11.2 ОСНОВНІ АЛГОРИТМИ ЗАМІЩЕННЯ СТОРІНОК

Розмір віртуальної пам'яті для кожного процесу може істотно перевершувати розмір основної пам'яті. Це означає, що при виділенні сторінки основній пам'яті з великою ймовірністю не вдасться знайти вільний сторінковий кадр. У цьому випадку операційна система відповідно до закладених в неї критеріїв повинна:

- знайти деяку зайняту сторінку основної пам'яті;
- перемістити в разі потреби її вміст в зовнішню пам'ять;
- переписати в цей сторінковий кадр вміст потрібної віртуальної сторінки із зовнішньої пам'яті;
- належним чином модифікувати необхідний елемент відповідної таблиці сторінок;
- продовжити виконання процесу, якому ця віртуальна сторінка знадобилася.

Відмітимо, що при заміщенні доводиться двічі передавати сторінку між основною і вторинною пам'яттю. Процес заміщення може бути оптимізований за рахунок використання біта модифікації (один з атрибутів сторінки в таблиці сторінок). Біт модифікації встановлюється комп'ютером, якщо хоч би один байт був записаний на сторінку. При виборі кандидата на заміщення перевіряється біт модифікації. Якщо біт не встановлений, немає необхідності переписувати цю сторінку на диск, її копія на диску вже є. Подібний метод також застосовується до read-only-сторінок, вони ніколи не модифікуються. Ця схема зменшує час обробки page fault.

Існує велика кількість різноманітних алгоритмів заміщення сторінок. Усі вони діляться на **локальні** і **глобальні**. Локальні алгоритми заміщення сторінок, на відміну від глобальних, виділяють для заміщення з фізичної пам'яті одну з його сторінок, а не сторінки інших процесів. Глобальний же алгоритм заміщення в разі виникнення виняткової ситуації задовольниться звільненням будь-якої фізичної сторінки, незалежно від того, якому процесу вона належала.

Глобальні алгоритми мають ряд недоліків. По-перше, вони роблять одні процеси чутливими до поведінки інших процесів. Наприклад, якщо один процес у системі одночасно використовує велику кількість сторінок пам'яті, то усі інші

процеси в результаті відчуватимуть сильне уповільнення через нестачу кадрів пам'яті для своєї роботи. По-друге, некоректно працюючий процес може підірвати роботу всієї системи, намагаючись захопити більше пам'яті. Тому в багатозадачній системі використовують складніші локальні алгоритми.

Застосування локальних алгоритмів вимагає зберігання в операційній системі списку фізичних кадрів, виділених кожному процесу. Цей список сторінок іноді називають резидентною множиною процесу.

Ефективність алгоритму оцінюється на конкретній послідовності посилань до пам'яті, для якої підраховується число page faults. Ця послідовність називається **рядком звернень** (**reference string**). Ми можемо генерувати рядок звернень штучним чином за допомогою датчика випадкових чисел або трасуючи конкретну систему.

Більшість процесорів мають прості апаратні засоби, що дозволяють збирати деяку статистику звернень до пам'яті. Ці засоби включають два спеціальні прапори на кожен елемент таблиці сторінок. Прапор посилання (reference біт) автоматично встановлюється, коли відбувається будь-яке звернення до цієї сторінки, а прапор зміни (modify біт) встановлюється, якщо робиться запис в цю сторінку. Операційна система періодично перевіряє установку таких прапорів, для того щоб виділити активно використовувані сторінки, після чого значення цих прапорів скидаються.

Незалежно від стратегії управління резидентною множиною є ряд основних алгоритмів для вибору сторінки на заміщення:

- оптимальний алгоритм (OPT);
- алгоритм «першим увійшов – першим вийшов» (FIFO);
- алгоритм виштовхування сторінки, яка найдовше не використовувалася;
- годинниковий і модифікований годинниковий алгоритми.

11.2.1 Оптимальний алгоритм

Оптимальна алгоритм полягає у виборі для заміщення тієї сторінки, звернення до якої буде через найбільший проміжок часу в порівнянні з усіма іншими сторінками. Як буде показано далі, цей алгоритм призводить до мінімальної кількості переривань через відсутність сторінки. Зрозуміло, що реалізувати такий алгоритм неможливо, оскільки для цього системі потрібно знати всі майбутні події. ОС не знає,

до якої сторінки буде наступне звернення. Однак цей алгоритм є стандартом якості, з яким порівнюються реальні алгоритми.

