Chương 7: Quản lý bộ nhớ

- Các kiểu địa chỉ nhớ
- Chuyển đổi địa chỉ nhớ
- Overlay và swapping
- Mô hình quản lý bộ nhớ đơn giản
  - Fixed partitioning
  - Dynamic partitioning
  - Cơ chế phân trang (paging)
  - Cơ chế phân đoạn (segmentation)
  - Segmentation with paging
- Bộ nhớ ảo

Khái niệm cơ sở

- Quản lý bộ nhớ là công việc của hệ điều hành với sự hỗ trợ của phần cứng nhằm phân phối, sắp xếp các process trong bộ nhớ sao cho hiệu quả.
- Mục tiêu cần đạt được là nạp càng nhiều process vào bộ nhớ càng tốt (gia tăng mức độ đa chương)
- Trong hầu hết các hệ thống, kernel sẽ chiếm một phần cố định của bộ nhớ; phần còn lại phân phối cho các process.
- Các yêu cầu đối với việc quản lý bộ nhớ
  - Cấp phát bộ nhớ cho các process
  - Tái định vị (relocation): khi swapping,...
  - Bảo vệ: phải kiểm tra truy xuất bộ nhớ có hợp lệ không
  - Chia sẻ: cho phép các process chia sẻ vùng nhớ chung
  - Kết gán địa chỉ nhớ logic của user vào địa chỉ thực

Các kiểu địa chỉ nhớ

- *Địa chỉ vật lý* (physical address) (địa chỉ *thực*) là một vị trí thực trong bộ nhớ chính.

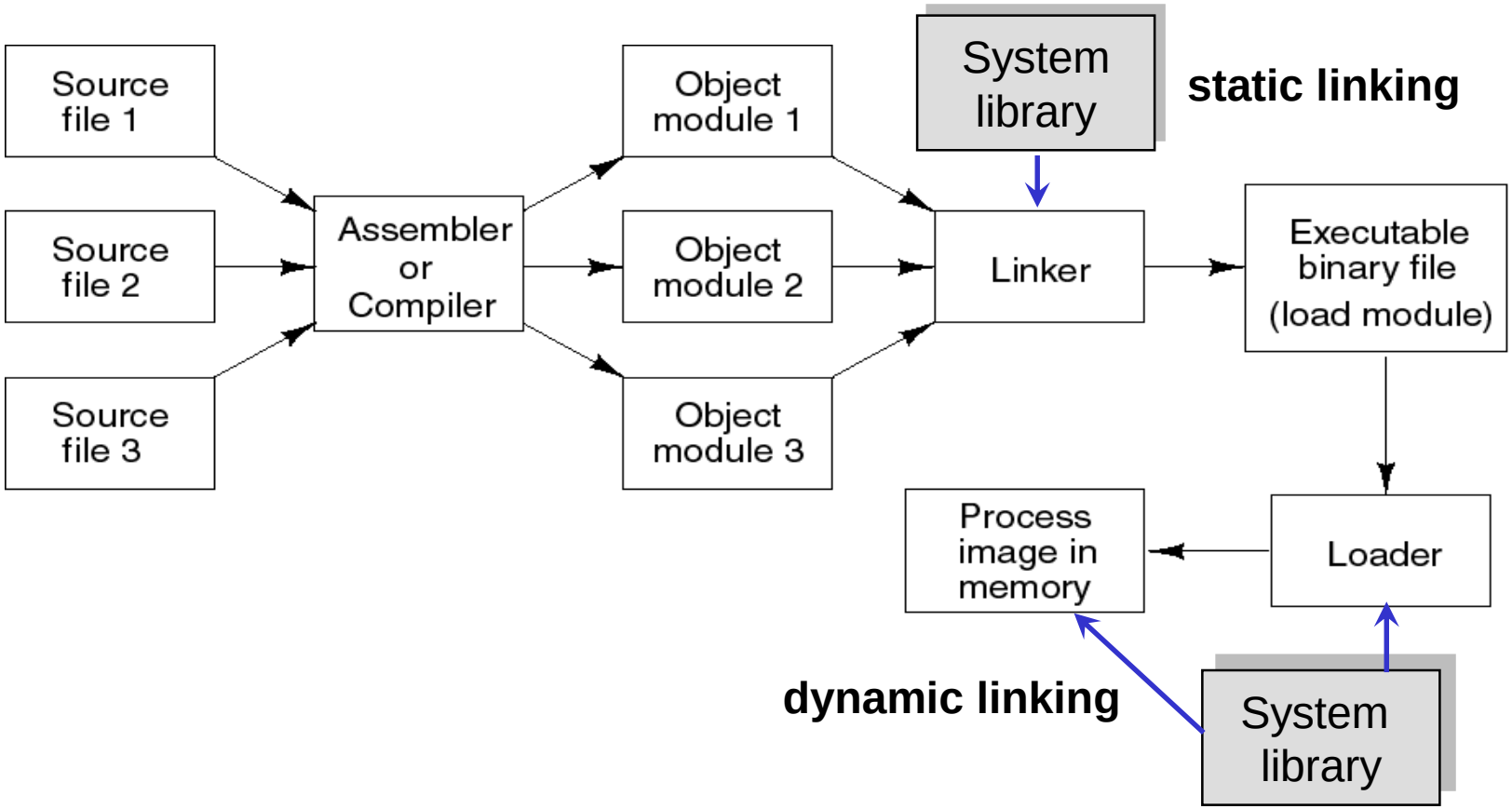
- *Địa chỉ logić* (logical address) là một vị trí nhớ được diễn tả trong một chương trình
  - Các trình biên dịch (compiler) tạo ra mã lệnh chương trình mà trong đó mọi tham chiếu bộ nhớ đều là địa chỉ logić
  - *Địa chỉ tương đối* (relative address) (địa chỉ *khả tái định vị*, relocatable address) là một kiểu địa chỉ logić trong đó các địa chỉ được biểu diễn tương đối so với một vị trí xác định nào đó trong chương trình.
    - Ví dụ: 12 byte so với vị trí bắt đầu chương trình,...
  - *Địa chỉ tuyệt đối* (absolute address): địa chỉ tương đương với địa chỉ thực.

Các kiểu địa chỉ nhớ (tt)

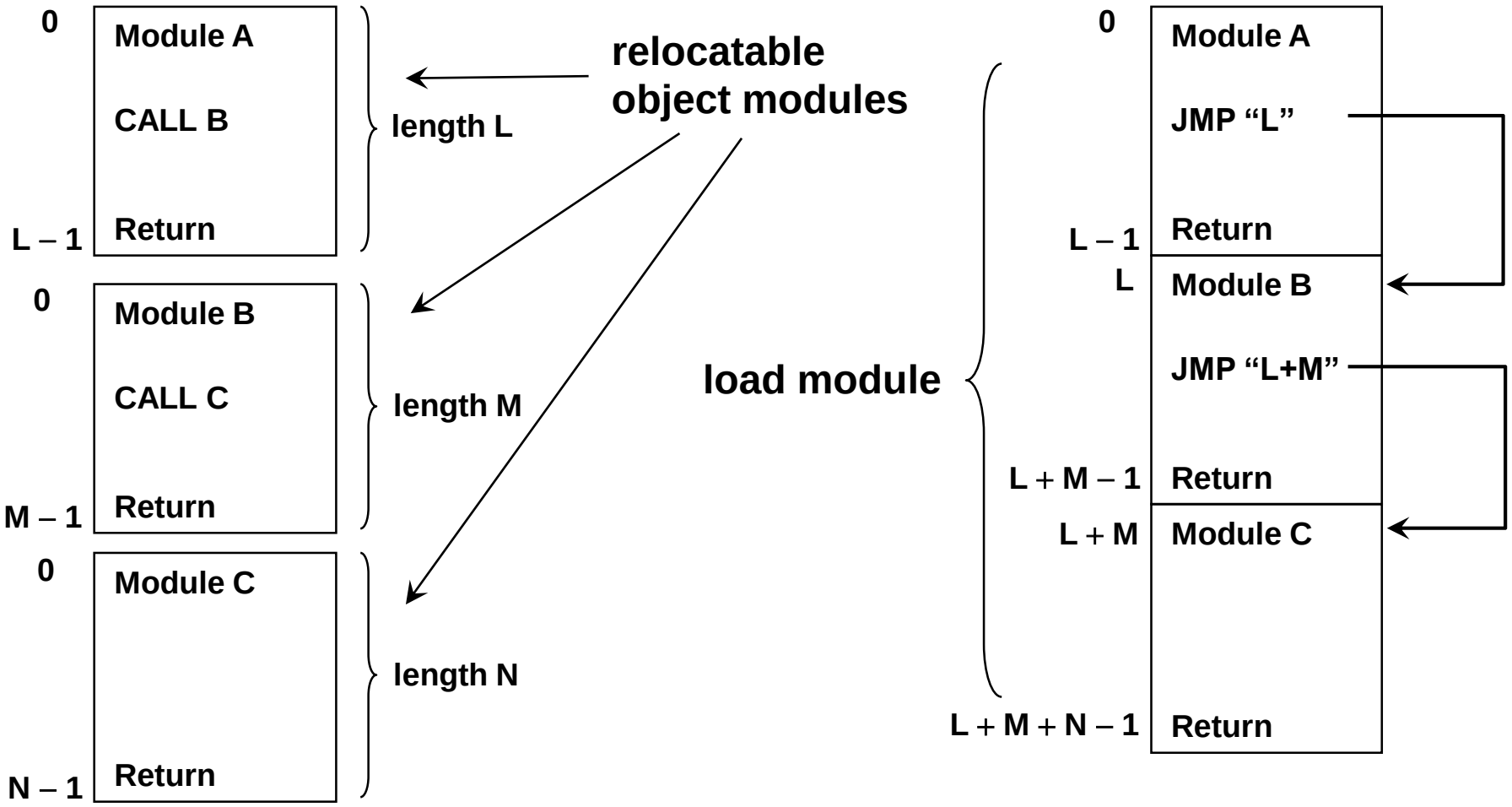
- Khi một lệnh được thực thi, các tham chiếu đến địa chỉ logic phải được chuyển đổi thành địa chỉ thực. Thao tác chuyển đổi này thường có sự hỗ trợ của phần cứng để đạt hiệu suất cao.

Nạp chương trình vào bộ nhớ

- Bộ linker: kết hợp các object module thành một file nhị phân khả thực thi gọi là *load module*.
- Bộ loader: nạp load module vào bộ nhớ chính

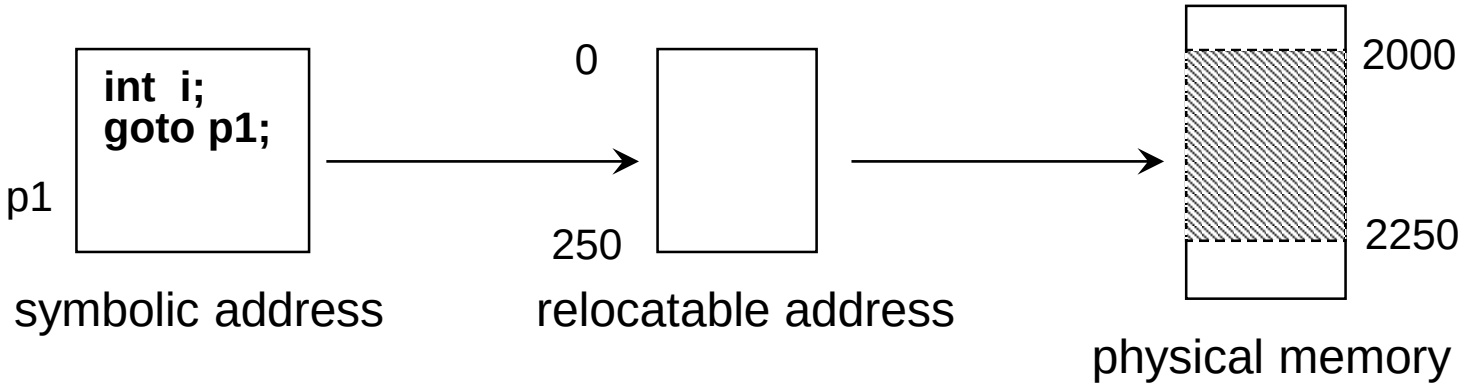


Cơ chế thực hiện linking



Chuyển đổi địa chỉ

- *Chuyển đổi địa chỉ*: tiến trình ánh xạ một địa chỉ từ không gian địa chỉ này sang không gian địa chỉ khác.
- Biểu diễn địa chỉ nhớ
  - Trong source code: symbolic (các biến, hằng, pointer,...)
  - Thời điểm biên dịch: thường là địa chỉ khả tái định vị
    - Ví dụ: a ở vị trí 14 bytes so với vị trí bắt đầu của module.
  - Thời điểm linking/loading: có thể là địa chỉ thực. Ví dụ: dữ liệu nằm tại địa chỉ bộ nhớ thực 2030

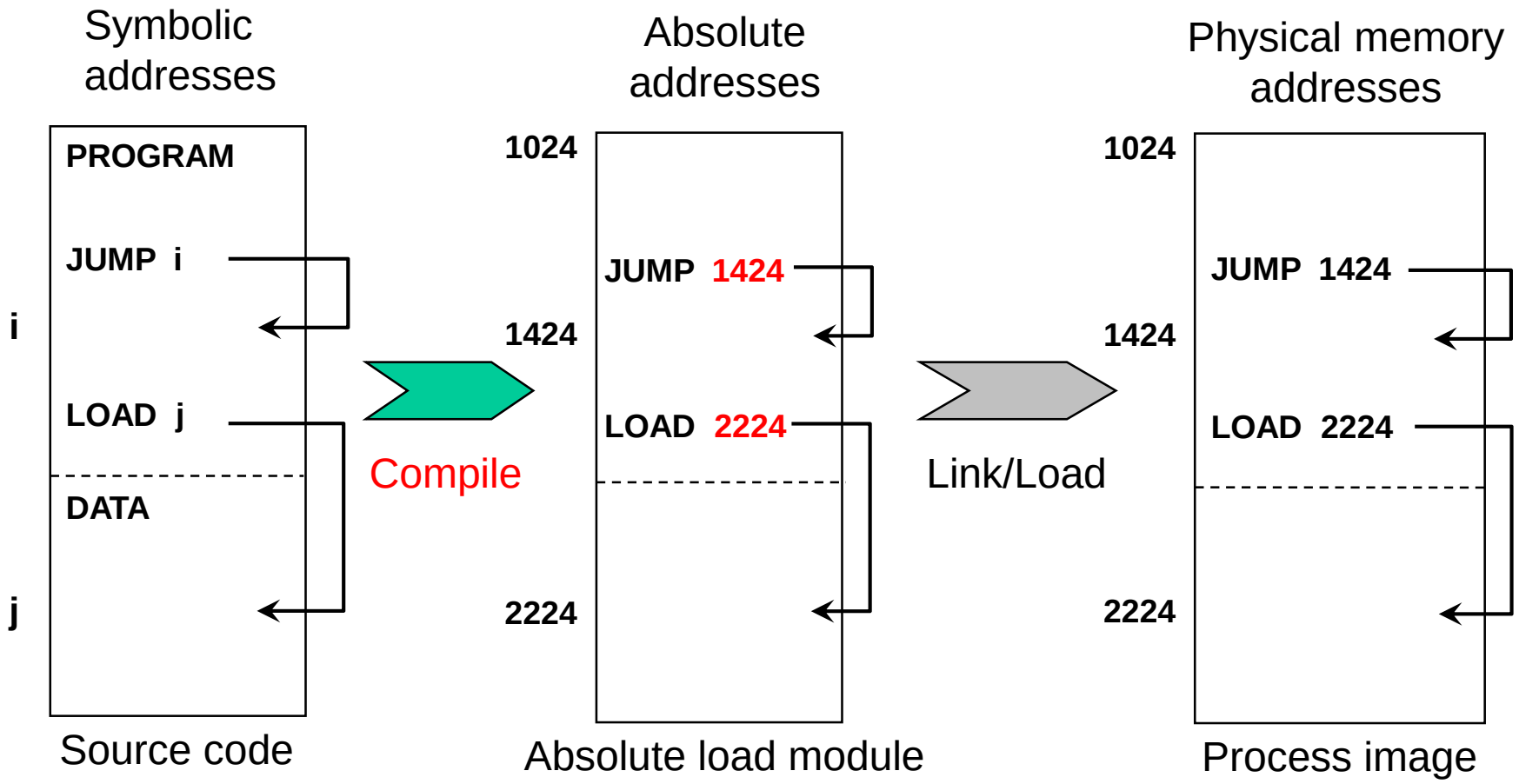


Chuyển đổi địa chỉ (tt)

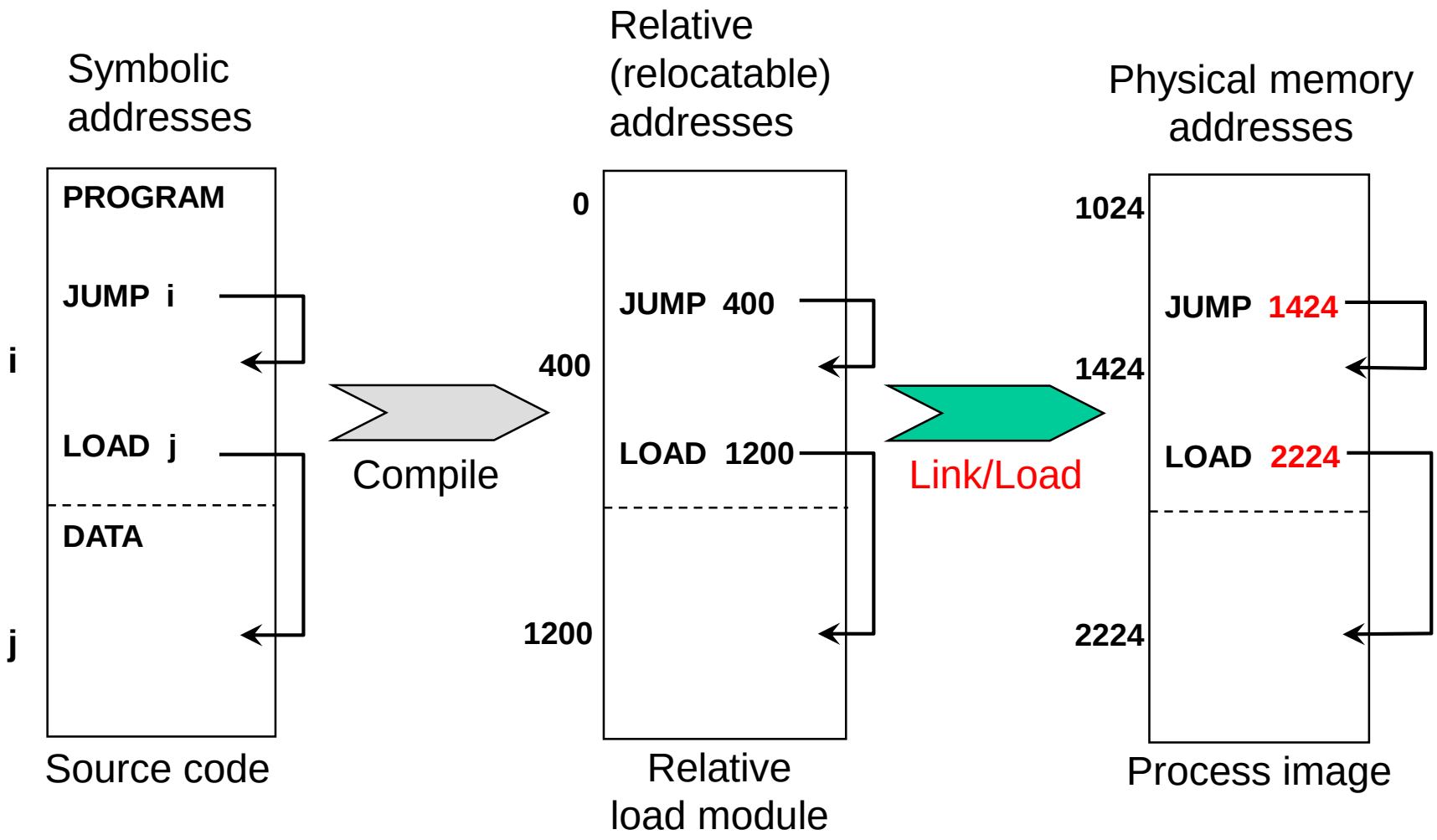
- Địa chỉ lệnh (instruction) và dữ liệu (data) được chuyển đổi thành địa chỉ thực có thể xảy ra tại ba thời điểm khác nhau
  - Compile time: nếu biết trước địa chỉ bộ nhớ của chương trình thì có thể kết gán địa chỉ tuyệt đối lúc biên dịch.
    - Ví dụ: chương trình .COM của MS-DOS, phát biểu assembly

      org xxx
    - Khuyết điểm: phải biên dịch lại nếu thay đổi địa chỉ nạp chương trình
  - Load time: tại thời điểm biên dịch, nếu chưa biết tiến trình sẽ nằm ở đâu trong bộ nhớ thì compiler phải sinh mã khả tái định vị. Vào thời điểm loading, *loader* phải chuyển đổi địa chỉ khả tái định vị thành địa chỉ thực dựa trên một *địa chỉ nền* (base address).
    - Địa chỉ thực được tính toán vào thời điểm nạp chương trình $\Rightarrow$ phải tiến hành reload nếu địa chỉ nền thay đổi.

Sinh địa chỉ tuyệt đối vào thời điểm dịch

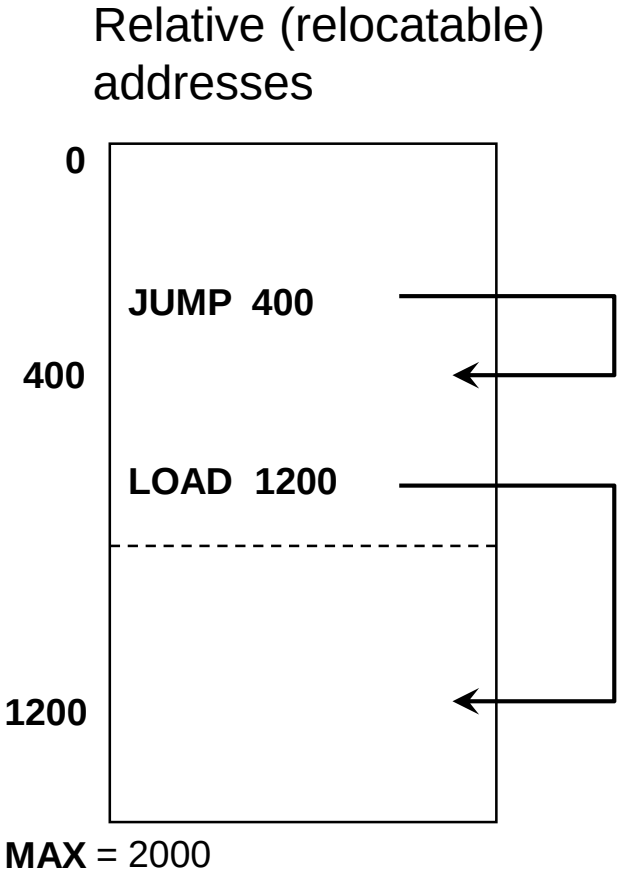


Sinh địa chỉ thực vào thời điểm nạp



Chuyển đổi địa chỉ (tt)

- Execution time: khi trong tiến trình thực thi, process có thể được di chuyển từ segment này sang segment khác trong bộ nhớ thì tiến trình chuyển đổi địa chỉ được trì hoãn đến thời điểm thực thi
  - CPU tạo ra địa chỉ logic cho process
  - Cần sự hỗ trợ của phần cứng cho việc ánh xạ địa chỉ.
    - Ví dụ: trường hợp địa chỉ logic là relocatable thì có thể dùng thanh ghi base và limit,...
  - Sử dụng trong đa số các OS đa dụng (general-purpose) trong đó có các cơ chế swapping, paging, segmentation



Dynamic linking

- tiến trình link đến một *module ngoài* (external module) được thực hiện sau khi đã tạo xong load module (i.e. file có thể thực thi, executable)
  - Ví dụ trong Windows: module ngoài là các file .DLL còn trong Unix, các module ngoài là các file .so (shared library)
- Load module chứa các stub tham chiếu (refer) đến routine của external module.
  - Lúc thực thi, khi stub được thực thi lần đầu (do process gọi routine lần đầu), stub nạp routine vào bộ nhớ, tự thay thế bằng địa chỉ của routine và routine được thực thi.
  - Các lần gọi routine sau sẽ xảy ra bình thường
- Stub cần sự hỗ trợ của OS (như kiểm tra xem routine đã được nạp vào bộ nhớ chưa).

Ưu điểm của dynamic linking

- Thông thường, external module là một thư viện cung cấp các tiện ích của OS. Các chương trình thực thi có thể dùng các phiên bản khác nhau của external module mà không cần sửa đổi, biên dịch lại.
- *Chia sẻ mã* (code sharing): một external module chỉ cần nạp vào bộ nhớ một lần. Các process cần dùng external module này thì cùng chia sẻ đoạn mã của external module ⇒ tiết kiệm không gian nhớ và đĩa.
- Phương pháp dynamic linking cần sự hỗ trợ của OS trong việc kiểm tra xem một thủ tục nào đó có thể được chia sẻ giữa các process hay là phần mã của riêng một process (bởi vì chỉ có OS mới có quyền thực hiện việc kiểm tra này).

