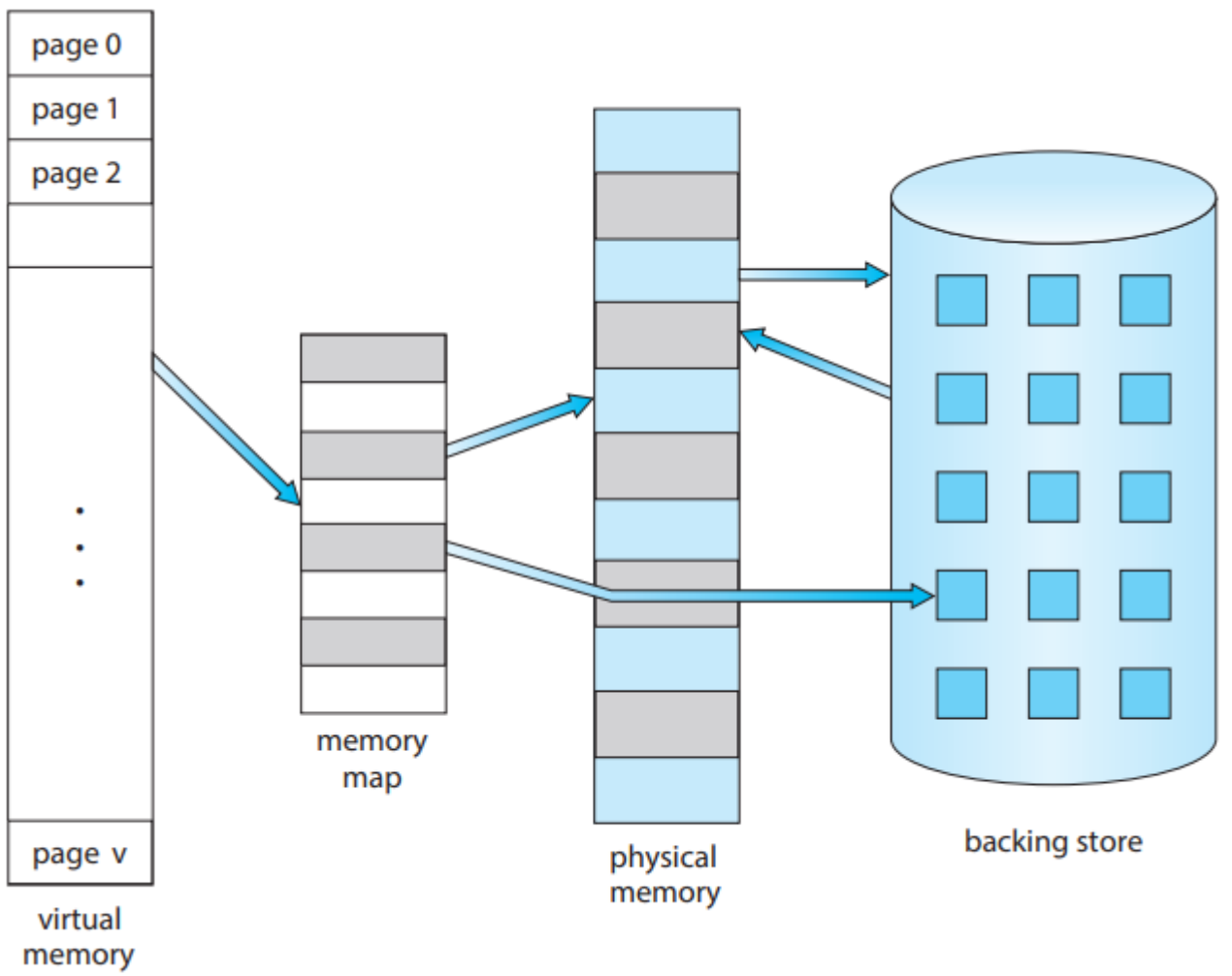
# ОСНОВНІ ПОНЯТТЯ В КОНТЕКСТІ ВІРТУАЛЬНОЇ ПАМ’ЯТІ