На рис. 11.1 наведено приклад оптимальної стратегії. Передбачається, що для даного процесу використовується фіксований розподіл кадрів (фіксований розмір резидентної множини, що складається з трьох кадрів). Виконання процесу призводить до звернення п'яти різних сторінок. Оптимальна стратегія призводить після заповнення всієї множини кадрів до трьох переривань (звернення до сторінок), які позначені на малюнку буквами F.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Оптимальний алгоритм	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
3 page faults					**F**		**F**			**F**		

Рисунок 11.1 – Приклад роботи оптимального алгоритму

11.2.2 Першим увійшов – першим вийшов

Стратегія «першим увійшов – першим вийшов» (FIFO, виштовхування першої сторінки, що надійшла) розглядає кадри сторінок процесу як циклічний буфер, з циклічним видаленням сторінок. Все, що потрібно для реалізації цього алгоритму, – це покажчик, який циклічно проходить по кадрах сторінок процесу.

Це одна з найпростіших в реалізації стратегій заміщення. Логіка її роботи полягає в тому, що заміщається сторінка, яка перебуває в основний пам'яті довше інших. Однак далеко не завжди ця сторінка використовується не часто; дуже часто деяка область даних або коду інтенсивно використовується програмою, і сторінки з цієї області при використанні описаної стратегії будуть завантажуватися і розвантажуватися. Робота алгоритму проілюстрована на рис. 11.2.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Алгоритм FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
6 page faults					**F**	**F**	**F**		**F**		**F**	**F**

Рисунок 11.2 – Приклад роботи алгоритму FIFO

Стратегія «першим увійшов – першим вийшов» реалізується дуже просто, але відносно не часто призводить до кращих результатів. Багато з алгоритмів FIFO є варіантами схеми, відомої як **годинникова стратегія** (**clock policy**).

На перший погляд здається очевидним, що чим більше в пам'яті сторінкових кадрів, тим рідше будуть мати місце page faults. Але це не завжди так. Як встановив **Біледі** (Belady), певні послідовності звернень до сторінок в дійсності призводять до збільшення числа сторінкових порушень при збільшенні кадрів, виділених процесу. Це явище носить назву «**Аномалії Біледі**» або «**аномалії FIFO**» (рис. 11.3) [12].

Черга сторінок	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**
Найстаріша сторінка	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Найновіша сторінка			0	1	2	3	0	0	0	1	4	4
(a) 9 page faults	**F**	**F**	**F**	**F**	**F**	**F**	**F**			**F**	**F**	
Черга сторінок	**0**	**1**	**2**	**3**	**0**	**1**	**4**	**0**	**1**	**2**	**3**	**4**
Найстаріша сторінка	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Найновіша сторінка				0	0	0	1	2	3	4	0	1
(b) 10 page faults	**F**	**F**	**F**	**F**			**F**	**F**	**F**	**F**	**F**	**F**

Рисунок 11.3 – Аномалія Біледі: (a) – FIFO з трьома сторінковими кадрами; (b) – FIFO з чотирма сторінковими кадрами

11.2.3 Годинниковий алгоритм

Годинниковий алгоритм – це модифікація попереднього алгоритму (FIFO), який є занадто неефективним, оскільки постійно пересуває сторінки за списком. Тому краще зберігати всі сторінкові блоки в кільцевому списку у формі годинника, при цьому стрілка годинника вказує на найстарішу сторінку. У найпростішій схемі годинникової стратегії з кожним кадром зв'язується один додатковий біт, відомий як **біт використання** (u-use, або **біт звернення** – r-referenced).

Коли сторінка вперше завантажується в кадр, біт використання встановлюється рівним 1. При наступних зверненнях до сторінки, які викликали переривання через її відсутність, цей біт встановлюється рівним 0. При роботі алгоритму заміщення певна кількість кадрів, які є кандидатами на заміщення (поточний процес, локальна область

видимості, вся основна пам'ять або глобальний контекст), розглядається як циклічний буфер, з яким пов'язаний покажчик. При заміщенні сторінки покажчик переміщається до наступного кадру в буфері (рис. 11.4).

