ЛЕКЦІЯ 8. ВЗАЄМОДІЯ З ДИСКОМ ПІД ЧАС КЕРУВАННЯ ПАМ'ЯТТЮ

План

9.1. Причини використання диска під час керування пам'яттю
9.2. Поняття підкачування
9.3. Завантаження сторінок на вимогу. Особливості підкачування сторінок
9.4. Проблеми реалізації підкачування сторінок
9.5. Заміщення сторінок
9.6. Зберігання сторінок на диску
9.7. Пробуксовування і керування резидентною множиною
9.8. Реалізація віртуальної пам'яті в Linux
9.9. Реалізація віртуальної пам'яті в Windows XP

9.1. Причини використання диска під час керування пам'яттю

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. Отже, процес або весь перебуває у пам'яті, або цілком зберігається на диску (про такий процес прийнято говорити, що він перебуває у вивантаженому стані). 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 з урахуванням 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.

9.2. 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. Як відомо, зняття вимоги неперервності фізичного простору, куди відображається адресний простір процесу**,** і можливість переміщення процесу в пам'яті під час його виконання дає змогу не тримати одночасно в основній пам'яті всі блоки пам'яті (сторінки або сегменти), які утворюють адресний простір цього процесу. Під час завантаження процесу в основну пам'ять у ній розміщують 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.

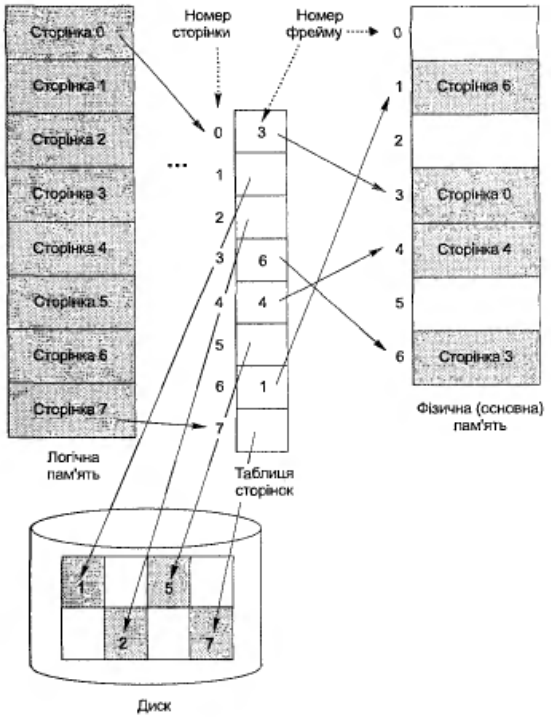

Рис. 9.1. Принцип дії підкачування

Внаслідок використання технології підкачування кількість виконуваних процесів збільшується (для кожного з них в основній пам'яті перебуватиме тільки частина блоків). Підкачування дає також змогу виконувати процеси, які за розміром більші, ніж основна пам'ять (для таких процесів у різні моменти часу в основну пам'ять відображатимуться різні блоки).

9.3. Завантаження сторінок на вимогу. Особливості підкачування сторінок

Базовий підхід, який використовують під час реалізації підкачування сторінок із диска, називають технологією *завантаження сторінок на вимогу* (demand paging).

Ця технологія діє у припущенні, що не всі сторінки процесу мають завантажуватися у пам'ять негайно. Завантажуються тільки ті, що необхідні для початку його роботи, інші — коли стають потрібні.

9.3.1. Апаратна підтримка підкачування сторінок

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. Якщо біт присутності сторінки дорівнює нулю, це означає, що дана сторінка перебуває на диску і має бути завантажена в основну пам'ять перед використанням.

Для сторінки може бути заданий *біт модифікації M.* Його спочатку (під час завантаження сторінки із диска) покладають рівним нулю, 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.

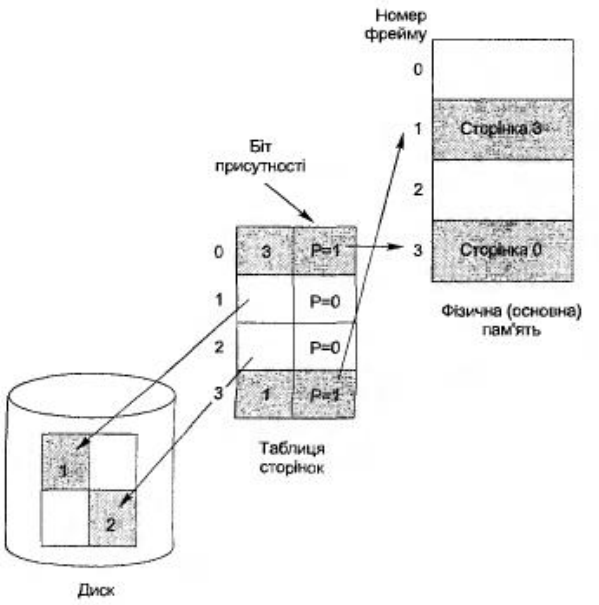

Рис. 9.2. Біт присутності сторінки

Якщо процес працює тільки зі сторінками, для яких **біт *P* дорівнює одиниці** (його резидента множина не змінюється), складається враження, що всі його сторінки є у пам'яті, хоча насправді це може бути і не так (просто сторінок, яких немає у пам'яті, у цьому разі взагалі не використовують).