Питання 3.3

# Бекграунд

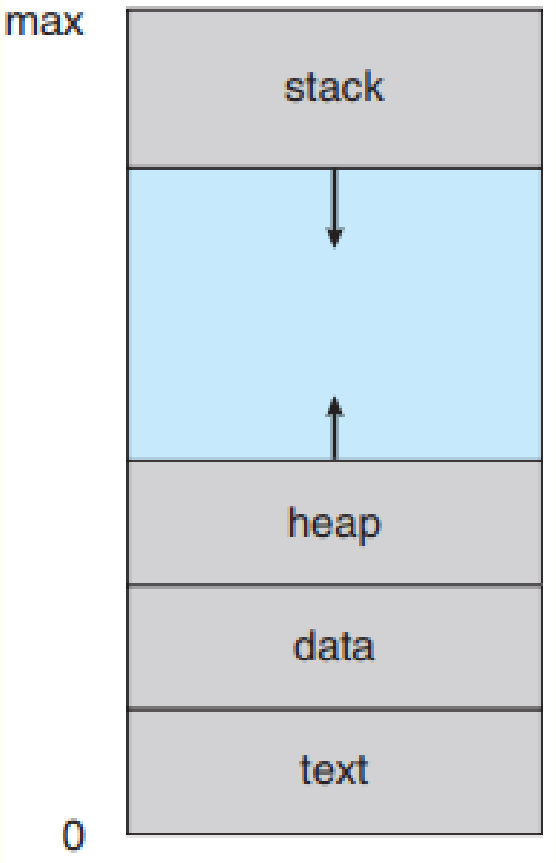
- Попередні алгоритми керування пам’яттю необхідні в зв’язку з тим, що виконувані інструкції повинні бути в основній пам’яті.
  - Перший підхід: розмістити весь логічний адресний простір у фізичній пам’яті.
  - Динамічне компонування допоможе спростити дане обмеження, проте загалом вимагає спеціальних заходів та додаткової роботи програміста.
- Вимога розміщення в фізичній пам’яті обмежує розмір програми розміром цієї пам’яті.
  - Фактичний огляд програм показує, що в багатьох випадках уся програма відразу не потрібна.
  - Наприклад, наступні ситуації:
    - Програми часто мають код для обробки нетипових ситуацій. Оскільки такі ситуації трапляються рідко, він майже не використовується.
    - Масивам, спискам та таблицям часто виділяється більше пам’яті, ніж їм потрібно.
    - Деякі опції та можливості програми можуть рідко застосовуватись. Наприклад, підпрограми уряду США для балансування бюджету не використовувались багато років.
  - Навіть у випадках, коли потрібна вся програма, це не означає, що потрібна відразу.

# Віртуальна пам’ять, яка більша за фізичну



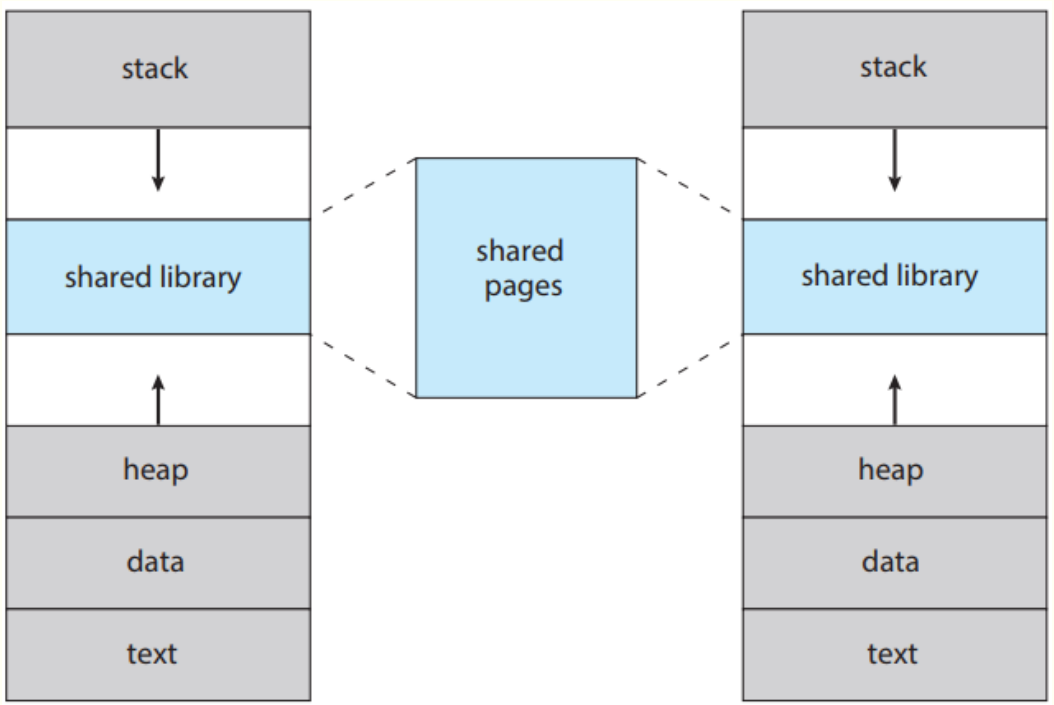

- **Віртуальна пам’ять** включає відокремлення логічної пам’яті від фізичної.
  - Вона набагато спрощує програмування, оскільки програміст більше не переживає за доступну фізичну пам’ять, а більше концентрується на задачі для розв’язування.
- Здатність виконувати програму, яка лише частково представлена в пам’яті, надає багато переваг:
  - Знімається обмеження на доступну фізичну пам’ять. Користувачі зможуть писати програми для дуже великих ***віртуальних*** адресних просторів.
  - Оскільки кожна програма споживає менше фізичної пам’яті, більше програм можна запустити одночасно, що також підвищує задіяність ЦП та пропускну здатність без зростання часу відгуку чи turnaround time.
  - Буде потрібно менше операцій вводу-виводу, щоб завантажити чи swap частини програми в пам’ять, тому кожна програма зможе працювати швидше.