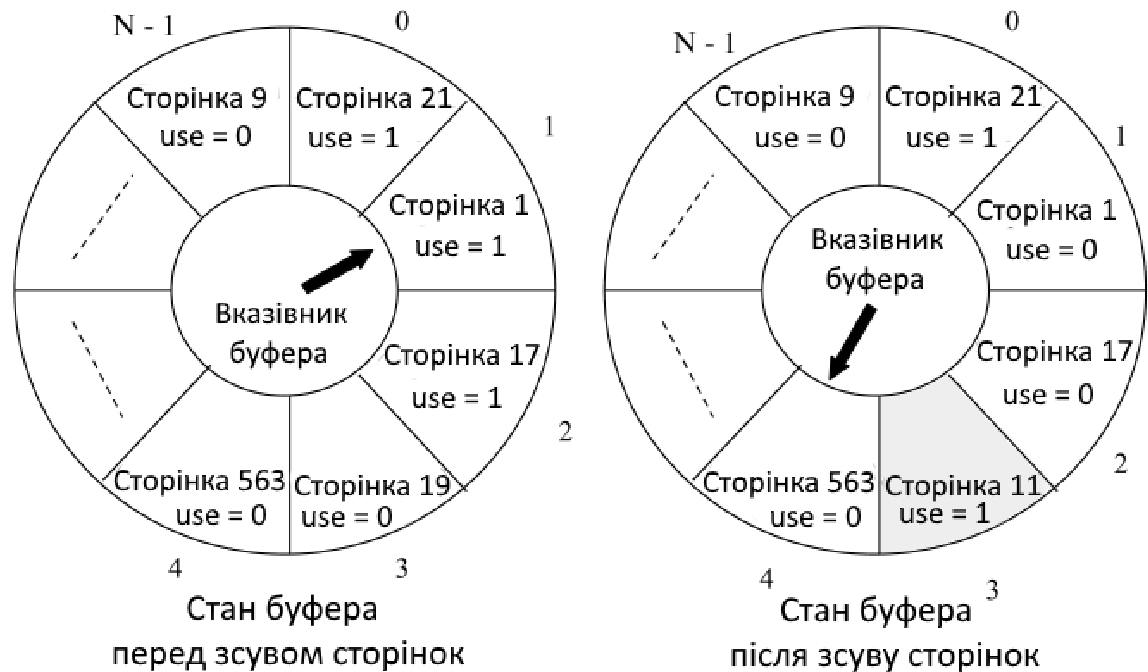

Рисунок 11.4 – Приклад роботи годинникового алгоритму

Коли настає час заміщення сторінки, ОС сканує буфер для пошуку кадру, біт використання якого дорівнює 0. Всякий раз, коли в процесі пошуку зустрічається кадр з бітом використання, рівним 1, він скидається в 0. Перший же зустрінутий кадр з нульовим бітом використання вибирається для заміщення. Якщо усі кадри мають біт використання, рівний 1, покажчик здійснює повний круг і повертається до початкового положення, замінюючи сторінку в цьому кадрі. Як бачимо, ця стратегія схожа із стратегією «першим увійшов – першим вийшов», але відрізняється тим, що кадри, які мають встановлений біт використання, пропускаються алгоритмом. Буфер кадрів сторінок представлений у вигляді круга, звідки і пішла назва стратегії (інша назва – **стратегія кругової заміни сторінок**). Ряд операційних систем використовує різні варіанти годинникової стратегії (наприклад, Multics).

На рис. 11.4 наведений простий приклад використання годинникової стратегії. Для заміщення доступні N-1 кадр основної пам'яті, представлені у вигляді циклічного буфера. Безпосередньо перед тим, як замістити сторінку у буфері завантажуваною з вторинної пам'яті сторінкою 11, покажчик буфера вказує на кадр 1, що містить сторінку 1. Тепер приступимо до виконання годинникового алгоритму.

