01.1. Управління кількістю сторінок пам’яті виділених процесу

1. Сторінки процеса розділені на дві частини:

1) сторінки, які знаходяться в оперативній пам’яті,

2) сторінки, які знаходяться у зовнішній пам’яті.

2. При звертанні до даних, які містяться у сторінці в зовнішній пам’яті (2-га частина сторінок) виникає помилка звертання до сторінки (page fault).

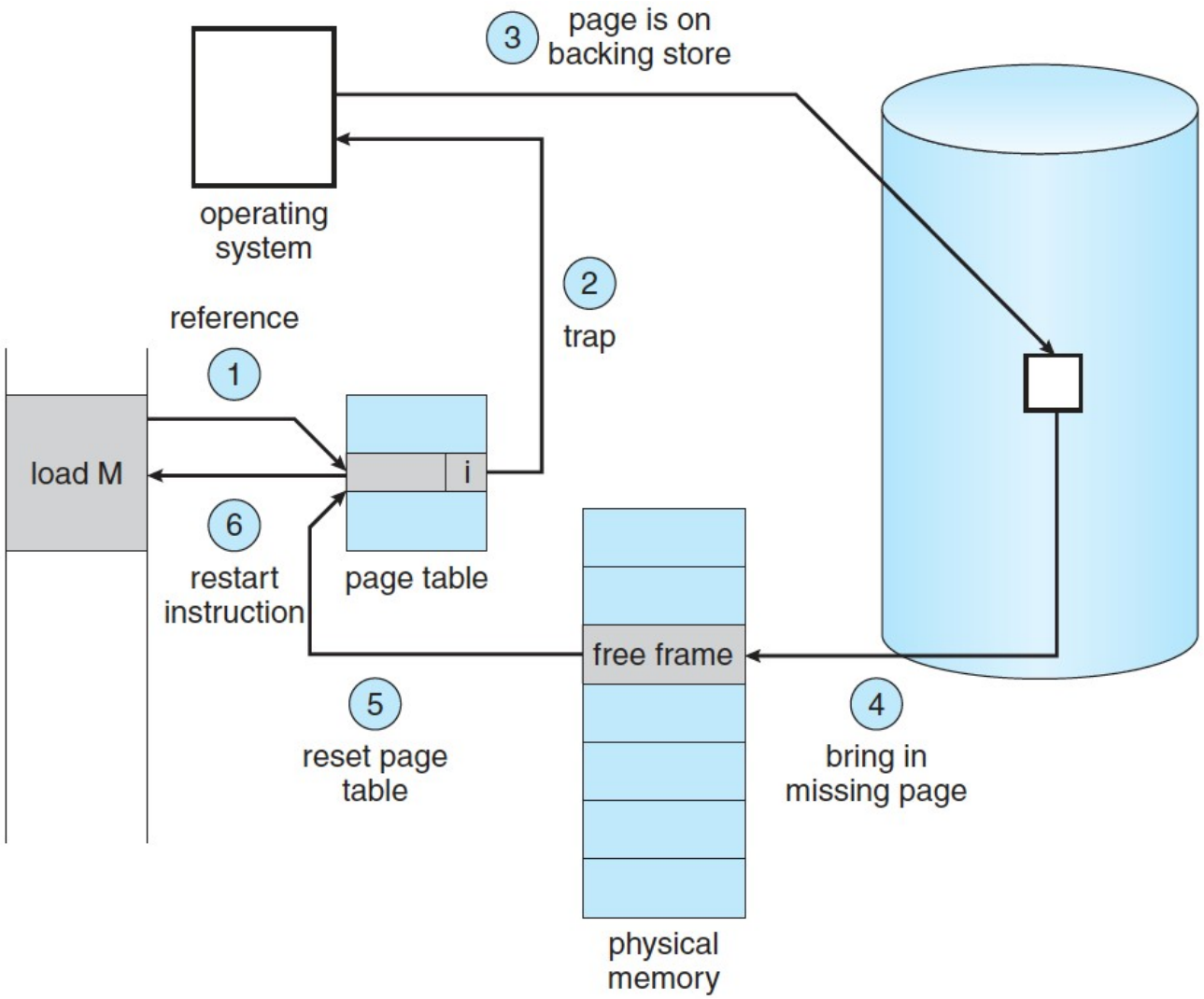
01.2. Управління кількістю сторінок пам’яті виділених процесу

3. Для «виправлення» page fault ОС підвантажує потрібну сторінку з зовнішньої пам’яті в оперативну, що призводить до затримки у роботі та відповідних витрат обчислювальних ресурсів.

4. Завдання: мінімізувати кількість page fault.

5. Дві перспективи: 1) для одного процеса, 2) для сукупності всіх процесів.

01.3. Управління кількістю сторінок пам’яті виділених процесу

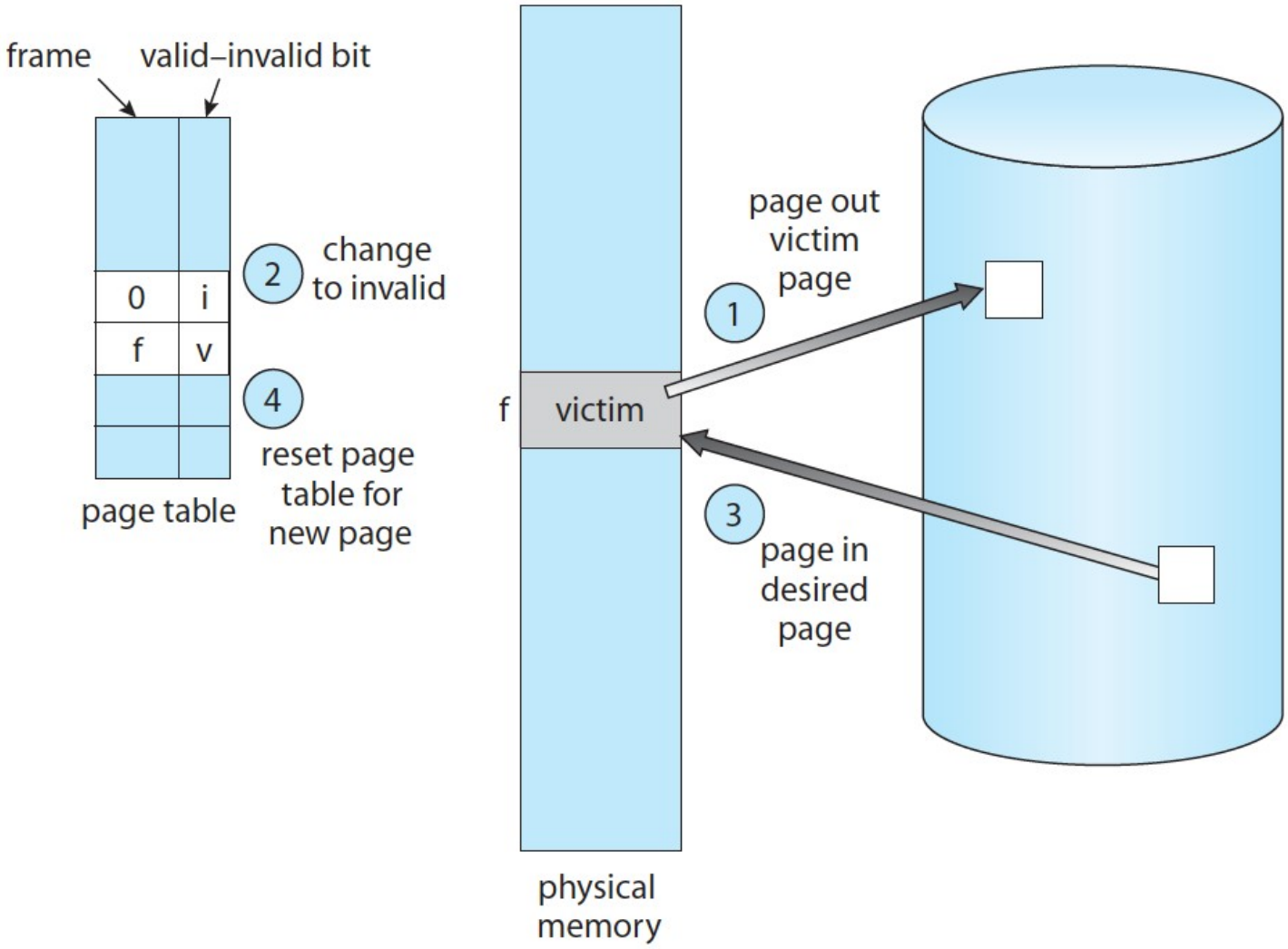


01.4. Управління кількістю сторінок пам’яті виділених процесу

- Багатозадачність: кожному процесу дозволяється тримати в оперативній пам’яті обмежену кількість сторінок (m = кількість сторінок, виділених процесу).

- У разі виникнення page fault відбувається заміна сторінки (page replacement): потрібна сторінка, яка підвантажується (swap in) з зовнішньої пам’яті, заміняє деяку сторінку даного процеса в оперативній пам’яті, яка в свою чергу вивантажується (swap out) у зовнішню пам’ять.

01.5. Управління кількістю сторінок пам’яті виділених процесу



01.6. Управління кількістю сторінок пам’яті виділених процесу

1. Мінімальна кількість сторінок, що виділяється процесу, визначається архітектурою комп’ютерної системи, виходячи з режимів адресації, які реалізовані у відповідному наборі машинних інструкцій (=кількість різних сторінок, посилання на які можуть міститись у одній інструкції).

2. Максимальна кількість сторінок, що виділяється процесу, обмежена об’ємом оперативної пам’яті.

3. З точки зору багатозадачності можна розглядати:
1) рівномірний розподіл кількості сторінок;
2) нерівномірний розподіл (наприклад, пропорційний до розміру віртуальної пам’яті процесів).