# Віртуальний адресний простір




- Це логічне(віртуальне) представлення того, як процес зберігається в пам’яті.
  - Переважно процес починається в деякій логічній адресі та безперервно займає ділянку пам’яті.
  - Фізична пам’ять організована в page frames та не обов’язково безперервна.
  - Блок керування пам’яттю (MMU) відображає логічні сторінки у фізичні page frames у пам’яті.
- Зауважте на рисунку зустрічне зростання кучі та стеку.
  - Дірка між стеком та кучою є частиною віртуального адресного простору, проте вимагає реальних фізичних сторінок тільки за умови росту стеку чи кучі.
  - Віртуальні адресні простори, які включають дірки, називають ***розрідженими (sparse)*.**
  - Їх перевагою є можливість заповнення дірок при розростанні сегментів пам’яті або динамічному підключенні бібліотек та спільних об’єктів протягом виконання програми.

# Додатково віртуальна пам’ять дозволяє ділитись файлами та пам’яттю серед кількох процесів через page sharing



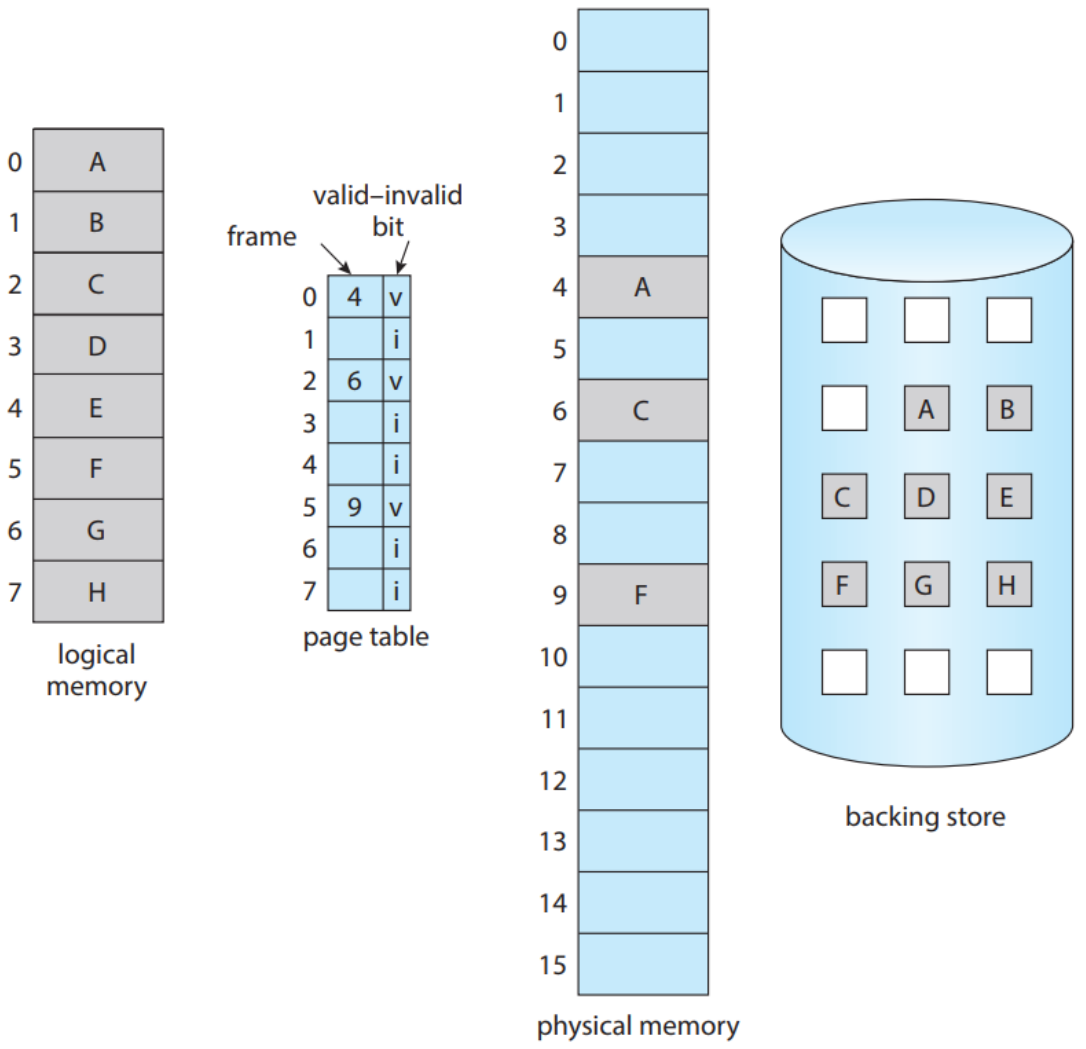

Маємо наступні переваги:

