ЛЕКЦІЯ 7. ВЗАЄМОДІЯ З ДИСКОМ ПІД ЧАС КЕРУВАННЯ ПАМ'ЯТТЮ. ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ

- Підкачування сторінок пам'яті
- Зберігання і заміщення сторінок пам'яті
- Резидентна множина і пробуксовування
- Організація віртуальної пам'яті

7.1. Підкачування сторінок пам'яті

Тимчасове збереження окремих частин адресного простору на диску допомагає розв'язати одну з основних проблем, що виникають під час реалізації керування пам'яттю в ОС, а саме організацію завантаження і виконання програм, які окремо або разом не вміщаються в основній пам'яті.

Найпростішим і найдавнішим підходом є завантаження і вивантаження всього адресного простору процесу за один прийом. Процеси завантажуються у пам'ять повністю, виконуються певний час, а потім так само повністю вивантажуються на диск. Така технологія має низку недоліків:

- її використання призводить до значної фрагментації зовнішньої пам'яті;
- вона не дає змоги виконувати процеси, які мають потребу у більшому обсязі пам'яті, ніж доступно у системі;
- погано підтримуються процеси, які можуть виділяти собі додаткову динамічну пам'ять (це необхідно робити зі врахуванням можливого розширення адресного простору процесу).

Вивантаження всього процесу із пам'яті у сучасних ОС можна використовувати як засіб зниження навантаження, але лише на доповнення до інших технологій взаємодії з диском.

Поняття підкачування

Описана технологія повного завантаження і вивантаження процесів традиційно називалася підкачуванням або *простим підкачуванням*, але тут вживатимемо цей термін у ширшому значенні. У сучасних ОС під *підкачуванням* (swapping) розуміють увесь набір технологій, які здійснюють взаємодію із диском під час реалізації віртуальної пам'яті, щоб дати можливість кожному процесу звертатися до великого діапазону логічних адрес за рахунок використання дискового простору.

Розглянемо загальні принципи підкачування. Під час завантаження процесу в основну пам'ять у ній розміщають лише кілька його блоків, які потрібні для початку роботи. Частину адресного простору процесу, що у конкретний момент часу відображається на основну пам'ять, називають *резидентною множиною* процесу (resident set). Поки процес звертається тільки до пам'яті резидентної множини, виконання процесу не переривають. Як тільки здійснюється посилання на блок, що перебуває за межами резидентної множини (тобто відображений на диск), відбувається апаратне переривання. Оброблювач цього переривання призупиняє процес і запускає дискову операцію читання потрібного блоку із диска в основну пам'ять. Коли блок зчитаний, операційну систему сповіщають про це, після чого процес переводять у стан готовності й, зрештою, поновлюють, після чого він продовжує свою роботу, ніби нічого й не сталося; на момент його поновлення потрібний блок уже перебуває в основній пам'яті, де процес і розраховував його знайти.

Реалізація підкачування використовує правило «дев'яносто до десяти». Ідеальною реалізацією керування пам'яттю є надання кожному процесові пам'яті, за розміром порівнянної із жорстким диском, а за швидкістю доступу – з основною пам'яттю. Оскільки за правилом «дев'яносто до десяти» на 10 % адресного простору припадає 90 % посилань на пам'ять, як деяке наближення до ідеальної реалізації можна розглядати такий підхід: зберігати ці 10 % в основній пам'яті, а інший адресний простір відображати на диск. Для прикладу, якщо частіше використовують сторінки 0, 3, 4, 6, тоді їхній вміст зберігають в основній пам'яті, а відповідно, сторінки 1, 2, 5, 7, які використовують рідше, і тому їхній вміст зберігають на диску (рис. 7.1).

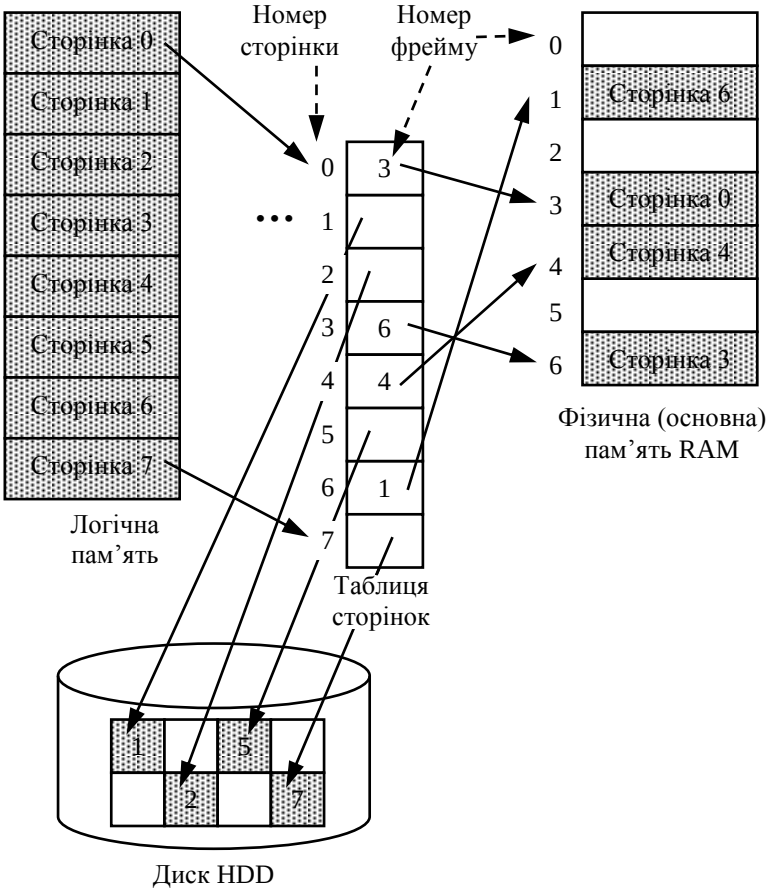


Рис. 7.1. Принцип дії підкачування

Базовий підхід, який використовують під час реалізації підкачування сторінок із диска, називають технологією *завантаження сторінок на вимогу* (demand paging). Процес переносу сторінки із диска у пам'ять називають *завантаженням із диска* (swapping in), процес переносу сторінки із пам'яті на диск – *вивантаженням на диск* (swapping out).

Для організації апаратної підтримки підкачування кожний елемент таблиці сторінок має містити спеціальний біт – біт присутності сторінки у пам'яті *P*. Коли він дорівнює одиниці, то це означає, що відповідна сторінка завантажена в основну пам'ять і їй відповідає деякий фрейм, інакше – сторінка перебуває на диску і має бути завантажена в основну пам'ять перед використанням (рис. 7.2). У таблиці сторінок архітектури IA-32 даний біт називають Present.

Для сторінки може бути заданий *біт модифікації M*. Його спочатку (під час завантаження сторінки із диска) прирівнюють нулю, а потім – одиниці, якщо сторінку модифікують, коли вона завантажена у фрейм основної пам'яті. Наявність такого біта дає змогу оптимізувати операції вивантаження сторінок на диск. Коли намагаються вивантажити

на диск вміст сторінки, для якої біт *M* дорівнює нулю, це означає, що записування на диск можна проігнорувати, бо вміст сторінки не змінився від часу завантаження у пам'ять.

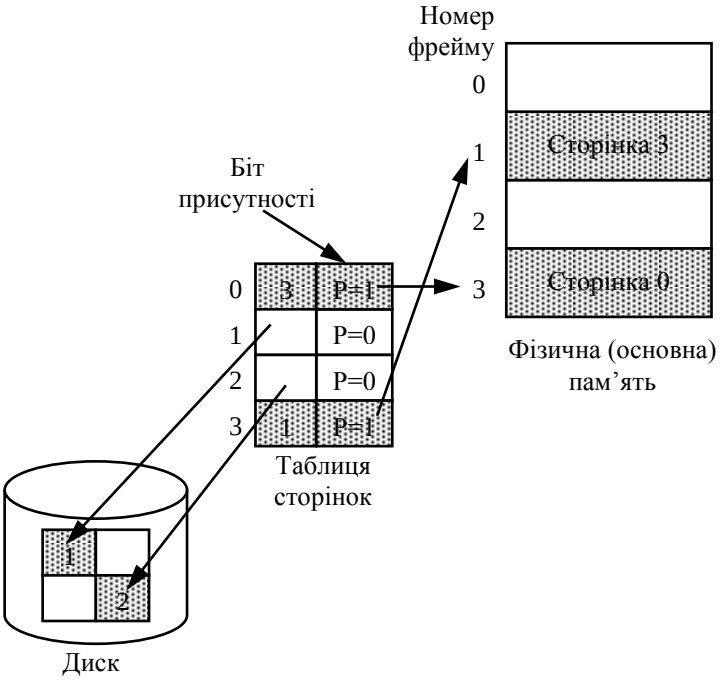


Рис. 7.2. Біт присутності сторінки

Коли процес робить спробу доступу до сторінки, для якої біт *P* дорівнює нулю, то відбувається апаратне переривання. Його називають *сторінковим перериванням або* ***сторінковою відмовою*** (page fault). ОС має обробити це переривання.

- Сторінку перевіряють на доступність для цього процесу (діапазон доступної пам'яті зберігається у структурі даних процесу).
- Якщо сторінка недоступна, процес переривають, якщо доступна, знаходять вільний фрейм фізичної пам'яті та ставлять у чергу дискову операцію читання потрібної сторінки в цей фрейм.
- Після читання модифікують відповідний запис таблиці сторінок (біт присутності покладають рівним одиниці).
- Перезапускають інструкцію, що викликала апаратне переривання.

Тепер процес може отримати доступ до сторінки, ніби вона завжди була в пам'яті. Процес поновлюють у тому самому стані, в якому він був перед перериванням.

Зазначимо, що в загальному випадку сторінкове переривання не обов'язково спричиняє підкачування сторінки з диска; воно може бути результатом звертання до сторінки, що не належить до адресного простору процесу. У цьому разі має бути згенерована помилка. З іншого боку, причиною сторінкового переривання під час чистого завантаження на вимогу може бути звертання до зовсім нової сторінки пам'яті, яка жодного разу не була використана процесом. Оброблювач може зарезервувати новий фрейм, проініціалізувати його (наприклад, заповнити нулями) і поставити йому у відповідність сторінку.

Реалізація завантаження на вимогу із підкачуванням сторінок із диска може істотно впливати на ***продуктивність*** ОС. Зробимо найпростіший розрахунок такого впливу.

Спрощену характеристику, яку використовують для оцінки продуктивності системи із завантаженням сторінок на вимогу, називають *ефективним часом доступу* і розраховують за такою формулою:

$$T_{ва} = (1 - P_{pf}) \times T_{ma} + P_{pf} \times T_{pf},$$

де P_{pf} – імовірність сторінкового переривання; T_{ma} – середній час доступу до пам'яті; T_{pf} – середній час обробки сторінкового переривання.

У сучасних системах час доступу до пам'яті T_{ma} становить від 20 до 200 нс. Розрахуємо час обробки сторінкового переривання T_{pf} .

Коли відбувається сторінкове переривання, система виконує багато різноманітних дій (генерацію переривання, виклик оброблювача, збереження регістрів процесора, перевірку коректності доступу до пам'яті, початок операції читання із диска, перемикання контексту на інший процес, обробку переривання від диска про завершення операції читання, корекцію таблиці сторінок тощо). У результаті кількість інструкцій процесора, необхідних для обробки сторінкового переривання, виявляється дуже великою (десятки тисяч), і середній час обробки сягає кількох мілісекунд.

Припустимо, що величина T_{ma} становить 100 не, T_{pf} – 10 мс, а ймовірність виникнення сторінкового переривання – 0,001 (одне переривання на 1000 звертань до пам'яті, цю величину називають *рівнем сторінкових переривань* – page fault rate). Тоді ефективний час доступу буде рівний:

$$T_{ва} = (1 - 0{,}001) \times 100 + 0{,}001 \times 10\,000\,000 = 10\,099{,}9 \text{ нс.}$$

Як бачимо, внаслідок сторінкових переривань, що виникають з імовірністю одна тисячна, ефективний час доступу виявився в 100 разів більшим, ніж під час роботи з основною пам'яттю, тобто продуктивність системи знизилася в 100 разів. Щоб здобути зниження продуктивності на прийнятну величину, наприклад на 10 %, необхідно, щоб імовірність сторінкового переривання була приблизно одна мільйонна:

$$P_{pf} = (0{,}1 \times T_{ma}) / (T_{pf} - T_{ma}) = 10 / 9\,999\,900 = 1 / 999\,990.$$

По суті, будь-які витрати часу на виконання найскладніших алгоритмів окупатимуться, якщо зменшується кількість сторінкових переривань, оскільки одне таке переривання обробляється повільніше, ніж виконується алгоритм майже будь-якої складності.

Під час реалізації підкачування потрібно дати відповідь на такі запитання.

1. Які сторінки потрібно завантажити із диска і в який час?

Відповідь на це запитання визначає прийнята в цій системі *стратегія вибірки сторінок.* Однією з таких стратегій є завантаження сторінок на вимогу, яку щойно було розглянуто, коли сторінку завантажують у пам'ять тільки під час обробки сторінкового переривання. Іншою стратегією такого роду є *попереднє завантаження сторінок* (prepaging), коли у пам'ять завантажують кілька сторінок, розташованих на диску послідовно для того щоб зменшити кількість таких сторінкових переривань.

