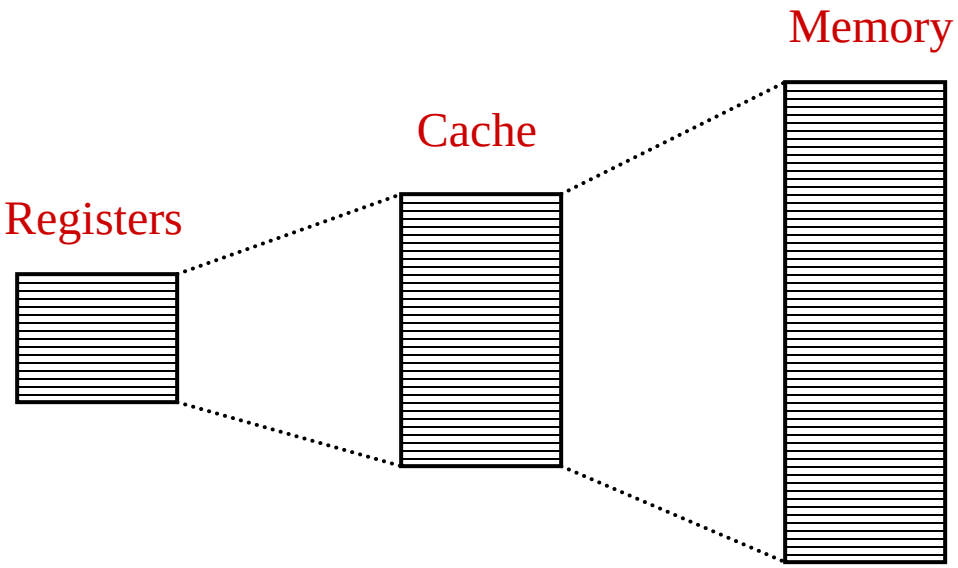
Quản lý bộ nhớ

TH 106: Hệ điều hành
Khoa CNTT
ĐH KHTN

Phân cấp bộ nhớ



Khái niệm cache

Các đặc điểm chung

- Truy suất nhanh
- Giảm tần xuất truy cập bộ nhớ
- Tăng dung lượng phục vụ của bộ xử lí chính
- Tăng kích thước đơn vị dữ liệu

Caches bộ nhớ

Ở gần processor hơn là bộ nhớ chính

Nhỏ và nhanh hơn bộ nhớ chính

Như là "bộ nhớ tạm": chứa giá trị vùng nhớ trên bộ nhớ chính nơi mới vừa truy cập.

Chuyển đổi dữ liệu giữa cache và bộ nhớ chính được tính theo đơn vị: blocks/lines

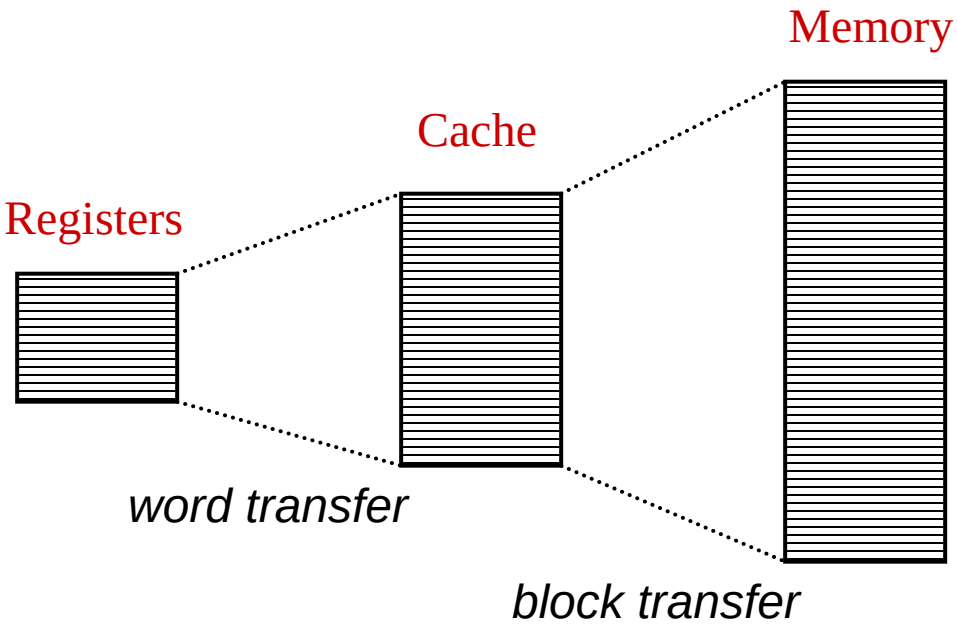
- Caches cũng chứa giá trị ô nhớ ở gần với ô nhớ vừa được truy xuất

Ánh xạ giữa bộ nhớ và cache là ánh xạ tĩnh (hầu hết)

- Xử lí nhanh khi xảy ra lỗi trang

Thông thường là có một cache chính và nhiều caches phụ (L1, L2, L3, ...)

Các vấn đề trong thiết kế Cache

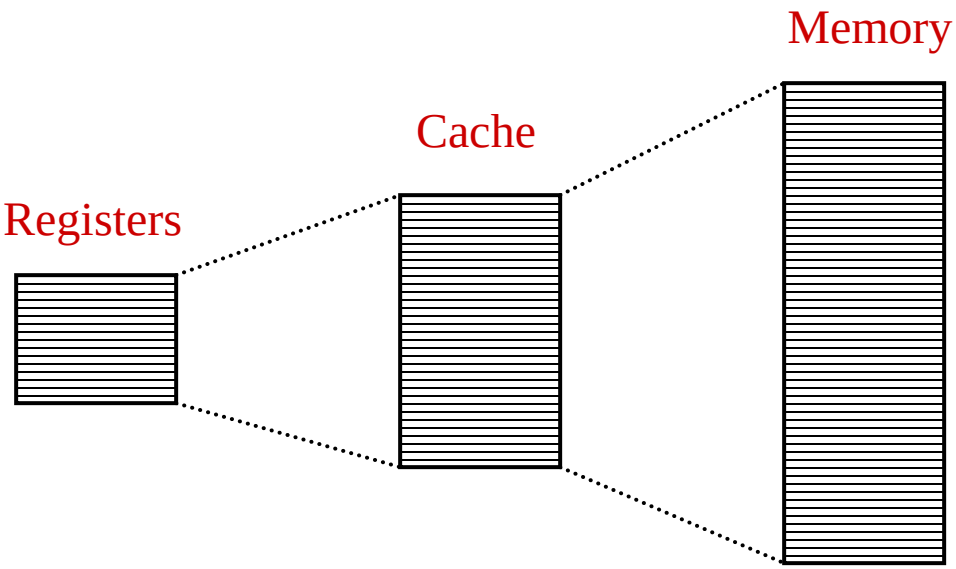


Kích thước cache và kích thước cache block

Ánh xạ: physical/virtual caches

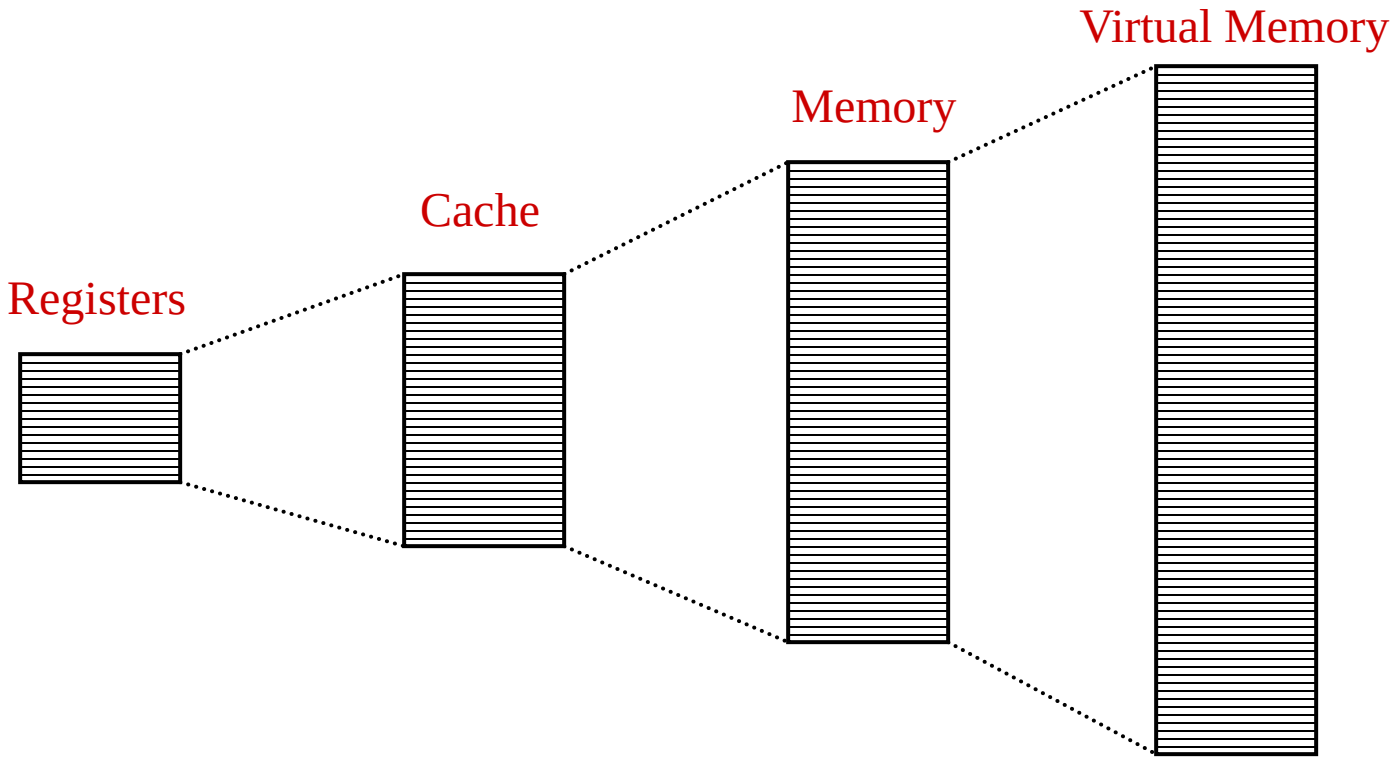
Thuật toán thay thế

Phân cấp bộ nhớ



Câu hỏi: Phải làm gì nếu ta muốn thực thi chương trình mà yêu cầu bộ nhớ lớn hơn bộ nhớ ta đang có sẵn?

Phân cấp bộ nhớ

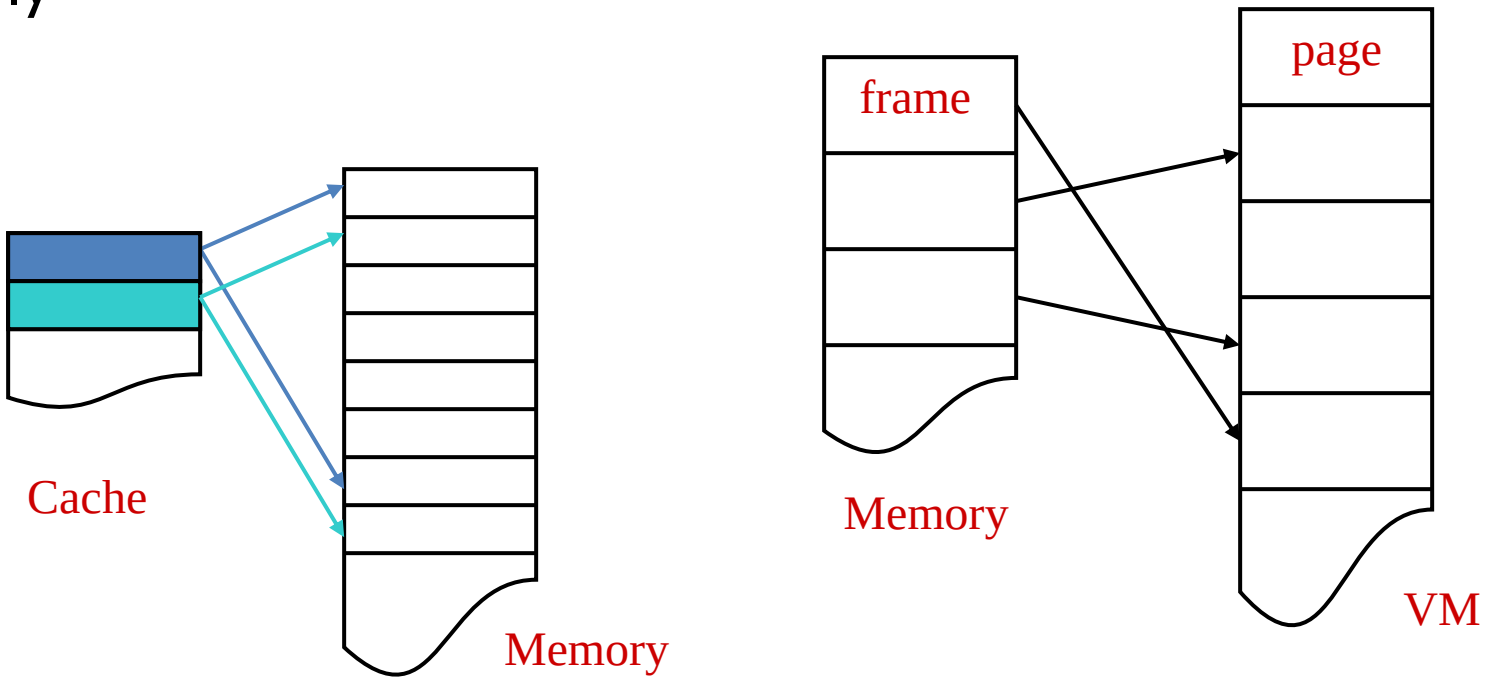


Trả lời: Giả lập như chúng ta có bộ nhớ lớn hơn: Bộ nhớ ảo

Bộ nhớ ảo: phân trang

Một trang là một đơn vị của bộ nhớ ảo (cache được)