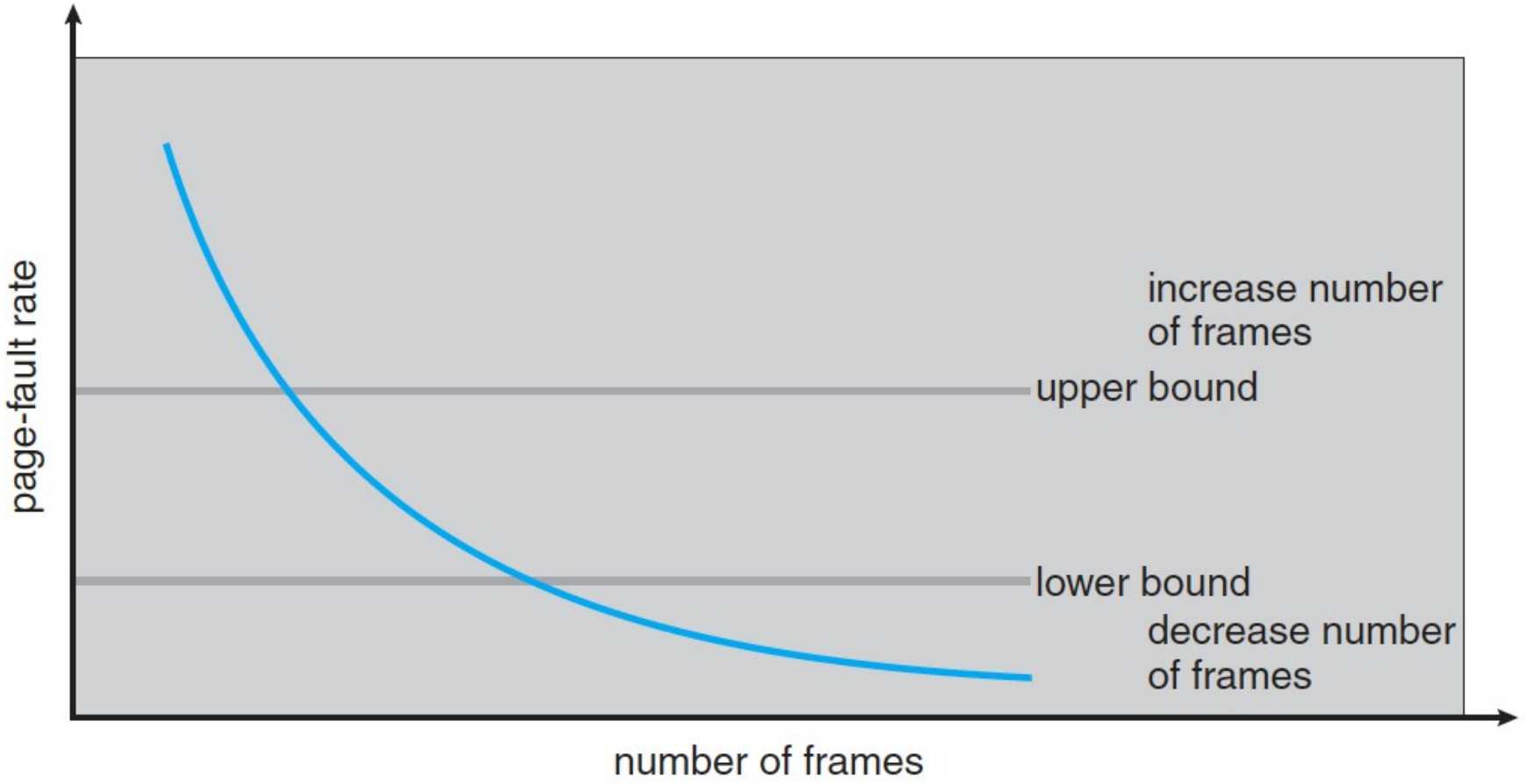
01.7. Управління кількістю сторінок пам’яті виділених процесу

1. Принцип: чим більше m, тим менше кількість помилок звертання до пам’яті.

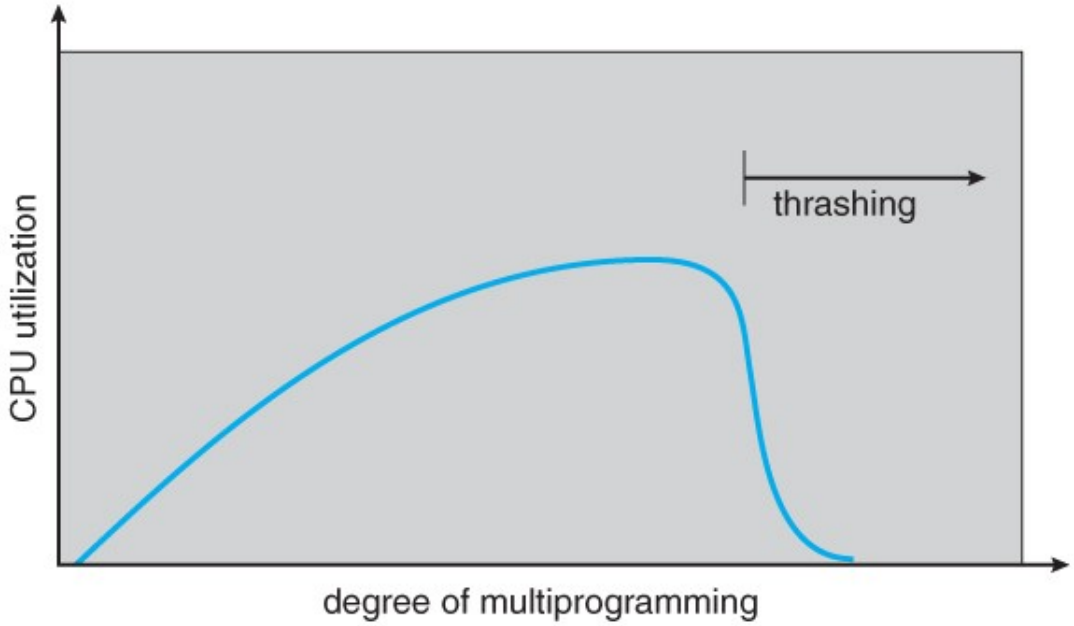
2. Порогові значення частоти помилок звертання до пам’яті: PF_{min}, PF_{max}

3. Ідея роботи алгоритмів управління кількістю сторінок:
якщо $PF(t) < PF_{min}$, то зменшити m
якщо $PF(t) > PF_{max}$, то збільшити m

01.8. Управління кількістю сторінок пам’яті виділених процесу



01.9. Управління кількістю сторінок пам’яті виділених процесу



Трешинг (thrashing) (=«пробуксовка»): процес витрачає більше часу на «виправлення» page faults, ніж на виконання обчислень.

Трешинг виникає внаслідок занадто малої кількості сторінок, виділених процесу (наприклад, внаслідок запуску великої кількості паралельних процесів).

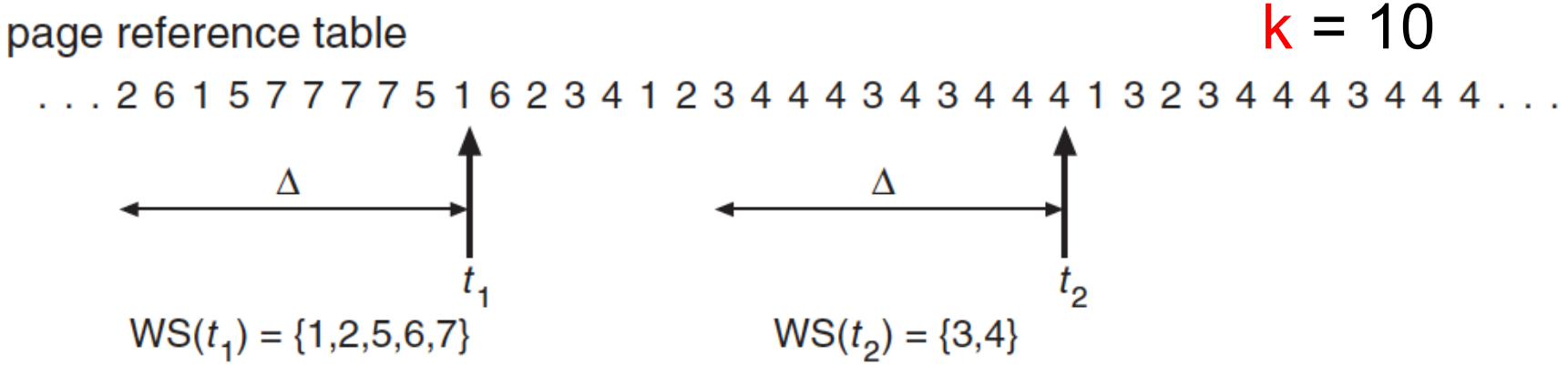
01.10. Управління кількістю сторінок пам’яті виділених процесу

1. Модель робочої множини [сторінок] (working-set model) - один з основних методів визначення кількості сторінок виділених процесу.

2. Модель робочої множини базується на принципі локальності звертань.

3. Вікно робочої множини (working-set window) — це k останніх звертань до пам’яті, для яких ми зберігаємо номери відповідних сторінок.

01.11. Управління кількістю сторінок пам’яті виділених процесу



4. Робоча множина (Working set) — це сторінки пам’яті, до яких відбулося звертарня у вікні робочої множини.

01.12. Управління кількістю сторінок пам’яті виділених процесу

5. Робочу множину можна розглядати, як наближену оцінку локальності звертань даного процеса.

6. Точність визначення робочої множини залежить від величини k:
1) якщо k занадто мале, то локальність не буде визначена в повному обсязі;
2) якщо k занадто велике, то робоча множина може «перекрити» одночасно декілька окремих локальностей.

01.13. Управління кількістю сторінок пам’яті виділених процесу

7. Кількість сторінок у робочій множині = кількість сторінок виділених процесу.

8. Проблема: вікно робочої множини весь час рухається і відповідно робоча множина змінюється → з яким інтервалом оцінювати зміну робочої множини?

9. Prepaging: збереження інформації про робочу множину процеса та її завантаження перед наступним стартом процеса (випереджаюча підкачка сторінок).