Dynamic loading

- **Cơ chế**: chỉ khi nào cần được gọi đến thì một thủ tục mới được nạp vào bộ nhớ chính ⇒ tăng độ hiệu dụng của bộ nhớ (memory utilization) bởi vì các thủ tục không được gọi đến sẽ không chiếm chỗ trong bộ nhớ

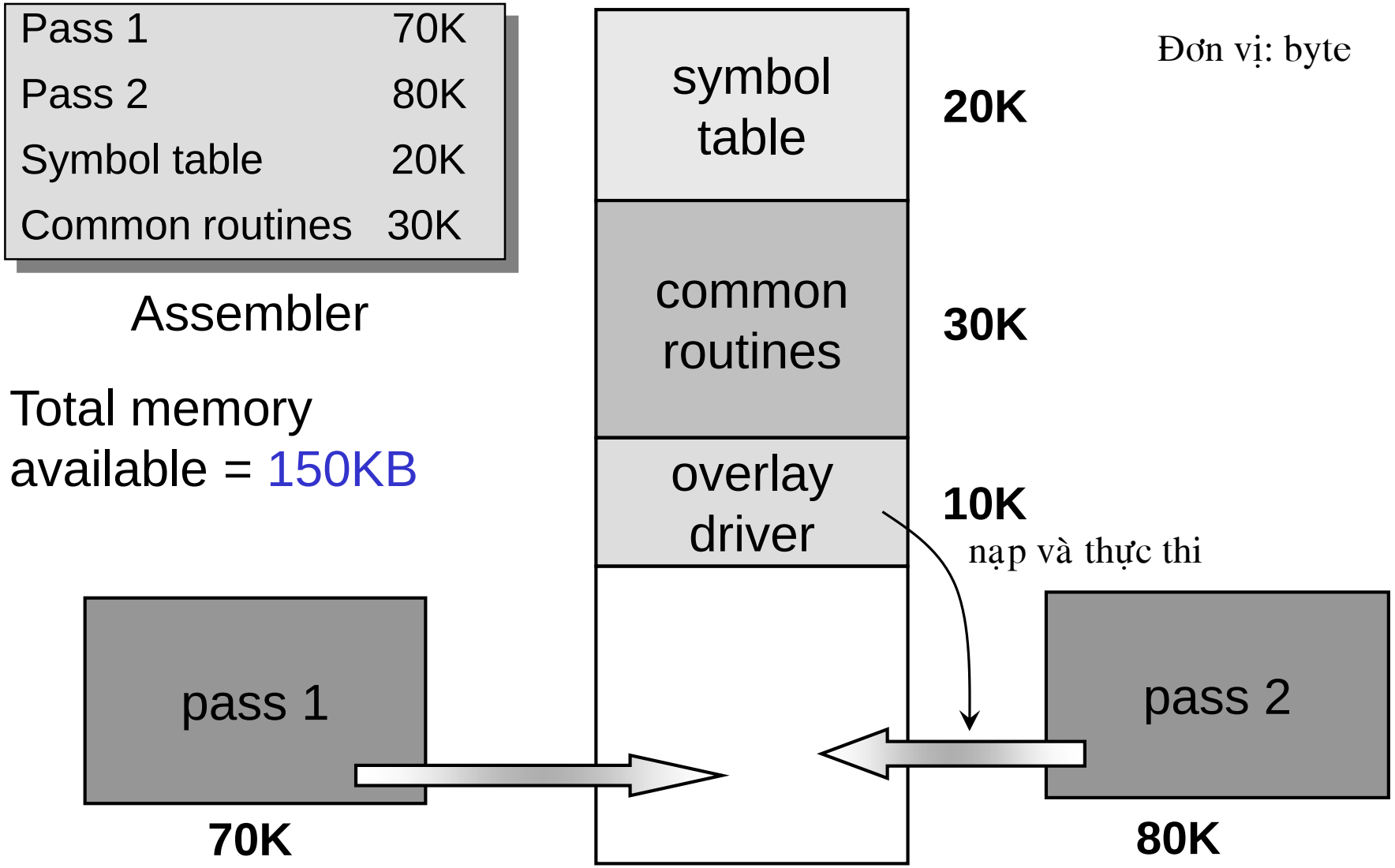
- Rất hiệu quả trong trường hợp tồn tại khối lượng lớn mã chương trình có tần suất sử dụng thấp, không được sử dụng thường xuyên (ví dụ các thủ tục xử lý lỗi)

- Hỗ trợ từ hệ điều hành
  - Thông thường, user chịu trách nhiệm thiết kế và hiện thực các chương trình có dynamic loading.
  - Hệ điều hành chủ yếu cung cấp một số thủ tục thư viện hỗ trợ, tạo điều kiện dễ dàng hơn cho lập trình viên.

Cơ chế overlay

- Tại mỗi thời điểm, chỉ giữ lại trong bộ nhớ những lệnh hoặc dữ liệu cần thiết, giải phóng các lệnh/dữ liệu chưa hoặc không cần dùng đến.
- Cơ chế này rất hữu dụng khi kích thước một process lớn hơn không gian bộ nhớ cấp cho process đó.
- Cơ chế này được điều khiển bởi người sử dụng (thông qua sự hỗ trợ của các thư viện lập trình) chứ không cần sự hỗ trợ của hệ điều hành

Cơ chế overlay (tt)



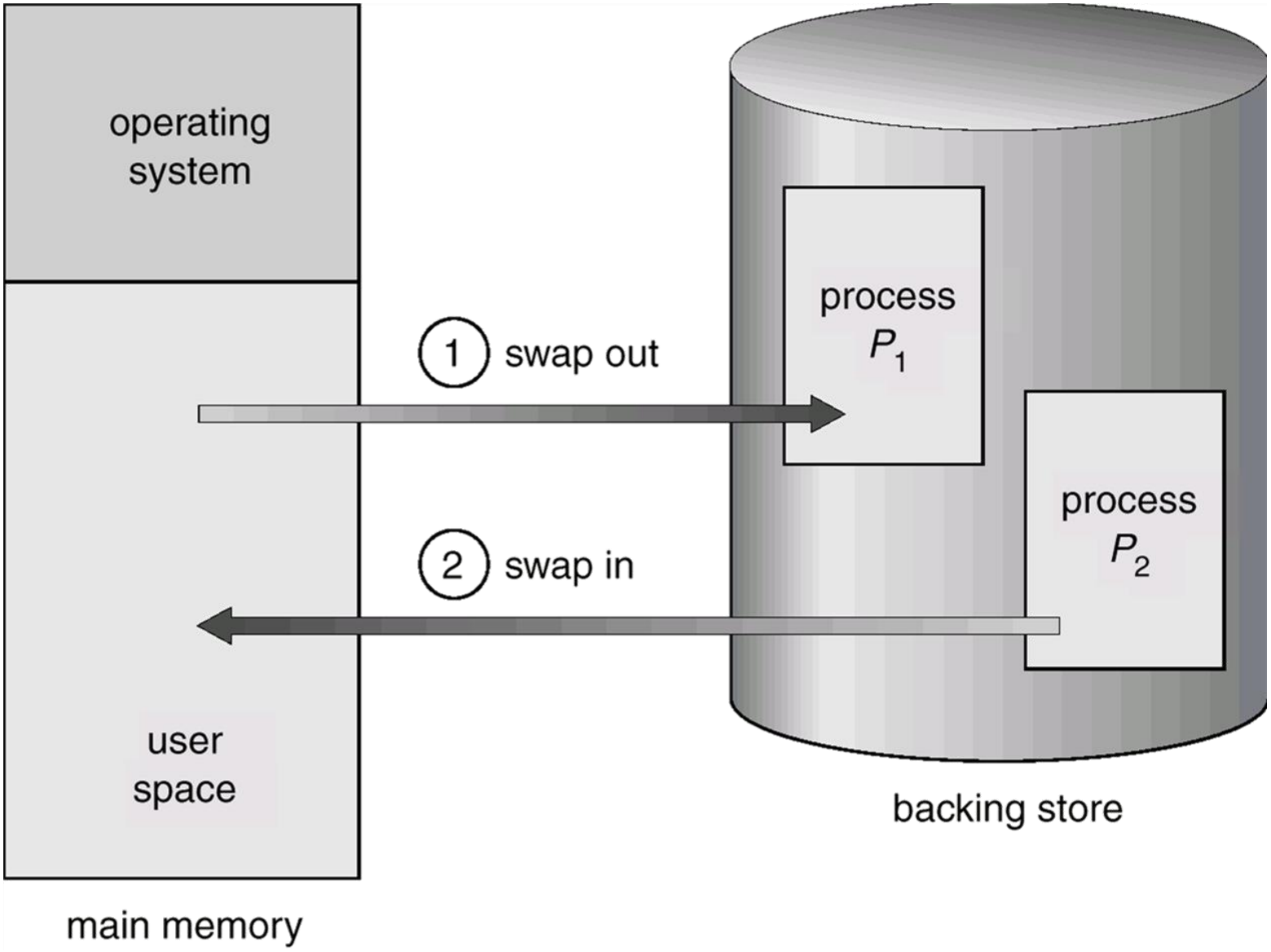
Cơ chế swapping

- Một process có thể tạm thời bị swap ra khỏi bộ nhớ chính và lưu trên một hệ thống lưu trữ phụ. Sau đó, process có thể được nạp lại vào bộ nhớ để tiếp tục tiến trình thực thi.

  Swapping policy: hai ví dụ

  - *Round-robin*: swap out P_1 (vừa tiêu thụ hết quantum của nó), swap in P_2, thực thi P_3,...
  - *Roll out, roll in*: dùng trong cơ chế định thời theo độ ưu tiên (priority-based scheduling)
    - Process có độ ưu tiên thấp hơn sẽ bị swap out nhường chỗ cho process có độ ưu tiên cao hơn mới đến được nạp vào bộ nhớ để thực thi
- Hiện nay, ít hệ thống sử dụng cơ chế swapping trên

Minh họa cơ chế swapping



Mô hình quản lý bộ nhớ

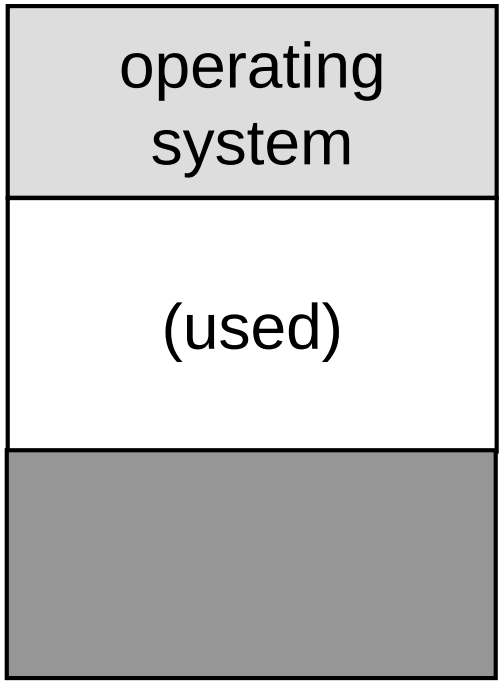
- Trong chương này, mô hình quản lý bộ nhớ là một mô hình đơn giản, không có bộ nhớ ảo.
- Một process phải được nạp hoàn toàn vào bộ nhớ thì mới được thực thi (ngoại trừ khi sử dụng cơ chế overlay).
- Các cơ chế quản lý bộ nhớ sau đây rất ít (hầu như không còn) được dùng trong các hệ thống hiện đại
  - Phân chia cố định (fixed partitioning)
  - Phân chia động (dynamic partitioning)
  - Phân trang đơn giản (simple paging)
  - Phân đoạn đơn giản (simple segmentation)

Phân mảnh (fragmentation)

- *Phân mảnh ngoại* (external fragmentation)
  - Kích thước không gian nhớ còn trống đủ để thỏa mãn một yêu cầu cấp phát, tuy nhiên không gian nhớ này không liên tục ⇒ có thể dùng cơ chế *kết khối* (compaction) để gom lại thành vùng nhớ liên tục.

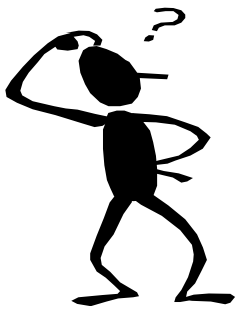
- *Phân mảnh nội* (internal fragmentation)
  - Kích thước vùng nhớ được cấp phát có thể hơi lớn hơn vùng nhớ yêu cầu.
    - Ví dụ: cấp một khoảng trống 18,464 bytes cho một process yêu cầu 18,462 bytes.
  - Hiện tượng phân mảnh nội thường xảy ra khi bộ nhớ thực được chia thành các khối kích thước cố định (fixed-sized block) và các process được cấp phát theo đơn vị khối. Ví dụ: cơ chế phân trang (paging).

Phân mảnh nội

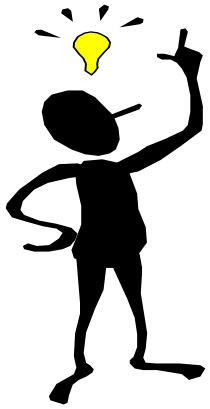


hole kích thước 18,464 bytes

yêu cầu kế tiếp là 18,462 bytes !!!



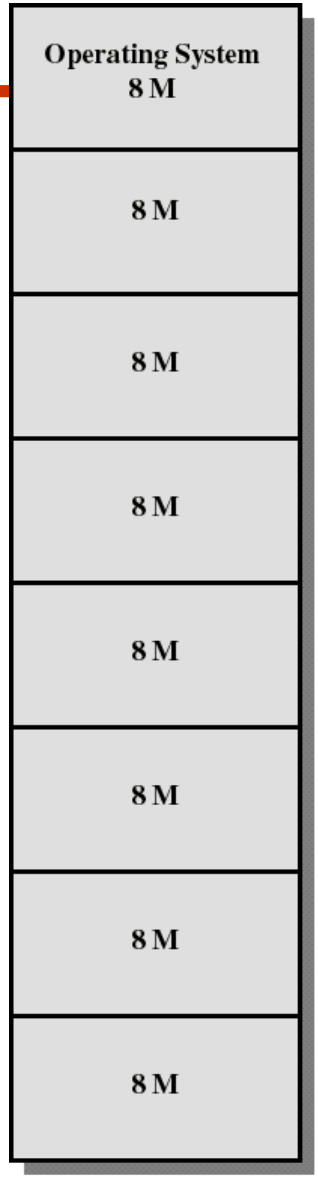
cần quản lý khoảng trống 2 bytes !?!



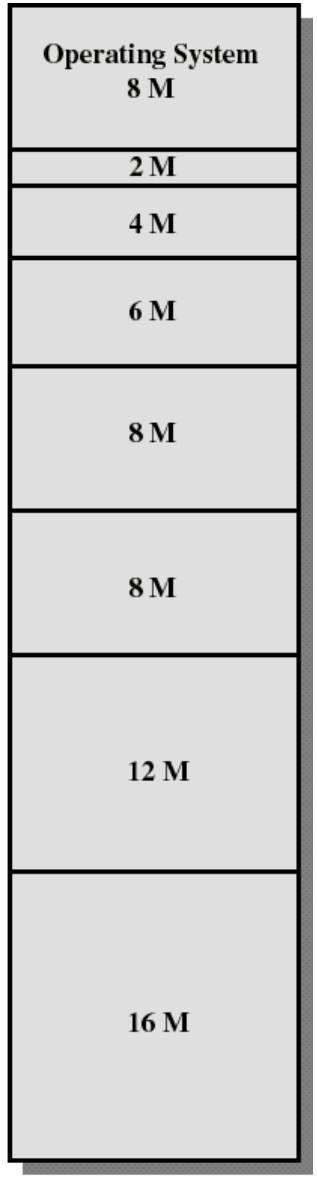
OS sẽ cấp phát hẳn khối 18,464 bytes cho process ⇒ dư ra 2 bytes không dùng!

Fixed partitioning

- Khi khởi động hệ thống, bộ nhớ chính được chia thành nhiều phần rời nhau gọi là các *partition* có kích thước bằng nhau hoặc khác nhau
- Process nào có kích thước nhỏ hơn hoặc bằng kích thước partition thì có thể được nạp vào partition đó.
- Nếu chương trình có kích thước lớn hơn partition thì phải dùng cơ chế overlay.
- Nhận xét
  - Không hiệu quả do bị phân mảnh nội: một chương trình dù lớn hay nhỏ đều được cấp phát trọn một partition.



Equal-size partitions



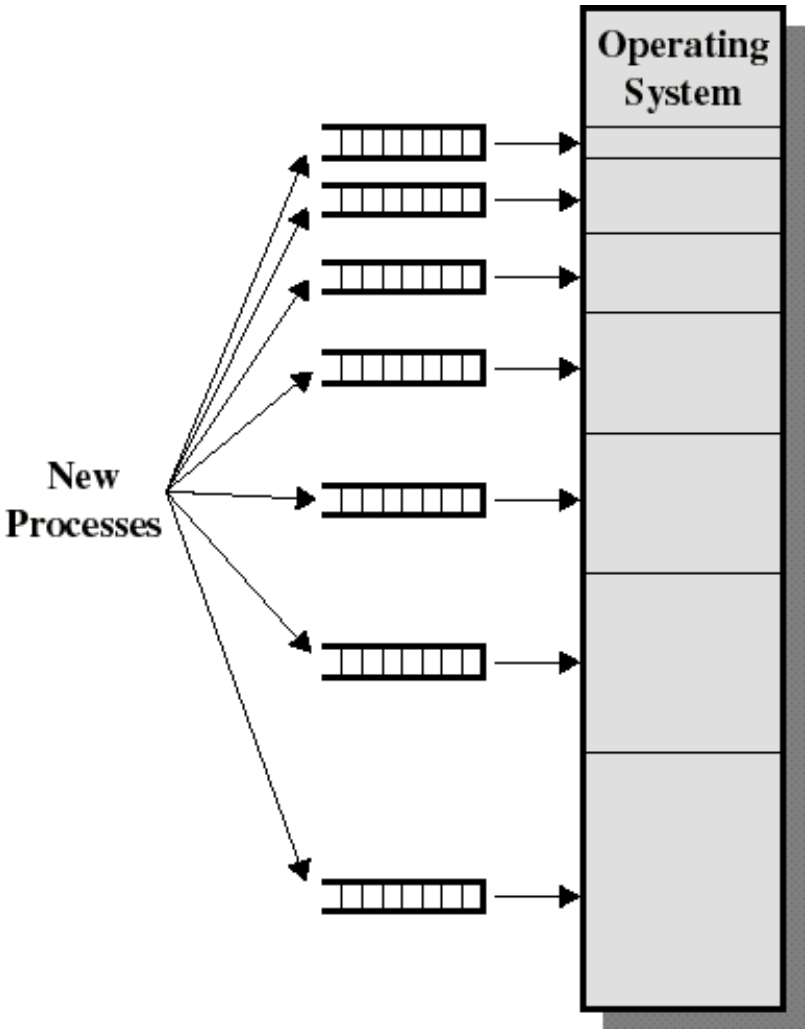
Unequal-size partitions

Chiến lược placement

- Partition có kích thước bằng nhau
  - Nếu còn partition trống $\Rightarrow$ process mới sẽ được nạp vào partition đó
  - Nếu không còn partition trống, nhưng trong đó có process đang bị blocked $\Rightarrow$ swap process đó ra bộ nhớ phụ nhường chỗ cho process mới.

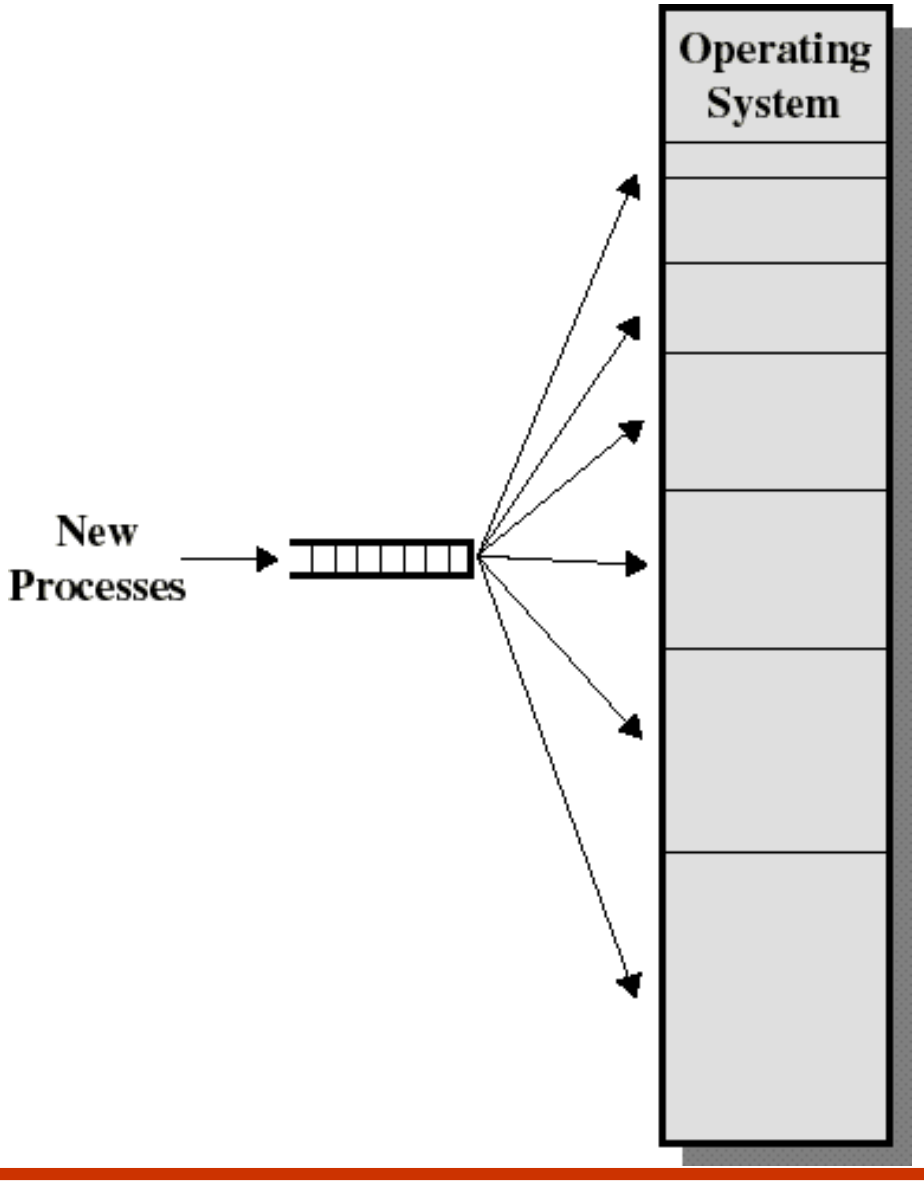
Chiến lược placement (tt)

- Partition có kích thước không bằng nhau: giải pháp 1
  - Gán mỗi process vào partition nhỏ nhất phù hợp với nó
  - Có hàng đợi cho mỗi partition
  - Giảm thiểu phân mảnh nội
  - Vấn đề: có thể có một số hàng đợi trống không (vì không có process với kích thước tương ứng) và hàng đợi dày đặc



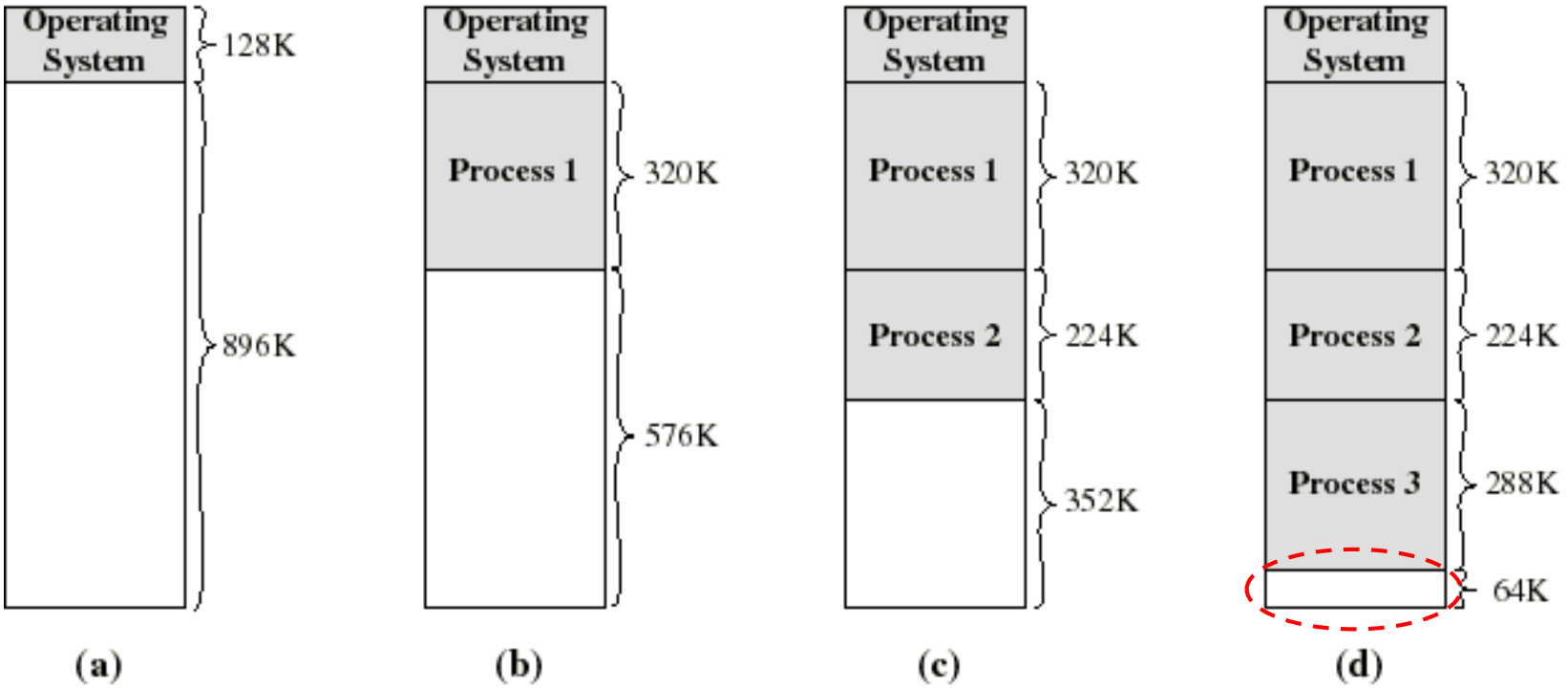
Chiến lược placement (tt)