2. Яку сторінку потрібно вивантажити на диск, коли вільних фреймів у основній пам'яті більше немає?

Відповідь на це запитання визначає *стратегія заміщення сторінок.* Різні підходи до реалізації цієї стратегії розглянемо у наступному питанні цієї лекції.

3. Де у фізичній пам'яті мають розміщуватися сторінки процесу?

Відповідь на це запитання визначає *стратегія розміщення.* Вона відіграє важливу роль під час реалізації підкачування на основі сегментації, у разі підкачування на основі сторінкової організації вона не така важлива, оскільки всі сторінки рівноправні й можуть перебувати у пам'яті де завгодно.

4. Яким чином організувати керування резидентною множиною?

Головне завдання такого керування – забезпечити, щоб у резидентній множині були тільки сторінки, які дійсно потрібні процесу.

7.2. Зберігання і заміщення сторінок пам'яті

Під час генерації сторінкового переривання може виникнути ситуація, коли жодного вільного фрейму у фізичній пам'яті немає.

За відсутності вільного фрейму в пам'яті ОС має вибрати, яку сторінку вилучити із пам'яті для того щоб вивільнити місце для нової сторінки. Процес цього вибору визначає *стратегія заміщення сторінок.* Коли сторінка, яку вилучають, була змінена, необхідно вивантажити її на диск для відображення цих змін, коли ж вона залишилася незмінною (наприклад, це була сторінка із програмним кодом), цього робити не потрібно. Для індикації того, чи була сторінка змінена, використовують біт модифікації сторінки *M.*

Яку саме сторінку потрібно вилучати із пам'яті, визначає *алгоритм заміщення сторінок* (page replacement algorithm). Вибір такого алгоритму відчутно впливає на продуктивність системи, тому що вдало підібраний алгоритм зменшує кількість сторінкових переривань, а невдала його реалізація може її значно збільшити (якщо вилучити часто використовувану сторінку, то вона незабаром може знову знадобитися і т. д.).

Слід зазначити, що алгоритми заміщення сторінок (особливо ті, які справді дають виграш у продуктивності) досить складні в реалізації і майже завжди потребують апаратної підтримки (хоча б у вигляді наявності біта модифікації сторінки *M).*

Ось який вигляд матиме алгоритм завантаження сторінки з урахуванням заміщення сторінок.

1. Знайти вільний фрейм у фізичній пам'яті:
   а) якщо вільний фрейм є, використати його (перейти до кроку 2);
   б) якщо вільного фрейму немає, використати алгоритм заміщення сторінок для того щоб знайти *фрейм-жертву* (victim frame);
   в) записати фрейм-жертву на диск (якщо біт модифікації для нього ненульовий), відповідно змінити таблицю сторінок і таблицю вільних фреймів.
2. Знайти потрібну сторінку на диску.
3. Прочитати потрібну сторінку у вільний фрейм (якщо раніше були виконані кроки б і в п. 1, цим фреймом буде той, котрий щойно вивільнився).
4. Знову запустити інструкцію, на якій відбулося переривання.

Оскільки реалізація алгоритмів заміщення сторінок важлива з погляду продуктивності системи, вони ретельно досліджувалися, були виділені критерії їхнього порівняння і формат даних для тестування.

Критерієм для порівняння алгоритмів заміщення звичайно є рівень сторінкових переривань: що він нижчий, то кращим вважають алгоритм. Оцінку алгоритму здійснюють через підрахунок кількості сторінкових переривань, які виникли внаслідок його запуску для конкретного набору посилань на сторінки у пам'яті. Набір посилань на сторінки подають *рядком посилань* (reference string) із номерами сторінок. Такий рядок можливо згенерувати випадково, а можна отримати відстеженням посилань на пам'ять у реальній системі (у цьому разі даних може бути зібрано дуже багато). Групу із кількох посилань, що йдуть підряд, на одну й ту саму сторінку в рядку відображають одним номером сторінки, оскільки такий набір посилань не може спричинити сторінкового переривання.

У цій лекції використовуватимемо такий рядок посилань: 1, 2, 3, 5, 2, 4, 2, 4, 3, 1, 4.

Алгоритми оцінюватимемо для трьох доступних фреймів.

Найпростішим у реалізації (за винятком алгоритму випадкового заміщення) є ***алгоритм FIFO***. Він дозволяє замінити сторінку, яка **перебувала у пам'яті найдовше**.

На рис. 7.3 зображена робота цього алгоритму на рядку посилань (кольором тут і далі виділено сторінкові переривання)

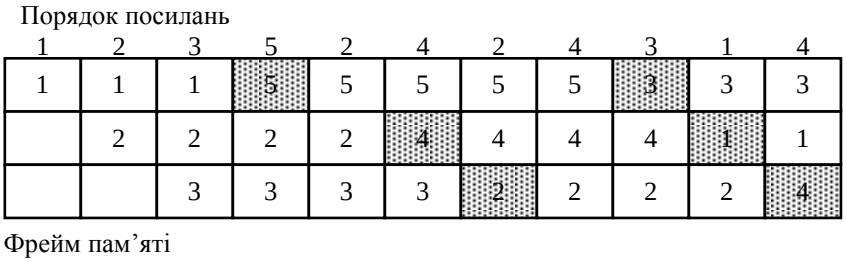


Рис. 7.3. FIFO-алгоритм заміщення сторінок

Для реалізації такого алгоритму досить підтримувати у пам'яті список усіх сторінок, організований за принципом FIFO-черги (звідси й назва алгоритму). Коли сторінку завантажують у пам'ять, її додають у хвіст черги, у разі заміщення її вилучають з голови черги.

Основними перевагами цього алгоритму є те, що він не потребує апаратної підтримки (тому для архітектур, які такої підтримки не надають, він може бути єдиним варіантом реалізації заміщення сторінок).

Головним недоліком алгоритму FIFO є те, що він не враховує інформації про використання сторінки. Такий алгоритм може вибрати для вилучення, наприклад, сторінку із важливою змінною, котра вперше отримала своє значення на початку роботи, і з того часу її постійно використовують та модифікують. Вилучення такої сторінки із пам'яті призводить до того, що система буде негайно змушена знову звернутися по неї на диск.

Наголосимо, однак, що жоден алгоритм заміщення не може призвести до некоректної роботи програми. Після того як сторінка буде завантажена у пам'ять із диска, робота з нею відбуватиметься так, ніби вона була в пам'яті від самого початку. Різниця полягатиме тільки у продуктивності (втім, ця різниця може виявитися дуже істотною).

Оптимальний алгоритм. Є алгоритм заміщення сторінок, оптимальність якого теоретично доведена (тобто він буде гарантовано кращим за будь-який інший алгоритм). Він зводиться до таких дій: замінити сторінку, яку **не використовуватимуть найдовше**. Приклад використання такого алгоритму зображено на рис. 7.4.

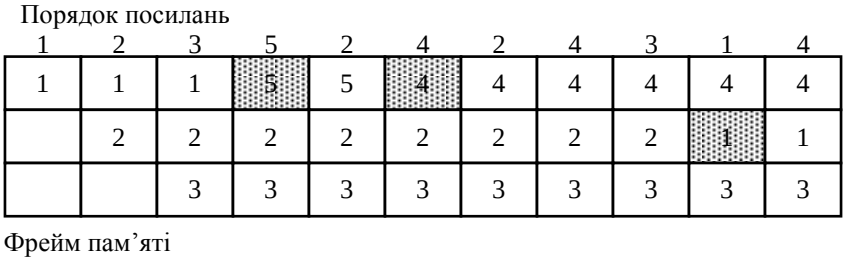


Рис. 7.4. Оптимальний алгоритм заміщення сторінок

На жаль, у загальному випадку реалізувати оптимальний алгоритм заміщення сторінок неможливо, бо він вимагає знання того, як у майбутньому буде поводитися процес.

Алгоритм LRU. Оскільки оптимальний алгоритм заміщення сторінок прямо реалізувати неможливо, основним завданням розробників має бути максимальне наближення до оптимального алгоритму. Опишемо основні принципи такого наближення.

Головною особливістю оптимального алгоритму (крім використання знання про майбутнє, що на практиці реалізувати не можна) є те, що він ґрунтується на збереженні для кожної сторінки інформації про те, коли до неї ***зверталися востаннє***. Збереження цієї інформації за умови заміни майбутнього часу на минулий привело до найефективнішого алгоритму з тих, які можна реалізувати – алгоритму *LRU* (Least

Recently Used – алгоритм заміщення сторінки, яка не використовується найдовше). Формулюють його так: замінити сторінку, що **не була використана упродовж найбільшого проміжку часу**.

Рис. 7.5 ілюструє роботу цього алгоритму. По суті, LRU – це оптимальний алгоритм, розгорнутий за часом назад, а не вперед

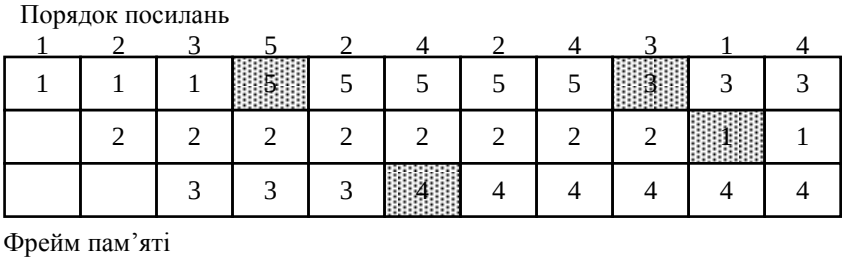


Рис. 7.5. LRU-алгоритм заміщення сторінок

Основні труднощі під час використання LRU-алгоритму полягають у тому, що його складно реалізувати, оскільки потрібно зберігати інформацію про кожне звертання до пам'яті так, щоб не страждала продуктивність. Потрібна набагато серйозніша апаратна підтримка, ніж наявність біта модифікації або асоціативна пам'ять. Таку підтримку можуть забезпечувати тільки деякі спеціалізовані архітектури. Розглянемо деякі можливі варіанти реалізації LRU-алгоритму.

- Можна організувати всі номери сторінок у вигляді двозв'язного списку. Під час кожного звертання до сторінки її вилучають зі списку (можливо, із середини) і поміщають у його початок. Тому сторінка, до якої зверталися найпізніше, буде завжди на початку списку, а та, до якої не зверталися найдовше (тобто жертва), – позаду.
- Можна організувати в процесорі глобальний лічильник (завдовжки, наприклад, 64 біти), збільшувати його на кожній інструкції та зберігати у відповідному елементі таблиці сторінок у разі звертання до кожної сторінки. Тоді потрібно замінювати сторінку із найменшим значенням лічильника.

Основний недолік цих підходів – низька продуктивність. Наприклад, якщо для поновлення лічильника або стека організовувати переривання, то оброблювач цього переривання буде виконуватися після кожної інструкції доступу до пам'яті, сповільнюючи доступ до пам'яті в кілька разів.

Базовий годинниковий алгоритм. Хоч алгоритм LRU реалізувати дуже важко і його апаратна підтримка є лише в деяких системах, все-таки сучасні апаратні архітектури дають змогу хоча б частково використати закладену в ньому ідею – виконати його наближення. Насамперед вони підтримують *біт використання сторінки* (reference bit, *R),* що перебуває в елементі таблиці сторінок і стає рівним одиниці у разі звертання до відповідної сторінки. Наявність такого біта дає змогу з'ясувати факт звертання до сторінки (не даючи змоги, однак, упорядкувати сторінки за часом звертання до них, що необхідно для LRU-алгоритму).

Наявність біта використання сторінки є основою *алгоритму другого шансу,* або *годинникового алгоритму* (clock algorithm), – одного з найефективніших реально застосовуваних алгоритмів.

Опишемо цей алгоритм. Передусім сторінки для нього мають бути організовані в кільцевий список (їх можна зобразити у вигляді циферблата годинника). Покажчик, який використовують під час сканування списку сторінок, ще називають *стрілкою* (подібно до стрілки годинника).

Спочатку беруть сторінку, що найдовше перебуває у пам'яті (як для FIFO). Якщо її біт використання *(R)* дорівнює 0, то сторінку замінюють на нову. Якщо ж біт *R* дорівнює 1 (до сторінки зверталися), тоді *R* прирівнюють до 0 (начебто ця сторінка тільки що завантажена у пам'ять), і прохід за списком триває далі, поки не буде знайдена сторінка з *R* = 0. Знайдену сторінку замінюють, після чого для нової сторінки задають *R* = 1 і наставляють на неї стрілку (рис. 7.6).

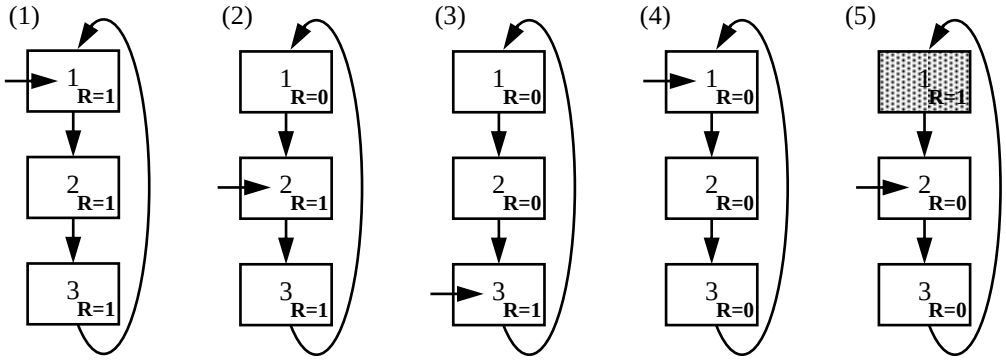


Рис. 7.6. Годинниковий алгоритм заміщення сторінок

На рис. 7.7 зображено результат застосування годинникового алгоритму для попереднього рядка посилань.

