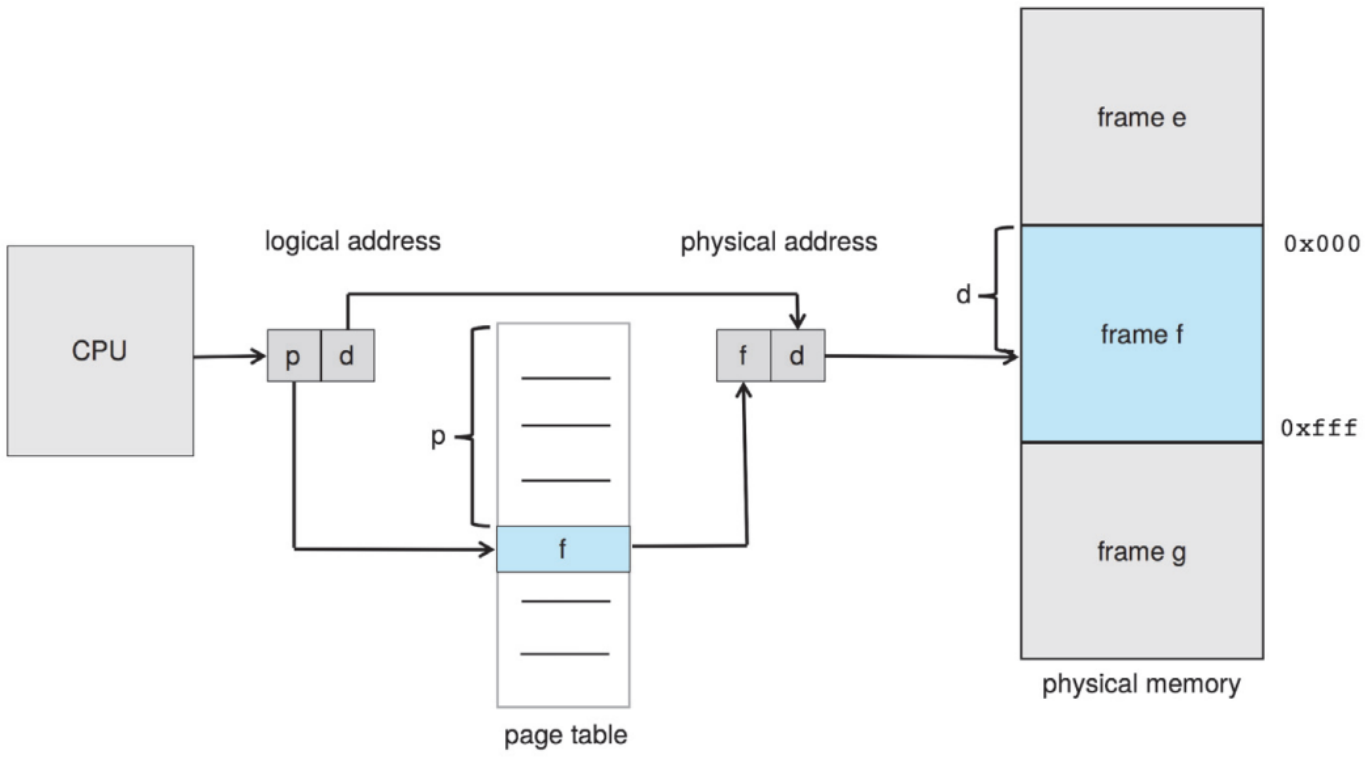
СТОРІНКОВА ОРГАНЗАЦІЯ ПАМ’ЯТІ

Питання 3.2

Підкачування сторінок (paging)

- Схема управління пам’яттю, яка дозволяє фізичному адресному простору процесу бути розривним (noncontiguous).
  - Усуває зовнішню фрагментацію та пов’язану з нею потребу в стисненні.
  - Завдяки багатьом своїм перевагам підкачування сторінок у різних формах використовується в більшості ОС.
  - Підкачування сторінок реалізується через кооперування між ОС та апаратним забезпеченням.
- Базовий метод реалізації підкачування сторінок включає розбиття фізичної пам’яті на блоки фіксованого розміру – ***фрейми***, а логічної пам’яті – на аналогічні блоки – ***сторінки***.
  - Коли процес має виконуватись, його сторінки завантажуються в доступні фрейми пам’яті з джерела (файлової системи чи файлу відкачки (backing store)).
  - Файл відкачки розбивається на блоки фіксованого розміру, такого ж, як і фрейми або кластери фреймів.
  - Наприклад, логічний адресний простір тепер повністю відокремлений від фізичного адресного простору, тому процес може мати 64-бітний логічний адресний простір навіть у системі, яка має менше 2^{64} байтів фізичної пам’яті.

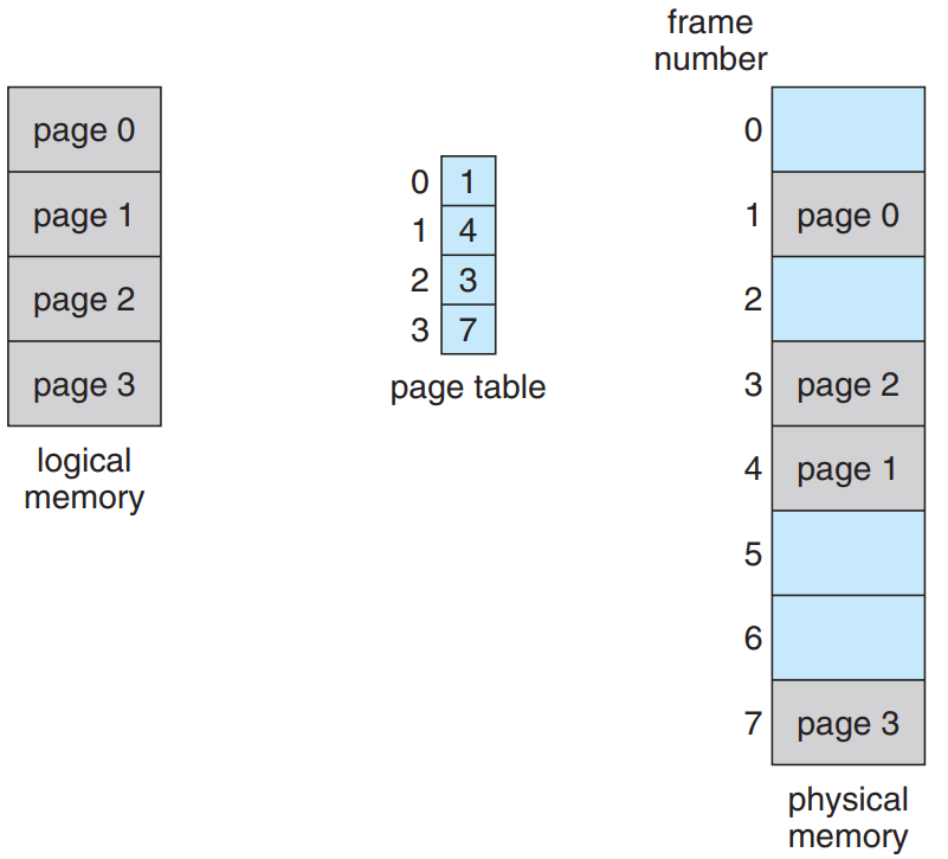
Апаратне забезпечення підкачування сторінок



- Кожна згенерована ЦП адреса розбивається на 2 частини: номер сторінки (page number, p) та зміщення сторінки (page offset, d):

  - Номер сторінки виступає індексом в таблиці сторінок для процесу.
  - Таблиця сторінок містить базову адресу кожного фрейму в фізичній пам'яті та зміщення – розташування в фреймі, на який посилаються.
  - Тому базова адреса фрейму комбінується з page offset, щоб визначити фізичну адресу пам'яті.

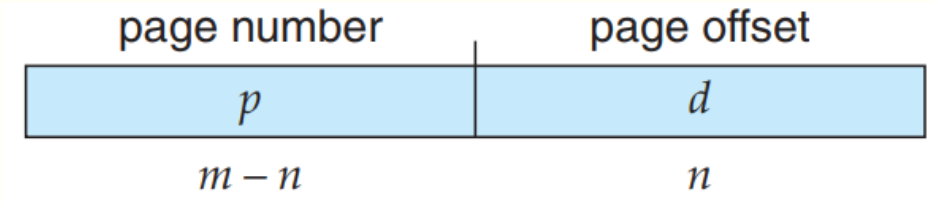
Модель підкачування для логічної та фізичної пам’яті



- MMU виконує наступні кроки для відображення логічної адреси, згенерованої ЦП, у фізичну адресу:
  - 1. Виокремити номер сторінки *p* та використати його як індекс таблиці сторінок.
  - 2. Виокремити відповідний номер фрейму *f* з таблиці сторінок.
  - 3. Замінити номер сторінки *p* в логічній адресі на номер фрейму *f*.
- Оскільки зміщення *d* не змінюється, воно разом з номером фрейму описує фізичну адресу.

