

1) алгоритм **FCFS**

2) алгоритм **SJF**

3. Розробити окремі програми для кожного з нижчезазначених алгоритмів та порівняти результати їх роботи:

1) алгоритм **SJF**

2) алгоритм **SRTF**

4. Розробити окремі програми для кожного з нижчезазначених алгоритмів та порівняти результати їх роботи:

1) алгоритм **FCFS**

2) алгоритм **RR**

5. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **FCFS**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він не переривається. Перехід на інтерактивний процес відбуватиметься тільки після завершення поточного фонового процесу.

6. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **FCFS**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він переривається. Його продовження відбуватиметься при наступному переході на обслуговування черги фонових процесів.

7. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **SJF**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він не переривається.

Перехід на інтерактивний процес відбуватиметься тільки після завершення поточного фонового процесу.

8. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **SJF**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він переривається**.** Його продовження відбуватиметься при наступному переході на обслуговування черги фонових процесів.

9. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **SRTF**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, він не переривається**.** Перехід на інтерактивний процес відбуватиметься тільки після завершення поточного фонового процесу.

10. Багаторівневі черги (2 рівні):

1) черга інтерактивних процесів – алгоритм **RR**

2) черга фонових процесів – алгоритм **SRTF**

Час розподіляється між чергою фонових і чергою інтерактивних процесів. Наприклад, для інтерактивних процесів – 80 % і для фонових – 20 % часу. Якщо при досягненні ліміту часу виконується фоновий процес, то він переривається. Його продовження відбуватиметься при наступному переході на обслуговування черги фонових процесів з урахуванням особливостей алгоритму **SRTF**.

11. Черги із зворотним зв'язком:

1) черга 1 – алгоритм **RR (**t кванта дорівнює, наприклад, 50 мс)

2) черга 2 – алгоритм **RR (**t кванта дорівнює, наприклад, 100 мс)

3) черга 3 – алгоритм **FCFS**

12. Черги із зворотним зв'язком:

1) черга 1 – алгоритм **RR (**t кванта дорівнює, наприклад, 50 мс)

2) черга 2 – алгоритм **RR (**t кванта дорівнює, наприклад, 100 мс)

3) черга 3 – алгоритм **SJF**

13. Черги із зворотним зв'язком:

1) черга 1 – алгоритм **RR (**t кванта дорівнює, наприклад, 50 мс)

2) черга 2 – алгоритм **RR (**t кванта дорівнює, наприклад, 100 мс)

3) черга 3 – алгоритм **SRJF**

14. Приорітетне планування з підвищенням приорітетів (t кванта та t порогове задати самостійно):

1) черга 1 – алгоритм **RR**

2) черга 2 – алгоритм **RR**

3) черга 3 – алгоритм **FCFS**

15. Приорітетне планування з підвищенням приорітетів (t кванта та t порогове задати самостійно):

1) черга 1 – алгоритм **FCFS**

2) черга 2 – алгоритм **FCFS**

3) черга 3 – алгоритм **RR**

Контрольні запитання

1. В чому полягають особливості витісняльної стратегії планування?
2. Яка стратегія планування є невитісняльною?
3. Яка з вищепредставлених стратегій надає найменший час відгуку?
4. Чому перевага надається інтерактивним процесам, а не фоновим?
5. В чому полягають переваги алгоритму RR?
6. Яка особливість черг із зворотним зв’язком?

ЛАБОРАТОРНА РОБОТА №2

Синхронізація процесів

Метою лабораторної роботи є ознайомлення з особливостями взаємодії процесів і засобами їх синхронізації при доступі до спільно використовуваних ресурсів.

Основні теоретичні відомості

Критичні ресурси, критичні секції, взаємовиключення

Кожному процесу для його виконання потрібні певні ресурси (апаратні, програмні, інформаційні), які система повинна мати у своєму розпорядженні та надати процесу за його вимогою. Найголовнішими ресурсами є процесорний час і оперативна пам’ять. Якщо деякий ресурс потрібен більш ніж одному процесу, виникає конкуренція.

Незважаючи на те, що ресурси можна розділяти, в кожен момент часу вони звичайно доступні тільки одному процесу. Ресурс, що допускає одночасно обслуговування тільки одного користувача з декількох, називається *критичним ресурсом*. Якщо кілька процесів хочуть використовувати критичний ресурс у режимі поділу часу, їм варто синхронізувати свої дії таким чином, щоб цей ресурс завжди перебував у розпорядженні не більш, ніж одного з них. Якщо ресурс зайнятий, то інші процеси, яким він потрібний, тимчасово одержують відмову й повинні чекати його звільнення.

Усередині кожного процесу можна виділити місця (послідовності команд), у яких відбувається звертання до критичних ресурсів. Вони називаються *критичними ділянками* або *секціями*.

Розглянемо два незалежних процеси: Процес_1 і Процес_2 (нижчепредставлені записи називаються записами на *псевдокоді*):

Parbegin

Процес_1: **do while** (true)

begin

Критична ділянка Процесу_1;

end;

частина Процесу_1, що залишилася;

end;

Процес_2: **do while** (true)

begin

Критична ділянка Процесу_2;

end;

частина Процессу_2, що залишилася;

end;

Parend

Конструкція вигляду

Parbegin

Оператор 1;

Оператор 2;

...

Оператор N;

Parend

означає, що операторі 1…N виконуються паралельно.

Критичні ділянки процесів відповідають тим групам операцій, які звертаються до поділюваного критичного ресурсу.

Для недопущення одночасного виконання обох критичних ділянок, у системі повинен бути передбачений механізм, який би синхронізував ці два процеси. Цей механізм повинен володіти наступними двома властивостями:

1. Якщо один або кілька процесів хочуть звернутися до своїх критичних ділянок, тоді тільки один з них повинен одержати дозвіл на входження до своєї критичної ділянки.
2. У кожен момент часу не більш ніж одному процесу дозволяється перебувати у своїй критичній ділянці.

У процесів виникає не тільки проблема синхронізації, але й необхідність в обміні інформацією. Назвемо *Постачальником* процес, що відправляє порції інформації іншому процесу, який назвемо *Споживачем*. Ці порції інформації

можна розглядати як повідомлення. Один з методів, застосовуваних при реалізації передачі повідомлень, полягає в створенні пулу вільних буферів, кожен з яких може містити одне повідомлення.

Між процесами *Постачальник* і *Споживач* є черга заповнених буферів, які містять повідомлення. Коли *Постачальник* хоче надіслати повідомлення, він додає у хвіст цієї черги ще один буфер. Для отримання повідомлення *Споживач* забирає буфер з голови черги. Така організація вимагає кооперації *Постачальника* і *Споживача* в багатьох областях. Вони повинні стежити за кількістю заповнених і вільних буферів. *Постачальник* може передавати повідомлення тільки при наявності вільних буферів. *Споживач* може одержувати повідомлення тільки, якщо черга не порожня.

Для обліку заповнених і вільних буферів потрібні поділювані змінні. Але оскільки змінні використовуються у режимі поділу, будуть потрібні й критичні ділянки. Отже, змінювати значення лічильника заповнених буферів потрібно у режимі *взаємного виключення*.

Та ж проблема виникає при реалізації *черги повідомлень*. Нехай черга реалізована у вигляді зв'язного списку. Якщо *Постачальник* додає повідомлення у чергу саме в той момент, коли *Споживач* забирає з неї останнє повідомлення, може статися хибне посилання до списку.

Такі процеси як *Постачальник* і *Споживач*, що спілкуються на однакових правах, називаються також *співпрограмами*. Ми ж будемо тут користуватися терміном "*процеси, що кооперуються*".

Якщо процесу *Постачальник* важливо знати, що його повідомлення отримане, тоді *Споживач* повинен підтверджувати отримання кожного повідомлення, тобто *Постачальник* і *Споживач* змінюються місцями.

Якщо у системі є декілька пар процесів типу *Постачальник-Споживач*, можна організувати поділ вільних буферів, об'єднавши їх у загальний пул.

Спочатку розглянемо синхронізацію двох паралельних процесів, а далі зупинимось на вирішенні проблем синхронізації відносин типу *Постачальник-Споживач*.

Синхронізація за допомогою елементарних прийомів нижнього рівня

Метод блокування пам'яті

Взаємне виключення реалізують апаратно, зробивши операції над пам'яттю неподільними. Тобто, якщо кожен із двох процесів намагається помістити якісь значення в одну й ту ж саму комірку, тоді суперечка вирішується апаратно: одному процесу дозволяється виконати операцію запису негайно, а іншому доводиться чекати, поки перший не закінчить. Таке рішення називається *блокуванням пам'яті*.

Повернемося до розгляду прикладу з двома паралельними процесами. Оскільки швидкості обох процесів є довільними, вищезазначені умови повинні виконуватися, якою б не була швидкодія кожного процесу відносно іншого. Здавалося б, найпростіше запропонувати наступне рішення:

```
Перемикач_1, Перемикач_2: boolean;
Begin
Перемикач_1:=False;
Перемикач_2:=False;
Parbegin
Процес_1: do while (True)
                Цикл_1:  do while (Перемикач_2)
                                end;
                Перемикач_1:=True;
                /* Критична ділянка Процесу_1*/
                Перемикач_1:=False;
                /* частина Процесу_1, що залишилася*/
                end;
Процес_2: do while (True)
                Цикл_2:  do while (Перемикач_1)
                                end;
                Перемикач_2:=True;
                /* Критична ділянка Процесу_2*/
                Перемикач_2:=False;
                /* частина Процесу_2, що залишилася*/
```

```
            end;
    Parend;
    End.
```

Однак (оскільки вважається, що швидкості роботи процесів є довільними) припустимо, що Процес_2 працює набагато швидше, ніж Процес_1. Настільки швидше, що фактично після виявлення Процесом_1 значення False в Перемикачі_2, але до встановлення ним значення True у Перемикачі_1, Процес_2 пробігає свою частину, що залишилася, і перескакує через Цикл_2 (оскільки в Перемикачі_1 все ще значення False). У такому випадку, обидва процеси перейдуть до виконання (одночасно) своїх критичних ділянок, і система буде працювати неправильно.

Пропонуємо самостійно спробувати змінити наведену програму так, щоб вона задовольняла обмеженням, які накладаються на рішення проблеми критичної ділянки. Ми ж приведемо варіант рішення, запропонованого Деккером **(***алгоритм Деккера***)**:

```
 C1, C2, Черга: integer;
Begin
C1:=0;
C2:=0;
Черга:=1;
Parbegin
Процес_1: begin C1:=1;
            do while (C2=1)
                  if Черга=2 then
                  begin
                  C1:=0;
                  do while (Черга=2);
                  end;
                  C1:=1;
            end;
            end;
            /*Критична ділянка Процесу_1;*/
            C1:=0;
```

```
                Черга:=2;
                /*залишок Процесу_1;*/
        end;
Процес_2: begin C2:=1;
                do while (C1=1)
                if Черга=1 then
                        begin
                        C2:=0;
                        do while (Черга=1);
                        end;
                        C2:=1;
                end;
end;
                /*Критична ділянка Процесу_2;*/
                C2:=0;
                Черга:=1;
                /* залишок Процесу_2;*/
        end;
Parend;
End.
```

Тут введена додаткова змінна *Черга*, яка визначає, чия саме черга спробувати увійти у критичну ділянку за умови, що обидва процеси хочуть виконати свої критичні ділянки. Це рішення має узагальнення для випадку довільного числа процесів, які конкурують за критичний ресурс.

Метод перевірки і установки

Ця машинна операція значно спрощує рішення проблеми критичної ділянки. До операції "*Перевірка і Установка*" звертаються з двома параметрами: *Local* і *Common*. Операція бере значення параметра *Common* і присвоює його змінній *Local*, а потім встановлює змінній *Common* значення 1. Ця операція є однією з апаратних засобів вирішення задачі критичної ділянки. Вперше вона була реалізована у IBM-360 і дістала назву *TS* (*T*est and *S*et). Змінна *Common* буде загальною для усіх процесів. Вона дорівнюватиме 1, коли

один із процесів знаходиться у критичній області. Якщо процес хоче увійти у критичну область, його локальна змінна приймає значення 1. Головна властивість цієї операції – її неподільність. Коли процес виконує операцію "*Перевірка й Установка*", ніяких інших дій не може відбуватися упродовж її виконання.

Змінна *Common* поділяється процесами, які підлягають синхронізації стосовно деякого критичного ресурсу. У кожного процесу є своя власна змінна *Local*. Якщо *Common = 1*, це означає, що якийсь процес перебуває у своїй критичній ділянці. Початкове значення змінної *Common = 0*.

Приведемо рішення проблеми методом *"Перевірка і установка"*. У цьому решенні припускається, що передбачене *блокування пам'яті*, тобто операція *Common := 0* є неподільною. Алгоритм наступний:

```
Var Common, Locall, Local2: integer;
begin
   Common:= 1;
Parbegin
   Процес_1: while true do
          begin
            Local1:= 1;
            while Local1=1 do TS(Local1, Common);
          < критична ділянка Процесу_1>;
            common:=0;
          < залишок Процесу_1>;
          end;
          end;
   Процес_2: while true do
          begin
          Local2:=1;
          while Local2=1 do TS(Local2, Common);
          < критична ділянка Процесу_2 >;
            common:=0;
          < залишок Процесу_2 >;
    end
```

```
    end;
Parend;
End.
```

Нехай першим хоче увійти у свою критичну ділянку процес Процес_1. У цьому випадку, *Local1* спочатку прийме значення 1, а після циклу перевірки за допомогою команди *TS (Local1, Common)* стане дорівнювати 0. При цьому значення *Common* буде дорівнювати 1. Процес_1 увійде у критичну ділянку. Після виходу з критичної ділянки виконується операція *Common:=0*, що дасть можливість Процес_2 увійти у свою критичну ділянку. При цьому ми вважаємо, що операція *Common:=0* є неподільною завдяки операції *блокування памяті.*

Недолік цього методу у тому, що у циклі перевірки значення змінної *Common* непродуктивно використовується процесорний час. Ситуація ще погіршується, якщо під час виконання Процес_1 у його критичній секції виникне переривання, а Процес_1 почне виконання. Він увійде у цикл перевірки і не вийде з нього доти, поки Процес_1 не вийде з переривання.

До речі, у мікропроцесорів Intel, починаючи з 80386, існують спеціальні команди під назвами *BTC* (***B**it **T**est and **C**onvert*), *BTS, BTR,* які є прототипами команди *"Перевірка та Установка"*. Обидва розглянутих методи синхронізації можуть виявитися досить неефективними, оскільки щоразу, коли один процес виконує свою критичну ділянку, будь-який інший процес, що теж хоче увійти у критичну ділянку, потрапляє у цикл перевірки і повинен там очікувати дозволу. При такому очікуванні в циклі, що називається *активним очікуванням*, дарма витрачається час процесора. Одним із методів, що дозволяють уникнути активного очікування, є використання *семафорів***.**

Семафори

Семафор – ціла змінна, значення якої можуть змінювати тільки операції *P* й *V*. Нехай *S* – семафор. Коли процес виконує операцію *P(S), S* зменшується на одиницю та

1) Якщо $S \geq 0$, процес продовжує роботу.

2) Якщо *S<0*, процес зупиняється й стає в чергу очікування, пов'язану з *S*. Він залишається заблокованим доти, поки операція *V(S)*, виконана іншим процесом, не звільнить його.

Коли процес виконує *V(S)*, *S* збільшується на одиницю та

1) Якщо *S>0*, процес продовжує роботу.
2) Якщо *S≤0*, тоді один процес вилучається із черги очікування й одержує дозвіл продовжити роботу. Процес, що звернувся до операції *V(S)*, теж може продовжувати роботу.

Крім того, операції *P* й *V* є неподільними. У кожен момент часу тільки один процес може виконувати операцію *P* або *V* над даним семафором. Тому, якщо *S=1* і два процеси одночасно спробують виконати операцію *P(S)*, тільки одному з них буде дозволено продовжити роботу. Інший процес буде заблокований і поставлений у чергу до семафора *S*. Операції *P* й *V* можна реалізувати всередині ядра операційної системи. Таким чином, забезпечується неподільність цієї операції.

Семафор, максимальне значення якого дорівнює одиниці, називається *двійковим семафором*. За допомогою двійкового семафора процеси можуть організувати взаємне виключення, використовуючи операції *P(S)* і *V(S)*. Наприклад:

```
Var ВІЛЬНИЙ:integer;
begin
 ВІЛЬНИЙ:=1;
 parbegin
  Процес_1: begin  do while (True);
                    <Початок Процесу 1>;
                    P(ВІЛЬНИЙ);
                    <Критична ділянка 1>;
                    V(ВІЛЬНИЙ);
                    <Залишок Процесу 1>;
                   end;
            end;
  Процес_2: begin  do while (True);
```

```
                <Початок Процесу 2>;
                P(ВІЛЬНИЙ);
                <Критична ділянка 2>;
                V(ВІЛЬНИЙ);
                <частина, що залишилася Процесу 2>;
              end;
        end;
 parend;
end.
```

Тут *S* приймає значення 1, 0, -1. Якщо *S=1*, це означає, що жоден процес не перебуває у своїй критичній ділянці. Якщо *S=0* – один процес перебуває у своїй критичній ділянці. Якщо *S=-1* – один процес перебуває у своїй критичній ділянці, а другий – у черзі очікування.

Рішення, що використовують двійкові семафори, так само застосовується і до великої кількості процесів. Якщо алгоритм Деккера стає дуже складним для більш ніж двох процесів, рішення із семафором залишається тривіальним. У цьому головна перевага семафора. Виконання семафорних операцій *P(S)* та *V(S)* можна реалізувати, наприклад, таким чином:

```
P(S:
   if S>=1 then S:=S-1;
  else Wait(S) {зупинити процес та помістити його у чергу очікування до семафору S}
V(S):
   if S<0 then Release(S);{помістити один з очікующих процесів з черги до семафору S
у чергу готових до виконання процесів}
S:=S+1;
```

Wait означає, що супервізор ОС повинен перевести процес у стан очікування, тобто відправити його у чергу до семафора *S*. Виклик *Release* означає звернення до диспетчера задач з проханням перевести перший процес з черги процесів до семафора *S* у стан готовності.

Розглянемо приклад з однопроцесорною системою. У ній неподільність операцій *P(S)* та *V(S)* можна забезпечити блокуванням переривань. Семафор *S* можна реалізувати у вигляді запису з двома полями. В одному – значення

змінної *S*, у другому – показчик на список процесів, заблокованих на семафорі *S*.

```
Type semafore= record
    лічильник: integer;
    показчик: pointer
    end;
var s:semaphore;
procedure P(var s: semaphore);
       begin
          <заборонити прерівания>;
           s.лічильник:=s. лічильник-1;
           if s.лічильник <0 then
           wait(s);{вставити процес, що звернувся, у список s.показчик та передати на виконання процесору процес з черги}
          <дозволити переривання >
      end;
procedureV( var s: semaphore);
  begin
     <заблокувати переривання>;
     s.лічильник:=s.лічильник +1:
      if s.лічильник <=0 then
      Release (s); {разблокувати перший процес із черги s.показчик}
     <дозволити переривання>;
  end;
procedure InitSem(var:semaphore);
  begin
      s.лічильник :=1;
      s.показчик:= nil;
end;
```

У багатопроцесорних системах реалізація семафорів є більш складною. Одночасний доступ до семафора *S* з двох процесів ми блокували обмеженням переривань. Цей механізм у багатопроцесорній системі не працює. Ми не зможемо заблокувати одночасний доступ декількох процесів до семафора *S* за допомогою раніше використаного механізму. Тут потрібен механізм, що

дозволяв би виключення доступу для декількох процесорів одночасно. Одним з рішень є використання вищерозглянутого методу *“Перевірка й Установка”*.