Оскільки біт використання сторінки 17 в кадрі 2 рівний 1, ця сторінка не заміщається. Замість цього її біт використання скидається, а покажчик переміщається до наступного кадру 3. Тут знаходиться сторінка 19, біт використання якої дорівнює 0. Ця сторінка вибирається для заміщення. На її місце завантажується сторінка 11, біт використання якої переводиться в 1. Покажчик перекладається на наступний кадр 4. На цьому виконання алгоритму завершується. На рис. 11.4 біт використання позначений як use.

На рис. 11.5 наведено приклад роботи годинникового алгоритму, для якого використовується фіксований розподіл кадрів (три) і та ж черга сторінок, як і для попередніх алгоритмів (* – use = 1).

Звернення до сторінок	2	3	2	1	5	2	4	5	3	2	5	2
Годинниковий алгоритм	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
5 page faults					**F**	**F**	**F**		**F**		**F**	

Рисунок 11.5 – Приклад роботи годинникового алгоритму для трьох кадрів

Підвищити ефективність годинникового алгоритму можна шляхом збільшення кількості використовуваних при його роботі бітів. Такий алгоритм називають **модифікованим годинниковим алгоритмом** або **алгоритмом не використовуваної останнім часом сторінки.**

У всіх процесорах, які підтримують сторінкову організацію, з кожною сторінкою в основний пам'яті (а отже, з кожним кадром) пов'язаний **біт модифікації (m).** Цей біт використовується для вказівки того, що дана сторінка не може бути заміщена до тих пір, поки її вміст не буде записано назад у вторинну пам'ять. Цей біт може використовуватися годинниковим алгоритмом наступним чином. Беручи до уваги біти використання і модифікації, всі кадри можна розділити на чотири категорії:

- використаний давно, не модифікований (u = 0, m = 0);
- використаний недавно, не модифікований (u = 1, m = 0);
- використаний давно, модифікований (u = 0, m = 1);
- використаний недавно, модифікований (u = 1, m = 1).

Використовуючи цю класифікацію, змінимо часовий алгоритм, який тепер буде описаний таким чином (рис. 11.6).

1. Скануємо буфер кадрів, починаючи з поточного положення. У процесі сканування біт використання не змінюється. Перша ж сторінка зі станом (u = 0, m = 0) заміщується.

2. Якщо виконання першого кроку алгоритму не увінчалося успіхом, шукаємо сторінку з параметрами (u = 0, m = 1). Якщо така сторінка знайдена, вона заміщується. У процесі виконання даного кроку у всіх переглянутих сторінок скидається біт використання.

3. Якщо виконання попереднього кроку не дало результату, покажчик повертається в початкове положення, але у всіх сторінок значення біта використання скинуто в 0. Повторимо крок 1 і, при необхідності, крок 2. Очевидно, на цей раз потрібна сторінка буде знайдена.

Такий алгоритм використаний в схемі віртуальної пам'яті ОС Macintosh. Позитивний момент цього алгоритму полягає, на відміну від простого годинникового алгоритму, в перевазі заміни сторінок, що не змінювалися, в порівнянні з заміною модифікованих сторінок.