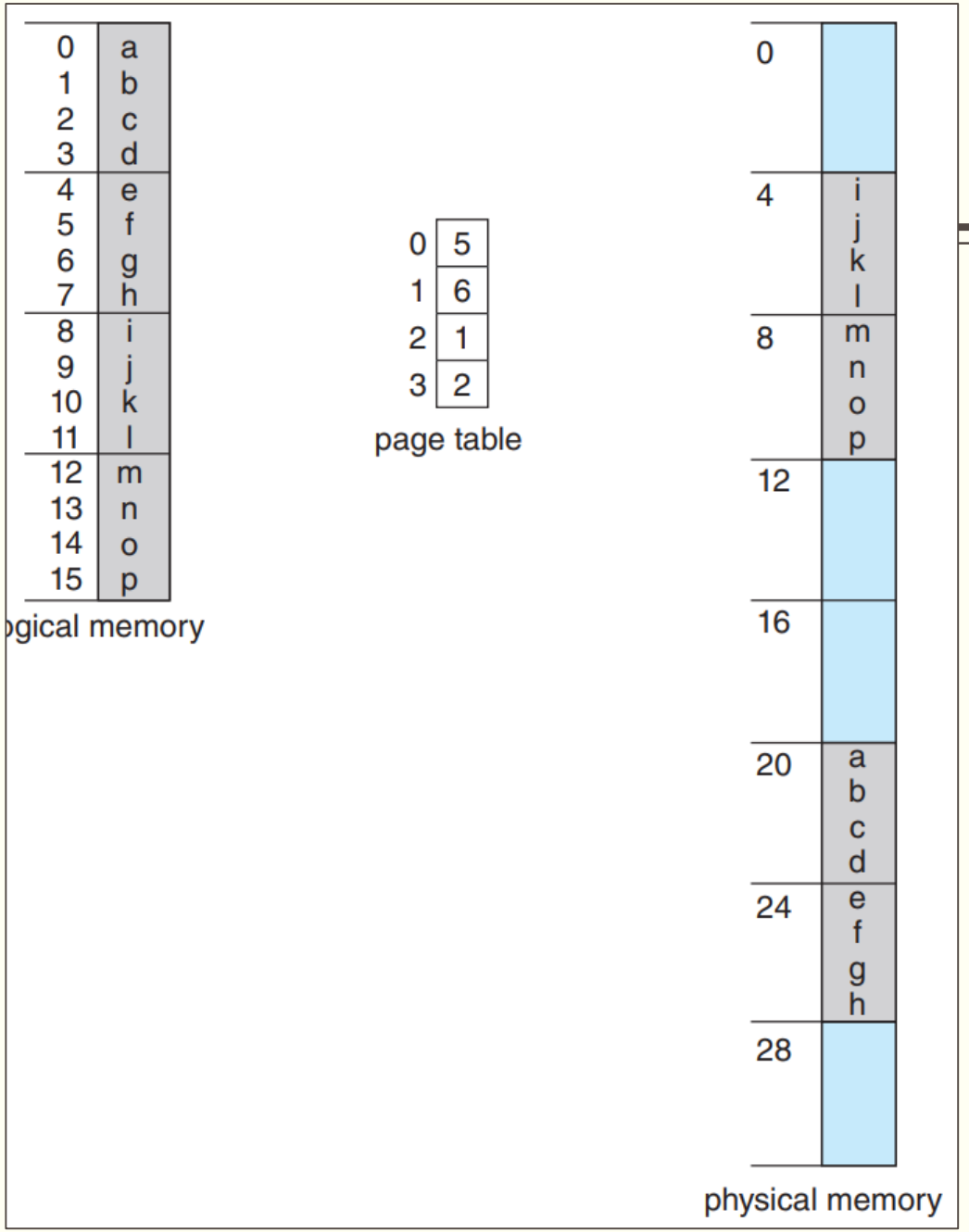
Модель підкачування для логічної та фізичної пам’яті

- Розмір сторінки (подібно до розміру фрейму) визначається апаратно.
  - Розмір сторінки є степенем 2, зазвичай коливаючись між 4 Кб та 1 Гб на сторінку, залежно від архітектури комп’ютера.
  - Вибір такого розміру сторінки спрощує відображення логічної адреси в номер сторінки та зміщення сторінки.
  - Якщо розмір логічного адресного простору - 2^m, а розмір сторінки - 2^n байтів, то high-order m-n бітів логічної адреси відводяться під номер сторінки, а n low-order бітів – під зміщення сторінки:



  - де p – індекс у таблиці сторінок, а d – зміщення всередині сторінки.

Приклад підкачування сторінок для 32-байтової пам'яті з 4-байтовими сторінками



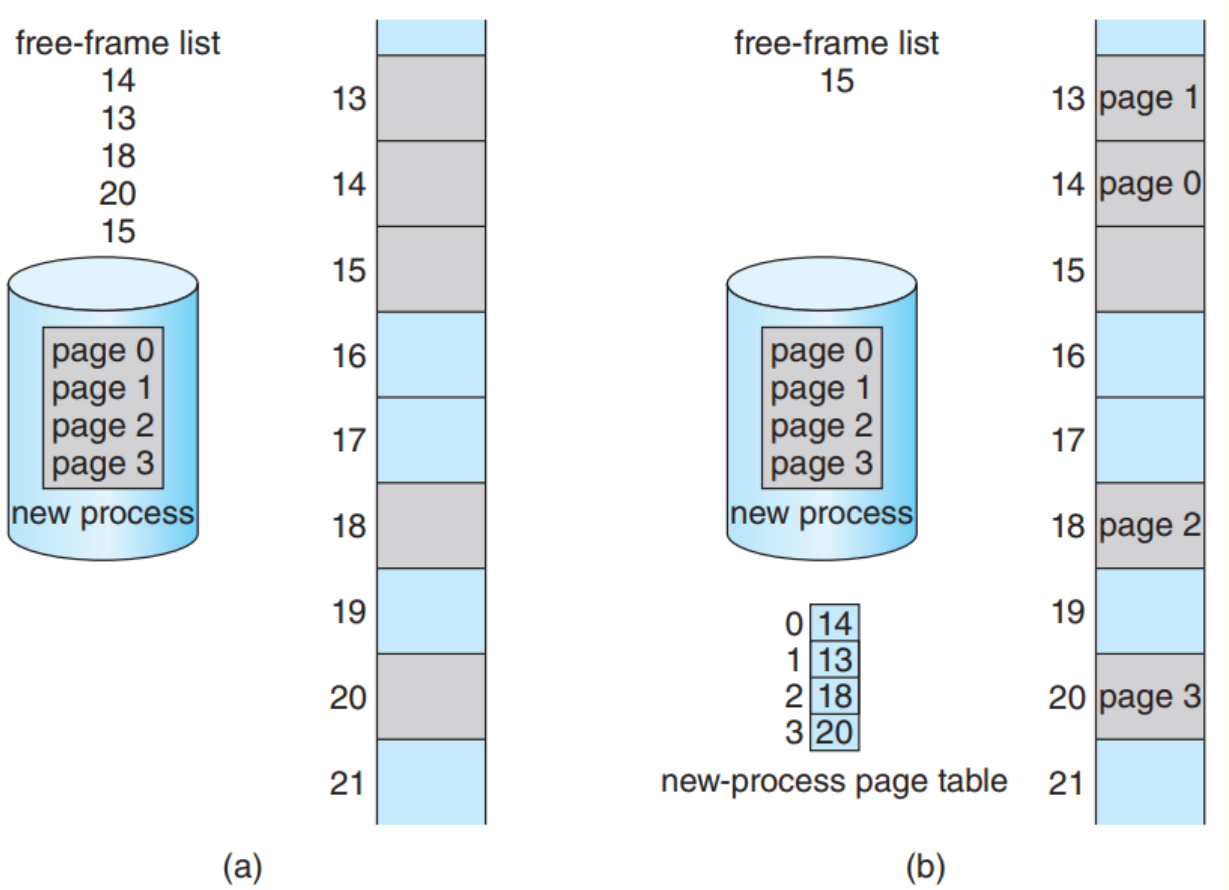
- Тут у логічних адресах $n = 2$, $m = 4$.
  - Логічна адреса 0 є сторінкою 0 зі зміщенням 0.
  - Індексування в таблиці сторінок показує, що сторінка 0 розміщена у фреймі 5: логічна адреса 0 відображається на фізичну адресу 20 [= (5 × 4) + 0].
  - Логічна адреса 3 (сторінка 0, зміщення 3) відображається на фізичну адресу 23 [= (5 × 4) + 3].
  - Логічна адреса 4 – це сторінка 1, зміщення 0; відображається в 6й фрейм, фізичну адресу 24 [= (6 × 4) + 0].
  - Логічна адреса 13 відображається в фізичну адресу 9.
- Ви могли помітити, що підкачування сторінок – форма динамічного переміщення.
  - Кожна логічна адреса прив'язана до деякої фізичної адреси за допомогою paging hardware.
  - Використання підкачування сторінок подібне до застосування таблиці регістрів бази (або переміщення), по одному на кожний фрейм пам'яті.