- Partition có kích thước không bằng nhau: giải pháp 2
  - Chỉ có một hàng đợi chung cho mọi partition
  - Khi cần nạp một process vào bộ nhớ chính ⇒ chọn partition nhỏ nhất còn trống



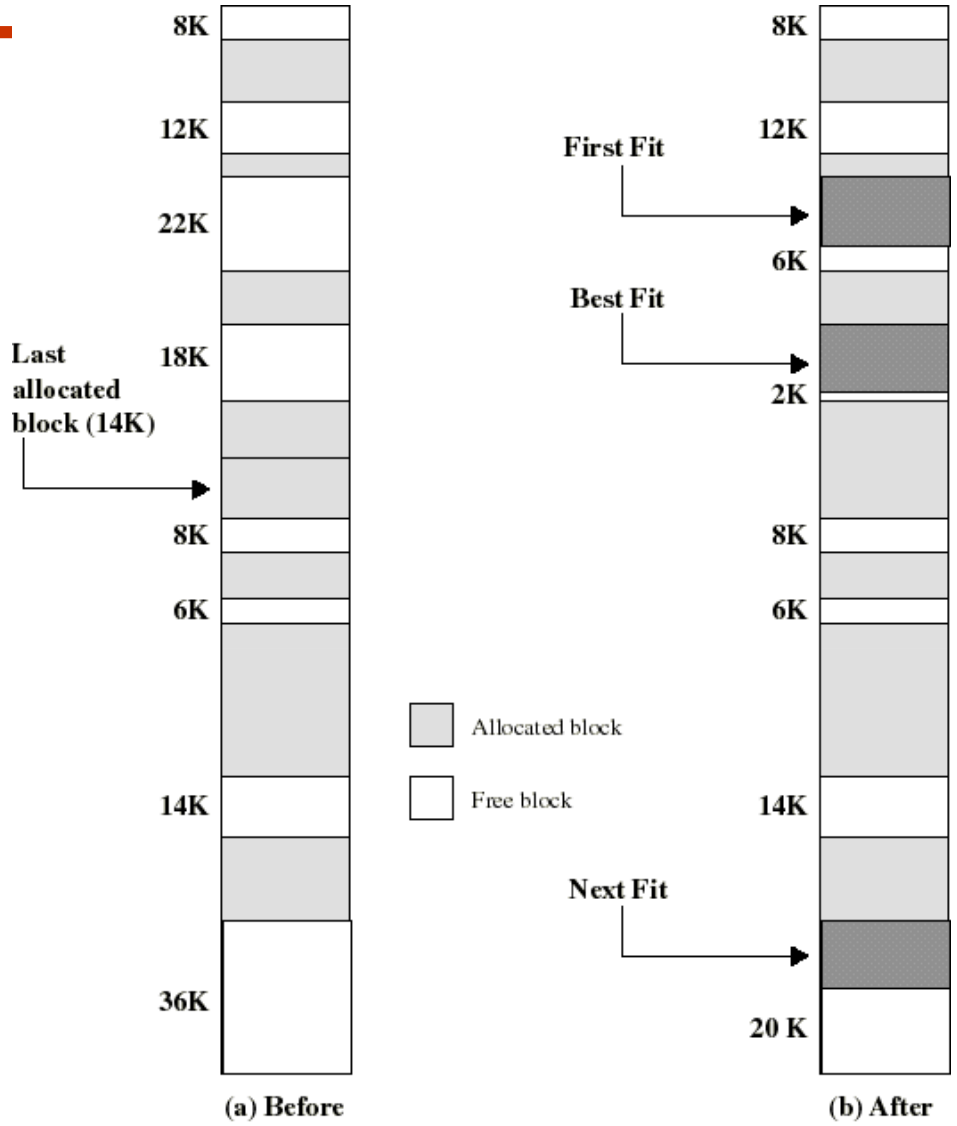
Dynamic partitioning

- Số lượng partition không cố định và partition có thể có kích thước khác nhau
- Mỗi process được cấp phát chính xác dung lượng bộ nhớ cần thiết
- Gây ra hiện tượng phân mảnh ngoại



Chiến lược placement

- Dùng để quyết định cấp phát khối bộ nhớ trống nào cho một process
- Mục tiêu: giảm chi phí compaction
- Các chiến lược placement
  - *Best-fit*: chọn khối nhớ trống nhỏ nhất
  - *First-fit*: chọn khối nhớ trống phù hợp đầu tiên kể từ đầu bộ nhớ
  - *Next-fit*: chọn khối nhớ trống phù hợp đầu tiên kể từ vị trí cấp phát cuối cùng
  - *Worst-fit*: chọn khối nhớ trống lớn nhất



Example Memory Configuration Before and After Allocation of 16 Kbyte Block

Cơ chế *phân trang* (paging)

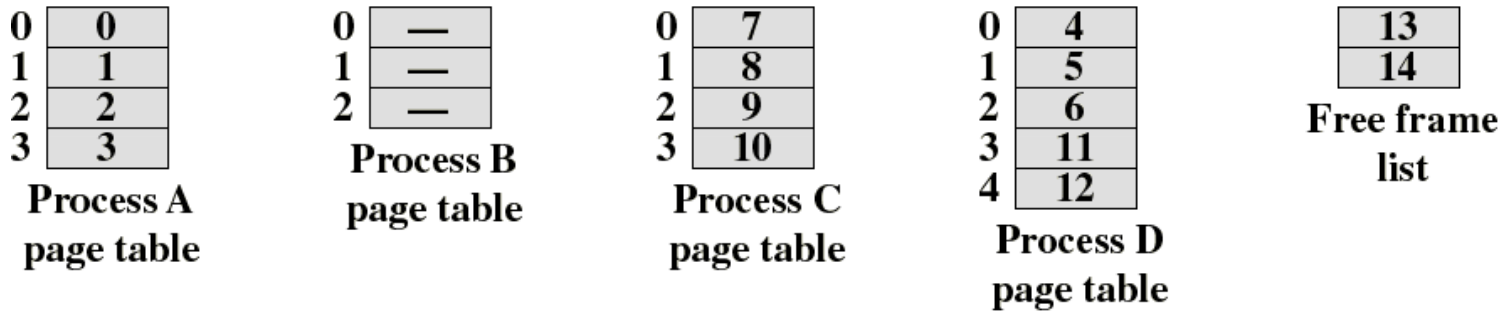
- Cơ chế phân trang cho phép không gian địa chỉ thực (physical address space) của một process có thể không liên tục nhau.
- Bộ nhớ thực được chia thành các khối cố định và có kích thước bằng nhau gọi là *frame*.
  - Thông thường kích thước của frame là lũy thừa của 2, từ khoảng 512 byte đến 16MB.
- *Bộ nhớ logić* (logical memory) hay *không gian địa chỉ logić* là tập mọi địa chỉ logić mà một chương trình bất kỳ có thể sinh ra.
  - Ví dụ

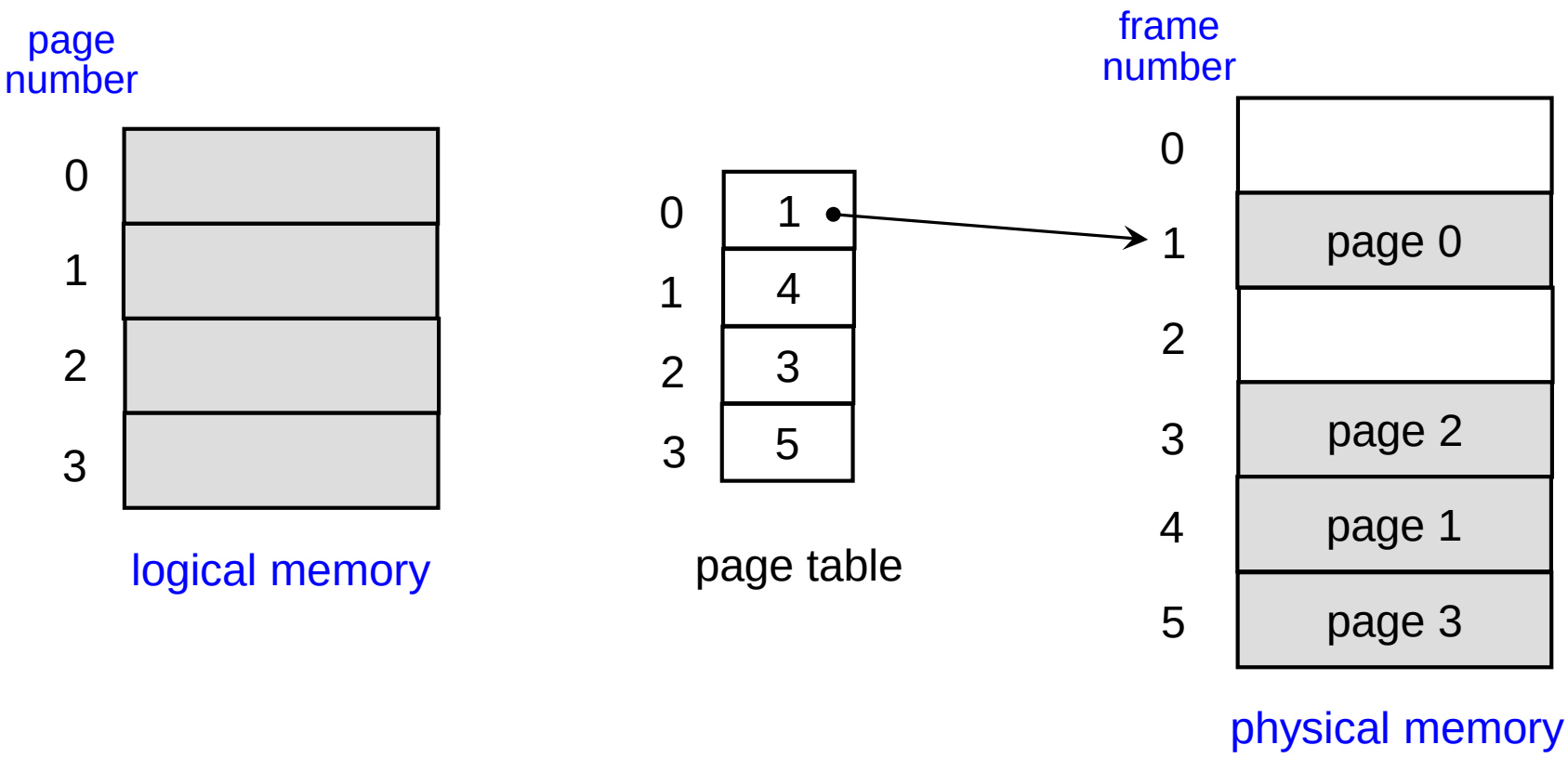
    MOV REG,1000 1000 là một địa chỉ logić
  - Địa chỉ logić còn có thể được chương trình sinh ra bằng cách dùng indexing, base register, segment register,...

Cơ chế phân trang (tt)

- *Bộ nhớ logic* cũng được chia thành các khối cố định có cùng kích thước gọi là *trang nhớ* (page).
- Frame và trang nhớ có kích thước bằng nhau.
- Hệ điều hành phải thiết lập một *bảng phân trang* (page table) để ánh xạ địa chỉ logic thành địa chỉ thực
  - Mỗi process có một bảng phân trang, được quản lý bằng một con trỏ lưu giữ trong PCB. Công việc thiết lập bảng phân trang cho process là một phần của chuyển ngữ cảnh
- Cơ chế phân trang khiến bộ nhớ bị phân mảnh nội, tuy nhiên lại khắc phục được phân mảnh ngoại.

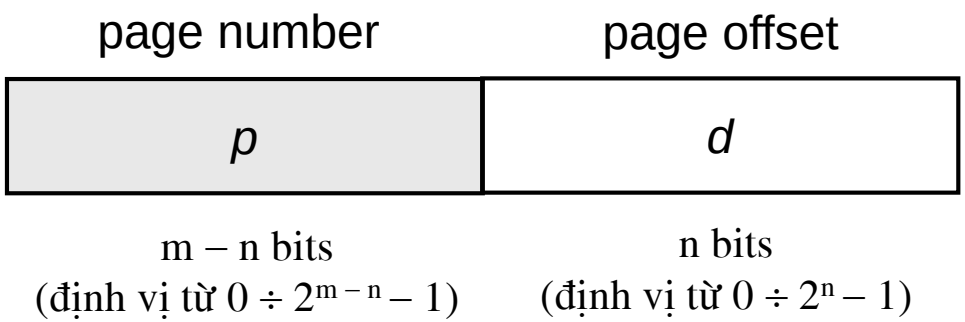
Cơ chế phân trang (tt)





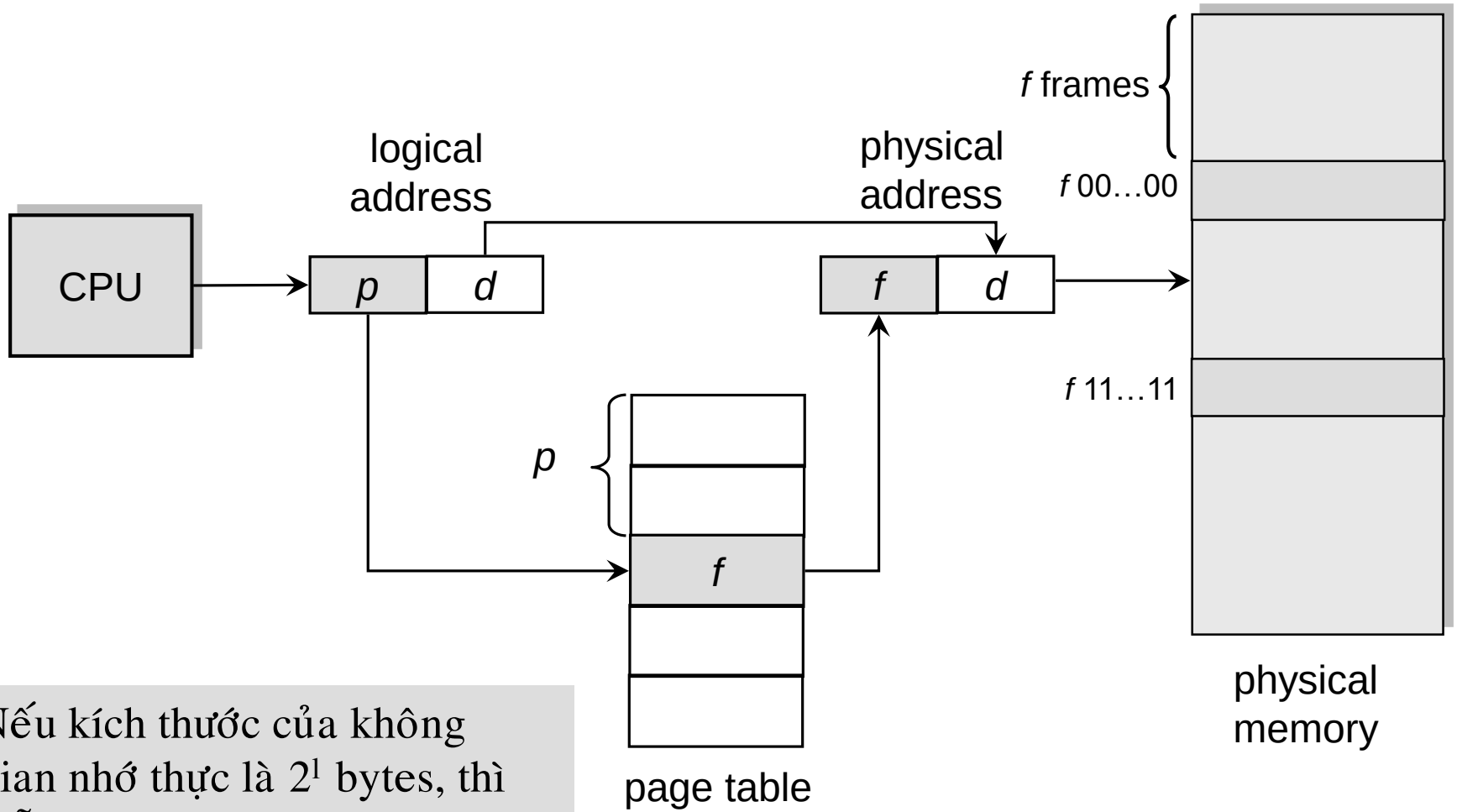
Chuyển đổi địa chỉ trong paging

- Địa chỉ logic gồm có:
  - *Page number*, *p*, được dùng làm chỉ mục dò tìm trong bảng phân trang. Mỗi mục trong bảng phân trang chứa chỉ số frame (còn gọi là số frame cho gọn) của trang tương ứng trong bộ nhớ thực.
  - *Page offset*, *d*, được kết hợp với *địa chỉ cơ sở* (base address) để định vị địa chỉ thực.
- Nếu kích thước của không gian địa chỉ ảo là 2^m, và kích thước của trang là 2^n (đơn vị là byte hay word tùy theo kiến trúc máy)

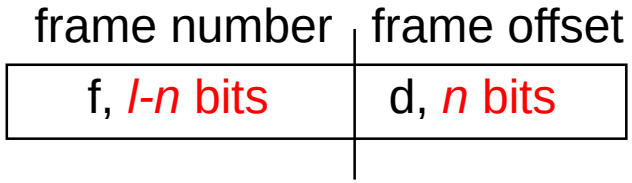


Bảng phân trang sẽ có tổng cộng $2^m/2^n = 2^{m-n}$ *mục* (entry)

Paging hardware

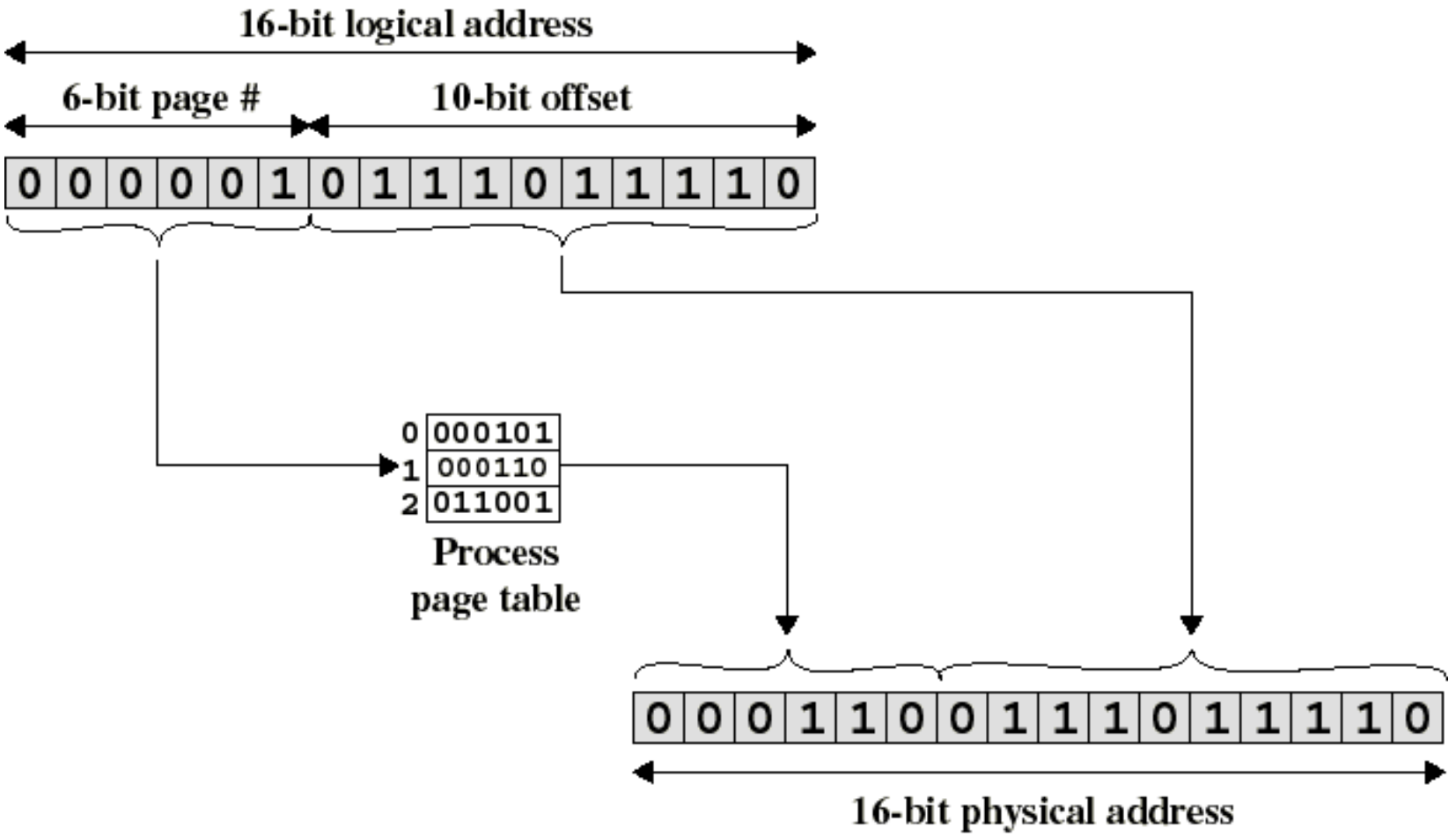


Nếu kích thước của không gian nhớ thực là 2^l bytes, thì mỗi mục của bảng phân trang có l – n bits



Chuyển đổi địa chỉ nhớ trong paging

Ví dụ:



Hiện thực bảng phân trang

- Bảng phân trang thường được lưu giữ trong bộ nhớ chính
  - Mỗi process được hệ điều hành cấp một bảng phân trang
  - Thanh ghi *page-table base* (PTBR) trỏ đến bảng phân trang
  - Thanh ghi *page-table length* (PTLR) biểu thị kích thước của bảng phân trang (có thể được dùng trong cơ chế bảo vệ bộ nhớ)

- Mỗi tác vụ truy cập dữ liệu/lệnh cần hai thao tác truy xuất vùng nhớ
  - Một thao tác dùng page number p làm index để truy cập mục trong bảng phân trang nhằm lấy số frame và kế đến là một thao tác dùng page offset d để truy xuất dữ liệu/lệnh trong frame
  - Thường dùng một bộ phận cache phần cứng có tốc độ truy xuất và tìm kiếm cao, gọi là *thanh ghi kết hợp* (associative register) hoặc *translation look-aside buffers* (TLBs)

Associative register (hardware)

- hay TLB, là thanh ghi hỗ trợ tìm kiếm truy xuất dữ liệu với tốc độ cực nhanh.