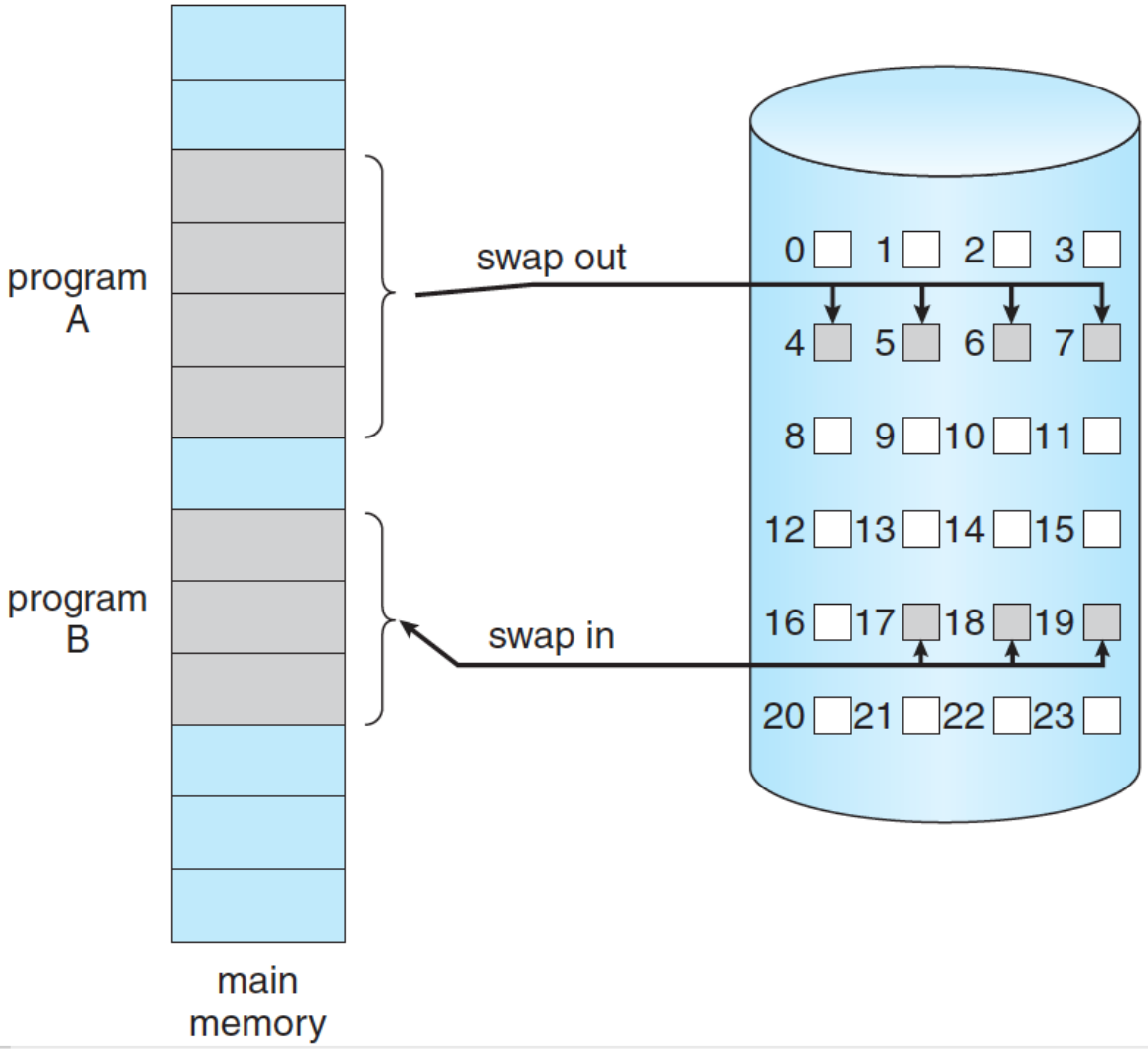
02.1. Управління заміщенням сторінок пам’яті процесу

Віртуальна пам’ять на основі сторінкової організації пам’яті → дві основні задачі:

1. Управління кількістю сторінок виділених процесу (скількі сторінок процеса в даний момент часу буде розміщено в ММ: для окремого процеса + з точки зору багатозадачності).

2. Вибір стратегії (алгоритма) заміщення сторінок процеса в ММ (свопінгу сторінок процеса між оперативною та зовнішньою пам’яттю).

02.2. Управління заміщенням сторінок пам’яті процесу



02.3. Управління заміщенням сторінок пам’яті процесу

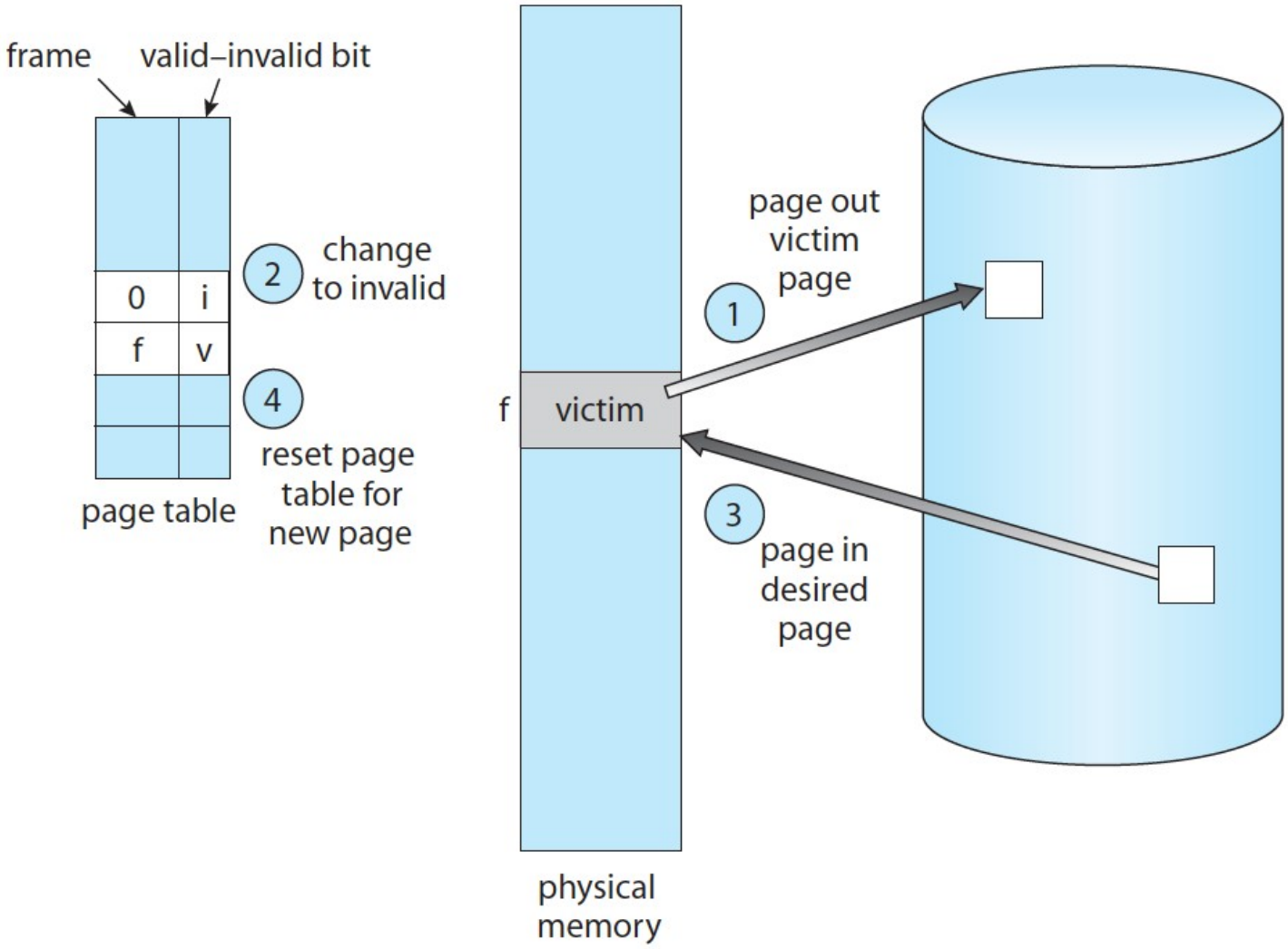
Сценарій:

1. Відбувається звертання до даних, що містяться у сторінці, яка знаходиться в зовнішній пам’яті.

2. Ця сторінка завантажується з зовнішньої пам’яті в оперативну (підкачка).

3. Для неї треба «звільнити місце» (оскільки кількість сторінок, яку процес може тримати у ММ обмежена), тобто знайти сторінку-«жертву» (victim), місце якої в ММ займе дана сторінка.

02.4. Управління заміщенням сторінок пам’яті процесу



02.5. Управління заміщенням сторінок пам’яті процесу

- Для прискорення операцій свопінгу в зовнішній пам’яті можна тримати копії всіх сторінок процеса.
- Тоді якщо сторінка-«жертва» не зазнала жодних змін (наприклад, з нею виконувались лише операції читання), то час на її запис у зовнішню пам’ять не витрачається (вона буде просто «затерта» новою сторінкою).
- Якщо ж сторінка-«жертва» була «забруднена» (в неї були внесені зміни і вона відрізняється від своєї копії у зовнішній пам’яті), то виконується її перезапис у зовнішню пам’ять.

02.6. Управління заміщенням сторінок пам’яті процесу

Алгоритми заміщення сторінок пам’яті (page replacement algorithms):

(1) FIFO Page Replacement

(2) Optimal Page Replacement

(3) LRU Page Replacement

(4) Second-Chance Algorithm

02.7. Управління заміщенням сторінок пам’яті процесу

(1) FIFO Page Replacement:

1. Для кожної сторінки фіксується час, коли вона була завантажена в ММ, або порядковий номер в послідовності завантаження сторінок в ММ (одно-направлений зв’язний список).

2. Для заміни обирається сторінка (victim), яка була завантажена в ММ ріньше за всі інші сторінки.

3. Основна перевага: простота реалізації.

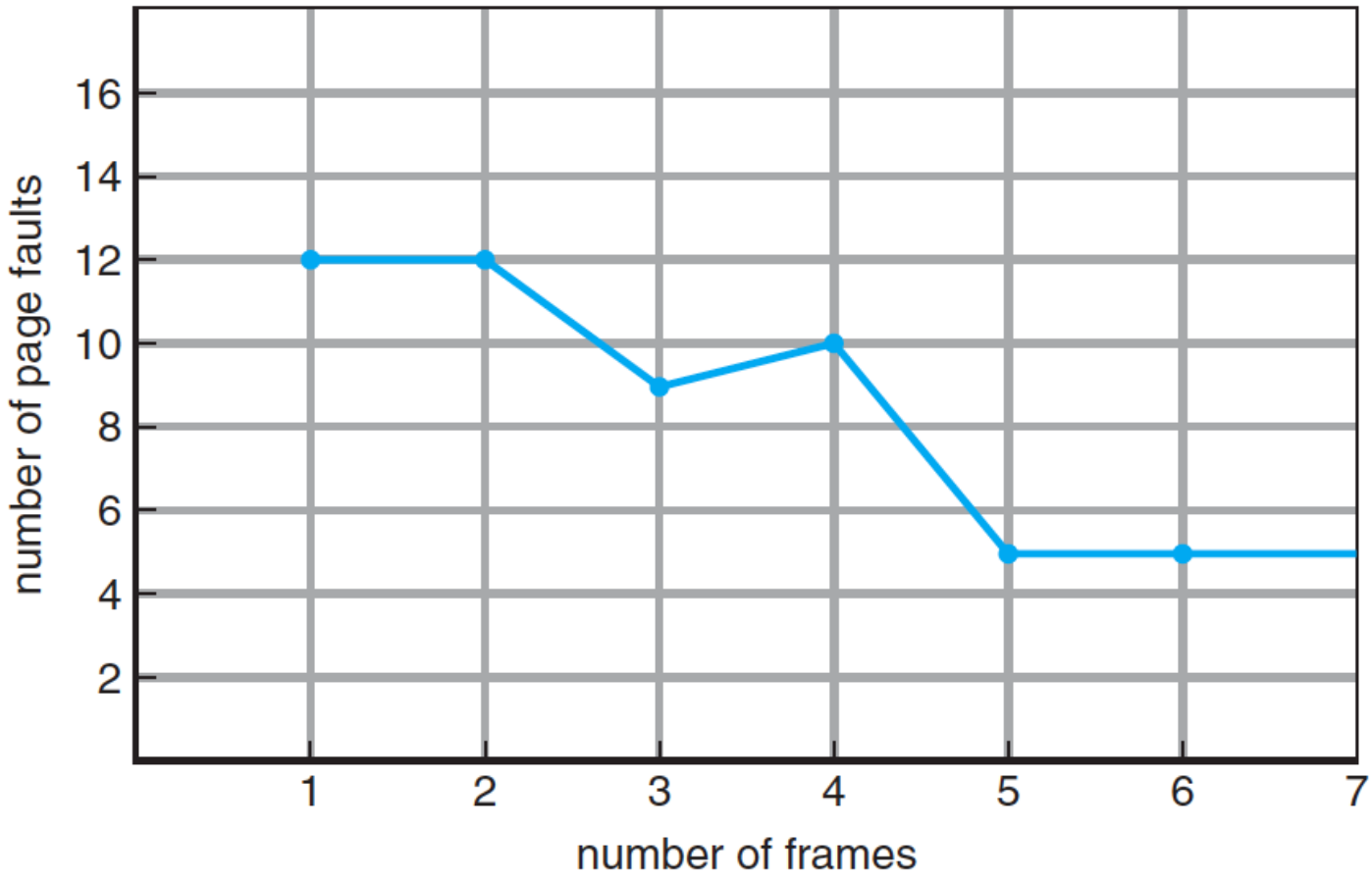
02.8. Управління заміщенням сторінок пам’яті процесу

Аномалія Беладі (Belady’s anomaly):

для деяких алгоритмів заміщення сторінок збільшення кількості сторінок процеса у ММ (number of allocated frames)

може призвести до збільшення частоти помилок звертання до сторінки (page-fault rate).