- Системні бібліотеки можна шерити на кілька процесів через відображення спільного об’єкта у віртуальний адресний простір.
  - Хоч кожний процес вважає бібліотеки частиною свого віртуального адресного простору, справжні сторінки, в яких розміщено бібліотеки в фізичній пам’яті, розподілені на всі процеси.
- Аналогічно процеси можуть мати спільну пам’ять.
  - Віртуальна пам’ять дозволяє одному процесу створювати область пам’яті, якою він може поділитись з іншим процесом.
- Сторінки можуть шеритись під час створення процесу за допомогою системного виклику fork(), таким чином прискорюючи подібне створення.

# Підкачування сторінок за вимогою (demand paging)

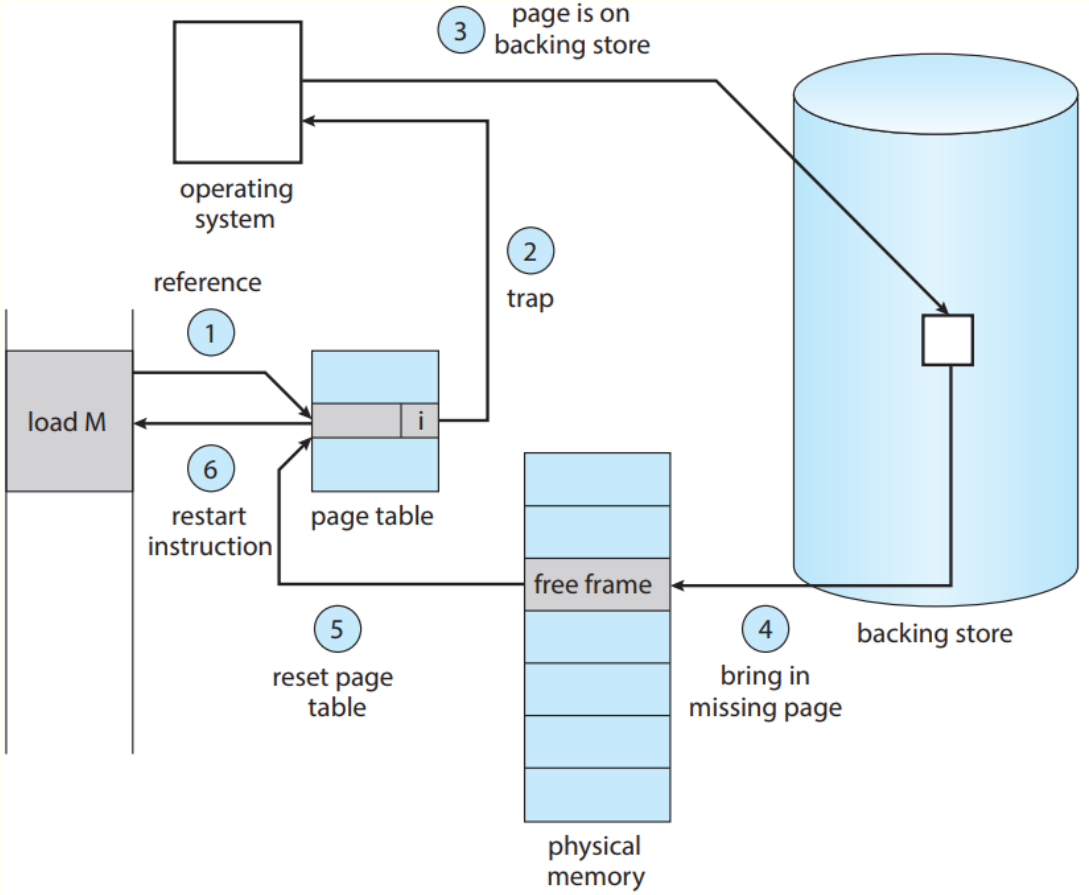
- Розглянемо, як виконувана програма може завантажуватись у пам’ять із вторинного сховища.
- Варіант 1: завантажити всю програму в фізичну пам’ять під час виконання цієї програми.
  - Проблема: може бути так, що нам не потрібна вся програма в пам’яті відразу.
  - Нехай програма починає роботу зі списку доступних операцій, одну з яких обирає користувач.
  - При завантаженні цілої програми в пам’ять маємо код ***усіх*** операцій, незалежно від потреби в них.
- Варіант 2: завантажувати сторінки лише за потреби.
  - Методика називається ***підкачуванням сторінок за потреби*** та широко використовується з системах з віртуальною пам’яттю.
  - Сторінки, доступ до яких не запитувався, ніколи не завантажуються у фізичну пам’ять.
  - Підкачування сторінок за вимогою подібне до сторінкової організації пам’яті зі свопінгом, де процеси розміщуються у вторинній пам’яті.