Page #	Frame #

Số mục của TLB khoảng 8 ÷ 2048

TLB là "cache" của bảng phân trang

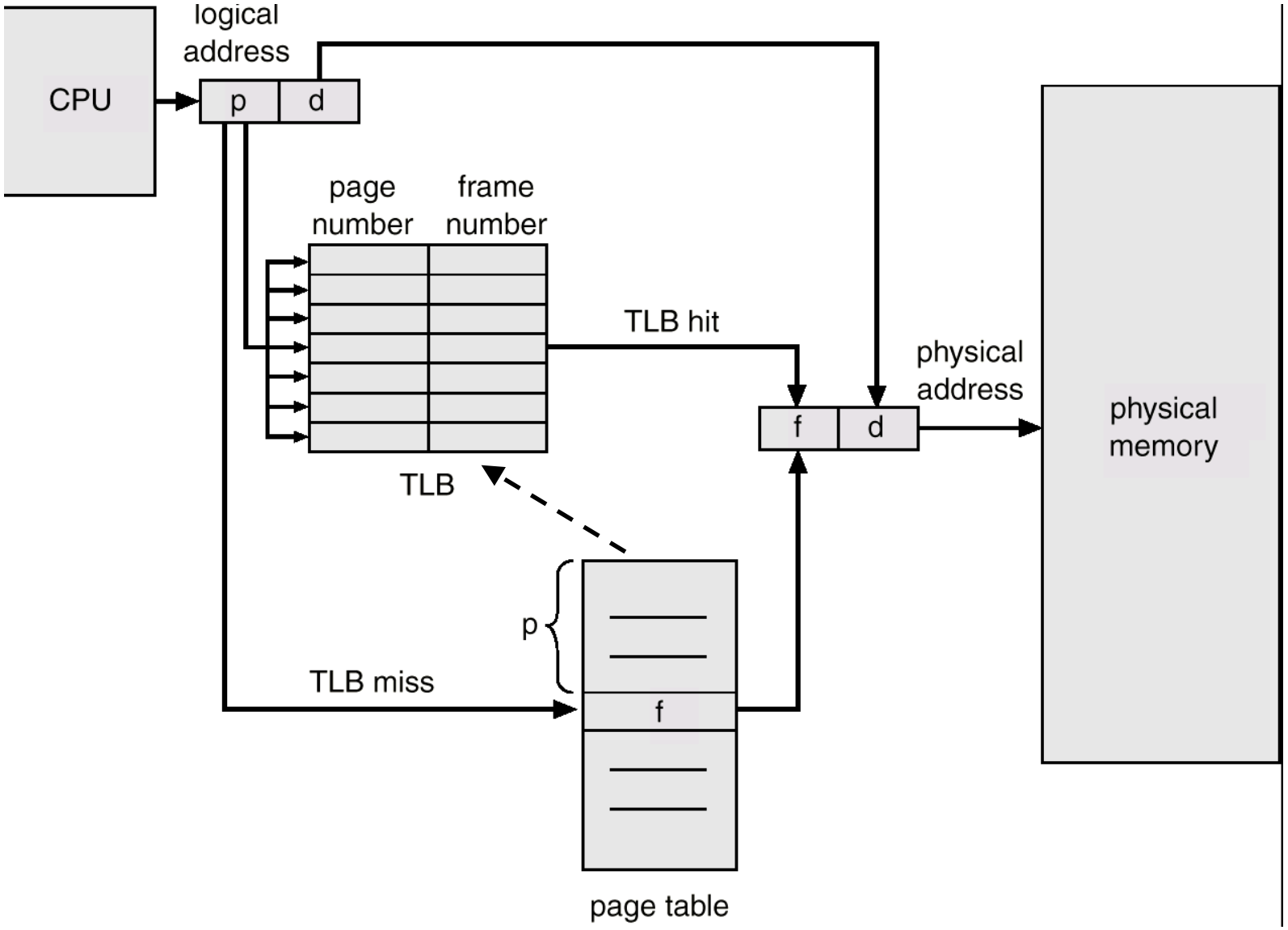
Khi có chuyển ngữ cảnh, TLB bị xóa

Khi TLB đầy, thay thế mục dùng LRU

Ánh xạ địa chỉ ảo

- Nếu page number nằm trong TLB (: hit, trúng) ⇒ lấy ngay được số frame ⇒ tiết kiệm được thời gian truy cập bộ nhớ chính để lấy số frame trong bảng phân trang.
- Ngược lại (: miss, trật), phải lấy số frame từ bảng phân trang như bình thường.

Paging hardware với TLB



Effective access time (EAT)

Tính thời gian truy xuất hiệu dụng (effective access time, EAT)

- Thời gian tìm kiếm trong TLB (associative lookup): ε
- Thời gian một chu kỳ truy xuất bộ nhớ: *x*
- *Hit ratio:* tỉ số giữa số lần chỉ số trang được tìm thấy (hit) trong TLB và số lần truy xuất khởi nguồn từ CPU
  - Kí hiệu hit ratio: α
- Thời gian cần thiết để có được chỉ số frame
  - Khi chỉ số trang có trong TLB (hit) ε + x
  - Khi chỉ số trang không có trong TLB (miss) ε + x + x
- *Thời gian truy xuất hiệu dụng*

$$\text{EAT} = (\varepsilon + x)\alpha + (\varepsilon + 2x)(1 - \alpha)$$

$$= (2 - \alpha)x + \varepsilon$$

Effective access time (tt)

- Ví dụ 1: đơn vị thời gian nano giây
  - Associative lookup = 20
  - Memory access = 100
  - Hit ratio = 0.8
  - EAT = (100 + 20) × 0.8 +
    (200 + 20) × 0.2
    = 1.2 × 100 + 20
    = 140

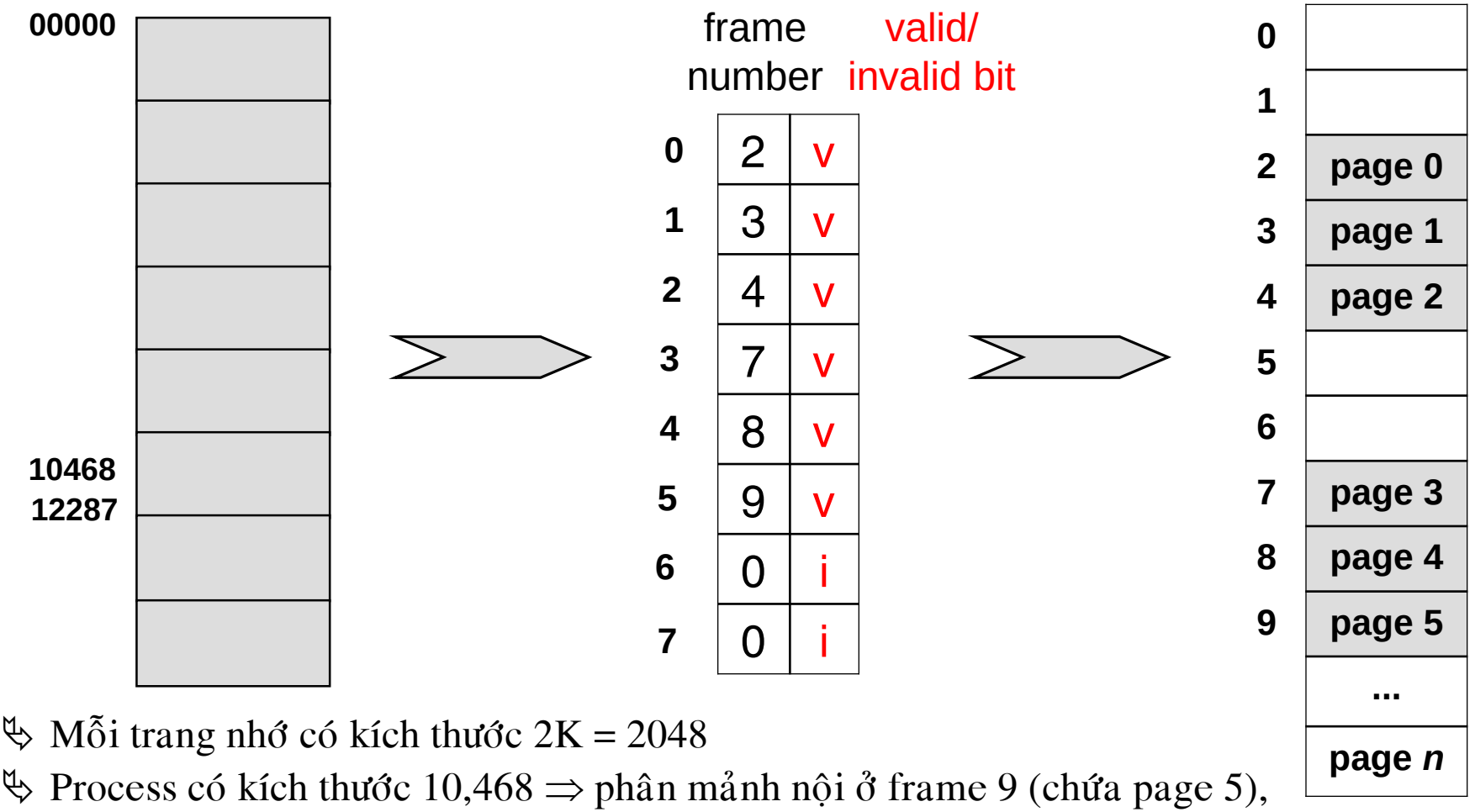
- Ví dụ 2
  - Associative lookup = 20
  - Memory access = 100
  - Hit ratio = 0.98
  - EAT = (100 + 20) × 0.98 +
    (200 + 20) × 0.02
    = 1.02 × 100 + 20
    = 122

Bảo vệ bộ nhớ

- Việc bảo vệ bộ nhớ được hiện thực bằng cách gắn với frame các *bit bảo vệ* (protection bits) được giữ trong bảng phân trang. Các bit này biểu thị các thuộc tính sau
  - read-only, read-write, execute-only

- Ngoài ra, còn có một *valid/invalid bit* gắn với mỗi mục trong bảng phân trang
  - "valid": cho biết là trang của process, do đó là một trang hợp lệ.
  - "invalid": cho biết là trang không của process, do đó là một trang bất hợp lệ.

Bảo vệ bằng valid/invalid bit



- Mỗi trang nhớ có kích thước 2K = 2048
- Process có kích thước 10,468 ⇒ phân mảnh nội ở frame 9 (chứa page 5), các địa chỉ ảo > 12287 là các địa chỉ invalid.
- Dùng PTLR để kiểm tra truy xuất đến bảng phân trang có nằm trong bảng hay không.

Bảng phân trang 2 mức

- Các hệ thống hiện đại đều hỗ trợ không gian địa chỉ ảo rất lớn (2^{32} đến 2^{64}), ở đây giả sử là 2^{32}
  - Giả sử kích thước trang nhớ là 4KB (= 2^{12})

    $\Rightarrow$ bảng phân trang sẽ có $2^{32}/2^{12} = 2^{20}$ = 1M mục.
  - Giả sử mỗi mục gồm 4 byte thì mỗi process cần 4MB cho bảng phân trang ☹

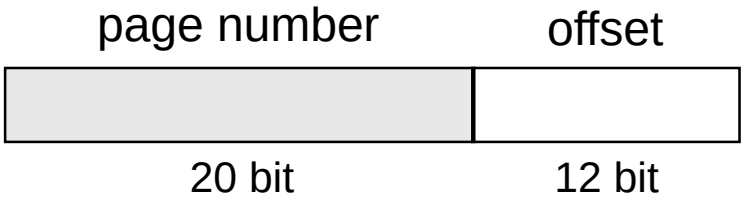
- Một giải pháp là, thay vì dùng một bảng phân trang duy nhất cho mỗi process, "paging" bảng phân trang này, và chỉ giữ những bảng phân trang cần thiết trong bộ nhớ $\rightarrow$ *bảng phân trang 2 mức* (two-level page table).

Bảng phân trang 2-mức (tt)

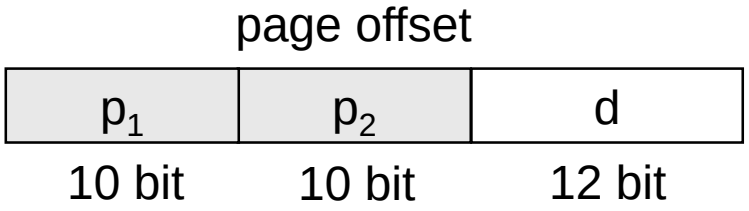
Ví dụ

- Một địa chỉ logic trên hệ thống 32-bit với trang nhớ 4K được chia thành các phần sau:
  - Page number: 20 bit
  - Page offset: 12 bit
    - Nếu mỗi mục dài 4 byte

    $\Rightarrow$ Cần 2^{20} * 4 byte = 4 MB cho mỗi page table



- Bảng phân trang cũng được chia nhỏ nên page number cũng được chia nhỏ thành 2 phần:
  - 10-bit page number
  - 10-bit page offset



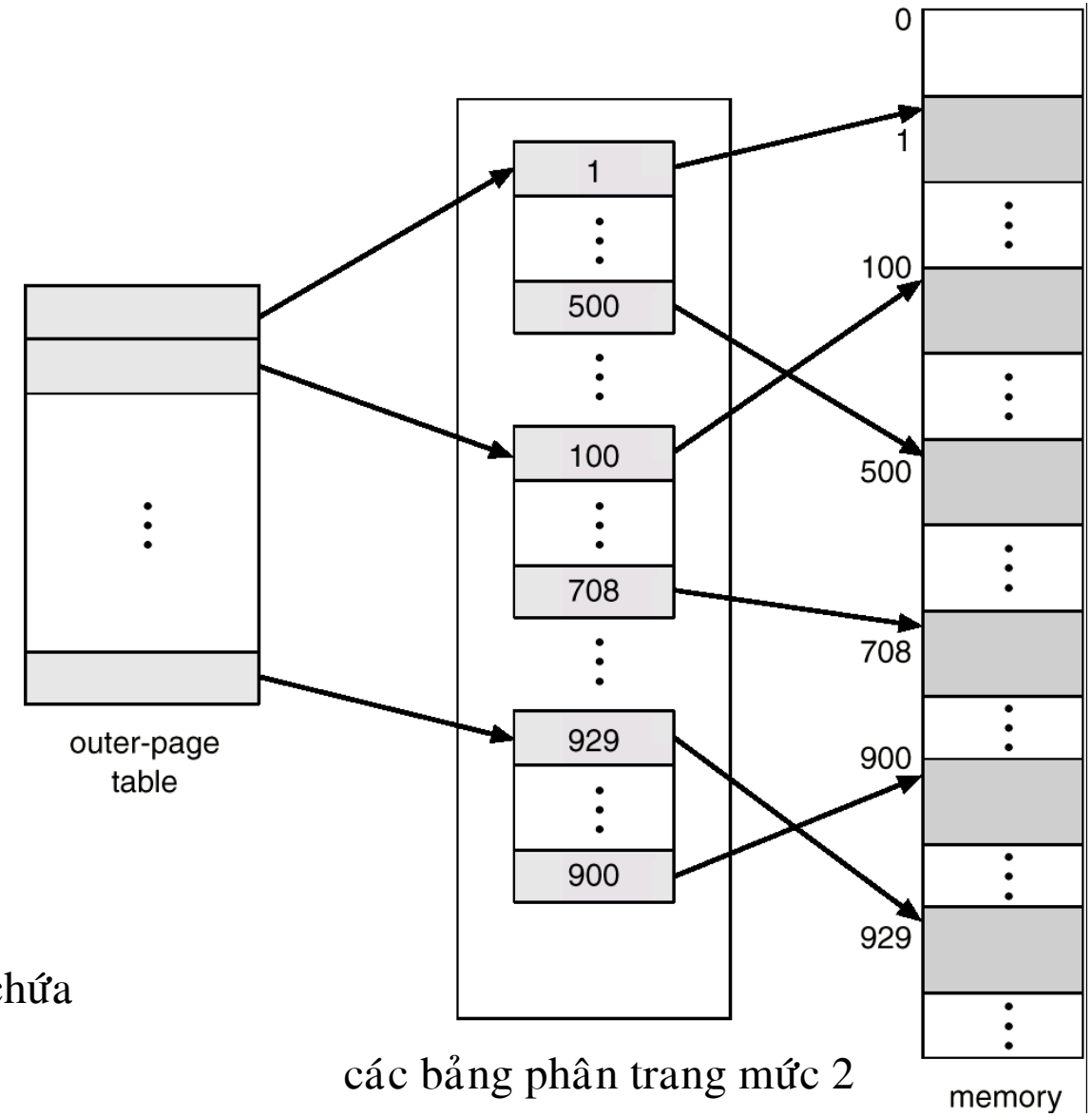
- Vì vậy, một địa chỉ logic sẽ như hình vẽ bên

  p_1 : chỉ số của trang trong *bảng phân trang* mức 1 (outer-page table)

  p_2 : chỉ số của trang trong *bảng phân trang* mức 2

Bảng phân trang 2-mức (tt)

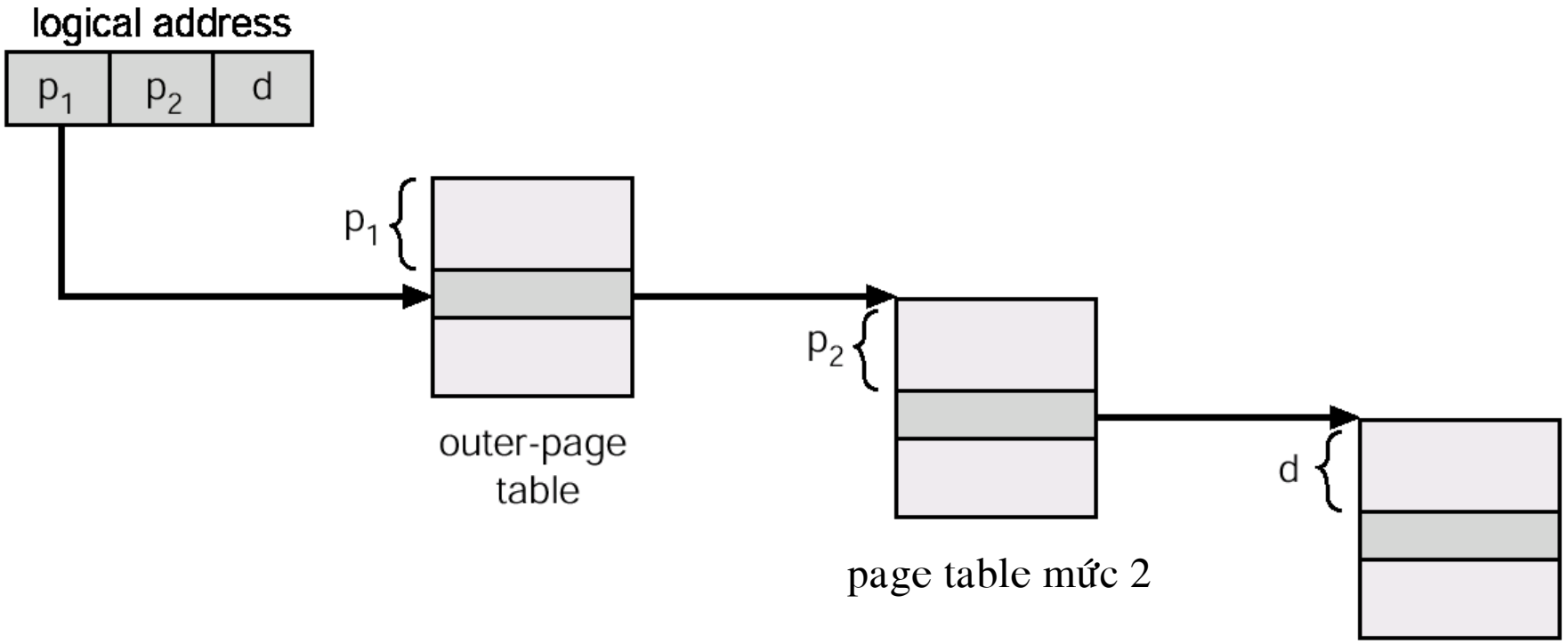
Minh họa



- Có 2^{p1} mục trong bảng phân trang mức 1
- Mỗi bảng phân trang mức 2 chứa 2^{p2} mục

Bảng phân trang 2 mức (tt)

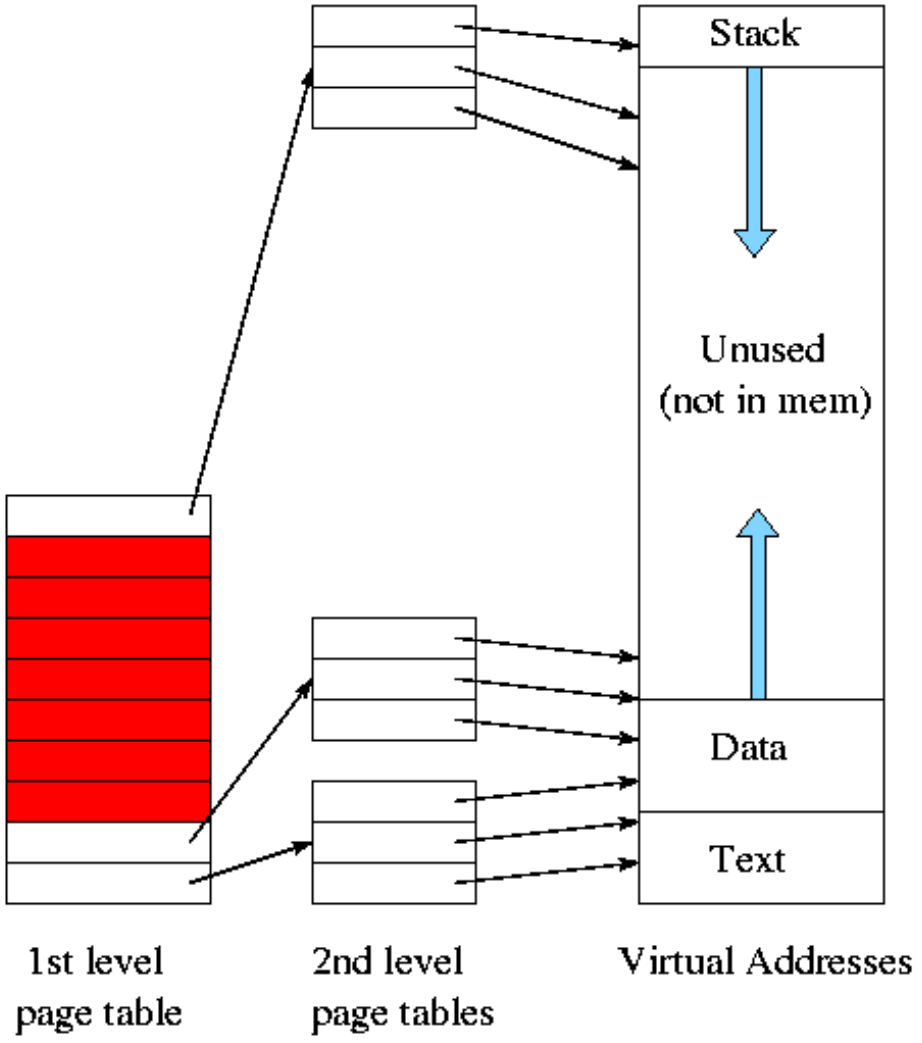
- Sơ đồ chuyển đổi địa chỉ (address-translation scheme) cho cơ chế bảng phân trang 2 mức, 32-bit địa chỉ



Bảng phân trang 2 mức (tt)

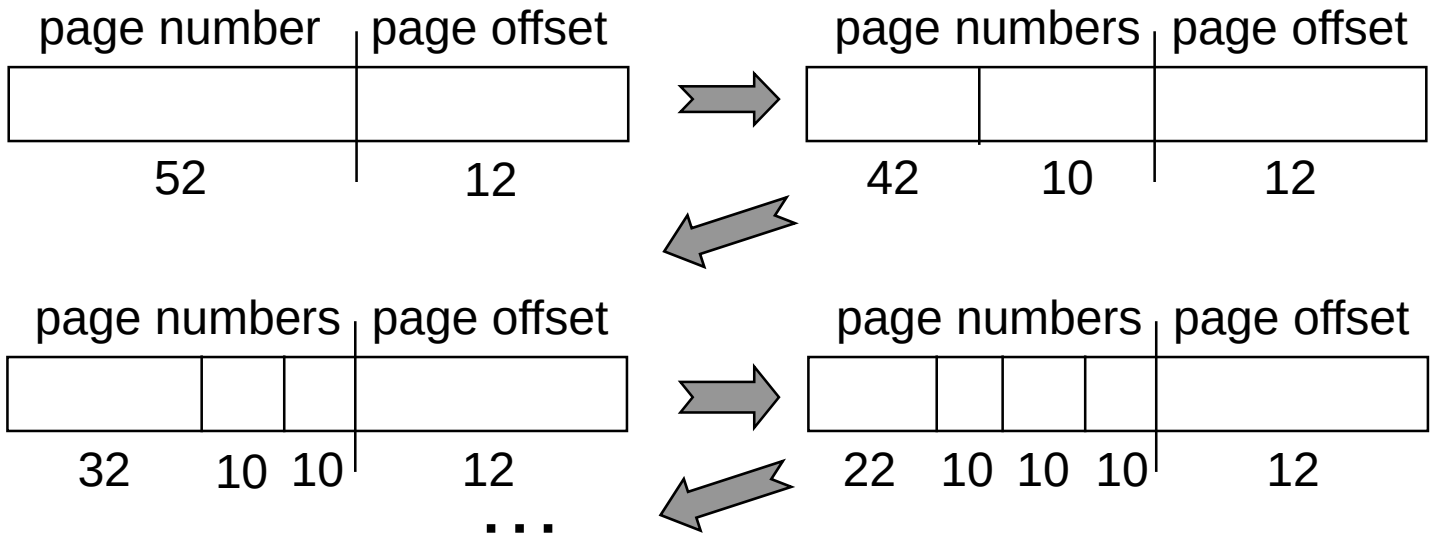
Bảng phân trang 2 mức giúp tiết kiệm bộ nhớ:

Vùng màu đỏ tương ứng với phần không được sử dụng của không gian địa chỉ ảo. Các mục màu đỏ được đánh dấu là không có frame nên sẽ gây ra page fault nếu được tham chiếu đến; khi đó OS sẽ tạo thêm bảng phân trang mức 2 mới.