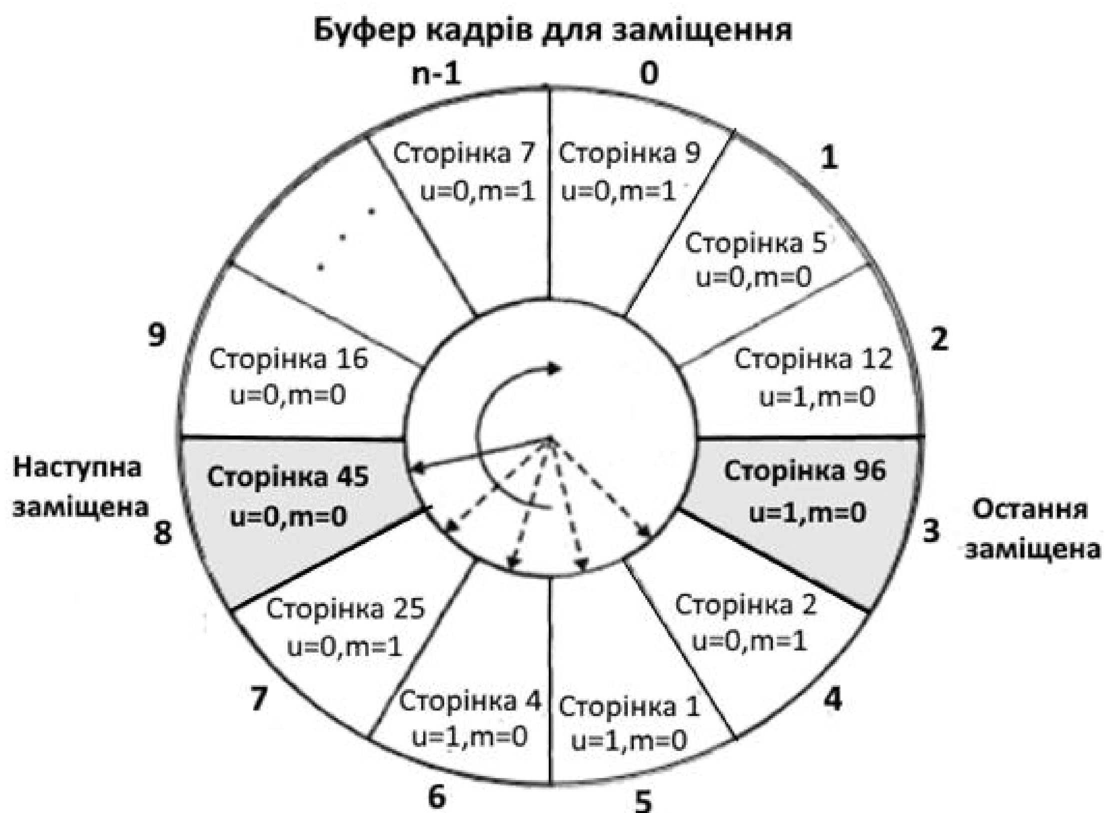

Рисунок 11.6 – Модифікований годинниковий алгоритм заміщення сторінок

Отже, годинниковий алгоритм циклічно проходить по всіх сторінках буфера в пошуках сторінки, яка не була модифікована з часу завантаження і давно не використовувалася. Така сторінка – хороший кандидат на заміщення, особливо з урахуванням того, що її не треба записувати на диск. Якщо при першому проході кандидатів на заміщення не знайшлося, алгоритм знову перевіряє буфер, тепер уже в пошуках модифікованої, давно не використовуваної сторінки. Хоча така сторінка і повинна бути записана перед заміщенням, відповідно до принципу локалізації вона навряд чи стане в нагоді в найближчому майбутньому. Якщо і цей прохід виявиться невдалим, всі сторінки позначаються як давно не використані, і виконується третій прохід.

За продуктивністю годинниковий алгоритм найближчий до алгоритму **найдовше невикористовуваної сторінки (LRU)**.

11.2.4 Заміщення сторінки, яка найдовше не використовувалася

Одним з наближень до алгоритму OPT є алгоритм, що виходить з евристичного правила, що недавнє минуле – хороший орієнтир для прогнозування найближчого майбутнього. Наприклад, має сенс заміщати сторінку, яка не використовувалася впродовж найдовшого часу. Такий підхід називається **Least Recently Used алгоритм** (LRU) – сторінка, що не використалася найдовше. Згідно з принципом локалізації можна чекати, що ця сторінка не використовуватиметься і в найближчому майбутньому. Ця стратегія недалека від оптимальної. Робота алгоритму проілюстрована на рис. 11.7.

Порівнюючи цей алгоритм з іншими (з тим же потоком даних), можна побачити, що використання LRU алгоритму дозволяє скоротити кількість сторінкових порушень.

LRU – хороший, але важкий в реалізації алгоритм. Необхідно мати зв'язаний список усіх сторінок в пам'яті, на початку якого будуть зберігатися нещодавно використані сторінки. Причому цей список повинен оновлюватися при кожному зверненні до пам'яті.

Звернення до сторінок	2	3	2	1	5	2	4	5	3	2	5	2
Алгоритм LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
4 page faults					F		F		F	F		

Рисунок 11.7 – Приклад роботи LRU алгоритму для трьох кадрів

Оскільки більшість сучасних процесорів не надають відповідної апаратної підтримки для реалізації алгоритму LRU, хотілося б мати алгоритм, досить близький до LRU, але такий, що не вимагає спеціальної підтримки.