9.3.2. Поняття сторінкового переривання

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.

9.4. Проблеми реалізації підкачування сторінок

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, 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.*

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.

9.5. Заміщення сторінок

Під час генерації сторінкового переривання може виникнути ситуація, коли жодного вільного фрейму у фізичній пам'яті немає. Причини появи цієї проблеми полягають в особливостях використання завантаження сторінок на вимогу.

Припустимо, що в нашій системі кожен процес у конкретний момент часу в середньому звертається тільки до половини своїх сторінок. Завантаження сторінок на вимогу залишає половину сторінок кожного процесу на диску, вивільняючи половину фреймів фізичної пам'яті, які міг би зайняти цей процес, для виконання інших процесів. Тому можна завантажити у пам'ять більше процесів.

Нехай основна пам'ять містить 60 фреймів. Якщо не використовувати завантаження на вимогу, в основну пам'ять можна завантажити шість процесів, кожен із яких використає 10 сторінок, причому кожній сторінці буде відповідати певний фрейм. У разі використання завантаження на вимогу ці шість процесів у конкретний момент часу використовуватимуть тільки 30 фреймів пам'яті, а інші 30 залишатимуться вільними, тому можна завантажити в них ще до шести процесів (у сумі — до дванадцяти).

Припустимо, що у пам'ять завантажено одночасно вісім процесів; у цьому разі процесор використовуватиметься активніше, ніж для шести запущених процесів, крім того, вивільниться 20 фреймів. Однак загальний обсяг адресного простору завантажених процесів перевищуватиме обсяг фізичної пам'яті.

Поки процеси використовують свої сторінки звичайним чином (на 50 %), всі вони будуть виконуватися в основній пам'яті без проблем. Але завжди може виникнути ситуація, коли процеси зажадають більше пам'яті, ніж їм це потрібно в середньому. Згадані вісім процесів можуть у якийсь момент зажадати весь свій адресний простір, тому всього їм знадобиться 80 фреймів основної пам'яті (а їх тільки 60). Що більше процесів одночасно виконується в системі, то вища ймовірність виникнення такої ситуації.

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.). Навіть невеликі поліпшення в методах заміщення сторінок можуть спричинити великий загальний виграш у продуктивності.

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).

Знову запустити інструкцію, на якій відбулося переривання.

9.5.1. Оцінка алгоритмів заміщення сторінок

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 цьому розділі використовуватимемо такий рядок посилань:

1, 2, 3, 5, 2, 4, 2, 4, 3, 1, 4.

Щоб оцінити алгоритми заміщення, потрібно також зазначити кількість доступних фреймів. Чим вона більша, тим меншим буде рівень сторінкових переривань, тому алгоритми оцінюють для однакових її значень. Зазначимо, що, коли доступний один фрейм, кожен елемент рядка посилань викликатиме сторінкове переривання; з іншого боку, якщо кількість доступних фреймів дорівнює кількості різних сторінок у рядку посилань, сторінкових переривань не буде зовсім. Алгоритми оцінюватимемо для трьох доступних фреймів.

9.5.2. Алгоритм FIFO

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

На рис. 8.3 зображена робота цього алгоритму на рядку посилань (кольором тут і далі виділено сторінкові переривання).

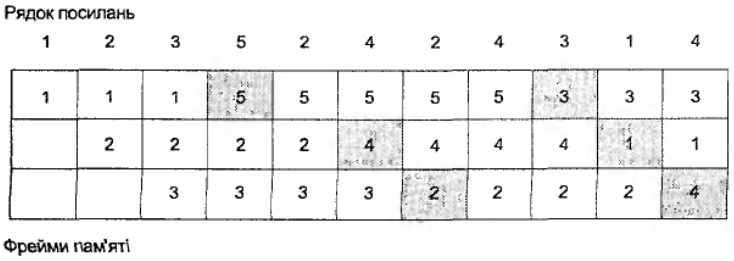

Рис. 9.3. FIFO-алгоритм заміщення сторінок

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.

9.5.3. 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. 8.4.

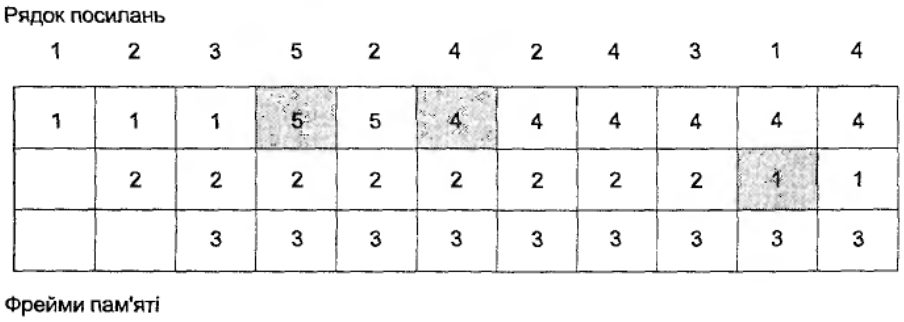

Рис. 9.4. Оптимальний алгоритм заміщення сторінок

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

9.5.4. 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 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, не використовуваної найдовше).