За використання підкачування сторінок немає зовнішньої фрагментації

- Будь-який вільний фрейм може бути виділеним процесу, який його потребує.
  - Проте можлива внутрішня сегментація, оскільки фрейми виділяються як units.
  - Якщо вимоги процесу до пам’яті не співпадають з межами сторінки, останній виділений фрейм може бути заповненим не повністю.
  - Наприклад, якщо розмір сторінки 2048 байтів, а процес з 72766 байтів потребуватиме 35 сторінок + 1086 байтів.
  - Йому виділиться 36 фреймів, а внутрішня фрагментація становитиме 2048 - 1086 = 962 байти.
  - У найгіршому випадку процесу знадобиться *n* сторінок + 1 байт. Буде виділено *n*+1 фреймів.
- Якщо розмір процесу не залежить від розміру сторінки, очікуємо середню фрагментацію близько половини сторінки на процес.
  - Логічно мати маленькі за розміром сторінки, проте накладні витрати на підтримку кожного запису в таблиці сторінок зростають.
  - Також дисковий ввід-вивід ефективніший, коли обсяг даних для передачі більший.
  - Загалом, розміри сторінок зростали з часом, оскільки процеси, набори даних та об’єм пам’яті ставили більше.
  - Нині сторінки зазвичай мають розмір 4Кб, 8Кб або навіть більше.
  - Деякі ЦП таОС навіть підтримують різні розміри сторінок. Наприклад, Windows 10 підтримує сторінки розмірами 4Кб та 2Мб.
  - Linux також підтримує 2 розміри: за умовчанням (зазвичай 4Кб) та залежний від архітектури більший розмір, який називають **huge pages**.

- Часто на 32-бітних ЦП кожний запис таблиці сторінок має довжину 4 байти, проте цей розмір теж може варіюватись.
  - 32-бітний запис може вказувати на один з 2^{32} фізичних сторінок (page frames).
  - Якщо розмір фрейму 4Кб (2^{12}), тоді система з 4-байтними записами зможе адресувати 2^{44} байтів (16Тб) фізичної пам'яті.
  - Слід зауважити, що розмір фізичної пам'яті в системі сторінкової організації пам'яті зазвичай відрізняється від максимального логічного розміру процесу.
- Далі представимо іншу інформацію, яку потрібно тримати в записах (page-table entries).
  - Вона зменшує кількість бітів, доступних для адресації page frames.
  - Тому система з 32-бітними page-table entries може адресувати менше можливого максимуму фізичної пам'яті.
- Коли процес надходить на виконання, система розглядає його розмір (у сторінках).
  - Кожна сторінка процесу потребує 1 фрейм, для *n* сторінок потрібно принаймні *n* доступних фреймів.

Вільні фрейми (a) до виділення та (b) після виділення



- Якщо доступно *n* фреймів, вони виділяться цьому процесу.
  - Перша сторінка процесу завантажується в один з виділених фреймів, а кількість фреймів вноситься в таблицю сторінок для процесу.
  - Наступна сторінка завантажується в інший фрейм, її frame number вноситься в таблицю сторінок і т. д.
- Важливий аспект підкачування сторінок – чітке відокремлення представлення пам'яті у програміста від реальної фізичної пам'яті.
  - Програміст бачить пам'ять як єдиний простір, який містить лише одну програму.
  - Фактично, користувацька програма розпорошена по фізичній пам'яті, яка також містить інші програми.
  - Логічні адреси транслюються в фізичні. Це відображення сховане від програміста та керується ОС.

Таблиця фреймів (frame table)

- Оскільки ОС керує фізичною пам’яттю, їй повинно бути відомо allocation details цієї пам’яті: виділені фрейми, доступні фрейми, їх загальна кількість тощо.
  - Ця інформація зазвичай тримається в єдиній системній структурі даних – ***таблиці фреймів***.
  - Вона має 1 запис (entry) для кожного фізичного page frame.
- Також ОС повинна знати, що користувацькі процеси працюють в просторі користувача, а всі логічні адреси необхідно відображати для отримання фізичних адрес.
  - Якщо користувач здійснює системний виклик (наприклад, для вводу-виводу) та постачає адресу як параметр (наприклад, буфер), цю адресу необхідно відобразити на коректну фізичну адресу.
  - ОС підтримує копію таблиці сторінок для кожного процесу, аналогічно до копій лічильника інструкцій та вмісту регістрів.
  - Дана копія використовується ОС для трансляції адрес та диспетчером ЦП для визначення апаратної таблиці сторінок, коли процесу буде виділено ЦП.
  - Підкачування сторінок increases час перемикання контексту.

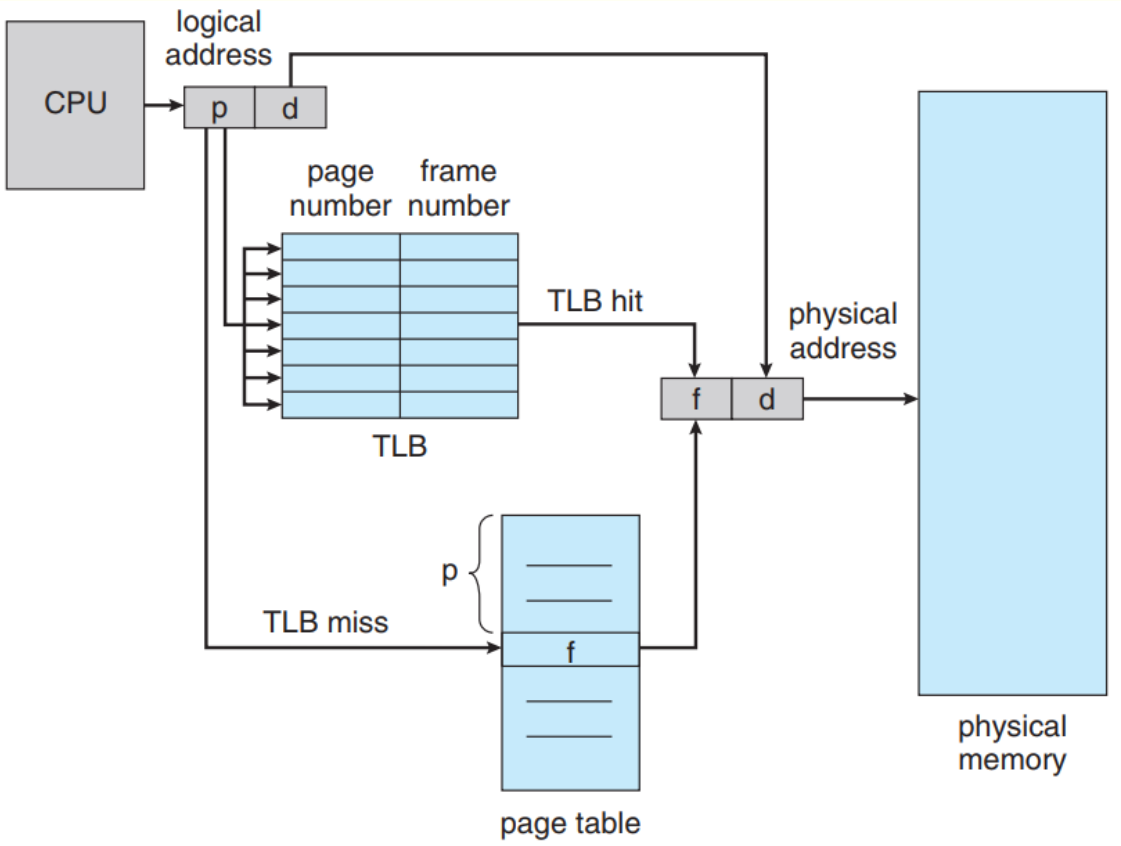
Апаратна підтримка

- Оскільки таблиці сторінок є окремою структурою даних для кожного процесу, вказівник на таблицю зберігається з іншими значеннями регістрів (як instruction pointer) у PCB кожного процесу.
  - Коли планувальник ЦП обирає процес на виконання, він повинен перезавантажити користувацькі регістри та відповідні апаратні page-table значення зі values from the stored user page table.
- Апаратна реалізація таблиці сторінок може виконуватись кількома способами.
  - Найпростіший – як набір виділених (dedicated) високошвидкісних апаратних регістрів, які роблять page-address відображення дуже ефективним.
  - Проте такий підхід нарощує тривалість перемикання контексту, оскільки кожний з цих регістрів потрібно обмінювати в процесі перемикання.
- Використання регістрів для таблиці сторінок задовільне, якщо така таблиця досить мала (наприклад, 256 записів).
  - Більшість сучасних ЦП підтримують набагато більші таблиці (наприклад, 2^{20} записів).
  - У таких випадках краще тримати таблицю сторінок в основній пам’яті, а на таблицю вказуватиме ***регістр таблиці сторінок (page-table base register, PTBR)***.
  - Заміна таблиць сторінок вимагає тільки цього одного регістра, значно знижуючи тривалість перемикання контексту.

