



# Chương 4: Quản lý bộ nhớ

---

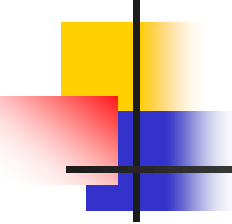
*Tìm hiểu về các cơ chế quản lý bộ nhớ trong của Hệ điều hành*



# Nội dung

---

- Các khái niệm
- Yêu cầu về chức năng quản lý bộ nhớ
- Các mô hình quản lý bộ nhớ
  - Dạng đơn giản
  - Quản lý bộ nhớ ảo



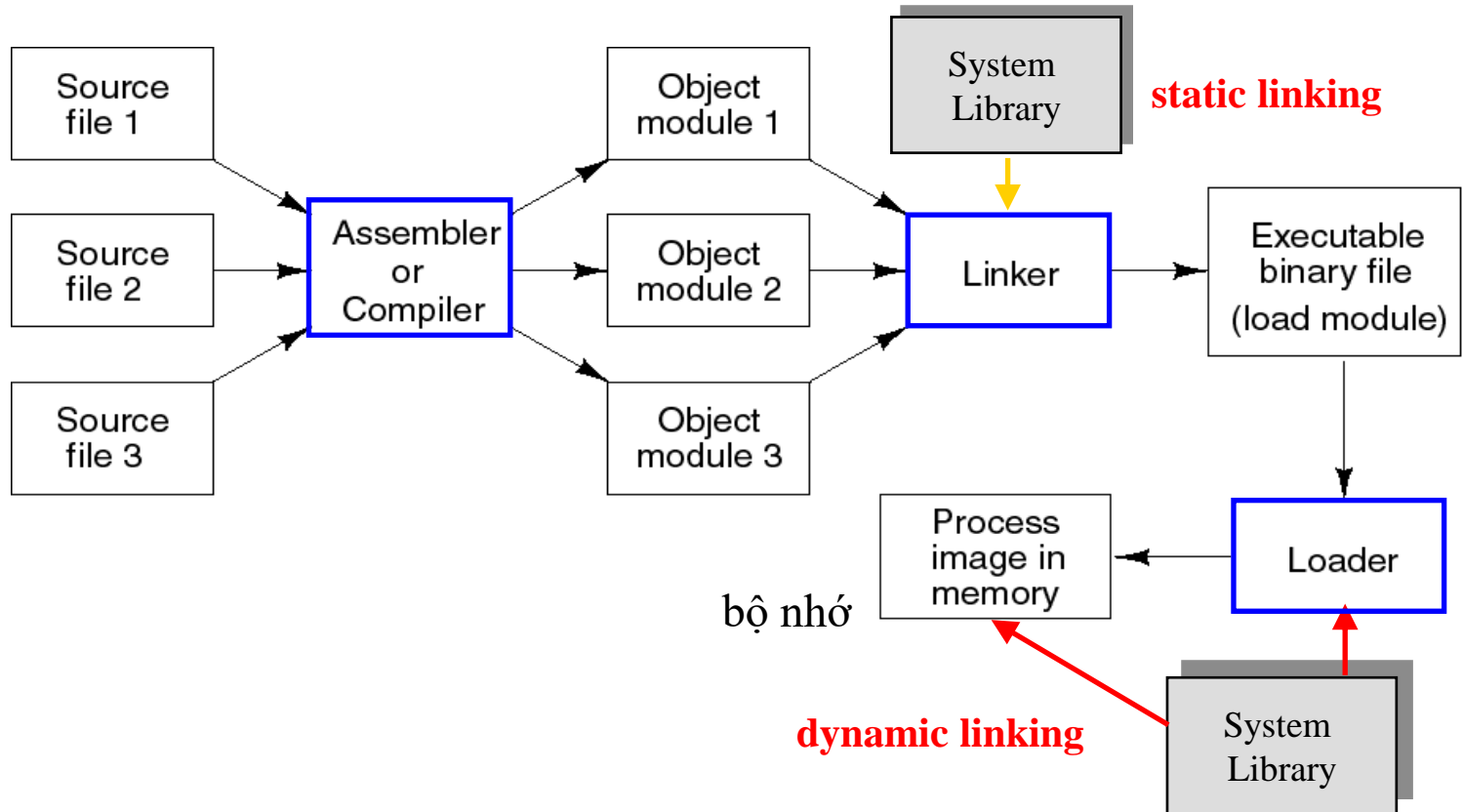
# 1. Khái niệm(1): *Kết nối địa chỉ*

---

- Kết nối địa chỉ:
  - Chương trình:
    - Bao gồm dữ liệu & mã lệnh
    - Phải được đưa từ đĩa vào bộ nhớ trong và được đặt vào một tiến trình để nó có thể chạy.