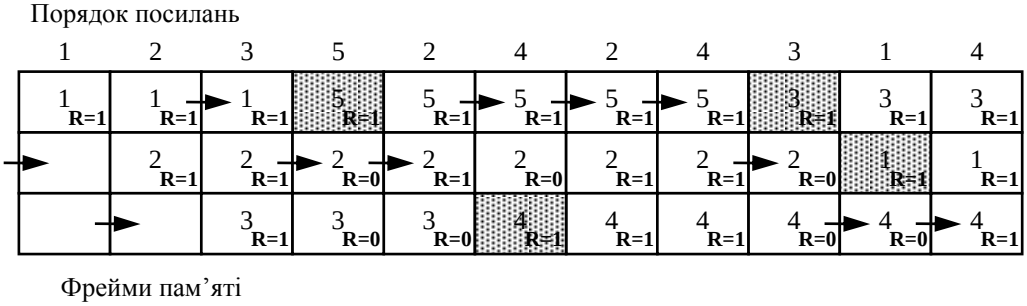


Рис. 7.7. Результат застосування годинникового алгоритму заміщення сторінок

На кожному кроці показане поточне положення стрілки і значення *R* для кожного фрейму. Зазначимо, що для цього рядка і трьох фреймів годинниковий алгоритм повністю ідентичний LRU (на практиці так буває не завжди).

Годинниковий алгоритм із додатковими бітами

Проблемою годинникового алгоритму є його підвищена чутливість до інтервалу часу між обходами стрілки. Якщо обхід відбувається надто рідко, виникає ситуація, коли всі або майже всі сторінки матимуть *R* = 1 і доводиться заходити на друге коло. Якщо ж обхід відбуватиметься надто часто, втрачатиметься інформація про використання сторінок (вона постійно затиратиметься стрілкою). Для вирішення цієї проблеми був запропонований *годинниковий алгоритм із додатковими бітами.* У разі використання цього алгоритму кожну сторінку супроводжують не один, а *n* бітів, які є лічильником використання сторінки C_u (наприклад, при *n*=8 лічильник можна розглядати як ціле завдовжки один байт), а також біт *R*. Під час використання сторінки біт *R* покладають рівним 1. Час від часу (за перериванням від таймера) ОС обходить циклічний список сторінок і поміщає значення біта *R* у старший біт лічильника, а інші біти зсуває вправо на 1 біт, при цьому молодший біт відкидають. Формула перерахування має такий вигляд:

$$C_u = (R \ll (n-1)) \mid (C_u \gg 1)$$

Біти лічильника в цьому випадку містять історію використання сторінки впродовж останніх *n* періодів часу між обходами списку. Наприклад, якщо сторінка за цей час не була використана жодного разу, всі біти C_u для неї будуть рівні 0, якщо її

використовували щоразу – одиниці. Кожне звертання до сторінки робить значення лічильника використання для неї більшим, ніж для всіх сторінок, які на цьому інтервалі не були використані. Наприклад, бітове значення 10000000 = 128 (сторінка щойно була використана) буде більшим за 01111111 = 127 (сторінка була використана на всіх попередніх інтервалах, крім останнього).

За необхідності вибору сторінки для заміщення береться сторінка із найменшим значенням лічильника C_u. Така сторінка буде LRU-сторінкою для останніх n періодів часу між обходами списку.

Якщо сторінок із найменшим значенням лічильника декілька, можна вилучити із пам'яті їх усі або використати FIFO-алгоритм для вибору однієї з них.

Ще однією важливою технологією заміщення сторінок є ***буферизація сторінок.*** При цьому в системі підтримуються два списки фреймів – модифікованих фреймів L_m і вільних фреймів L_f. У *разі* заміщення сторінку не вилучають із пам'яті, а її фрейм вносять в один із цих списків: у список L_m якщо для неї біт M = 1, і у список L_f, якщо M = 0. Насправді сторінку фізично не вилучають із її фрейму: відповідний їй запис просто вилучають із таблиці сторінок, а інформацію з її фрейму поміщають у кінець відповідного списку.

Список L_f містить фрейми, які можна використати для розміщення сторінок після сторінкового переривання. У разі необхідності такого розміщення сторінку завантажують у перший фрейм списку, колишній вміст цього фрейму затирають, а сам він вилучається із L_f (тільки в цьому разі відбувається заміщення сторінки у звичайному розумінні).

Список L_m містить фрейми із модифікованими сторінками, які мають бути записані на диск перед розміщенням у відповідних фреймах нових даних. Таке записування може відбуватися не по одній сторінці, а в пакетному режимі по кілька сторінок за раз, при цьому заощаджується час на дискові операції. Після збереження сторінки на диску відповідний фрейм переносять у список L_f.

Найважливішою властивістю цього алгоритму є те, що списки L_f і L_m працюють як кеші сторінок. Справді, сторінки, що відповідають фреймам із цих списків, продовжують перебувати в пам'яті, а отже, якщо знову виникає необхідність використати таку сторінку, усе, що потрібно зробити, – це повернути відповідний елемент назад у таблицю сторінок, оскільки всі дані цієї сторінки продовжують перебувати у відповідному фреймі.

Під час сторінкового переривання спочатку перевіряють, чи є відповідний фрейм у списку L_f або L_m, якщо є – його використовують негайно і звертання до диска не відбувається, якщо ж немає, береться перший фрейм зі списку вільних і в нього завантажують сторінку із диска.

Зазначимо, що використання буферизації сторінок дає змогу обмежитися найпростішим алгоритмом для безпосереднього вибору заміщуваної сторінки, наприклад алгоритмом FIFO. Сторінки, помилково заміщені FIFO-алгоритмом, потраплять не відразу на диск, а спочатку в один зі списків L_f або L_m, тому якщо вони незабаром знадобляться знову, то все ще міститимуться у пам'яті.

Деякі простіші реалізації об'єднують списки L_f і L_m в один. Такий список працює аналогічно до L_f , тільки у разі заміщення сторінки на початку списку при M - 1 вона буде спочатку записана на диск.

Буферизацію сторінок використовують майже в усіх сучасних операційних системах.

Заміщення сторінок можна розділити на ***глобальне і локальне.*** За глобального заміщення процес може вибрати фрейм, що належить будь-якому процесові, і заванта-

жити в нього свою сторінку. Воно може призвести до того, що одна й та сама програма виконуватиметься з різною продуктивністю в різних умовах, оскільки на заміщення сторінок відповідного процесу впливає поведінка інших процесів у системі. За локального заміщення процес може вибрати тільки свій власний фрейм (пул вільних фреймів підтримують окремо для кожного процесу). Як результат, ділянки пам'яті процесу, які мало використовуються, виявляються втраченими для інших процесів.

У більшості сучасних ОС реалізоване глобальне заміщення сторінок або змішаний варіант, за якого більшу частину часу використовують локальне заміщення, а у разі нестачі пам'яті – глобальне.

Дотепер ми виходили з того, що будь-яка невикористовувана сторінка може бути заміщена у будь-який момент часу. Насправді це не так. Розглянемо послідовність подій, які можуть статися під час використання глобальної стратегії заміщення сторінок.

1. Процес А збирається виконати операцію введення, при цьому введені дані мають бути збережені в деякому буфері. Його розміщають у сторінці пам'яті, завантаженій в цей момент у відповідний фрейм. Пристрій введення має окремий контролер, який розуміє команду пересилання даних. Одним із параметрів цієї команди є адреса фізичної пам'яті, починаючи з якої зберігатимуться введені дані.
2. Процес А видає команду контролеру (вказавши як параметр адресу буфера) і призупиняється, чекаючи введення.
3. ОС перемикає контекст і передає керування процесу В.
4. Процес В генерує сторінкове переривання.
5. ОС виявляє, що вільних фреймів немає, тому застосовує алгоритм заміщення для пошуку сторінки, яку потрібно вивантажити на диск.
6. Внаслідок пошуку алгоритм вибирає для заміщення сторінку, де розташований буфер введення процесу А.
7. ОС заміщає сторінку, зберігаючи її дані на диску і завантажуючи у фрейм нову сторінку.
8. ОС перемикає контекст і передає керування знову процесу А.
9. Контролер починає запис даних за адресою, заданою на кроці 2.
10. Ця адреса тепер відповідає фрейму, у який завантажена сторінка процесу В, у результаті дані процесу В будуть перезаписані, і він швидше за все завершиться аварійно.

Щоб цього не сталося, сторінки, подібні до використаної для буфера введення, мають блокуватися в пам'яті (про них кажуть, що вони *пришпилені* – pinned). Звичайно таке блокування реалізоване на рівні додаткового біта в елементі таблиці сторінок. Якщо такий біт дорівнює одиниці, ця сторінка не може бути вибрана алгоритмом для заміщення сторінок, а відповідний фрейм не може стати фреймом-жертвою.

Використання блокування сторінок у пам'яті не обмежене описаною ситуацією. Наприклад, код і дані ОС мають увесь час перебувати в основній пам'яті, тому всі відповідні сторінки варто заблокувати у пам'яті. Пам'ять, що не бере участі у заміщенні сторінок, називають *невивантажуваною* (nonpaged memory) на противагу *вивантажуваній* (paged memory).

Блокування сторінки у пам'яті потенційно небезпечне тим, що ця сторінка взагалі не буде розблокована і залишиться в основній пам'яті надовго. На практиці це зазвичай зводиться до обмеження блокування сторінок процесами користувача (наприклад, деякі ОС приймають «підказки» від процесів із приводу блокування сторінок, але залишають за собою право ігнорувати їх у разі нестачі пам'яті або коли процес збирається блокувати занадто багато сторінок).

Дотепер ми припускали, що заміщення сторінок відбувається за необхідності (коли процес генерує сторінкове переривання, а ОС не знаходить вільного фрейму). Насправді із погляду продуктивності такий підхід не є оптимальним. У більшості сучасних ОС реалізується інший підхід – ***фонове заміщення сторінок.***

Під час запуску ОС стартує спеціальний фоновий процес, або потік, який називають *процесом підкачування* або *сторінковим демоном* (swapper). Цей процес час від часу перевіряє, скільки вільних фреймів є в системі. Якщо їхня кількість менша за деяку допустиму межу, сторінковий демон починає заміщувати сторінки різних процесів, вивільняючи відповідні фрейми. Він продовжує займатися цим доти, поки кількість вільних фреймів не перевищить потрібної межі чи він не вивільнить деяку наперед відому кількість фреймів. Таким чином кількість вільних фреймів у системі підтримують на певному рівні, що підвищує її продуктивність.

Фонове заміщення сторінок звичайно використовують разом із буферизацією сторінок, у цьому разі сторінковий демон переносить фрейми у список L_f або зберігає на диску сторінки зі списку L_m.

Зберігання сторінок на диску

Дисковий простір, що використовують для зберігання сторінок, називають *простором підтримки* (backing store) або *простором підкачування* (swap space), а пристрій (диск), на якому він перебуває, – *пристроєм підкачування* (swap device). Цей пристрій повинен бути якомога продуктивнішим.

У більшості випадків обмін даними із простором підкачування відбувається швидше, ніж із файловою системою, оскільки дані розподіляються більшими блоками і не потрібно працювати з каталогами і файлами. Ця перевага може бути використана по-різному. Наприклад, під час запуску процесу можна заздалегідь копіювати весь його адресний простір у простір підкачування і вести весь подальший обмін даними тільки із цим простором. У результаті на диску завжди перебуватиме образ процесу. При цьому одним сторінкам на диску відповідатимуть фрейми пам'яті, іншим – ні, але кожна сторінка, завантажена у фрейм пам'яті, завжди відповідатиме сторінці на диску. Така стратегія вимагає багато місця для простору підкачування.

7.3. Резидентна множина і пробуксовування

Поняття пробуксовування

Пробуксовуванням (thrashing) називають стан процесу, коли через сторінкові переривання він витрачає більше часу на підкачування сторінок, аніж власне на виконання. У такому стані процес фактично непрацездатний.

Пробуксовування виникає тоді, коли процес часто вивантажує із пам'яті сторінки, які йому незабаром знову будуть потрібні. У результаті більшу частину часу такі процеси перебувають у призупиненому стані, очікуючи завершення операції введення-виведення для читання сторінки із диска.

Назвемо деякі причини пробуксовування.

1. Процес не використовує пам'ять повторно (для нього не працює правило «дев'яносто до десяти»).
2. Процес використовує пам'ять повторно, але він надто великий за обсягом, тому його резидентна множина не поміщається у фізичній пам'яті.
3. Запущено надто багато процесів, тому їхня сумарна резидентна множина не поміщається у фізичній пам'яті.

У перших двох випадках ситуація майже не піддається виправленню, найкраща по-

рада, яку тут можна дати, – це збільшити обсяг фізичної пам'яті. У третьому випадку ОС може почати такі дії: з'ясувати, скільки пам'яті необхідно кожному процесові, і змінити пріоритети планування так, щоб процеси ставилися на виконання групами, вимоги яких до пам'яті можуть бути задоволені; заборонити або обмежити запуск нових процесів.