HĐH quản lý việc ánh xạ giữa các trang của VM và bộ nhớ vật lý



Hai quan điểm về bộ nhớ

Nhìn từ phần cứng – chia sẻ bộ nhớ vật lý

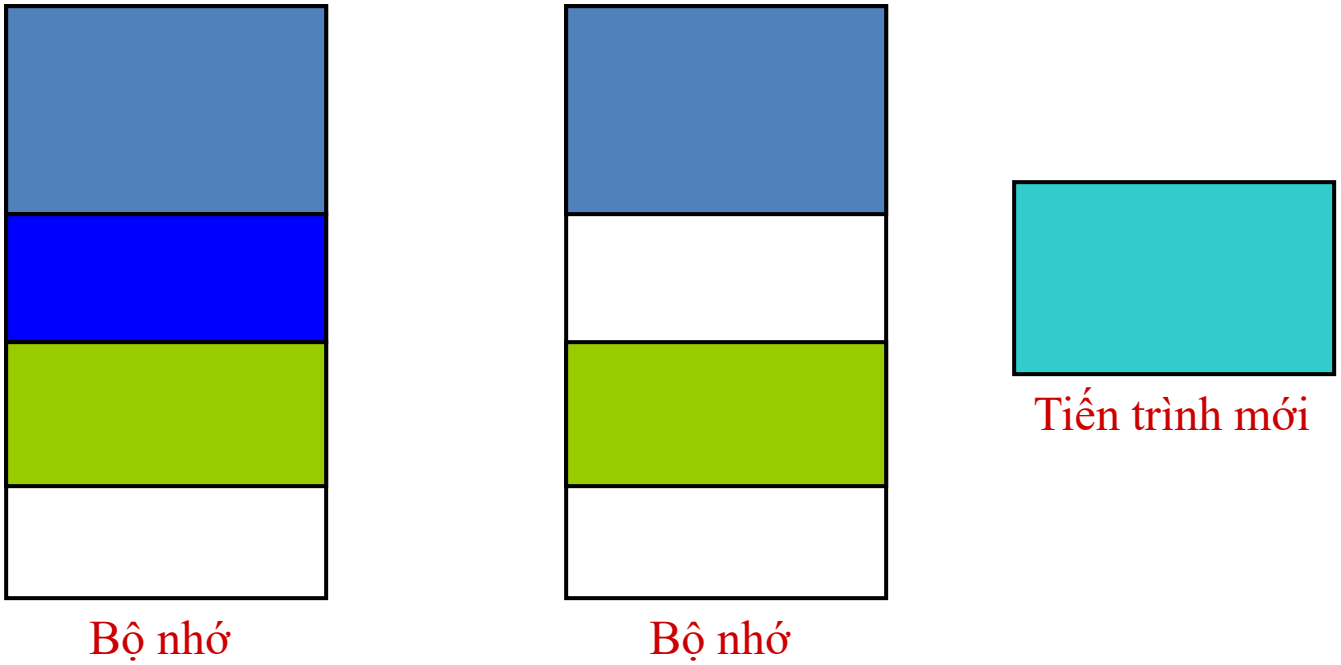
Nhìn từ phần mềm – một tiến trình sẽ chỉ "thấy": không gian địa chỉ ảo của nó

Quản lý bộ nhớ của HĐH là kết hợp hai cách nhìn trên

- Bền vững (Consistency): các bộ nhớ vật lý trông "giống nhau"
- Cấp phát địa chỉ (Relocation): tiến trình có thể được nạp lên tại bất kì địa chỉ vật lý nào
- Bảo vệ (Protection): một tiến trình không thể truy cập vùng nhớ của tiến trình khác
- Chia sẻ (Sharing): cho phép chia sẻ bộ nhớ vật lý (phải cài đặt điều khiển)

Bộ nhớ bị phân mảnh

Vấn đề phân mảnh trong môi trường đa chương



Phân mảnh

Phân mảnh ngoại vi (External Fragmentation) – tổng bộ nhớ trống thỏa yêu cầu, nhưng không liên tục

Phân mảnh nội vi (Internal Fragmentation) – mỗi block được cấp phát lớn hơn yêu cầu bộ nhớ một ít

Giải pháp phân mảnh ngoại vi: **kết hợp**

- Chuyển các vùng trống thành một khối bộ nhớ liên tục
- Chỉ thực hiện được nếu HĐH hỗ trợ biên dịch địa chỉ trong thời gian thực thi

Bài toán cấp phát bộ nhớ động

Cấp phát bộ nhớ kích thước X được thực hiện như thế nào?

- First-fit: cấp phát vùng trống đầu tiên đủ cho yêu cầu.
- Best-fit: cấp phát vùng trống nhỏ nhất vừa đủ yêu cầu; phải duyệt toàn danh sách, nếu không sắp theo thứ tự. Sẽ tạo ra vùng nhớ trống dư ra nhỏ nhất.
- Worst-fit: cấp phát vùng trống lớn nhất; phải duyệt toàn danh sách. Sẽ tạo những ô trống dư ra lớn nhất.

First-fit và best-fit tốt hơn worst-fit về mặt tốc độ và việc tận dụng bộ nhớ.

Bộ nhớ ảo

Bộ nhớ ảo là sự trừu tượng hóa của HĐH, nó cung cấp người lập trình một không gian địa chỉ lớn hơn không gian địa chỉ vật lý thật sự

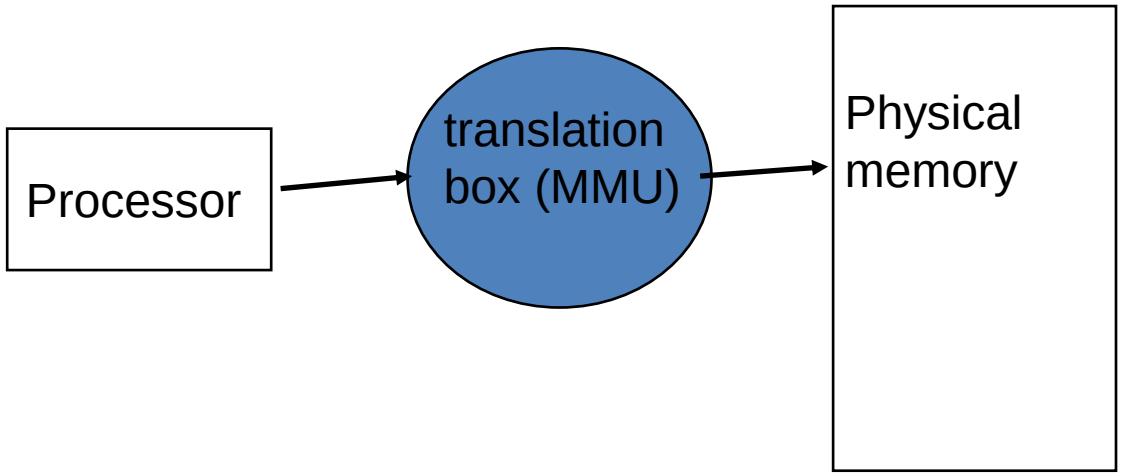
Bộ nhớ ảo có thể được triển khai bằng cách phân trang hoặc phân đoạn, hiện tại phân trang thông dụng hơn

- Mô hình kết hợp cũng thường được dùng, phân đoạn thường khá đơn giản (v.d., một số lượng xác định các đoạn cùng kích thước)

Hữu ích của bộ nhớ ảo:

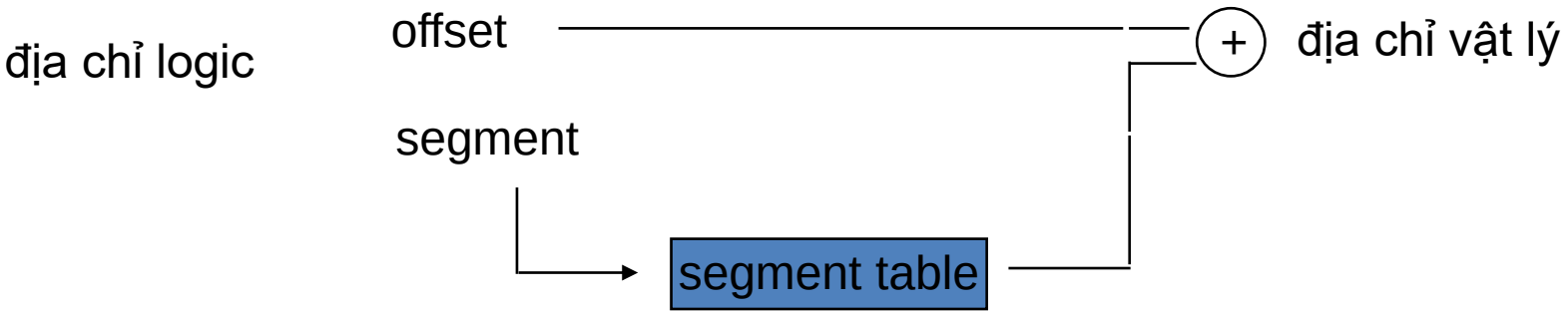
- Lập trình viên không lo lắng với việc các máy tính khác nhau có kích thước bộ nhớ vật lý khác nhau
- Phân mảnh trong môi trường đa chương

Chuyển đổi địa chỉ



MMU: memory management unit

Phân đoạn



Phân đoạn

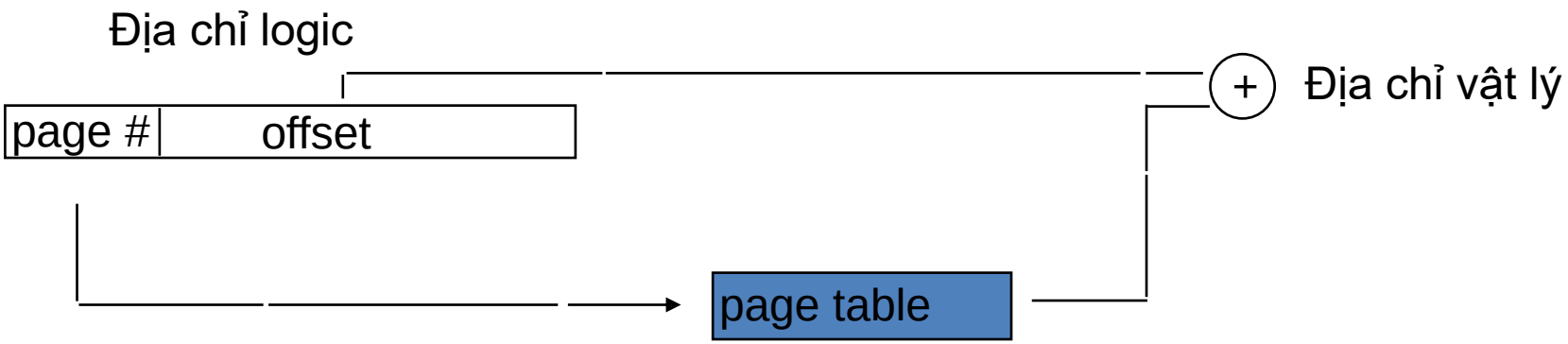
Các đoạn có kích thước khác nhau

Biên dịch địa chỉ dựa vào các thanh ghi (base, size, state) – bảng phân đoạn

Trạng thái (state): valid/invalid, access permission, reference bit, modified bit

Các đoạn có thể trực quan với lập trình viên và để tiện lợi, chia ra hai loại đoạn, dùng cho mã chương trình hay dữ liệu (nghĩa là code segment hoặc là data segments)

Phân trang hardware



Phân trang

- Các trang có kích thước cố định
- Bộ nhớ vật lý tương ứng với trang gọi là page frame
- Chuyển đổi địa chỉ thông qua bảng trang, được đánh chỉ mục bằng page number
- Mỗi mục tin trong bảng trang lưu một con số đại diện page frame mà trang đó ánh xạ tới và trạng thái của trang trong bộ nhớ
- Trạng thái: valid/invalid, access permission, reference bit, modified bit, caching
- Việc phân trang là "trong suốt" với người lập trình

Kết hợp phân trang và phân đoạn

Một vài MMU kết hợp phân trang và phân đoạn

Chuyển đổi địa chỉ phân đoạn trước

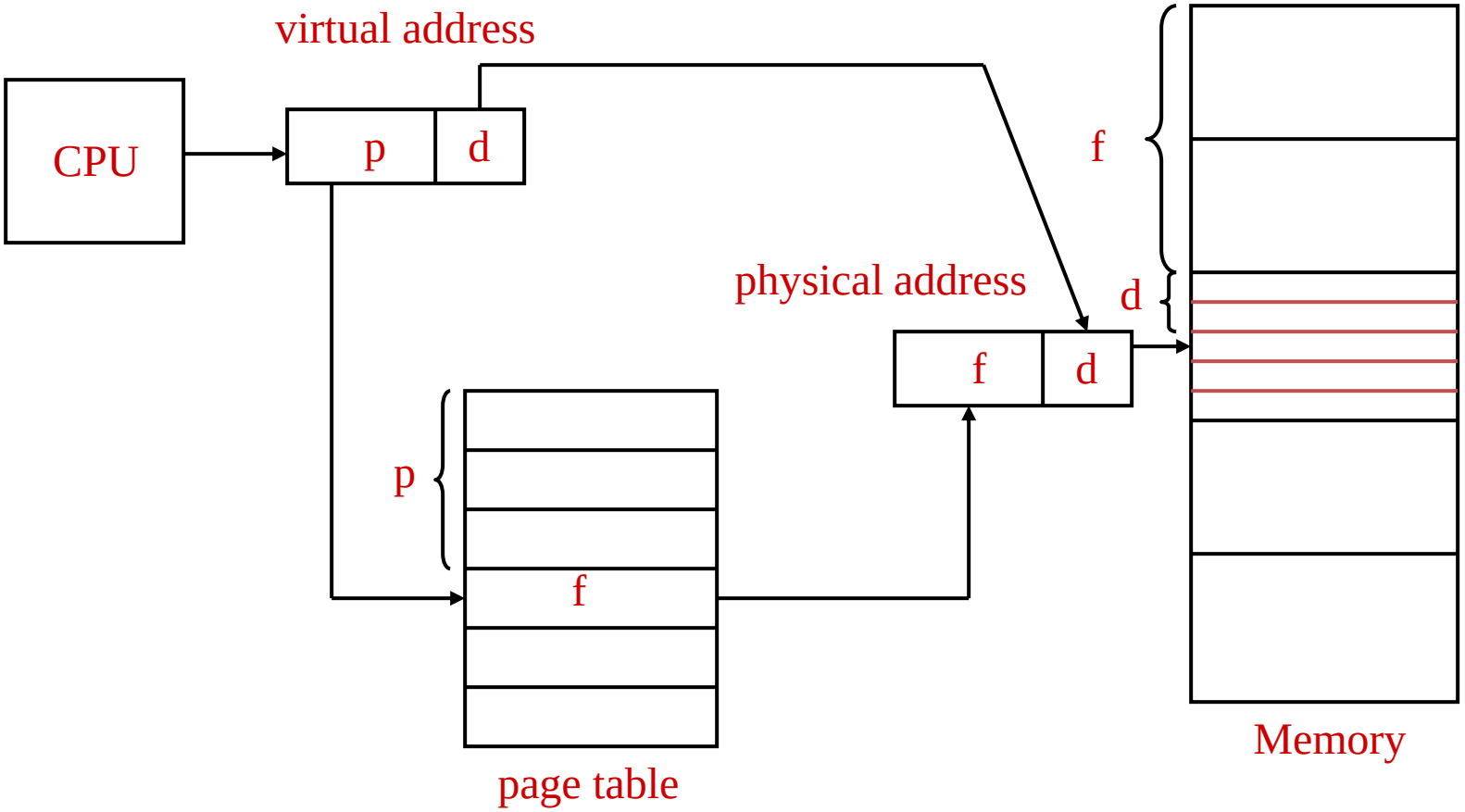
Địa chỉ đoạn lưu địa chỉ bảng trang cho đoạn đó

Bảng trang được đánh chỉ mục bằng phần page number trong địa chỉ ảo và ánh xạ tới page frame tương ứng