Буфер асоціативної трансляції (Translation Look-Aside Buffer)

- Зберігання таблиці сторінок в основній пам'яті може як пришвидшити перемикання контексту, так і сповільнити доступ до пам'яті.
  - Нехай потрібно отримати доступ до адреси *i.*
  - Спочатку перейдемо в таблицю сторінок за індексом, використовуючи значення в PTBR offset за номером таблиці для *i*.
  - Потрібен 1 доступ до пам'яті, який забезпечує номер фрейму, що комбінується з page offset, щоб отримати actual address.
  - Потім можемо отримати доступ до бажаного місця в пам'яті.
  - За цією схемою ***2*** доступи до пам'яті потрібні для доступу до даних (1 – для page-table запису, 1 – для даних).
  - Таким чином, доступ до пам'яті сповільнюється вдвічі, що недопустимо в багатьох випадках.
- Стандартне вирішення цієї проблеми – використовувати спеціальний невеликий кеш – **translation look-aside buffer** (**TLB**).
  - Кожен запис у TLB складається з 2 частин: ключа (тегу) та значення. Пошук елементу відбувається за ключем.
  - TLB lookup у сучасному апаратному забезпеченні є частиною конвеєру команд, практично не знижуючи продуктивність.
  - Проте для виконання пошуку на стадії конвеєра TLB повинен бути малим (зазвичай 32-1024 записи).
  - Деякі ЦП реалізують окремі адреси інструкцій та даних TLB. Це подвоює кількість доступних TLB-записів, оскільки обхід виконується на різних стадіях конвеєра.

Апаратне підкачування сторінок з TLB



- Коли логічна адреса генерується ЦП, блок MMU спочатку перевіряє, чи номер сторінки присутній в TLB.
  - Якщо номер сторінки знайдено, її номер фрейму відразу використовується для доступу до пам'яті.
  - Якщо не знайдено (**TLB miss**), виконується стандартна процедура конструювання посилання на таблицю сторінок у пам'яті.
  - У добавок номер сторінки та номер фрейму додаються в TLB, щоб швидко знаходитись при наступному посиланні.
- Якщо TLB заповнений, обраний існуючий запис потрібно замінити.
  - Політика заміщення варіюється від least recently used (LRU) до round-robin чи random.
  - Деякі ЦП дозволяють ОС брати участь у заміні записів методом LRU, а інші справляються власними силами.
  - Також деякі TLB дозволяють закріпити (**wire down**) деякі записи, тобто ці записи неможливо видалити з TLB. Зазвичай, для записів стосовно коду ядра.

Ідентифікатор адресного простору (address-space identifier, ASID)

- Деякі TLB зберігають ідентифікатор адресного простору у кожному TLB-записі.
  - ASID унікально визначає кожний процес та використовується для захисту адресного простору цього процесу.
  - Коли TLB намагається to resolve номери віртуальних сторінок, він перевіряє, що ASID поточного запущеного процесу відповідає пов’язаний з віртуальною сторінкою ASID. Якщо не відповідає, спроба трактується як TLB miss.
  - Крім захисту адресного простору, ASID дозволяє TLB містити записи кількох різних процесів одночасно.
  - Якщо TLB не підтримує окремі ASID-и, тоді при кожному виборі нової таблиці сторінок (наприклад, з кожним перемиканням контексту), TLB потрібно очистити **(flush)**, щоб наступний процес не використовував застарілу інформацію щодо відображень.

Ефективний час доступу

- Відсоток разів, коли знаходиться шуканий номер сторінки в TLB, називають ***відношенням влучань (hit ratio)***.
  - 80%-ий hit ratio означає, що потрібний номер сторінки знаходився в TLB у 80% випадків.
  - Якщо для доступу до пам’яті треба 10нс, тоді доступ до mapped memory буде 10нс, коли номер сторінки в TLB. Інакше спочатку отримуємо доступ до таблиці сторінок і номеру фрейму (10нс) + доступ до бажаного байту в пам’яті (10нс).
    - Припускаємо, що page-table lookup відбувається лише з одним доступом до пам’яті, проте насправді їх може бути більше.
- Для обчислення ефективного часу доступу до пам’яті використовуємо ймовірності:
  - effective access time = 0.80 × 10 + (1 – 0.80) × 20 = 12нс
- Для 99% влучань матимемо
  - effective access time = 0.99 × 10 + 0.01 × 20 = 10.1нс
  - Маємо лише 1%-ве сповільнення часу доступу.

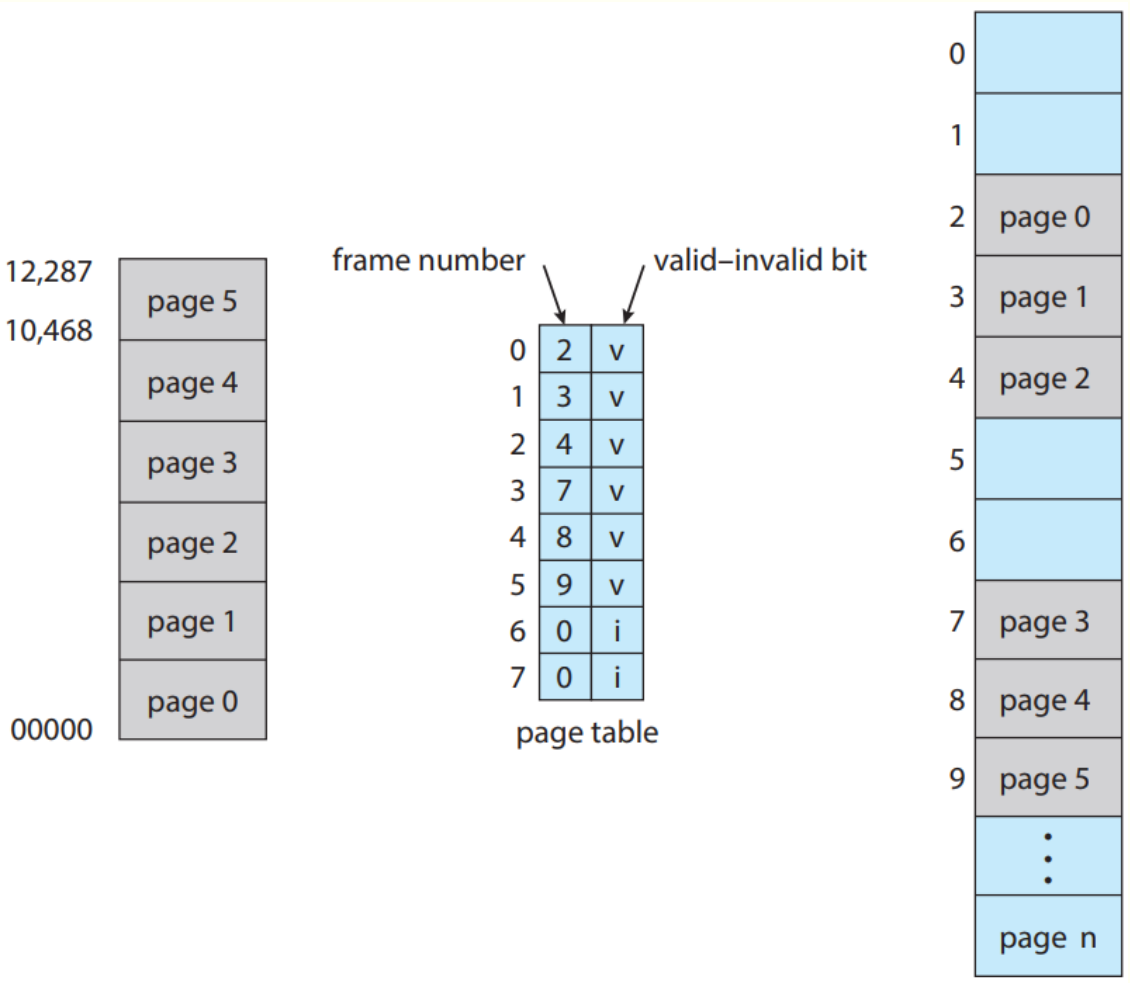
Багаторівневі TLB

- Обчислення часу доступу в сучасних ЦП набагато складніше, ніж у попередньому прикладі.
  - Наприклад, для Intel Core i7 з L1-кешем на 128 записів TLB для інструкцій та 64 записи для даних.
  - Промах на рівні L1 забирає 6 тактів ЦП, щоб перевірити запис з L2-кешу на 512 записів у TLB.
  - Промах на рівні L2 означає, що ЦП повинен пройтись по page-table записах у пам’яті, щоб знайти відповідні адреси фреймів, що може зайняти сотні циклів (не враховуючи переривань від ОС, які відкладуть цю роботу).
  - Повний аналіз швидкодії підкачування сторінок у такій системі потребує інформацію про відсоток промахів (miss-rate information) щодо кожного рівня TLB.
- TLBs є апаратною характеристикою, тому мало турбує ОС та її проектувальників.
  - Проте проектувальникам корисно розуміти функції та характеристики TLBs, які варіюються залежно від апаратної платформи.
  - Для оптимальної роботи архітектура ОС для заданої платформи повинна реалізовувати підкачування сторінок відповідно до TLB-архітектури платформи.