Один з різновидів схеми LRU називається **алгоритмом нечастого затребування – NFU (Not Frequently Used)**. Для цього алгоритму потрібний програмний лічильник, пов'язаний з кожною сторінкою в пам'яті, спочатку він рівний нулю. Під час кожного переривання по таймеру (а не після кожної інструкції) операційна система досліджує усі сторінки в пам'яті. Біт use кожної сторінки (він дорівнює 0 або 1) додається до лічильника, а потім прапор звернення скидається.

За допомогою лічильника намагаються відстежити, як часто відбувалося звернення до кожної сторінки. При сторінковому перериванні для заміщення вибирається сторінка з найменшим значенням лічильника.

Основна проблема, що виникає при роботі з алгоритмом NFU, полягає в тому, що він ніколи нічого не забуває. Наприклад, сторінка, до якої дуже часто зверталися впродовж деякого часу, а потім звертатися перестали, все одно не буде видалена з пам'яті, тому що її лічильник містить велику величину. Наприклад, у багатопрохідному компіляторі сторінки, які часто використовувалися під час першого проходу, можуть все ще мати високе значення лічильника при пізніших проходах. На щастя, невеликі зміни в алгоритмі дозволяють розв'язати цю проблему, яка дозволяє йому «забувати», досить добре моделювати алгоритм LRU:

- кожен лічильник зрушується вправо на один розряд перед збільшенням біта use;
- біт use додається в крайній ліворуч, а не в крайній справа біт лічильника.

Коли відбувається сторінкове переривання, видаляється та сторінка, чий лічильник має найменшу величину. Ясно, що лічильник сторінки, до якої не було

звернень. наприклад, за чотири тіка (tick – період таймера), розпочинатиметься з чотирьох нулів і, таким чином, матиме нижче значення, ніж лічильник сторінки, на яку не посилалися впродовж тільки трьох тіків годинника.

На рис. 11.8 показано, як працює модифікований алгоритм, відомий ще як алгоритм старіння. Припустимо, що після першого переривання від таймера біт use(U) для сторінок від 0 до 5 має, відповідно, значення 1, 0, 1, 0, 1 і 1. Іншими словами, між перериваннями від таймера, що відповідають тактам 0 і 1, було звернення до сторінок 0, 2, 4 і 5, в результаті якого їх біти U були встановлені в 1, а у інших сторінок їх значення залишилося рівним 0. Після того, як були зміщені значення шести відповідних лічильників і ліворуч було вставлено значення біта U, вони набули значень, показаних на рис. 11.8, а. У чотирьох стовпцях, що залишилися, показані стани шести лічильників після наступних чотирьох переривань від таймера.

Цей алгоритм відрізняється від алгоритму LRU двома особливостями. Розглянемо сторінки 3 і 5 на рис. 11.8, д. Ні до однієї з них за два переривання від таймера не було жодного звернення, але до обох було звернення за переривання від таймера, що передувало цим двом. Відповідно до алгоритму LRU якщо сторінка має бути видалена, то алгоритм повинен вибрати одну з цих двох сторінок. Проблема в тому, що не відомо, до якої з них зверталися в останню чергу між тактом 1 і тактом 2.