Формулюють його так:

замінити сторінку, що не була використана упродовж найбільшого проміжку часу.

Рис. 9.5 ілюструє роботу цього алгоритму.

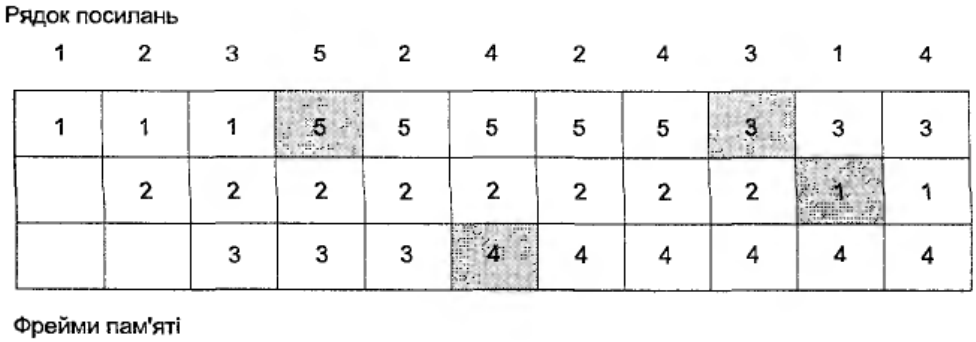

Рис. 9.5. LRU-алгоритм заміщення сторінок

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.

9.5.5. Годинниковий алгоритм

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 негайно замінюють, поміщаючи на її місце нову. Якщо ж біт *R* дорівнює 1 (до сторінки зверталися), то його покладають рівним 0 (начебто ця сторінка тільки що завантажена у пам'ять), і прохід за списком триває далі, поки не буде знайдена сторінка з *R* = 0 (а доти біт *R* для всіх сторінок покладають рівним 0). Знайдену сторінку замінюють, після чого для нової сторінки задають *R* = 1 і наставляють на неї стрілку (рис. 8.6). У найгіршому випадку (якщо для всіх сторінок біт *R* дорівнює одиниці) почнеться друге коло обходу списку (другий шанс) і буде замінена найстарша сторінка із тих, що були пройдені на першому колі. У цьому разі алгоритм зводиться до алгоритму FIFO

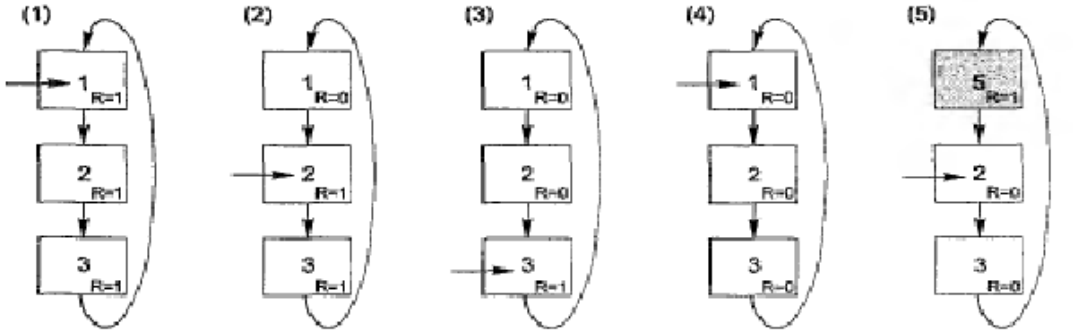

Рис. 9.6. Годинниковий алгоритм заміщення сторінок

На рис. 9.7 зображено результат застосування годинникового алгоритму для попереднього рядка посилань; 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.

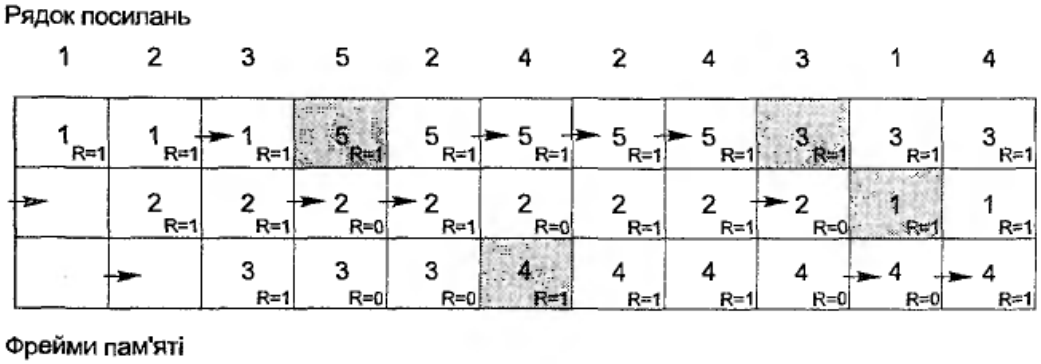

Рис. 9.7. Результат застосування годинникового алгоритму заміщення сторінок

9.5.7. Глобальне і локальне заміщення сторінок

Заміщення сторінок можна розділити на глобальне і локальне. За глобального заміщення процес може вибрати фрейм, що належить будь-якому процесові, і завантажити 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, маловикористовувані ділянки пам'яті процесу виявляються втраченими для інших процесів.