# Таблиця сторінок, коли деякі сторінки не в основній пам’яті




- Поки процес виконується, деякі сторінки будуть у пам’яті, а деякі – у вторинному сховищі.
  - Тому потрібна деяка форма апаратної підтримки для їх розрізнення – вже згаданий valid–invalid біт.
- Проте цього разу, коли біт встановлюється як “valid”, відповідна сторінка легальна та розміщена в пам’яті.
  - Для значення “invalid” сторінка або не знаходиться в логічному адресному просторі процесу, або сторінка валідна, проте на даний момент знаходиться у вторинному сховищі.

# Що трапляється, коли процес намагається отримати доступ до сторінки, не внесеної в пам'ять?




- Доступ до сторінки, позначеної як invalid, спричиняє ***відмову (збій) сторінки (page fault)***.
  - Апаратне забезпечення для сторінкової організації пам'яті при трансляції адреси через таблицю сторінок помітить, що встановлено invalid-біт, тому ОС буде передано виняток.
- Процедура обробки збою сторінки:
  - **1.** Перевіряємо внутрішню таблицю (зазвичай зберігається в PCB) для даного процесу, щоб визначити, чи валідне посилання для доступу до пам'яті.
  - **2.** Якщо посилання було invalid, перериваємо процес. Якщо посилання валідне, але сторінка не в пам'яті, підкачуємо її.
  - **3.** Знаходимо вільний фрейм (наприклад, з free-frame list).
  - **4.** Плануємо операцію зчитування бажаної сторінки з вторинного сховища в нововиділений фрейм.
  - **5.** Коли зчитування зі сховища завершене, змінюємо внутрішню таблицю, що постачається з процесом, та таблицю сторінок, щоб вказати на появу сторінки в пам'яті.
  - **6.** Перезапускаємо перервану винятком інструкцію. Тепер процес може отримати доступ до сторінки.

# У крайньому випадку, можемо почати виконання процесу без сторінок у пам’яті

- Коли ОС встановлює instruction pointer на першу інструкцію процесу, яка знаходиться в незавантаженій у пам’ять сторінці, процес негайно faults for the page.
  - Після  внесення сторінки в пам’ять процес продовжує виконуватись, faulting за потреби, поки кожна необхідна сторінка не буде в пам’яті.
  - Це схема ***чистої сторінкової організації пам’яті за вимогою (pure demand paging)***: ніколи не вносити в пам’ять незатребувану сторінку.
- Теоретично деякі програми можуть отримати доступ до кількох нових сторінок пам’яті при кожному виконанні інструкції (1 сторінка на інструкцію та багато для даних), ймовірно, спричиняючи множинні збої сторінок на інструкцію.
  - Така ситуація неприйнятна в контексті продуктивності системи.
  - Аналіз запущених процесів показує, що така поведінка дуже малоймовірна.

- Апаратна підтримка demand paging така ж, як і для paging та свопінгу: таблиця сторінок + вторинна пам’ять
- Важливою вимогою до demand paging є здатність перезапускати будь-яку інструкцію після збою сторінки.
  - Оскільки стан (registers, condition code, instruction counter) перерваного процесу зберігається, коли трапляється відмова сторінки, нам необхідно перезапустити процес у ***точно*** тому ж місці та стані.
  - У більшості випадків виконати таку вимогу просто.
  - Збій сторінки може трапитись при будь-якому memory reference.
  - Якщо збій сторінки трапляється при фетчингу інструкції, можемо перезапустити його знову.
  - Якщо відмова сторінки відбувається протягом фетчингу операнда, необхідно витягувати та декодувати інструкцію заново, а потім витягувати операнд.