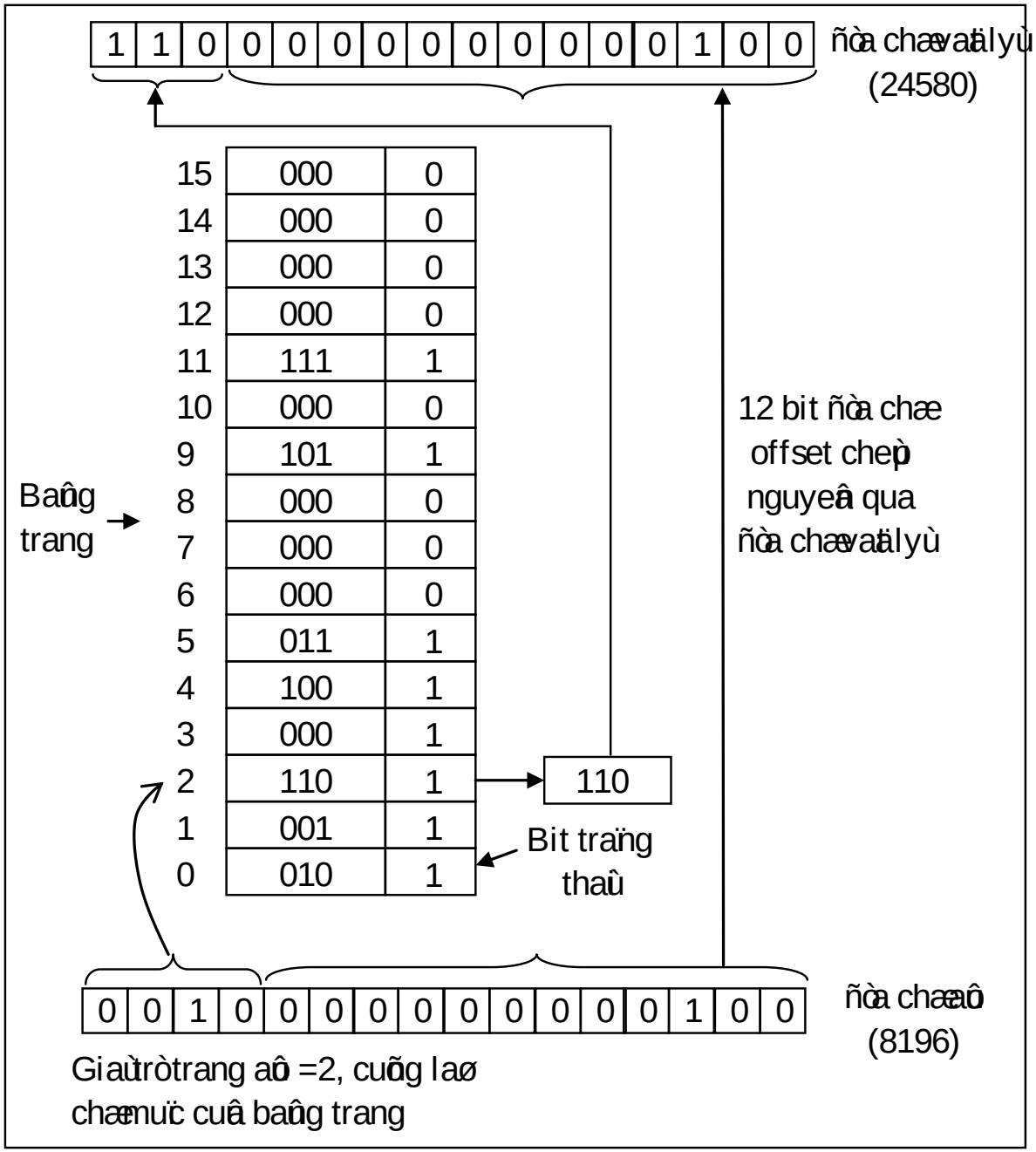
Ngày nay người ta không còn dùng phân đoạn nhiều nữa

- UNIX sử dụng mô hình phân đoạn đơn giản nhưng không yêu cầu hỗ trợ của phần cứng

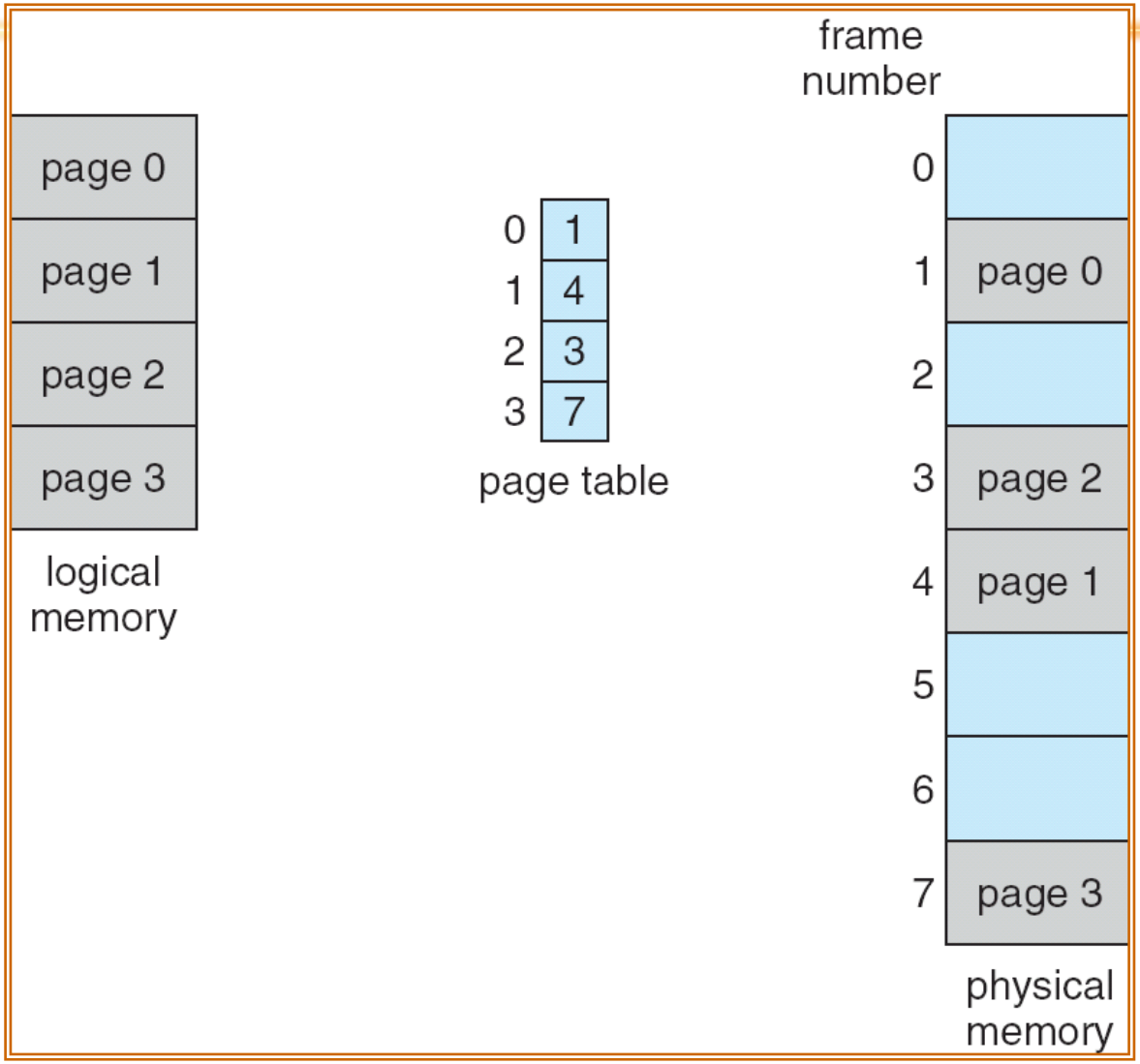
Chuyển đổi địa chỉ trong phân trang



Bảng trang 16-bit,
Mỗi trang kích thước 4KB



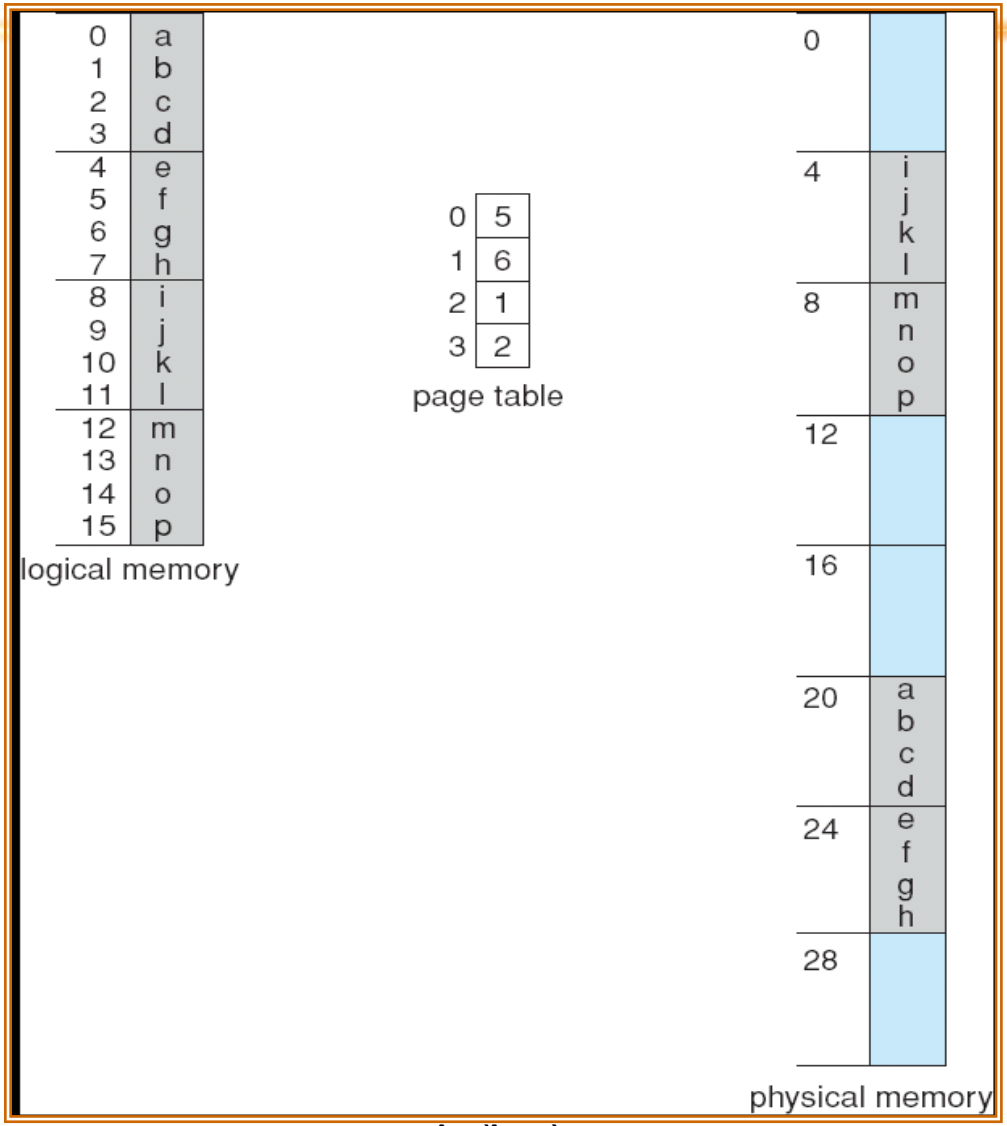
Mô hình phân trang



Ví dụ phân trang

Bộ nhớ 32-byte

và mỗi trang 4 byte



Translation Lookaside Buffers (TLB)

Mỗi truy cập bộ nhớ ⇒ tìm trang vật lý tương ứng của trang ảo ⇒ phải rất nhanh

Caching, lựa chọn đầu tiên…

Chúng ta vẫn phải tìm kiếm địa chỉ vật lý của trang từ các mẫu tin trên bảng trang?

Tương tự như các cache bộ nhớ thông thường

Translation Lookaside Buffer

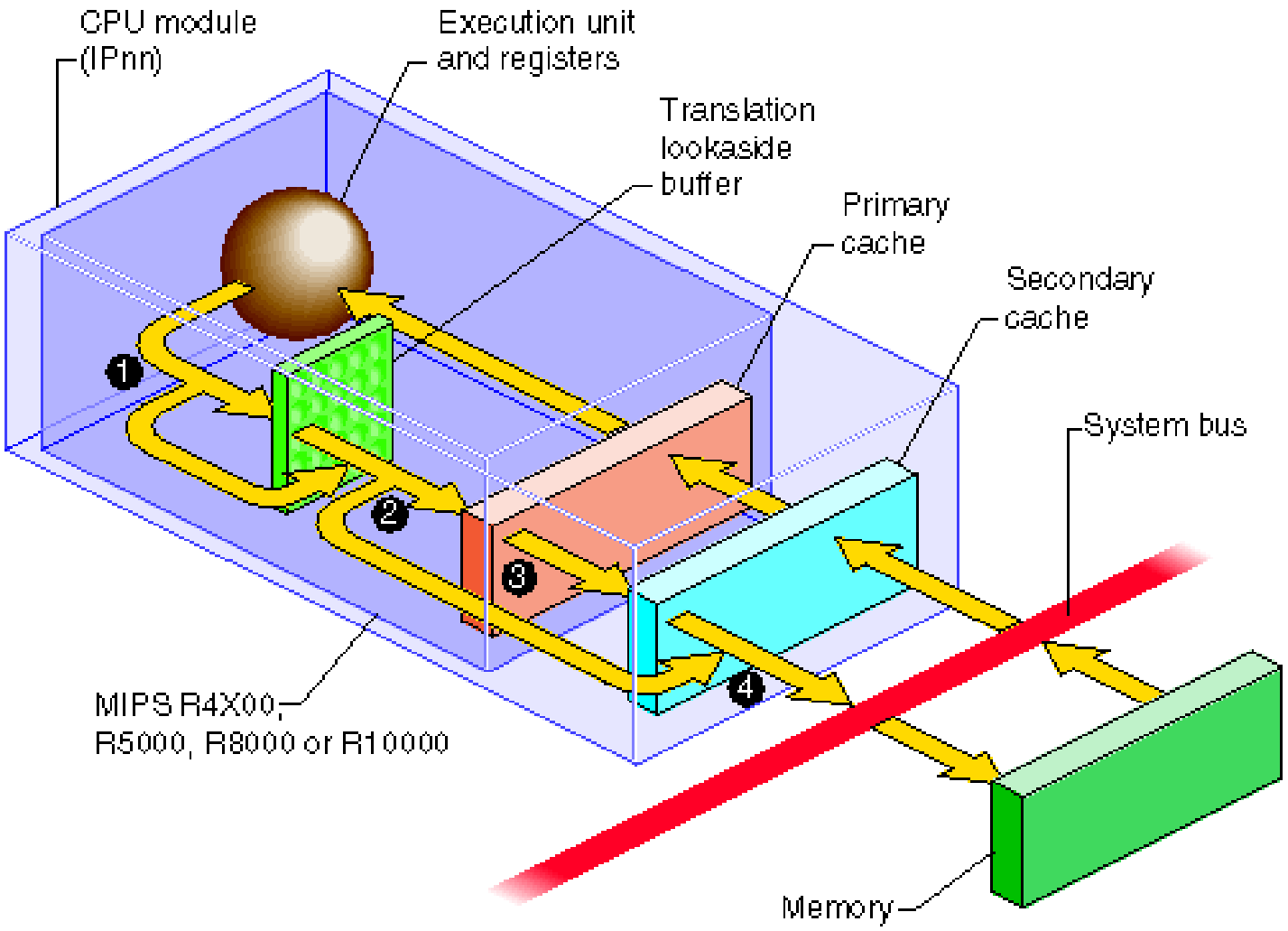
Cache cho các mẫu tin trong bảng trang gọi là Translation Lookaside Buffer (TLB)