Можна створити таку програму, яка постійно звертатиметься до різних сторінок, розкиданих великим адресним простором, генеруючи багато сторінкових переривань. Насправді реальні застосування працюють не так: вони зберігають ***локальність посилань*** (locality of reference), коли на різних етапах виконання процес посилається тільки на деяку невелику підмножину своїх сторінок, що є одним із наслідків відомого правила «дев'яносто до десяти».

Якщо виділено достатньо пам'яті для всіх сторінок поточної локальності, сторінкові переривання до переходу до наступної локальності не генеруватимуться. Якщо пам'яті недостатньо, система перебуватиме у стані ***пробуксовування***.

Завданням керування резидентною множиною є така його динамічна корекція, щоб у будь-який момент часу ця множина давала змогу розміщувати всі сторінки поточної локальності. Для того щоб коректно керувати резидентною множиною процесу, необхідно знати той набір сторінок, який йому знадобиться під час виконання. Звичайно, у повному обсязі таке знання реалізоване бути не може, бо це вимагає вміння вгадувати майбутнє. Спроби оцінити потрібний процесу набір сторінок, ґрунтуючись на особливостях використання сторінок у недалекому минулому, привели до ***концепції робочого набору*** (working set).

Найважливішою характеристикою цього підходу є інтервал часу довжиною *T,* який відлічують від поточного моменту часу назад. Якщо *T* вимірюють у секундах, то цей інтервал визначає останні *T* секунд виконання процесу і задає вікно робочого набору. Ширину вікна завжди залишають постійною, а межі його зсуваються із часом; верхньою межею завжди є поточний момент часу, а нижньою – поточний момент мінус *T*.

Під робочим набором процесу розуміють усі сторінки, до яких він звертався у вікні робочого набору. У кожний новий момент часу робочий набір буде різним, оскільки вікно робочого процесу переміщується, сторінки, до яких не було доступу за час *T,* із набору вилучають, щойно використані сторінки додають у набір (рис. 7.8).

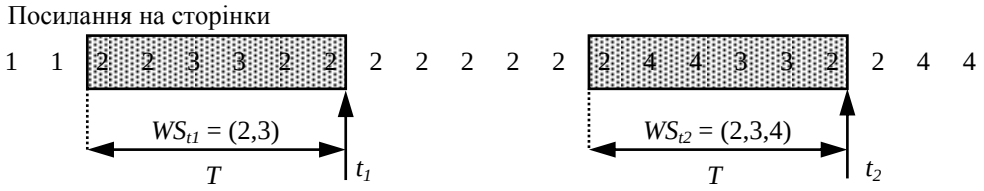


Рис. 7.8. Робочий набір процесу

Розглянемо, як можна використати концепцію робочого набору.

1. Виділяючи пам'ять, можна дати кожному процесові стільки сторінок, щоб у них помістився його робочий набір.
2. У разі заміщення сторінок можна вибирати для заміщення переважно ті сторінки, що не належать до робочого набору.
3. Під час планування процесів можна не ставити процес на виконання доти, поки всі сторінки його робочого набору не опиняться у пам'яті.

Концепцію робочого набору часто використовують у поєднанні із локальним заміщенням і буферизацією сторінок. Наприклад, можна постійно коригувати локальний кеш сторінок процесів для того щоб він приблизно відповідав за розміром його робочому набору. Цим підвищують гнучкість локального заміщення (є можливість дозволити

процесам використати через певний час вільні фрейми інших процесів, зберігши при цьому ізольованість одних процесів від інших).

У більшості випадків зниження ймовірності пробуксовування пов'язане із прогресом в апаратному забезпеченні.

- У комп'ютерах зараз встановлюють більше основної пам'яті, тому потреба у підкачуванні знижується, а сторінкові переривання трапляються рідше.
- Оскільки процесори стають швидшими, процеси виконуються із більшою швидкістю, раніше вивільняючи зайняту пам'ять.
- Хоча в системі завжди є процеси, які можуть використати всю доступну пам'ять, кількість процесів, яким достатньо виділеної пам'яті, у середньому збільшується. Фактично, найдієвіший спосіб боротьби із пробуксовуванням полягає у придбанні та встановленні додаткової основної пам'яті.

7.4. Реалізація віртуальної пам'яті в Windows

Сторінки і простір підтримки

Сторінки адресного простору процесу можуть бути *вільні* (free), *зарезервовані* (reserved) і *підтверджені* (committed).

Вільні сторінки не можна використати процесом прямо, їх потрібно спочатку зарезервувати. Це можливо зробити будь-яким процесом системи. Після цього інші процеси резервувати ту саму сторінку не можуть.

У свою чергу, процес, що зарезервував сторінку, не може цю сторінку використати до її підтвердження, тому що зв'язок із конкретними даними або програмами для неї не визначений.

Підтверджені сторінки безпосередньо пов'язані із простором підтримки на диску. Такі сторінки можуть бути двох типів.

- Дані для сторінок першого типу перебувають у звичайних файлах на диску. До таких сторінок належать сторінки коду (їм відповідають виконувані файли) і сторінки, що відповідають файлам даних, відображеним у пам'ять. Для таких сторінок простором підтримки будуть відповідні файли. Один і той самий файл може підтримувати блоки адресного простору різних процесів.
- Сторінки другого типу не пов'язані прямо із файлами на диску. Це може бути, наприклад, сторінка, що містить глобальні змінні програми. Для таких сторінок простором підтримки є спеціальний файл підкачування (swap file). У ньому не резервують простір доти, поки не з'явиться необхідність вивантаження сторінок на диск. Сторінки, зарезервовані у файлі підкачування, називають ще тіньовими сторінками (shadow pages).

На рис. 7.9 показано, як різні частини адресного простору можуть бути пов'язані з різними типами простору підтримки.

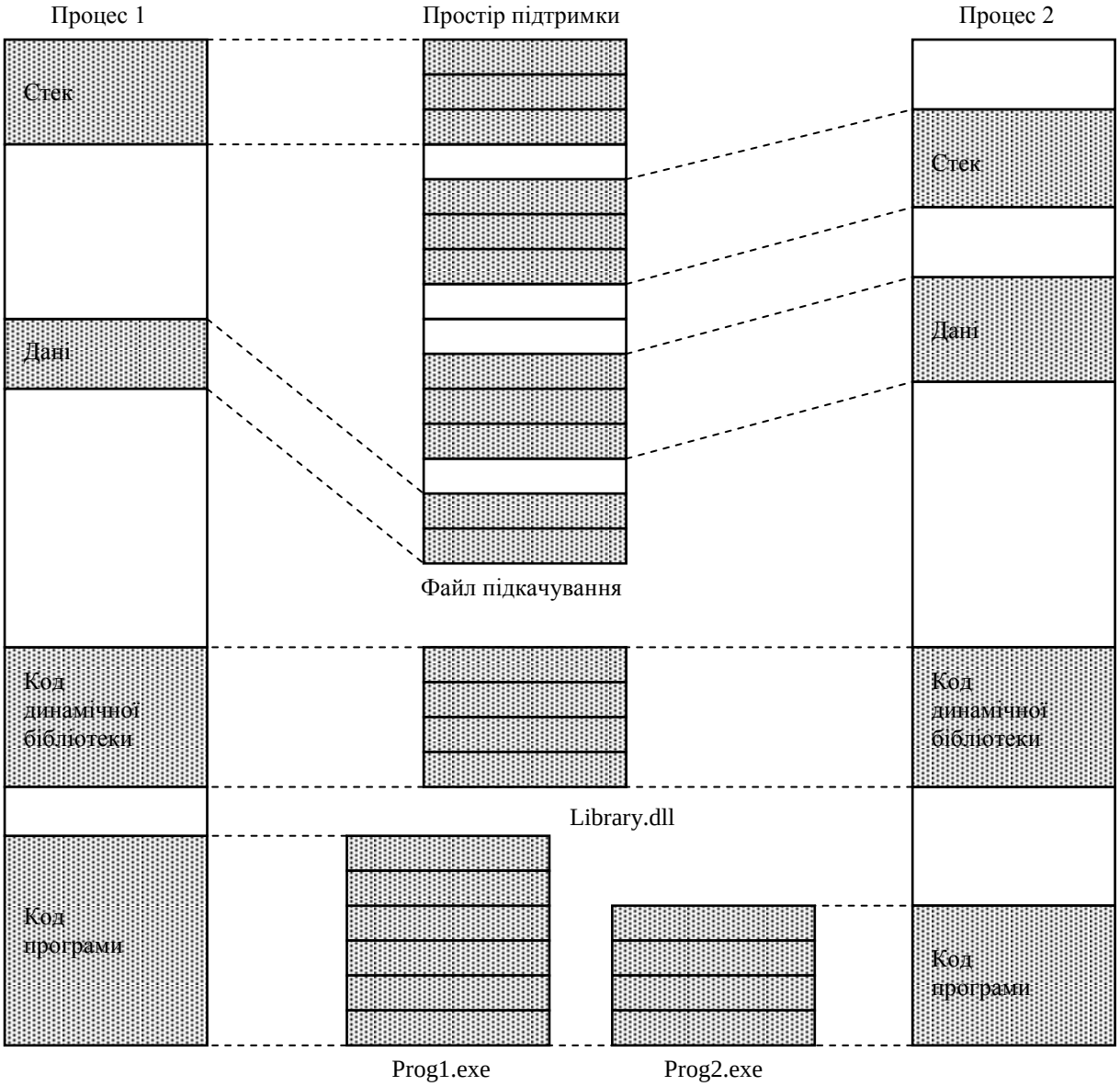


Рис. 7.9. Процеси і простір підтримки у Windows

Поняття регіону пам'яті. Резервування і підтвердження регіонів

Коли для процесу виділяють адресний простір, більшу його частину становлять вільні сторінки, які не можуть бути використані негайно. Для того щоб використати блоки цього простору, процес спочатку має зарезервувати в ньому відповідні регіони *пам'яті.*

Регіон пам'яті відображає неперервний блок логічного адресного простору процесу, який може використати застосування. Регіони характеризуються початковою адресою і довжиною. Початкова адреса регіону повинна бути кратною 64 Кбайт, а його розмір – кратним розміру сторінки (4 Кбайт).

Для резервування регіону пам'яті використовують функцію `VirtualAlloc()` (одним із її параметрів має бути прапорець `MEM_RESERVE`). Після того як регіон був зарезервований, інші запити виділення пам'яті не можуть його повторно резервувати. Ось приклад виклику `VirtualAlloc()` для резервування регіону розміру `size`:

```
PVOID addr = VirtualAlloc(NULL, size, MEM_RESERVE, PAGE_READWRITE);
```

Першим параметром `VirtualAlloc()` є адреса пам'яті, за якою роблять виділення, якщо вона дорівнює `NULL`, пам'ять виділяють у довільній ділянці адресного простору процесу. Останнім параметром є режим резервування пам'яті, серед можливих значень

цього параметра можна виділити PAGE_READONLY – резервування тільки для читання; PAGE_READWRITE – резервування для читання і записування.

Результатом виконання VirtualAlloc() буде адреса виділеного регіону пам'яті.

Однак і після резервування регіону будь-яка спроба доступу до відповідної пам'яті спричинятиме помилку. Щоб такою пам'яттю можна було користуватися, резервування регіону має бути підтверджене (commit). Підтвердження зводиться до виділення місця у файлі підкачування сторінок (для сторінок регіону створюють відповідні тіньові сторінки). Для підтвердження резервування регіону теж використовують функцію VirtualAlloc(), але їй потрібно передати прапорець MEM_COMMIT:

```
VirtualAlloc(addr, size, MEM_COMMIT, PAGE_READWRITE);
```

Звичайною стратегією для застосування є резервування максимально великого регіону пам'яті, що може знадобитися для його роботи, а потім підтвердження цього резервування для невеликих блоків всередині регіону в разі необхідності. Це підвищує продуктивність, тому що операція резервування не потребує доступу до диска і виконується швидше, ніж операція підтвердження.

Можна зарезервувати і підтвердити регіон пам'яті протягом одного виклику функції

```
VirtualAlloc(addr, size, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
```

Після того, як використання регіону завершене, його резервування потрібно скасувати. Для цього використовують функцію

```
VirtualFree(addr, 0, MEM_RELEASE);
```

Зазначимо, що цій функції передають не розмір пам'яті, яка звільняється, а нуль (ОС сама визначає розмір регіону).

Наведемо приклад використання резервування і підтвердження пам'яті. Припустимо, що необхідно працювати із масивом у пам'яті, але в конкретний момент часу буде потрібний тільки один із його елементів. У цьому разі доцільно зарезервувати пам'ять для всього масиву, після чого підтверджувати пам'ять для окремих елементів.

```
// резервування масиву зі 100 елементів
int *array = (int *) VirtualAlloc(NULL,
    100 * sizeof(int), MEM_RESERVE, PAGE_READWRITE);
// підтвердження елемента масиву з індексом 10
VirtualAlloc(&array[10], sizeof(int), MEM_COMMIT,
             PAGE_READWRITE);
// робота із пам'яттю
array[10] = 200;
printf(*дані у пам'яті: %d\n", array[10]);
// вивільнення масиву
VirtualFree(array, O, MEM_RELEASE);
```

Обробка сторінкових переривань

Назвемо причини виникнення сторінкових переривань у Windows.