Bảng phân trang đa mức

- Ví dụ: Không gian địa chỉ logic 64-bit với trang nhớ 4K
  - Bảng phân trang 2-mức vẫn còn quá lớn! Tương tự bảng phân trang 2 mức, ta có bảng phân trang 3, 4,..., n mức

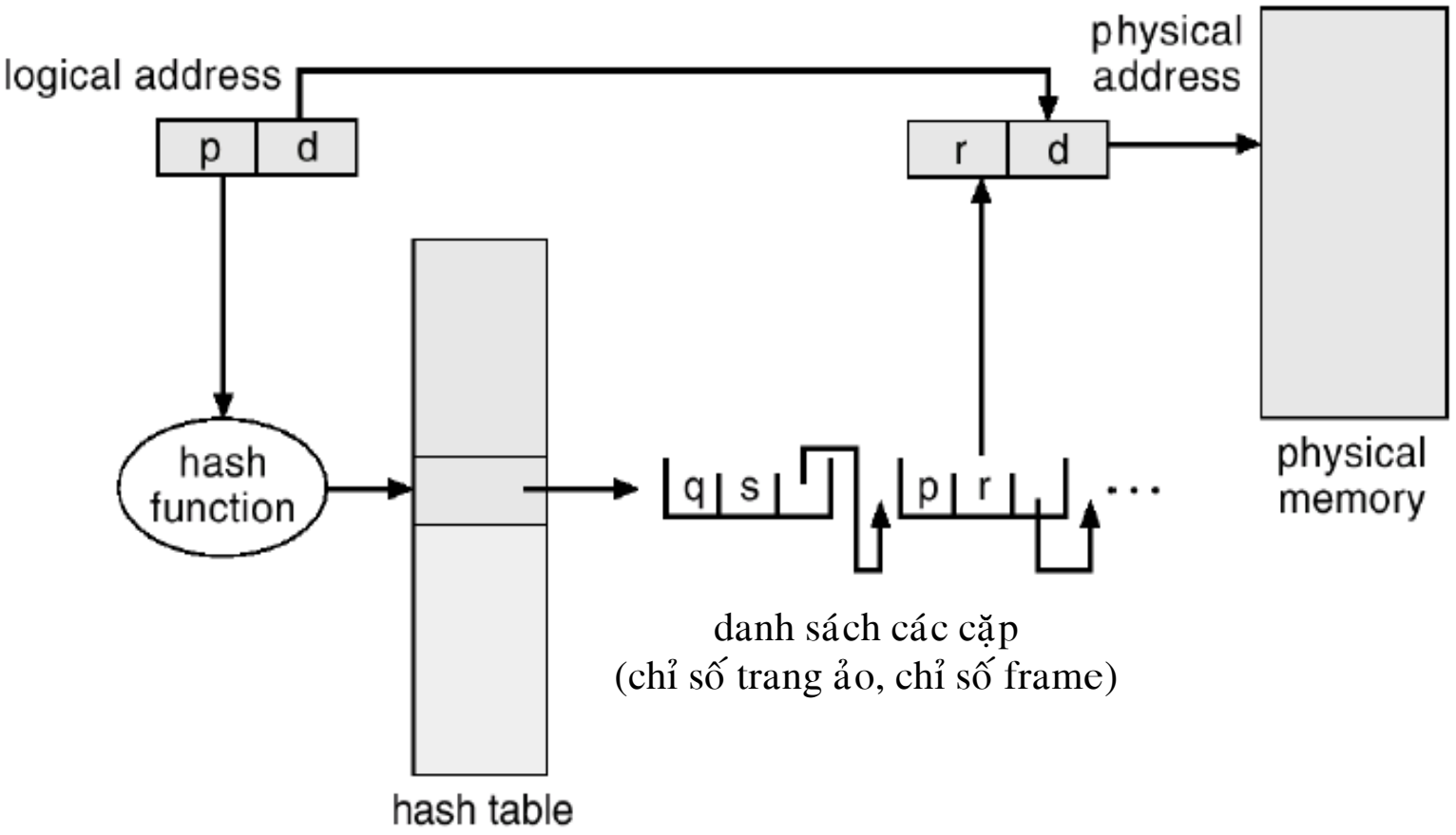


- Tiết kiệm chổ trong bộ nhớ chính bằng cách chỉ giữ trong bộ nhớ chính các bảng phân trang mà process hiện đang cần.

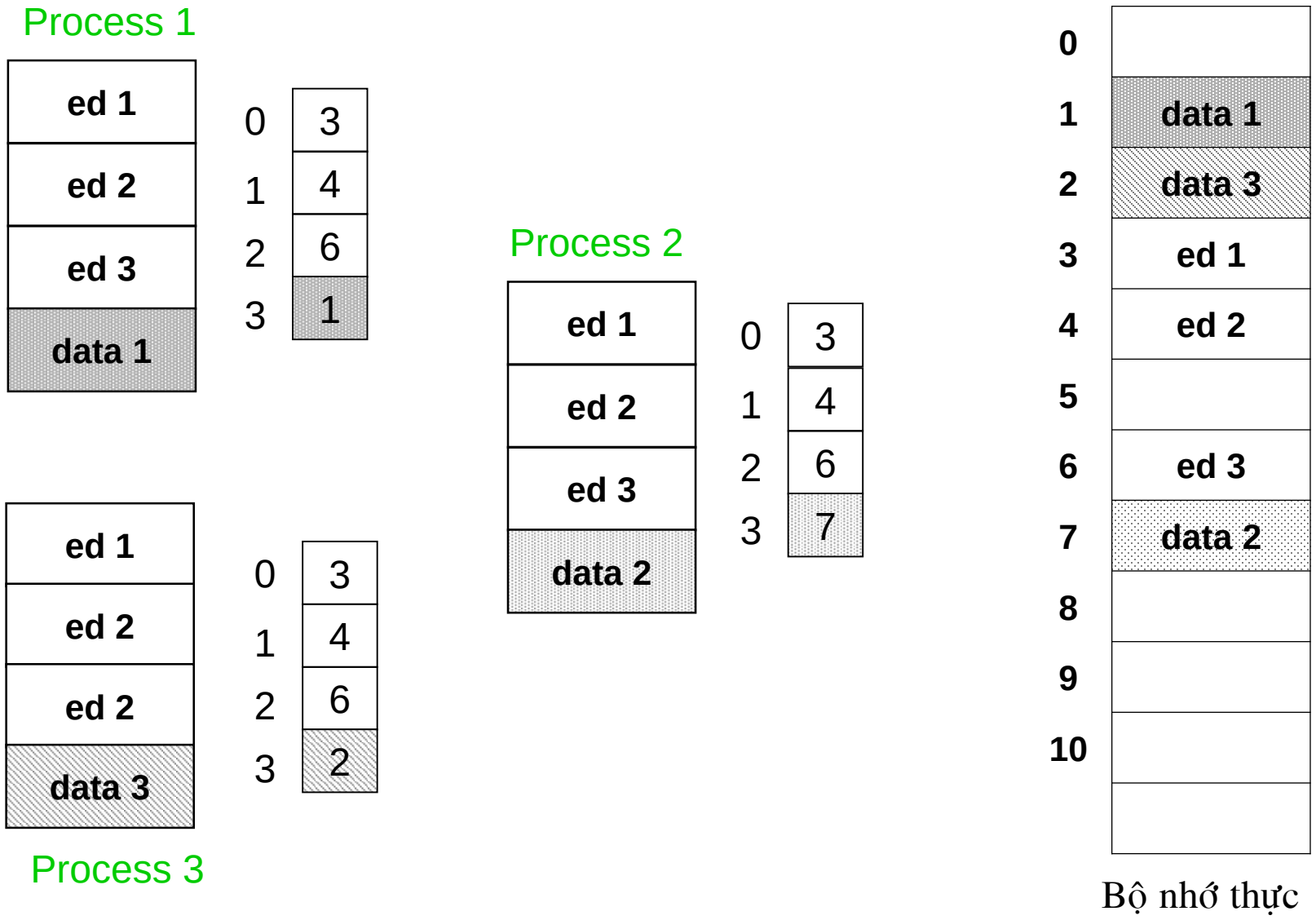
Bảng phân trang băm (hashed page table)

- Dùng bảng băm để giảm không gian bảng phân trang
  - Rất phổ biến trong các hệ thống lớn hơn 32 bit địa chỉ.
- Để giải quyết đụng độ, mỗi mục của bảng băm được gắn một danh sách liên kết. Mỗi phần tử của danh sách là một cặp (chỉ số trang ảo, chỉ số frame).
  - Chỉ số trang ảo (virtual page number) được biến đổi qua hàm băm thành một *hashed value*. Cặp (chỉ số trang ảo, chỉ số frame) sẽ được lưu vào danh sách liên kết tại mục có chỉ số là hashed value.
- Giải thuật tìm trang:
  - Chỉ số trang ảo được biến đổi thành hashed value (với cùng hàm băm như trên). Hashed value là chỉ số của mục cần truy cập trong bảng băm. Sau đó, tìm trong danh sách liên kết phần tử chứa chỉ số trang ảo để trích ra được chỉ số frame tương ứng.

Hashed page table (tt)



Chia sẻ các trang nhớ



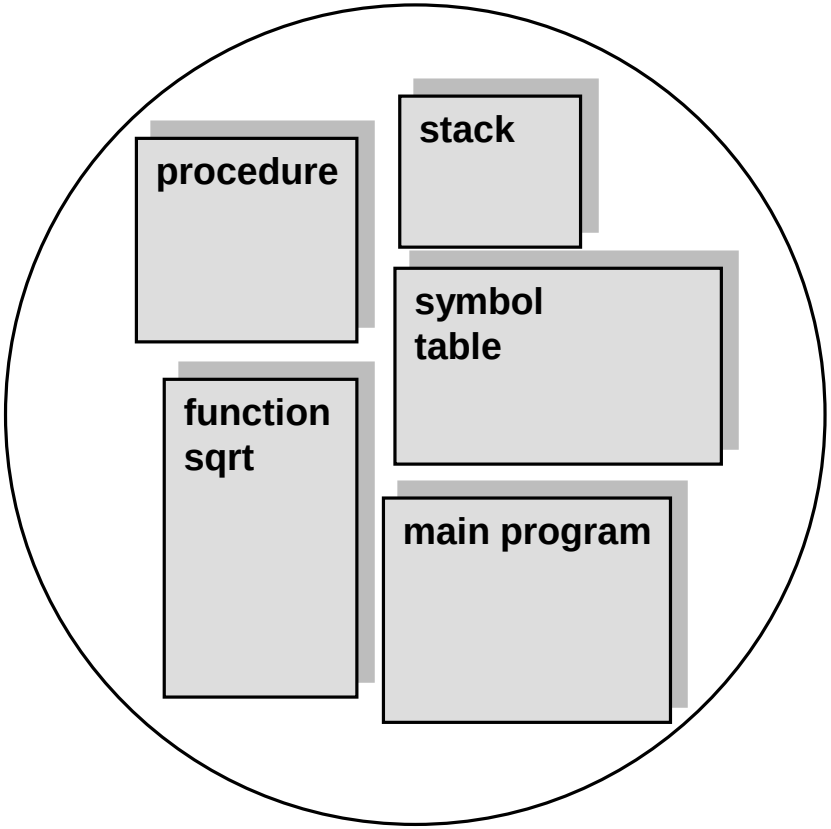
Phân đoạn (segmentation)

- Nhìn lại cơ chế phân trang
  - user view (không gian địa chỉ ảo) tách biệt với không gian bộ nhớ thực. Cơ chế phân trang thực hiện phép ánh xạ user-view vào bộ nhớ thực.

- Trong thực tế, dưới góc nhìn của user, một chương trình cấu thành từ nhiều *đoạn* (segment). Mỗi đoạn là một đơn vị logic của chương trình, như
  - main program, procedure, function
  - local variables, global variables, common block, stack, symbol table, arrays,...

User view của một chương trình

- Thông thường, một chương trình được biên dịch. Trình biên dịch sẽ tự động xây dựng các segment.
- Ví dụ, trình biên dịch Pascal sẽ tạo ra các segment sau:
  - Global variables
  - Procedure call stack
  - Procedure/function code
  - Local variable
- Trình loader sẽ gán mỗi segment một số định danh riêng.

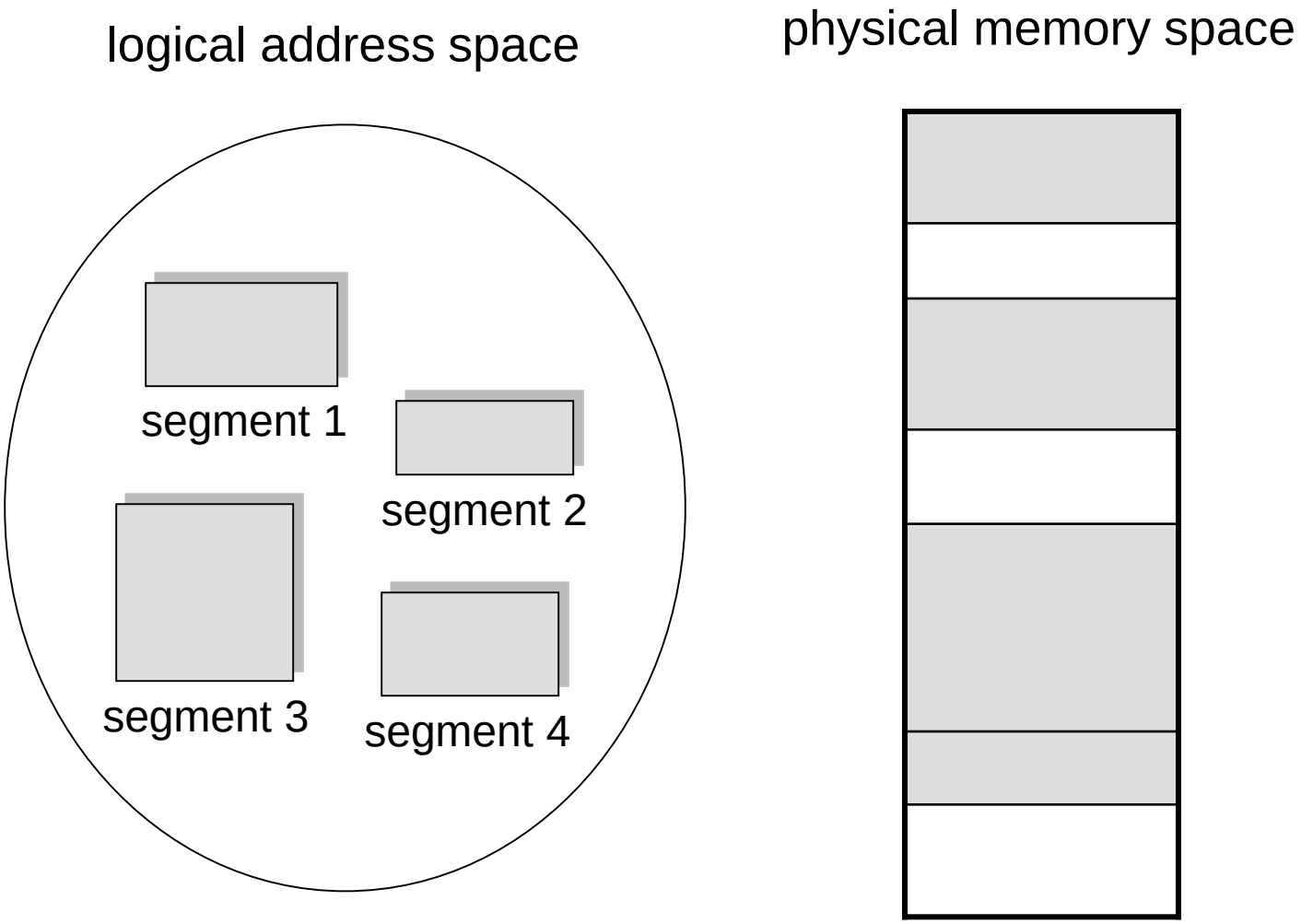


Logical address space

Phân đoạn

- Dùng cơ chế *phân đoạn* để quản lý bộ nhớ có hỗ trợ user view
  - *Không gian địa chỉ ảo* là một tập các đoạn, mỗi đoạn có tên và kích thước riêng.
  - Một địa chỉ logic được định vị bằng tên đoạn và độ dời (offset) bên trong đoạn đó (so sánh với phân trang!)

Phân đoạn (tt)


logical address space
segment 1
segment 2
segment 3
segment 4
physical memory space

Hiện thực phân đoạn

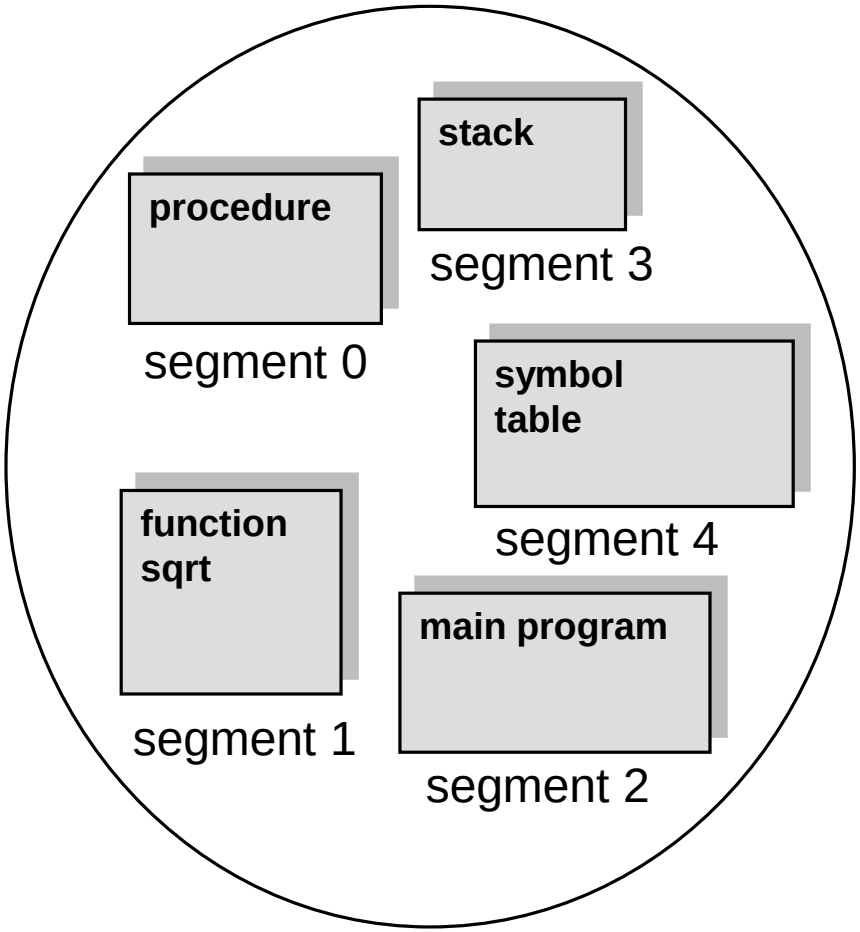
- *Địa chỉ logić* là một cặp giá trị

  (segment number, offset)

- *Bảng phân đoạn* (segment table): gồm nhiều mục, mỗi mục chứa
  - base, chứa địa chỉ khởi đầu của segment trong bộ nhớ
  - limit, xác định kích thước của segment
- *Segment-table base register* (STBR): trỏ đến vị trí bảng phân đoạn trong bộ nhớ
- *Segment-table length register* (STLR): số lượng segment của chương trình

  $\Rightarrow$ Một chỉ số segment s là hợp lệ nếu $s < STLR$

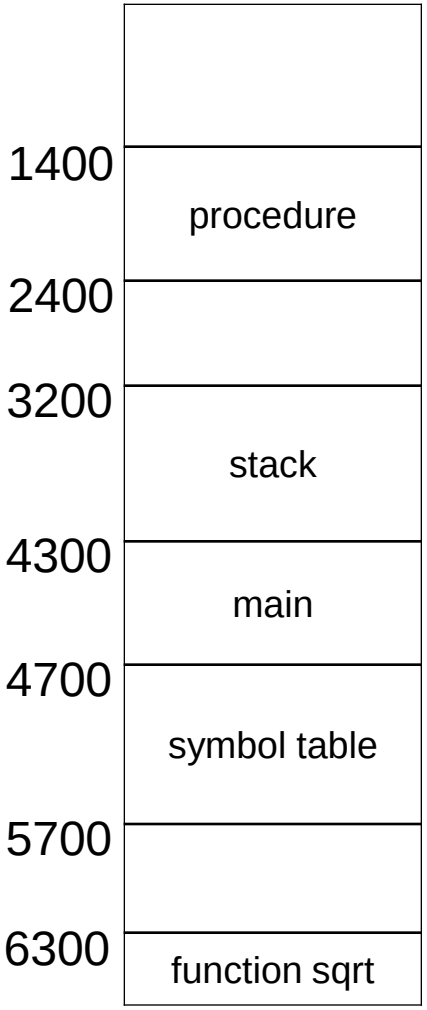
Một ví dụ về phân đoạn



logical address space

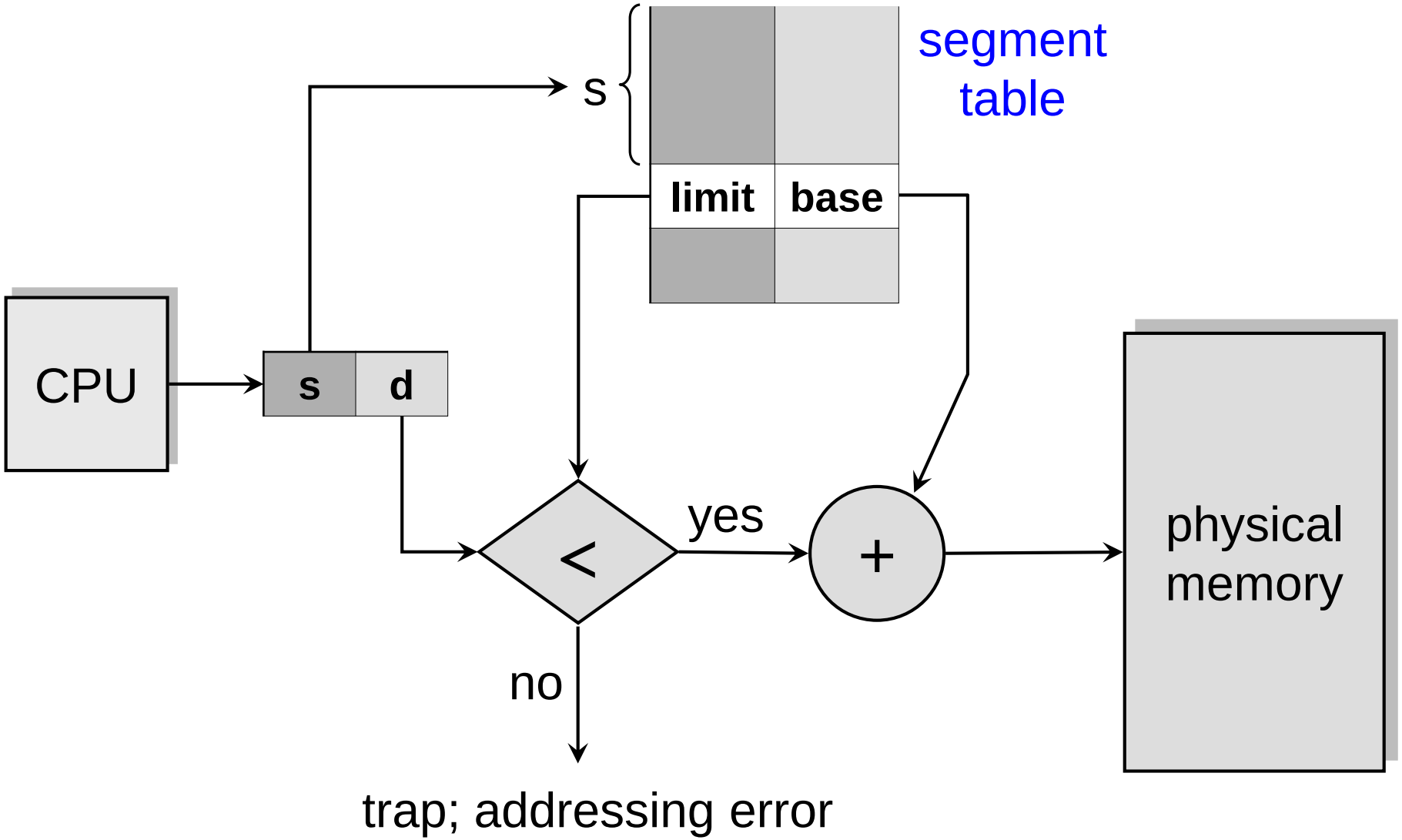
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