Thường là 64 mẫu tin

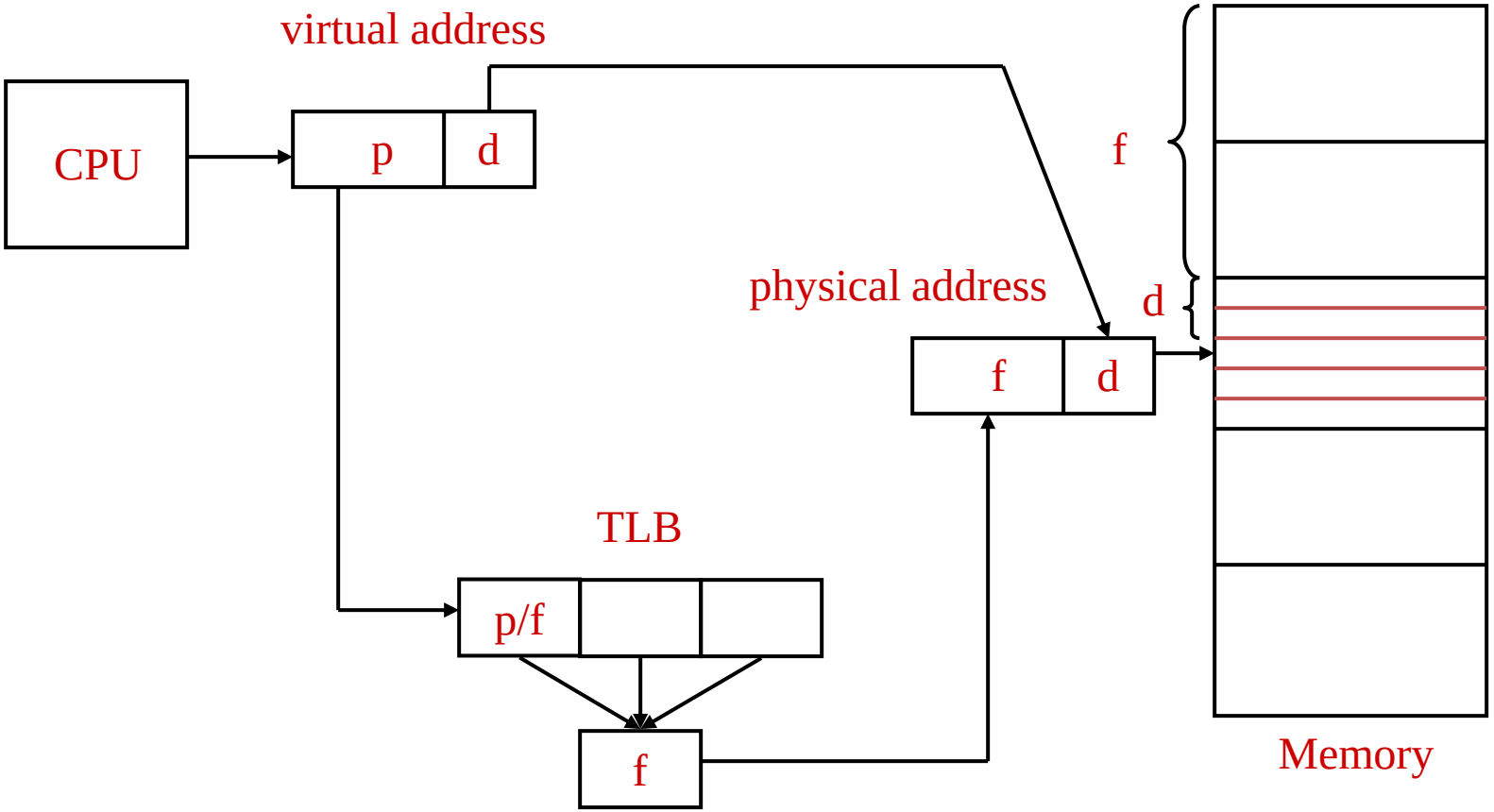
Mỗi mẫu tin của TLB chứa 1 page number và một mẫu tin của bảng trang tương ứng

Mỗi lần truy cập bộ nhớ, chúng ta tìm page number $\Rightarrow$ frame được ánh xạ trong TLB bởi trang này

Translation Lookaside Buffer



Chuyển đổi địa chỉ dùng TLB



TLB Miss

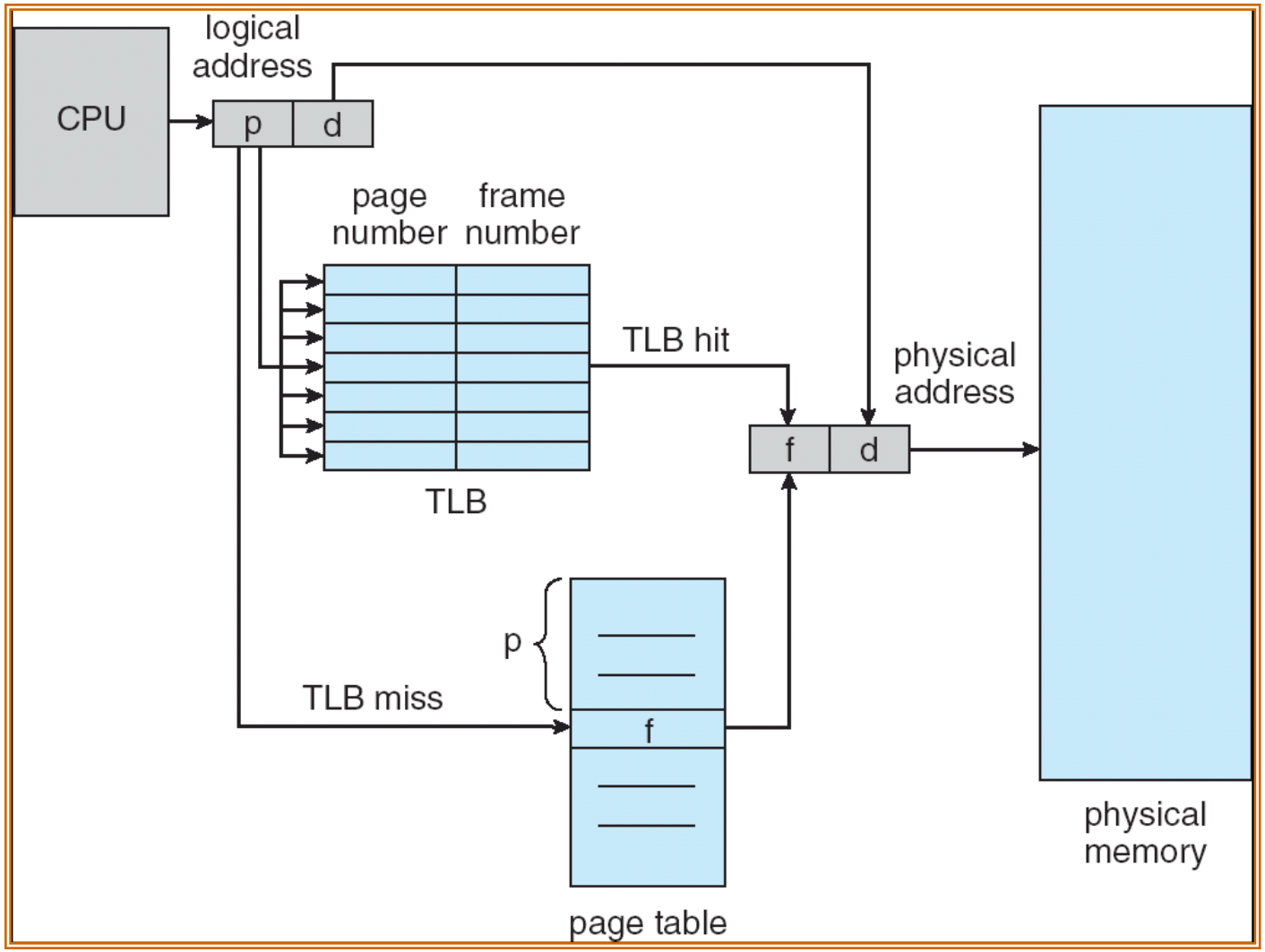
Điều gì xảy ra nếu TLB không chứa thông tin bảng trang truy cập?

- TLB miss
- Thu hồi một mẫu tin trên TLB nếu không có mẫu tin đang rãnh
  - Chính sách thay thế?
- Chép vào TLB mẫu tin thiếu từ bảng trang

TLB misses có thể được xử lý bằng phần cứng hoặc phần mềm

- Phần mềm cho phép ứng dụng hỗ trợ quyết định thay thế

Chuyển đổi địa chỉ dùng TLB



Lưu không gian địa chỉ ở đâu?

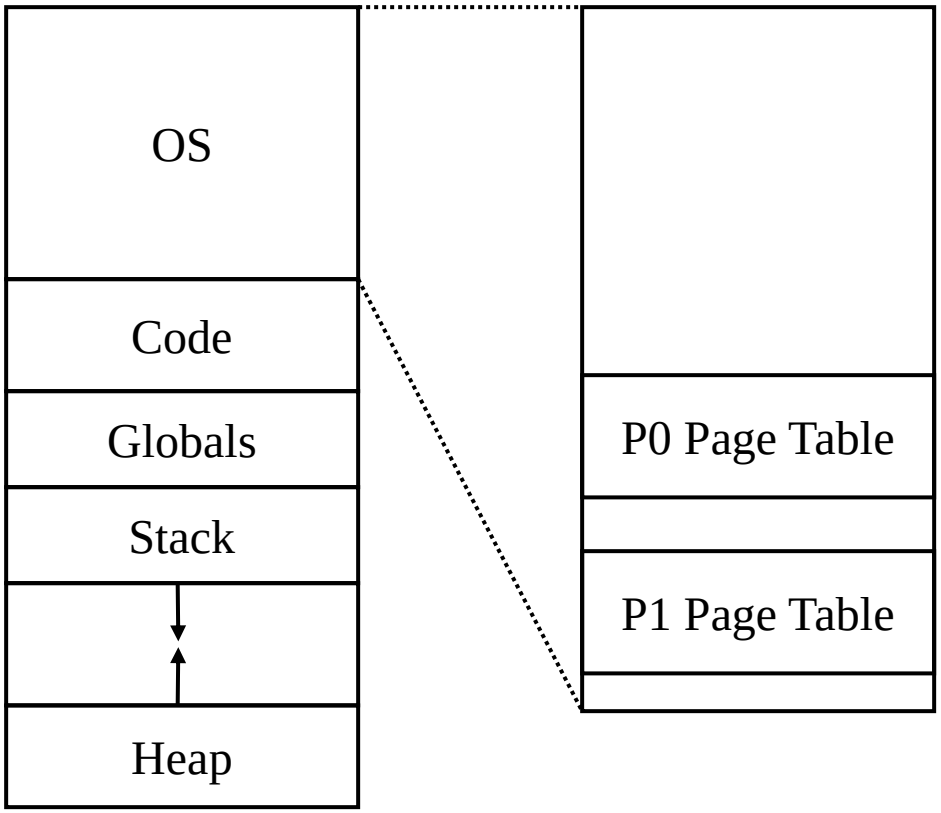
Không gian địa chỉ có thể lớn hơn bộ nhớ vật lý

Chúng ta lưu nó ở đâu?

Chúng ta lưu bảng trang ở đâu?

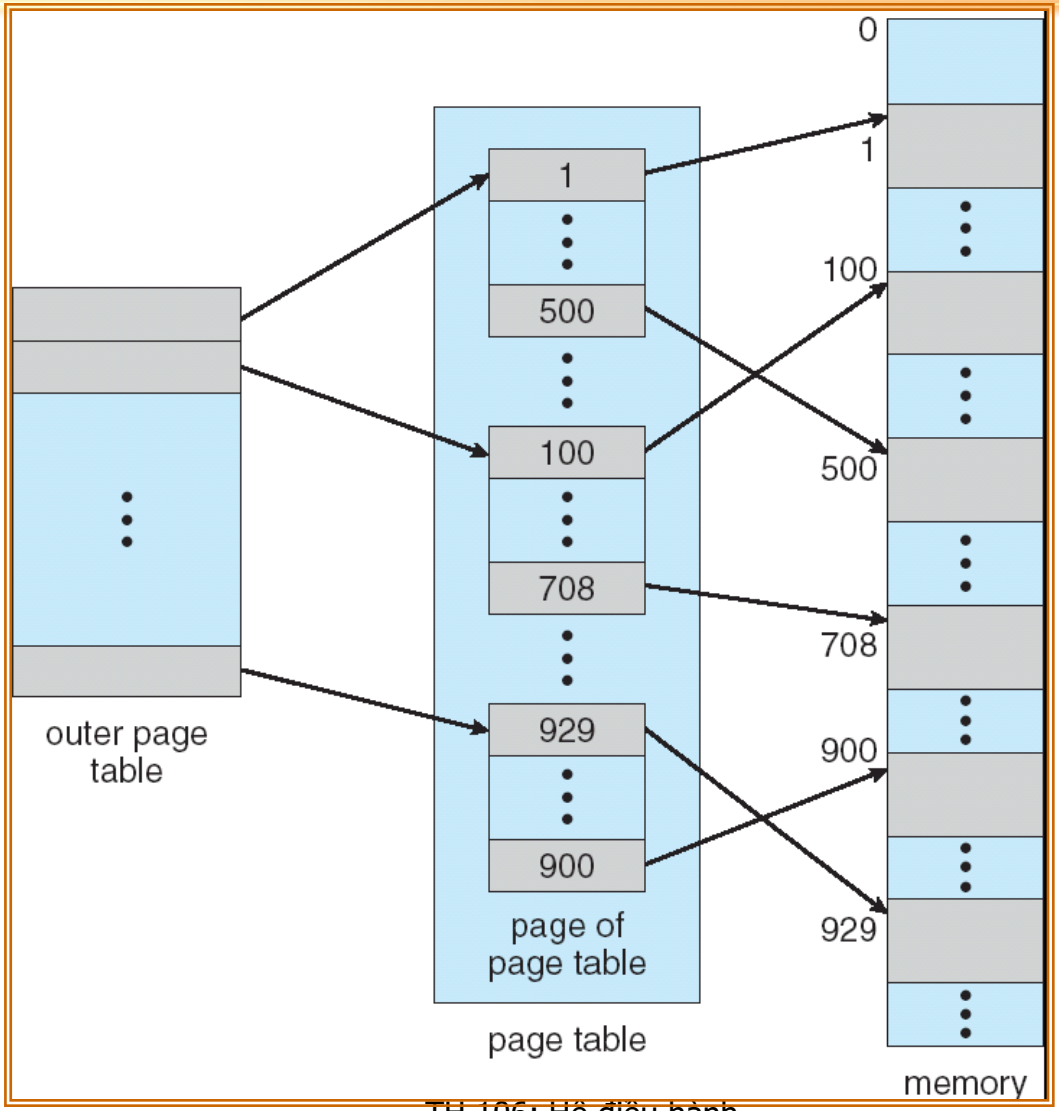
Lưu bảng trang ở đâu?