- Звертання до сторінки, що не була підтверджена.
- Звертання до сторінки із недостатніми правами. Ці два випадки є фатальними помилками і виправленню не підлягають.
- Звертання для записування до сторінки, спільно використовуваної процесами. У цьому разі можна скористатися технологією копіювання під час записування.
- Необхідність розширення стека процесу. У цьому разі оброблювач переривання має виділити новий фрейм і заповнити його нулями.

- Звертання до сторінки, що була підтверджена, але в конкретний момент не завантажена у фізичну пам'ять. Під час обробки такої ситуації використовують локальність посилань: із диска завантажують не лише безпосередньо потрібну сторінку, але й кілька прилеглих до неї, тому наступного разу їх уже не доведеться заново підкачувати. Цим зменшують загальну кількість сторінкових переривань.

Поточну кількість сторінкових переривань для процесу можна отримати за допомогою функції `GetProcessMemoryInfo()`:

```
#include <psapi .h>
PROCESS_MEMORY_COUNTERS Info;
GetProcessMemoryInfo(GetCurrentProcess(),&info,sizeof(info));
Printf("усього сторінкових збоїв: %d\n", info.PageFaultCount);
```

Організація заміщення сторінок

Базовий принцип реалізації заміщення сторінок у Windows – підтримка деякої мінімальної кількості вільних сторінок у пам'яті. Для цього використовують кілька концепцій: робочі набори, буферизацію, старіння, фонове заміщення і зворотне відображення сторінок.

Керування робочим набором і фонове заміщення сторінок

Поняття робочого набору є центральним для заміщення сторінок у Windows. У цій ОС під робочим набором розуміють множину підтверджених сторінок процесу, завантажених в основну пам'ять. Під час звертання до таких сторінок не виникатиме сторінкових переривань. Кожний набір описують двома параметрами: його нижньою і верхньою межами. Ці межі не є фіксованими, за певних умов процес може за них виходити; крім того, вони пізніше можуть мінятися. Початкове значення меж однакове для всіх процесів (нижня межа має бути в діапазоні 20-50, верхня – 45-345 сторінок).

Менеджер пам'яті постійно контролює сторінкові переривання для процесу, коригуючи його робочий набір. Якщо під час обробки сторінкового переривання виявляють, що розмір робочого набору процесу менший за мінімальне значення, до цього набору додають сторінку, якщо ж більший за максимальне значення - із набору вилучають сторінку. Оскільки всі ці дії стосуються робочого набору того процесу, що викликав сторінкове переривання, базова стратегія заміщення сторінок є локальною. Втім, локальність заміщення є відносною: у деяких ситуаціях система може коригувати робочий набір одного процесу за рахунок інших (наприклад, якщо для одного процесу бракує фізичної пам'яті, а для інших її достатньо).

Крім локального коригування робочого набору в оброблювачах сторінкових переривань, у системі також реалізовано глобальне фонове заміщення сторінок. Спеціальний потік ядра, який називають *менеджером балансової множини* (balance set manager), виконується за таймером раз за секунду і перевіряє, чи не опустилася кількість вільних сторінок у системі нижче за допустиму межу. Якщо так, потік запускає інший потік ядра – *менеджер робочих наборів* (working set manager), що забирає додаткові сторінки у процесів, коригуючи їхні робочі набори.

Під час вибору сторінок для вилучення із робочого набору застосовують модифікацію годинникового алгоритму із використанням концепції старіння сторінок. Із кожною сторінкою пов'язаний цілочисловий лічильник ступеня старіння. Усі сторінки набору обходять по черзі. Якщо біт R для сторінки дорівнює 0, лічильник збільшують на одиницю, якщо R дорівнює 1, лічильник обнулюється. Внаслідок обходу заміщуються сторінки із найбільшим значенням лічильника.

Заміщені сторінки не зберігають негайно на диску, а буферизують.

Організація буферизації сторінок у Windows доволі складна, але переважно дотримується базової схеми, описаної раніше. Система підтримує 5 списків сторінок, переходи між якими показані на рис. 7.11.

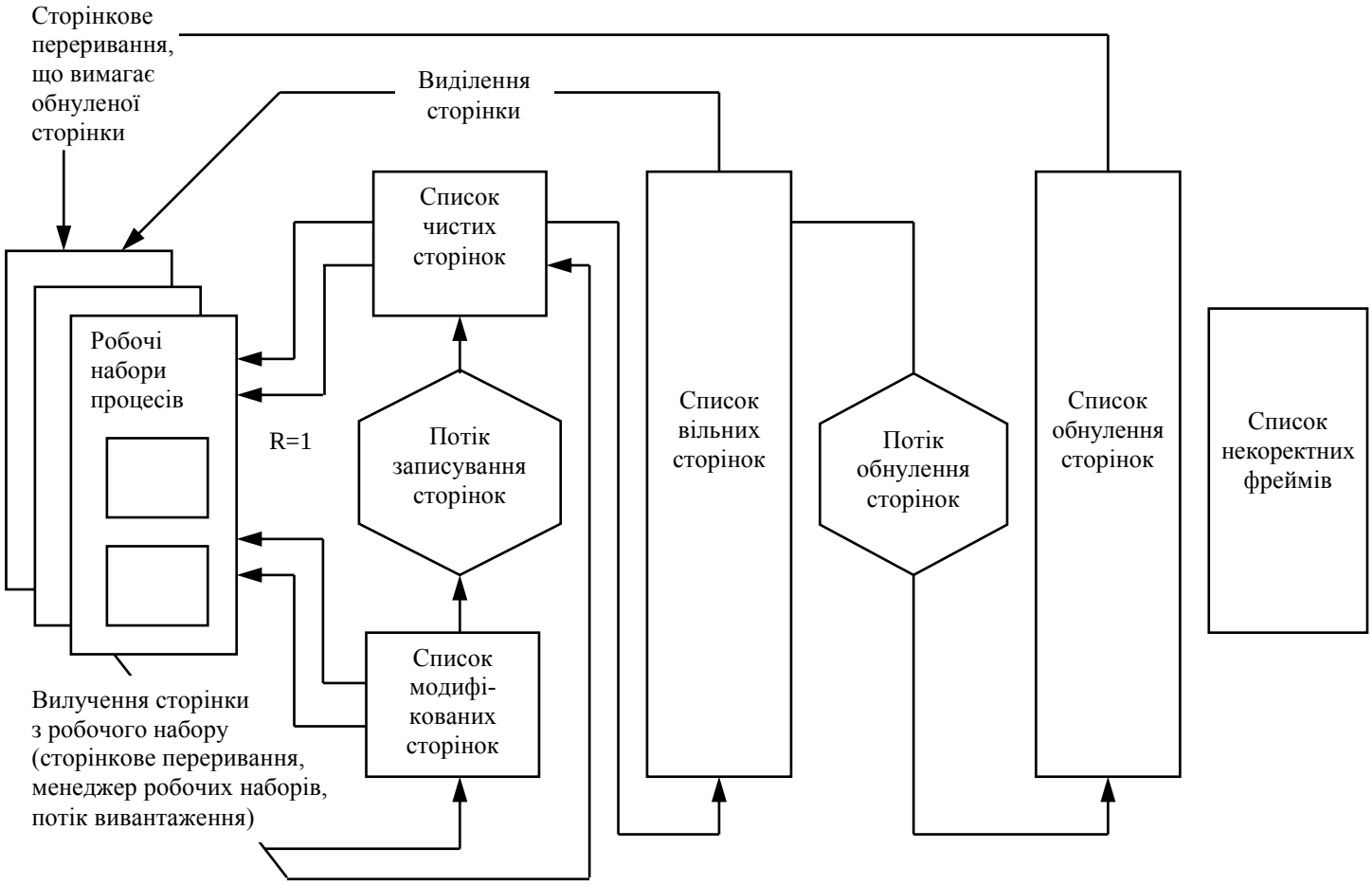


Рис. 7.11. Списки сторінок у Windows

- Модифікованих (`modified page list`). Містить вилучені із робочих наборів сторінки, які були модифіковані й котрі потрібно зберегти на диску. За принципом використання він аналогічний списку L_m.
- Чистих (`standby page list`). Містить вилучені із робочих наборів сторінки, які можна негайно використати для виділення пам'яті. Він аналогічний списку L_f. Списки чистих і модифікованих сторінок працюють як кеші сторінок.
- Вільних (`free page list`). У ньому перебувають сторінки, які не містять осмисленої інформації (немає сенсу повертати їх в робочі набори процесів).
- Обнулених (`zero page list`). Аналогічний списку вільних сторінок, але сторінки в ньому заповнені нулями.
- Некоректних фреймів фізичної пам'яті (`bad page list`). Фрейми в ньому відповідають ділянкам фізичної пам'яті, під час доступу до яких відбувався апаратний збій.

Фрейми із цього списку ніколи не будуть використані менеджером пам'яті. Зазначимо, що у Windows відсутня концепція списку активних сторінок, таким списком є робочі набори процесів.

Під час вилучення із робочого набору процесу (за сторінкового переривання або внаслідок роботи менеджера робочих потоків) сторінка потрапляє в кінець списку чистих або модифікованих сторінок (залежно від стану її біта M). Після завершення процесу в ці

списки переміщують усі його сторінки.

Крім того, за переміщення сторінок між списками відповідають кілька спеціалізованих потоків ядра.

- *Потік вивантаження* (swapper), який займається вивільненням сторінок неактивних процесів. Його запускають за таймером через 4 с. Якщо цей потік знаходить процес, усі потоки якого перебували у стані очікування упродовж деякого часу (від 3 до 7 с), він переміщує всі сторінки його робочого набору у списки чистих і модифікованих сторінок.
- *Потік записування модифікованих сторінок* (modified page writer), який виконує запис модифікованих сторінок на диск. Після того, як цей потік запише сторінку на диск, її переміщують у кінець списку чистих сторінок.
- Низькопріоритетний *потік обнулення сторінок* (zero page thread), переважно виконується, коли система не навантажена.

Коли потрібна нова сторінка, процес спочатку переглядає список вільних сторінок. Якщо він порожній, процес звертається до списку обнулених сторінок, якщо і в ньому немає жодного елемента, сторінку беруть зі списку чистих сторінок.

Попереднє завантаження сторінок

У Windows з'явилася підтримка попереднього завантаження сторінок. Вона ґрунтується на спостереженні за завантаженням коду програми. Воно часто сповільнюється внаслідок сторінкових переривань, які призводять до читання даних із різних файлів.

Для зменшення кількості файлів, до яких потрібно звертатися під час завантаження програмного коду, виконавча система Windows відслідковує сторінкові переривання упродовж 10с під час першого запуску застосування. Після цього зібрану інформацію, зокрема, сторінки, завантажені у пам'ять, зберігають у спеціальному *файлі попереднього завантаження застосування* у підкаталозі `Prefetch` системного каталогу Windows.

Під час наступних спроб запуску застосування перевіряють, чи не був для нього уже створений файл попереднього завантаження. Якщо це так, відбувається завантаження у пам'ять збережених сторінок даних із цього файла. При цьому звертатися до файлів, з яких були завантажені ці сторінки під час першого запуску застосування, не потрібно.

Аналогічні дії відбуваються й під час завантаження всієї системи, коли створюють *файл попереднього завантаження Windows*. Дані з цього файла будуть зчитані у пам'ять під час наступних завантажень системи.

Блокування сторінок у пам'яті

Як і в Linux, у Windows можна блокувати сторінки у пам'яті з коду режиму користувача. Для цього використовують функції Win32 API `VirtualLock()` і `VirtualUnlock()`. Відповідні сторінки мають бути до цього часу підтверджені.

```
SYSTEM_INFO info;  GetSystemInfo(&info);
DWORD pagesize = info.dwPageSize;
// резервування і підтвердження сторінки
char *pageaddr = (char *)VirtualAlloc(NULL,
       pagesize, MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
// блокування сторінки у пам'яті
VirtualLock(pageaddr, pagesize);
// ... розблокування
VirtualUnlock(pageaddr, pagesize);
```

Оскільки блокування сторінок призводить до того, що відповідна фізична пам'ять

виявляється втраченою для інших застосувань і системи, обсяг пам'яті, дозволеної для блокування, не може перевищувати нижньої межі робочого набору процесу.

7.6. Динамічний розподіл пам'яті

Динамічний розподіл пам'яті – це розподіл за запитами процесів користувача або ядра під час їхнього виконання. Розрізняють динамічне виділення та динамічне вивільнення пам'яті. Прикладом бібліотечної та мовної підтримки засобів динамічного розподілу пам'яті є функції `malloc()`, `free()` і `realloc()` бібліотеки мови C, оператори `new` і `delete` в C++. Усі ці засоби спираються на підтримку з боку операційної системи.

Зазначимо, що алгоритми цього розділу передбачають наявність суцільного лінійного адресного простору, абстрагуючись від того, як цей простір став доступний процесу, тобто від реалізації віртуальної пам'яті. Ці алгоритми є засобами розподілу пам'яті вищого рівня порівняно із засобами реалізації віртуальної пам'яті.

З іншого боку, деякі з цих алгоритмів можна використати для реалізації розподілу не тільки логічної, але й неперервної фізичної пам'яті, зокрема й для потреб ядра системи. У цьому розділі йтиметься саме про такі алгоритми на прикладах динамічного розподілу пам'яті ядром Linux і Windows.