Захист пам’яті

- При сторінковій організації здійснюється за допомогою protection bits, пов’язаних з кожним фреймом. Зазвичай ці біти тримаються в таблиці сторінок.
- Один біт може визначати доступ до сторінки як read–write або read-only.
  - Кожне посилання на пам’ять проходить через таблицю сторінок, щоб знайти коректний номер фрейму.
  - Під час обчислення фізичної адреси можна перевіряти й біти захисту.
  - Спроба запису в сторінку, доступну тільки для читання, спричиняє апаратний виняток в ОС (або memory-protection violation).
- Цей підхід можна розширити для забезпечення кращого рівня захисту.
  - Можна створити апаратне забезпечення для постачання read-only, read–write або execute-only захисту;
  - або забезпечувати окремі захисні біти для кожного виду доступу, комбінуючи їх потім.
  - Некоректні аргументи будуть перехоплюватись ОС.
- Додатковий біт зазвичай прив’язується до кожного запису таблиці сторінок: **valid–invalid** біт.
  - Коли біт має значення «*valid»,* відповідна сторінка знаходиться в логічному адресному просторі процесу. Інакше – «*invalid».*
  - Нелегальні адреси перехоплюються згідно з даним бітом. ОС встановлює цей біт для кожної сторінки, щоб керувати доступом до неї.

Valid (v) або invalid (i) біт у таблиці сторінок

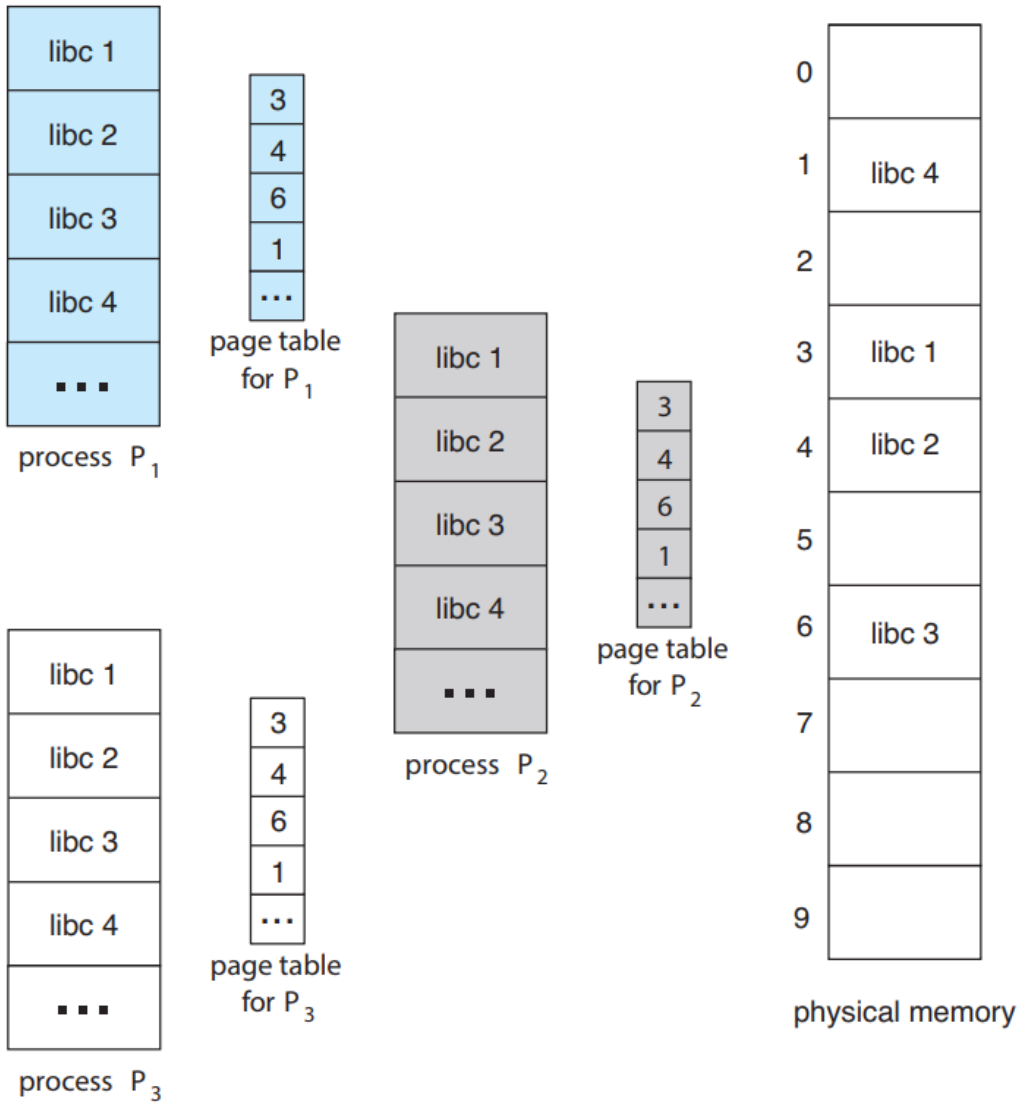


- Нехай у системі з 14-бітними адресами (від 0 до 16383) маємо програму, яка має використовувати лише адреси від 0 до 10468.
  - Для сторінок розміром 2 Кб ситуація зображена на рисунку.
  - Адреси в сторінках 0, 1, 2, 3, 4, 5 відображаються нормально.
  - Будь-яка спроба оперувати адресами зі сторінок 6 чи 7 призведе до винятку (invalid page reference).
- Дана схема створила проблему.
  - Усі адреси після 10468 нелегальні, проте в останній сторінці доступні адреси включно до 12287.
  - Дана проблема відображає внутрішню фрагментацію при розмірі сторінок 2Кб.

Дуже рідко процес використовує весь свій адресний простір

- Багато процесів використовують лише маленьку частину доступного їм адресного простору.
  - Створювати таблицю сторінок із записом для кожної сторінки – затратно, оскільки більшість з них не будуть використовуватись, проте займатимуть місце.
  - Деякі системи постачають апаратне забезпечення у формі ***регістру довжини таблиці сторінок (page-table length register, PTLR)***, який містить розмір таблиці сторінок.
  - Це значення звіряється з кожною логічною адресою, щоб перевірити, чи вона знаходиться в легальному діапазоні для процесу.
  - Відмова такого тесту спричиняє виняток в роботі ОС.