Trong bộ nhớ …



- Điều thú vị là sử dụng bộ nhớ để mở rộng địa chỉ bộ nhớ, *kích thước bộ nhớ vật lý giảm*
- Trade-off như thế nào!
- Cần hiểu đặc tính của các ứng dụng
- Bảng trang có thể rất lớn! Giải pháp?

Bảng trang 2 cấp



Ví dụ bảng trang 2 cấp

Một địa chỉ logic (máy 32-bit kích thước trang 4K) được chia thành:

Page number: 20 bits.

Page offset: 12 bits.

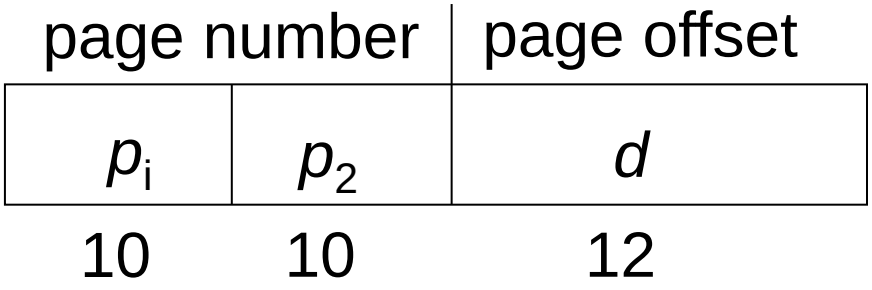
Vì bảng trang lại được phân trang, page number lại được chia thành:

10-bit: page number.

10-bit: page offset.

Ví dụ bảng trang 2 cấp

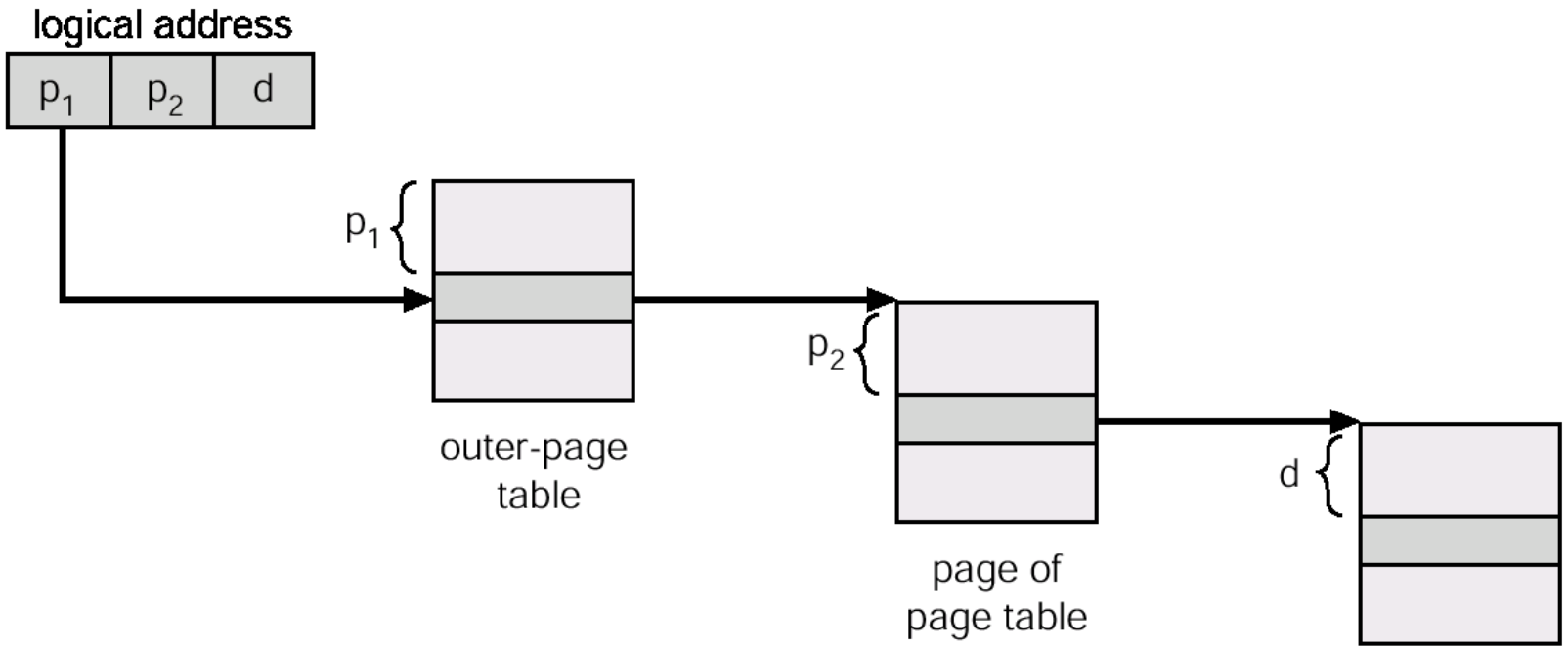
Vậy, địa chỉ logic như sau:



Với pi là chỉ mục trong outer page table, và p2 là chỉ mục của một trang thuộc outer page table.

Lược đồ chuyển đổi địa chỉ

Lược đồ chuyển đổi địa chỉ của kiến trúc phân trang 2 cấp 32-bit



Hiệu quả thực thi hân trang đa cấp

Vì mỗi cấp được lưu như một bảng phân biệt trong bộ nhớ, chuyển đổi địa chỉ logic thành địa chỉ vật lý tốn tới 4 lần truy cập bộ nhớ.

Caching cho phép các tính toán này khả thi.

Cache đạt 98% hit thì:

effective access time = 0.98 x 120 + 0.02 x 520

= 128 nanoseconds.

Bị chậm lại chỉ có 28% trong việc truy cập bộ nhớ.

128-100 = 28 nanoseconds. Giả sử 1 truy cập bộ nhớ là 100 ns, thời gian tìm trên TLB là 20 ns, nên nếu 1 hit trên LTB => tốn 120 ns. Còn nếu miss thì 4*100 + 20(search tren TLB) + 100 truy cập frame = 520ns

Bảng trang nghịch đảo

- Mục tiêu của bảng trang là để tìm ra trang vật lý tương ứng của từng trang ảo
- Tuy nhiên, số lượng trang ảo rất lớn → kích thước bảng trang có thể chiếm một không gian lớn trên bộ nhớ
- Ví dụ hệ thống 64-bit địa chỉ, kích thước mỗi trang là 4KB, vậy bảng trang cần 2^{52} mẫu tin. Nếu mỗi mẫu tin 8 bytes thì bảng trang chiếm 30 triệu GB.

Bảng trang nghịch đảo

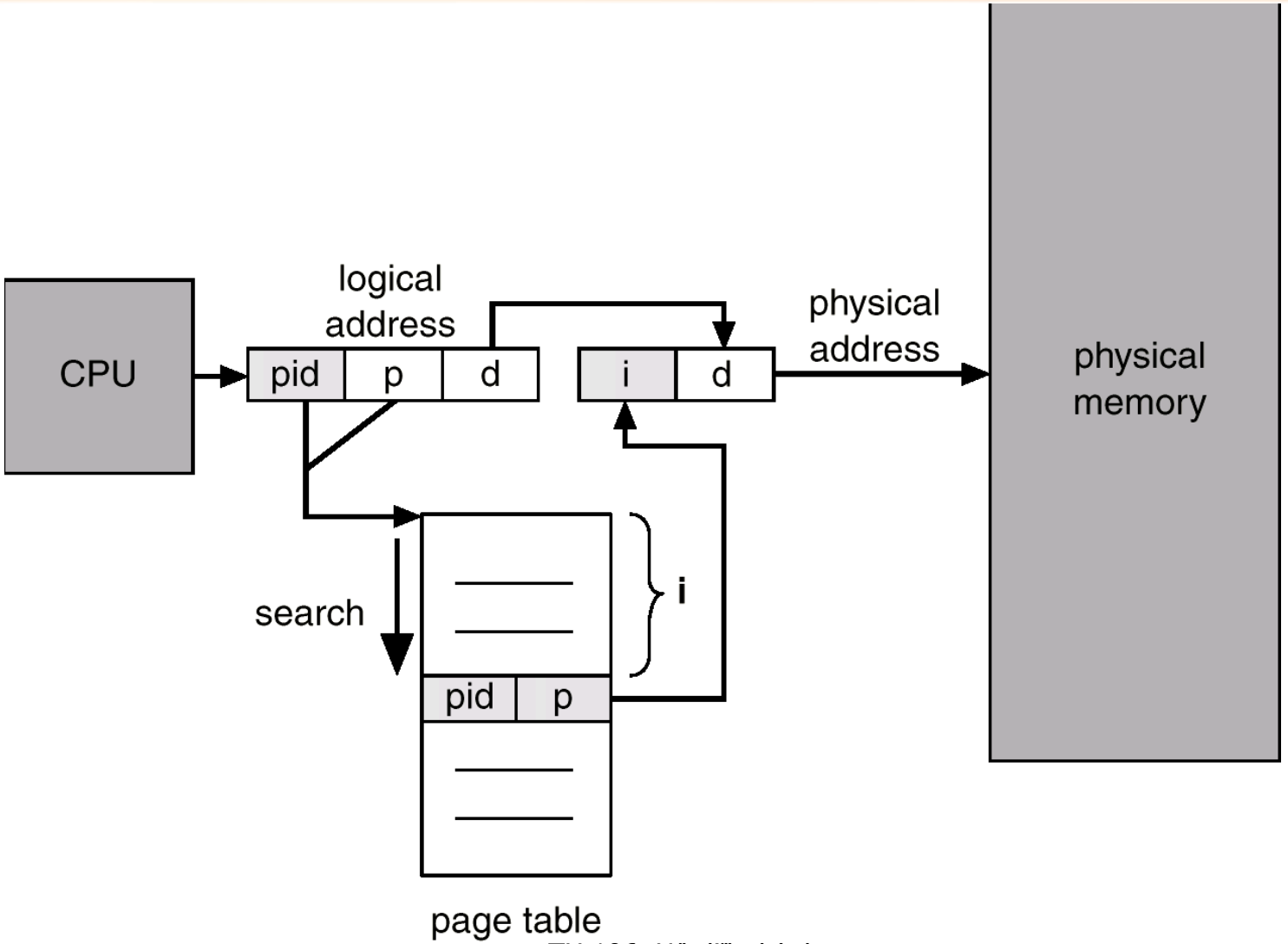
Mỗi mẫu tin dành cho 1 trang thật (frame) trên bộ nhớ.

Mỗi mẫu tin gồm địa chỉ ảo của trang, cùng với thông tin về tiến trình đang dùng trang này. <pid, p>

Giảm bộ nhớ cần thiết để lưu mỗi trang, nhưng tăng thời gian để tìm bảng trang.

Sử dụng CTDL bảng băm (hash table) để tăng tốc độ tìm kiếm.

Kiến trúc bảng trang nghịch đảo



Giải quyết vấn đề địa chỉ ảo > địa chỉ VLý ntn?

Nếu không gian địa chỉ ảo của một tiến trình nhỏ hơn bộ nhớ vật lý thì không có vấn đề gì

- Chỉ lo giải quyết vấn đề phân mảnh

Khi bộ nhớ ảo của một tiến trình lớn hơn bộ nhớ vật lý

- Một phần lưu trên bộ nhớ
- Một phần lưu trên đĩa

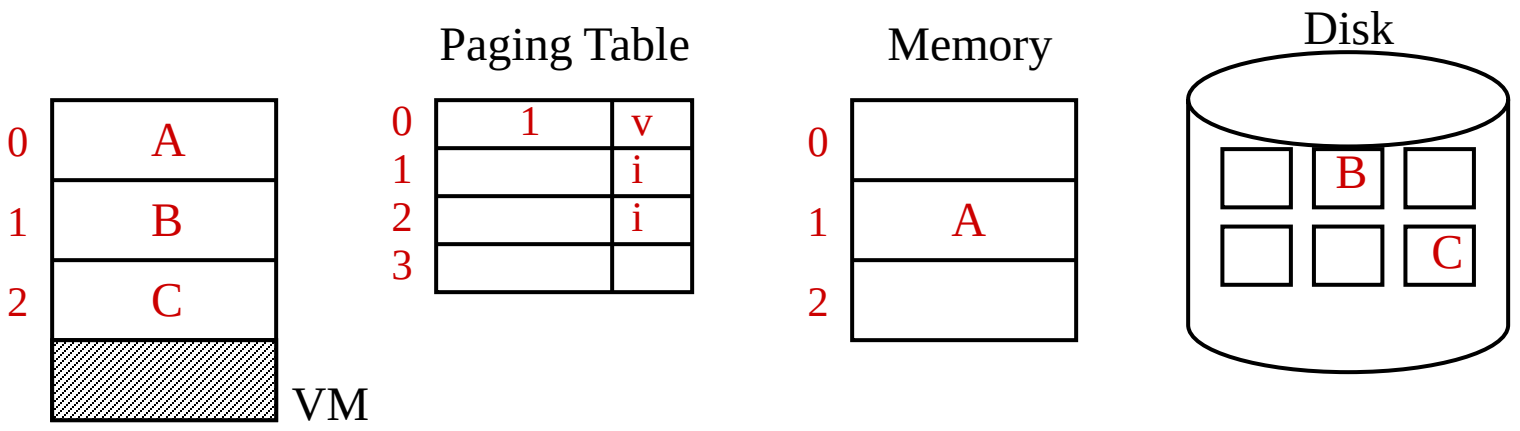
Giải quyết ntn?