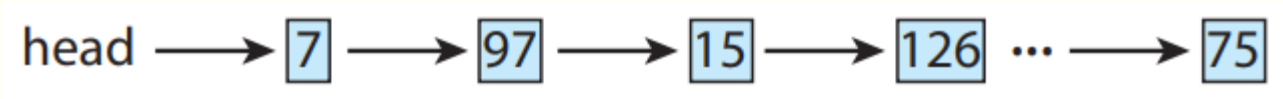
# Найгірший випадок

- Нехай триадресна інструкція, наприклад, ADD, працює із вмістом A та B, розміщуючи результат у C. Кроки виконання:
  - **1.** Витягнути (Fetch) та декодувати інструкцію (ADD).
    **2.** Витягнути A.
    **3.** Витягнути B.
    **4.** Додати A та B.
    **5.** Зберегти суму в C.
- Якщо збій відбувається при збереженні в C (оскільки C в незавантаженій у пам'ять сторінці), буде потрібно отримати бажану сторінку, внести її в пам'ять, скоригувати таблицю сторінок та перезапустити інструкцію.
  - Перезапуск вимагає заново витягнути інструкцію, декодувати її, витягнути 2 операнди та додати їх значення. Маємо багато повторної роботи.
- Основна складність виникає, коли одна інструкція може змінювати кілька різних розташувань.
  - Наприклад, інструкція MVC (move character) в IBM System 360/370 може переміщувати до 256 байтів з одного місця в інше (можливо, з перетинанням).
  - Якщо either block (source or destination) straddles a page boundary, a page fault might occur after the move is partially done.
  - In addition, if the source and destination blocks overlap, the source block may have been modified, in which case we cannot simply restart the instruction.

- Проблему можна вирішити кількома способами.
  - 1) мікрокод обчислює та намагається отримати доступ до both ends обох блоків. Якщо траплятиметься збій сторінки, він станеться на цьому кроці, до внесення будь-яких змін. Потім зможе відбутись переміщення.
  - 2) використовувати тимчасові регістри для зберігання значень з перезаписаних розташувань. Якщо виникає page fault, all the old values are written back into memory before the trap occurs. This action restores memory to its state before the instruction was started, so that the instruction can be repeated.
- Існує єдина архітектурна проблема включення demand paging до існуючої архітектури.
  - Підкачування сторінок додається між ЦП та пам’яттю в комп’ютерній системі. Воно повинно бути прозорим для процесу.
  - Тому люди часто вважають, що paging можна додати в будь-яку систему..
  - Дане припущення істинне в non-demand-paging environment, де збій сторінки представляє fatal error.
  - Проте воно хибне в середовищі, де збій сторінки означає лише необхідність внесення в пам’ять додаткової сторінки та перезапуск процесу.

# Список вільних фреймів (Free-Frame List)

- Коли стається збій сторінки, ОС повинна внести бажану сторінку із вторинного сховища в основну пам’ять.
  - Для обробки збійних сторінок більшість ОС підтримують **free-frame list** – пул вільних фреймів для задоволення подібних запитів (також вільні фрейми виділяються при розширенні стеку чи кучі процесу).




- ОС зазвичай виділяють вільні фрейми за допомогою підходу ***заповнення пустоти за потреби (zero-fill-on-demand)***.
  - Zero-fill-on-demand фрейми «очищаються» перед виділенням.
  - Коли система запускається, уся доступна пам’ять розміщується в списку вільних фреймів.
  - По мірі запитування вільних фреймів (наприклад, в контексті сторінкової організації пам’яті за вимогою), розмір списку вільних фреймів зменшується.
  - У деякий момент список спустошується або зменшується до розміру, нижче порогового значення.
  - У цей момент список повинен заново наповнюватись (repopulate).

# Продуктивність сторінкової організації пам’яті за вимогою

- Demand paging може значно впливати на швидкодію комп’ютерної системи.
  - Обчислимо **effective access time** для demand-paged пам’яті.
  - Нехай memory-access time (*ma) = 10нс.* Поки не буде відмов сторінок, цей час дорівнює ефективному часу доступу.
  - При траплянні збою сторінки спочатку необхідно зчитати потрібну сторінку з вторинного сховища, а потім отримати доступ до бажаного слова.
- Нехай *p* – ймовірність збою сторінки (0 ≤ *p* ≤ 1). Очікуємо, що *p* близька до 0.
  - effective access time = (1 - *p*) × *ma* + *p* × page fault time.

# Потрібно знати, скільки часу обробляється збій сторінки

- Обробка збою передбачає наступні кроки:
  - **1.** Згенерувати виняткову ситуацію для ОС.
    **2.** Зберегти регістри та стан процесу.
    **3.** Визначити, що переривання відбулось через збій сторінки.
    **4.** Перевірити, що посилання на сторінку було легальним та визначити розташування сторінки у вторинній пам'яті.
    **5.** Виконати зчитування із вторинної пам'яті у вільний фрейм:
    - Очікувати в черзі, поки запит на зчитування не буде виконано.
    - Очікувати відповідно до часу пошуку та/або латентності для пристрою.
    - Почати передачу сторінки у вільний фрейм.
  - **6.** Протягом очікування виділити ядро ЦП деякому іншому процесу.
    **7.** Отримати переривання від підсистеми вводу-виводу сховища (ввід-вивід завершено).
    **8.** Зберегти регістри та стан іншого процесу (якщо виконується крок 6).
    **9.** Визначити, що переривання було від пристрою вторинного сховища.
    **10.** Виправити таблицю сторінок та інші таблиці, щоб показати наявність бажаної сторінки в пам'яті.
    **11.** Очікувати, поки ядро ЦП знову буде виділене даному процесу.
    **12.** Відновити регістри, стан процесу, нову таблицю сторінок, а потім відновити виконання перерваної інструкції.