02.9. Управління заміщенням сторінок пам’яті процесу



Приклад аномалії Беладі для алгоритму FIFO

02.10. Управління заміщенням сторінок пам’яті процесу

(2) Optimal Page Replacement (OPT):

1. Завдання: знайти алгоритм, який би давав найменшу серед усіх інших алгоритмів частоту помилок звертання до сторінки (page-fault rate) та гарантував відсутність аномалії Беладі.

2. Алогоритм OPT: Для заміни обирається сторінка (victim), до якої не буде звертання на протязі найдовшого проміжку часу.

02.11. Управління заміщенням сторінок пам’яті процесу

3. Проблема: для роботи алгоритму OPT потрібно знати наперед [всю] послідовність звертань до сторінок процеса.

4. Відтак алгоритм OPT використовується лише як «взірець» для експериментальної перевірки продуктивності інших алгоритмів (на відомих тестових послідовностях звертань).

02.12. Управління заміщенням сторінок пам’яті процесу

(3) LRU (least recently used) Page Replacement:

1. Алгоритм LRU апроксимує роботу алгоритму OPT, використовуючи недавню історію звертань до сторінки для прогнозування проміжку часу, на протязі якого звертань до сторінки не буде.

2. Для кожної сторінки запам’ятовується час останнього звертання.

02.13. Управління заміщенням сторінок пам’яті процесу

3. Для заміни обирається сторінка (victim) з найдавнішим останнім звертанням (тобто сторінка, яка не використовувалася найдовший час).

4. Алгоритм LRU вважається достатньо ефективним та гарантує відсутність аномалії Беладі.

5. Проблема: робота алгоритма LRU потребує значної апаратної підтримки, оскільки його реалізація передбачає спрацювання при кожному звертанні до ММ.

02.14. Управління заміщенням сторінок пам'яті процесу

(4) Second-Chance Algorithm:

1. Для кожної сторінки зберігається біт звертання (reference bit): на початку роботи дорівнює 0, при будь якому звертанні (read/write) встановлюється в 1. Значення цього біта «дуже приблизно» апроксимує час останнього звертання (LRU).

2. Також для всіх сторінок зберігається порядковий номер завантаження в ММ (у вигляді списку, так само як в алгоритмі FIFO).

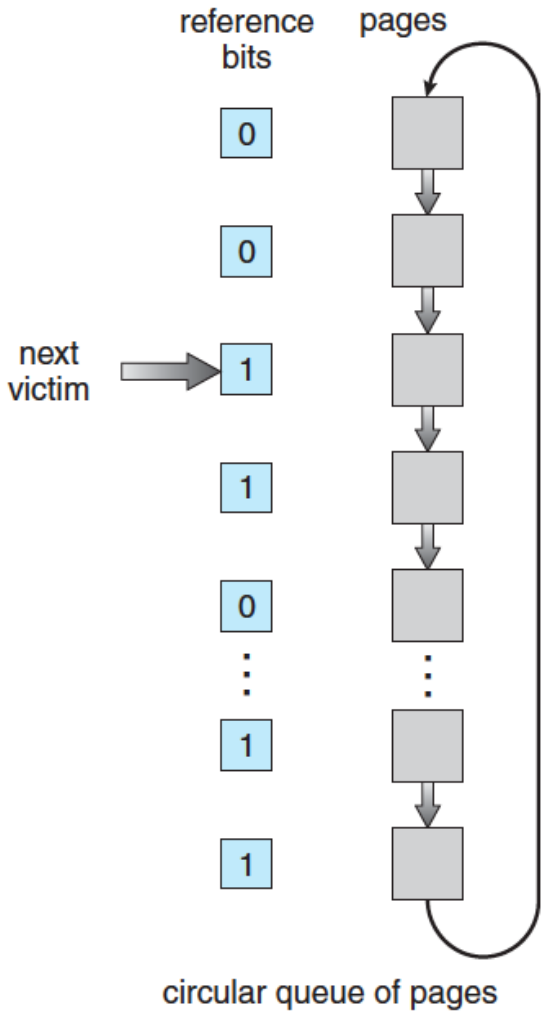
02.15. Управління заміщенням сторінок пам’яті процесу

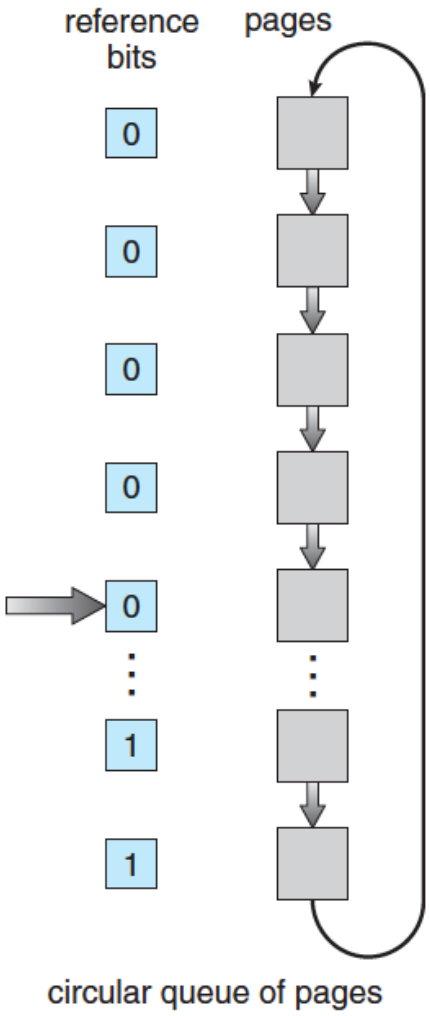
3. Для заміни обирається сторінка (victim) з найменшим порядковим номером та значенням біту звертання рівним 0.

4. Якщо біт звертання = 1, то сторінці дається «другий шанс»: вона переноситься в кінець списку (її порядковий номер стає найбільшим), а біт звертання скидається в 0.

5. Якщо біт звертання у всіх сторінок був встановлений в 1, то алгоритм зробить повне коло і повернеться до першої сторінки, якій був наданий «другий шанс», і обере її для заміни.

02.16. Управління заміщенням сторінок пам’яті процесу





Приклад реалізації Second-Chance Algorithm за допомогою кільцевого списку (clock algorithm).

03.1. Основні задачі управління пам’яттю

1. Збільшення швидкості доступу до даних з використанням ієрархії пристроїв пам’яті та принципу локальності звертань. Частковий випадок: управління заміщенням сторінок пам’яті.

2. Розподіл пам’яті між обчислювальними процесами → управління кількістю сторінок виділених процесу.

3. Збільшення зручності та виправлення помилок при роботі з пам’яттю в режимі динамічного виділення (dynamic memory allocation).

03.2. Основні задачі управління пам’яттю

В рамках 3-ої задачі розрізняють:

1) Ручне управління пам’яттю (manual memory management): програміст в явний спосіб звертається до системи з запитами про виділення та звільнення пам’яті.

2) Автоматичне управління пам’яттю (garbage collection): програміст лише створює потрібні структури даних; операції виділення відповідних обсягів пам’яті та їх звільнення виконуються в автоматичному режимі без участі програміста.

03.3. Основні задачі управління пам’яттю

Проблеми, які виникають при ручному управлінні пам’яттю:

1. wild pointer → звертання до пам’яті перед її виділенням викликом malloc().

2. dangling pointer («висячий» покажчик) → звертання до пам’яті після її звільнення викликом free().

03.4. Основні задачі управління пам’яттю

3. double free → проблема повторного звільнення вже звільненої пам’яті, яке призводить до втрат потрібних даних.

4. memory leaks («витік пам’яті») → поступове засмічення пам’яті вчасно невидаленими даними.

04.1. Механізм автоматичного управління пам’яттю

1. У випадку автоматичного управління пам’яттю використовується спеціальний програмний модуль, який бере на себе відповідні функції (цей модуль, як правило, називають Garbage Collector (GC)).

2. Відтак управління пам’яттю стає «прозорим» (прихованим від програміста).

3. Даний підхід до управління пам’яттю виник та розвивається в рамках об’єктно-орієнтованого підходу до програмування.

4. В якості основного елемента виділення та звільнення пам’яті розглядається примірник об’єкта в пам’яті об’єктів.

04.2. Механізм автоматичного управління пам’яттю

Автоматизація операцій:

1. Виділення пам’яті відбувається автоматично при оголошенні (або першому звертанні) до примірника об’єкта:

- реалізація відносно проста,
- проблема: де розмістити новий об’єкт в пам’яті об’єктів (дефрагментація).

04.3. Механізм автоматичного управління пам’яттю

2. Звільнення пам’яті (garbage collection) виконується за таким алгоритмом:

2.1. Визначити об’єкти, які не потрібні для подальшої роботи (=garbage).

2.2. Звільнити ресурси (пам’ять), що використовувались цими об’єктами.

2.3. Виконати перерозподіл ресурсів між потрібними об’єктами (оптимізація).