Зазвичай загальна продуктивність для глобального заміщення виявляється вищою. Глобальне заміщений також є простішим у реалізації.

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.

9.5.8. Блокування сторінок у пам'яті

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.

1. ОС виявляє, що вільних фреймів немає, тому застосовує алгоритм заміщення для пошуку сторінки, яку потрібно вивантажити на диск.

2. Внаслідок пошуку алгоритм вибирає для заміщення сторінку, де розташований буфер введення процесу А.

3. ОС заміщає сторінку, зберігаючи її дані на диску і завантажуючи у фрейм нову сторінку.

4. ОС перемикає контекст і передає керування знову процесу А.

5.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. Приміром, 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).

Зазначимо, що іншим способом розв'язання проблеми вивантаження сторінки під час введення може бути заборона на копіювання даних контролером безпосередньо у пам'ять користувача. У цьому випадку ОС може помістити свій буфер у невивантажувану пам'ять, ввести туди дані, а потім скопіювати їх у буфер користувача

9.5.9. Фонове заміщення 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.

9.6. 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. Можна також під час першого завантаження на вимогу звертатися до файла із файлової системи та зчитувати дані з нього, а в разі заміщення сторінок зберігати їх у просторі підкачування. Тому там зберігатимуться тільки заміщені сторінки, і не кожній сторінці у пам'яті відповідатиме сторінка на диску, тобто образ процесу на диску не зберігатиметься.

9.7. Пробуксовування і керування резидентною множиною

9.7.1. 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. Отож, на додачу до пам'яті, за розміром порівнянної із диском, отримують пам'ять, порівнянну із диском і за часом доступу.

Назвемо деякі причини пробуксовування.

1. Процес не використовує пам'ять повторно.

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 порада, 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.

9.7.2. Практичні аспекти боротьби з пробуксовуванням

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.

9.8. Реалізація віртуальної пам'яті в Linux

9.8.1. Керування адресним простором процесу

Структура адресного простору процесу

Адресний простір процесу складається з усіх лінійних адрес, які йому дозволено використовувати. Ядро може динамічно змінювати адресний простір процесу шляхом додавання або вилучення інтервалів лінійних адрес.

Інтервали лінійних адрес зображуються спеціальними структурами даних — *регіонами пам'яті* (memory regions). Кожний регіон описують *дескриптором регіону*, що містить його початкову лінійну адресу, першу адресу після його кінцевої, прапорці прав доступу (читання, запис, виконання, заборона вивантаження на диск тощо). Розмір регіону має бути кратним 4 Кбайт, щоб його дані заповнювали всі призначені фрейми пам'яті. Регіони пам'яті процесу ніколи не перекриваються, ядро намагається з'єднувати сусідні регіони у більший регіон.

Усю інформацію про адресний простір процесу описують спеціальною структурою даних — *дескриптором пам'яті* (memory descriptor, mmstruct). Покажчик на дескриптор пам'яті процесу зберігають у керуючому блоці процесу. У цьому дескрипторі зберігають таку інформацію, як кількість регіонів пам'яті, покажчик на глобальний каталог сторінок, адреси різних ділянок пам'яті (коду, даних, динамічної ділянки, стека).

Крім того, у дескрипторі пам'яті процесу є покажчики на дві структури даних, призначених для забезпечення доступу до регіонів його пам'яті. Перша з них — однозв'язний список усіх регіонів процесу, який використовують для прискорення сканування всього адресного простору, друга — спеціальне бінарне дерево пошуку (що теж об'єднує всі регіони процесу), використовуване для прискорення пошуку конкретної адреси пам'яті.

Для виділення інтервалу вдаються до такого.

1. Відшукують вільний інтервал потрібної довжини (із використанням бінарного дерева).
2. Визначають регіон, що передує цьому інтервалу, і регіон, що йде за ним (коли при цьому отримують регіон, який вже використовує цей інтервал, усе починають спочатку).
3. Роблять спробу об'єднати попередній і наступний регіони із цим інтервалом. Якщо це не вдається, у пам'яті розміщують дескриптор нового регіону. Його ініціалізують адресами початку і кінця інтервалу і додають у список і дерево регіонів процесу.
4. Повертають початкову лінійну адресу нового або об'єднаного регіону.

У разі вилучення інтервалу лінійних адрес із адресного простору процесу важливо враховувати, що такий інтервал звичайно не збігається із регіоном пам'яті, він може бути частиною регіону або охоплювати кілька регіонів. Виконують такі дії.

1. Сканують список усіх регіонів пам'яті, які належать процесу, та вилучають із цього списку всі регіони, які перетинаються із заданим інтервалом.
2. Вилучені регіони скорочують або розбивають на два (залежно від характеру перетинання регіону з інтервалом).
3. Вивільняють фізичну пам'ять і змінюють таблиці сторінок процесу для вилученого інтервалу; скорочені регіони повертають назад у список і дерево регіонів процесу.

Робота з адресним простором процесу з режиму користувача

Із використанням описаних алгоритмів виділення і вилучення інтервалів лінійних адрес реалізовано засоби роботи з адресним простором процесу з режиму користувача. До

цих засобів належать системний виклик mmap(), його головним призначенням є реалізація відображуваної пам'яті, а також можливість організувати виділення регіонів пам'яті заданого розміру.