За допомогою загальних семафорів та використання поштової скриньки легко вирішується проблема *“Постачальник–Споживач”*. Подільні змінні (у цьому випадку, лічильники вільних та зайнятих буферів) повинні бути захищені від обох процесів (див. описи представлених вище процедур *P(S)* та *V(S)*).

```
Var s.вільно, s.заповнено, s.взаємовиключено: semaphore;
begin
    Init Sem (s.вільно,N);    { N-кількість комірок буферу}
    Init Sem (s.заповнено,0);
    Init Sem (s.взаємовиключено,1);
parbegin
    Постачальник:   while true do
                        begin
                          <приготувати повідомлення>;
                          P(s.вільно);
                          P(s.взаємовиключено);
                           <передача повідомлення>;
                           V(s.заповнено);
                           V(s.взаємовиключено);
                        end
                    end;
    Споживач:       while true do
                        begin
                           P(s.заповнено);
                           P(s.взаємовиключено);
                            <отримати повідомлення>;
                           V(s.вільно);
                           V(s.взаємовиключено>;
                            <обробка повідомлення>;
                        end
                    end
parend
end.
```

s.вільно, s.заповнено – цілочисельні семафори. Вони використовуються як лічильники вільних та зайнятих буферів. *s.взаємовиключення* – двійковий семафор.

За допомогою семафорів можна організувати синхронізацію процесів, коли завершення одного процесу пов’язано з очікуванням завершення іншого.

Нехай є два процеси: Процес_1 та Процес_2. Необхідно, щоб Процес_1 запускав процес Процес_2 та не продовжував би своє виконання до тих пір, поки процес Процес_2 до кінця не виконав би свою роботу.

```
Var s:semaphore;
    begin
      Init Sem(s,0);
       Пр1:  begin
                  <початок Процес_1>;
                   ON(Пр2);  {Поставити на виконання Процес_2}
                    P(s);
                    <завершення Процес_1>;
                    STOP
                end;
        Пр2:  begin
                    <виконаня Процес_2>;
                    V(s);
                     STOP
                end
     end.
```

Початкове значення семафора дорівнює 0 (див. вище процедуру *InitSem*). Якщо Процес_1 почав виконуватися першим, тоді через деякий час він поставить на виконання Процес_2. Після цього буде виконана процедура *P(S),* і він перейде у стан очікування на семафорі *S*. Процес Процес_2 виконає усі необхідні дії та виконає процедуру *V(S)*, що відкриє семафор *S*. Після цього Процес_1 знову готовий до виконання.

За допомогою загальних семафорів можна організувати керування ресурсами, наприклад дисків, і використати для синхронізації процесів. Однак, при використанні семафорів для синхронізації й керування ресурсами виникають труднощі, пов'язані з організацією черги до семафору.

Якщо значення семафора *S* стає негативним, це означає, що один або кілька процесів очікують черги, пов'язаної з *S*. Коли виконується чергова операція *V(S),* планувальник повинен вибрати, який процес треба взяти із черги. Планувальник може обслуговувати чергу в порядку надходження або використати пріоритетну схему обслуговування. Порядок обслуговування черги повинен обумовлюватися цілями застосування даного семафора. Питання визначення дисципліни обслуговування при проектуванні ОС тісно пов'язане з моделюванням обчислювальних процесів, для чого можуть бути ефективно використані моделі масового обслуговування. Пріоритет процесу можна визначати динамічно відповідно до кількості ресурсів, якими цей процес розпоряджається.

Вважається, що семафори – досить ефективний засіб для реалізації синхронізації й спілкування між процесами. Проте, часто виявляється, що *семафори* є незручним засобом. Наприклад, складно реалізувати схему передавання повідомлень між декількома процесами.

Елементарні прийоми синхронізації на верхньому рівні

Поштові скриньки

Якщо процес *P1* хоче спілкуватися із процесом *P2*, *P1* просить систему створити поштову скриньку для зв'язку цих двох процесів таким чином, щоб вони могли передавати один одному повідомлення. Для того, щоб послати процесу *P2* якесь повідомлення, процес *P1* просто поміщає це повідомлення у поштову скриньку, звідки процес *P2* може його в будь-який час взяти. При застосуванні поштової скриньки процес *P2* зрештою обов'язково отримає повідомлення, коли звернеться за ним, якщо взагалі звернеться.

Поштова скринька – це інформаційна структура, для якої задаються певні правила, що описують її роботу. Вона складається з *головного* елемента з описом даної поштової скриньки і з декількох *гнізд*, в які поміщають повідомлення. Розмір кожного гнізда та кількість гнізд звичайно задаються при створенні поштової скриньки.

Правила роботи можуть бути різними, в залежності від складності поштової скриньки.

У найпростішому випадку, повідомлення передаються тільки в *одному напрямку*. Процес *P1* може надсилати повідомлення доти, поки є вільні гнізда. Якщо всі гнізда заповнені, процес *P1* може або чекати, або зайнятися іншими справами й спробувати надіслати повідомлення пізніше. У свою чергу, *P2* може одержувати повідомлення доти, поки є заповнені гнізда.

*Двосторонній зв'язок***.** Використовується при необхідності передавати підтвердження про одержання повідомлення. При цьому дозволяється передача повідомлень через поштову скриньку в обох напрямках. Якщо передавальний процес *P1* працює швидше, ніж приймаючий процес *P2*, *P1* може заповнити всі гнізда, не залишивши *P2* гнізд для відповідних повідомлень. Щоб цього не сталося, пересилання відповідей має відбуватися у тих самих гніздах, де перебували їх повідомлення.

*Багатовходові поштові скриньки***.** Використовуються при необхідності декільком процесам спілкуватися з одним процесом. Прикладом може служити система керування файлами. Такі скриньки ефективніші порівняно з окремими скриньками для кожного окремого процесу, але їх реалізація складніша.

Для створення поштової скриньки операційна система визначає показчики на область пам’яті для операцій читання/запису та відповідні змінні для захисту доступу. Основними методами реалізації є або буфер, розмір якого задається, або зв’язаний список, який не накладає обмежень на кількість повідомлень.

Повернемось до семафора. Аналіз задач синхронізації показує, що поряд з очевидними перевагами реалізації (простота, відсутність активного очікування) семафорні механізми мають цілий ряд недоліків. Так, вони не вказують безпосередньо на умову синхронізації. Тому при побудові складних схем синхронізації алгоритми синхронізації виявляються досить складними. Вирішенням проблеми може стати застосування монітора.

Метод монітору Хоара

Монітор – це набір процедур і інформаційних структур, якими процеси користуються у режимі поділу, причому в кожен момент часу монітором може користуватися тільки один процес.

Можна представити монітор як кімнату, від якої є тільки один ключ. Якщо якийсь процес має намір скористатися цією кімнатою, і ключ перебуває зовні, тоді цей процес може відімкнути кімнату, увійти й скористатися однією із процедур монітора. Якщо ключа зовні немає, процесу доведеться чекати, поки той, хто користується кімнатою у цей момент, не вийде з неї й не віддасть ключа. Крім того, у кімнаті не можна залишатися назавжди.

Як приклад, розглянемо ресурс, котрий розподіляє деяка програма-планувальник. Щоразу, коли процес хоче одержати у своє розпорядження деякий ресурс, він повинен звернутися до планувальника. Процедуру планувальника розділяють усі процеси, і кожен процес може в будь-який момент звернутися до неї. Але планувальник не в змозі обслуговувати одночасно більш ніж один процес. Тобто, планувальник являє собою приклад монітора.

Іноді монітору необхідно затримати процес, якому потрібен вже кимось використовуваний ресурс, до можливості надання цього ресурсу.

Варто підкреслити, що монітор – це пасивний об'єкт, як кімната. Це – не процес. Монітор оживає тільки тоді, коли який-небудь процес вирішує скористатися його послугами. *Особливість* програми монітора полягає в тому, що в будь-який момент її може виконати *тільки один процес.*

Таким чином, монітор – це механізм забезпечення паралелізму, який має як дані, так і процедури, що необхідні для реалізації динамічного розподілу загального ресурсу або групи ресурсів. Процес, якому потрібен ресурс, повинен звернутися до монітора, який або забезпечить доступ, або відмовить. Вхід у монітор знаходиться під жорстким контролем – виконується взаємовиключення процесів, оскільки у кожен момент часу у монітор може увійти тільки один

процес, а інші процеси чекають, причому режимом очікування керує сам монітор. Він вказує умову, за якою процес має очікувати.

Внутрішні дані монітору можуть бути глобальними або локальними. До усіх даних або ресурсів можна звертатись тільки із середини монітора. Процеси можуть звертатись тільки до процедур та не можуть безпосередньо звертатись до даних. Якщо процес звертається до деякої процедури і виявляє, що відповідний ресурс зайнятий, процедура видає команду очікування *Wait* з певною умовою очікування. Процес, що займав ресурс, може його звільнити, однак, може статися, що вже є процеси, що чекали звільнення цього ж ресурсу. Монітор виконує операцію *SIGNAL*, після чого один з процесів з черги до цього ресурсу може його отримати. Якщо процес повертає ресурс, а він нікому не потрібен, тоді ресурс просто заноситься в список вільних ресурсів.

Іноді роблять так, щоб процес з черги до ресурсу мав пріоритет вищий ніж новий процес, який хоче оволодіти цим ресурсом. Розглянемо приклад виділення одного ресурсу.

```
monitor Resours;
condition free;  {умова вільний}
var busy: boolean ;    { busy - зайнятий}
 procedure Request;  {запит}
begin
    if busy then Wait(free) ;
    busy:=true;
    Take OFF;  {видати ресурс}
 end;
Procedure Release;  {звільнити}
begin
    Take ON  { отримати ресурс}
     busy:= false;
     SIGNAL(free);
end;
begin
    buse:=false;
end.
```

Ресурс запитується та звільняється процедурами *Request* (запит) та *Release* (звільнення). Якщо процес звернувся до процедури *Request*, коли ресурс використовується, *busy:= true*, і процедура виконує команду *Wait(free)*. Ця операція блокує не команду *Request*, а процес, що до неї звернувся. Процес розміщується наприкінці черги очікуючих процесів, доки не буде виконана умова *free(вільний).* Коли процес, що використовує ресурс, звертається до процедури *Release*, операція монітора *SIGNAL* деблокує процес, що стоїть першим у черзі за даним ресурсом, не дозволяючи виконуватись ніякій іншій процедурі всередині монітора. Цей деблокований процес готовий поновити виконання процедури *REQUEST* одразу після операції *WAIT(free)*, яка його блокувала. Якщо операція *SIGNAL(free)* виконується у той час, коли немає процеса, який очікує умову *free*, ніякі дії не виконуються.

При синхронізації процесів семантика монітора гарантує, що, якщо хоча б один процес очікує виконання умови, ніякий інший, що звернувся, не може втрутитись між сигналом про виконання цієї умови й продовженням одного із процесів, що очікують.

Переваги моніторів, порівняно з семафорами, полягають у наступному:

- Монітори – дуже гнучкий інструмент. Наприклад, з його допомогою можна реалізувати поштову скриньку
- Здійснюється локалізація усіх поділюваних змінних усередині тіла монітора, що дозволяє позбутися малоприємних конструкцій у процессах, які синхронізуються.

Завдання на виконання роботи

1. Розробити програму, що моделює роботу заданого об’єкта, використовуючи для доступу процесів до подільних ресурсів засоби синхронізації. Пристрій, що моделюється, і засоби синхронізації процесів визначаються варіантом завдання.

Вхідні дані студент задає самостійно з урахуванням особливостей індивідуального завдання.

2. Забезпечити візуалізацію роботи моделі з наглядною демонстрацією результатів.

3. Проаналізувати та пояснити отримані результати. За результатами роботи надати висновки щодо використаних засобів синхронізації.

Варіанти завдання

1. Об'єкт моделювання: *Автомат для розміну грошей (монет)*.

Автомат приймає монету та ідентифікує її (визначає номінал). Ідентифікація моделюється датчиком випадкових чисел. Приймаються монети номіналом 1, 2, 5, 10, 25, 50 коп. і 1 грн. Розмін монети здійснюється на монети номіналу, що вводиться з клавіатури. Кількість монет різного номіналу (1, 2, 5, 10, 25, 50 коп.), що містить автомат для розміну та видачі здачі, наперед задається. Якщо розмін и видача монет можливі, вони виконуються. Інакше здійснюється відмова від виконання операції розміну і видається повідомлення про її причину. Монети, що приймаються для розміну, додаються до наявних в автоматі, та, у свою чергу, можуть використовуватися для розміну.

Кількість процесів: Модель роботи автомату представити у вигляді двох взаємодіючих процесів *A* і *B*: процес *A* визначає факт надходження монети та ідентифікує її номінал, а процес *B* здійснює розмін і видає гроші або відмову виконання операції розміну.

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *семафори*.

2. Об'єкт моделювання: *Автомат для розміну грошей (монет) – див. варіант 1.*

Кількість процесів: *див. варіант 1.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *алгоритм Деккера*.

3. Об'єкт моделювання: *Автомат для розміну грошей (монет) – див. варіант 1.*

Кількість процесів: *див. варіант 1.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *поштові скриньки.*

4. Об'єкт моделювання: *Автомат для розміну грошей (монет) – див. варіант 1.*

Кількість процесів: *див. варіант 1.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *операцію "Перевірка й Установка" (або її модель).*

5. Об'єкт моделювання: *Автомат для продажу авіабілетів.*

Автомат приймає гроші (тут тільки одного визначеного номіналу – 1грн.) і видає здачу монетами вартістю до 1 грн. (1, 2, 5, 10, 25, 50 коп.). Сума здачі розраховується. Початкова кількість монет кожного номіналу задається і становить:

1 коп. – 50 шт., 2 коп. – 25 шт.,

5 коп. – 20 шт., 10 коп. – 15 шт.,

25 коп. – 10 шт., 50 коп. – 5 шт.,

Введення запиту на продаж здійснюється шляхом вибору певного пункту меню:

0 – включити автомат; 1 – купити білет до Києва вартістю 28 коп.;

2 – білет до Москви вартістю 37 коп.; 3 –білет до Лондона вартістю 50 коп.;

4 – білет до Берлина вартістю 77 коп.; 5 –білет до Парижа вартістю 91 коп.

Якщо здачу видати можливо, програма формує потрібний набір монет для здачі (також коригує банк монет) і формує сигнал на видачу. Якщо потрібних купюр для здачі не достає, формується відповідне повідомлення. Вимоги на видачу грошей надходять після чергового сеансу продажу або відмови.

Кількість терміналів і процесів: Автомат має *2 термінали (A і B)*, що працюють незалежно. Модель автомата представити у вигляді трьох взаємодіючих процесів *A, B і C*. Процеси A і B визначають факт надходження вимоги на продаж, потрібну суму здачі і переходять до стану блокування. Процес C очікує момент появи необхідності видати здачу і, якщо величина здачі відома, визначає кількість і номінали купюр або неможливість видачі

здачі. Потім він активізує процес А або В (від кого прийшла вимога), а сам очікує на черговий запит.

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *семафори.*

6. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: – *див. варіант 5.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *поштові скриньки.*

7. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: – *див. варіант 5.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *алгоритм Деккера.*

8. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: Модель автомата представити у вигляді *двох* взаємодіючих процесів А і В. Процес А визначає факти надходження вимог на продаж і потрібну суму здачі. Процес В очікує момент появи необхідності видати здачу і, якщо величина здачі відома, визначає кількість і номінали потрібних монет або неможливість видачі.

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *семафори.*

9. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: – *див. варіант 8.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *поштові скриньки.*

10. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: – *див. варіант 8.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *алгоритм Деккера.*

11. Об’єкт моделювання: *Автомат для продажу авіабілетів – див. варіант 5.*

Кількість терміналів і процесів: *– див. варіант 8.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *операцію “Перевірка й Установка” (або її модель).*

12. Об’єкт моделювання: *Банкомат.*

Банкомат приймає заявку на виплату грошей на суму X, порівнює її з кількістю грошей на рахунку клієнта Y і, якщо X<Y, здійснює спробу виплати певними купюрами номіналом 1, 2, 5, 10, 20, 50, 100 грн. Величини X і Y – випадкові. числа. Початкова кількість купюр різних номіналів у банкоматі задається студентом. Якщо сума X підлягає видачі, програма визначає, чи можна цю суму виплатити наявними у даний момент купюрами. Якщо так – гроші видаються (також коригується кількість наявних у банкоматі купюр відповідного номіналу), якщо ні – банкомат повідомляє про відмову.

Вимоги на видачу грошей надходять до банкомату після чергового обслуговування клієнта (або відмови).

Кількість терміналів і процесів: Модель автомата представити у вигляді двох взаємодіючих процесів *A і B*.

Процес *A* визначає факт надходження вимоги на виплату та можливість виплати потрібної суми, фіксує можливість виплати.

Процес *B* очікує момент появи необхідності зробити виплату і, якщо виплата можлива, визначається необхідна кількість купюр кожного номіналу для виплати, коригується рахунок клієнта і кількість готівки у банкоматі, йде виплата. Якщо виплата неможлива – клієнту надається про це повідомлення.

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *семафори.*

13. Об’єкт моделювання: *Банкомат – див. варіант 12.*

Кількість терміналів і процесів: *див. варіант 12.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *поштові скриньки.*

14. Об’єкт моделювання: *Банкомат – див. варіант 12.*

Кількість терміналів і процесів: *див. варіант 12.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *алгоритм Деккера*.

15. Об’єкт моделювання: *Банкомат – див. варіант 12.*

Кількість терміналів і процесів: *див. варіант 12.*

Засоби синхронізації: Для організації доступу до подільних ресурсів використати *операцію “Перевірка й Установка” (або її модель)*.

Контрольні запитання

1. У чому полягають особливості алгоритму Деккера при вирішенні проблеми критичної ділянки?
2. Яким чином спрощується вирішення проблеми критичної ділянки за допомогою операції *“Перевірка й Установка”?*
3. Який основний недолік методу Деккера та операції *“Перевірка й Установка”?*
4. Які переваги для синхронізації процесів має використання семафорів?
5. У чому відмінність двійкового семафору від м’ютекса?
6. У чому полягають недоліки використання семафорів?
7. Які існують основні способи реалізації поштових скриньок?
8. Які переваги має використання моніторів у порівнянні з семафорами?

ЛАБОРАТОРНА РОБОТА №3

Керування пам'яттю

Метою лабораторної роботи є ознайомлення з існуючими способами структурування пам’яті, алгоритмами керування пам’яттю та перетворення віртуальних адрес у фізичні адреси.