- Не усі кроки завжди є необхідними.
  - Наприклад, у кроці 6 передбачаємо, що ЦП виділений іншому процесу, поки відбувається ввід-вивід.
  - Ця угода дозволяє підтримувати задіяність ЦП, проте вимагає додаткового часу, щоб відновити підпрограму обробки зброю сторінки, коли I/O transfer завершилась.

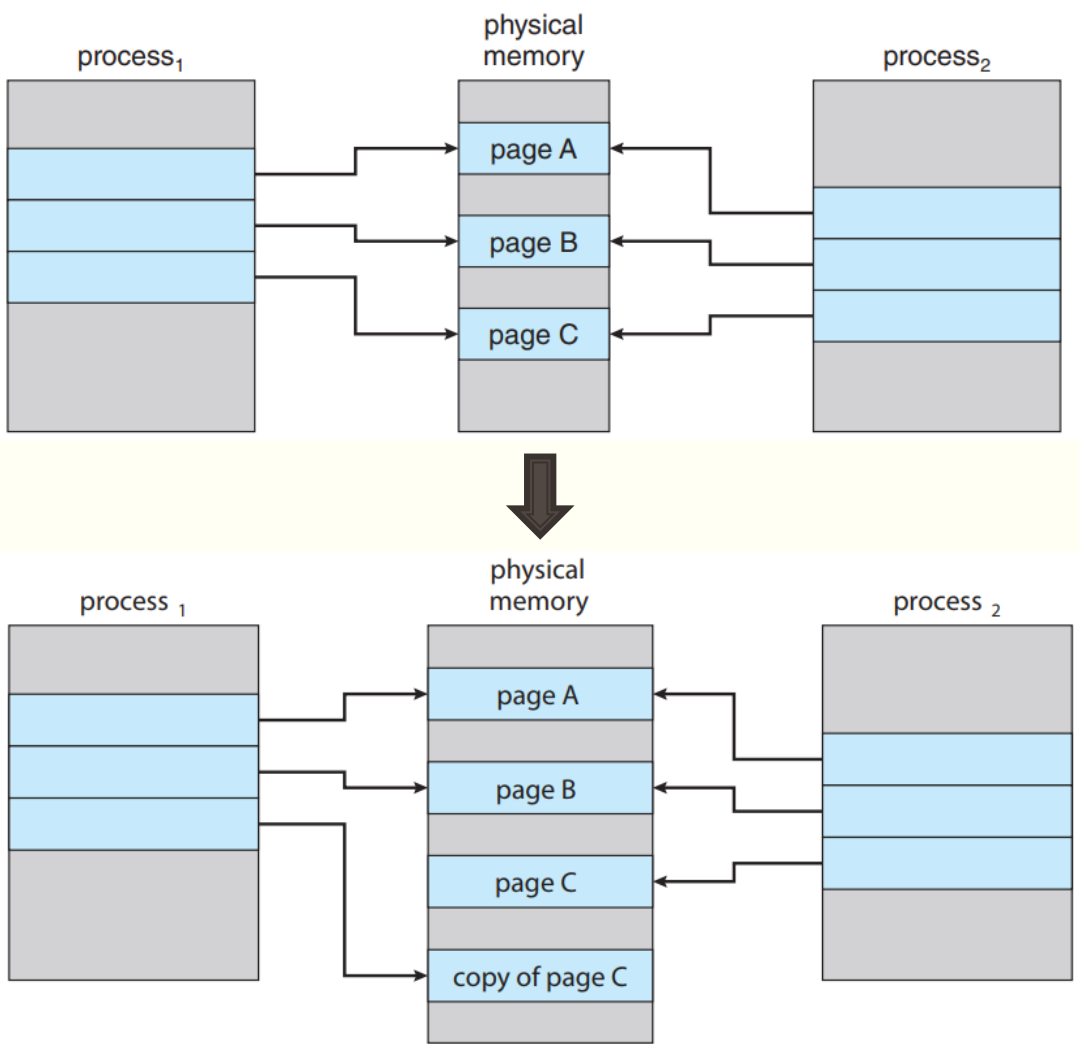
- У будь-якому разі, присутні 3 головні компоненти задачі:
  - **1.** Обробити переривання від збою сторінки.
  - **2.** Зчитати сторінку в пам’ять.
  - **3.** Перезапустити процес.
- Перший та третій компонент можна спростити за допомогою ефективного коду до кількох сотень інструкцій.
  - Ці компоненти можуть забирати від 1 до 100 мкс кожний.
- Розглянемо випадок HDD в якості пристрою для підкачування.
  - Час page-switch буде близьким до 8мс (середня латентність доступу – 3мс + середній пошук – 5мс + час передачі - 0.05мс).
  - Пам’ятайте, що розглядаємо зараз тільки device-service час.
  - Якщо черга процесів очікує на пристрій, потрібно додати час перебування в черзі (queuing time), поки очікуємо на звільнення paging device для обслуговування запиту. Тоді час зросте ще більше.