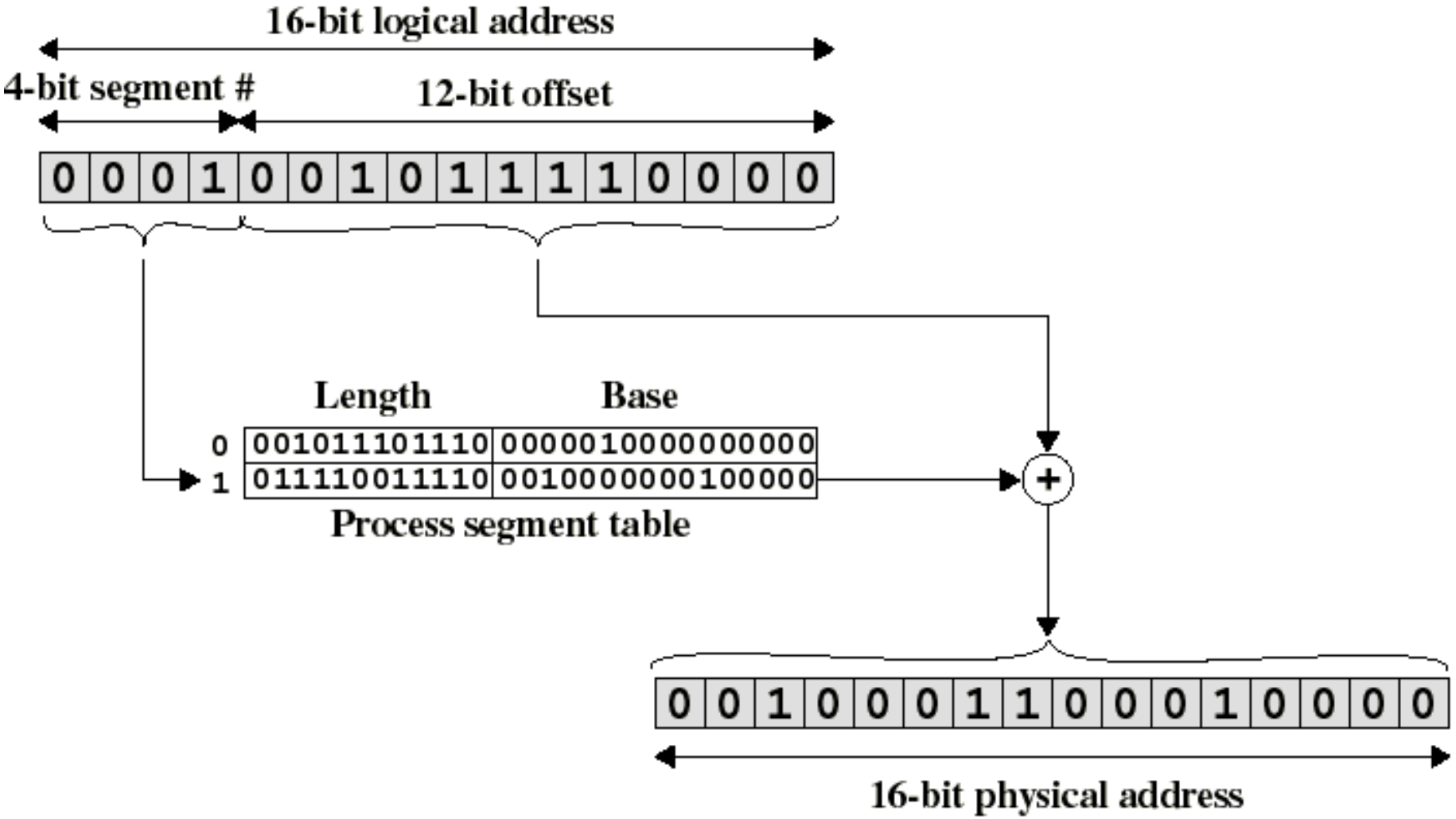
physical memory space

Phần cứng hỗ trợ phân đoạn

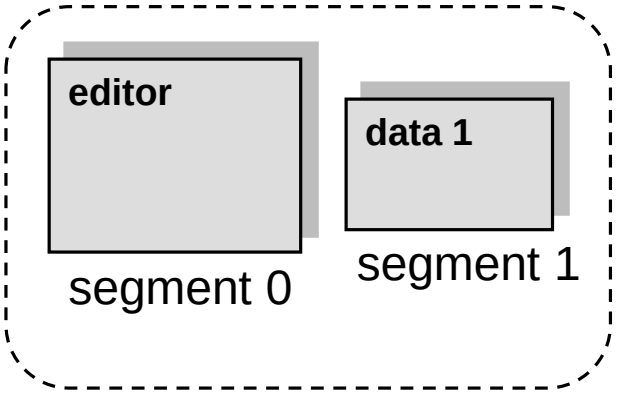


Chuyển đổi địa chỉ trong cơ chế phân đoạn

Ví dụ



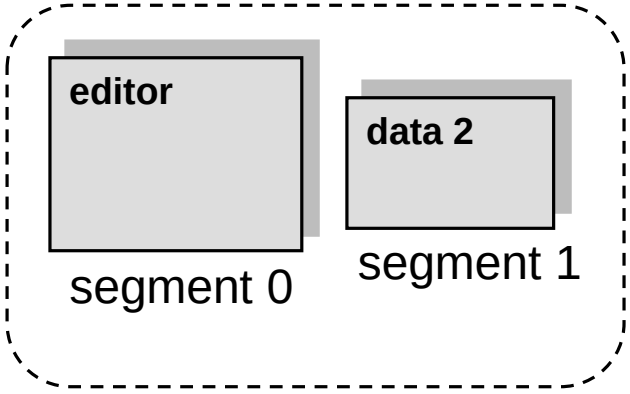
Chia sẻ các đoạn



logical address space
process P_1

	limit	base
0	25286	43062
1	4425	68348

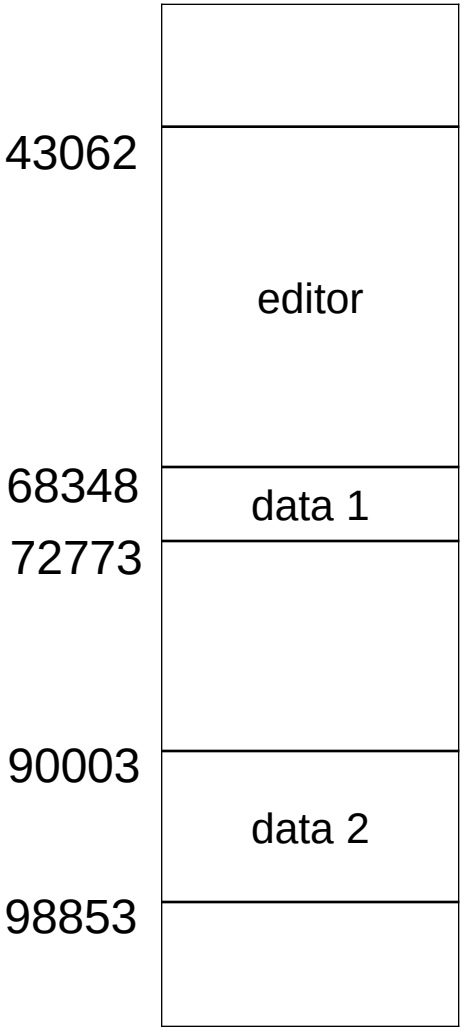
segment table
process P_1



logical address space
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2



physical memory

Kết hợp phân trang và phân đoạn

- Kết hợp phân trang và phân đoạn nhằm kết hợp các ưu điểm đồng thời hạn chế các khuyết điểm của phân trang và phân đoạn:
  - Vấn đề của phân đoạn: Nếu một đoạn quá lớn thì có thể không nạp nó được vào bộ nhớ.
  - Ý tưởng giải quyết: paging đoạn, khi đó chỉ cần giữ trong bộ nhớ các page của đoạn hiện đang cần.

Kết hợp phân trang và phân đoạn (tt)

- Có nhiều cách kết hợp. Sau đây là một cách đơn giản, gọi là *segmentation with paging*

Mỗi process sẽ có:

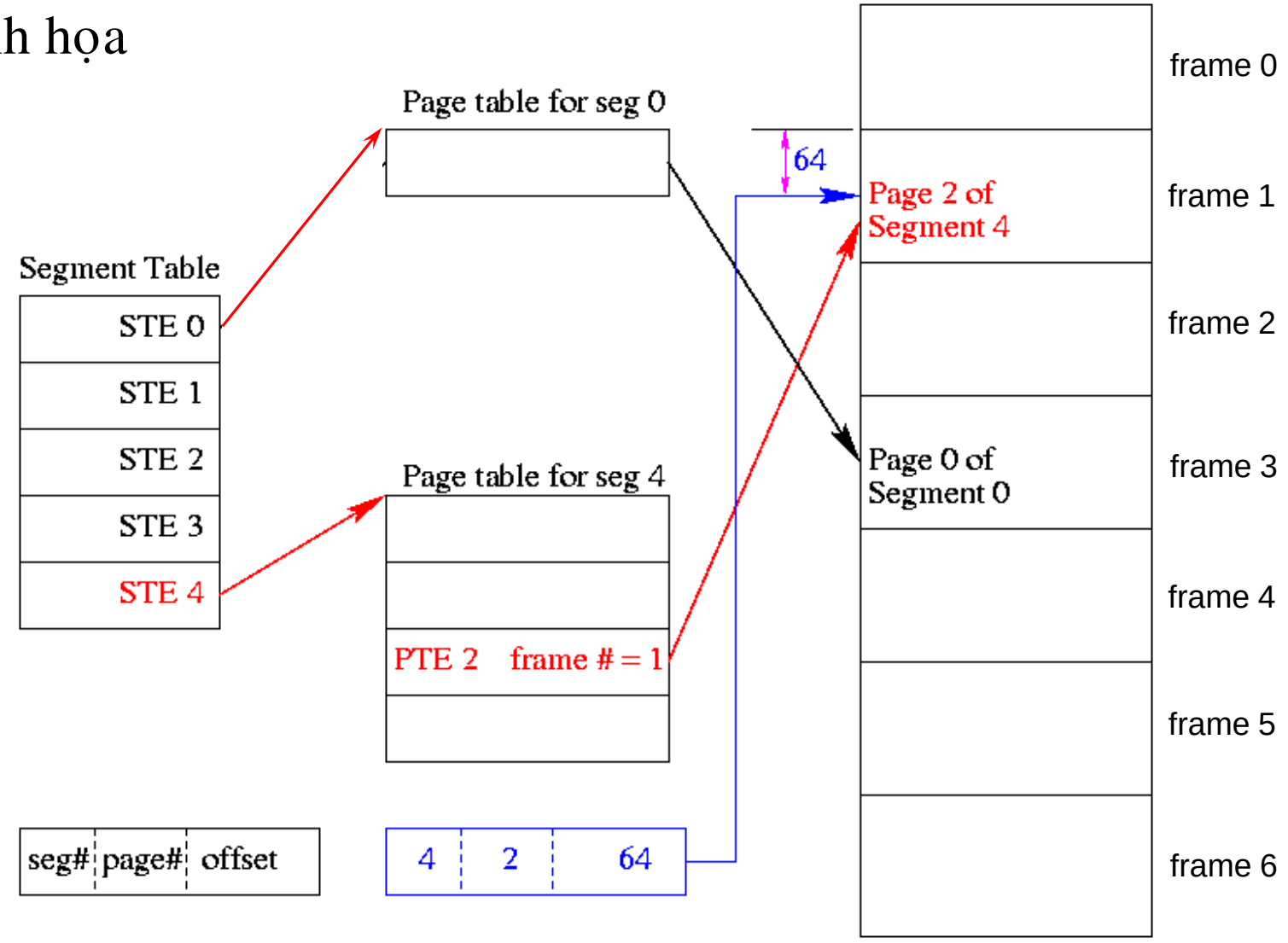
- Một bảng phân đoạn
- Nhiều bảng phân trang: mỗi đoạn có một bảng phân trang

Một *địa chỉ logić* (địa chỉ ảo) bao gồm:

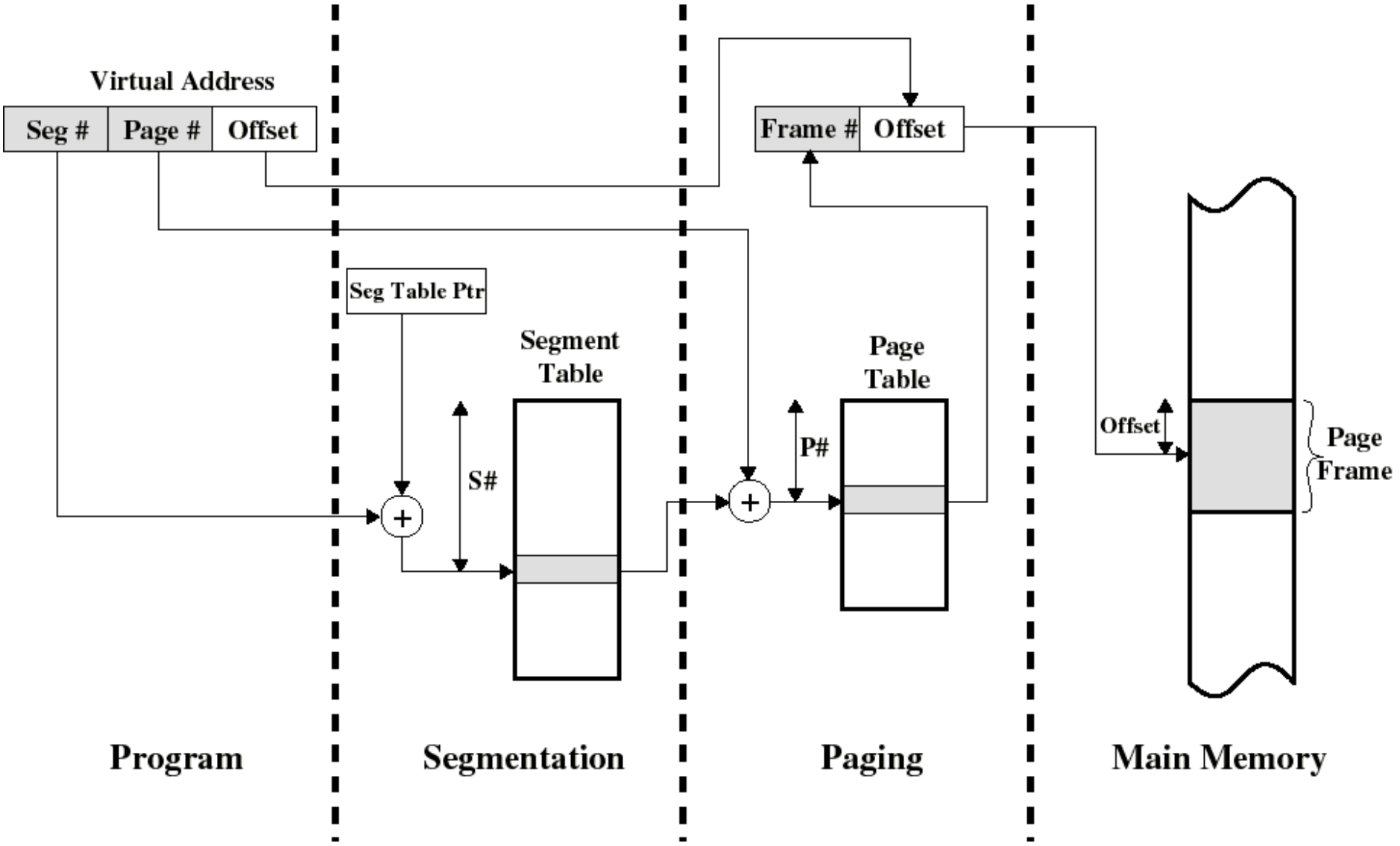
- *segment number:* là chỉ số của một mục trong bảng phân đoạn, mục này chứa địa chỉ nền (base address) của bảng phân trang cho đoạn đó
- *page number:* là chỉ số của một mục trong bảng phân trang, mục này chứa chỉ số frame trong bộ nhớ thực
- *offset:* độ dời của vị trí nhớ trong frame nói trên.

Segmentation with paging (1)

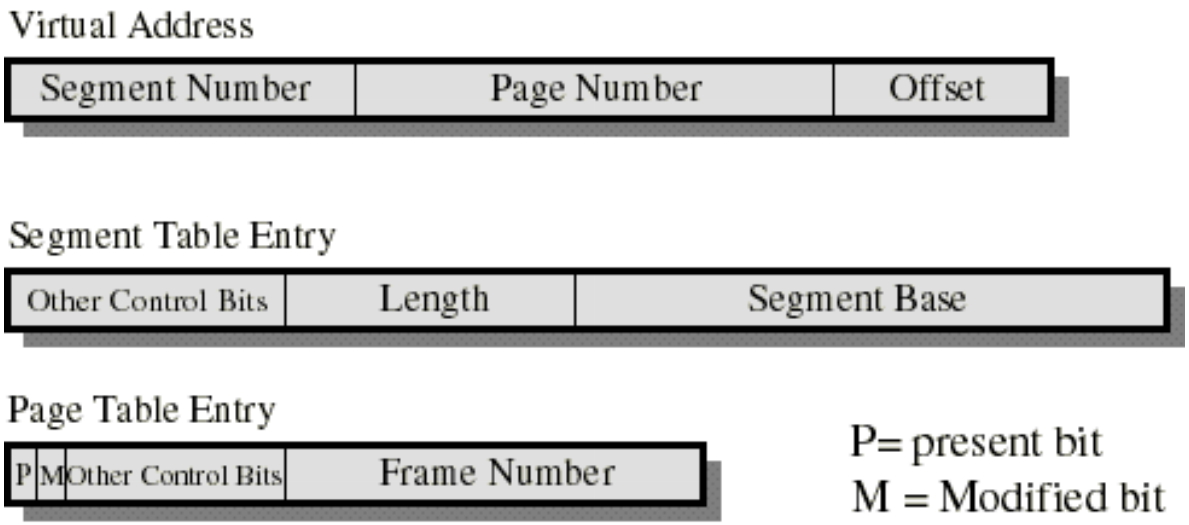
- Minh họa



Segmentation with paging (2)



Segmentation with paging (3)

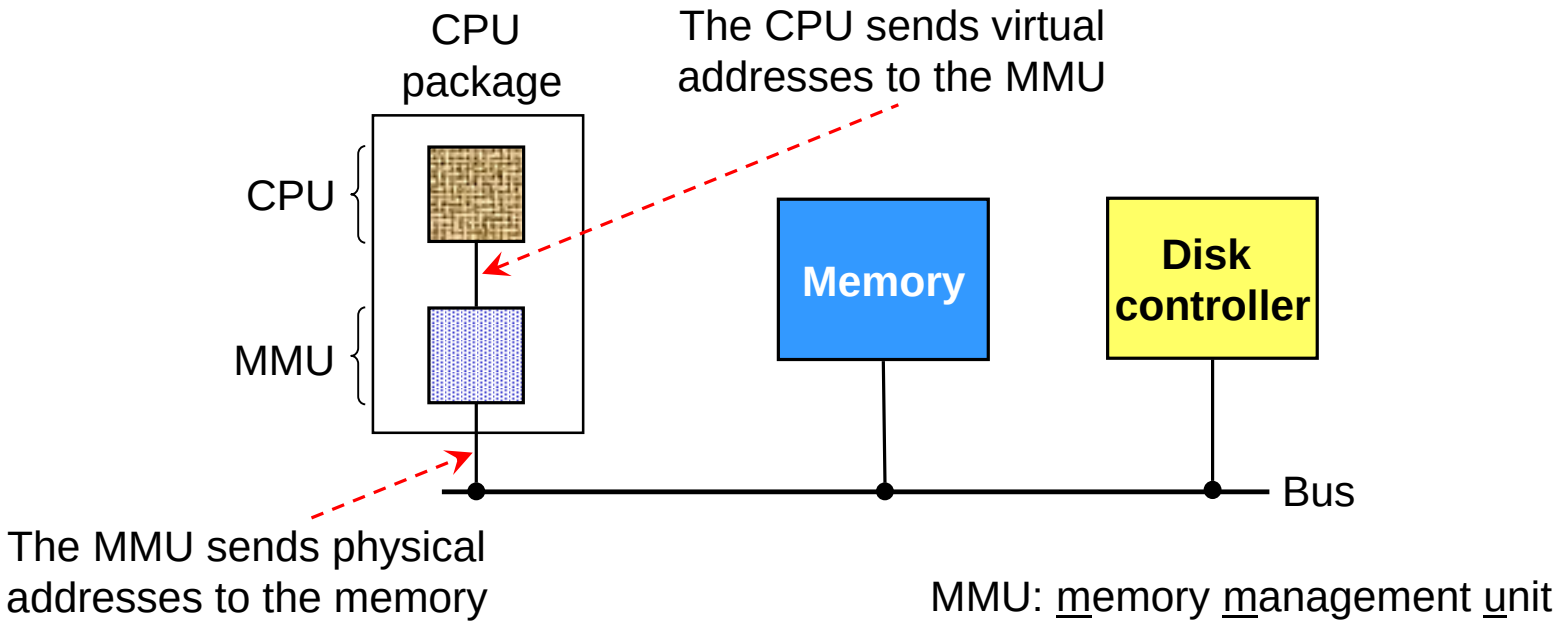


- Segment base: địa chỉ thực của bảng phân trang
- Present bit và modified bit chỉ tồn tại trong bảng phân trang
- Các thông tin bảo vệ và chia sẻ vùng nhớ thường nằm trong bảng phân đoạn
  - Ví dụ: read-only/read-write bit,...

Bộ Nhớ Ảo

Nhìn lại paging và segmentation

- Các tham chiếu đến bộ nhớ được chuyển đổi động thành địa chỉ thực lúc process đang thực thi



- Một process gồm các phần nhỏ (page hay segment), các phần này được nạp vào các vùng có thể không liên tục trong bộ nhớ chính

Bộ nhớ ảo (1)

- Nhận xét: không phải tất cả các phần của một process cần thiết phải được nạp vào bộ nhớ chính tại cùng một thời điểm

  Ví dụ
  - Đoạn mã điều khiển các lỗi hiếm khi xảy ra
  - Các arrays, list, tables được cấp phát bộ nhớ (cấp phát tĩnh) nhiều hơn yêu cầu thực sự
  - Một số tính năng ít khi được dùng của một chương trình

  Ngay cả khi toàn bộ chương trình đều cần dùng thì có thể không cần dùng toàn bộ cùng một lúc.

Bộ nhớ ảo (2)

- *Bộ nhớ ảo* (virtual memory)
  - Cơ chế được hiện thực trong hệ điều hành để cho phép thực thi một tiến trình mà chỉ cần giữ trong bộ nhớ chính một phần của không gian địa chỉ logic của nó, còn phần còn lại được giữ trên bộ nhớ phụ (đĩa).

- Ưu điểm của bộ nhớ ảo
  - Số lượng process trong bộ nhớ nhiều hơn
  - Một process có thể thực thi ngay cả khi kích thước của nó lớn hơn bộ nhớ thực

Bộ nhớ ảo (3)

- Thông thường phần của không gian địa chỉ logic của tiến trình, nếu chưa cần nạp vào bộ nhớ chính, được giữ ở một vùng đặc biệt trên đĩa gọi là *không gian tráo đổi* (swap space).

  Ví dụ:

  - swap partition trong Linux
  - file pagefile.sys trong Windows 2K

Tổng quan về hiện thực bộ nhớ ảo

- Phần cứng memory management phải hỗ trợ paging và/hoặc segmentation
- OS phải quản lý sự di chuyển của trang/đoạn giữa bộ nhớ chính và bộ nhớ thứ cấp
- Trong chương này,
  - Chỉ quan tâm đến paging
  - Phần cứng hỗ trợ hiện thực bộ nhớ ảo
  - Các giải thuật của hệ điều hành

Phần cứng hỗ trợ bộ nhớ ảo

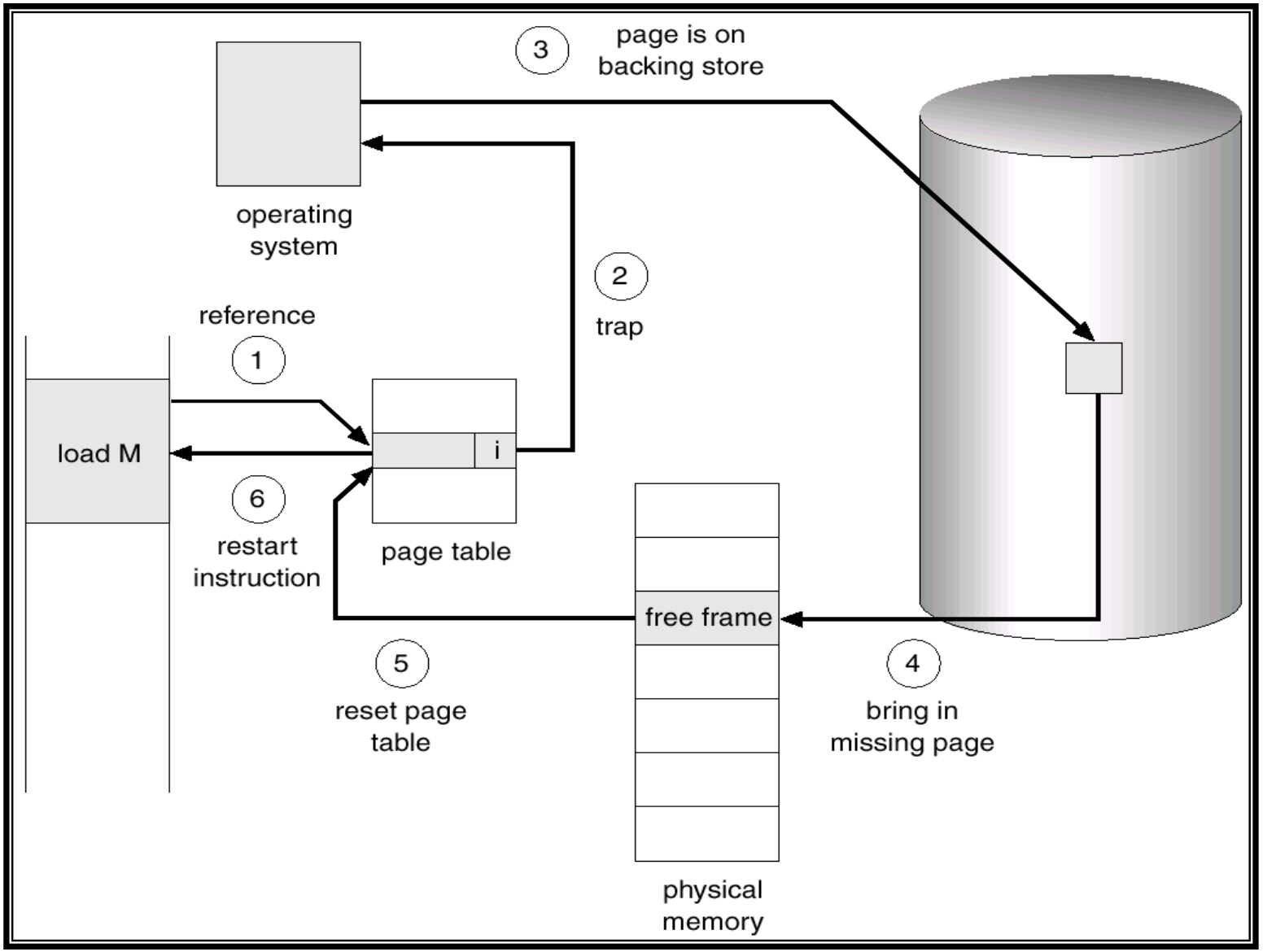
- Sự hỗ trợ của phần cứng đối với phân trang đã được khảo sát trong chương trước. Chỉ có một điểm khác biệt là mỗi mục của bảng phân trang có thêm các bit trạng thái đặc biệt
  - *Present bit* = 1 ⇒ trang hợp lệ và hiện trong memory
    = 0 ⇒ trang không hợp lệ hoặc không trong memory
  - *Modified bit:* cho biết trang có thay đổi kể từ khi được nạp vào memory hay không

Hiện thực bộ nhớ ảo: demand paging

Demand paging: các trang của tiến trình chỉ được nạp vào bộ nhớ chính khi được yêu cầu.