Nạp trang theo yêu cầu

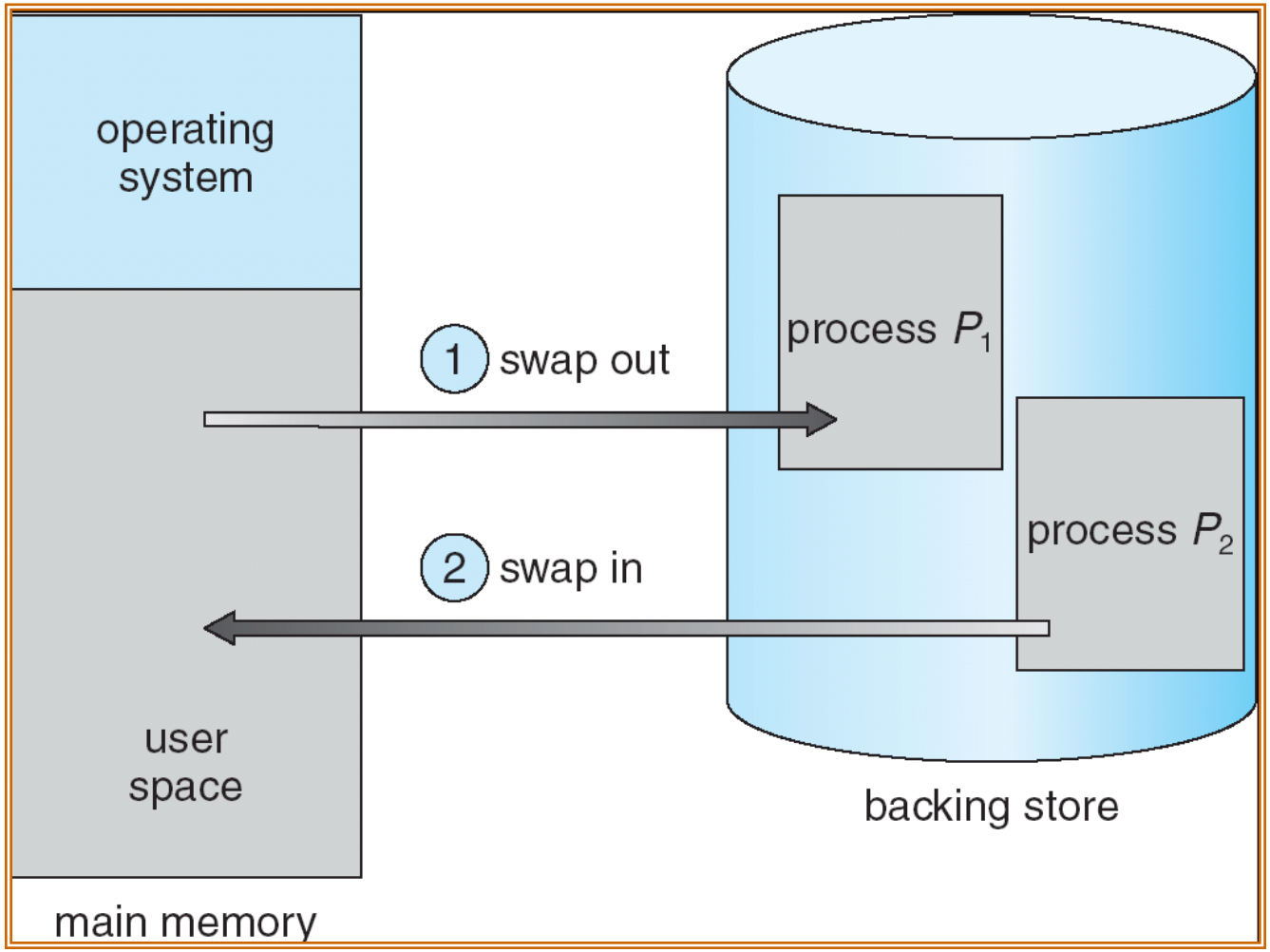
Để bắt đầu một tiến trình (chương trình), chỉ nạp trang chứa đoạn mã cho tiến trình bắt đầu thực thi

Khi tiến trình có yêu cầu tới vùng nhớ (chứa lệnh hay dữ liệu) nằm ngoài trang đã nạp, thì nạp trang đó lên

Làm sao biểu diễn một trang của máy ảo mà nó chưa nạp lên bộ nhớ?



Swapping


operating system
1 swap out
process P_1
process P_2
2 swap in
user space
backing store
main memory

Lỗi trang

Điều gì xảy ra khi tiến trình yêu cầu một trang bị đánh dấu lỗi?

- Trap *lỗi trang*
- Kiểm tra có phải truy xuất hợp lệ (có trang vật lý đúng)
- Tìm 1 frame bộ nhớ trống
- Đọc trang cần thiết từ bộ nhớ phụ (ổ đĩa)
- Đổi valid bit của trang thành v (hợp lệ)
- Bắt đầu lại lệnh bị ngắt bởi trap

Nếu không có frame trống thì sao?

Lỗi trang (tt)

Các tình huống khi truy cập bộ nhớ?

- Nếu TLB miss $\Rightarrow$ đọc mẫu tin trong bảng trang
- Và nếu, lỗi trang (=>thay trang)
- Và nếu, tất cả các frames đang dùng $\Rightarrow$ cần thu hồi một trang $\Rightarrow$ thay đổi giá trị trong bảng trang của tiến trình
- Đọc trang cần thiết, cập nhật mẫu tin của bảng trang, cập nhật TLB

Chi phí xử lý lỗi trang

Trap, kiểm tra bảng trang, tìm frame trống (hoặc tìm trang thay thế) … khoảng 200 - 600 μs

Tìm và đọc trên đĩa … khoảng 10 ms

Truy cập bộ nhớ … khoảng 100 ns

Lỗi trang làm chậm thực thi khoảng ~100,000 lần!!!!!

Đó là chưa kể phát sinh có thể xảy ra trong các bước trên

Tốt nhất là không để xảy ra nhiều lỗi trang!

Nếu muốn sự ảnh hưởng ít hơn 10%, chỉ cho phép 1 lỗi trang trong 1,000,000 lần truy cập bộ nhớ

Thay trang

Nếu không còn frame trống khi bị lỗi trang?

- Lấy lại một frame đang được sử dụng
  - Chọn frame để thay thế (nạn nhân)
  - Lưu trang nạn nhân vào ổ đĩa
  - Cập nhật lại bảng trang (trang nạn nhân thành invalid)
  - Đọc trang cần thiết vào frame vừa chọn
  - Cập nhật lại bảng trang (trang vừa thay thế là valid)
  - Bắt đầu lại lệnh đã gây ra lỗi trang

Tối ưu hóa: không phải ghi trang nạn nhân trở lại nếu như nó vẫn chưa bị thay đổi (cần thêm dirty bit cho mỗi trang).

Thay trang

Không dễ dàng để tìm được chính sách thay thế trang tốt

- Khi thu hồi một trang, làm sao chúng ta biết là trang tốt nhất có thể giảm thiểu lỗi trang sau này?

Có tồn tại thuật toán thay thế trang tối ưu?

Nếu có, thuật toán thay thế trang tối ưu là gì?

Xem ví dụ sau:

- Giả sử chúng ta có 3 frames và chạy chương trình theo mẫu sau
- 7, 0, 1, 2, 0, 3, 0, 4, 2, 3

Giả sử chúng ta biết thứ tự yêu cầu trang

Thay trang

Giả sử chúng ta biết thứ tự yêu cầu trang như sau

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

Thuật toán tối ưu là **thay thế trang sẽ không dùng lại lâu nhất**

Vấn đề của thuật toán này là gì?

Giải pháp thực tế là dự đoán tương lai(sẽ yêu cầu trang nào) bằng quá khứ

Có thể đúng vì tính cục bộ

FIFO

First-in, First-out

- Công bằng, thời gian mỗi trang trên bộ nhớ gần như tương đương nhau

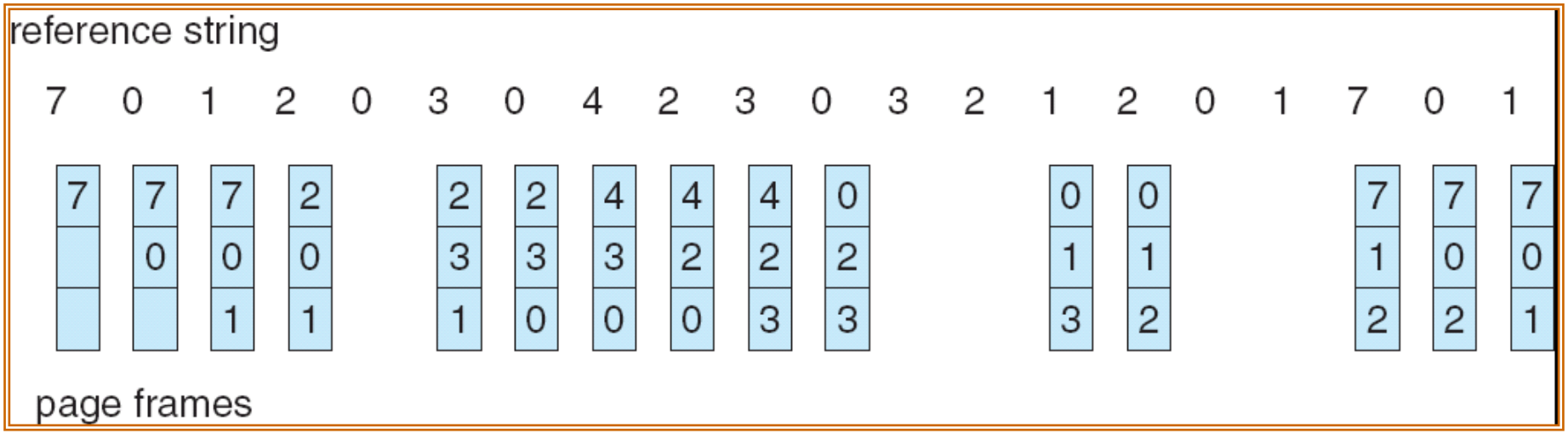
Có vấn đề gì không?

- Có phù hợp với yêu cầu của một chương trình?

Có hiệu quả với ví dụ của chúng ta?

7, 0, 1, 2, 0, 3, 0, 4, 2, 3

Thay thế trang FIFO



Ví dụ khác FIFO

Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frames (3 trang có thể đồng thờitrong bộ nhớ tại mỗi thời điểm)

1	1	4	5	
2	2	1	3	9 page faults
3	3	2	4	

4 frames

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

Belady's Anomaly: more frames $\Rightarrow$ more page faults

LRU

Least Recently Used (ít sử dụng gần đây nhất)

- Mỗi lần truy cập trang, dán nhãn thời gian lại
- Khi cần thu hồi một trang, chọn trang với nhãn thời gian lâu nhất

LRU có phải tối ưu nhất?

- Trong thực tế, LRU là giải pháp tốt cho hầu hết chương trình

Có dễ dàng cài đặt?

Thay thế trang ít thường sử dụng nhất

Dùng 1 reference bit và bộ đếm cho mỗi trang (frame)

Mỗi ngắt đồng hồ, HĐH cộng reference bit vào biến counter rồi xóa reference bit (bật reference bit nếu trang được sử dụng)

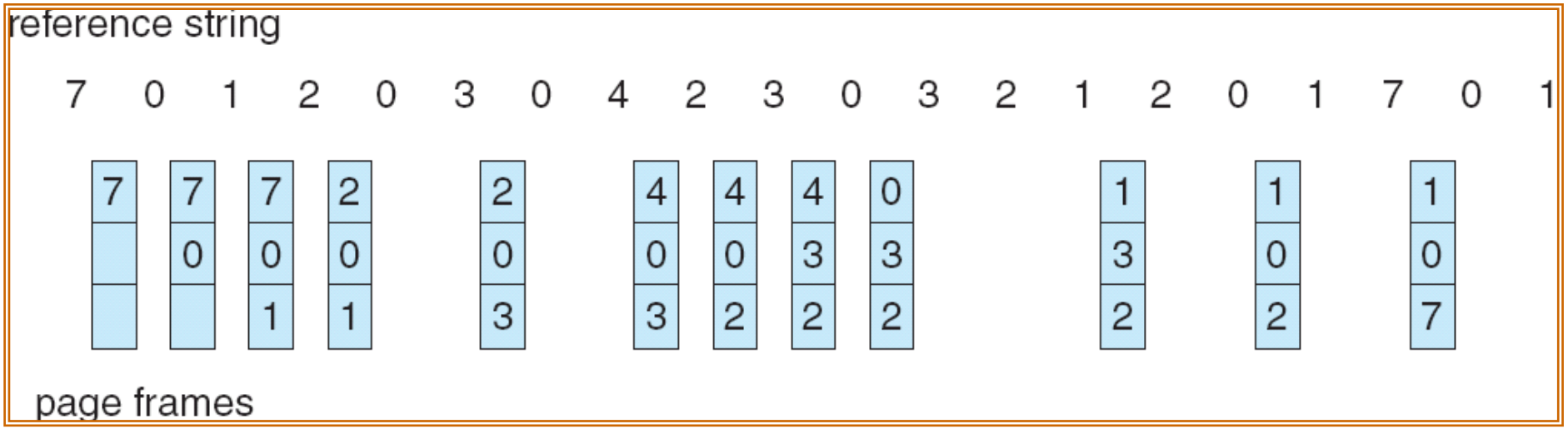
Khi cần thay trang, chọn trang có số đếm ít nhất

Có vấn đề gì không?

- Lưu vết tất cả, khó thu hồi trang đã dùng rất nhiều trong quá khứ, nhưng hiện tại không còn dùng nữa
- Chi phí cao để quản lý counter, bởi vì bộ nhớ ngày càng lớn

Có thể cải tiến bằng lược đồ độ tuổi: counter được shift qua phải trước khi cộng reference bit và reference bit được cộng vào bit trái nhất

Ví dụ LRU



Clock (Cơ hội thứ hai)

Sắp xếp các trang thành vòng tròn, và dùng 1 đồng hồ

Dùng 1 use bit cho mỗi frame. Bật use bit lên khi mà frame đó được dùng.

- Nếu use bit = 0, trang không sử dụng

Khi lỗi trang:

- Di chuyển kim đồng hồ
- Kiểm tra use bit
  - If 1, mới sử dụng, xóa và tiếp tục
  - If 0, chọn trang này để thay thế

Liệu chúng ta có thể luôn tìm được trang để thay thế?

Cơ hội thứ Nth

Tương tự ý tưởng trên nhưng,

Dùng 1 counter và 1 ***use bit***

Khi lỗi trang:

- Dịch kim đồng hồ
- Kiểm tra ***use bit***
  - If 1, xóa use bit và đặt counter = 0
  - If 0, tăng counter, if counter < N, tiếp tục, ngược lại chọn trang này để thay thế

Nhận xét

- N lớn $\Rightarrow$ gần giống kết quả với LRU

Nếu N quá lớn thì gặp vấn đề gì?

Tiếp cận khác của cơ hội thứ 2nd

Luôn giữ một danh sách n > 0 trang trống

- Khi lỗi trang, nếu danh sách trống có hơn n frames, chọn 1 frame từ danh sách trống
- Nếu danh sách trống chỉ có n frames, chọn 1 frame từ danh sách, rồi chọn một frame đang sử dụng để đặt vào danh sách trống

Khi lỗi trang, nếu trang lỗi là trong danh sách trống, không phải đọc lại trang đó lên bộ nhớ.

Triển khai trên VAX … hiệu quả gần đạt LRU

Môi trường đa chương

Vì sao?

- Tận dụng tốt hơn resource (CPU, disks, memory, etc.)