Динамічна ділянка пам'яті процесу

Динамічна ділянка пам'яті процесу відображає спеціальну частину його адресного простору, у якій відбувається розподіл пам'яті за запитами з його коду (подібно до виклику `malloc()` для виділення пам'яті або `free()` для її вивільнення). Такий розподіл пам'яті відбувається в режимі користувача, звичайно в коді системних бібліотек (цей розподіл відбувається всередині динамічної ділянки, і ядро в ньому участі не бере).

Такі операції у більшості випадків не ведуть до зміни розмірів динамічної ділянки, але якщо така зміна необхідна (наприклад, якщо вільного місця в динамічній ділянці не залишилося), ОС пропонує для цього спеціальний системний виклик. В UNIX-системах його зазвичай називають `brk()`. Його пряме використання у прикладних програмах не рекомендоване, хоч і можливе.

Динамічний розподіл пам'яті відбувається на двох рівнях. Спочатку ядро резервує динамічну ділянку пам'яті процесу за системним викликом `brk()`. Потім усередині цієї ділянки процеси користувача можуть розподіляти пам'ять, викликаючи стандартні функції типу `malloc()` і `free()`. У разі потреби динамічну ділянку можна розширювати далі або скорочувати (знову-таки за допомогою виклику `brk()`).

Особливості розробки розподілювачів пам'яті

Розподілювач пам'яті (memory allocator) – частина системної бібліотеки або ядра системи, яка відповідає за динамічний розподіл пам'яті.

Головним завданням такого розподілювача є відстеження використання процесом ділянок пам'яті в конкретний момент. Основними цілями роботи розподілювача є мінімізація втраченого внаслідок фрагментації простору і витрат часу на виконання операцій (ці характеристики є основними критеріями порівняння алгоритмів розподілу пам'яті). При цьому розподілювач має враховувати принцип локальності (блоки пам'яті, які використовуються разом, виділяються поруч один з одним).

Сучасні розподілювачі пам'яті не можуть задовольнити всі ці вимоги і є певною мірою компромісними рішеннями. Доведено, що для будь-якого алгоритму розподілу

пам'яті *A* можна знайти алгоритм *B,* що за відповідних умов буде кращим за продуктивністю, ніж *A*. На практиці, однак, можуть бути розроблені алгоритми, які використовують особливості поведінки реальних програм і працюють в основному добре.

Звичайні розподілювачі не можуть контролювати розмір і кількість використовуваних блоків пам'яті: цим займаються процеси користувача, а розподілювач просто відповідає на їхні запити. Крім того, розподілювач не може займатися *дефрагментацією пам'яті* (переміщенням її блоків для того щоб зробити вільну пам'ять неперервною), оскільки для нього недопустимо змінювати покажчики на пам'ять, виділену процесові користувача (цей процес має бути впевнений у тому, що адреси, які він використовує, не змінюватимуться без його відома). Якщо було ухвалене рішення про виділення блоку пам'яті, цей блок залишається на своєму місці доти, поки застосування не вивільнить його явно. Розподілювач фактично має справу тільки із вільною пам'яттю, основне рішення, яке він має приймати, – де виділити наступний потрібний блок.

Динамічний розподіл пам'яті може спричиняти зовнішню та внутрішню фрагментацію. Саме ступінь фрагментації є основним критерієм оцінки розподілювачів пам'яті.

Фрагментація виникає з двох причин.

- Різні об'єкти мають різний час життя (якщо об'єкти, що перебувають у пам'яті поруч, вивільняються в різний час). Розподілювачі пам'яті можуть використати цю закономірність, виділяючи пам'ять так, щоб об'єкти, які можуть бути знищені одночасно, розташовувалися поруч.
- Різні об'єкти мають різний розмір. Якби всі запити вимагали виділення пам'яті одного розміру, зовнішньої фрагментації не було б (цього ефекту домагаються, наприклад, при використанні сторінкової організації пам'яті). Для боротьби із внутрішньою фрагментацією більшість розподілювачів пам'яті розділяють блоки пам'яті на менші частини, виділяють одну з них і далі розглядають інші частини як вільні. Крім того, часто використовують злиття сусідніх вільних блоків, щоб задовольняти запити на блоки більшого розміру.

Структури даних розподілювачів пам'яті

Найпростіший підхід до динамічного розподілу пам'яті реалізований у системному виклику `brk()`. Він може бути виконаний надзвичайно ефективно, тому що для його реалізації достатньо зберегти покажчик на початок вільної динамічної ділянки пам'яті та збільшувати його після кожної операції виділення пам'яті (рис. 7.12).

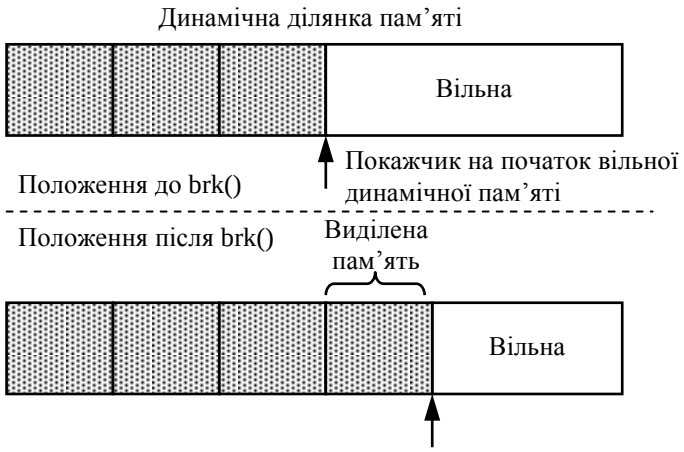


Рис. 7.12. Зміна розміру динамічної ділянки пам’яті

На жаль, реалізувати вивільнення пам'яті на основі такого виклику неможливо, оскільки воно не може бути виконане послідовно в порядку, зворотному до виділення

пам'яті. У результаті після вивільнення пам'яті в ній з'являтимуться «діри» (зовнішня фрагментація). Покажчик, який використовується `brk()`, є найпростішим прикладом структури даних розподілювача пам'яті.

Послідовний пошук підходящого блоку

Розглянемо конкретні підходи до реалізації динамічного розподілу пам'яті. Спочатку зупинимося на алгоритмах, які зводяться до послідовного перегляду вільних блоків системи і вибору одного з них. Такі алгоритми відомі досить давно і докладно описані в літературі. До цієї групи належать алгоритми *найкращого підходящого* (best fit), *першого підходящого* (first fit), *наступного підходящого* (next fit) і деякі інші, близькі до них. Ця група алгоритмів дістала назву алгоритми послідовного пошуку підходящого блоку (sequential fits).

Алгоритм найкращого підходящого зводиться до виділення пам'яті із вільного блоку, розмір якого найближчий до необхідного обсягу пам'яті. Після такого виділення у пам'яті залишатимуться найменші вільні фрагменти. Структури даних вільних блоків у цьому випадку можуть об'єднуватися у список, кожний елемент якого містить розмір блоку і покажчик на наступний блок. Під час вивільнення пам'яті має сенс поєднувати суміжні вільні блоки.

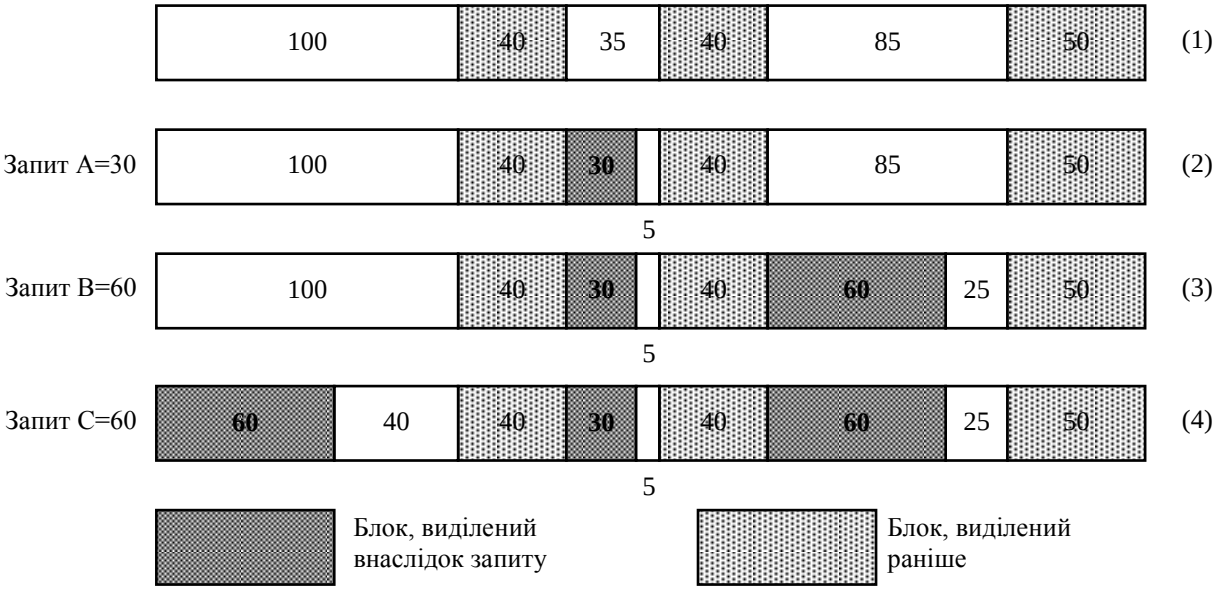
Рис. 7.13. Алгоритм найкращого підходящого

Основною особливістю розподілу пам'яті відповідно до цього алгоритму є те, що при цьому залишаються вільні блоки пам'яті малого розміру (їх називають «ошурками», sawdust), які погано розподіляються далі. Однак, практичне використання цього алгоритму свідчить, що цей недолік суттєво не впливає на його продуктивність.

Алгоритм першого підходящого полягає в тому, що вибирають перший блок, що підходить за розміром (рис. 7.14). Структури даних для цього алгоритму можуть бути різними: стек (LIFO), черга (FIFO), список, відсортований за адресою. Алгоритм зводиться до сканування списку і вибору першого підходящого блоку. Якщо блок значно більший за розміром, ніж потрібно, він може бути розділений на кілька блоків.

Різні модифікації цього алгоритму на практиці виявляють себе по-різному.

Так, алгоритм першого підходящого, який використовує стек, зводиться до того, що вивільнений блок поміщають на початок списку (вершину стека). Такий підхід простий у реалізації, але може спричинити значну фрагментацію. Прикладом такої фрагментації є



ситуація, коли одночасно виділяють більші блоки пам'яті на короткий час і малі блоки –

на довгий. У цьому разі вивільнений великий блок буде, швидше за все, відразу використаний для виділення пам'яті відповідно до запиту на малий обсяг (і після цього не вивільниться найближчим часом).

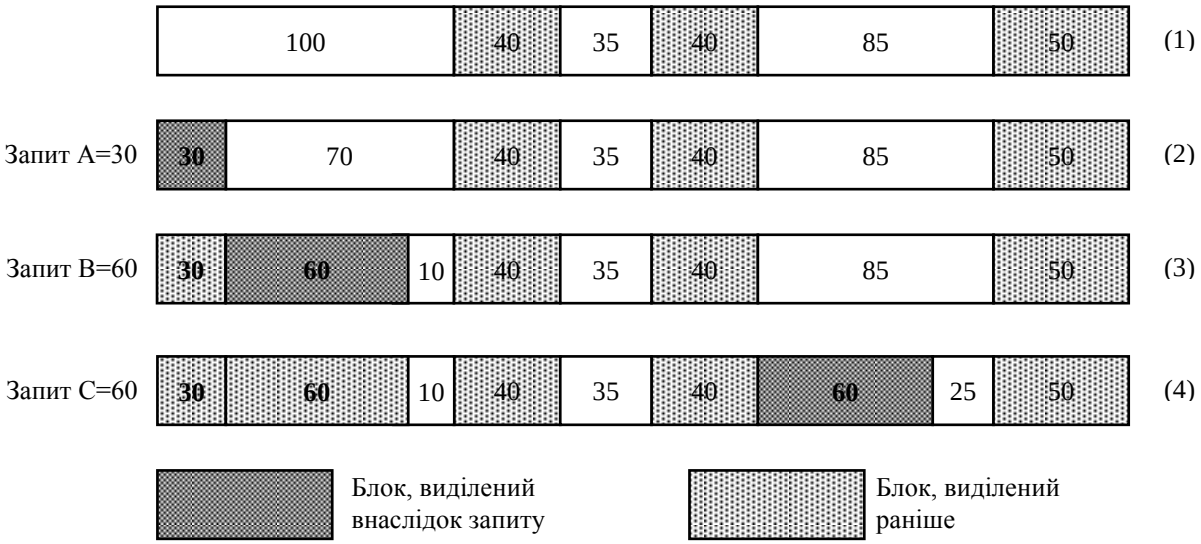


Рис. 7 14. Алгоритм першого підходящого

З іншого боку, така проблема не виникає, якщо список вільних блоків організовують за принципом FIFO (вільні блоки додають у кінець цього списку) або впорядковують за адресою. Ці модифікації алгоритму першого підходящого використовують найчастіше.