Обробка сторінкових переривань

У разі спроби доступу до логічної адреси пам'яті, якій у конкретний момент не відповідає фізична адреса, виникає сторінкове переривання. Це може бути результатом помилки програміста, частиною механізму завантаження сторінок на вимогу або технології копіювання під час записування.

Коли переривання відбулося в режимі ядра, поточний процес негайно завершують (причиною переривання в цьому разі може бути передавання невірного параметра в системний виклик або помилка в коді ядра).

Коли переривання відбулося в режимі користувача, визначають, якому із регіонів пам'яті процесу відповідає лінійна адреса, що спричинила це переривання. За відсутності такого регіону процес завершують (відбулося звертання за невірною адресою). Крім того, процес завершують, якщо з'ясовується, що переривання викликане спробою записування в регіон, відкритий для читання. Завершення процесу здійснюють відсиланням йому сигналу SIGSEGV.

У разі незавершення процесу після всіх цих перевірок вважають, що він має право отримати нову сторінку. Для цього таблицю сторінок процесу перевіряють на наявність у ній сторінки, що відповідає адресі, яка викликала переривання.

♦ За відсутності такої сторінки, якщо вона не завантажена в жодний фрейм, ядро створює новий фрейм і завантажує в нього сторінку — так реалізують завантаження сторінок на вимогу. Якщо процес не звертався до цієї сторінки жодного разу, створюють нову, заповнену нулями, тобто необхідну сторінку завантажують із диска.

4 Коли така сторінка наявна, але позначена «тільки для читання», ядро створює новий фрейм і копіює в нього її дані — так реалізують технологію копіювання під час записування.

Реалізація завантаження сторінок на вимогу в Linux пов'язана з тим, що запити процесів на розподіл динамічної пам'яті (на відміну від запитів ядра) вважають не терміновими. Наприклад, під час завантаження процесу у пам'ять імовірність того, що йому негайно знадобляться всі сторінки із програмним кодом, мала. Беручи це до уваги, ядро намагається не завантажувати сторінки у пам'ять доти, поки вони не знадобляться.

9.8.2. Організація заміщення сторінок

Списки сторінок

Організація заміщення сторінок у Linux ґрунтується на їх буферизації. Організовують два списки сторінок: *список активних* (active_list) — містить сторінки, які використовують процеси і визначає робочий набір процесів; *список неактивних* (inactive_list) — містить сторінки, які не так важливі для процесів (не використовуються в цей момент часу). Модифікована сторінка перебуває в списку неактивних якийсь час, перш ніж її збережуть на диску.

Нові сторінки додають у початок списку неактивних сторінок. За нестачі пам'яті частину сторінок переміщують з кінця списку активних сторінок у початок списку неактивних, а потім починають вивільнення сторінок із кінця списку неактивних сторінок (рис. 9.8).

Рис. 9.8. Списки сторінок менеджера віртуальної пам'яті Linux

Обробка відображених, змінених і заблокованих сторінок

Під час обходу списку неактивних сторінок для вивільнення пам'яті може виникнути ситуація, коли цього відразу зробити не можна.

♦ Сторінка може бути відображена в адресний простір процесу або кількох процесів. Щоб вивільнити сторінку, відображення спочатку потрібно вилучити для кожного такого процесу.

♦ Сторінка може бути заблокована у пам'яті (наприклад, у ній виділено буфер, який використовують для введення даних із пристрою). Під час обходу таку сторінку пропускають і роблять спробу знайти іншу сторінку для вивільнення. Ця сторінка може бути знову перевірена у такому проході.

♦ Сторінка була модифікована, тому її спочатку треба записати на диск. Як тільки не відображену в адресний простір процесу модифіковану сторінку переміщують із початку в кінець списку неактивних сторінок, система скидає її вміст на диск.

Блокування сторінок у пам'яті

Для організації блокування сторінок у пам'яті в Linux використовують системні виклики mlock() і munlock(), прототипи яких визначені в заголовному файлі sys/mman.h. Системний виклик mlock() блокує у пам'яті набір сторінок, munlock() знімає це блокування. Обидва ці виклики приймають два параметри: адресу, з якої починається блокований набір сторінок, і розмір пам'яті, яку потрібно заблокувати або розблокувати.

9.9. Реалізація віртуальної пам'яті в Windows XP

9.9.1. Віртуальний адресний простір процесу

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.

1. 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.

2. 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. 8.9 показано, як різні частини адресного простору можуть бути пов'язані з різними типами простору підтримки.

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()

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.