Основні теоретичні відомості

Функції ОС по керуванню пам'яттю

Під *пам'яттю* розуміється оперативна пам'ять комп'ютера. На відміну від пам'яті жорсткого диску, яку називають *зовнішньою пам'яттю,* оперативна пам'ять для збереження інформації вимагає постійного електроживлення. Особлива роль пам'яті полягає в тому, що процесор може виконувати інструкції програми за умови, якщо вони перебувають у пам'яті. Пам'ять розподіляється як між модулями прикладних програм, так і між модулями самої операційної системи.

Функціями ОС по керуванню пам'яттю в мультипрограмній системі є:

- відслідковування вільної й зайнятої пам'яті;
- виділення пам'яті процесам і звільнення пам'яті по завершенні процесів;
- витискання кодів і даних процесів з оперативної пам'яті на диск, якщо розміру основної пам'яті недостатньо для розміщення в ній усіх процесів, і повернення їх в оперативну пам'ять, коли в ній звільняється місце;
- настроювання адрес програми на конкретну область фізичної пам'яті.

Під час роботи ОС доводиться створювати нові службові інформаційні структури, такі як описувачі процесів і потоків, різні таблиці розподілу ресурсів, буфери для обміну даними й т.п. Всі ці системні об'єкти вимагають пам'яті. У деяких ОС під час установки резервується певний фіксований обсяг

пам'яті для системних потреб. В інших же ОС використовується більш гнучкий підхід, при якому пам'ять для системних цілей виділяється динамічно.

Захист пам'яті — ще одне важливе завдання ОС. Воно полягає у тому, щоб не дозволити виконуваному процесу записувати або читати дані з пам'яті, яка призначена іншому процесу.

Типи адрес

Для ідентифікації змінних і команд на різних етапах життєвого циклу програми використовуються символьні імена, віртуальні адреси й фізичні адреси (рис. 3.1).

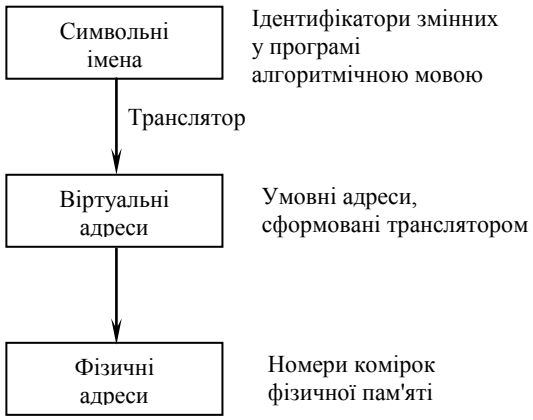

Рис. 3.1 – Типи адрес

Символьні імена задає користувач при написанні програми.

Віртуальні (умовні) адреси формує транслятор, що перекладає програму на машинну мову.

Фізичні адреси відповідають номерам комірок оперативної пам'яті, де фактично розташовані змінні й команди.

Сукупність віртуальних адрес процесу називається *віртуальним адресним простором процесу*. В ньому зберігається *образ процесу*. Діапазон можливих адрес віртуального простору для усіх процесів є однаковим. Проте, кожен процес має власний віртуальний адресний простір – транслятор надає віртуальні адреси змінним і командам кожної програми незалежно.

У різних ОС використовуються різні способи структуризації адресного простору, а саме:

- *Лінійна* послідовність віртуальних адрес. При цьому віртуальна адреса – число, що представляє собою зміщення відносно початку віртуального адресного простору. Таку структуру адресного простору називають також *пласкою (flat).* Адресу ж називають *лінійною віртуальною адресою*.
- Віртуальний адресний простір ділиться на частини, називані сегментами. Віртуальна адреса представляє собою *пари чисел* (*n*, *m*), де *n* визначає сегмент, а *m* — зміщення всередині сегмента.
- складніші способи структуризації, коли віртуальна адреса утворюється трьома числами (або навіть більше).

Існують два принципово різних підходи до перетворення віртуальних адрес у фізичні:

- Заміна віртуальних адрес на фізичні виконується одноразово для кожного процесу під час початкового завантаження програми у пам’ять. Виконує її системна програма — *переміщуючий завантажувальник***.** На підставі наявних вихідних даних про початкову адресу фізичної пам'яті, а також інформації, наданої транслятором про адресно-залежні елементи програми, *завантажувальник* виконує завантаження програми із заміною віртуальних адрес на фізичні.
- Програма завантажується у незмінному вигляді з віртуальними адресами. При завантаженні ОС фіксує зміщення дійсного розташування програмного коду щодо віртуального адресного простору. В процесі виконання програми при кожному звертанні до оперативної пам'яті виконується перетворення віртуальної адреси у фізичну. Виконує це *динамічний завантажувальник***.**

Другий спосіб є більш гнучким. У той час як переміщуючий завантажувальник жорстко прив'язує програму до початку виділеної їй ділянки пам'яті, динамічне перетворення віртуальних адрес дозволяє переміщати програмний код процесу протягом усього періоду його виконання.

Як правило, обсяг віртуального адресного простору перевищує доступний обсяг оперативної пам'яті. У такому випадку, для зберігання даних віртуального адресного простору ОС використовує зовнішню пам'ять. Однак співвідношення обсягів віртуальної й фізичної пам'яті може бути й зворотним.

Варто пам'ятати, що у загальному випадку механізми віртуального адресного простору і віртуальної пам'яті– не одне й теж саме. Можна уявити собі ОС, у якій підтримуються віртуальні адресні простори для процесів, але відсутній механізм віртуальної пам'яті. Це можливо тільки у тому випадку, якщо розмір віртуального адресного простору кожного процесу менший ніж обсяг фізичної пам'яті.

Ключовою проблемою віртуальної пам'яті, що виникає в результаті багаторазової зміни місця розташування в оперативній пам'яті образів процесів або їхніх частин, є *перетворення віртуальних адрес у фізичні.*

Сьогодні існуючі реалізації віртуальної пам'яті можуть бути представлені трьома наступними класами:

- *Сторінкова віртуальна пам'ять*. Переміщення даних здійснюється сторінками – частинами віртуального адресного простору фіксованого й порівняно невеликого розміру.
- *Сегментна віртуальна пам'ять*. Переміщення здійснюється сегментами — частинами віртуального адресного простору довільного розміру.
- *Сегментно-сторінкова віртуальна пам'ять.* Використовується дворівневий розподіл: віртуальний адресний простір ділиться на сегменти, а потім сегменти діляться на сторінки. Одиницею переміщення тут є сторінка.

Для тимчасового зберігання сегментів і сторінок на диску надається або спеціальна ділянка чи спеціальний файл, що зветься сторінковим файлом. Поточний розмір сторінкового файлу є важливим параметром ОС. Чим він більший, тим більше програм можуть одночасно виконуватися ОС. Однак, зі збільшенням сторінкового файлу збільшується час на перекачування інформації

й загальна корисна продуктивність системи зменшується. У сучасних ОС розмір сторінкового файлу є настроюваним параметром.

Алгоритми розподілу пам’яті

Алгоритми розподілу пам'яті розділені на два класи: алгоритми, у яких використовується переміщення частин процесів між оперативною пам'яттю й диском, і алгоритми, у яких зовнішня пам'ять не залучається (рис. 3.2).

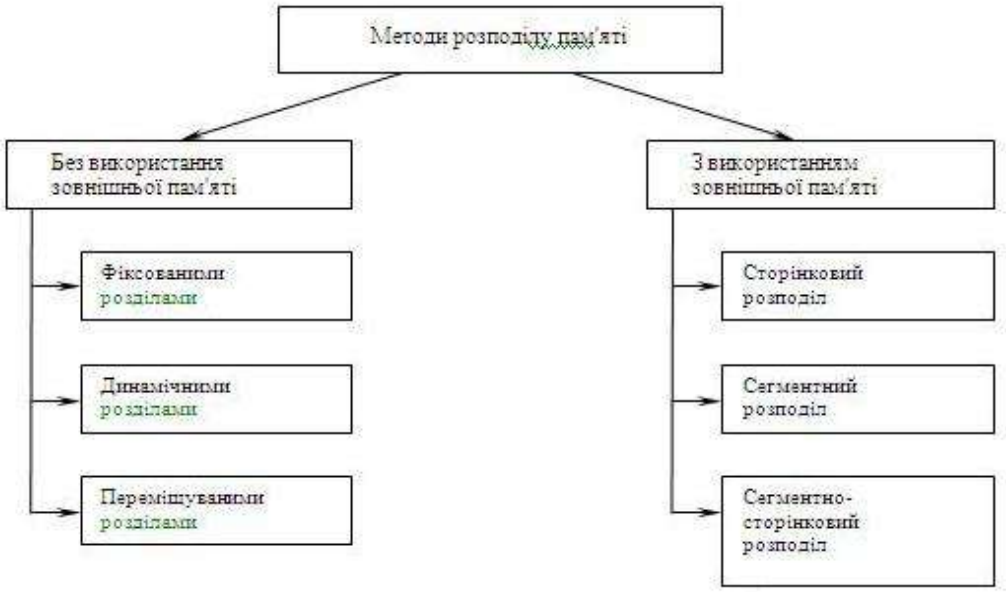

Рис 3.2 – Класиікація методів розподілу памяті

Розподіл пам'яті фіксованими розділами

Це найпростіший спосіб керування пам'яттю. Пам'ять поділяється на декілька областей фіксованого розміру, називаних *розділами* (рис. 3.3). Це розбиття може бути виконане вручну оператором під час старту системи або її установки. Потім границі розділів не змінюються. Новий процес, що надійшов на виконання, поміщається або у загальну чергу, або у чергу до деякого розділу.

Система керування пам'яттю вирішує наступні завдання:

- Порівнює обсяг пам'яті, необхідний для нового процесу, з розмірами вільних розділів і вибирає підходящий розділ.
- Здійснює завантаження програми в один з розділів і настроювання адрес.

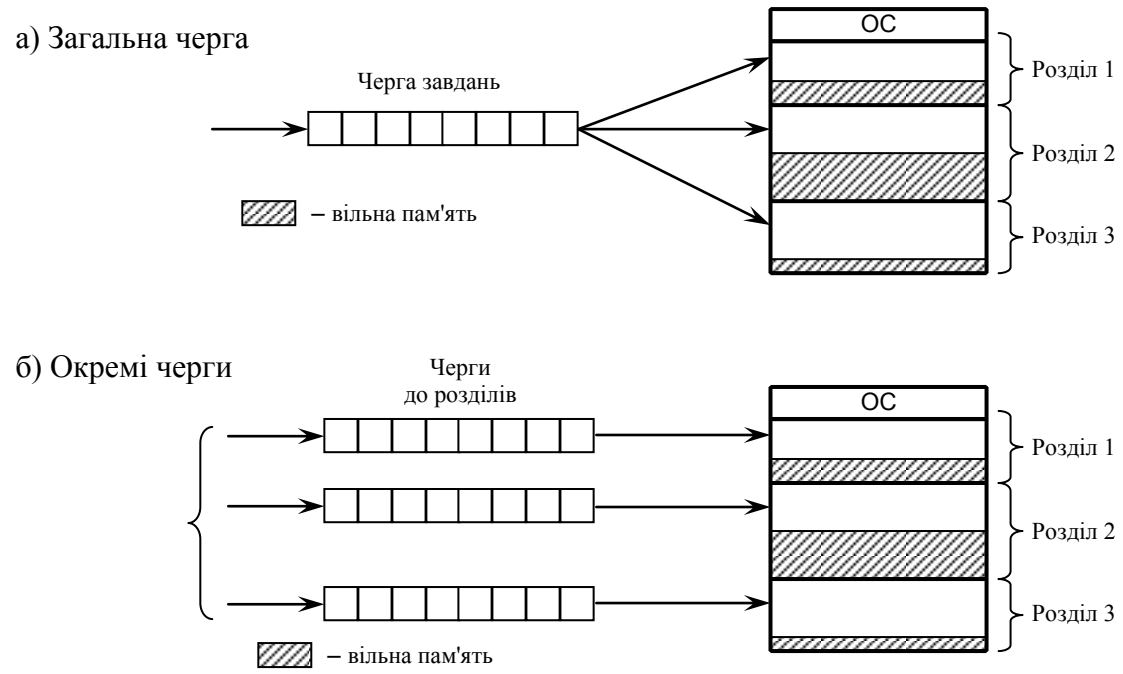

Рис. 3.3 – Розподіл пам'яті фіксованими розділами

Уже на етапі трансляції розробник програми може задати розділ, у якому її варто виконувати. Це дозволяє одразу без використання переміщуючого завантажувальника одержати машинний код, настроєний на конкретну область пам'яті. Істотний недолік – жорсткість прив'язки до певної адреси. Рівень мультипрограмування заздалегідь обмежений числом розділів. Незалежно від розміру програми, вона буде займати весь розділ. З іншої сторони, процес, що вимагає кілька розділів, не може бути виконаний. Цей найпростіший метод розподілу пам'яті зараз знаходить застосування тільки у системах реального часу завдяки детермінованості обчислювального процесу.

Розподіл пам'яті динамічними розділами

Кожному готовому для виконання процесу виділяється вся необхідна йому пам'ять, а якщо її не вистачає – процес не запускається.

Функції ОС для реалізації такого розподілу полягають у наступному:

- Ведення таблиць вільних і зайнятих областей, де вказуються початкові адреси й розміри ділянок пам'яті.

- При створенні нового процесу – аналіз вимог до пам'яті, перегляд таблиці вільних областей і вибір розділу. Вибір розділу може здійснюватися за різними правилами: перший знайдений розділ достатнього розміру, розділ з найменшим достатнім розміром, розділ з найбільшим достатнім розміром.
- Завантаження програми у наданий їй розділ.
- Коригування таблиць вільних і зайнятих областей.

Цьому методу властивий серйозний недолік – *фрагментація пам'яті*.

Фрагментація – наявність великої кількості несуміжних ділянок вільної пам'яті такого маленького розміру, що жодна з нових програм не може розміститися в них. В той же час, сумарний обсяг вільних фрагментів може перевищувати необхідний обсяг пам'яті.

Розподіл пам'яті динамічними розділами

Боротьба із фрагментацією полягає у переміщенні всіх зайнятих ділянок убік старших або молодших адрес таким чином, щоб уся вільна пам'ять утворила єдину вільну область. На додаток до функцій, які виконує ОС із динамічними розділами, вона ще час від часу копіює вміст розділів з одного місця пам'яті в інше, коригуючи таблиці вільних і зайнятих областей (рис.3.4).

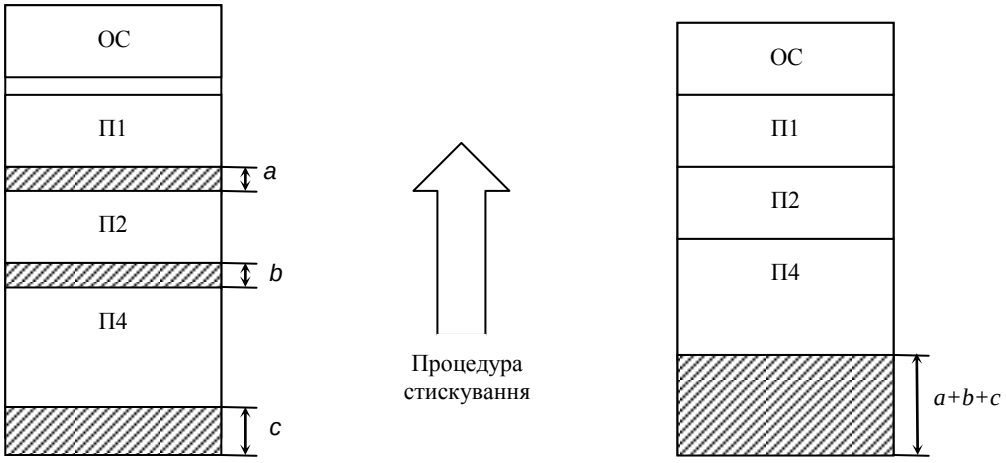

Рис. 3.4 – Розподіл пам'яті переміщуваними розділами

Ця процедура називається *стискуванням*. Стискування виконується або при кожному завершенні процесу, або якщо для створюваного процесу немає

вільного розділу достатнього розміру. Оскільки програми переміщаються в оперативній пам'яті в процесі свого виконання, неможливо виконати настроювання адрес за допомогою переміщуючого завантажувальника. Тут необхідна реалізація динамічного перетворення адрес.

Хоча цей метод призводить до більш ефективного використання пам'яті, проте він вимагає значних накладних витрат, що переважує його позитиви.

Сторінковий розподіл

Віртуальний адресний простір кожного процесу поділяється на частини однакового, фіксованого для даної системи розміру – *віртуальні сторінки.* У загальному випадку, розмір віртуального адресного простору процесу не є кратним розміру сторінки. Тому остання сторінка кожного процесу доповнюється фіктивною областю. Вся оперативна пам'ять також поділяється на частині цього ж розміру, називані *фізичними сторінками* (рис. 3.5).

Розмір сторінки вибирається як степінь двійки: 512, 1024, 2048, 4096 байт тощо, що дозволяє спростити механізм перетворення адрес.

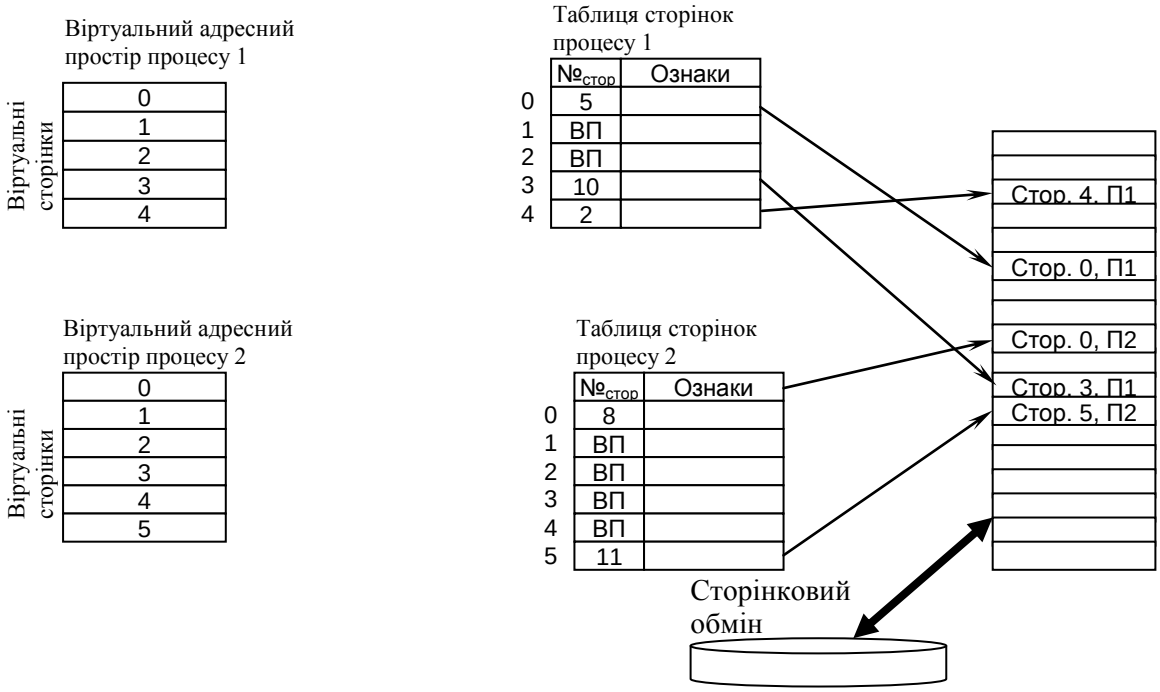

Рис. 3.5 – Сторінковий розподіл пам'яті

Для кожного процесу ОС створює *таблицю сторінок* — інформаційну структуру, що містить записи про всі віртуальні сторінки процесу. Запис таблиці, названий *дескриптором сторінки*, включає наступну інформацію:

- *номер фізичної сторінки*, у яку завантажена дана віртуальна сторінка;
- *ознака присутності* (присвоюється 1, якщо віртуальна сторінка перебуває в оперативній пам'яті);
- *ознака модифікації сторінки*, (присвоюється 1 щораз при виконанні запису за адресою, що відноситься до даної сторінки);
- *ознака звертання до сторінки* (біт доступу), що встановлюється в 1 при кожному звертанні за адресою, що відноситься до даної сторінки.