- З середнім часом обслуговування збою сторінки у 8мс та часом доступу до пам’яті в 200нс, ефективний час доступу в наносекундах такий:
  - effective access time = (1 - *p*) × 200 + *p* × 8,000,000 = 200 + 7,999,800 × *p.*
- Ефективний час доступу прямо пропорційний **ймовірності відмови сторінки (page-fault rate)**.
  - При 1 збої на 1000 доступів маємо ефективний час 8.2 мкс.
  - Комп’ютер сповільниться в 40(!) разів через demand paging!
  - Якщо бажаємо зниження продуктивності не більше 10%, необхідно підтримувати ймовірність збою сторінки на наступному рівні:
    - $220 > 200 + 7{,}999{,}800 \times p,$
      $20 > 7{,}999{,}800 \times p,$
      $p < 0.0000025.$
  - Тому можемо дозволити не більше ніж 1 збійний доступ на 399,990 доступів до пам’яті.

# Додатковий аспект demand paging – обробка та загальне використання swap space

- Ввід-вивід у простір підкачування загалом швидший, ніж до файлової системи.
  - Це пов’язано з набагато більшими блоками та застосовуваними file lookups і непрямими методами алокації.
  - 1) копіювання цілого образу файлу в swap space при запуску процесу з подальшим виконанням demand paging зі swap space. Очевидний недолік – копіювання file image при запуску програми.
  - 2) опція, яка застосовується на практиці кількома ОС, включаючи Linux та Windows: спочатку підкачувати сторінку за вимогою з файлової системи, а записувати сторінки у swap space по мірі їх заміщення. Даний підхід забезпечить можливість зчитування лише потрібних сторінок з файлової системи, проте усе наступне підкачування виконується зі swap space.
- Деякі системи намагаються обмежити обсяг swap space, який використовується для demand paging бінарних виконуваних файлів.
  - Demand pages для таких файлів вносяться напряму з файлової системи.
  - Проте при виклику заміщення сторінки ці фрейми можуть бути перезаписаними (оскільки до цього в них не вносились зміни), а сторінки and the pages can be read in from the file system again if needed.
  - Using this approach, the file system itself serves as the backing store.
  - However, swap space must still be used for pages not associated with a file (known as **anonymous memory**); these pages include the stack and heap for a process.
  - This method appears to be a good compromise and is used in several systems, including Linux and BSD UNIX.

# Копіювання при запису (Copy-on-Write)




- Раніше було продемонстровано, як процес може швидко запуститись за допомогою demand paging у сторінці, яка містить першу інструкцію.
  - However, process creation using the fork() system call may initially bypass the need for demand paging by using a technique similar to page sharing.
  - This technique provides rapid process creation and minimizes the number of new pages that must be allocated to the newly created process.
- Recall that the fork() system call creates a child process that is a duplicate of its parent.
  - Traditionally, fork() worked by creating a copy of the parent's address space for the child, duplicating the pages belonging to the parent.
  - However, considering that many child processes invoke the exec() system call immediately after creation, the copying of the parent's address space may be unnecessary.
  - Instead, we can use a technique known as **copy-on-write**, which works by allowing the parent and child processes initially to share the same pages.
  - These shared pages are marked as copy-on-write pages, meaning that if either process writes to a shared page, a copy of the shared page is created.

- For example, assume that the child process attempts to modify a page containing portions of the stack, with the pages set to be copy-on-write.
  - The operating system will obtain a frame from the free-frame list and create a copy of this page, mapping it to the address space of the child process.
  - The child process will then modify its copied page and not the page belonging to the parent process.
  - Obviously, when the copy-on-write technique is used, only the pages that are modified by either process are copied; all unmodified pages can be shared by the parent and child processes.
  - Note, too, that only pages that can be modified need be marked as copy-on-write.
  - Pages that cannot be modified (pages containing executable code) can be shared by the parent and child.
  - Copy-on-write is a common technique used by several operating systems, including Windows, Linux, and macOS.
- Several versions of UNIX (including Linux, macOS, and BSD UNIX) provide a variation of the fork() system call—vfork() (for **virtual memory fork**)—that operates differently from fork() with copy-on-write.
  - With vfork(), the parent process is suspended, and the child process uses the address space of the parent.
  - Because vfork() does not use copy-on-write, if the child process changes any pages of the parent's address space, the altered pages will be visible to the parent once it resumes.
  - Therefore, vfork() must be used with caution to ensure that the child process does not modify the address space of the parent.
  - vfork() is intended to be used when the child process calls exec() immediately after creation.
  - Because no copying of pages takes place, vfork() is an extremely efficient method of process creation and is sometimes used to implement UNIX command-line shell interfaces.