04.4. Механізм автоматичного управління пам’яттю

Режими запуску garbage collection:

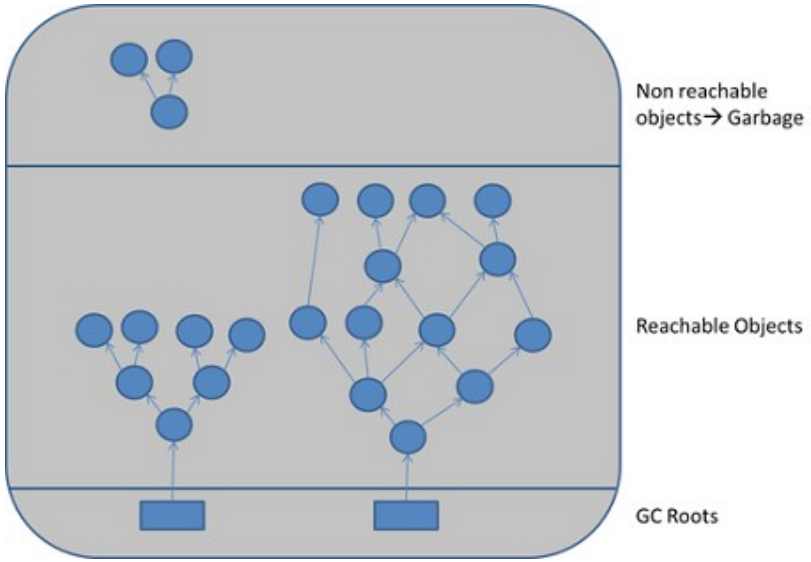
1. У фоновому режимі: процес (потік) GC запускається («просинається») періодично (в першу чергу в моменти «простою»).

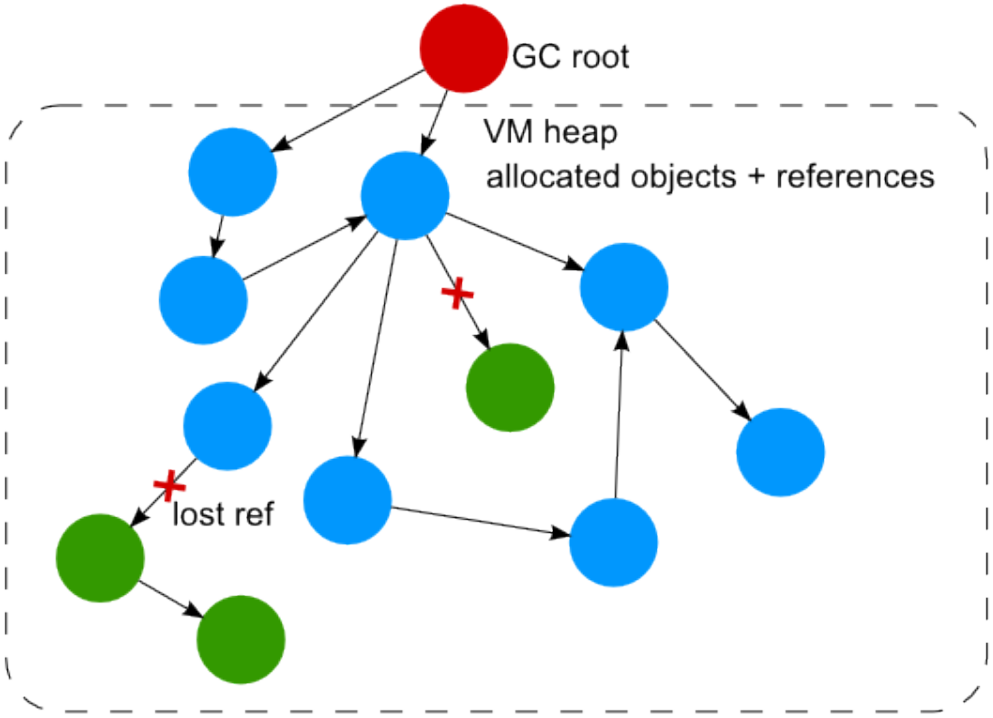
2. При потребі: процес (потік) GC запускається в критичні моменти нестачі вільної пам’яті.

04.5. Механізм автоматичного управління пам’яттю

- Основна проблема: як визначити, що даний об’єкт більше не потрібний?
- Основний принцип: визначення «досяжності» об’єкта (наявність посилань на об’єкт в інших об’єктах).
- Garbage = недосяжний об’єкт = об’єкт, на який нема посилань.
- Кореневі об’єкти (root set) = об’єкти, які потрібні завжди без врахування наявності на них посилань (наприклад, глобальні змінні).

04.6. Механізм автоматичного управління пам’яттю





Приклади схем посилань між об’єктами

05.1. Недоліки та переваги автоматичного управління пам’яттю

Переваги:

1. Забезпечення більшого рівня безпеки при роботі з пам’яттю.

2. Більша зручність програмування в порівнянні з ручним управлінням пам’яттю (спрощення процеса програмування).

3. Функціональна декомпозиція: усі «низькорівневі» операції з пам’яттю віділені у окремий програмний модуль (GC).

05.2. Недоліки та переваги автоматичного управління пам’яттю

Недоліки:

1. Втрати у продуктивності: 1.1) GC споживає ресурси системи; 1.2) в окремих випадках спостерігається низька ефективність збирання сміття (програш універсального рішення спеціалізованому).

2. Запуск (спрацювання) GC є «непередбачуваним» у часі з точки зору окремої програми і може призвести до затримок у її роботі (що, наприклад, неприпустимо для систем реального часу).

3. Семантичне звільнення пам’яті (semantic garbage collection) залишається відкритим питанням: GC «не вміє» враховувати логіку роботи програми.

06. Алгоритми роботи Garbage Collector

Tracing garbage collection
- Алгоритм позначок доступності об’єктів (mark and sweep).
- Алгоритм Tri-color marking.
- Алгоритм Card Marking.

Reference counting
- Алгоритм підрахунку посилань (reference counting).
- Weighted reference counting.
- Automatic Reference Counting (ARC).

Інші
- Алгоритм поколінь об’єктів (generational collection).
- Incremental and concurrent garbage collectors.

06.1.1. Алгоритм позначок доступності об’єктів (mark and sweep)

Для роботи алгоритма потрібно:

1. Виділити біт-позначку (=прапорець доступності) d для кожного об’єкту даних (data object) в пам’яті об’єктів.

2. Забезпечити доступ GC до всіх об’єктів даних у пам’яті (не зважаючи на те доступні вони чи ні).

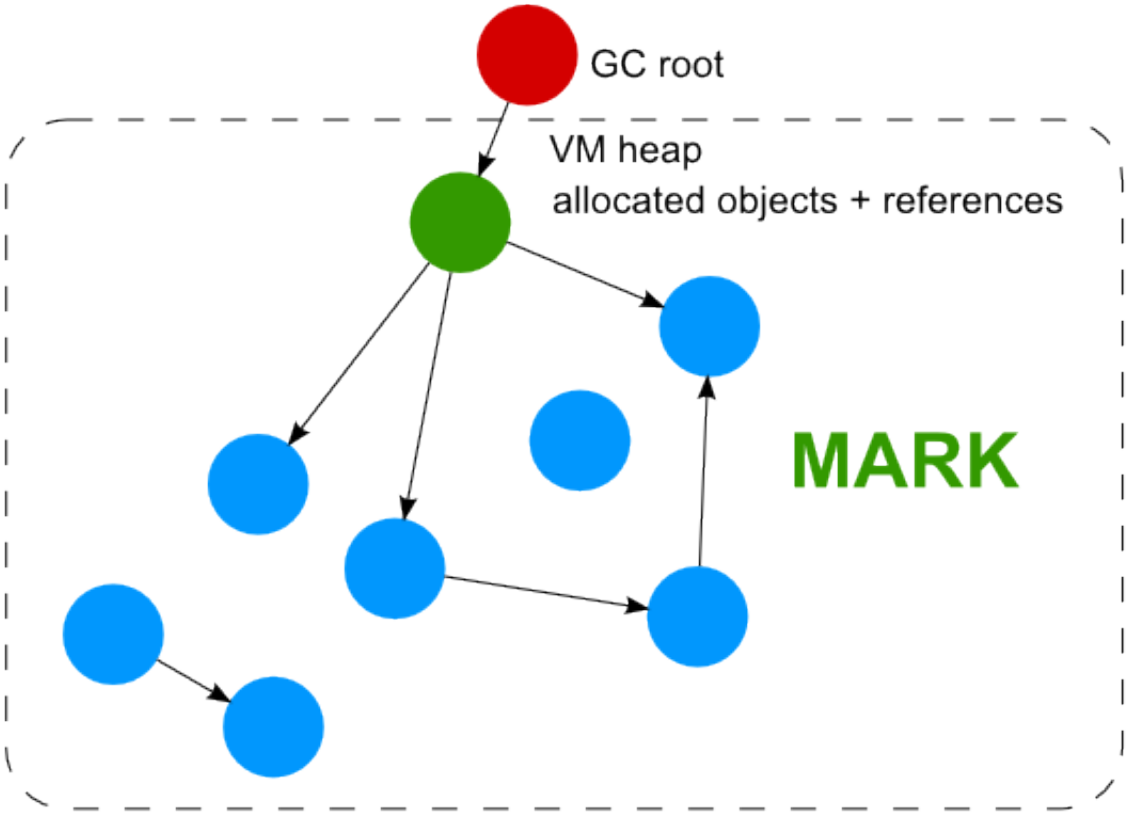
3. Визначити множину кореневих об’єктів (root set): статичні змінні та інщі об’єкти даних, які доступні «відразу» без посилань.

06.1.2. Алгоритм позначок доступності об’єктів (mark and sweep)

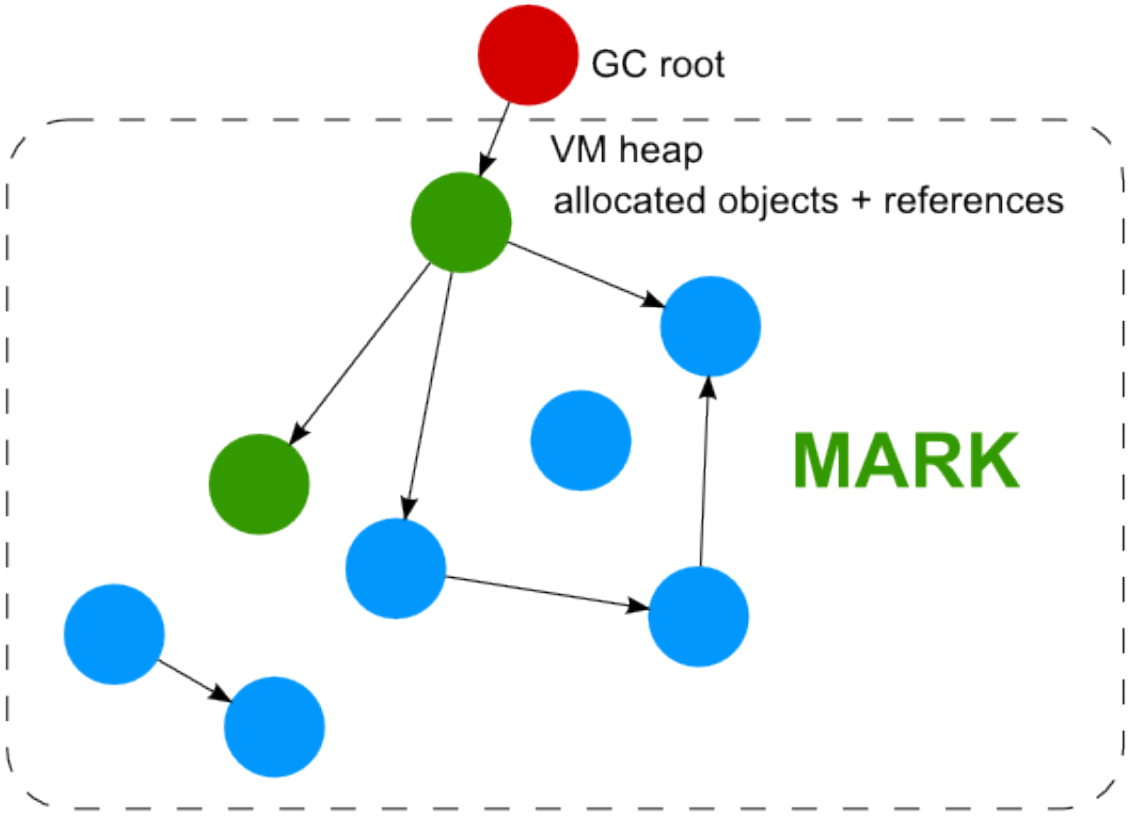
Алгоритм складається з трьох кроків:

1. Позначити всі об’єкти даних як недоступні («обнулення» прапорців доступності): в циклі по всіх об’єктах d := 0
2. Mark phase:
2.1. в циклі по об’єктах, що входять в root set, для кожного з них знайти усі посиланя на інші об’єкти і позначити ці об’єкти як доступні (d := 1);
2.2. рекурсія: повторити крок 2.1. для об’єктів позначених як доступні;
2.3. правило зупинки: якщо не виявлено жодного посилання, то зупинитись і перейти до п.3, інакше повторити пп. 2.1-2.2.
3. Sweep phase: в циклі по всіх об’єктах: якщо об’єкт має позначку «недоступний» (d = 0), то звільнити пам’ять, яку він займає.

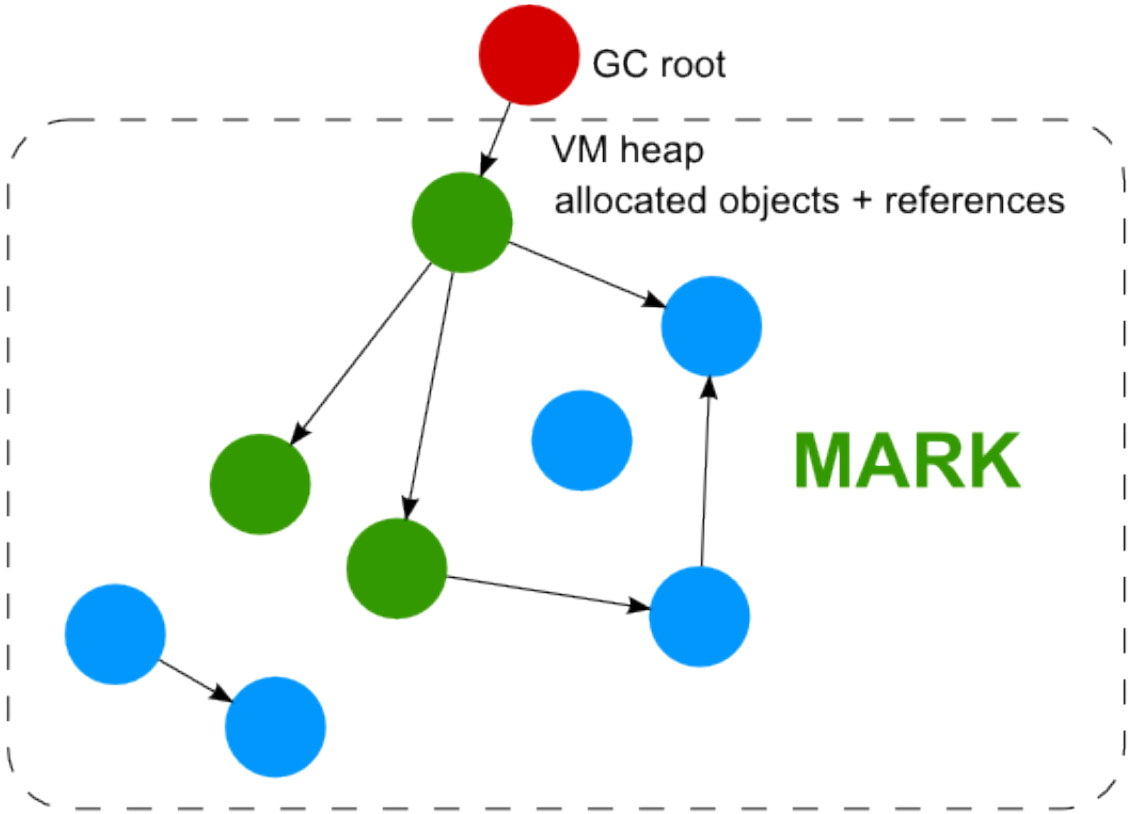
06.1.3. Алгоритм позначок доступності об’єктів (mark and sweep)



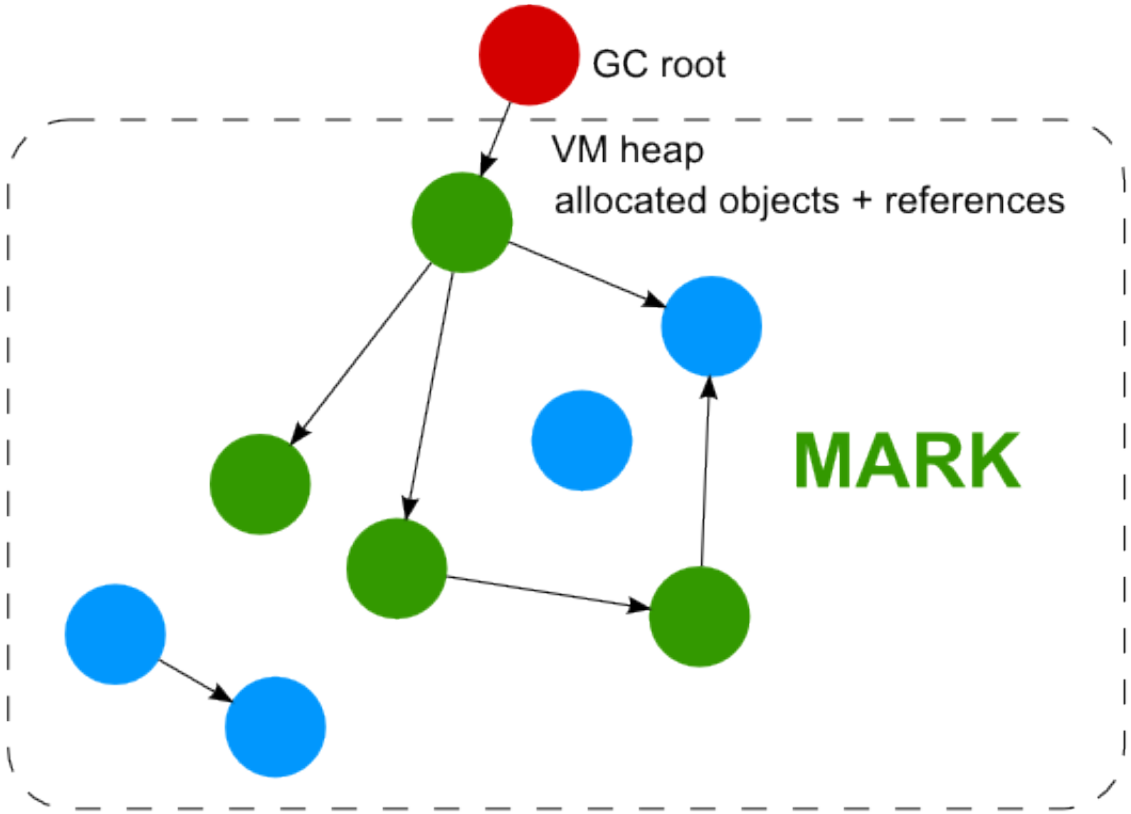
06.1.4. Алгоритм позначок доступності об’єктів (mark and sweep)



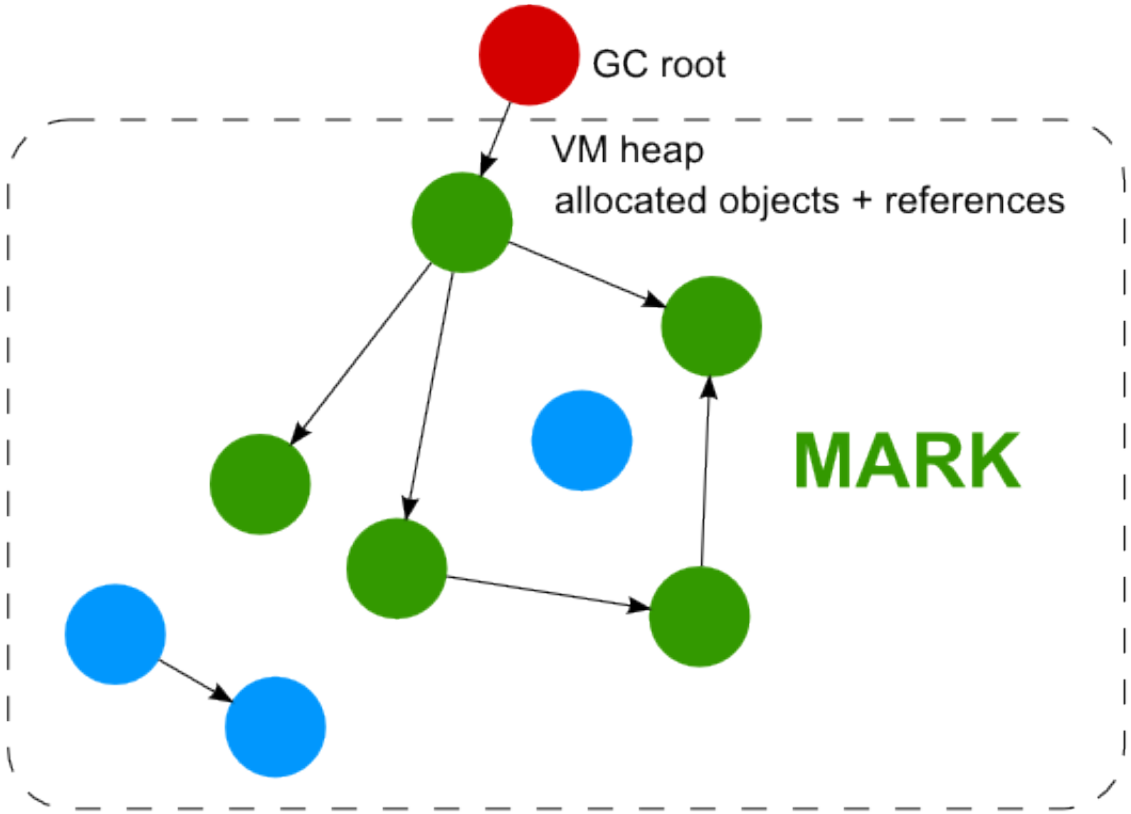
06.1.5. Алгоритм позначок доступності об’єктів (mark and sweep)