Shared Pages



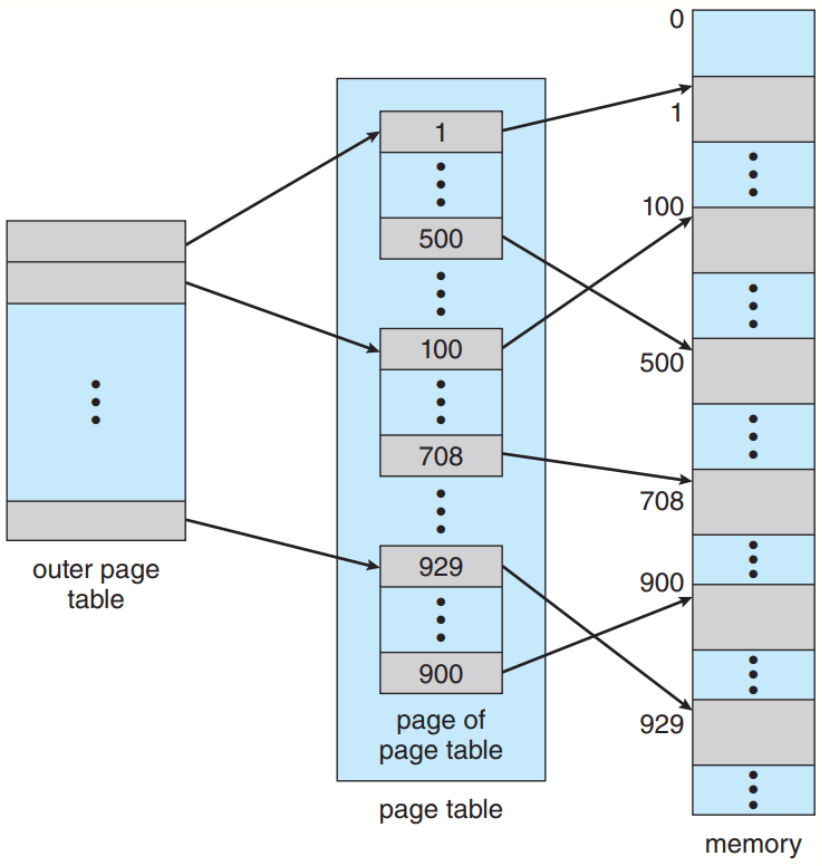
Перевага сторінкової організації пам'яті – можливість ділитись спільним кодом.

- На рисунку бачимо 3 процеси, які мають спільні сторінки для стандартної бібліотеки libc (хоч на риснку libc займає 4 сторінки, насправді їх більше).
- Код з можливістю повторного входу (Reentrant code) є кодом без можливості самомодифікації: він ніколи не змінюється протягом виконання.
- Кожен процес має власну копію регістрів та сховище даних для зберігання даних для виконання процесу.
- Лише одну копію стандартної бібліотеки мови С потрібно зберігати Iв фізичній пам'яті, а таблиця сторінок кожного процесу вказує на одну фізичну копію libc.
- Додатково до run-time бібліотек, на зразок libc, інші активно використовувані програми можуть шеритись: компілятори, віконні системи, бази даних тощо.

Структура таблиці сторінок

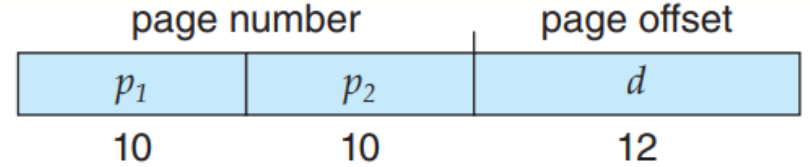
- Більшість сучасних комп'ютерних систем підтримують великі логічні адресні простори (від 2^{32} до 2^{64}). Таблиця сторінок в них стає надто великою.
  - Наприклад, розглянемо систему з 32-бітним логічним адресним простором.
  - Якщо розмір сторінки в такій системі буде 4Кб (2^{12}), тоді таблиця сторінок може містити понад мільйон записів ($2^{20} = 2^{32}/2^{12}$).
- Припускаючи, що кожний запис складається з 4 байтів, кожному процесу знадобиться до 4Мб фізичного адресного простору лише для самої таблиці.
  - Очевидно, що таблиці сторінок небажано розміщувати безперервно в основній пам'яті.
  - Просте рішення – розбивати таблиці сторінок на менші частини.

Ієрархічне підкачування сторінок. Дворівнева page-table схема

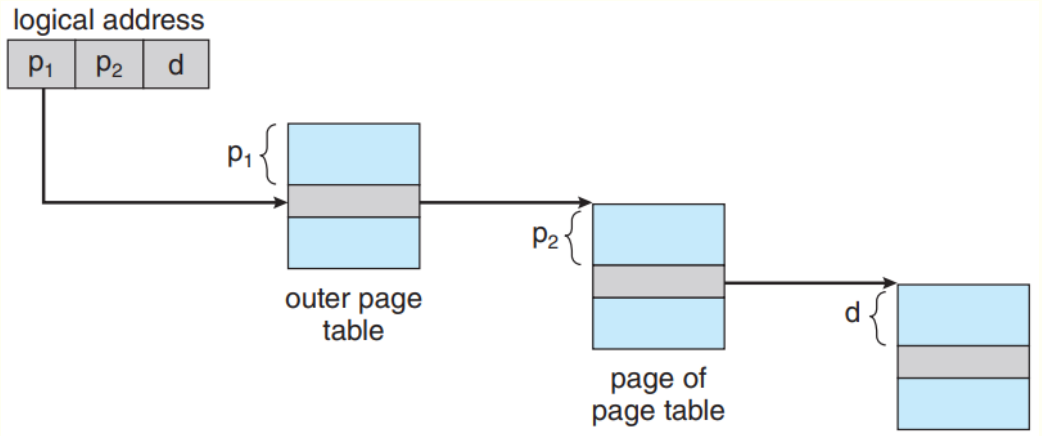


Один зі способів – використовувати дворівневий алгоритм підкачування сторінок, у якому таблиця сторінок також підкачується.

- Наприклад, для попередньої системи логічна адреса розбивається на номер сторінки (20 бітів) та зміщення сторінки (12 бітів).
- При підкачуванні таблиці сторінок відбувається подальший поділ: 10-бітовий номер сторінки та 10-бітове зміщення.:



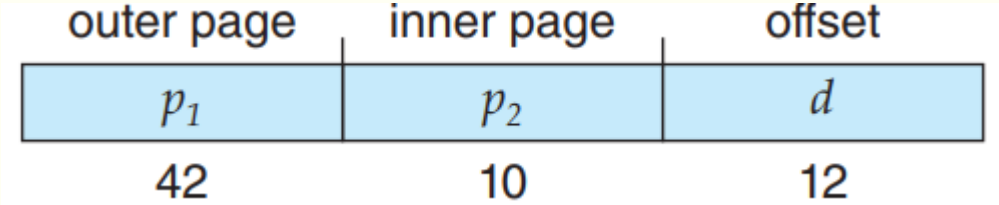
- де p_1 – індекс у зовнішній таблиці сторінок, а p_2 – зміщення всередині сторінки внутрішньої таблиці сторінок.



Дана схема також відома як **forward-mapped** page table.

Для 64-бітної системи дворівнева схема вже неприйнятна

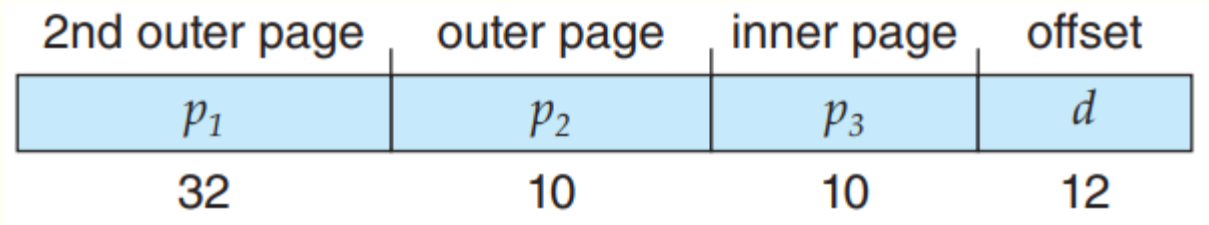
- Для сторінки розмірок 4Кб (2^{12}) таблиця сторінок міститиме до 2^{52} записів.
  - За умови використання дворівневої схеми підкачування внутрішня таблиця сторінок може зручно представлятись однією великою сторінкою або набором з 2^{10} 4-байтних записів:



- Зовнішня таблиця сторінок містить 2^{42} записів, або 2^{44} байтів.
  - Очевидний спосіб уникнути проблему – розділити зовнішню таблицю на менші частини.
  - Такий підхід інколи використовується і в 32-бітних процесорах, додаючи гнучкості та ефективності.