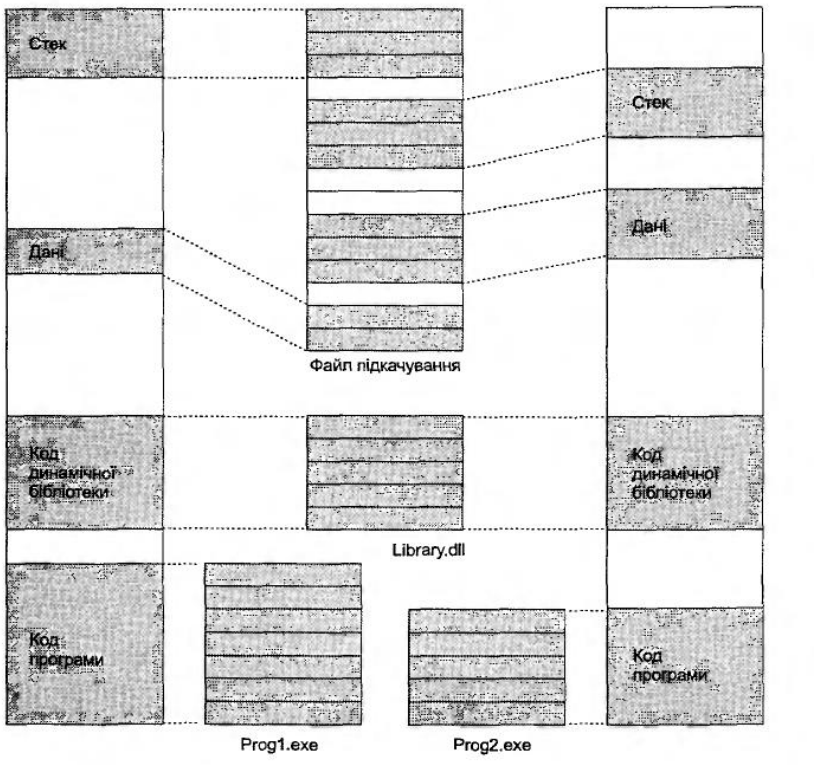

Рис. 9.9. Процеси і простір підтримки у Windows XP

Обробка сторінкових переривань

Причини виникнення сторінкових переривань у Windows XP.

♦ 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.

9.9.2. Організація заміщення сторінок

Базовий принцип реалізації *заміщення сторінок* у Windows XP — 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 XP. 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

Попереднє завантаження сторінок

У Windows XP 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 XP відслідковує сторінкові переривання упродовж 10 с 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 XP.

Висновки

1. 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.

2. 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.

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. Які операції з асоціативною пам'яттю має виконати ОС під час заміщення сторінки?
3. Як впливає вибір алгоритму заміщення сторінок на вибір алгоритму планування процесів?
4. 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).
5. Що означає поняття "резидентна множина" і де воно використовується?
6. Особливості підкачування сторінок при завантаженні на вимогу
7. Назвіть проблеми реалізації підкачування сторінок
8. Опишіть алгоритм FIFO. Коли його краще застосовувати?
9. Опишіть годинниковий алгоритм завантаження сторінок.
10. В яких випадках може виникнути пробуксовування при підкачуванні сторінок?

Література до лекції 9 .

1. Шеховцов В. А. Операційні системи. Підручник для ВНЗ. – К.: ВНV, 2008. –576 с. Розділ 9.
2. Таненбаум Э. Операционные системы– СПб: Питер. – 2002 г. – 1040 с.

ДИНАМІЧНИЙ РОЗПОДІЛ ПАМ'ЯТІ

План

10.1. Поняття динамічної пам'яті та її розподіл
10.2. Послідовний пошук підходящого блоку
10.3. Ізольовані списки вільних блоків
10.4. Системи двійників
10.5. Підрахунок посилань та збирання сміття
10.6. Реалізація динамічного керування пам'яттю в Linux
10.7. Реалізація динамічного керування пам'яттю в Windows XP

10.1. Поняття динамічної пам'яті та її розподіл

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() бібліотеки мови С, оператори new і delete в С++, оператор new в С#. Усі ці засоби спираються на підтримку з боку операційної системи.

Динамічна ділянка пам'яті процесу (купа) відображає спеціальну частину його адресного простору, у якій відбувається розподіл пам'яті за запитами з його коду. Такий розподіл пам'яті відбувається в режимі користувача, звичайно в коді системних бібліотек.

Розподілювач пам'яті (memory allocator) — частина системної бібліотеки або ядра системи, відповідальна 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).

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.

10.1.1. Фрагментація у разі динамічного розподілу пам'яті

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:

1) 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.

2) 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.

10.1.2. 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. 10.1). 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.

Складніші розподілювачі потребують складніших структур даних. Відстеження вільного простору у пам'яті може відбуватися за допомогою бітових карт, де кожний біт відповідає ділянці пам'яті (якщо він дорівнює одиниці, ця ділянка пам'яті зайнята, якщо нулю — вільна), або зв'язних списків зайнятих і вільних ділянок пам'яті, де кожний елемент списку містить індикатор зайнятості ділянки пам'яті та довжину цієї ділянки.

10.2. 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. 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).*

10.2.1. Алгоритм найкращого підходящого

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. 10.1).