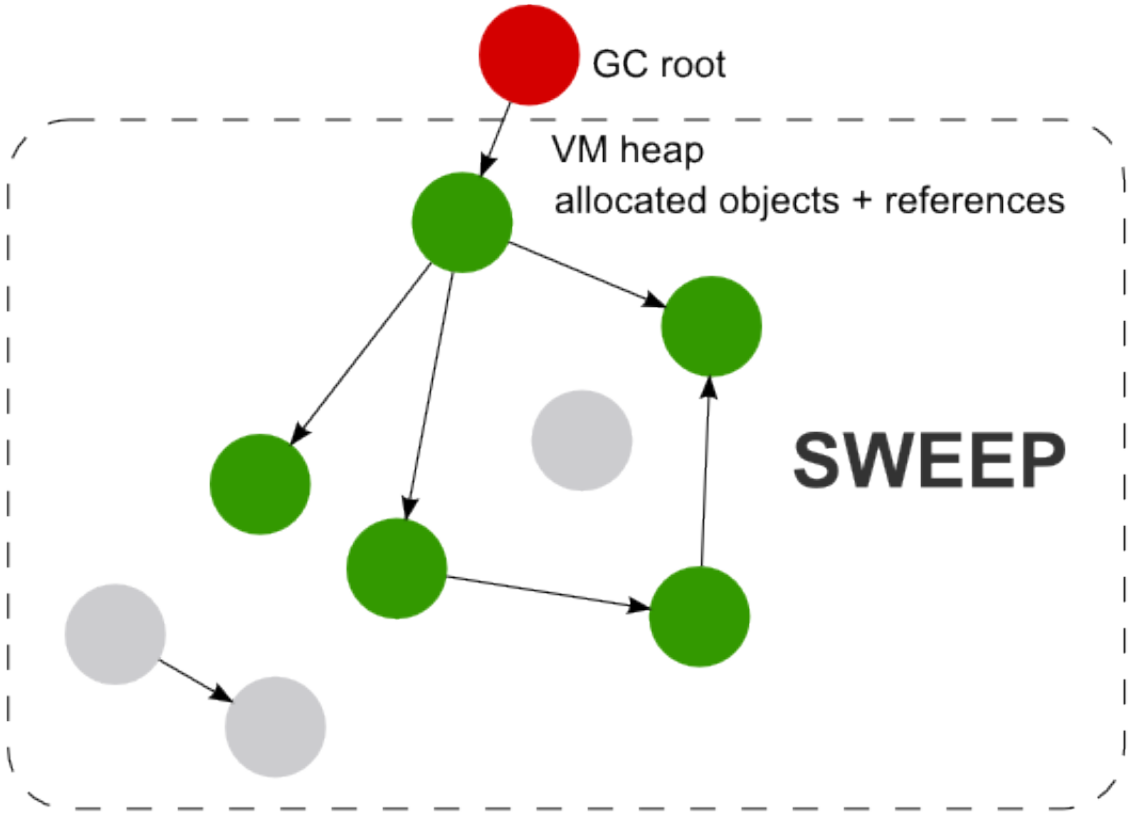
06.1.6. Алгоритм позначок доступності об’єктів (mark and sweep)



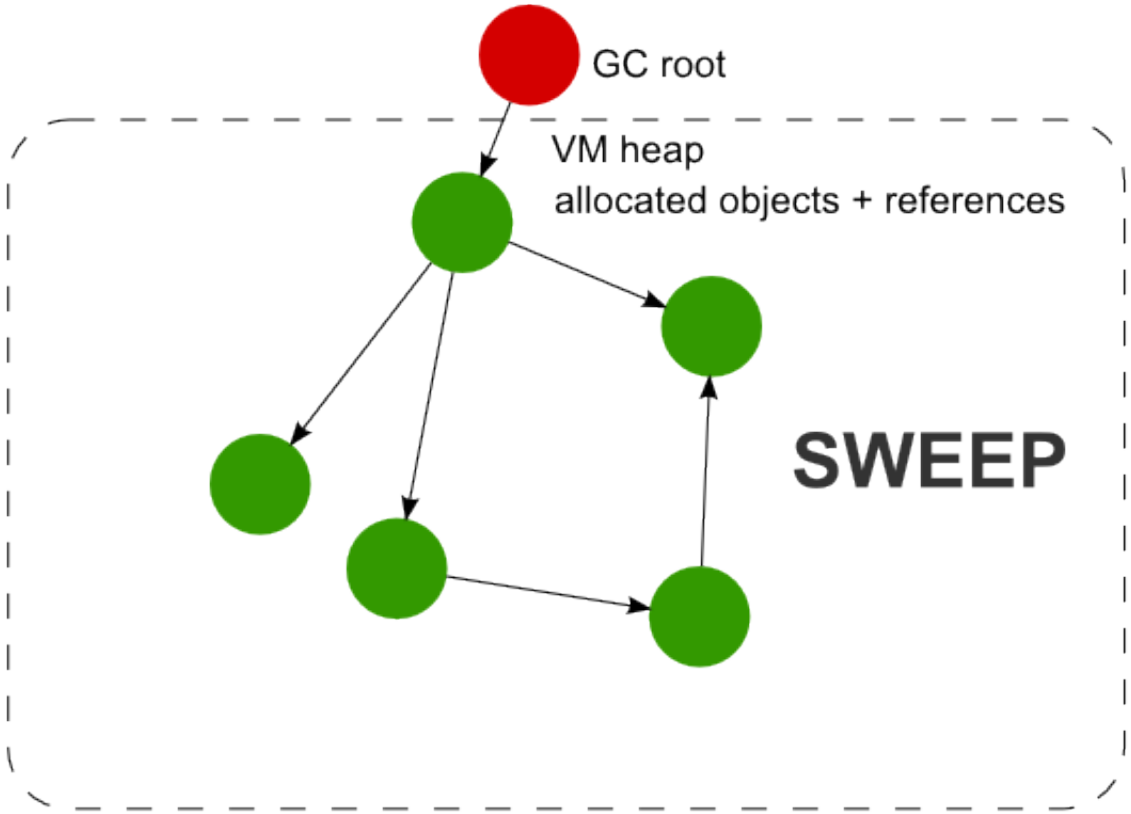
06.1.7. Алгоритм позначок доступності об’єктів (mark and sweep)



06.1.8. Алгоритм позначок доступності об’єктів (mark and sweep)



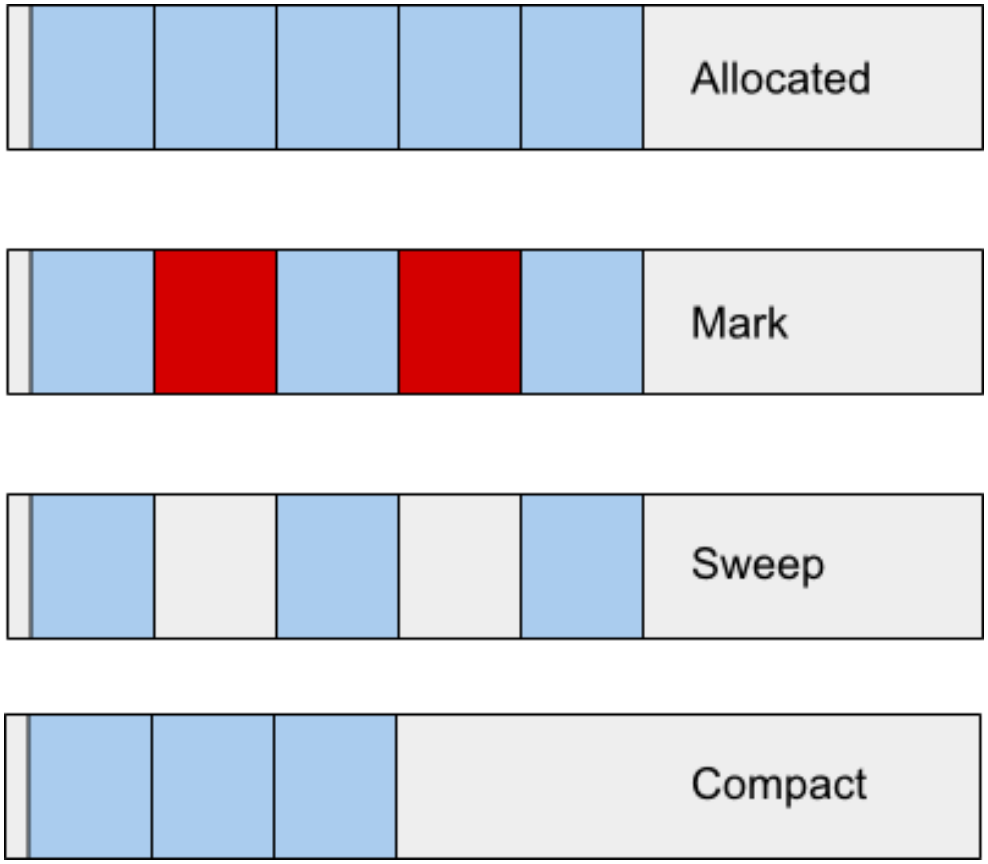
06.1.9. Алгоритм позначок доступності об’єктів (mark and sweep)



06.1.10. Алгоритм позначок доступності об’єктів (mark and sweep)

- Алгоритм Mark-and-Sweep є базовим і найбільш простим серед інших алгоритмів.
- Алгоритм Mark-and-Sweep виконує пошук в ширину для кожного граф-дерева з коренем у відповідному об’єкті опорної множини (root set).
- Варіат: алгоритм Mark-Compact, в якому додається 4-ий крок: «ущільнення» (дефрагментація) пам’яті об’єктів.

06.1.11. Алгоритм позначок доступності об’єктів (mark and sweep)



Приклад роботи алгоритма Mark-Compact

06.1.12. Алгоритм позначок доступності об’єктів (mark and sweep)

Переваги:

- простота реалізації;
- надійність роботи (будь які підмножини недостуних об’єктів будуть гарантовано виявлені).