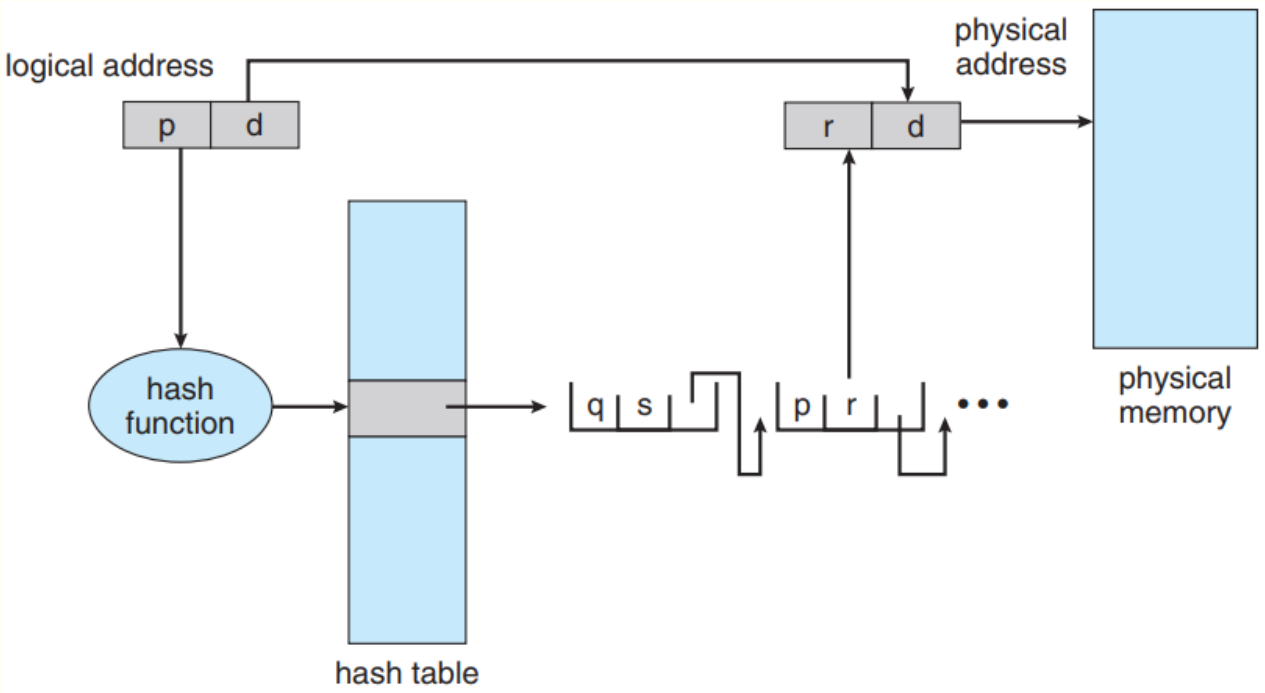
Розділити зовнішню таблицю сторінок можна по-різному

- Наприклад, можемо підкачувати зовнішню таблицю сторінок за допомогою трирівневої схеми.
  - Нехай зовнішня таблиця сторінок складається зі сторінок стандартного розміру (2^{10} записів, або 2^{12} байтів). 64-бітний адресний простір виглядає лячно:



  - Зовнішня таблиця сторінок все ще має розмір 2^{34} байтів (16Гб).
- Наступний крок – чотирирівнева система, де другорівнева зовнішня таблиця сторінок також підкачується, і т. д.
  - 64-бітний процесор UltraSPARC вимагає 7 рівнів підкачування сторінок для трансляції кожної логічної адреси.
  - Саме тому ієрархічні таблиці сторінок недоречні для 64-бітних архітектур.

Хешовані таблиці сторінок

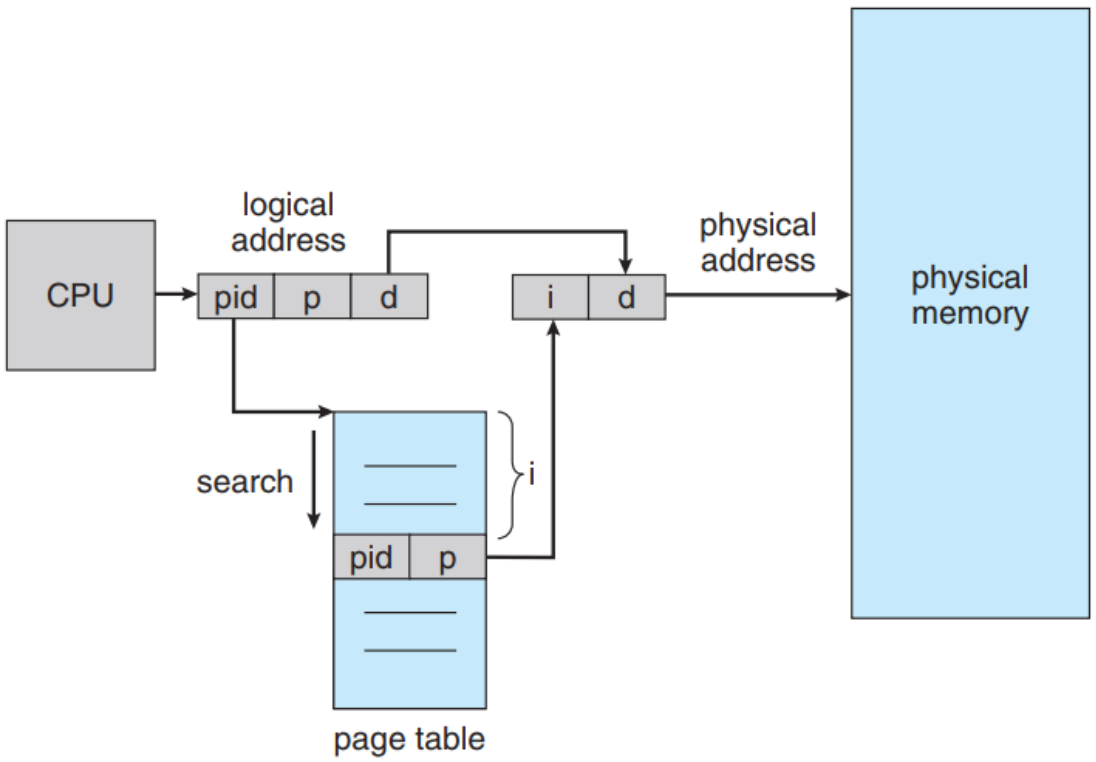


- Значенням хешу буде номер віртуальної сторінки.
  - Вирішення колізій на базі зв'язних списків.
  - Кожен елемент складається з трьох полів:
  - (1) номер віртуальної сторінки, (2) значення відображеного page frame, (3) вказівник на наступний елемент у зв'язному списку.
- Алгоритм: номер віртуальної сторінки у віртуальній адресі хешується в хеш-таблицю.
  - Номер віртуальної сторінки порівнюється з полем 1 у першому елементі зв'язного списку.
  - Якщо співпадіння немає, перевіряємо далі по списку.
- Варіація даної схеми для 64-бітних адресних просторів використовує кластеризовані таблиці сторінок
  - кожен запис у хеш-таблиці відповідає кільком сторінкам, наприклад, 16ти.

Інвертовані таблиці сторінок

- Зазвичай кожен процес має пов’язану з ним таблицю сторінок.
  - Вона має один запис на кожну сторінку, що використовує процес (один слот на віртуальну адресу, незалежно від її визначеної далі валідності).
  - Потім ОС повинна транслювати дане посилання в адреси фізичної пам’яті.
  - Оскільки таблиця відсортована за віртуальними адресами, ОС зможе обчислити, де в таблиці розташовано відповідну фізичну адресу та використати значення напряму.
  - Один з недоліків методу – кожна таблиця сторінок може містити мільйони записів. Такі таблиці можуть споживати багато ресурсів лише для відстеження використання іншої фізичної пам’яті.
- Для вирішення проблеми можна застосувати ***інвертовану таблицю сторінок***.
  - Вона має один запис для кожної реальної сторінки (або фрейму) пам’яті.
  - Кожний запис складається з віртуальної адреси сторінки, що збережена в цій дійсній пам’яті, з інформацією про процес-власник сторінки.
  - Таким чином, маємо лише 1 таблицю сторінок в системі, а вона має лише 1 запис для кожної сторінки фізичної пам’яті.