Ознаки присутності, модифікації й звертання у більшості сучасних процесорів встановлюються апаратно при операціях з пам'яттю. Інформація з таблиць сторінок використовується для рішення щодо необхідності переміщення тієї чи іншої сторінки між пам'яттю й диском, а також для перетворення віртуальної адреси у фізичну.

Таблиці сторінок розміщуються в оперативній пам'яті. Адреса таблиці сторінок включається до контексту відповідного процесу. При активізації чергового процесу ОС завантажує адресу його таблиці сторінок у спеціальний регістр процесора.

При кожному звертанні до пам'яті виконується пошук номера віртуальної сторінки, що відповідає адресі запиту. Потім за цим номером визначається потрібний елемент таблиці сторінок, і з нього витягається інформація, що описує сторінку. Аналізується ознака присутності, і, якщо сторінка перебуває в оперативній пам'яті, виконується перетворення віртуальної адреси у фізичну. Якщо ж потрібна віртуальна сторінка у цей момент вивантажена на диск, відбувається *сторінкове переривання.* Виконуваний процес переводиться у стан очікування, і активізується інший процес із черги процесів зі станом готовності. Паралельно програма обробки сторінкового переривання знаходить на диску необхідну віртуальну сторінку й намагається завантажити її в оперативну пам'ять. Якщо у пам'яті є вільна фізична сторінка, завантаження виконується

негайно. Якщо ж вільних сторінок немає, на підставі прийнятої у даній системі стратегії заміщення сторінок, вирішується питання про те, яку саме сторінку варто вивантажити з оперативної пам'яті.

Після визначення віртуальної сторінки, її біту присутності присвоюється 0 й аналізується її ознака модифікації. Якщо за час останнього перебування в оперативній пам'яті вона була модифікована, її нова версія повинна бути переписана на диск. Якщо ж ні, тоді ніякого запису на диск не робиться (вона й так там присутня). Фізична сторінка оголошується вільною.

Віртуальна адреса при сторінковому розподілі може бути представлена у вигляді пари (p, s_v), де p — порядковий номер віртуальної сторінки процесу (нумерація сторінок починається з 0), а s_v — зсув у межах віртуальної сторінки.

Фізична адреса також може бути представлена у вигляді пари (n, S_f), де n — номер фізичної сторінки, а S_f — зсув у фізичній сторінці. Завдання підсистеми віртуальної пам'яті полягає у перетворенні пари (p, s_v) у (n, S_f).

Розглянемо дві базисні властивості сторінкової організації.

Перша з них полягає в тому, що розмір сторінки дорівнює степені двійки — 2^k . Із цього випливає, що зсув s може бути отримано простим відділенням k молодших розрядів у двійковому записі адреси, а решта старших розрядів являтимуть собою двійковий запис номера сторінки. Наприклад:

Розмір сторінки 1 Кбайт (2^{10}). Двійковий код адреси – $1010\ 0011\ 1001_2$. Адреса належить сторінці 10_2 і зміщена відносно її початку на $10\ 0011\ 1001_2$ байт.

Друга властивість полягає в тому, що в межах сторінки неперервна послідовність віртуальних адрес однозначно відображається у неперервну послідовність фізичних адрес, тобто $s_v = S_f$.

Звідси випливає проста схема перетворення віртуальних адрес у фізичні.

Молодші розряди фізичної адреси, що відповідають зсуву, отримують перенесенням такої ж кількості молодших розрядів з віртуальної адреси. Старші розряди фізичної адреси, що відповідають номеру фізичної сторінки,

визначаються з таблиці сторінок, де вказується відповідність віртуальних і фізичних сторінок.

Якщо відбувається звертання до пам'яті за деякою віртуальною адресою, апаратними схемами процесора виконуються наступні дії (рис. 3.6):

- зі спеціального регістра процесора витягається адреса АТ таблиці сторінок активного процесу. На основі початкової адреси таблиці сторінок, номера віртуальної сторінки p (старші розряди віртуальної адреси) і довжини окремого запису в таблиці сторінок L (системна константа) визначається адреса потрібного дескриптора у таблиці сторінок: a =AT+(px).
- із цього дескриптора береться номер відповідної фізичної сторінки n.
- до номера фізичної сторінки приєднується зсув s (молодші розряди віртуальної адреси)

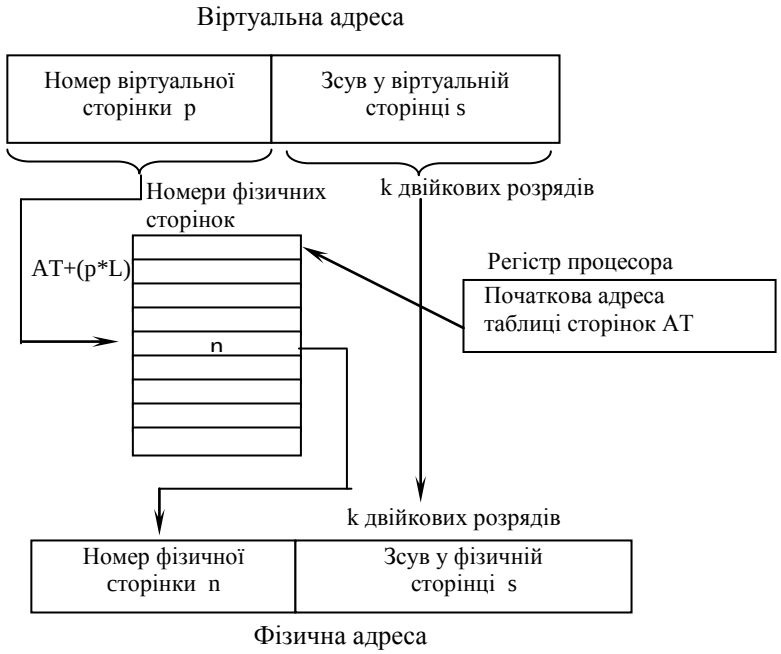

Рис. 3.6 – Перетворення адреси при сторінковому розподілі пам'яті

Апаратний механізм отримання фізичної адреси за віртуальною передбачений для зменшення часу перетворення адрес. Іншим важливим фактором, що впливає на продуктивність системи, є частота сторінкових переривань. На неї, у свою чергу, впливає розмір сторінки й прийняті в даній системі правила вибору сторінок для вивантаження й завантаження.

При виборі сторінки на вивантаження використовується наступний критерій: на диск виштовхується сторінка, до якої найдовше не буде звертань. Однак, точно пророчити хід обчислювального процесу неможливо, а отже і точно визначити сторінку, що підлягає вивантаженню. Рішення приймається на основі деяких емпіричних критеріїв. Наприклад, якщо сторінка не використовувалася довгий час, робиться висновок про те, що вона не буде потрібна й зараз.

Найбільш популярним критерієм є число звертань до сторінки за останній період часу. ОС веде для кожної сторінки програмний лічильник. Коли виникає необхідність видалити якусь сторінку з пам'яті, ОС знаходить сторінку, лічильник звертань якої має найменше значення. Усі лічильники періодично обнуляються.

Розмір сторінки також впливає на кількість записів у таблицях сторінок процесів. Так у сучасних процесорах максимальний обсяг віртуального адресного простору процесу не менший за 4 Гбайт (2^{32}), тому при розмірі сторінки 4 Кбайт (2^{12}) і довжині запису 4 байти для зберігання таблиці сторінок потрібно 4 Мбайт пам'яті! Виходом з такої ситуації є зберігання у пам'яті тільки тієї частини таблиці сторінок, що активно використовуються у даний період часу, тобто таблицю сторінок можна тимчасово витісняти з оперативної пам'яті.

Саме такий результат може бути досягнутий шляхом ускладнення структуризації віртуального адресного простору, при якому вся сукупність віртуальних адрес процесу ділиться на *розділи,* а розділи діляться на *сторінки* (рис. 3.7). Усі сторінки мають однаковий розмір, а розділи містять однакову кількість сторінок. Якщо розмір сторінки й кількість сторінок у розділі вибрати як степінь двійки (2^k й 2^n відповідно), то приналежність віртуальної адреси до розділу й сторінки, а також зсув усередині сторінки визначається дуже просто: молодші *k* двійкових розрядів дають *зсув*, наступні *n* розрядів – *номер віртуальної сторінки*, а решта старших розрядів містять *номер розділу*. Для кожного розділу будується власна *таблиця сторінок*. Кількість дескрипторів у таблиці та їх розмір підбираються таким чином, щоб розмір таблиці дорівнював

розміру сторінки. Наприклад, у процесорі Pentium при розмірі сторінки 4 Кбайт довжина дескриптора становить 4 байти, а кількість записів у таблиці сторінок, що вміщується на сторінку, дорівнює 1024. Кожна таблиця сторінок описується дескриптором, структура якого повністю збігається зі структурою дескриптора звичайної сторінки. Ці дескриптори зведені у таблицю розділів, називану також каталогом сторінок. Фізична адреса таблиці розділів активного процесу міститься у спеціальному регістрі процесора й тому завжди відома ОС. Сторінка з таблицею розділів ніколи не вивантажується з пам'яті.

Вивантаження сторінок з таблицями сторінок дозволяє економити пам'ять, проте призводить до додаткових витрат при отриманні фізичної адреси. Дійсно, може трапитися так, що саме таблиця сторінок з потрібним дескриптором у цей момент вивантажена на диск. Тоді процес перетворення адреси призупиняється до завантаження необхідної сторінки у пам'ять.

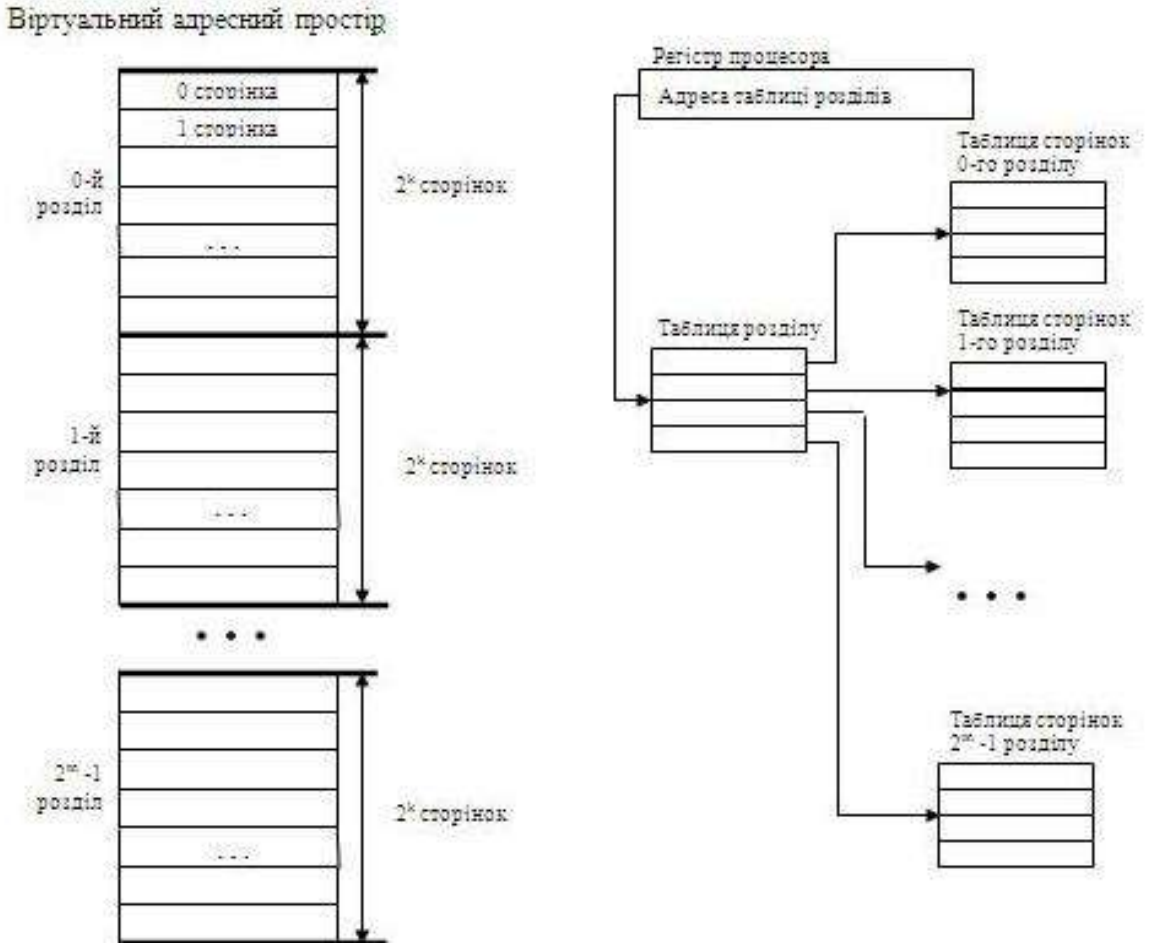

Рис. 3.7 – Сторінковий розподіл пам'яті з розділами

Для зменшення ймовірності відсутності сторінки у пам'яті використовують різні прийоми, основним з яких є кешування (див. нижче).

Розглянемо докладніше схему перетворення адрес для випадку дворівневої структуризації віртуального адресного простору (рис. 3.8):

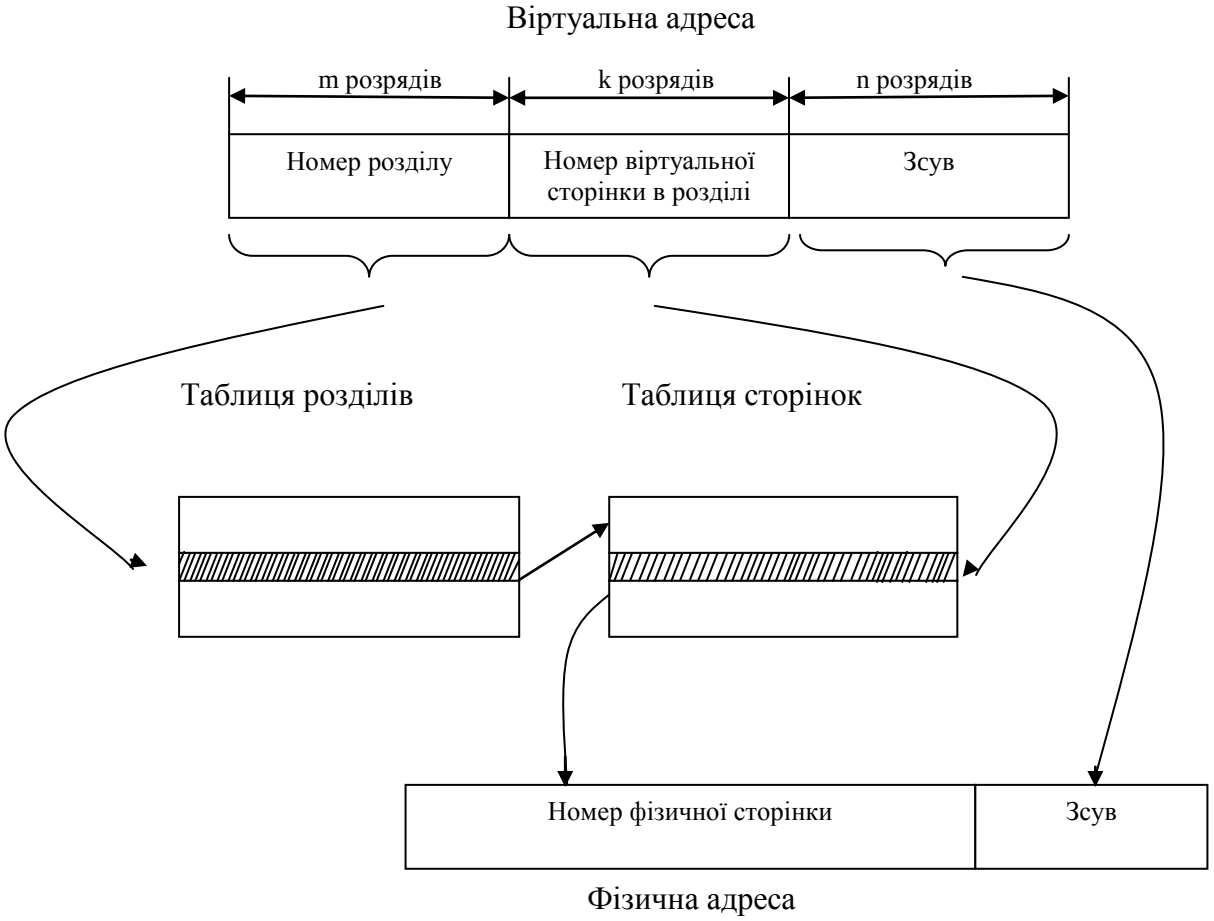

Рис. 3.8 – Перетворення віртуальної адреси у фізичну при сторінковому розподілі з використанням розділів

Номер розділу, до якого належить віртуальна адреса, визначається шляхом відкидання $k+n$ молодших розрядів у віртуальній адресі. Далі з таблиці розділів витягається дескриптор відповідної таблиці сторінок. Перевіряється, чи перебуває дана таблиця сторінок у пам'яті. Якщо ні – відбувається сторінкове переривання, й система завантажує потрібну сторінку з диску. Далі із таблиці сторінок витягається дескриптор віртуальної сторінки, номер якої міститься у середніх n розрядах перетвореної віртуальної адреси. Знову виконується перевірка наявності даної сторінки у пам'яті й при необхідності її завантаження.

З дескриптора визначається номер (базова адреса) фізичної сторінки, в яку завантажена дана віртуальна сторінка. До номера фізичної сторінки пристиковується зсув, узятий з k молодших розрядів віртуальної адреси. Таким чином, отримаємо шукану фізичну адресу.