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

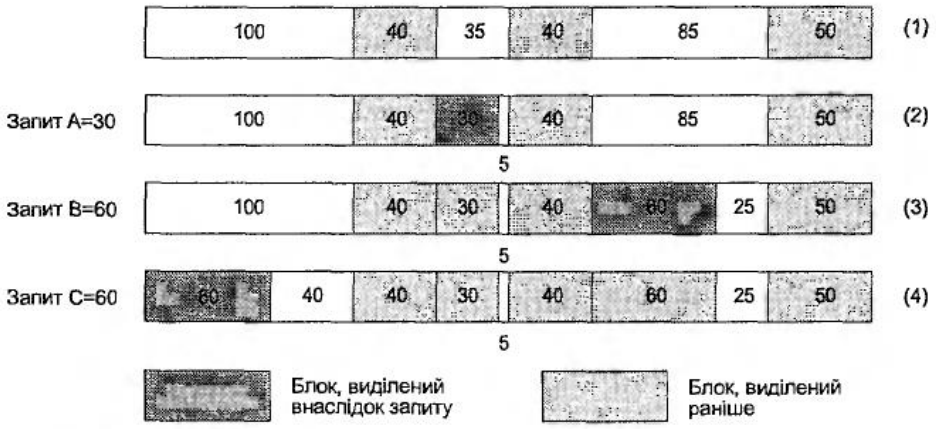

Рис. 10.1. Алгоритм найкращого підходящого

10.2.2. Алгоритм першого підходящого

Алгоритм першого підходящого полягає в тому, що вибирають перший блок, що підходить за розміром (рис. 10.2). 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.

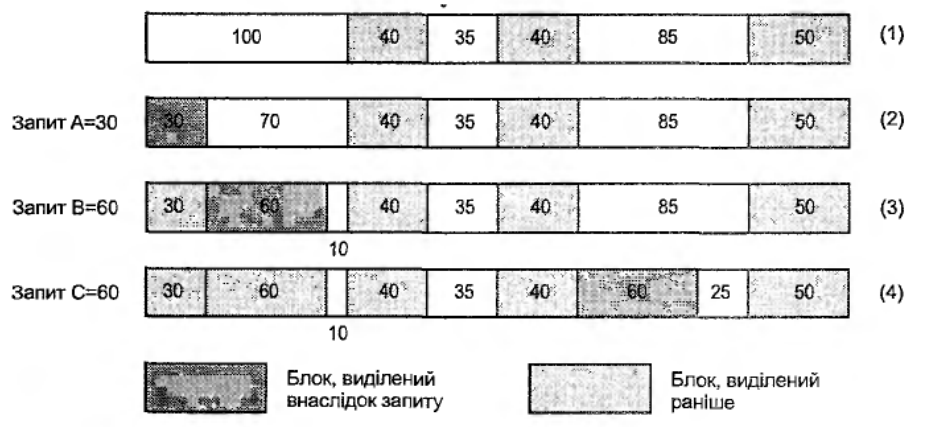

Рис. 10.2. Алгоритм першого підходящого

10.2.3. Порівняння алгоритмів послідовного пошуку підходящого блоку

З погляду фрагментації алгоритми найкращого підходящого та першого підходящого практично рівноцінні (фрагментацію за умов реального навантаження підтримують приблизно на рівні 20 %).

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.

10.3. 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. 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).

10.4. 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, $L < K < U$, де 2^i — 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 $2^{K\sim 1} < M < 2^K$). Якщо блок такого розміру є, його виділяють.
2. У разі відсутності блоку такого розміру беруть блок розміру $2^{K\sim\wedge 1}$, 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 більшим. Внаслідок поділу отримують N пар блоків-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).

Заданий блок розміру 2^K вивільняють.

Коли цей блок має двійника і він вільний, їх об'єднують в один блок удвічі більшого розміру 2^{K+1}.

Якщо блок, отриманний на кроці 2, має теж двійника, їх об'єднують у блок роз-міру 2^{K+2}.

Цей процес об'єднання блоків повторюють доти, поки не буде отримано блок, для якого не знайдеться вільного двійника.

Переваги і недоліки системи двійників

Цей підхід за продуктивністю випереджає інші алгоритми динамічного розподілу, він особливо ефективний для великих масивів пам'яті. Головним його недоліком є значна внутрішня фрагментація тому для розподілу блоків малого розміру частіше використовують інші підходи, зокрема розглянуті раніше методи послідовного пошуку або ізольованих списків вільних блоків.

10.5. 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).

10.5.1. Підрахунок посилань

Підрахунок посилань зводиться до того, що для кожного об'єкта підтримують внутрішній лічильник, збільшуючи його щоразу при заданні 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).

10.5.2. Збирання сміття

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.

10.6. Реалізація динамічного керування пам'яттю в Linux

У цьому розділі опишемо особливості динамічного керування пам'яттю Linux.

Спочатку зупинимося на динамічному розподілі пам'яті ядром системи Як зазначалося, деяку частину фізичної пам'яті ядро використовує для розміщення свого коду і його статичних структур даних. Цю пам'ять називають статичною пам'яттю, вона закріплена за ядром постійно, відповідні фрейми не можуть бути вивільнені. Уся інша пам'ять є динамічною і може бути розподілена на вимогу. Тут охарактеризуємо деякі методи, які використовуються ядром для реалізації такого розподілу.

10.6.1. Розподіл фізичної пам'яті ядром

Для динамічного розподілу великих блоків фізичної пам'яті (за розміром кратних фрейму) у Linux використовують систему двійників. Використання таких блоків дає змогу знизити необхідність модифікації таблиць сторінок під час роботи застосування, а це, в свою чергу, знижує ймовірність очищення кеша трансляції.