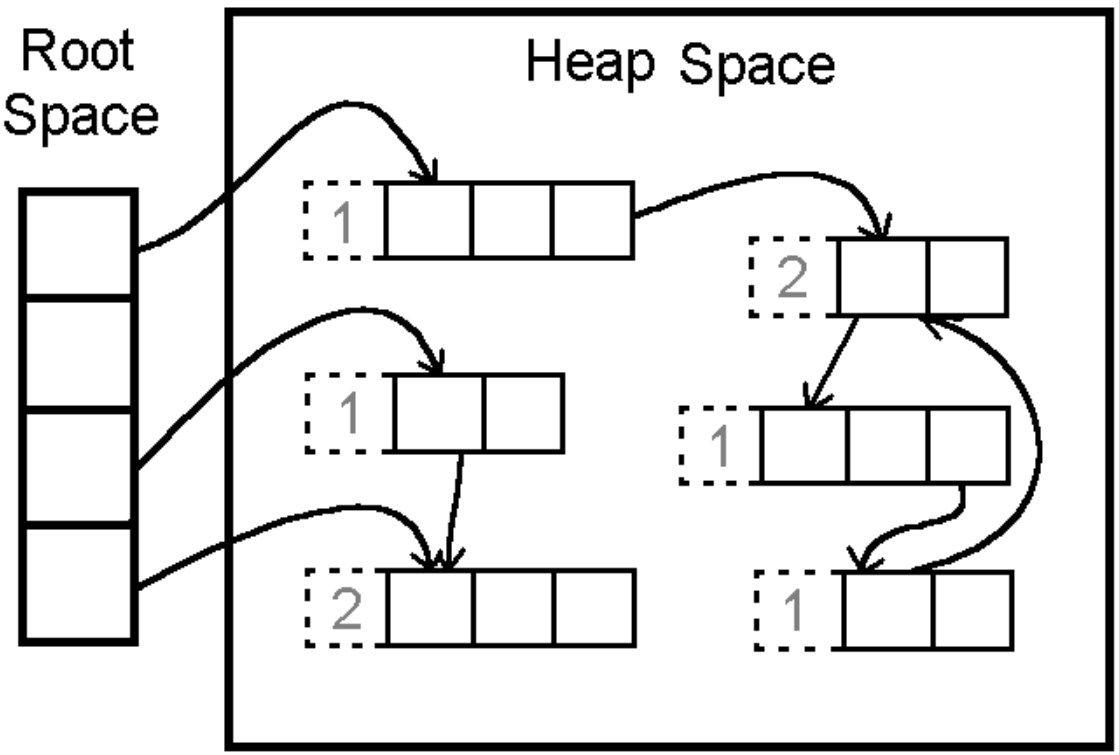
Недоліки:

- на час роботи алгоритма система має бути повністю зупинена (щоб не вносились зміни у граф посилань), що може призвести до непередбачуваних зависань програми;
- алгоритм перебирає усі об’єкти в пам’яті двічі (+ доступні тричі), що займає багато часу (а також може потребувати підкачки сторінок з зовнішньої пам’яті).

06.2.1. Алгоритм підрахунку посилань (reference counting)

- Для кожного об’єкту даних визначається кількість посилань на нього.
- Алгоритм спрацьовує при кожній зміні у схемі посилань між об’єктами, змінюючи відповідним чином лічильник посилань.
- Зауваження: в багато-поточних програмах (multi-threading) має бути забезпечена атомарність операцій з посиланнями на об’єкти.
- Об’єкт даних, лічильник посилань якого скидається до нуля, знищується.

06.2.2. Алгоритм підрахунку посилань (reference counting)



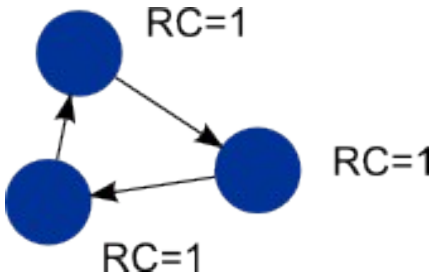


Схема роботи алгоритма Reference Counting

06.2.3. Алгоритм підрахунку посилань (reference counting)

Переваги:

- малий час відгуку: об’єкт видаляється відразу після того, як стає непотрібним (це підходить для систем реального часу);
- відсутній циклічний перегляд всієї пам’яті: алгоритм спрацьовує по факту змін.

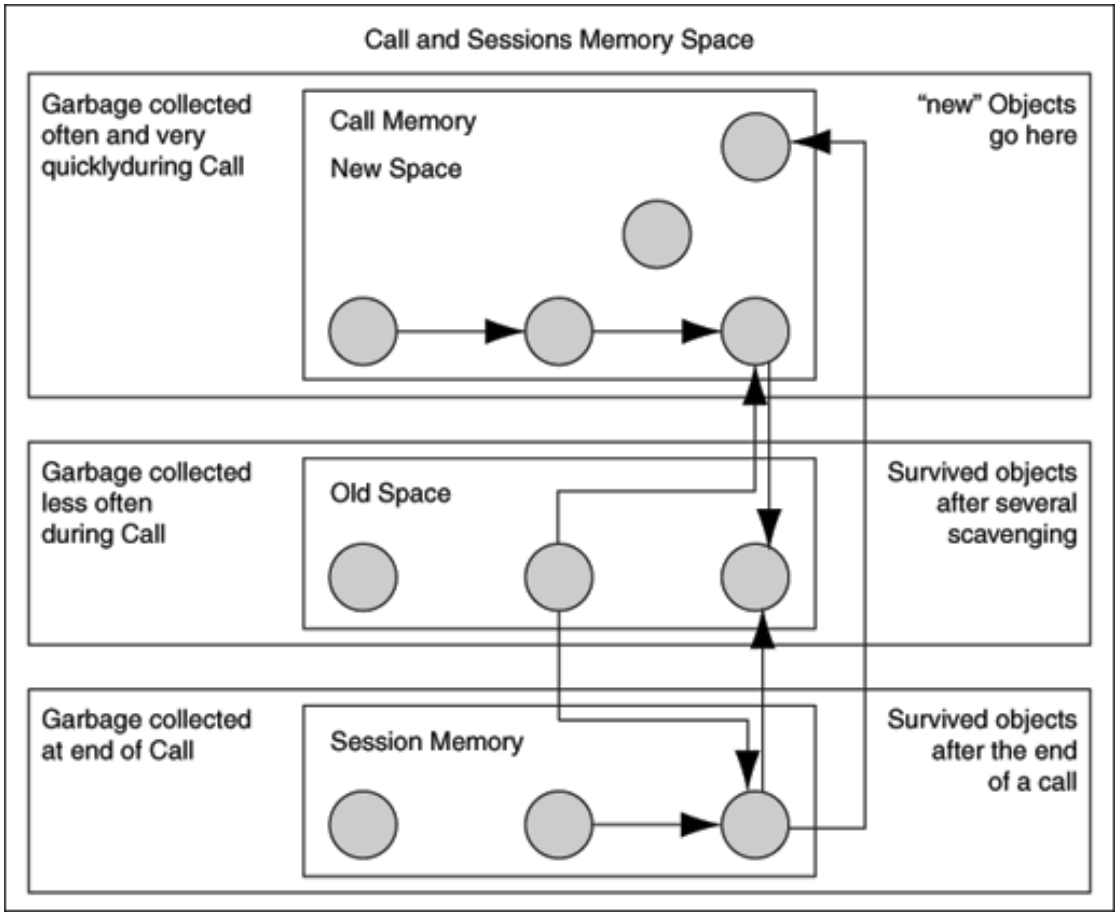
Недоліки:

- уповільнення операцій з посиланнями на об’єкти;
- неможливість виявлення та видалення циклічних структур в графі посилань, які «відокремлені» від основного графа.

06.3.1. Алгоритм поколінь об’єктів (generational collection)

- Основний принцип: нещодавно створені об’єкти «живуть» менше за інших («most objects die young»).
- Для ново-створених об’єктів ймовірність того, що вони в найближчий час стануть непотрбіними є найбільшою.
- Всі об’єкти розбиваються на декілько «поколінь» (generations) за часом присутності в пам’яті об’єктів.

06.3.2. Алгоритм поколінь об’єктів (generational collection)

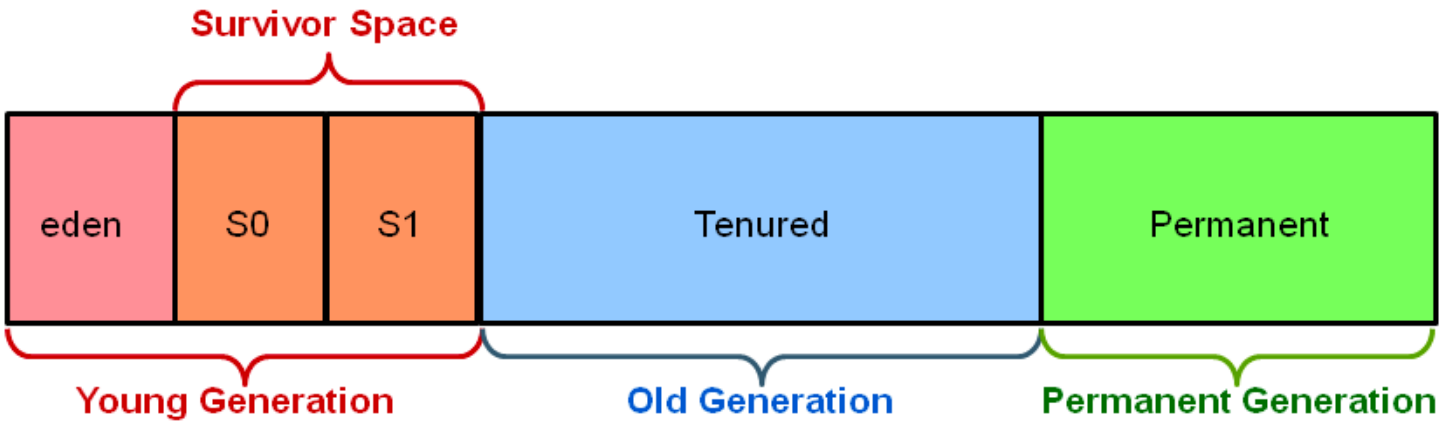


Приклад роботи алгоритма Generational Collection

06.3.3. Алгоритм поколінь об’єктів (generational collection)

- В разі необхідності (коли потрібно звільнити пам’ять для нових даних) в першу чергу переглядаються наймолодші об’єкти: визначається чи є на них посилання з root set та з об’єктів старших «поколінь» (алгоритм Mark-and-Sweep).
- Якщо звільненого місця недостатньо, запускається перегляд наступного «покоління» і т.д.
- Якщо об’єкт пережив «чистку» (одну або декілька), то він переходить у наступне «покоління» об’єктів.

06.3.4. Алгоритм поколінь об’єктів (generational collection)



Приклад: JVM Java HotSpot → Heap Structure → три «покоління» об’єктів: Young, Old, Permanent.

06.3.5. Алгоритм поколінь об’єктів (generational collection)

Переваги:

- скорочення циклу збирання сміття за рахунок зменшення кількості об’єктів, що переглядаються;
- для «чистки» різних «поколінь» об’єктів можна використовувати різні алгоритми (балансування надійності Mark-and-Sweep і швидкодії Reference Counting).

Недоліки:

- необхідність окремо відслідковувати зв’язки між об’єктами різних «поколінь» (додаткові структури даних та витрати обчислювальних ресурсів);
- витрати на переміщення об’єктів між «поколіннями» (з точки зору необхідності вирішення проблеми дефрагментації пам’яті об’єктів).