Сегментний розподіл

При сторінковій організації віртуальний адресний простір процесу ділиться на рівні частини *механічно,* без урахування змісту даних. Однак, розбиття віртуального адресного простору на «змістовні» частини робить принципово можливим спільне використання фрагментів програм різними процесами. Наприклад, двом процесам потрібно одна й та ж сама підпрограма, яка до того ж є реєнтерабельною. Тоді коди цієї підпрограми можуть бути оформлені у вигляді окремого сегмента й включені у віртуальні адресні простори обох процесів. При відображенні у фізичну пам'ять сегменти, що містять коди підпрограми з обох віртуальних просторів, проектуються на одну й ту ж саму область фізичної пам'яті. Таким чином, обидва процеси одержать доступ до однієї й тієї ж копії підпрограми. Віртуальний адресний простір процесу ділиться на частині — *сегменти* (рис.3.9).

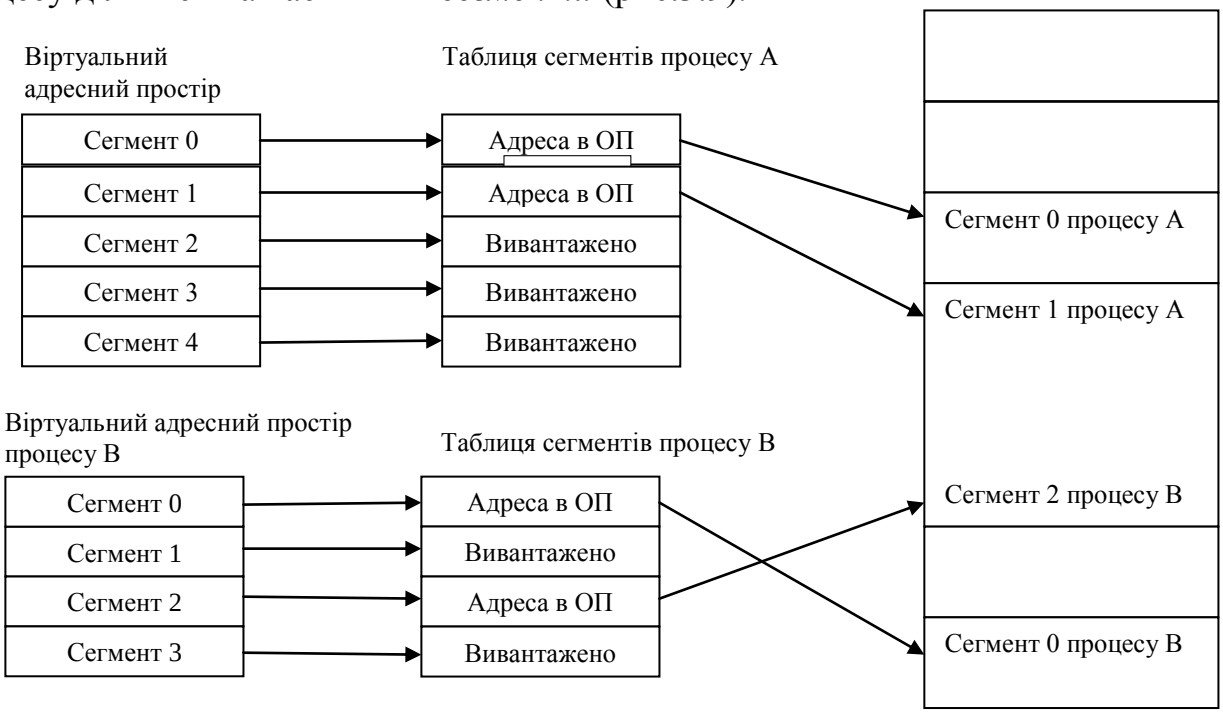

Рис. 3.9 – Перетворення віртуальної адреси на фізичну при сегментному розподілі пам'яті

Розмір сегментів визначається з урахуванням їх змісту. Окремий сегмент може являти собою підпрограму, масив даних тощо. Поділ віртуального адресного простору на сегменти здійснюється компілятором на основі вказівок програміста або за замовчуванням, на основі прийнятих у системі угод. Максимальний розмір сегменту визначається розрядністю віртуальної адреси, наприклад, при 32-розрядній організації процесора він дорівнює 4 Гбайт. При цьому, максимально можливий віртуальний адресний простір процесу являє собою набір з N віртуальних сегментів, кожен розміром 4 Гбайт. Для кожного сегмента діапазон віртуальних адрес складає 00000000_{16} … $FFFFFFFF_{16}$.

Сегменти не упорядковуються відносно один одного, тому для них не існує загальної лінійної віртуальної адреси. Віртуальна адреса задається парою чисел: *номером сегмента й лінійною віртуальною адресою*.

При завантаженні процеса в оперативну пам'ять записується тільки частина його сегментів, а повна копія віртуального адресного простору перебуває у дисковій пам'яті. Для кожного сегмента, що завантажується, ОС шукає неперервну ділянку вільної пам'яті достатнього розміру. Суміжні у віртуальній пам'яті сегменти одного процесу можуть займати в оперативній пам'яті несуміжні ділянки. Якщо під час виконання процесу відбувається звертання за відсутньою у цей час у пам'яті віртуальною адресою (що відноситься до сегменту), відбувається переривання. ОС припиняє активний процес, запускає на виконання наступний процес із черги, а паралельно організує завантаження потрібного сегмента. При відсутності у пам'яті місця, ОС вибирає сегмент на вивантаження.

На етапі створення процесу при завантаженні його образу в оперативну пам'ять система створює *таблицю сегментів* процесу (подібну до таблиці сторінок), де для кожного сегмента вказується:

- базова фізична адреса сегмента в оперативній пам'яті;
- розмір сегмента;
- правила доступу до сегмента;

- ознаки модифікації, присутності й звертання до даного сегмента, а також деяка інша інформація.

Якщо віртуальні адресні простори декількох процесів включають один і той самий сегмент, тоді у таблицях сегментів цих процесів робляться однакові посилання на ділянку оперативної пам'яті, де даний сегмент завантажується в одиничному екземплярі.

Механізми перетворення адрес при сторінковому й сегментному способах організації пам'яті досить схожі. Однак у них є й істотні відмінності. Ці відмінності є наслідком того, що сегменти, на відміну від сторінок, мають довільний розмір.

Віртуальна адреса при сегментній організації може бути представлена парою *(g, s)*, де *g* – номер сегмента, а *s* – зсув (рис. 3.10). Сторінки мають однаковий розмір, звідси їх початкові адреси є кратними цьому розміру. Тому ОС заносить у таблиці сторінок не повні адреси, а номери фізичних сторінок, які збігаються зі старшими розрядами базових адрес.

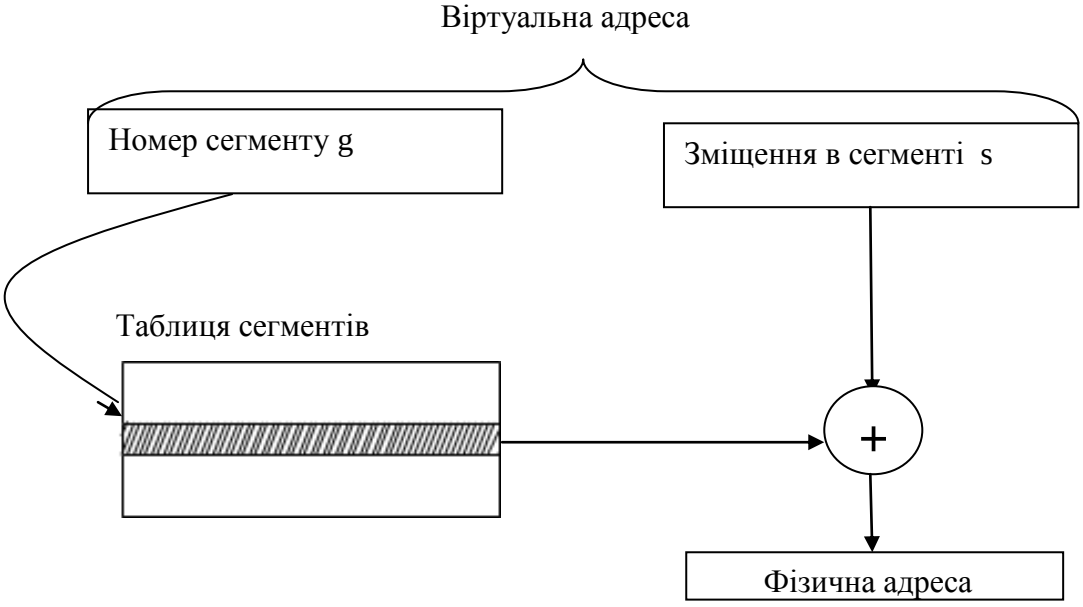

Рис. 3.10 – Перетворення віртуальної адреси у фізичну адресу при сегментному розподілі пам'яті

У загальному випадку, сегмент може розташовуватися у фізичній пам'яті, починаючи з будь-якої адреси. Отже, для визначення місця розташування сегмент у пам'яті необхідно задавати його повну початкову фізичну адресу.

Використання операції додавання сповільнює процедуру перетворення віртуальної адреси у фізичну порівняно зі сторінковою організацією (там має місце просте об'єднання старших і молодших розрядів). Іншим недоліком сегментного розподілу є надмірність. При сегментній організації одиницею переміщення між пам'яттю й диском є сегмент, розмір якого, як правило, більший ніж сторінка. Але в багатьох випадках для роботи програми не потрібно завантажувати весь сегмент повністю, досить було б однієї або двох сторінок. Також при відсутності вільного місця не слід вивантажувати цілий сегмент, якщо можна обмежиться декількома сторінками. Але головний недолік – це фрагментація, що виникає через непередбачуваність розмірів сегментів.

Найбільш істотною відмінністю сегментної організації від сторінкової є можливість завдання диференційованих прав доступу процесу до його сегментів. Наприклад, сегмент даних вихідної інформації може мати права доступу «тільки читання», а сегмент результатів – «читання й запис». Ця властивість визначає перевагу перед сторінковою організацією.

Сегментно-сторінковий розподіл

Даний метод є спробою об'єднати переваги обох методів. Як і при сегментній організації пам'яті, віртуальний адресний простір процесу поділяється на сегменти. Це дозволяє визначати різні права доступу до різних даних. Проте переміщення даних між пам'яттю й диском здійснюється не сегментами, а сторінками. Для цього кожен сегмент і фізична пам'ять діляться на сторінки однакового розміру, що дозволяє мінімізувати фрагментацію

У більшості сучасних реалізацій сегментно-сторінкової організації всі віртуальні сегменти утворюють один неперервний лінійний віртуальний адресний простір (рис.3.11).

Координати байта при цьому можна задати двома способами. По-перше,

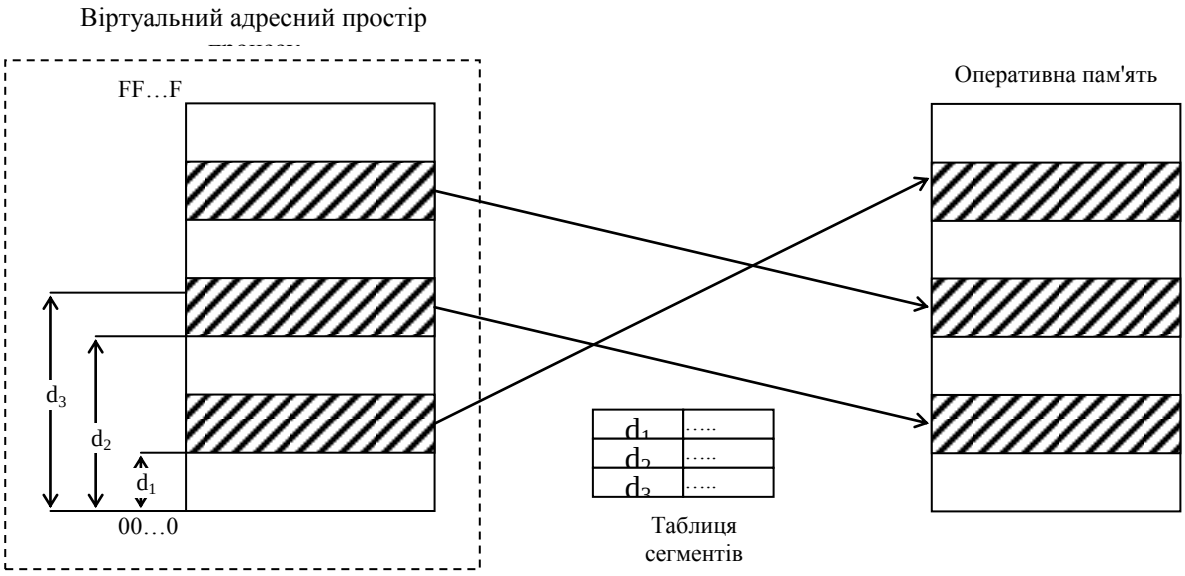

Рис. 3.11 – Сегментно-сторінковий розподіл пам'яті

лінійною віртуальною адресою, що дорівнює зсуву даного байта відносно початку загального лінійного віртуального простору, по-друге, парою чисел, одне з яких є номером сегмента, а інше – зсувом відносно початку сегмента. При цьому, для однозначного завдання віртуальної адреси другим способом, на відміну від сегментної моделі, необхідно вказати також початкову віртуальну адресу сегмента з даним номером. Найчастіше використовується другий спосіб, оскільки він дозволяє безпосередньо визначити приналежність адреси деякому сегменту й перевірити права доступу процесу до нього.

Для кожного процесу ОС створює окрему таблицю сегментів, у якій утримуються дескриптори (описувачі) усіх сегментів процесу. Опис сегмента включає призначені йому права доступу й інші характеристики, що подібні тим, які містяться в дескрипторах сегментів при сегментній організації пам'яті. Однак є і відмінності. У полі базової адреси вказується не початкова фізична адреса сегмента, а початкова лінійна віртуальна адреса сегмента у просторі віртуальних адрес. Це дозволяє однозначно перетворити адресу, задану парою (g, s), у лінійну віртуальну адресу байта, що потім перетвориться у фізичну адресу сторінковим механізмом. Розподіл загального лінійного віртуального адресного простору процесу й фізичної пам'яті на сторінки здійснюється так само, як це робиться при сторінковій організації пам'яті.

Базові адреси таблиці сегментів і таблиці сторінок процесу є частиною його контексту. При активізації процесу ці адреси завантажуються у спеціальні регістри процесів і використовуються механізми перетворення адрес.

Перетворення віртуальної адреси у фізичну має два етапи (рис. 3.12):

1) *На першому етапі* працює механізм *сегментації*. Вихідна віртуальна адреса пари (*g*, *s*) перетворюється у лінійну віртуальну адресу.

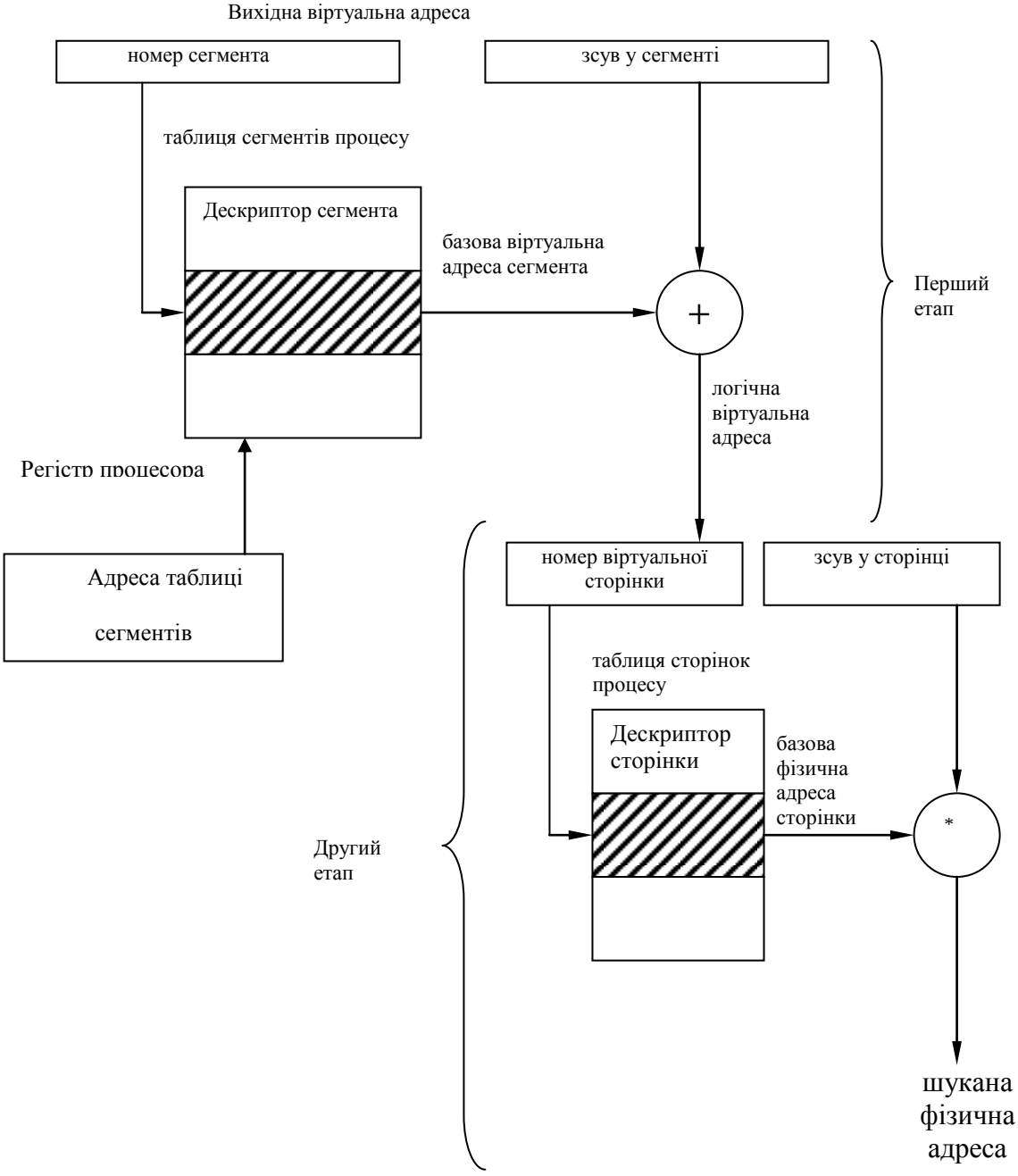

Рис. 3.12 – Перетворення віртуальної адреси на фізичну при сегментно-сторінковому розподілі

Для цього на підставі базової адреси сегментів і номера сегмента обчислюється поле дескриптора й виконується перевірка можливості виконання заданої операції. Якщо доступ до сегмента дозволено, обчислюється лінійна віртуальна адреса шляхом додавання базової адреси сегмента, витягнутої з дескриптора, і зсуву, заданого у вихідній віртуальній адресі.