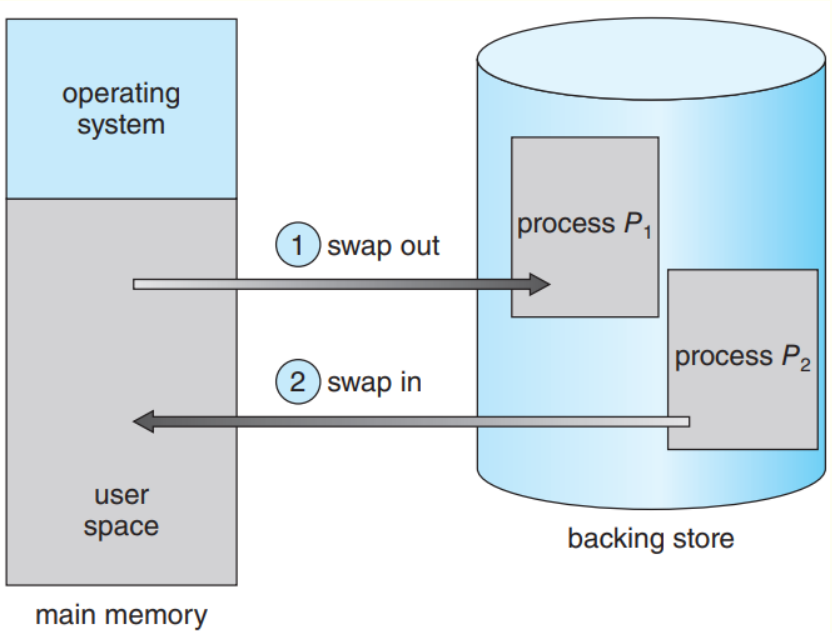
Інвертовані таблиці сторінок



- Часто вимагають, щоб ASID зберігався в кожному записі таблиці сторінок, оскільки таблиця зазвичай містить кілька різних адресних просторів, відображених у фізичну пам'ять.
  - Зберігання ASID забезпечує відображення логічної сторінки на відповідний фізичний page frame.
  - Приклади систем, що використовуються інвертовані таблиці - 64-бітний UltraSPARC та PowerPC.
- Для ілюстрації опишемо спрощену версію інвертованої таблиці сторінок, що використовується в IBM RT.
  - Кожна віртуальна адреса в системі складається з трійки: <process-id, page-number, offset>.
  - Кожний запис інвертованої таблиці сторінок є парою <process-id, page-number>, де process-id виконує роль ASID.
  - Коли трапляється посилання на пам'ять, частина віртуальної адреси, що містить пару <process-id, page-number>, представляється підсистемі пам'яті.
  - В інвертованій таблиці сторінок відбувається зіставлення.
  - Якщо знайдено співпадіння (наприклад, запис *i*), тоді фізична адреса <*i,* offset> генерується.
  - Якщо не знайдено, тоді відбувається доступ за нелегальною адресою.

- Хоч дана схема зменшує обсяг пам’яті, потрібний для зберігання кожної таблиці сторінок, вона сповільнює пошук таблиці, коли трапляється посилання на сторінку (page reference).
  - Оскільки інвертована таблиця сторінок відсортована за фізичними адресами, проте пошук здійснює за віртуальними, цілу таблицю може знадобитись обшукати перед знаходженням потрібних адрес (надто довго).
  - Для зменшення проблеми використаємо хеш-таблицю, щоб обмежити пошук до кількох page-table записів.
  - Звісно, кожний доступ до хеш-таблиці додає memory reference на процедуру, тому посилання на одну віртуальну адресу вимагає принаймні 2 дійсних зчитувань пам’яті – для запису хеш-таблиці та для таблиці сторінок (також зауважте, що пошук в TLB відбувається до звернення до хеш-таблиці).
- Одна проблема інвертованих таблиць сторінок задіює спільну пам’ять.
  - При стандартній сторінковій організації кожен процес має власну таблицю сторінок, що дозволяє кільком віртуальним адресам відображатись на одну фізичну адресу.
  - Цей метод не працює з інвертованими таблицями сторінок; оскільки існує лише 1 запис віртуальної сторінки для кожної фізичної сторінки.
  - Таким чином, для інвертованих таблиць сторінок тільки 1 відображення віртуальної адреси на спільну фізичну адресу може трапитись в даний момент. Посилання від іншого процесу призведе до відмови сторінки та замінить відображення іншою віртуальною адресою.

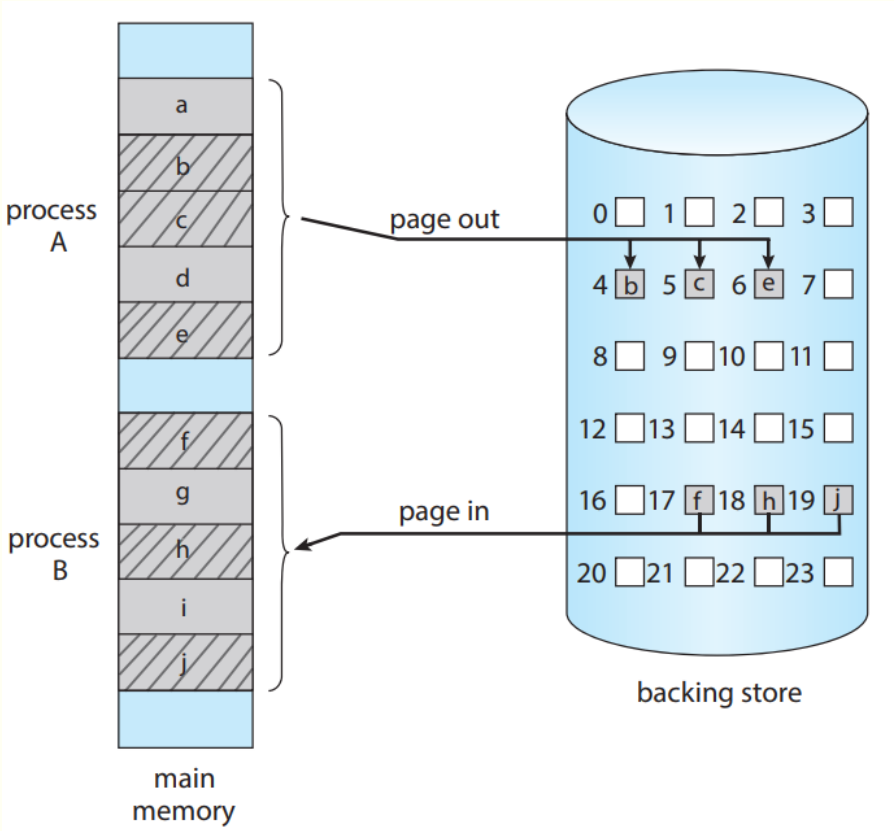
Свопінг (swapping)



- Інструкції процесу та дані, якими вони оперують, повинні бути в пам'яті для виконання.
  - Проте процес чи його частина може тимчасово вивантажуватись (**swap out**) з пам'яті у ***файл відкачки (backing store)***, а потім повернутись назад для продовження роботи.
  - Свопінг дозволяє фізичному адресному простору всіх процесів перевищувати обсяг реальної фізичної пам'яті в системі.
- Стандартний свопінг задіює переміщення цілих процесів між основною пам'яттю та файлом відкачки (на відносно швидкому вторинному сховищі).
  - Цей файл повинен бути достатньо великий, щоб to accommodate whatever parts of processes need to be stored and retrieved, and it must provide direct access to these memory images.
  - Коли процес або його частина вивантажуються на файл відкачки, пов'язані структури даних також викачуються. Для багатопоточних процесів усі дані на потік також вивантажуються.
  - ОС повинна також підтримувати мета-дані для процесу, що було вивантажено, тому можливе відновлення викачаного процесу назад у пам'ять.

- Перевага стандартного свопінгу – можливість фізичної пам'яті to be oversubscribed, щоб система могла accommodate більше процесів, ніж може вмістити реальна фізична пам'ять.
  - Бездіяльні (Idle) або практично бездіяльні процеси є хорошими кандидатами на вивантаження.
  - Якщо вивантажений неактивний процес знову стає активним, його необхідно завантажити назад.

Свопінг зі сторінковою організацією пам’яті



- Стандартний свопінг використовувався в традиційних UNIX-системах, проте переважно не використовується в сучасних ОС.
  - Причина: обсяг часу, потрібний для переміщення цілих процесів між пам’яттю та файлом відкачки.
  - Винятком є Solaris, яка все ще використовує стандартний свопінг, проте лише в спеціальних ситуаціях, коли пам’яті надзвичайно мало.
- Більшість систем, включаючи Linux та Windows, використовують варіант свопінгу, в якому сторінки процесу, а не цілий процес, може вивантажуватись.
  - Дана стратегія все ще дозволяє фізичній пам’яті to be oversubscribed, but does not incur the cost of swapping entire processes, as presumably only a small number of pages will be involved in swapping.
  - Фактично, термін ***swapping*** тепер відноситься до стандартного свопінгу, а ***paging*** – свопінг зі сторінковою організацією пам’яті.
  - Операція **page out** переміщає сторінку з пам’яті у файл відкачки; зворотна операція відома як **page in**.

Свопінг на мобільних системах

- Більшість ОС для ПК та серверів підтримують підкачування сторінок.
  - Мобільні ОС навпаки не підтримують свопінг в жодній формі, оскільки мобільні пристрої переважно використовують флеш-пам’ять в якості сховища.
  - Обмеження вбудованої пам’яті – ще одна причина уникання свопінгу.
  - Інші причини: обмежена кількість циклів запису флеш-пам’яті, низька пропускна здатність між основною пам’яттю та флеш-пам’яттю.
- Замість свопінгу переважно використовується добровільне або насильне звільнення виділеної пам’яті.
  - Read-only дані (зокрема, код) видаляються з основної пам’яті, а потім повторно завантажуються з флеш-пам’яті за потреби.
  - Змінювані дані (наприклад, стек) ніколи не видаляються.
  - Проте багато додатків, які не можуть звільнити достатню кількість пам’яті, може переривати ОС.