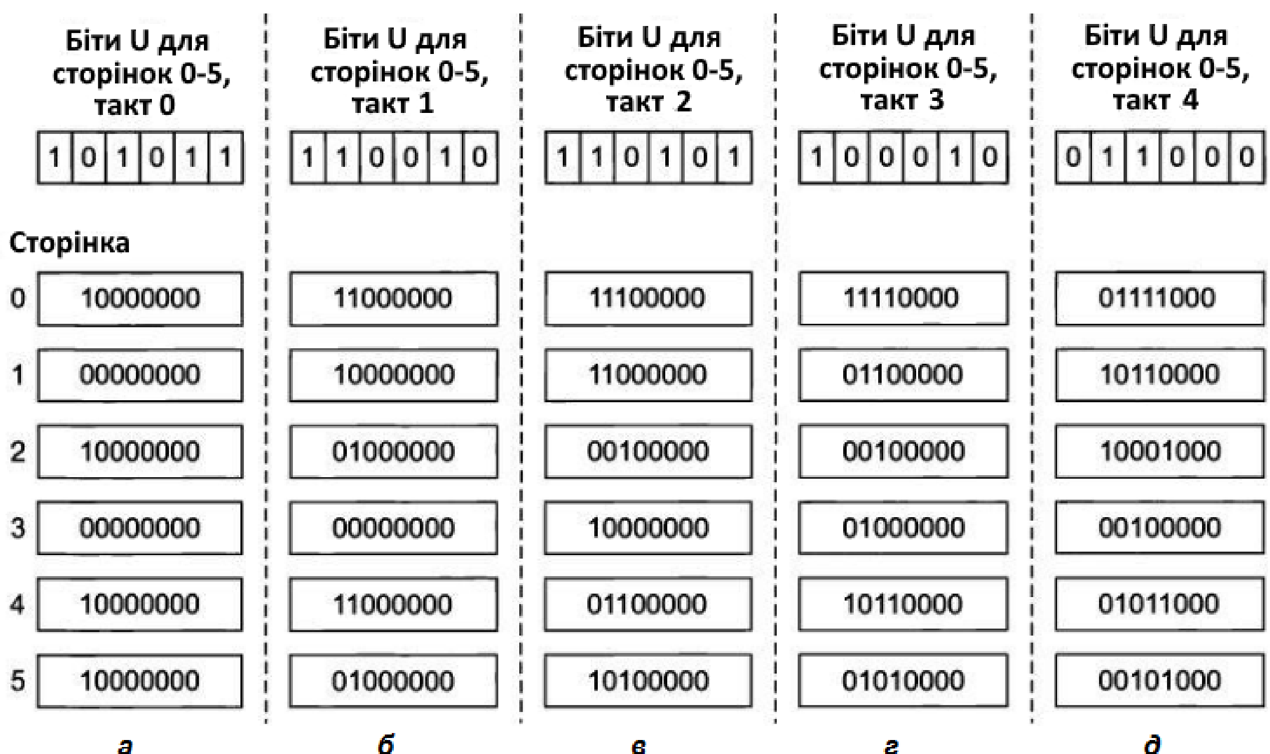

Рисунок 11.8 – Приклад роботи алгоритму старіння (NFU)

При записі тільки одного біта за інтервал між двома периваннями від таймера ми втратили можливість відрізнити раніше звернення від пізнішого. Все, що ми можемо зробити, – це видалити сторінку 3, оскільки до сторінки 5 також було звернення двома тактами раніше, а до сторінки 3 такі звернення не були.

Друга відмінність між алгоритмом LRU і алгоритмом старіння полягає в тому, що в алгоритмі старіння лічильник має обмежену кількість біт (у цьому прикладі – 8 біт), яка звужує горизонт минулого, що переглядається ним. Припустимо, що у кожної з двох сторінок значення лічильника дорівнює нулю. Все, що ми можемо зробити, це вибрати одну з них довільним чином. Насправді цілком може виявитися, що до однієї з цих сторінок останнє звернення було 9 тактів назад, а до другої – 100 тактів назад. І цю обставину встановити неможливо. Але на практиці 8 біт цілком достатньо, якщо між перериваннями від таймера проходить приблизно 20 мс. Якщо до сторінки не було звернень впродовж 160 мс, то вона, напевно, вже не так важлива.

Ми розглянули декілька різних алгоритмів заміщення сторінок. Оптимальний алгоритм замінює ту сторінку, звернення до якої робилося раніше інших, що знаходяться в даний момент в пам'яті. На жаль, не існує способу визначення того, яка сторінка буде останньою, тому цей алгоритм не може використовуватися на практиці. Але він корисний в якості тестової задачі, відносно якого можна оцінювати інші алгоритми. На рис 11.9 зображена поведінка різних стратегій заміщення сторінок відносно оптимального алгоритму.