2) *Другий етап* використовує *сторінковий механізм*. Лінійна віртуальна адреса перетворюється у фізичну. Як результат перетворення, лінійна віртуальна адреса представляється як при сторінковій організації пам'яті (пара: номер сторінки, зсув у сторінці). А далі, як вже визначалося раніше. Старші розряди лінійної віртуальної адреси, що містять номер віртуальної сторінки, замінюються номером фізичної сторінки з таблиці сторінок, а молодші розряди віртуальної адреси, що містять зсув, залишаються без зміни.

Як слідує з розгляду, механізм сегментації й сторінковий розподіл діють досить незалежно один від другого.

Кешування даних

Пам'ять ЕОМ – це ієрархія запам’ятовуючих пристроїв (ЗП), що відрізняються середнім часом доступу до даних, обсягом і вартістю зберігання 1 біта інформації. Фундаментом цієї піраміди є пам'ять на жорстких дисках, але час доступу до диску обчислюється мілісекундами. Оперативна пам'ять має час доступу 10-20 нс (від декількох мегабайт до декількох гігабайт). Надоперативна пам'ять (десятки-сотні кілобайт) – 8 нс. Внутрішні регістри процесора – кілька десятків байт з часом доступу 2-3 нс.

Кеш пам'ять або просто *кеш* є способом спільного функціонування двох типів ЗП, що відрізняються часом доступу й вартістю зберігання даних. Динамічне копіювання найчастіше використовуваної інформації з «повільного» ЗП у швидкі ЗП дозволяє, з одного боку, зменшити середній час доступу до даних, з іншого боку – заощадити дорожчу швидкодіючу пам'ять. Особливістю кешування є те, що система не вимагає ніякої зовнішньої інформації про інтенсивність використання даних. Ні користувачі, ні програми не приймають

ніякої участі в переміщенні даних із запам’ятовуючого пристрою одного типу в ЗП іншого типу – все це робиться автоматично системними засобами.

Кешем часто називають не тільки спосіб організації двох типів ЗП, але й один із пристроїв – «швидкий ЗП». Він дорожчий й порівняно невеликого обсягу на противагу «повільному» ЗП – оперативної пам'яті. Якщо *кешування* використовують для зменшення середнього часу доступу до оперативної пам'яті, тоді в якості кешу використовують більш дорогу й швидкодіючу статичну пам'ять.

Якщо *кешування* використовується системою введення-виведення для прискорення доступу до даних, що *зберігаються на диску***,** роль кеш пам'яті грають буфери, що реалізовані в оперативній пам'яті.

Віртуальну пам'ять теж можна розглядати як окремий випадок *кешування*.

Принцип дії кеш пам'яті

Вміст кеш пам'яті являє собою сукупність записів про всі завантажені в неї елементи даних з основної пам'яті. Кожен запис включає:

- елемент даних;
- адресу елемента в основній пам'яті;
- додаткову інформацію для реалізації алгоритму, яка включає ознаку модифікації й ознаку дійсності даних.

При кожному звертанні до основної пам'яті за фізичною адресою проглядається вміст кеш пам'яті з метою визначення наявності там потрібних даних.

Кеш пам'ять є безадресною, тому пошук потрібних даних здійснюється за вмістом – за взятим із запиту значенням поля адреси в оперативній пам'яті.

Далі можливі два варіанти розвитку:

- дані виявлені в кеш пам'яті, тобто має місце *кеш-влучання* (*cache-hit*), вони зчитуються й передаються джерелу запиту;

- потрібні дані відсутні, тобто відбувся *кеш-промах* (*cache-miss*), вони зчитуються з основної пам'яті, передаються джерелу запиту й одночасно копіюються в кеш пам'ять (рис 3.13).

Ефективність кешування залежить від ймовірності влучення в кеш.

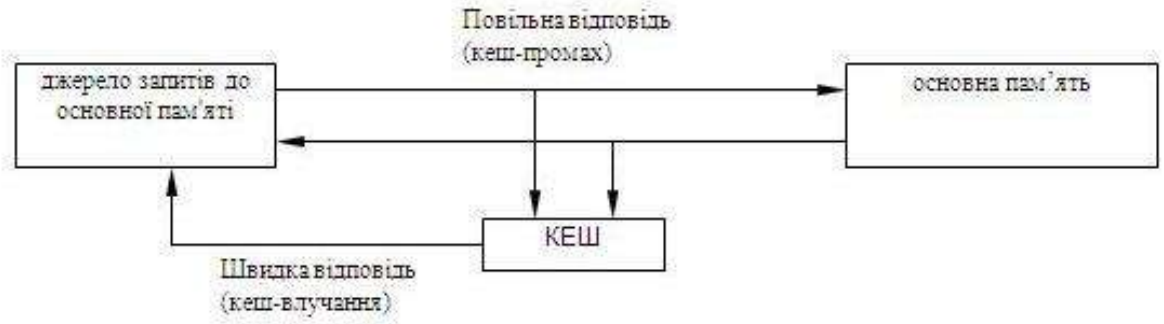

Рис. 3.13 – Реалізація запиту при наявності кеш пам’яті

Якщо позначити ймовірність кеш-влучання через *p*, а час доступу до основної пам'яті через *t1*, час доступу до кеш через *t2*, то за формулою повної ймовірності середній час доступу дорівнюватиме:

$t = t2p + t1(1-p)$

Якщо $p=1$, час доступу дорівнює t_2.

Ймовірність виявлення даних у кеші залежить від різних факторів:

- обсягу кеша;
- обсягу кешуємої пам'яті;
- алгоритму заміщення даних у кеші;
- особливостей виконуваної програми й т.п.

На практиці відсоток влучень виявляється досить високим – порядку 90%. Такий відсоток обумовлюється наявністю в даних об'єктивних властивостей, а саме *просторової* й *тимчасової л*окальності.

Просторова локальність. Якщо відбулося звертання за деякою адресою, тоді з високою ймовірністю найближчим часом відбудеться звертання за сусідніми адресами.

Часова локальність. Якщо відбулося звертання за деякою адресою, тоді наступне звертання за тією ж самою адресою з великою ймовірністю відбудеться найближчим часом.

Оскільки при виконанні програми дуже висока ймовірність того, що команди вибираються з пам'яті одна за одною із сусідніх комірок, має сенс завантажувати в кеш цілий фрагмент програми. Аналогічно і з масивами даних.

У процесі роботи вміст кеш пам'яті постійно оновлюється. Витиснення даних означає або просто оголошення вільною деякої області кеш пам'яті (скидання біта дійсності) або на додаток до цього копіювання даних в основну пам'ять, якщо вони модифікувалися.

Наявність у комп'ютері двох копій даних – в основній пам'яті й у кеші – породжує проблему узгодження даних.

Існують два підходи до рішення цієї проблеми:

- *наскрізний запис* (*write through*). При запиті до основної пам'яті (у тому числі, при записі) проглядається кеш. Якщо дані за запитуваною адресою відсутні, запис виконується тільки в основну пам'ять. Якщо дані перебувають у кеші, запис робиться й у кеш, й у пам'ять.
- *зворотний запис* (*write back*). Виконується перегляд кешу та, при відсутності там даних, запис робиться в основну пам'ять. У противному випадку, запис робиться тільки в кеш. При цьому встановлюється ознака модифікації. При витисненні даних з кешу ці дані будуть переписані в основну пам'ять.

Способи відображення основної пам'яті на КЕШ

Алгоритми пошуку й заміщення даних у кеші залежать від того, як основна пам'ять відображається на кеш.

Використовуються дві схеми відображення:

- *випадкове відображення*;
- *детерміноване відображення*.

При *випадковому відображенні* елемент оперативної пам'яті може бути розміщений у довільному місці кеш пам'яті (разом зі своєю адресою в оперативній пам'яті). Пошук інформації здійснюється за цією адресою. Це – процедури простого перебору адрес при великих тимчасових витратах. Тому використовується так званий *асоціативний пошук*, коли порівняння виконується не послідовно з кожним записом у кеші, а паралельно з усіма його записами. Ознака, за якою виконується порівняння, називається *тегом* (*tag*). У даному випадку, тегом є адреса даних в оперативній пам'яті. Апаратна реалізація такого пошуку значно здорожчує кеш пам'ять. Тому цей метод використовується для забезпечення високого відсотка влучення при невеликому обсязі кеш пам'яті.

У кешах на основі *випадкового відображення*, витиснення старих даних відбувається тільки тоді, коли в кеш пам'яті немає вільного місця. Вибір даних на вивантаження ґрунтується на тих самих принципах, що й при заміщенні сторінок (давно немає звертань, найменша кількість звертань тощо).

При *детермінованому* способі відображення, будь-який елемент основної пам'яті відображається в одне й те ж саме місце кеш пам'яті. У цьому випадку, кеш пам'ять розділена на рядки, кожен з яких призначений для зберігання одного запису про один елемент даних і має свій номер.

Між номерами рядків кеш пам'яті й адресами оперативної пам'яті встановлюється відповідність «*один до багатьох*»: одному номеру рядка відповідає декілька (досить багато) адрес оперативної пам'яті. Для відображення може використовуватися просте виділення декількох розрядів з адреси оперативної пам'яті, які інтерпретуються як номер рядка кеш пам'яті. Таке відображення називається *прямим*. Наприклад, кеш розрахований на 1024 записи (1024 рядки). Тоді будь-яка адреса оперативної пам'яті може бути відображена на адресу кеш пам'яті простим відділенням 10 двійкових розрядів (рис. 3.14).

При пошуку даних у кеші використовується швидкий прямий доступ до запису за номером рядка, отриманим з адреси оперативної пам'яті із запиту.

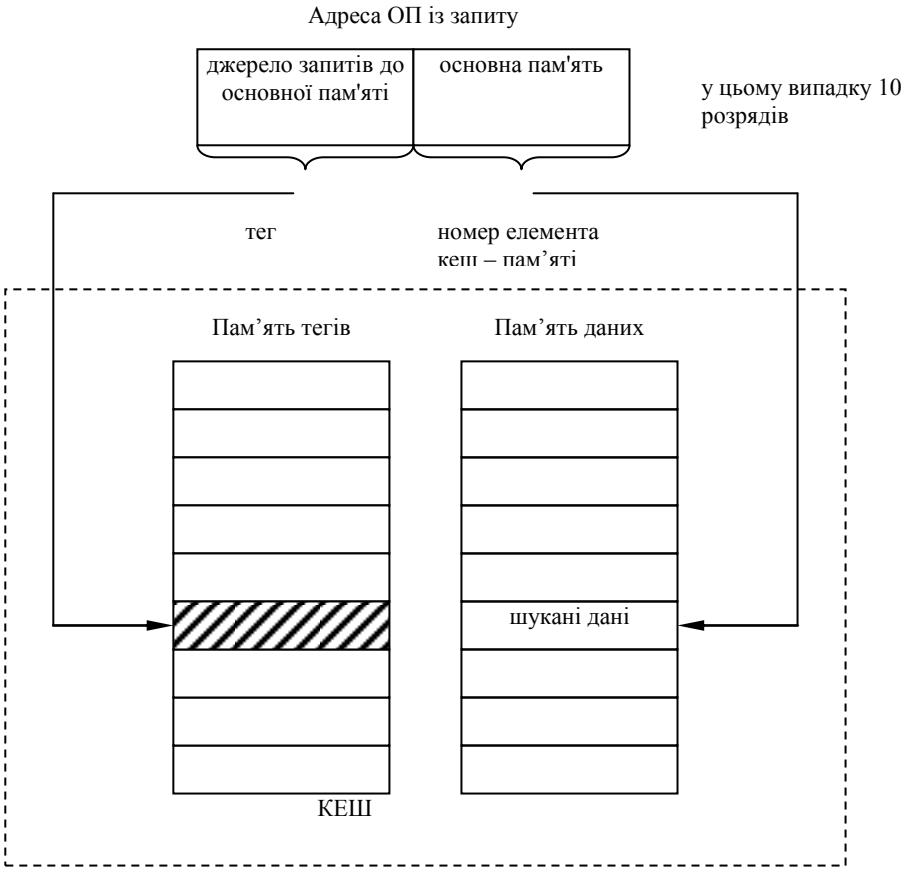

Рис. 3.14 – Детерміноване відображення на кеш пам'ять.

При пошуку даних у кеші використовується швидкий прямий доступ до запису за номером рядка, отриманим з адреси оперативної пам'яті із запиту. Крім того, виконується додаткова перевірка на збіг тегу з відповідною частиною адреси із запиту. При збігу тегу констатується кеш-влучення. У противному випадку, констатується кеш-промах, дані зчитуються з ОП і копіюються в кеш.

У багатьох сучасних процесорах кеш пам'ять будується на основі поєднання цих двох підходів. При *змішаному підході* довільна адреса оперативної пам'яті відображається не на одну адресу кеш пам'яті (як це характерно для *прямого відображення*), і не на будь-яку адресу кеш пам'яті (як при *випадковому відображенні*), а на деяку групу адрес. Всі групи пронумеровані. Пошук у кеші здійснюється спочатку за номером групи, отриманим з адреси ОП із запиту, а потім у межах групи шляхом асоціативного перегляду всіх записів групи на збіг старших частин адрес ОП (рис. 3.15).

При промаху дані копіюються за будь-якою вільною адресою з заданої групи. При відсутності вільних адрес у групі виконується витиснення даних.

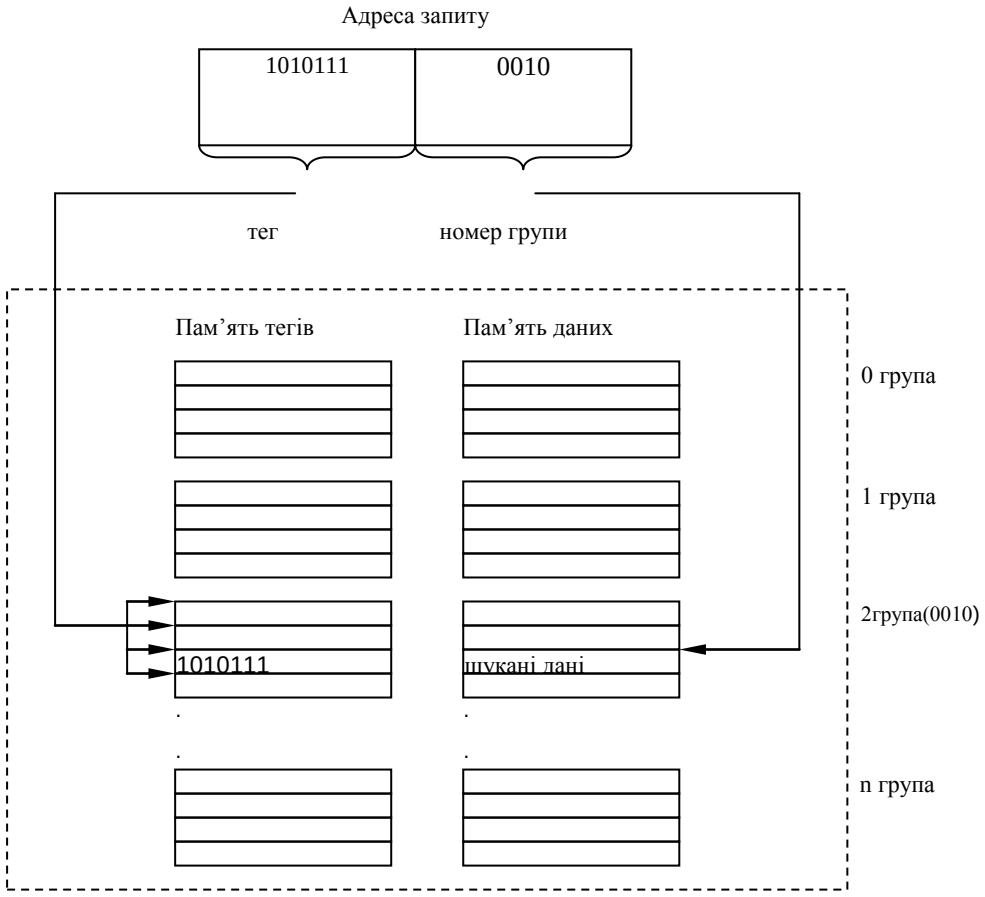

Рис. 3.15 – Комбінування прямого й випадкового відображення на КЕШ

При промаху дані копіюються за будь-якою вільною адресою з заданої групи. При відсутності вільних адрес у групі виконується витиснення даних. Оскільки кандидатів на вивантаження кілька (всі записи з даної групи), алгоритм заміщення може врахувати інтенсивність звертання до даних і тим самим підвищити ймовірність влучень у майбутньому.

Як ми бачили, кеш проглядається тільки з метою узгодження вмісту кеша й основної пам'яті. Якщо відбувається промах, то запити на запис не викликають ніяких змін у кеші. У деяких реалізаціях кеш пам'яті при відсутності даних у кеші вони копіюються туди з основної пам'яті незалежно від того, виконується запит на читання або запис.

Відповідно до описаної логіки роботи кеш пам'яті, при виникненні запиту спочатку проглядається кеш, а потім, якщо відбувся промах, виконується звертання до основної пам'яті. Однак, часто реалізується й інша схема роботи

кеша: *пошук у кеші й основній пам'яті починається одночасно***,** потім, за результатом перегляду кеша, операція в основній пам'яті або триває, або переривається.

У ряді обчислювальних систем використовується дворівневе кешування (рис. 3.16). Кеш першого рівня має менший обсяг і більш високу швидкодію, ніж кеш другого рівня. Кеш другого рівня відіграє роль основної пам'яті стосовно кешу першого рівня.

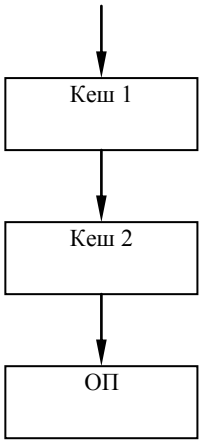

Рис. 3.16 – Дворівневе кешування

Спочатку робиться спроба виявити дані в кеші 1-го рівня. Якщо відбувся промах, пошук триває в кеші другого рівня. Якщо потрібні дані відсутні й там, тоді відбувається зчитування даних з основної пам'яті. При зчитуванні даних з ОП відбувається їхнє копіювання в кеш другого рівня, а якщо дані зчитуються з кеша другого рівня, то вони копіюються в кеш першого рівня.

Кеши різних рівнів можуть узгоджувати дані різними способами. Наприклад, у процесорі Pentium кеш першого рівня використовує зворотний запис.

Ми розглядали системи з одним кешем на кожному рівні. Однак існує цілий ряд систем (розподілених систем обробки інформації), у яких на кожному рівні є кілька кешей.

Завдання на виконання роботи

1. Написати програму, що моделює процес управління пам’яттю (розподілу пам’яті для процесів), перетворення віртуальної адреси у фізичну, пошук у пам’яті за запитами процесів, вивільнення пам’яті) при заданому варіантом способі організації пам’яті (перелік варіантів представлений нижче).

Вхідні дані – розмір пам’яті, що підлягає розподілу, розміри сторінок (розділів, сегментів тощо), розміри потрібної процесам пам’яті та ін. задаються самостійно та у відповідності до завдання.