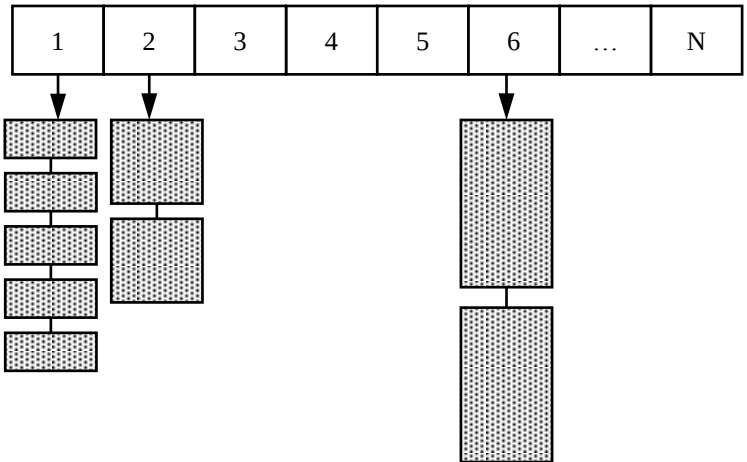
З погляду фрагментації алгоритми найкращого підходящого та першого підходящого практично рівноцінні (фрагментацію за умов реального навантаження підтримують приблизно на рівні 20 %). Такий результат може здатися досить несподіваним, оскільки в основі цих алгоритмів лежать різні принципи. Можливе пояснення полягає в тому, що у разі використання алгоритму першого підходящого список вільних блоків згодом стає впорядкованим за розміром (блоки меншого розміру накопичуються на початку списку), тому для невеликих об'єктів (а саме такі об'єкти здебільшого виділяє розподілювач) він працює майже так само, як і алгоритм найкращого підходящого.

Алгоритм першого підхожого звичайно працює швидше, оскільки пошук найкращого підходящого вимагає перегляду всього списку вільних блоків.

Головним недоліком алгоритмів послідовного пошуку вільних блоків є їхня недостатня масштабованість у разі збільшення обсягу пам'яті. Що більше пам'яті, то довшими стають списки, внаслідок чого зростає час їхнього перегляду.

Ізольовані списки вільних блоків

Інший важливий клас алгоритмів динамічного розподілу пам'яті зводиться до організації списків вільних блоків у структуру даних, що містить масив розмірів блоків, причому кожен елемент масиву пов'язаний зі списком описувачів вільних блоків (рис. 7.15). Пошук потрібного блоку зводиться до пошуку потрібного розміру в масиві та вибору елемента із



відповідного списку. Таку технологію називають технологією *ізольованих списків вільних блоків* (segregated free lists) або *ізольованою пам'яттю* (segregated storage).

Рис. 7.15. Ізольовані списки вільних блоків

Розрізняють два базові підходи до організації такого виду списків: *проста ізольована пам'ять* (simple segregated storage) та *ізольований пошук підходящого блоку* (segregated fits).

У разі використання ***простої ізольованої пам'яті*** поділу більших блоків не відбувається, щоб задовольнити запит на виділення блоку меншого розміру. Коли під час обслуговування запиту на виділення пам'яті з'ясовують, що список вільних блоків цього розміру порожній, видають запит операційній системі на розширення динамічної ділянки пам'яті (наприклад, через системний виклик `brk()`). Після цього виділені сторінки розбивають на блоки одного розміру, які й додають у відповідний список вільних блоків. У результаті сторінка пам'яті завжди містить блоки одного розміру.

Основними перевагами такого підходу є те, що немає необхідності зберігати для кожного об'єкта інформацію про його розмір. Тепер така інформація може бути збережена для цілої сторінки об'єктів, що особливо вигідно тоді, коли розмір об'єкта малий і на сторінці таких об'єктів міститься багато.

Продуктивність цього підходу досить висока, особливо через те, що об'єкти одного розміру постійно виділяють і вивільняють упродовж малого періоду часу.

Головним недоліком простої ізольованої пам'яті є доволі відчутна зовнішня фрагментація. Не робиться жодних спроб змінювати розмір блоків (розбивати на менші або поєднувати) для того щоб задовольняти запити на блоки іншого розміру. Наприклад, якщо програма запитуватиме блоки тільки одного розміру, система із вичерпуванням їхнього запасу розширюватиме динамічну пам'ять, незважаючи на наявність вільних блоків інших розмірів. Одним із компромісних підходів є відстеження кількості об'єктів, виділених на кожній сторінці, і вивільнення сторінки у разі відсутності об'єктів.

Ізольований пошук підходящого блоку також підтримує структуру даних, що складається із масиву списків вільних блоків, але організація пошуку виглядає інакше. Пошук у масиві відбувається за правилами для послідовного пошуку підходящого блоку. Якщо блок потрібного розміру не знайдено, алгоритм намагається знайти блок більшого розміру і розбити його на менші. Запит на розширення динамічної ділянки пам'яті виконують лише тоді, коли жоден блок більшого розміру знайти не вдалося. Розрізняють три категорії підходів ізольованого пошуку підхожого блоку.

- Під час використання підходу точних списків підтримують масив списків для кожного можливого розміру блоку. Такий масив може бути досить великим, але достатньо зберігати тільки ті його елементи, для яких справді задані списки.
- Підхід точних класів розмірів із заокругленням зводиться до зберігання в масиві обмеженого набору розмірів (наприклад, ступенів числа 2) і пошуку найближчого підходящого. Цей підхід ефективніший, але більш схильний до внутрішньої фрагментації.
- Підхід класів розмірів зі списками діапазонів зводиться до того, що в кожному списку містяться описувачі вільних блоків, довжина яких потрапляє в деякий діапазон. Це означає, що блоки у списку можуть бути різного розміру (у межах діапазону). Для пошуку у списку використовують алгоритм найкращого підходящого або першого підходящого.

Системи двійників

Система двійників (buddy system) – підхід до динамічного розподілу пам'яті, який дає змогу рідше розбивати на частини більші блоки для виділення пам'яті під блоки меншого розміру, знижуючи цим зовнішню фрагментацію. Вона містить у собі два алгоритми: виділення та вивільнення пам'яті. Розглянемо найпростішу *бінарну* систему двійників (binary buddy system).

У разі використання цієї системи пам'ять розбивають на блоки, розмір яких є степенем числа 2: 2^K, $L \le K \le U$, де 2^L – мінімальний розмір блоку; 2^U – максимальний розмір блоку (він може бути розміром доступної пам'яті, а може бути й меншим).

Алгоритм виділення пам'яті

Опишемо принцип роботи алгоритму виділення пам'яті.

1. Коли надходить запит на виділення блоку пам'яті розміру *M,* відбувається пошук вільного блоку підходящого розміру (такого, що $2^{K-1} \le M \le 2^K$). Якщо блок такого розміру є, його виділяють.
2. У разі відсутності блоку такого розміру беруть блок розміру 2^{K+1}, ділять навпіл на два блоки розміру 2^K і перший із цих блоків виділяють; другий залишається вільним і стає *двійником* (buddy) першого. Робота алгоритму на цьому завершується.
3. За відсутності блоку розміром 2^{K+1} беруть найближчий вільний блок, більший за розміром від *M,* наприклад блок розміру 2^{K+N}. Він стає поточним блоком. Якщо немає жодного блоку, більшого за M, повертають помилку.
4. Після цього починають рекурсивний процес розподілу блоку. На кожному кроці цього процесу поточний блок ділиться навпіл, два отриманих блоки стають двійниками один одного, після цього перший із них стає поточним (і ділиться далі), а другий залишають вільним і надалі не розглядають. Для блоку розміру 2^{K+N} процес завершують через N кроків поділу отриманням двох блоків розміру 2^K. Перший із цих блоків виділяють, другий залишають вільним. Внаслідок поділу отримують N пар блоків-двійників.

Для ілюстрації цього алгоритму наведемо приклад (рис. 7.16).

1. Припустимо, що в системі є вільний блок розміром 512 Кбайт (1) і надійшов запит на виділення блоку на 100 Кбайт (блоку А).
2. Ділимо блок на два по 256 Кбайт, з них перший робимо поточним, а другий залишаємо його двійником (2). Після цього знову ділимо перший блок на два по 128 Кбайт. Це – потрібний розмір, тому виділяємо для А перший із цих блоків, а другий залишаємо його двійником (3). Тепер маємо один вільний блок на 128 Кбайт (двійник виділеного блоку) і один – на 256 Кбайт. У результаті для блоку А виділено 128 Кбайт.
3. Тепер надходить запит на виділення блоку на 30 Кбайт (блоку В). У цьому разі найближчим за розміром більшим вільним блоком буде блок на 128 Кбайт; система розділить його на два двійники по 64 Кбайт (4), а потім перший з них – на два по 32 Кбайт. Це – потрібний розмір, тому для В буде виділено перший із цих блоків, а другий залишать його двійником (5). У результаті для блоку В виділено 32 Кбайт.

4. Нарешті, надходить запит на виділення блоку на 200 Кбайт (блоку С). У цьому разі є підходящий блок на 256 Кбайт, тому його негайно виділяють (6).

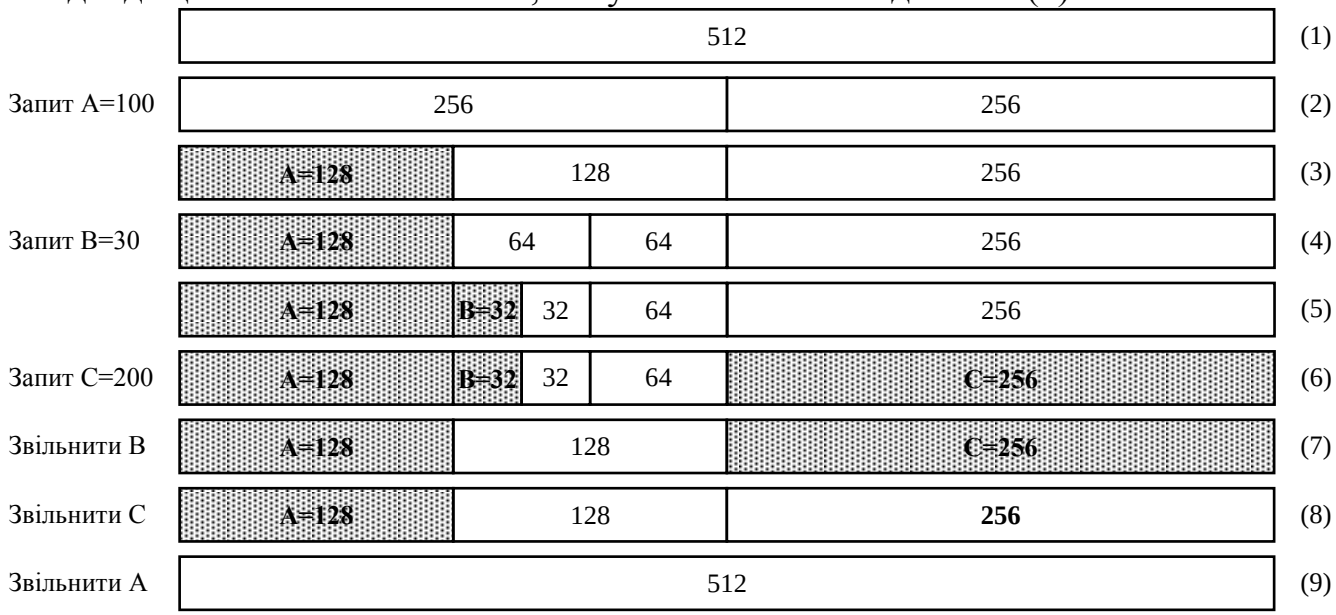


Рис. 7.16. Система двійників

Алгоритм вивільнення пам'яті

Алгоритм вивільнення пам'яті використовує утворення двійників у процесі виділення пам'яті (насправді двійниками можуть вважатися будь-які два суміжні блоки однакового розміру).

1. Заданий блок розміру 2^K вивільняють.
2. Коли цей блок має двійника і він вільний, їх об'єднують в один блок удвічі більшого розміру 2^{K+1}.
3. Якщо блок, отриманий на кроці 2, має теж двійника, їх об'єднують у блок розміру 2^{K+2}.
4. Цей процес об'єднання блоків повторюють доти, поки не буде отримано блок, для якого не знайдеться вільного двійника.

Розглянемо, як вивільнятиметься пам'ять у наведеному раніше прикладі.

1. Спочатку вивільнимо пам'ять з-під блоку B. Для нього є вільний двійник, тому їх об'єднують в один блок розміру 64 Кбайт.
2. Для цього блоку також є вільний двійник (див. (4)), і їх теж об'єднують у блок розміру 128 Кбайт (7).
3. Тепер вивільняють пам'ять з-під блоку C. Він вільних двійників не має, тому об'єднання не відбувається (8).
4. Нарешті вивільняють пам'ять з-під блоку A. Його об'єднують із двійником у блок розміру 256 Кбайт, але в того теж є вільний двійник (вивільнений з-під C), його об'єднують з тим; у результаті знову виходить один вільний блок на 512 Кбайт (9).

Цей підхід за продуктивністю випереджає інші алгоритми динамічного розподілу, він особливо ефективний для великих масивів пам'яті. Головним його недоліком є значна внутрішня фрагментація (як можна помітити, у наведеному прикладі було виділено на 86 Кбайт пам'яті більше, ніж потрібно), тому для розподілу блоків малого розміру частіше використовують інші підходи, зокрема розглянуті раніше методи послідовного пошуку або ізольованих списків вільних блоків.

Підрахунок посилань і збирання сміття

Опишемо реалізацію вивільнення пам'яті, зайнятої об'єктами. Основна проблема, що

виникає при цьому, пов'язана із необхідністю відслідковувати використання кожного об'єкта у програмі. Це потрібно для того щоб визначити, коли можна вивільнити пам'ять, яку займає об'єкт. Для реалізації такого відстеження використовують два основні підходи: *підрахунок посилань* (reference counting) і *збирання сміття* (garbage collection).