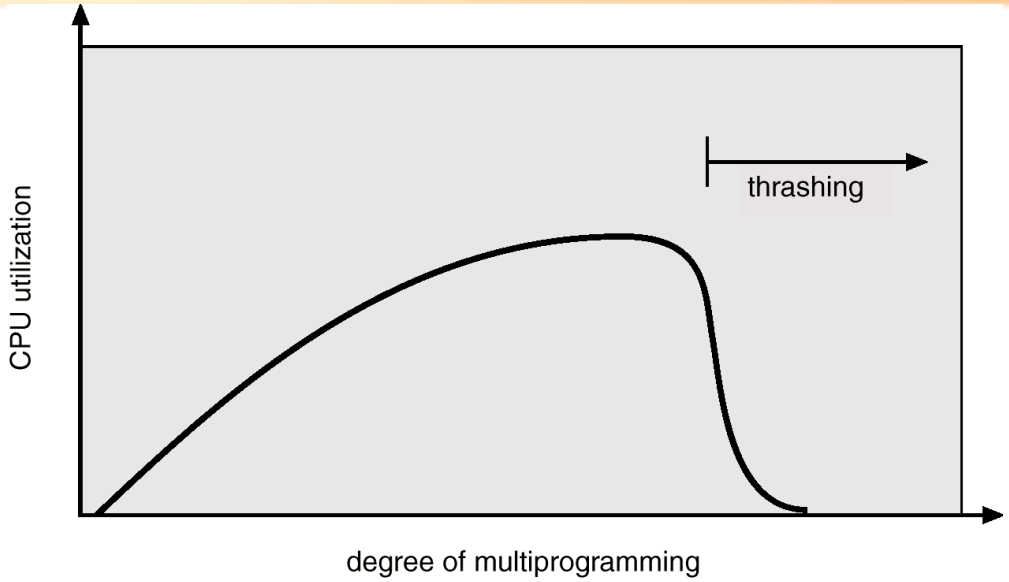
Bài toán?

- Caching – TLB?
- Sự công bằng?
- Bộ nhớ giới hạn

Các vấn đề có thể phát sinh?

- Mỗi tiến trình cần một tập các trang(*working set*) để thực thi
- Nếu quá nhiều tiến trình thực thi, có thể trì trệ vì thay trang (thrash)

Biểu đồ trì trệ hệ thống (Thrashing)



Trì trệ (thrashing): hệ thống chỉ bận rộn cho việc thay trang

Khi nào xảy ra sự trì trệ?
Σ kích thước working sets > tổng kích thước bộ nhớ

Hỗ trợ đa tiến trình

Nhiều không gian địa chỉ tiến trình có thể được nạp lên bộ nhớ

Thanh ghi trỏ tới bảng trang hiện tại

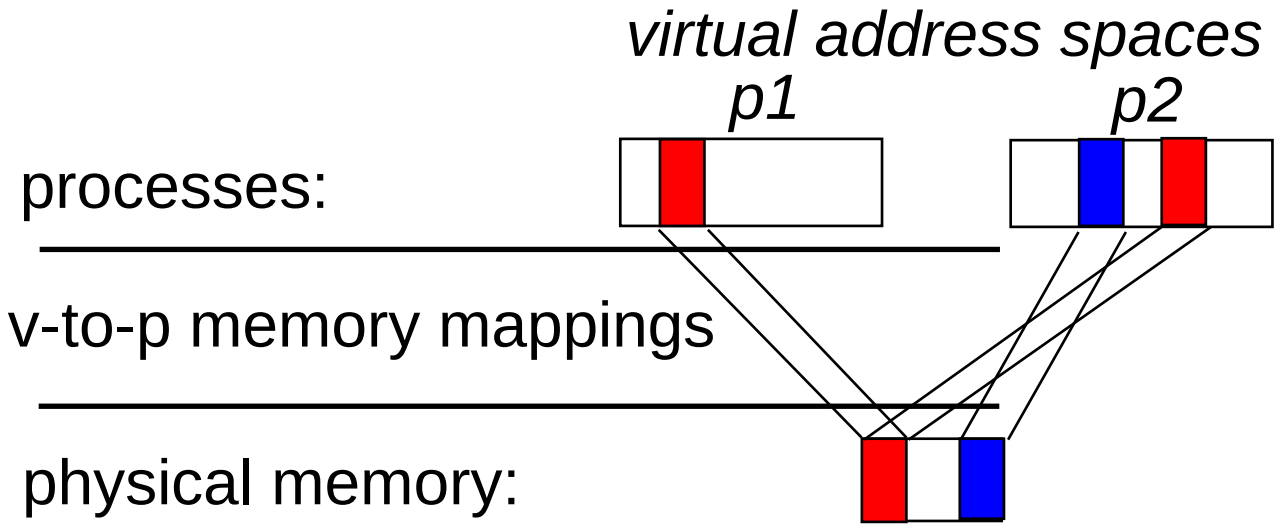
HĐH phải cập nhật lại thanh ghi khi context switching giữa các tiểu trình khác tiến trình

Đa số TLBs có thể cache nhiều bảng trang

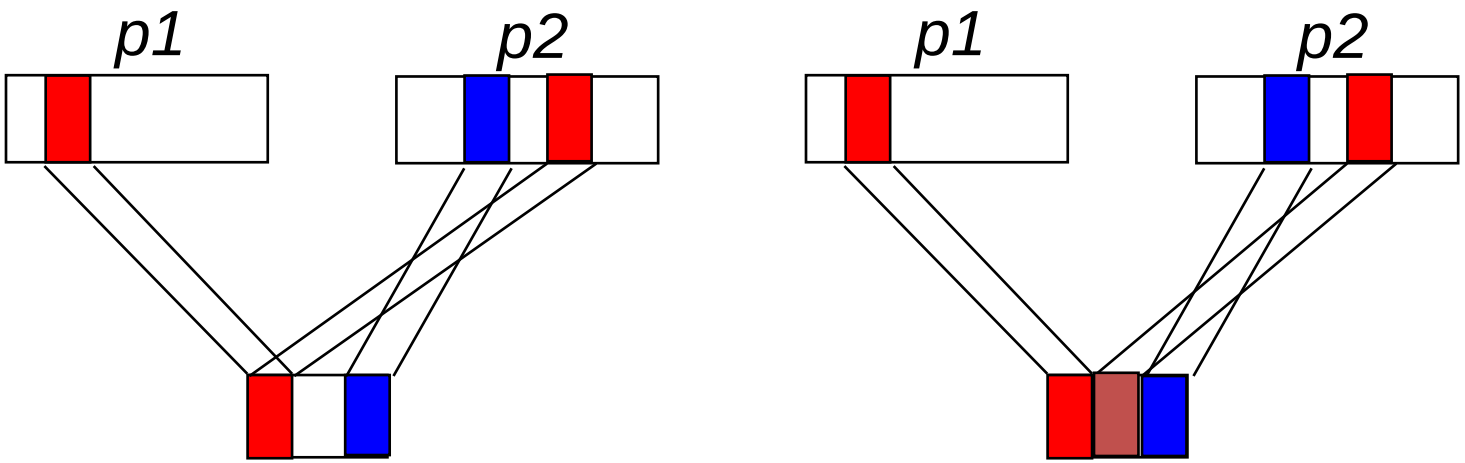
- Lưu thêm process id để phân biệt địa chỉ logic thuộc các tiến trình khác nhau

Nếu TLB chỉ caches 1 bảng trang thì nó phải xóa hết bảng trang khi context switch.

Chia sẻ


virtual address spaces
p1
p2
processes:
v-to-p memory mappings
physical memory:

Copy-on-Write



Tập trang thường trú (working set)

Một tiến trình nên nạp bao nhiêu trang lên bộ nhớ ?

Số lượng trang thường trú này có thể cố định hoặc thay đổi

Miền thay thế là cục bộ hay toàn cục

Lược đồ hay được sử dụng:

- Thay trang toàn cục: đơn giản – kích thước tập trang thường trú của tiến trình thay đổi mỗi lần thay trang
- Thay trang cục bộ: phức tạp hơn – kích thước tập trang thường trú phải thay đổi xung quanh giá trị kích thước tập trang thường trú của tiến trình (working set size)

Working Set

Là một tập các trang được sử dụng trong khoảng thời gian gần đây nhất

Kích thước của working set có thể thay đổi trong suốt quá trình thực thi của tiến trình

Nếu số lượng trang được cấp nhiều hơn working set thì số lỗi trang sẽ nhỏ

Chỉ điều phối cho tiến trình khi mà bộ nhớ đủ để nạp working set của nó

Làm sao để xác định/(xấp xỉ) kích thước của working set?

Working-Set

$\Delta \equiv$ working-set window $\equiv$ số trang được gọi
Ví dụ: Δ = 10,000 lệnh

WSS_i (working set của tiến trình P_i) =
tổng số trang được gọi trong khoảng tgian Δ vừa rồi(có thể thay đổi)

if Δ quá nhỏ thì không đủ chứa tập trang thường trú.

if Δ quá lớn thì có thể chứa nhiều tập trang thường trú.

if $\Delta = \infty \Rightarrow$ sẽ chứa tập trang toàn chương trình.

$D = \Sigma\ WSS_i \equiv$ tổng trang được yêu cầu

if $D > m \Rightarrow$ Trì trệ (Thrashing)

Chính sách if $D >$ m, thì dừng một tiến trình.

Lưu vết Working Set

Xấp xỉ khoảng thời gian + dùng một reference bit

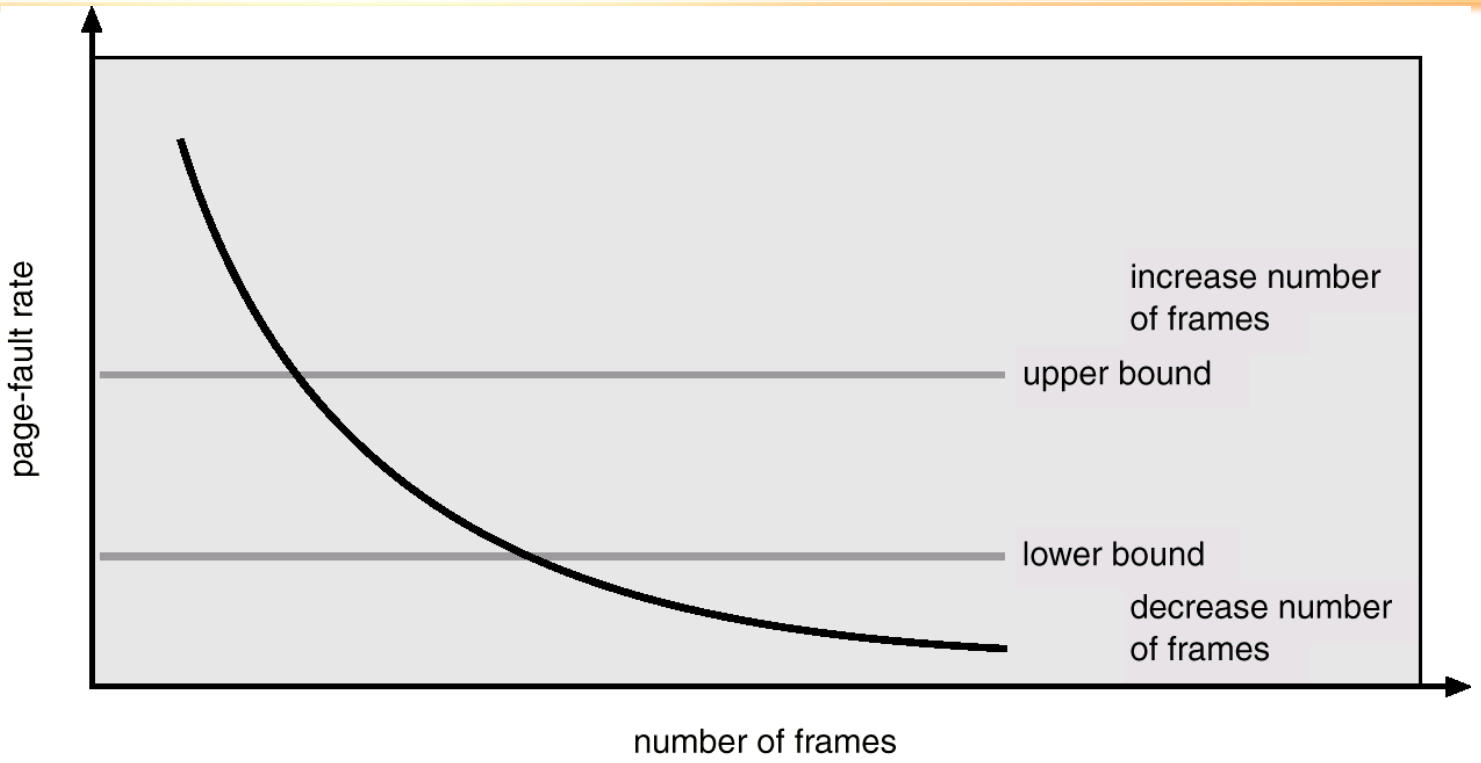
Ví dụ: Δ = 10,000

- Đồng hồ ngắt sau mỗi 5000 đơn vị.
- Dùng 2 reference bít cho mỗi trang.
- Mỗi lần đồng hồ ngắt, thì lưu lại và gán lại giá trị 0 cho cả 2 reference bit.
- Nếu 1 bit = 1 $\Rightarrow$ trang trong working set.

Tại sao không thật sự chính xác?

Cải tiến = dùng 10 bits và ngắt mỗi 1000 đơn vị.

Biểu đồ tầng suất lỗi trang



Xác định tầng suất lỗi trang chấp nhận được.

Nếu tỉ lệ lỗi trang nhỏ, tiến trình bỏ bớt frame.

Nếu tỉ lệ lỗi trang cao, tiến trình cấp thêm frame.

Tần suất lỗi trang

Một bộ đếm cho mỗi trang để đếm "thời gian" giữa các lỗi trang("thời gian" = có thể là số lần trang được truy cập)

Định nghĩa một ngưỡng trên cho biến "thời gian"

Nếu thời gian giữa 2 lỗi trang nhỏ hơn ngưỡng trên, thì trang được thêm vào tập thường trú

Và cũng cần một ngưỡng dưới để giảm bớt khung trang của tiến trình

Tăng suất lỗi trang

Cấu trúc chương trình

Mãng A[1024, 1024] số nguyên

Mỗi dòng lưu trên một trang

Một frame

Chương trình 1

```
for j := 1 to 1024 do
for i := 1 to 1024 do
    A[i,j] := 0;
```

1024 x 1024 lỗi trang

Chương trình 2

```
for i := 1 to 1024 do
for j := 1 to 1024 do
    A[i,j] := 0;
```

1024 lỗi trang

Phân đoạn

Lược đồ quản lý bộ nhớ cho phép người dùng thấy được bộ nhớ.