2. Продемонструвати роботу моделі з виконанням основних операцій з пам’яттю: надання пам’яті потрібного розміру за запитом процесу, перетворення віртуальної адреси у “фізичну” при зверненні до комірки пам’яті, здійснення запису або читання, вивільнення пам’яті при завершенні процесу. Завдання операцій можна реалізувати за допомогою меню.

Якщо потрібно за алгоритмом, додатково продемонструвати процес вивантаження-завантаження сегментів або сторінок.

Вихідні дані – наглядна інформація про поточний розподіл пам’яті (карта пам’яті), що містить адреси ділянок пам’яті, стан (вільно чи зайнято та ким) після кожної операції з пам’яттю.

Окремо показати коректність перетворення віртуальної адреси звернення до комірки пам’яті за запитом у “фізичну” адресу і здійснення запису до комірки та читання з неї.

Примітка. При моделюванні алгоритмів без використання зовнішньої пам’яті вважати, що вихідні адреси кожного незавантаженого процесу починаються з 0000..00, а розміри кожного із процесів – довільно задані. Кожній адресі незавантаженого процесу за запитом варто поставити у відповідність реальну адресу пам’яті, якщо процес завантажується. Завантаження та вивантаження (при необхідності) процесів виконувати згідно з заданою чергою.

При моделюванні алгоритмів з використанням зовнішньої пам'яті слід задати віртуальний адресний простір кожного процесу. Звернення до пам'яті виконувати за віртуальною адресою, здійснювати при необхідності завантаження і вивантаження відповідних частин процесів, показуючи значення конкретної фізичної адреси, що відповідає заданій віртуальній.

При моделюванні алгоритмів роботи кеш пам'яті, адресний простір кеш пам'яті і основної пам'яті може задаватися довільно при співвідношенні їх обсягів не менш ніж 1:10. Пошук, запис і заміщення інформації в кеш пам'яті повинно виконуватися шляхом завдання шуканих адрес основної пам'яті.

Варіанти завдання

1. *Фіксовані розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *лінійний* адресний простір. Розміри процесів задаються випадково.

2. *Динамічні розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *лінійний* адресний простір. Розміри процесів задаються випадково.

3. *Переміщувальні розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам’яті, виконується процедура «стискування» в напрямку *старших адрес*. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *лінійний* адресний простір. Розміри процесів задаються випадково.

4. *Переміщувальні розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам’яті, виконується процедура «стискування» в напрямку *молодших адрес*. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *лінійний* адресний простір. Розміри процесів задаються випадково.

5. *Фіксовані розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *сегментований* адресний простір. Сегменти можуть розміщуватися в різних розділах. Розміри процесів задаються випадково.

6. *Динамічні розділи* (без використання зовнішньої пам’яті). Кількість розділів меньша, ніж кількість процесів. Процеси утворюють загальну чергу до розділів пам’яті. Використовується *сегментований* адресний простір. Сегменти

можуть розміщуватися в різних розділах. Розміри процесів задаються випадково.

7. *Переміщувальні розділи* (без використання зовнішньої пам’яті). Кількість розділів – меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам’яті, виконується процедура «стискування» в напрямку *старших адрес.* Процеси утворюють загальну чергу до розділів пам’яті. Використовується *сегментований* адресний простір. Сегменти можуть розміщуватися в різних розділах. Розміри процесів задаються випадково.

8. *Переміщувальні розділи* (без використання зовнішньої пам’яті). Кількість розділів – меньша, ніж кількість процесів. Якщо черговий розділ неможливо розмістити у пам’яті, виконується процедура «стискування» в напрямку *молодших адрес.* Процеси утворюють загальну чергу до розділів пам’яті. Використовується *сегментований* адресний простір. Сегменти можуть розміщуватися в різних розділах. Розміри процесів задаються випадково.

9. *Сторінкова організація пам’яті.* Кількість процесів довільна. Віртуальний адресний простір поділяється *на розділи*, а розділи – *на сторінки.* Загальна кількість віртуальних сторінок процесів повинна перебільшувати загальну кількість фізичних сторінок. Додатково продемонструвати процес вивантаження-завантаження сторінок

10. *Сегментна організація пам’яті.* Кількість процесів довільна. Розміри і кількість сегментів процесів у віртуальному просторі довільна. Віртуальний адресний простір поділяється на сегменти Загальна кількість віртуальних сегментів процесів повинна перебільшувати загальну кількість фізичних, які одночасно можуть розміщуватися у пам’яті. Додатково продемонструвати процес вивантаження-завантаження сегментів.

11. *Сегментно-сторінкова організація пам’яті.* Кількість процесів довільна. Кількість сегментів і сторінок в сегменті у віртуальному просторі процесів довільна. Віртуальний адресний простір поділяється на сегменти, а сегменти – на сторінки. Загальна кількість віртуальних сторінок процесів повинна перебільшувати загальну кількість фізичних сторінок, які одночасно

можуть розміщуватися у пам’яті. Додатково продемонструвати процес вивантаження-завантаження сторінок.

12. *Кеш пам’ять.* Розробити модель, яка реалізує алгоритм пошуку та заміщення даних в кеш пам’яті для детермінованого відображения даних на кеш. Узгодження даних основної та кеш пам’яті виконати за принципом наскрізного запису

13. *Кеш пам’ять.* Розробити модель, яка реалізує алгоритм пошуку та заміщення даних в кеш пам’яті для детермінованого відображения даних на кеш. Узгодження даних основної та кеш пам’яті виконати за принципом зворотного запису

14. *Кеш пам’ять.* Розробити модель, яка реалізує алгоритм пошуку та заміщення даних в кеш пам’яті для випадкового відображения даних на кеш. Узгодження даних основної та кеш пам’яті виконати за принципом зворотного запису

15. *Кеш пам’ять.* Розробити модель, яка реалізує алгоритм пошуку та заміщення даних в кеш пам’яті для випадкового відображения даних на кеш. Узгодження даних основної та кеш пам’яті виконати за принципом наскрізного запису.

Контрольні запитання

1. Що визначає поняття *фрагментація пам’яті*?

2 Дайте визначення термінам: *віртуальний адресний простір*, *віртуальна адреса, фізична адреса*?

3. Які переваги та недоліки мають сегментний та сторінковий методи організації пам’яті?

4. Які правила використовуються для вирішення проблеми вивантаження сторінок?

5. Які переваги та недоліки мають методи прямого та випадкового методів відображення пам’яті на КЕШ?

6. Які існують методи узгодження даних основної та кеш пам’яті?

ЛАБОРАТОРНА РОБОТА №4

Файлові системи

Метою лабораторної роботи є ознайомлення з основними підходами до фізичної організації файлових систем та їх реалізацією у файлових системах.

Основні теоретичні відомості

Фізична організація і адресація файлу

Фізична організація файлу – це спосіб розміщення файлу на диску. Основними критеріями ефективності фізичної організації файлів є:

- швидкість доступу до даних;
- обсяг адресної інформації файлу;
- ступінь фрагментованості дискового простору;
- максимально можливий розмір файлу.

Неперервне розміщення – найпростіший варіант фізичної організації. При цьому, файлу надається послідовність кластерів диску, що утворюють неперервну ділянку дискової пам'яті. *Основне достоїнство* – висока швидкість доступу, оскільки витрати на пошук і зчитування кластерів файлу є мінімальними. Також мінімальний обсяг адресної інформації – достатньо зберігати тільки номер першого кластера та обсяг файлу. Максимально можливий розмір файлу не обмежується. Цей варіант має істотний недолік:

- неможливо спрогнозувати, якого розміру повинна бути неперервна область, якщо файл при кожній модифікації може збільшувати розмір;
- значною проблемою є фрагментація (багато вільних областей малого розміру).

Розміщення файлу у вигляді зв'язаного списку кластерів

При цьому способі на початку кожного кластера міститься покажчик на наступний кластер. Адресна інформація мінімальна: розташування файлу задається одним числом – номером першого кластера. Фрагментація на рівні

кластерів відсутня. *Недоліком є* складність реалізації доступу до довільно заданого місця файлу (потрібно простежити весь ланцюжок кластерів від початку).

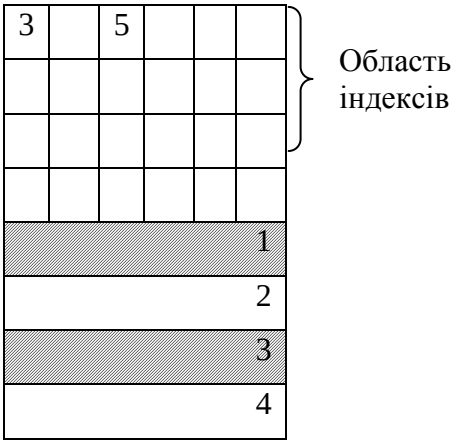

Рис 4.1 – Використання зв'язаного списку адрес

Якщо деякий кластер N призначений деякому файлу, тоді індекс цього кластера дорівнюватиме або номеру M наступного кластера даного файла або приймає спеціальне значення, що є ознакою того, що цей кластер є останнім для файла. Індекс же попереднього кластера файла приймає значення N, вказуючи на новий призначений кластер. Цей спосіб використовується у файловій системі (ФС) FAT.

Ще один спосіб полягає у простому перерахуванні номерів кластерів, які займає файл. Цей перелік і слугує адресою файла.

Недолік є очевидним: довжина адреси залежить від розміру файла і для великого файла може скласти значну величину. *Достоїнство* – висока швидкість доступу до довільного кластера, тому що тут використовується пряма адресація, а не перегляд ланцюжків. Фрагментація також відсутня.

Файлова система FAT

Абревіатура FAT (File allocation table) або таблиця розміщення файлів відноситься до лінійної таблиці з відомостями про файли: їх найменування, атрибути та інші дані, що визначають місцезнаходження файлів у середовищі

FAT. Елемент FAT визначає фактичну область диску, де зберігається початок фізичного файлу.

У файловій системі FAT логічний дисковий простір поділяється на дві області: системну та область даних. Системна область логічного диску створюється та ініціалізується при форматуванні. Область даних логічного диску вміщує файли та каталоги, що підпорядковуються кореневому. На відміну від системної області, область даних є доступною через користувацький інтерфейс DOS. Системна область складається з наступних компонентів, які розміщені у логічному адресному просторі послідовно:

- завантажувальний сектор;
- таблиці розміщення файлів;
- кореневий каталог.

Таблиця розміщення файлів

Ця таблиця являє собою карту або образ області даних, де прописаний стан кожної частини області даних. Область даних розбивається на кластери. Кластер – це один або декілька суміжних секторів у логічному дисковому адресному просторі. У таблиці FAT кластери, що належать одному файлу (тобто некореневому каталогу) зв’язані у ланцюги. Для визначення номера кластера у системі FAT-16 використовується 16 бітове слово, тобто можна мати 65536 кластерів. Файл (каталог) займає ціле число кластерів. Останній кластер може бути задіяний не повністю, що призводить до значної втрати дискового простору при великому розмірі кластера. На дискетах кластер займає один або два сектори, а на жорстких дисках – в залежності від їх ємності (табл.. 4.1).

Номер кластера завжди відноситься до області даних диску. Перший допустимий номер кластера завжди починається з 2. Номери кластерів відповідають області елементів таблиці розміщення файлів.

Логічна розбивка області даних на кластери як сукупності секторів, замість використання одиночних секторів, має такий сенс: по-перше, зменшується розмір самої таблиці FAT; зменшується можлива фрагментація

файлів; прискорюється доступ до файлу, оскільки у декілька разів зменшується довжина ланцюжків фрагментів дискового проекту.

Таблиця 4.1 – Таблиця розміщення файлів

Ємність розділу	Кількість секторів у кластері	Розмір кластерів (Кбайт)
16-127	4	2
128-255	8	4
256-511	16	8
512-1023	32	16
1024-2047	64	32

Однак великий розмір кластерів веде до неефективного використання області даних, особливо у випадку великої кількості маленьких файлів. Наприклад, при кластері з 32 секторами, тобто розмірі кластера 16 Кбайт, середній розмір втрат становить 8 Кбайт. Якщо кількість файлів становить декілька тисяч, витрати можуть становити більш ніж 100 Мбайт. Тому у більшості сучасних ОС розміри кластерів зменшуються до 512 байт-4 Кбайт.

Ідея розміщення файлової системи з використанням таблиць може бути проілюстрована рис. 4.2.

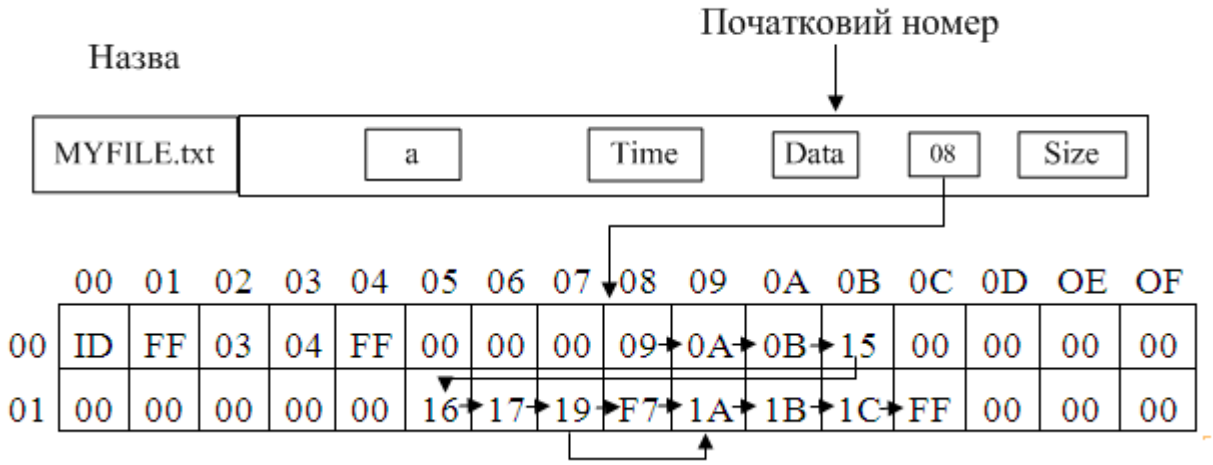

Рис 4.2 – Розміщення файлової системи з використанням таблиць

Файл починається з восьмого кластеру. Всього файл займає 11 кластерів, 18-й кластер позначено спеціальним кодом F7 як дефектний, а 1C – кодом FF, що визначає кінець файлу. Вільні кластери позначено кодом 00.

Оскільки файли на диску змінюються, дані одного файлу можуть розміщуватись не у сусідніх кластерах, а утворювати досить складні ланцюги. Це призводить до суттєвого зменшення швидкодії при роботі з файлами.

У зв’язку з важливістю цілісності FAT, вона зберігається у двох ідентичних примірниках. Обновлюються копії одночасно. Для роботи використовується тільки перший екземпляр. Якщо з якихось причин він зруйнується, тоді буде використано другий екземпляр. Так у системі ОС Windows існує утиліта перевірки та відновлення файлової структури ScanDisc, яка при виникненні невідповідності між першою та резервною копіями FAT пропонує відновити головну таблицю, використовуючи копію.

Кореневий каталог відрізняється від звичайного тим, що він має фіксовану кількість елементів. Для кожного файлу та каталогу інформація зберігається у відповідності з наступною структурою.

Таблиця 4.2 – Структура зберігання інформації

Розмір поля даних (байт)	Зміст поля даних
1	Ім’я файлу
1	Атрибути файлу
1	Резервне поле
3	Час створення
2	Дата створення
2	Дата останнього доступу
2	Зарезервовано
2	Час останньої модифікації
2	Дата останньої модифікації
2	Номер початкового кластера FAT
4	Розмір файла

Структура системи файлів – ієрархічна. У ній елементом каталогу може бути файл, який, у свою чергу, сам може бути каталогом (рис 4.3).

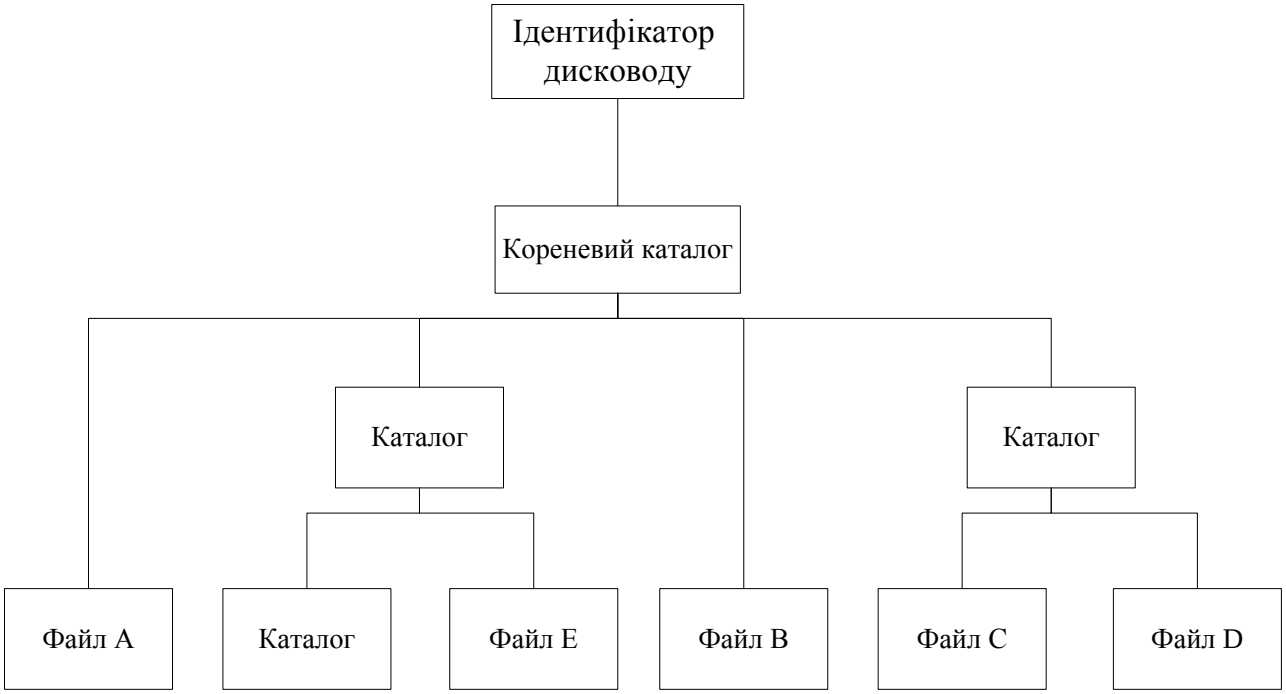

Рис 4.3 – Ієрархічна структура системи файлів

Файлові системи VFAT та FAT-32

Ми бачимо, що дуже важливою характеристикою FAT було використання імен файлів формату “8.3”, де 8 символів відводять для назви файлу, а 3 – для його розширення. Пізніше до стандартної FAT-16 додали ще 2 різновиди: VFAT (віртуальний FAT) та FAT-32.

VFAT була розроблена для використання файлового введення-виведення у захищеному режимі. Також VFAT підтримує довгі найменування файлів. При цьому, вона зберігає сумісність із початковим варіантом FAT-16, тобто підтримує формат “8.3”.