Підрахунок посилань зводиться до того, що для кожного об'єкта підтримують внутрішній лічильник, збільшуючи його щоразу при задаванні на нього посилань і зменшуючи при їх ліквідації. Коли значення лічильника дорівнює нулю, пам'ять з-під об'єкта може бути вивільнена.

Цей підхід добре працює для структур даних з ієрархічною організацією і менш пристосований до рекурсивних структур (можуть бути ситуації, коли кілька об'єктів мають посилання один на одного, а більше жоден об'єкт із ними не пов'язаний; за цієї ситуації такі об'єкти не можуть бути вивільнені)..

Тут розглянемо ***збирання сміття*** *із маркуванням і очищенням* (mark and sweep garbage collection). Така технологія дає змогу знайти всю пам'ять, яку може адресувати процес, починаючи із деяких заданих покажчиків. Усю іншу пам'ять при цьому вважають недосяжною, і вона може бути вивільнена. Алгоритм такого збирання сміття виконують у два етапи.

1. На етапі маркування виділяють спеціальні адреси пам'яті, які називають кореневими адресами. Як джерела для таких адрес звичайно використовують усі глобальні та локальні змінні процесу, які є покажчиками. Після цього всі об'єкти, які містяться у пам'яті за цими кореневими адресами, спеціальним чином позначаються. Далі аналогічно позначають всю пам'ять, яка може бути адресована покажчиками, що містяться у позначених раніше об'єктах, і т. д.
2. На етапі збирання сміття деякий фоновий процес (збирач сміття) проходить всіма об'єктами і вивільняє пам'ять з-під тих із них, які не були позначені на етапі маркування.

7.7. Реалізація динамічного керування пам'яттю в Windows

Windows реалізує два способи динамічного розподілу пам'яті для використання в режимі ядра: *системні пули пам'яті* (system memory pools) і *списки передісторії* (look-aside lists).

Розрізняють невивантажувані (nonpaged) і вивантажувані (paged) системні пули пам'яті. Обидва види пулів перебувають у системній ділянці пам'яті та відображаються в адресний простір будь-якого процесу.

- Невивантажувані містять діапазони адрес, які завжди відповідають фізичній пам'яті, тому доступ до них ніколи не спричиняє сторінкового переривання.
- Вивантажувані відповідають пам'яті, сторінки якої можуть бути вивантажені на диск.

Обидва види системних пулів можна використати для виділення блоків пам'яті довільного наперед невідомого розміру.

Списки передісторії є швидшим способом розподілу пам'яті і багато в чому схожі на кеші кускового розподілювача Linux.

Вони дають змогу виділяти пам'ять для блоків одного наперед відомого розміру. Звичайно їх спеціально створюють для виділення часто використовуваних об'єктів (наприклад, такі списки формують різні компоненти виконавчої системи для своїх структур даних). Як і для кешів кускового розподілювача, є списки загального призначення для виділення блоків заданого розміру. У разі вивільнення об'єктів вони повертаються назад у відповідний список. Якщо список довго не використовували, його

автоматично скорочують.

У Windows, як і в Linux, кожний процес має доступ до спеціальної ділянки пам'яті, її також називають ***динамічною ділянкою пам'яті*** або кучею (heap). Особливістю цієї ОС є те, що таких динамічних ділянок для процесу може бути створено кілька, і всередині кожної з них розподілювач пам'яті може окремо виділяти блоки меншого розміру.

Розподілювач пам'яті у Windows називають *менеджером динамічних ділянок пам'яті* (heap manager). Цей менеджер дає змогу процесам розподіляти пам'ять блоками довільного розміру, а не тільки кратними розміру сторінки, як під час керування регіонами.

Кожний процес запускають із динамічною ділянкою за замовчуванням, розмір якої становить 1 Мбайт (під час компонування програми цей розмір може бути змінений). Для роботи із цією ділянкою процес має спочатку отримати її дескриптор за допомогою виклику `GetProcessHeap()`.

```
HANDLE default_heap = GetProcessHeap();
```

Для виділення блоків пам'яті використовують функцію `HeapAlloc()`, для вивільнення – `HeapFree()`.

```
// виділення в ділянці heap блоку пам'яті розміру size
LPVOID addr = HeapAlloc(heap, options, size);
// вивільнення в ділянці heap блоку пам'яті за адресою addr
HeapFree(heap, options, addr);
```

Першим параметром для цих функцій є дескриптор динамічної ділянки, одним із можливих значень параметра options для `HeapAlloc()` є HEAP_ZERO_MEMORY, який свідчить, що виділена пам'ять буде ініціалізована нулями.

Для своїх потреб процес може створювати ***додаткові динамічні ділянки*** за допомогою функції `HeapCreate()` і знищувати їх за допомогою `HeapDestroy()`.

Функція `HeapCreate()` приймає три цілочислові параметри і повертає дескриптор створеної ділянки.

```
HANDLE heap = HeapCreatetoptions. initial_size, max_size);
```

Перший параметр задає режим створення ділянки (нульове значення задає режим за замовчуванням), другий – її початковий розмір, третій – максимальний. Якщо `max_size` дорівнює нулю, динамічна ділянка може збільшуватися необмежено. Задавання ділянок обмеженого розміру дає змогу уникнути помилок, пов'язаних із втратами пам'яті (втрата у кожному разі не зможе перевищити максимального розміру ділянки).

Розглянемо приклад використання додаткової динамічної ділянки пам'яті для розміщення масиву зі 100 елементів типу int.

```
int array_size = 100 * sizeof(int);
// розміщення динамічної ділянки
HANDLE heap = HeapCreate(0, array_size, array_size);
// розміщення масиву в цій ділянці, заповненого нулями
int *array=(int *)HeapAlloc(heap, HEAP_ZERO_MEMORY, array_size);
// робота із масивом у динамічній ділянці
array[20] = 1;
// знищення динамічної ділянки та вивільнення пам'яті
HeapDestroy(heap);
```

Зазначимо, що виклик `HeapFree()` у цьому прикладі не потрібний, тому що виклик `HeapDestroy()` призводить до коректного вивільнення всієї пам'яті, виділеної в `heap`. Це є однією із переваг використання окремих динамічних ділянок.

Висновки

+ Основною технологією взаємодії із диском в організації віртуальної пам'яті є завантаження сторінок на вимогу. При цьому сторінки не завантажують у пам'ять доти, поки до них не звернуться. Звертання до сторінки, не завантаженої у пам'ять, викликає сторінкове переривання, оброблювач такого переривання знаходить потрібну сторінку на диску і завантажує її у фізичну пам'ять. Така обробка займає багато часу, тому для досягнення прийнятної продуктивності кількість сторінкових переривань має бути якомога меншою. Невірна реалізація завантаження сторінок на вимогу може спричинити пробуксовування, коли процес витрачає більше часу на обмін із диском під час завантаження сторінок, ніж на корисну роботу.

+ Коли потрібна вільна сторінка, а у фізичній пам'яті місця немає, потрібно зробити заміщення сторінки. Заміщувана сторінка буде збережена на диску, а у її фрейм завантажиться нова. Є багато алгоритмів визначення заміщуваної сторінки, найефективнішим на практиці є алгоритм LRU і його наближення, зокрема годинниковий алгоритм. На продуктивність заміщення сторінок впливають й інші фактори такі, як стратегія заміщення (глобальне або локальне) і визначення робочого набору програми (множини сторінок, які вона використовує в конкретний момент). На практиці часто використовують буферизацію сторінок, коли заміщені сторінки не записують відразу на диск, а зберігають у пам'яті у спеціальних списках, що відіграють роль кеша сторінок.

+ Динамічний розподіл пам'яті - це керування ділянкою пам'яті, якою процес може розпоряджатися на свій розсуд. Засоби керування динамічною пам'яттю працюють винятково з логічними адресами і ґрунтуються на реалізації віртуальної пам'яті. Основною проблемою динамічного розподілу пам'яті є фрагментація. Для боротьби з нею використовують різні підходи.

+ Найбільш розповсюдженими технологіями динамічного розподілу пам'яті є система двійників, динамічні списки вільних блоків і послідовний пошук підходящого вільного блоку.

Контрольні запитання та завдання

1.У чому принципова відмінність обробки сторінкового переривання від обробки системного виклику? У якому випадку необхідно зберігати більший обсяг інформації?

2.У системі використовують дворівневі таблиці сторінок з асоціативною пам'яттю і підкачуванням на вимогу. Розрахуйте ефективний час доступу до пам'яті у випадку, якщо час доступу до основної пам'яті становить 100 не, відсоток влучення асоціативної пам'яті – 80 %, ймовірність сторінкового переривання – 0,001, час обробки сторінкового переривання – 10 мс. Час доступу до асоціативної пам'яті вважайте нульовим.

3.Які операції з асоціативною пам'яттю має виконати ОС під час заміщення сторінки?

4.Як впливає вибір алгоритму заміщення сторінок на вибір алгоритму планування процесів?

5.Відомо, що кількість сторінкових переривань обернено пропорційна кількості доступних фреймів пам'яті. Припустімо, що сторінкове переривання в середньому обробляють 10 мс. Програма виконується в деякій ділянці пам'яті і генерує 5000 сторінкових переривань. Загальний час виконання програми - 1 хвилина. Як зміниться загальний час виконання програми, якщо обсяг доступної пам'яті збільшити вдвічі? У скільки разів потрібно збільшити обсяг пам'яті, щоб на обробку сторінкових переривань

затрачалося стільки ж часу, що й на виконання інструкцій програми?

6.Якщо сторінка до моменту заміщення була змінена, її потрібно записати на диск. Як впливає розмір сторінки на кількість операцій записування змінених сторінок?

7.У деяких ОС під час роботи зі сторінками коду пристрою підкачування не використовують. Сторінки зчитують безпосередньо з файла програми, що виконується. У чому полягає недолік цього методу? Як можна його усунути?

8.Операція читання з диска на рівні контролера потребує задання адреси буфера фізичної пам'яті. Системний виклик зчитування з диска у деякій ОС приймає як параметр логічну адресу буфера, перевіряє її на коректність, перетворює у фізичну адресу і передає контролеру. У чому недоліки такої схеми і як можна її поліпшити?

9.Визначте кількість сторінкових переривань і кінцевий стан пам'яті для рядка посилань: 1, 2,3,4, 2,1, 5,6, 2,1,2,3, 7,6 у системі з чотирма вільними фреймами пам'яті у разі використання наступних алгоритмів:

а) FIFO;

б) оптимального;

в) LRU;

г) годинникового (без додаткових бітів).

10. Як відомо, для сторінкової організації пам'яті апаратно підтримують захист пам'яті на рівні окремих сторінок. Як програмне реалізувати задавання різних прав для ділянок пам'яті в межах однієї й тієї ж сторінки (наприклад, для першої половини сторінки – доступ для читання, для другої – для записування)?

11. Чи використовує принцип локальності доступу:

а) LRU-алгоритм заміщення сторінок;

б) FIFO-алгоритм заміщення сторінок? Поясніть Ваші відповіді.

12. Як впливає розмір сторінки на загальний розмір (у байтах) робочого набору процесу і на кількість сторінок у робочому наборі?

13.Динамічна ділянка пам'яті процесу становить один неперервний блок розміром 256 байт. Опишіть кроки, які знадобляться для виділення послідовності блоків пам'яті розміром 7, 26, 34, 19 байт за допомогою алгоритму системи двійників. Які блоки залишаться вільними?

14.Наведіть початкову конфігурацію динамічної ділянки пам'яті і послідовність виділених блоків:

а) якщо використання алгоритму першого підходящого призводитиме до меншої фрагментації, ніж алгоритм найкращого підходящого;

б) якщо використання алгоритму найкращого підходящого призводитиме до меншої фрагментації, ніж алгоритм першого підходящого.

15.Наведіть приклад коду, у якому неявно передбачено, що розподілювач пам'яті не буде займатися дефрагментацією динамічної ділянки пам'яті процесу.

3.Який вид фрагментації має місце:

а) для сегментації;

б) для сторінкової організації пам'яті;

в) у керуванні динамічною пам'яттю процесу?

16.У якому випадку потрібно надавати більшої уваги можливій фрагментації: у випадку реалізації сторінкової організації пам'яті чи у випадку реалізації динамічного розподілу пам'яті процесу? Поясніть свою відповідь.

17.Як впливає ступінь фрагментації динамічної ділянки пам'яті процесу на кількість

сторінкових переривань, що виникають під час його виконання?

18. Чим відрізняються дії, які повинні виконувати динамічні розподілювачі ядра і режиму користувача у разі нестачі пам'яті?
19. Для яких структур даних використання збирання сміття завжди приводить до коректного звільнення пам'яті, а підрахунок посилань – ні?
20. Опишіть причини, за якими системний виклик brk() повертає помилку. Чи можлива фрагментація пам'яті у разі його використання?
21. Виділіть спільні риси і відмінності в реалізації:
    а) керування адресним простором процесу;
    б) динамічного керування пам'яттю ядром;
    в) алгоритму заміщення сторінок;
    г) буферизації сторінок у Linux і Windows.

22. Сформуйте структуру даних для завдання 10 розділу 5 з використанням функцій керування динамічною областю пам'яті Win32 API. Для кожного екземпляра структури виділіть окрему динамічну ділянку пам'яті.