- Khi có một tham chiếu đến một trang mà không có trong bộ nhớ chính (present bit = 0) thì phần cứng sẽ gây ra một ngắt (gọi là *page-fault trap*) kích khởi *page-fault service routine* (PFSR) của hệ điều hành. PFSR như sau:
  1. Chuyển process về trạng thái blocked
  2. Phát ra một yêu cầu đọc đĩa để nạp trang được tham chiếu vào một frame trống; trong khi đợi I/O, một process khác được cấp CPU để thực thi
  3. Sau khi I/O hoàn tất, đĩa gây ra một ngắt đến hệ điều hành; PFSR cập nhật page table và chuyển process về trạng thái ready.

Page fault và các bước xử lý

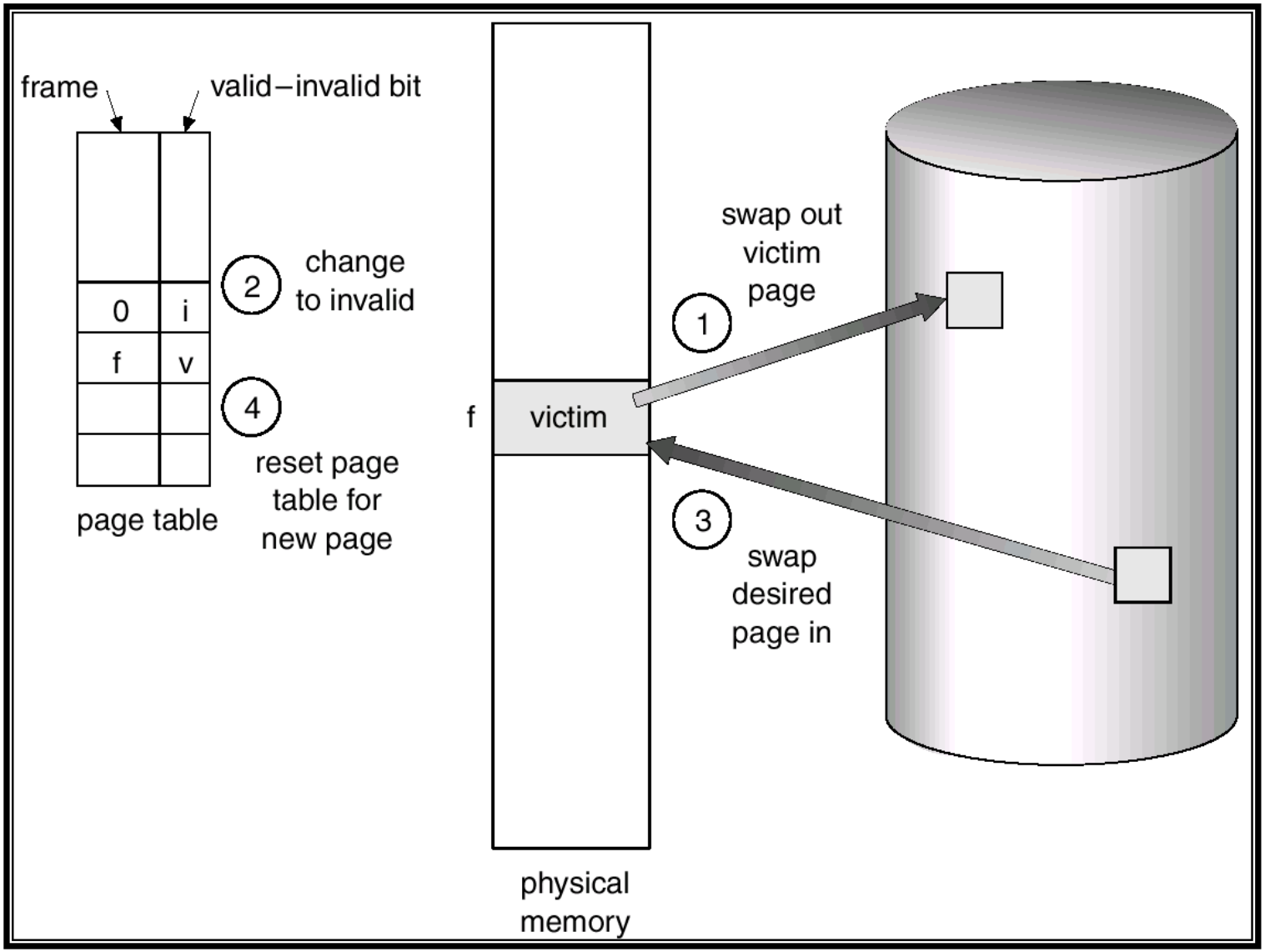


Thay thế trang nhớ (1)

- Bước 2 của PFSR giả sử tìm được frame trống. Để xử lý được cả trường hợp phải *thay trang* vì không tìm được frame trống, PFSR được bổ sung như sau:

1. Xác định vị trí trên đĩa của trang đang cần
2. Tìm một frame trống:
   a. Nếu có frame trống thì dùng nó
   b. Nếu không có frame trống thì dùng một giải thuật thay trang để chọn một *trang hy sinh* (victim page)
   c. Ghi victim page lên đĩa; cập nhật page table và frame table tương ứng
3. Đọc trang đang cần vào frame trống (đã có được từ bước 2); cập nhật page table và frame table tương ứng.

Thay thế trang nhớ (2)



Hiện thực demand paging

Hai vấn đề chủ yếu:

- Frame-allocation algorithm
  - Cấp phát cho process bao nhiêu frame của bộ nhớ thực?
- Page-replacement algorithm
  - Chọn frame của process sẽ được thay thế trang nhớ
  - Mục tiêu: số lượng page-fault nhỏ nhất
  - Được đánh giá bằng cách thực thi giải thuật đối với một *chuỗi tham chiếu bộ nhớ* (memory reference string) và xác định số lần xảy ra page fault

- Ví dụ

  Thứ tự tham chiếu các địa chỉ nhớ, với page size = 100:

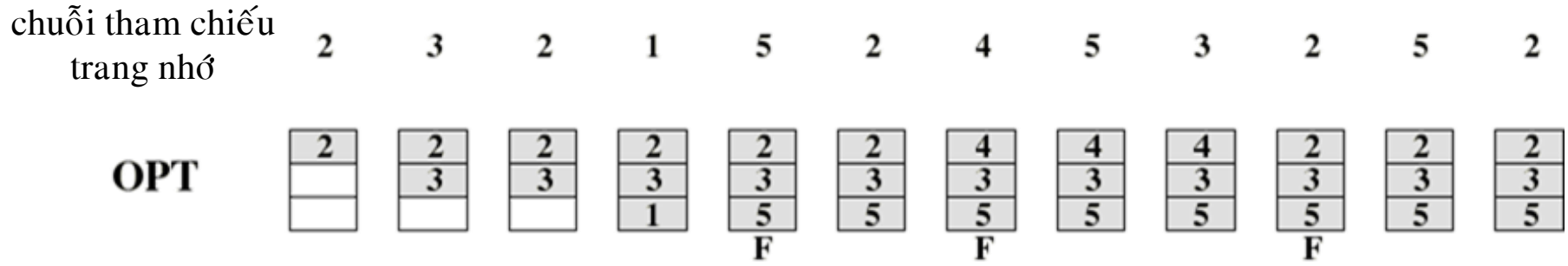
  0100, **04**32, **01**01, **06**12, **01**02, **01**03, **01**04, **01**01, **06**11, **01**02, **01**03, **01**04, **01**01, **06**10, **01**02, **01**03, **01**04, **01**01, **06**09, **01**02, **01**05

⇒ các trang nhớ sau được tham chiếu lần lượt = chuỗi tham chiếu bộ nhớ (trang nhớ)

1, **4**, **1**, **6**, **1**, **1**, **1**, **1**, **6**, **1**, **1**, **1**, **1**, **6**, **1**, **1**, **1**, **1**, **6**, **1**, **1**

Giải thuật thay trang *OPT* (Optimal Page Replacement)

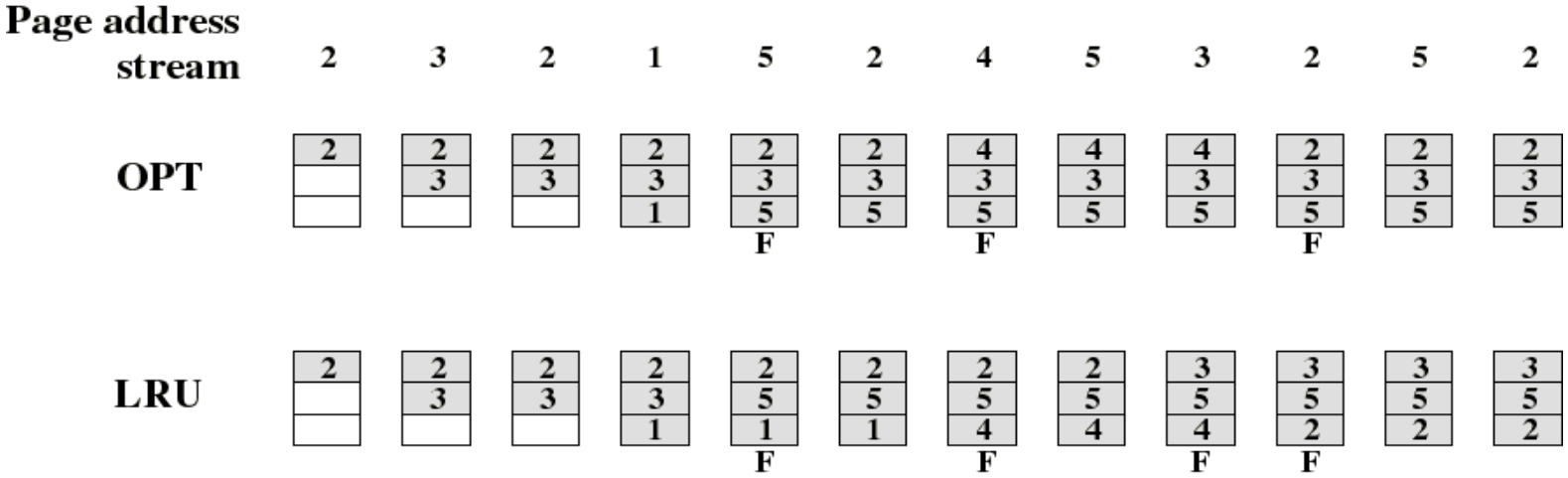
- Giải thuật thay trang OPT
  - Thay thế trang sẽ lâu được sử dụng nhất trong tương lai.
- Ví dụ: một process có 5 trang, và được cấp 3 frame



Thuật toán này bảo đảm số lượng lỗi trang phát sinh là thấp nhất , nó cũng không gánh chịu nghịch lý Belady (số lượng lỗi trang xảy ra sẽ tăng lên khi số lượng khung trang sử dụng tăng), tuy nhiên đây là một thuật toán không khả thi trong thực tế, vì không thể biết trước chuỗi truy xuất của tiến trình!

Giải thuật thay trang *Least Recently Used* (LRU)

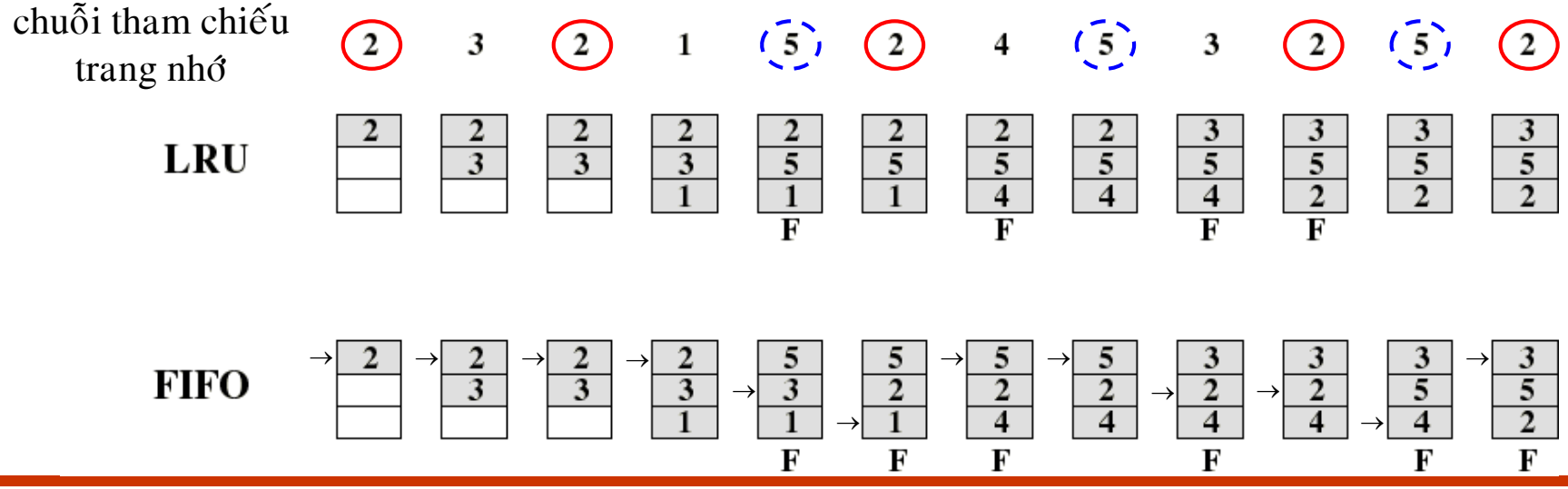
- Thay thế trang nhớ lâu nhất chưa được sử dụng (tính lùi so với OPT)
- (Với mỗi trang, ghi nhận thời điểm cuối cùng trang được truy cập, trang được chọn để thay thế sẽ là trang lâu nhất chưa được truy xuất)
- Ví dụ: một process có 5 trang, và được cấp 3 frame



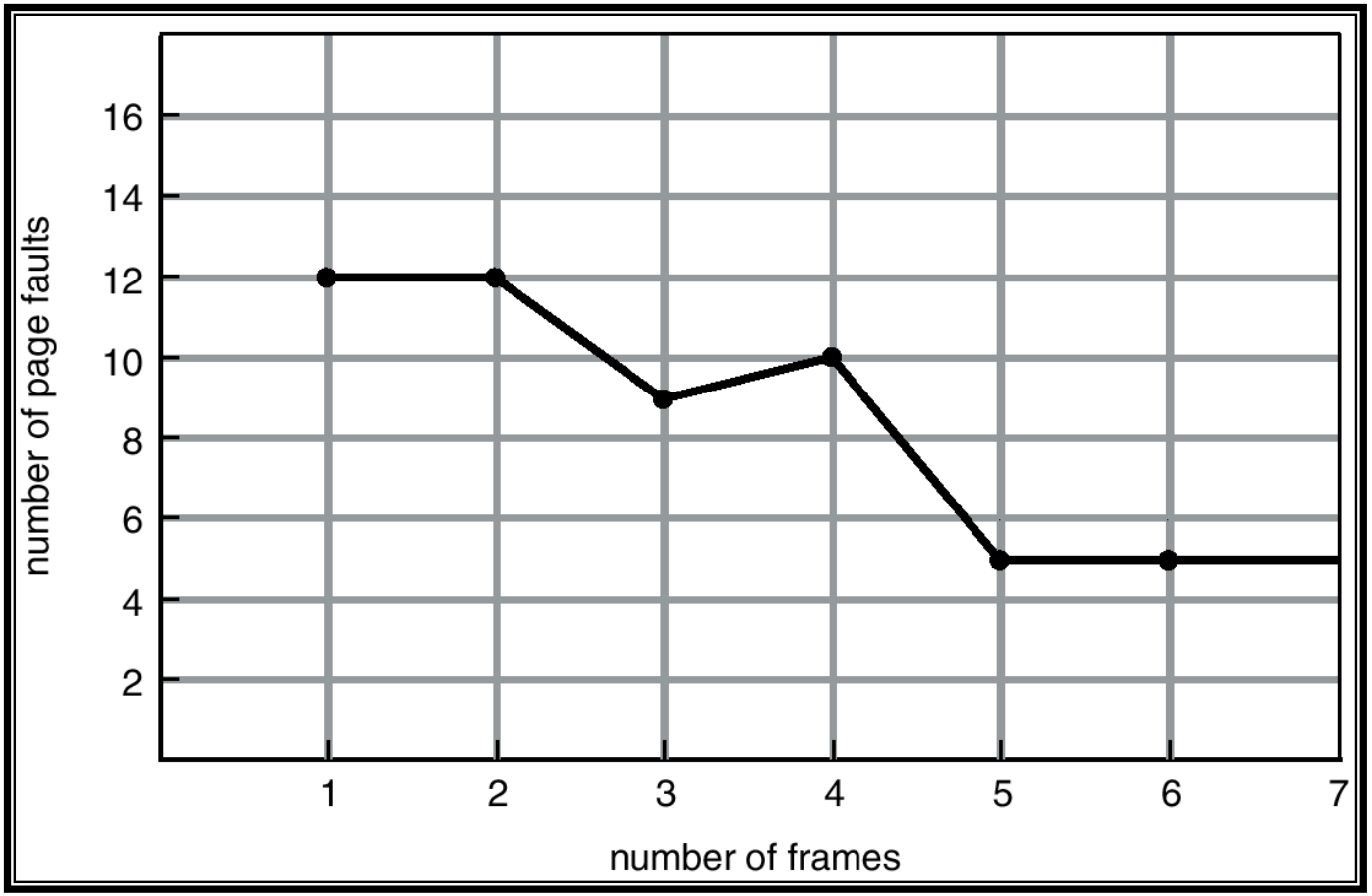
- Mỗi trang được ghi nhận (trong bảng phân trang) thời điểm được tham chiếu ⇒ trang LRU là trang nhớ có thời điểm tham chiếu nhỏ nhất (OS tốn chi phí tìm kiếm trang nhớ LRU này mỗi khi có page fault)
- Do vậy, LRU cần sự hỗ trợ của phần cứng (bộ đếm, hoặc stack) và chi phí cho việc tìm kiếm. Ít CPU cung cấp đủ sự hỗ trợ phần cứng cho giải thuật LRU.

Giải thuật thay trang *FIFO*

- Xem các frame được cấp phát cho process như là circular buffer
  - Khi bộ đệm đầy, trang nhớ cũ nhất sẽ được thay thế: first-in first-out
  - Một trang nhớ hay được dùng sẽ thường là trang cũ nhất ⇒ hay bị thay thế bởi giải thuật FIFO
  - Hiện thực đơn giản: chỉ cần một con trỏ xoay vòng các frame của process
- So sánh các giải thuật thay trang LRU và FIFO



Giải thuật FIFO: *Belady's anomaly*

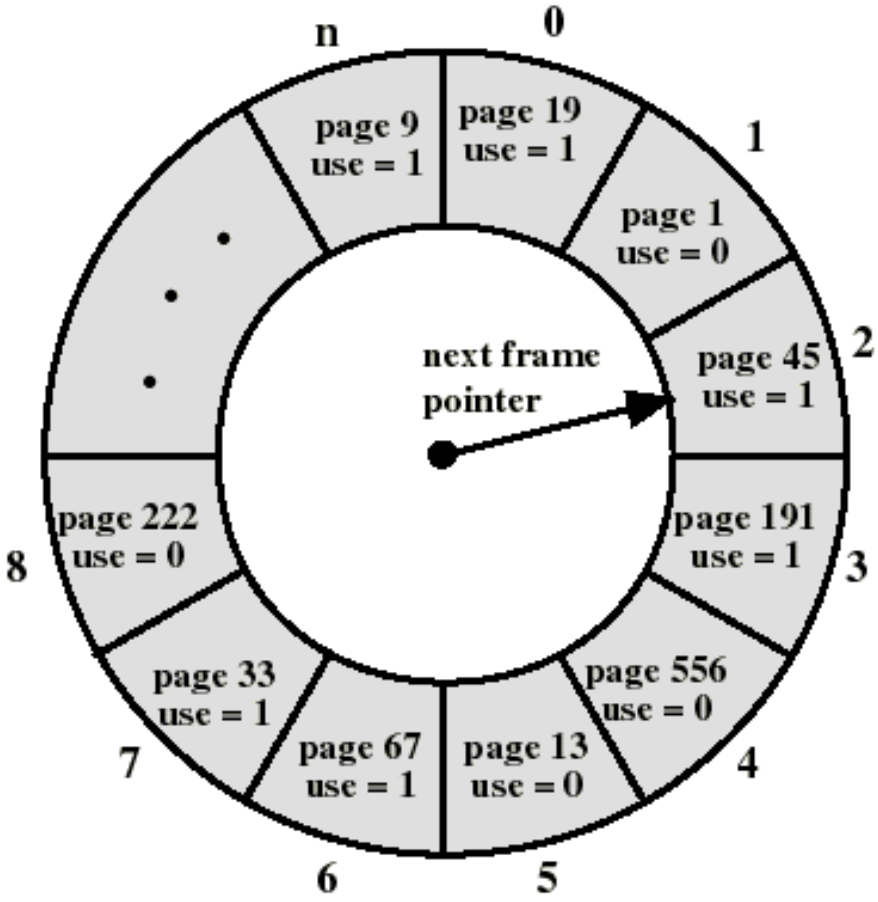


Bất thường (anomaly) Belady: số page fault tăng mặc dầu tiến trình đã được cấp nhiều frame hơn.