Розподіл об'єктів ядром

Алгоритм двійників не підходить для розподілу блоків пам'яті довільного розміру, оскільки мінімальний обсяг пам'яті, який він може виділити або вивільнити, становить один фрейм (4 Кбайт), що спричиняє значну внутрішню фрагментацію.

Для розподілу пам'яті під окремі об'єкти застосовують *кусковий розподілювач* (slab allocator). При його розробці намагаються організувати кешування часто використовуваних об'єктів ядра в ініціалізованому стані (оскільки більша частина часу під час розміщення об'єкта витрачається на його ініціалізацію, а не на саме виділення пам'яті), а також виділення пам'яті блоками малого розміру без внутрішньої фрагментації, властивої алгоритму системи двійників.

Кеші об'єктів та їхні види

Структура даних кускового розподілювача складається зі змінної кількості кешів об'єктів, об'єднаних у двозв'язний циклічний список, який називають ланцюжком кешів. Кожний такий кеш відповідає за розподіл об'єктів конкретного типу або конкретного розміру.

Розрізняють два види кешів об'єктів:

- Спеціалізовані кеші, які створюють різні компоненти ядра системи для зберігання об'єктів конкретного типу. Для таких кешів звичайно задають *конструктор* і *деструктор,* а також унікальне ім'я, що залежить зазвичай від типу об'єкта. Під конструктором розуміють функцію, яку викликають під час ініціалізації об'єктів цього типу, під деструктором — функцію, яку викликають під час вивільнення пам'яті з-під об'єкта. Прикладом можуть бути такі кеші, як uid_cache (кеш ідентифікаторів користувачів), vm_area_struct (кеш дескрипторів регіонів) тощо. Компоненти режиму ядра (наприклад, драйвери пристроїв) можуть створювати додаткові кеші для своїх об'єктів.
- Кеші загального призначення, які використовують для зберігання блоків пам'яті довільного призначення конкретного розміру. Є кеші для блоків розміром від 2^5 = 32 біт до 2^{17} = 13 1072 біт, їх називають *size-N (N* - розмір блоку кеша у байтах, наприклад, size-128).

Кускові блоки

Пам'ять для кеша виділяють у вигляді *кускових блоків* (slabs). Кусковий блок складається із одного або кількох неперервно розташованих фреймів пам'яті, розділених на фрагменти пам'яті (chunks) одного розміру, які містять об'єкти. Розмір фрагмента пам'яті залежить від типу кеша, для якого виділяють кусковий блок (він дорівнює розміру об'єкта, екземпляри якого потрібно розподіляти за допомогою цього кеша). Використання таких блоків для розподілу пам'яті знижує як зовнішню, так і внутрішню фрагментацію.

Під час створення кусковий блок негайно розділяють на певну кількість фрагментів, при цьому, якщо для відповідного кеша був заданий конструктор, його викликають для ініціалізації всіх виділених фрагментів, які перетворюються на ініціалізовані й готові до використання об'єкти. Після цього запит на виділення пам'яті під відповідний об'єкт може бути негайно задоволений, при цьому фактично ані виділення пам'яті, ані ініціалізації не відбувається (об'єкт просто позначають як виділений).

За запитом на вивільнення пам'яті об'єкт позначають як вивільнений, але з кускового блоку не вилучають і деструктор для нього не викликають; згодом його можна використати знову. Деструктор для всіх об'єктів блоку викликають тільки під час справжнього вивільнення пам'яті з-під кускового блоку або всього кеша. Зазначимо, що таке вивільнення відбувається тільки тоді, коли в цьому блоці не виділено жодного об'єкта, а система відчуває нестачу пам'яті.

10.6.2. Керування динамічною ділянкою пам'яті процесу

Кожний процес має доступ до спеціальної ділянки пам'яті, яку називають *динамічною ділянкою пам'яті,* або *кучею* (heap). Початкова і кінцева адреси цієї ділянки містяться в полях startbrk і brk дескриптора пам'яті процесу.

Для доступу до динамічної ділянки пам'яті використовують системний виклик brk(). Він намагається змінити її розмір, за параметр приймає нову кінцеву адресу цієї ділянки.

10.7. Реалізація динамічного керування пам'яттю в Windows XP

Windows XP 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).

10.7.1. Системні пули пам'яті ядра

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.

Висновки

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 грунтуються 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. 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?

2. 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.

3. Який вид фрагментації має місце:

а) для сегментації;

б) для сторінкової організації пам'яті;

в) у керуванні динамічною пам'яттю процесу?

4. Чим відрізняються дії, які повинні виконувати динамічні розподілювачі ядра і режиму користувача у разі нестачі пам'яті?

5. 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 XP.

6. Можливі причини фрагментації пам'яті при динамічному розподілі?

7. Алгоритми пошуку вільних блоків та їх порівняння.

8. Алгоритм виділення та вивільнення пам'яті.

9. Динамічний розподіл пам'яті в Linux?

10. Особливості розподілу пам'яті Windows XP?

Література до лекції 10 .

3. Шеховцов В. А. Операційні системи. Підручник для ВНЗ. – К.: ВНV, 2008. –576 с. Розділ 10.