Звернення до сторінок	**2**	**3**	**2**	**1**	**5**	**2**	**4**	**5**	**3**	**2**	**5**	**2**
Оптимальний алгоритм	2	2	2	2	2	2	4	4	4	2	2	2
		3	3	3	3	3	3	3	3	3	3	3
				1	5	5	5	5	5	5	5	5
3 page faults					**F**		**F**			**F**		
Алгоритм LRU	2	2	2	2	2	2	2	2	3	3	3	3
		3	3	3	5	5	5	5	5	5	5	5
				1	1	1	4	4	4	2	2	2
4 page faults					**F**		**F**		**F**	**F**		
Алгоритм FIFO	2	2	2	2	5	5	5	5	3	3	3	3
		3	3	3	3	2	2	2	2	2	5	5
				1	1	1	4	4	4	4	4	2
6 page faults					**F**	**F**	**F**		**F**		**F**	**F**
Годинниковий алгоритм	2*	2*	2*	2*	5*	5*	5*	5*	3*	3*	3*	3*
		3*	3*	3*	3	2*	2*	2*	2	2*	2	2*
				1*	1	1	4*	4*	4	4	5*	5*
5 page faults					**F**	**F**	**F**		**F**		**F**	

Рисунок 11.9 – Поведінка чотирьох алгоритмів заміщення сторінок

11.3.4 Буферизація сторінок

Хоча алгоритми «найдовше невикористаний» і «годинниковий» перевершують алгоритм «першим увійшов – першим вийшов», вони обидва складні і мають високі накладні витрати в порівнянні з останнім. Крім того, слід враховувати, що вартість заміщення модифікованої сторінки перевищує номінальну вартість заміщення немодифікованої сторінки, яку не треба записувати у вторинну пам'ять.

Є ще одна цікава стратегія, яка може підвищити продуктивність сторінкової організації при використанні найпростішого алгоритму заміщення. Це – **буферизація сторінок**, використана в VAX VMS. В якості алгоритму заміщення сторінок використовується найпростіший алгоритм «першим увійшов – першим вийшов». Для підвищення його продуктивності сторінка, що зміщується, не втрачається, а вноситься в один з двох списків: в список вільних сторінок, якщо сторінка не модифікувалася, або в список модифікованих сторінок. Зауважимо, що фізично сторінка не переміщується – замість цього її запис видаляється з таблиці сторінок і переноситься в список вільних або модифікованих сторінок.

Список вільних сторінок являє собою список кадрів сторінок, доступних для читання. VMS намагається постійно підтримувати деяку невелику кількість вільних кадрів. Коли сторінка зчитується в кадр, використовується кадр, розташований на початку списку; при цьому сторінка, яка перебувала в ньому раніше, знищується. При заміщенні немодифікованої сторінки вона залишається в пам'яті, а її кадр додається до кінця списку вільних сторінок; аналогічно, модифікована сторінка додається до списку модифікованих сторінок.

Важливим аспектом цих переміщень є те, що заміщувані сторінки залишаються в пам'яті. Отже, якщо процес звертається до такої сторінки, вона повертається в резидентну множину процесу без значних витрат. Насправді, списки вільних і модифікованих сторінок працюють в якості кеша сторінок. Список модифікованих сторінок дозволяє записувати їх не по одній, а кластерами, що істотно знижує кількість операцій введення-виведення, а, отже, і час звернення до диска.

11.3.5 Стратегія заміщення і розмір кеша

Як зазначалося раніше, розмір основної пам'яті з часом стає все більше, як і розмір додатків. Втіхою може служити те, що розміри кешів також збільшуються. При використанні кешів великого розміру заміщення сторінок віртуальної пам'яті може впливати на продуктивність. Якщо кадр сторінки, що обраний для заміщення, розташовується в кеші, то разом з втратою сторінки з блоку кеша втрачається весь блок.

У системах з використанням буферизації того чи іншого виду продуктивність кеша можна збільшити шляхом додавання до стратегії заміщення стратегію розміщення сторінок у буфері. Більшість операційних систем розміщують сторінки в буфері в довільних кадрах, як правило, з використанням алгоритму «першим увійшов – першим вийшов». Дослідження показали, що правильний вибір стратегії розміщення може призвести до зменшення неуспішних пошуків в кеші на 10-20%.

Суть цих стратегій полягає в розміщенні послідовних сторінок в основній пам'яті таким чином, щоб мінімізувати кількість кадрів сторінок, що відображаються в одні і ті ж слоти кеша.