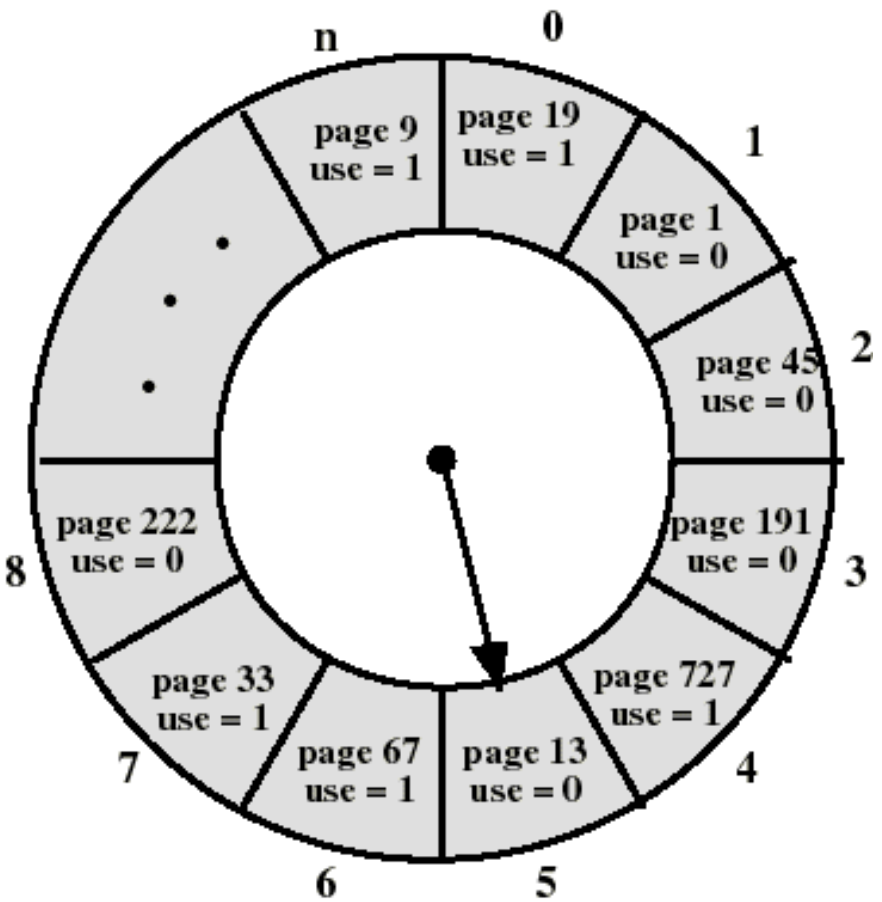
Giải thuật thay trang *clock* (1)

- Các frame cấp cho process được xem như một bộ đệm xoay vòng (circular buffer)
- Khi một trang được thay, con trỏ sẽ chỉ đến frame kế tiếp trong buffer
- Mỗi frame có một *use bit*. Bit này được thiết lập trị 1 khi
  - Một trang được nạp lần đầu vào frame
  - Trang chứa trong frame được tham chiếu
- Khi cần thay thế một trang nhớ, trang nhớ nằm trong frame đầu tiên có use bit bằng 0 sẽ được thay thế.
  - Trên đường đi tìm trang nhớ thay thế, tất cả use bit được reset về 0

Giải thuật thay trang clock (2)

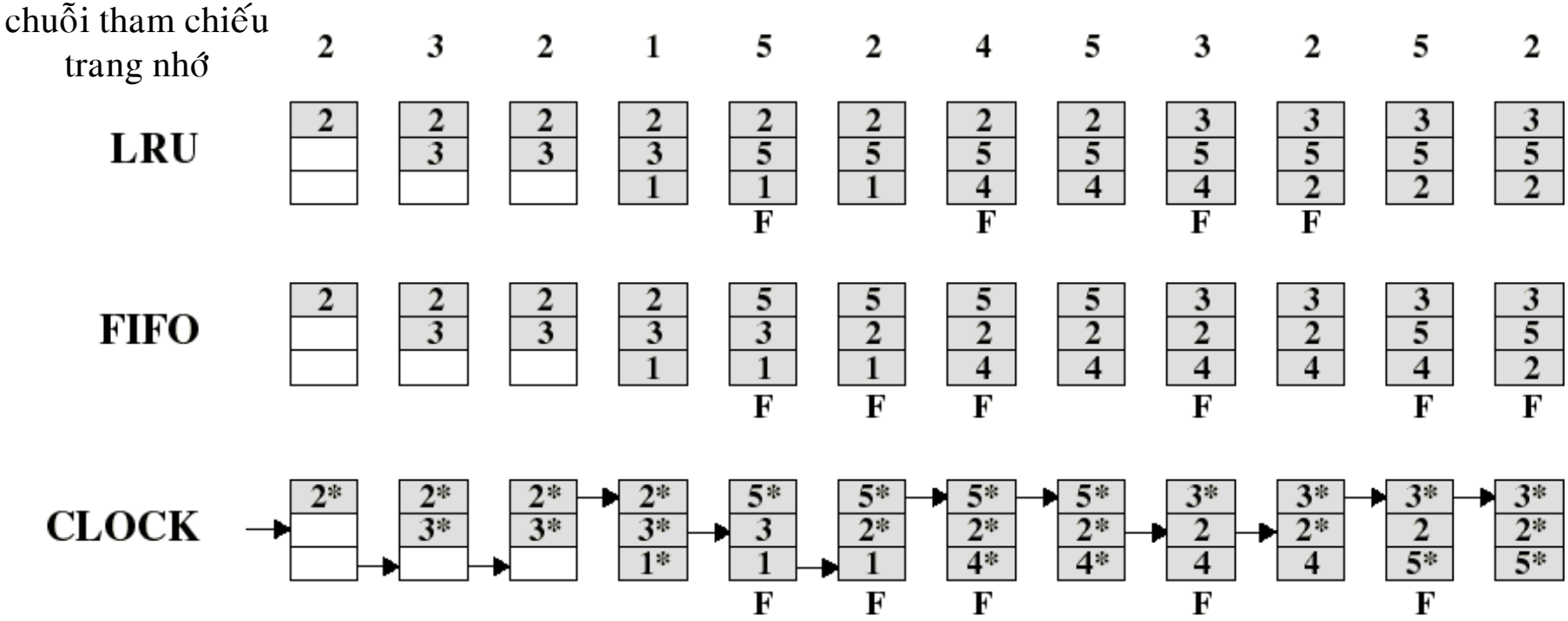


(a) State of buffer just prior to a page replacement



(b) State of buffer just after the next page replacement

So sánh LRU, FIFO, và clock



- Dấu *: use bit tương ứng được thiết lập trị 1
- Giải thuật clock bảo vệ các trang thường được tham chiếu bằng cách thiết lập use bit bằng 1 với mỗi lần tham chiếu
- Một số kết quả thực nghiệm cho thấy clock có hiệu suất gần với LRU

Số lượng frame cấp cho process

- OS phải quyết định cấp cho mỗi process bao nhiêu frame.
  - Cấp ít frame ⇒ nhiều page fault
  - Cấp nhiều frame ⇒ giảm mức độ multiprogramming

- Chiến lược cấp phát tĩnh (fixed-allocation)
  - Số frame cấp cho mỗi process không đổi, được xác định vào thời điểm loading và có thể tùy thuộc vào từng ứng dụng (kích thước của nó,...)
- Chiến lược cấp phát động (variable-allocation)
  - Số frame cấp cho mỗi process có thể thay đổi trong khi nó chạy
    - Nếu tỷ lệ page-fault cao ⇒ cấp thêm frame
    - Nếu tỷ lệ page-fault thấp ⇒ giảm bớt frame
  - OS phải mất chi phí để ước định các process

Chiến lược cấp phát tĩnh

- *Cấp phát bằng nhau:* Ví dụ, có 100 frame và 5 process
  → mỗi process được 20 frame
- *Cấp phát theo tỉ lệ:* dựa vào kích thước process

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

Ví dụ:

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

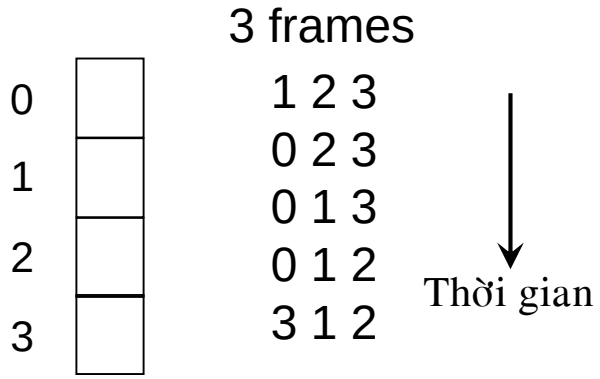
$a_2 = \frac{127}{137} \times 64 \approx 59$

Thrashing

- Nếu một process không có đủ số frame cần thiết thì tỉ số page faults/sec rất cao. Điều này khiến giảm hiệu suất CPU rất nhiều.

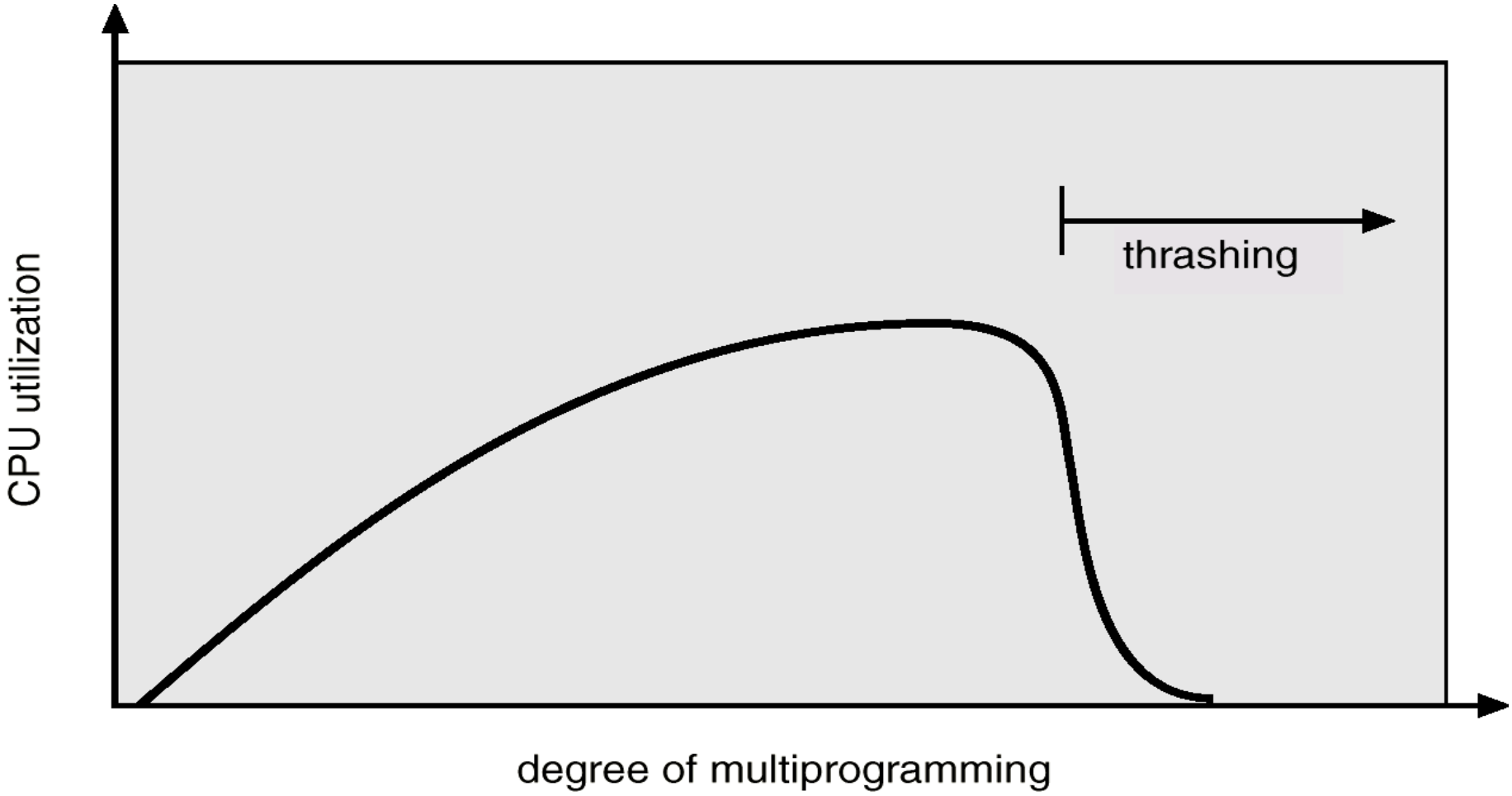
  Ví dụ: một vòng lặp N lần, mỗi lần tham chiếu đến địa chỉ nằm trong 4 trang nhớ trong khi đó process chỉ được cấp 3 frames.

Process có 4 trang, được cấp phát 3 frame
Chuỗi tham chiếu trang:
123023013012312



- *Thrashing:* hiện tượng các trang nhớ của một process bị hoán chuyển vào/ra liên tục.

Thrashing diagram



Nguyên lý locality

- Để hạn chế thrashing, hệ điều hành phải cung cấp cho process càng "đủ" frame càng tốt. Bao nhiêu frame thì đủ cho một process thực thi hiệu quả?

  Nguyên lý locality (locality principle)

  - *Locality* là tập các trang được tham chiếu gần nhau
    - Trong ví dụ trước, locality sẽ bao gồm 4 trang
  - Một process gồm nhiều locality, và trong tiến trình thực thi, process sẽ chuyển từ locality này sang locality khác
    - Ví dụ khi một thủ tục được gọi thì sẽ có một locality mới. Trong locality này, tham chiếu bộ nhớ bao gồm lệnh của thủ tục, biến cục bộ và một phần biến toàn cục. Khi thủ tục kết thúc, process sẽ thoát khỏi locality này (và có thể quay lại sau này).

- Vì sao hiện tượng thrashing xuất hiện?

  Khi Σ size of locality > memory size

- The working set idea was based on an implicit assumption that the pages seen in the backward window were highly likely to be used again in the immediate future. (P. Denning)

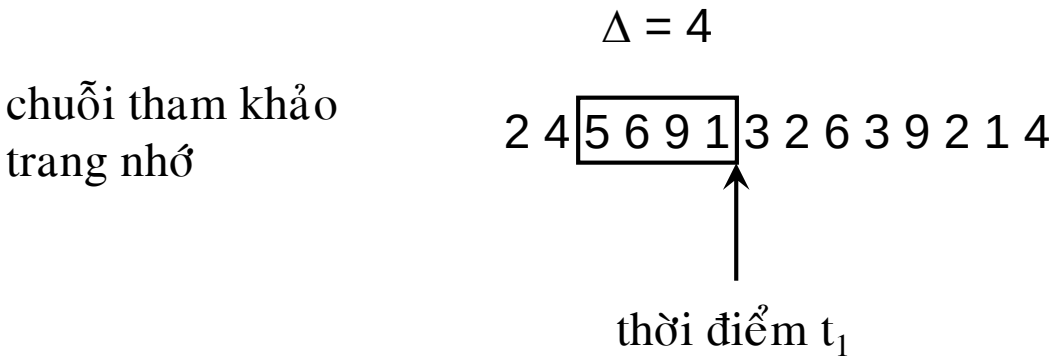
Hạn chế thrashing: Giải pháp working set (1)

Còn được gọi là *working set model*.

Được thiết kế dựa trên nguyên lý locality.

- Xác định xem process thực sự sử dụng bao nhiêu frame.
- Định nghĩa: tham số Δ của *working-set window* ≡ tham số xác định số lượng các tham chiếu trang nhớ của process gần đây nhất cần được quan sát.

Ví dụ:



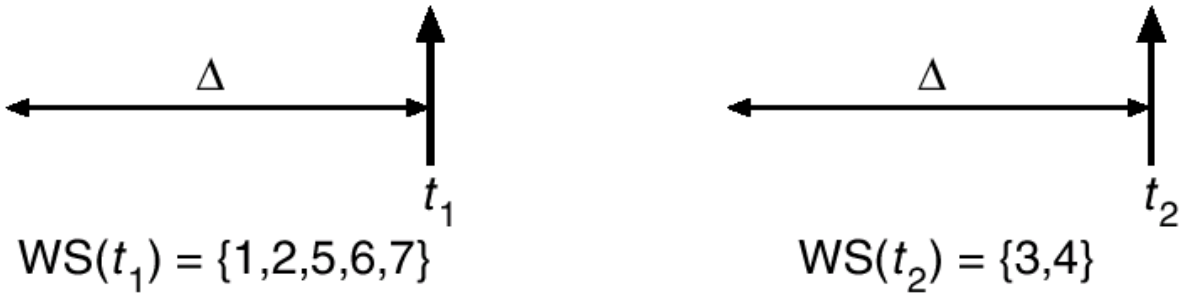
Hạn chế thrashing: Giải pháp working set (2)

- Định nghĩa: *working set của process P_i* , ký hiệu WS_i , là tập gồm Δ các trang được sử dụng gần đây nhất.

Ví dụ: Δ = 10 ỹ

chuỗi tham khảo trang

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



- Nhận xét:

  Δ quá nhỏ ⇒ không đủ bao phủ toàn bộ locality.

  Δ quá lớn ⇒ bao phủ nhiều locality khác nhau.

  Δ = ∞ ⇒ bao gồm tất cả các trang được sử dụng.

  Dùng working set của một process để xấp xỉ locality của nó.

Hạn chế thrashing: Giải pháp working set (3)

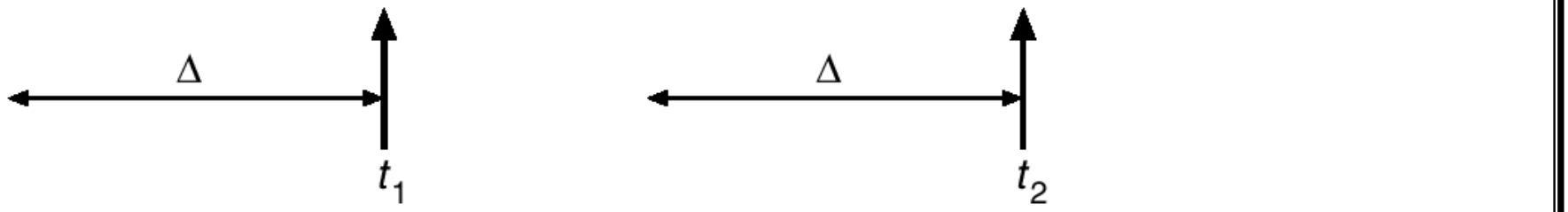
Định nghĩa WSS_i là kích thước của working set của P_i :

WSS_i = số lượng các trang trong WS_i

Ví dụ (tiếp): $\Delta = 10$ ỳ

chuỗi tham khảo trang

. . . 2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 . . .



$WS(t_1) = \{1,2,5,6,7\}$

$WSS(t_1) = 5$

$WS(t_2) = \{3,4\}$

$WSS(t_2) = 2$

Hạn chế thrashing: Giải pháp working set (4)

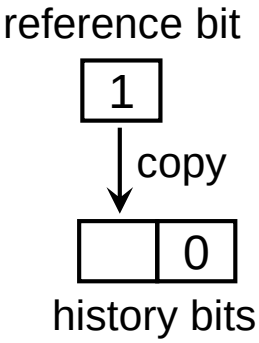
Đặt $D = \sum WSS_i$ = tổng các working-set size của mọi process trong hệ thống.

- Nhận xét: Nếu D > m (số frame của hệ thống) ⇒ sẽ xảy ra thrashing.

- *Giải pháp working set:*
  - Khi khởi tạo một tiến trình: cung cấp cho tiến trình số lượng frame thỏa mản working-set size của nó.
  - Nếu D > m ⇒ suspend một trong các process.
    - Các trang của tiến trình được chuyển ra đĩa cứng và các frame của nó được thu hồi.

Xấp xỉ working set

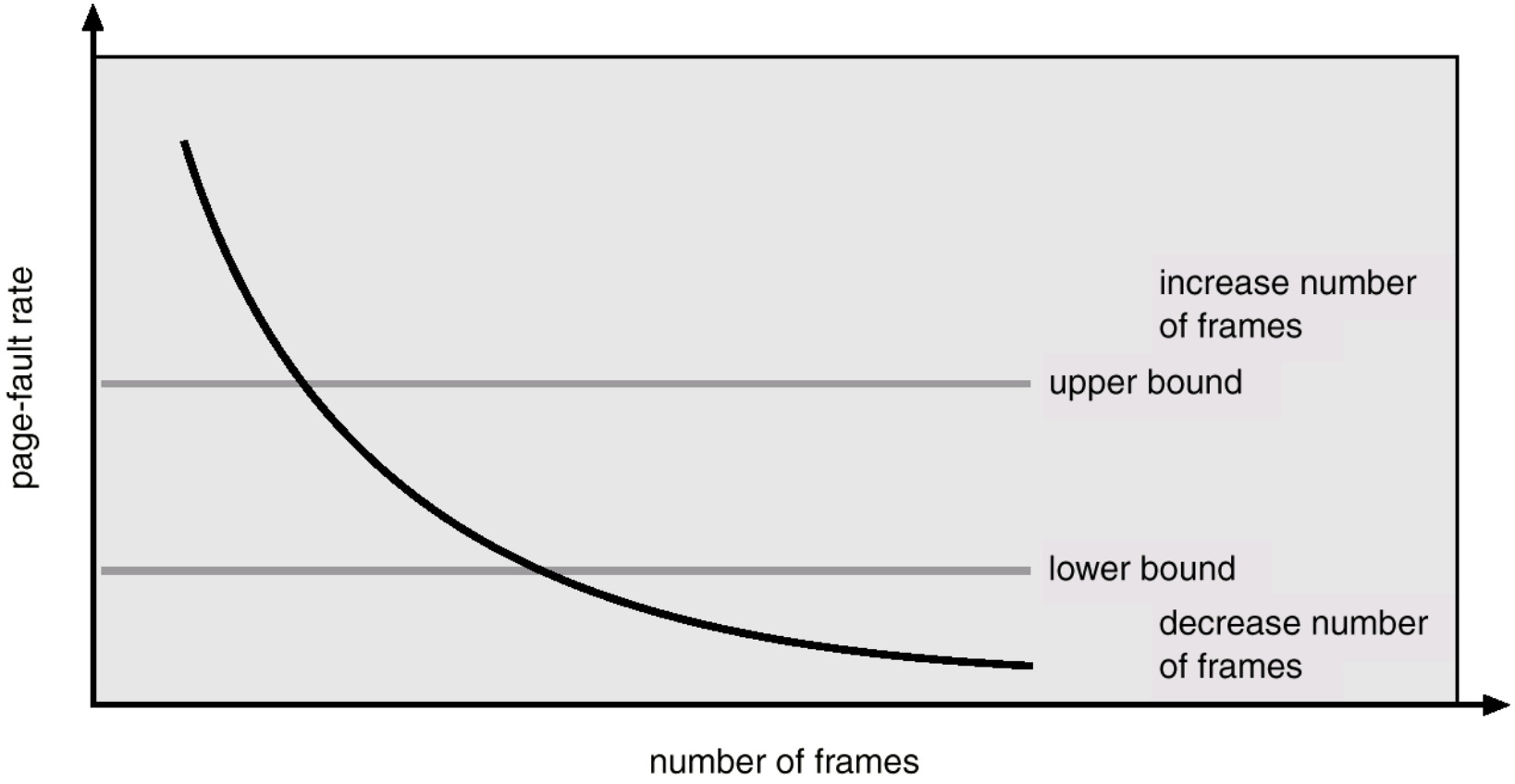
- Theo định nghĩa, một trang là ở trong working set nếu nó được tham chiếu trong working-set window
- Giả sử hardware hỗ trợ một *reference bit* cho mỗi page: khi page được tham chiếu, reference bit được set thành 1.
- Dùng interval timer kết hợp với reference bit để xấp xỉ working set



- Ví dụ: $\Delta = 10.000$
  - Timer interrupt định kỳ, sau mỗi 5000 tham chiếu.
  - Giữ trong bộ nhớ 2 bit (*history bits*) cho mỗi trang nhớ.
  - Khi timer interrupt xảy ra, shift history bits một vị trí sang phải, copy reference bit vào history bit trái, và reset reference bit = 0.
  - Trang nào có history bits chứa 1 thì thuộc về working set.

Để xấp xỉ chính xác hơn: ví dụ dùng 10 history bit và interrupt timer định kỳ sau mỗi 1000 tham chiếu.

Hạn chế thrashing: Điều khiển page-fault rate



- Dùng giải thuật PFF (Page-Fault Frequency) để điều khiển *page-fault rate* (số page-faults/sec) của process:
  - Page-fault rate quá thấp: process có quá nhiều frame → giảm số frame.
  - Page-fault rate quá cao: process cần thêm frame → cấp thêm frame.

Bài tập 1

- Một hệ thống máy tính với bộ nhớ chính có kích thước 320MB. Hệ thống sử dụng địa chỉ logic 48 bit. Kích thước trang được sử dụng là 8K. Yêu cầu xác định các thông số sau:
  a. Cho biết số bit dùng cho địa chỉ offset.
  b. Số khung trang vật lý.
  c. Số trang logic trong không gian tiến trình.
  d. Cho địa chỉ logic 20030, yêu cầu đổi sang dạng <p, d>.

Đáp án:

a) Số bit dùng cho địa chỉ offset?

Là số bit cần dùng để mô tả tất cả các địa chỉ trong một trang

Kích thước một trang: 8K = 8192B = 2^{13}B ➔ Số bit cần dùng là 13 bit.

b) Số khung trang vật lý

$$\frac{\text{Kích thước bộ nhớ vật lý}}{\text{Kích thước trang}} = \frac{320\ \text{MB}}{8\ ^{\text{KB}}/_{\text{trang}}} = \frac{320 * 2^{20}\ \text{B}}{8 * 2^{10}\ ^{\text{B}}/_{\text{trang}}} = 40 * 2^{10} = 40960\ \text{trang}$$

c) Số trang logic trong không gian tiến trình

$$\frac{\text{Kích thước không gian tiến trình}}{\text{Kích thước trang}} = \frac{2^{48}\ \text{B}}{8\ ^{\text{KB}}/_{\text{trang}}} = \frac{2^{48}\ \text{B}}{2^{13}\ ^{\text{B}}/_{\text{trang}}} = 2^{25}\ \text{trang}$$

d) Đổi địa chỉ 20030 sang dạng <p, d>

Do kích thước trang là 8192, lấy 20030 chia cho 8192 được 2 dư 3646.

20030 được đổi thành <p = 2, d = 3646>.

Bài tập 2

- Một máy tính sử dụng địa chỉ logic 64bit có dung lượng bộ nhớ 64MB. Hệ điều hành sử dụng 12 bit để làm địa chỉ offset. Yêu cầu tính số trang logic, số trang vật lý và kích thước trang.