Основні недоліки FAT та VFAT полягають у тому, що мають місце великі втрати на кластерізацію при великих розмірах логічного диску та суттєві обмеження на розмір самого логічного диску. Це стало причиною розробки нової файлової системи з використанням старої ідеї використання таблиць – FAT-32. FAT-32 – 32 розрядна файлова система, яка доповнена деякими удосконаленнями у порівнянні з FAT-16 та VFAT.

Принциповим є те, що FAT-32 більш ефектно використовує дисковий простір. Вона використовує кластери локального розміру (оскільки у старій

версії були обмеження на 65535 кластерів і при збільшенні дискового простору треба було збільшувати розміри кластера). Наприклад, для дисків у 8 Гбайт FAT-32 може використовувати кластери у 4 Кбайт. Як результат, економія в порівнянні з FAT-16 досягла 10-15%.

FAT-32 може переміщувати кореневий каталог та використовувати резервну копію замість стандартної. Кореневий каталог FAT-32 задається у вигляді звичайного ланцюжка кластерів. Тобто кореневий каталог може знаходитись у довільному місці диску, що знімає старе обмеження на розмір кореневого каталогу у 512 елементів.

Окрім збільшення ємності FAT до 4 Тбайт (Терабайт=1024 Гбайт), FAT-32 має необхідні вдосконалення у структурі кореневого каталогу. FAT-16 вимагала, щоб вся інформація кореневого каталогу була розміщена у одному дисковому кластері. При цьому, вся його інформація мала не перевищувати 512 файлів. Необхідність представляти довгі імена та забезпечення сумісності з попередніми версіями FAT привели розробників до використання для представлення довгого імені елементів каталогу (рис. 4.4 – 4.6).

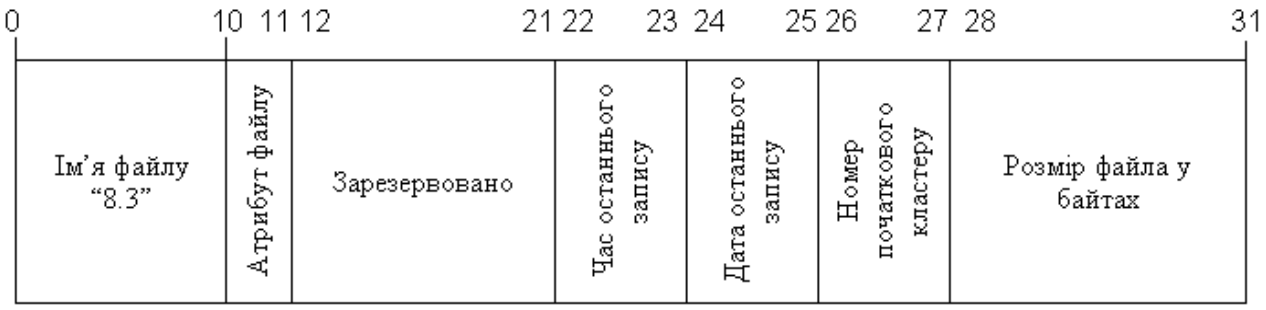

Рис 4. 4 – Елемент каталогу для короткого імені FAT-16

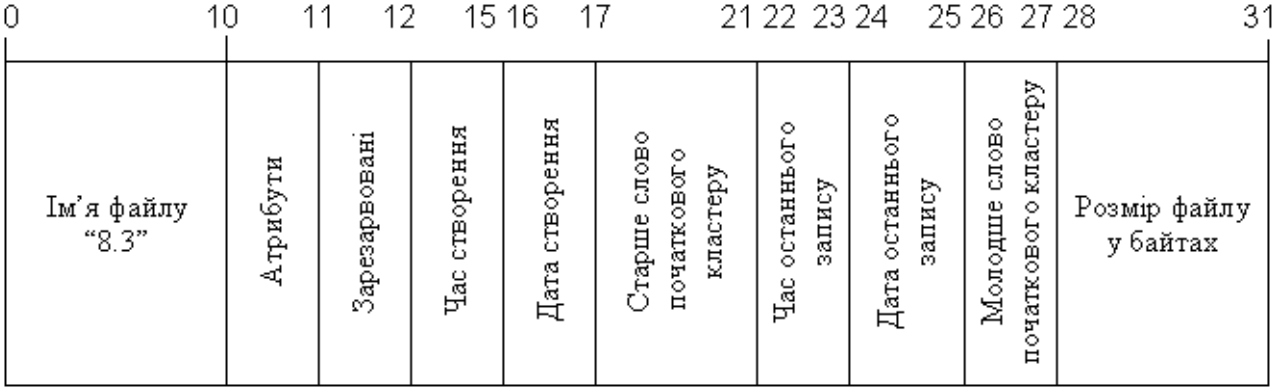

Рис 4.5 – Елемент каталогу для короткого імені FAT 32

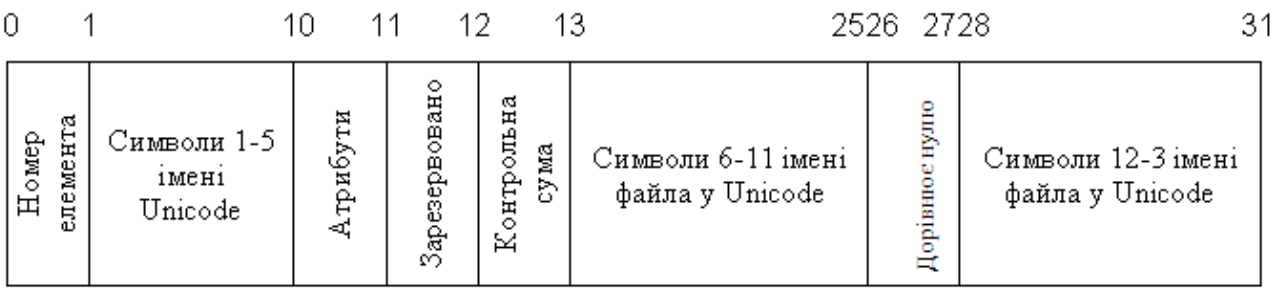

Рис 4.6 – Елемент каталогу для довгого імені FAT 32

FAT-32 успішно справляється з проблемою довгих імен у кореневому каталозі, але проблеми з обмеженням довжини повної файлової специфікації залишається. Тому рекомендовано обмежити довгі імена 75-80 символами, щоб залишити достатньо місця для задання шляху (180-185 символів).

Файлова система NTFS

Файли зберігаються у каталогах (папках). На відміну від FAT, робота NTFS з дисками великого розміру виконується більш ефективно. У її розпорядженні також існують засоби для обмеження доступу до файлів та каталогів, введені механізми підвищення надійності файлової системи, знято багато обмежень на максимальну кількість дискових секторів або кластерів.

Файли та каталоги базуються на фундаментальному об’єкті, що дістав назву *F-Node* (рис.4-6). Кожний файл і каталог має свій файловий вузол, який має у своєму складі 15 символів імені (ідентифікатора) файла та додаткову інформацію про файл. NTFS добре працює при роботі з томами до 300-400Мбайт. Максимальний розмір тому та файлу становить 16 Ебайт (Екзабайт $\approx 2^{64} \approx$ 16000 млрд. Гігабайт). Кількість файлів у кореневому та кореневих каталогах не обмежена.

Основою структури каталогів NTFS є структура даних, яка дістала назву “сбалансоване бінарне дерево” (B-Tree).

Бінарне дерево називають сбалансованим тоді, коли висоти двох під-дерев (лівого та правого) відрізняються за рівнем не більше ніж на одиницю. З такими деревами можна виконувати такі операції з подальшим балансуванням:

1) Знайти вершину за даним ключем (ідентифікатором).

2) Додати нову вершину з конкретним ідентифікатором.

3) Виключити вершину з заданим ідентифікатором.

Алгоритми виконання цих операцій наведені в [6]. Вони ідентичні алгоритмам пошуку, додавання та виключення файлів із системи. У файловій системі FAT каталог має лінійну структуру, яка спеціально не упорядкована, тому при пошуку файлу ми послідовно проглядали її із самого початку.

У випадку B-Tree структура каталогу являє собою збалансоване дерево з записами, упорядкованими за алфавітом. Кожен запис, що входить до цього дерева, містить атрибути файлу, вказівник на відповідний файловий вузол, інформацію про час та дату створення, час і дату останнього оновлення та звертання, дані про розширені атрибути, лічильник звертань до файлу, довжину імені файлу, саме ім’я та іншу інформацію.

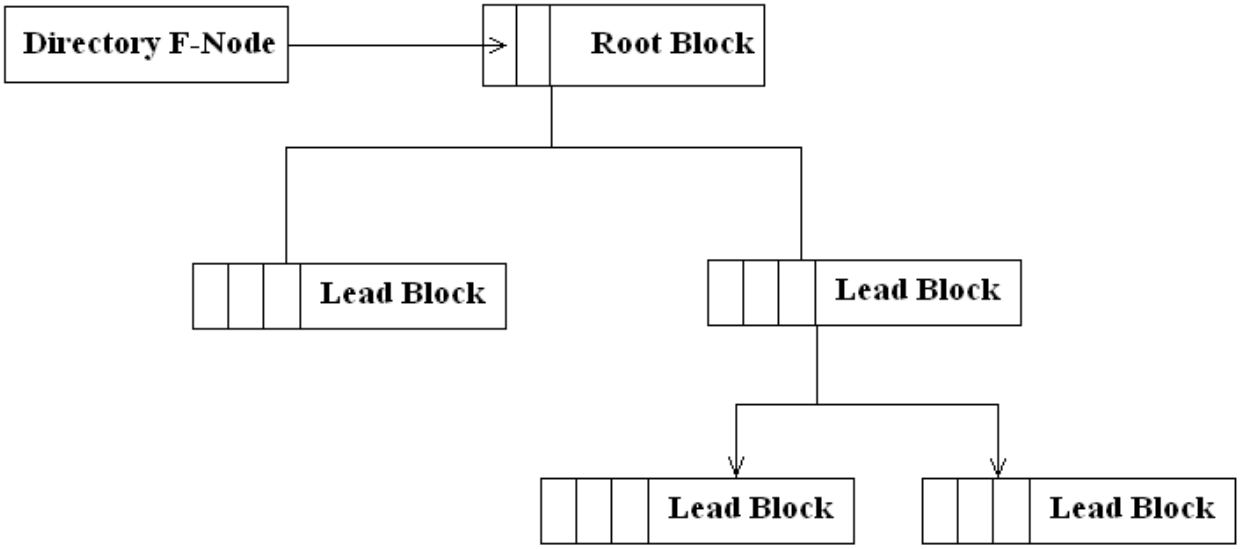

Рис 4.7 – Збалансоване двійкове дерево

При пошуку файлу у каталозі файлова система проглядає тільки необхідні гілки двійкового дерева. Такий метод ефективніший ніж послідовне читання усіх записів у каталозі, що мало місце у FAT.

Для того, щоб знайти необхідний файл у каталозі (точніше, вказати на його інформаційну структуру F-Node), більшість записів взагалі читати не

обов’язково. У результаті пошуку інформації про файл буде необхідно виконати значно меншу кількість операцій читання з диску.

Дійсно, якщо каталог містить 4096 файлів, при пошуку FAT потребує читання у середньому 64 секторів. NTFS буде читати лише 2-4 сектори та знайде файл.

Дійсно, при використанні 40 входів на блок (сектор диску) каталоги дерева з двома рівнями можуть мати 1640 входів, а каталоги дерева з трьома рівнями – 65640 входів. Тобто деякий файл може бути знайдено у типовому каталозі із 65640 файлів максимум за 3 звертання. При роботі з FAT потрібно буде у найгіршому випадку більш ніж 4000 секторів.

Одним з основних понять, що використані при роботі з NTFS, є поняття тому. NTFS поділяє дисковий простір тому на кластери – блоки даних, що адресуються як одиниці даних. Вона підтримує розміри кластерів від 512 байт до 64 Кбайт. Стандартним вважається кластер розміром 2 або 4 Кбайт. Увесь дисковий простір поділено на 2 неоднакові частини (рис 4.8).

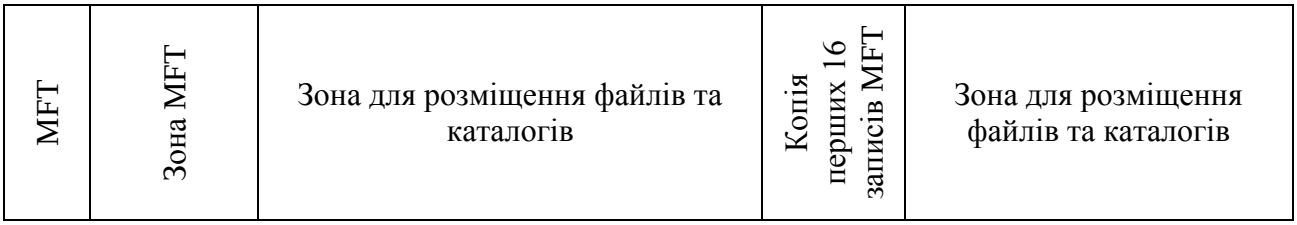

Рис 4.8 – Розподіл дискового простору NTFS

Перші 12 % диску відводяться під так звану MFT-зону – простір, який може займати, збільшуючись у розмірі, головний службовий метафайл. MFT (master file table) – спеціальний файл, головна системна структура даних, яка дозволяє знаходити місцезнаходження всіх інших файлів.

MFT являє собою централізований каталог усіх інших файлів диску, у тому числі і самого себе. MFT поділено на записи фіксованого розміру 1 Кбайт, де кожен запис відповідає конкретному файлу. Перші 16 файлів носять службовий характер і недоступні ОС; вони звуться метафайлами, причому самий перший з них сам MFT.

Ці перші 16 елементів в MFT є єдиною частиною диску з чітко фіксованим положенням. Копія цих самих 16 записів для надійності зберігається у середині тому, оскільки вони дуже важливі. Інші частини MFT файла можуть бути розміщені, як і інші файли, у довільних місцях диску – поновити його положення можна за допомогою його ж самого, якщо "закріпитись" за дану основу – за перший елемент MFT.

Завдання на виконання роботи

1. Написати програму, що моделює роботу складових заданої файлової системи згідно варіанта (перелік варіантів представлений нижче).

Вхідні дані студент задає самостійно з урахуванням особливостей індивідуального варіанта завдання.

2. Зробити візуалізацію роботи програми і кінцевих результатів на різних наборах вхідних даних.

Варіанти завдань

1. *Побудовати таблиці ідентифікаторів за методом бінарного дерева*

Дерево повинно бути ідеально сбалансованим і упорядкованим. Забезпечити можливість додаткового включення та виключення ідентифікаторів при збереженні сбалансованості.

Забезпечити можливість пошуку заданого ідентифікатора з роздруківкою шляху пошуку та відображенням вигляду дерева.

Додаткові умови – у якості ідентифікаторів використовувати не більше двох символів: букв та/або цифр

2. *Побудовати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати три букви.

3. *Побудовати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати три цифри.

4. *Побудувати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати дві букви та одну цифру.

5. *Побудувати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати одну букву та одну цифру.

6. *Побудувати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати дві букви та дві цифри.

7. *Побудувати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати одну цифру.

8. *Побудувати таблиці ідентифікаторів за методом бінарного дерева – див. варіант 1.*

Додаткові умови – у якості ідентифікаторів використовувати одну букву та дві цифри.

9. *Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16.*

Імітувати процеси створення файлів заданого у кластерах розміру, збільшення файлів на задану кількість кластерів та знищення файлів. Передбачити пошук файлів за їх ідентифікатором та роздруківку номерів кластерів, які займає файл.

Додаткові умови – Задати максимально можливу кількість кластерів.

10.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16.*

Імітувати процеси створення файлів заданого у кластерах розміру, збільшення та зменшення файлів на задану кількість кластерів. Передбачити

пошук файлів за їх ідентифікатором та роздруківку номерів кластерів, які займає файл.

Додаткові умови – Задати максимально можливу кількість кластерів, визначивши при цьому певну кількість дефектних кластерів.

11.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16.*

Імітувати процеси створення файлів заданого у кластерах розміру, зменшення файлів на задану кількість кластерів та знищення файлів. Передбачити пошук файлів за їх ідентифікатором та роздруківку номерів кластерів, які займає файл.

Додаткові умови – Задати максимально можливу кількість кластерів.

12.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16.*

Імітувати процеси створення файлів заданого у кластерах розміру, збільшення та зменшення файлів на задану кількість кластерів, а також знищення файлів. Передбачити пошук файлів за їх ідентифікатором та роздруківку номерів кластерів, які займає файл.

Додаткові умови – Задати максимально можливу кількість кластерів, визначивши при цьому певну кількість дефектних кластерів.

13.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16 .*

Імітувати процеси створення файлів заданого у кластерах розміру, збільшення файлів на задану кількість кластерів, а також знищення файлів. Передбачити пошук файлів за їх ідентифікатором та роздруківку номерів кластерів, які займає файл.

Додаткові умови – Задати максимально можливу кількість кластерів, визначивши при цьому певну кількість дефектних кластерів.

14.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16 – див. варіант 9.*

Додаткові умови – Задати максимально можливу кількість кластерів, визначивши при цьому певну кількість дефектних кластерів з початку таблиці.

15.*Побудувати таблицю розміщення файлів відповідно до моделі файлової системи FAT-16 – див. варіант 11.*

Додаткові умови – Задати максимально можливу кількість кластерів, визначивши при цьому певну кількість дефектних кластерів почергово із звичайними.

Контрольні запитання

1. За якими принципами побудована файлова система FAT-16?
2. Що означає поняття *кластер*?
3. Які переваги та недоліки розміщення файлу у вигляді зв’язаного списку адрес?
4. Яка інформація знаходиться у таблицях FAT?
5. У чому переваги файлової системи FAT-32 у порівнянні з файловою системою FAT-16?
6. У чому полягають переваги файлової системи NTFS у порівнянні з FAT?
7. За рахунок чого продуктивність файлової системи NTFS вища ніж FAT?
8. За якими принципами побудована файлова система NTFS?

РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Основна література

1. Шеховцов В. А. Операційні системи / В. А. Шеховцов – К.: Вид. гр. BHV, 2005. – 576 с.

2. Гордеев А. В. Системное программное обеспечение / А. В. Гордеев, А. Ю. Молчанов – СПб: Питер, 2001. – 736 с.

3. Олифер В.Г. Сетевые операционные системы / В.Г. Олифер, Н.А. Олифер. – СПб: Питер, 2001. – 544 с.

Допоміжна література

4. Столлингс В. Операционные системы / В. Столлингс – М.: Издательский дом “Вильямс”, 2002. – 848 с.

5. Таненбаум Э. Современные операционные системы / Э. Таненбаум – СПб.: Питер. 2002. – 1040 с.

6. Вирт. Н. Алгоритмы и структуры данных. / Н. Вирт – М.: Мир, 1989. – 360 с.

7. Иртегов Д.В. Введение в операционные системы / Д. В. Иртегов – СПб.:БХВ- Петербург, 2002. – 624 с.