Một chương trình là một tập các đoạn. Một đoạn là một đơn vị logic, như là:

main program,

procedure,

function,

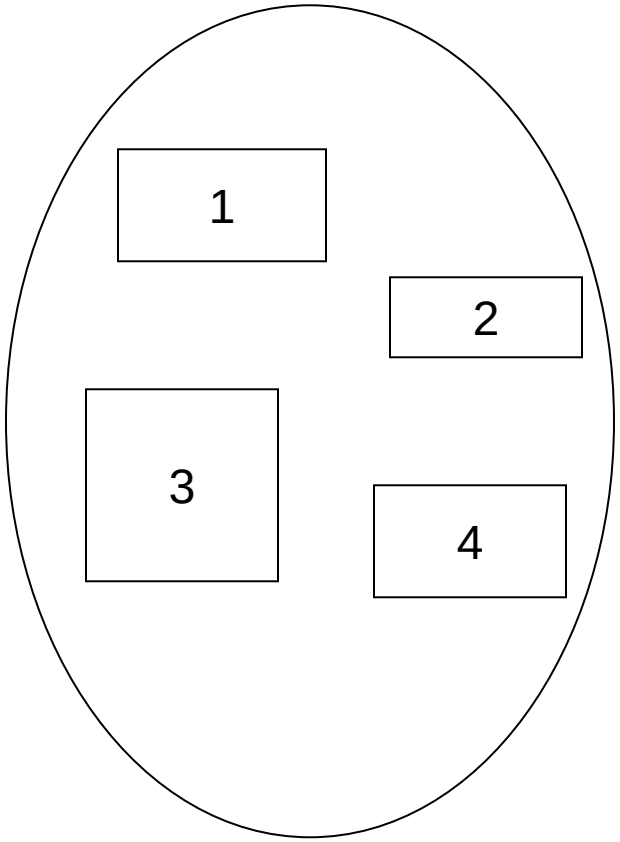
local variables, global variables,

common block,

stack,

symbol table, arrays

Sơ đồ logic của phân đoạn



Không gian người dùng

Không gian vật lý

Kiến trúc phân đoạn

Địa chỉ logic bao gồm 2 tham số: <segment-number, offset>

Segment table – ánh xạ địa chỉ vật lý; mỗi mục tin trên bảng lưu:

- base – lưu địa chỉ vật lý bắt đầu trên bộ nhớ.
- limit – xác định chiều dài của đoạn.

Segment-table base register (STBR) lưu vị trí của segment table trên bộ nhớ.

Segment-table length register (STLR) lưu số segment được sử dụng bởi người dùng; segment s là hợp lệ nếu s < STLR.

Kiến trúc phân đoạn (tt.)

Chia lại vùng nhớ.

- Động
- Thông qua segment table

Chia sẻ.

- Chia sẻ các đoạn
- Cùng số segment

Cấp phát.

- first fit/best fit
- Phân mảnh ngoài

Kiến trúc phân đoạn (tt.)

Bảo vệ. Mỗi mục tin trong segment table có thêm:

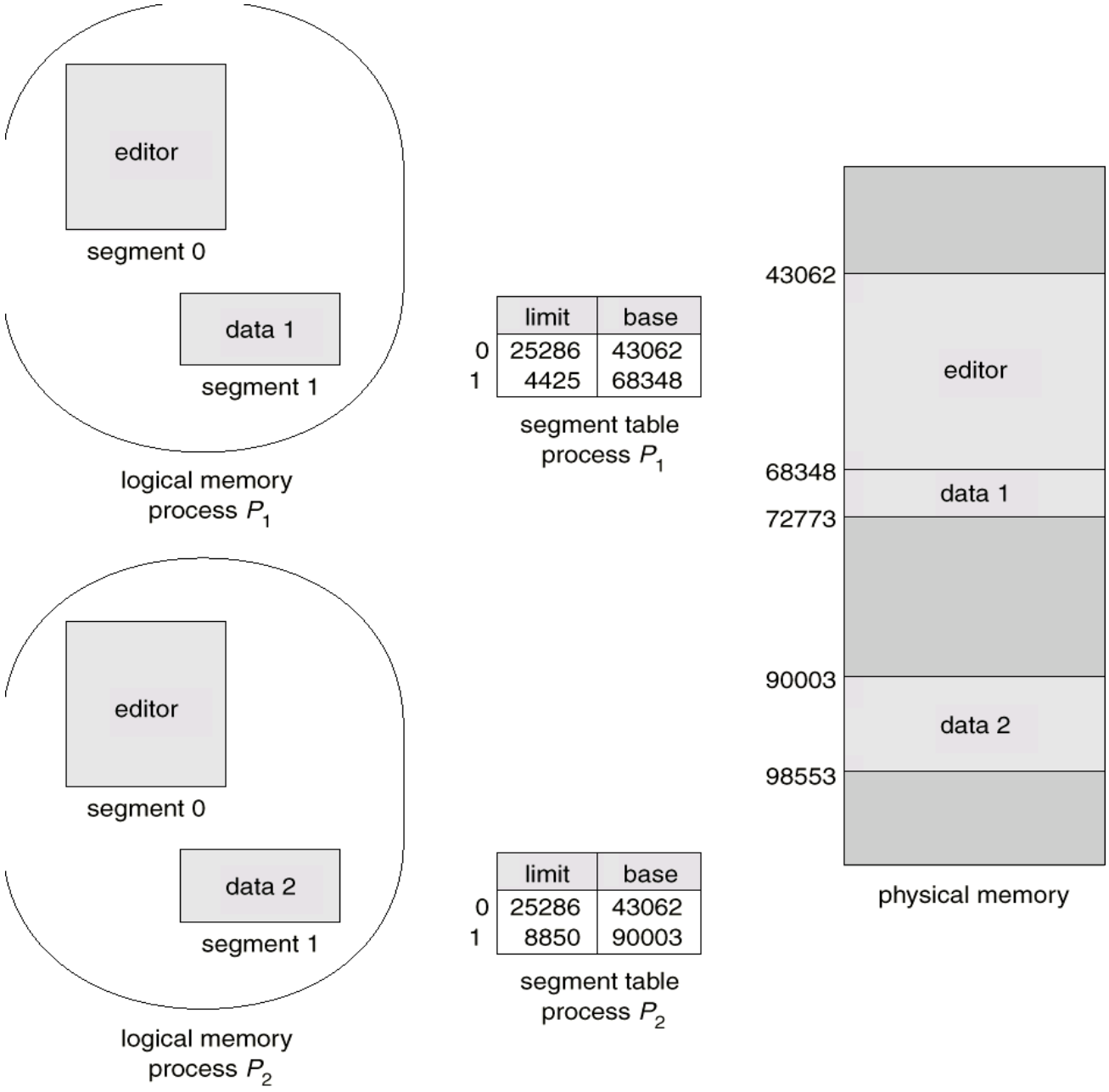
validation bit = 0 $\Rightarrow$ segment không hợp lệ

Quyền read/write/execute

Các bit bảo vệ gán với các segments

Bởi vì các đoạn độ dài khác nhau, cấp phát bộ nhớ là bài toán cấp phát vùng nhớ động.

Một ví dụ phân đoạn

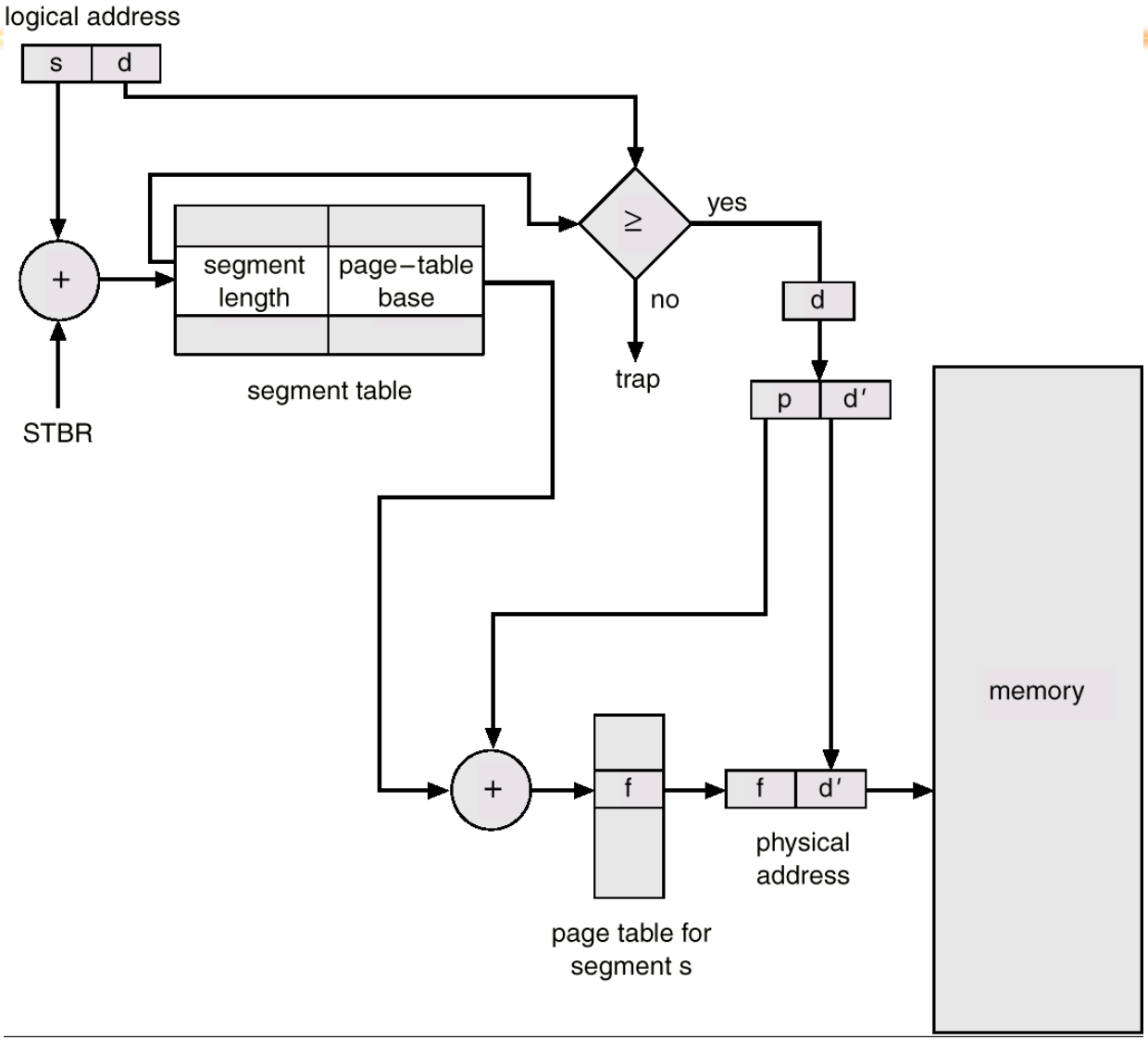

editor
segment 0
data 1
segment 1
logical memory
process P_1
limit
base
0
25286
43062
1
4425
68348
segment table
process P_1
43062
editor
68348
data 1
72773
90003
data 2
98553
physical memory
editor
segment 0
data 2
segment 1
logical memory
process P_2
limit
base
0
25286
43062
1
8850
90003
segment table
process P_2

Dùng phân trang để phân đoạn – MULTICS

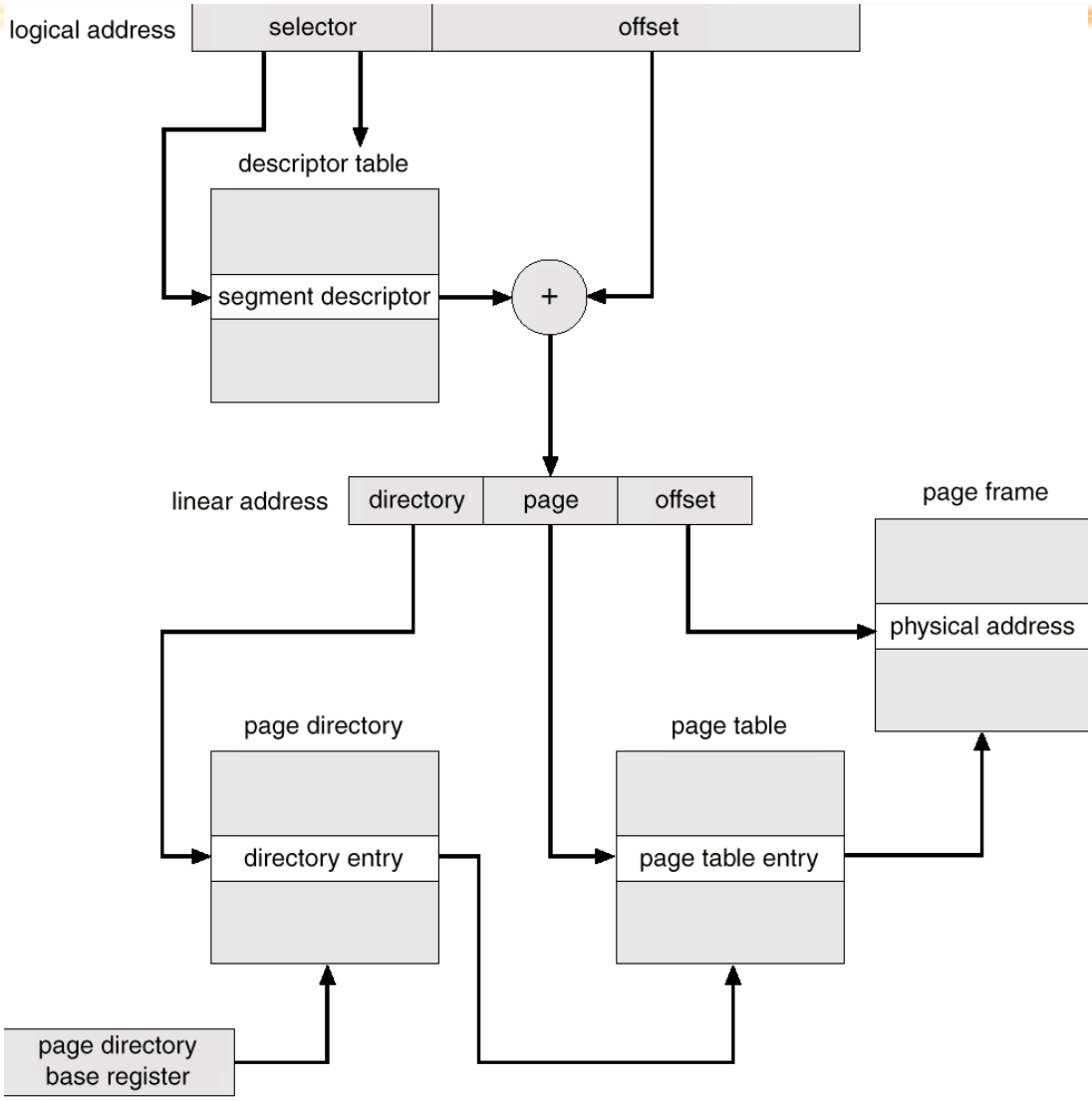
The MULTICS giải quyết bài toán phân mảnh ngoại vi và tìm kiếm các đoạn bằng cách phân trang các đoạn.

Giải pháp là: mẫu tin của segment-table không lưu địa chỉ base của segment, mà lưu địa chỉ base của *page table* cho segment này.

Sơ đồ chuyển đổi địa chỉ của MULTICS



Phân đoạn + phân trang Pentium 386



Tóm tắt

Bộ nhớ ảo tạo thành một lớp riêng biệt trong mô hình bộ nhớ đa cấp của chúng ta, để biểu diễn dung lượng bộ nhớ thật sự của hệ thống

- Điều này rất quan trọng "dễ dàng để lập trình"
- Hình dung bài toán phải kiểm tra cụ thể về kích thước vật lý của bộ nhớ và quản lý nó cho mỗi chương trình cụ thể đang thực thi

Hữu dụng cho việc ngăn chặn phân mảnh trong môi trường đa chương

Có thể cài đặt bằng phân trang (phân đoạn, hoặc cả hai)

Lỗi trang chi phí khá lớn,

- Cần có chính sách thay thế trang thật tốt

Phải cẩn thận vấn đề trì trệ hệ thống (thrashing)!!