  - *Input queue* – tập hợp các tiến trình trên đĩa đang chờ để được đưa vào trong bộ nhớ để chạy chương trình.
  - Chương trình của người sử dụng phải đi qua một số bước trước khi được chạy.

# 1. Khái niệm(2): *Kết nối địa chỉ*

- Các bước thực hiện chương trình người dùng:





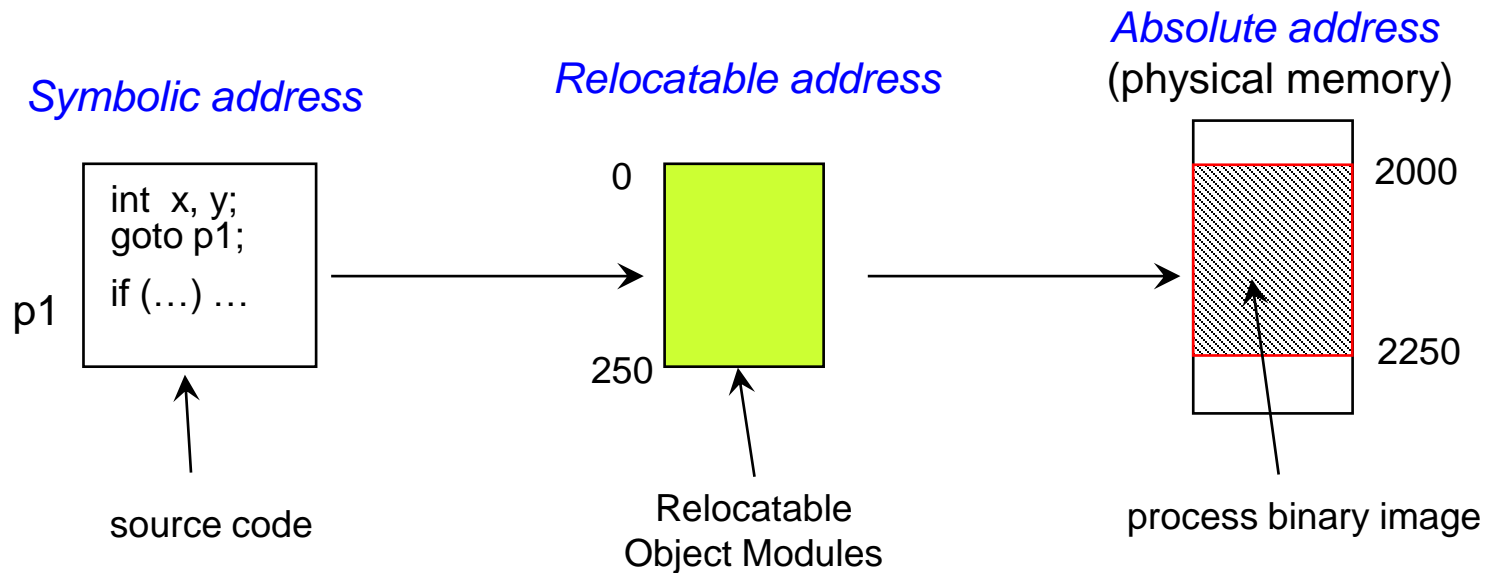
# 1. Khái niệm(3): *Kết nối địa chỉ*

---

- **Kết nối các lệnh, dữ liệu tới địa chỉ bộ nhớ: quá trình gán địa chỉ của các lệnh và dữ liệu tới các địa chỉ bộ nhớ có thể xảy ra 3 giai đoạn khác nhau:**
  - 1. Compile time:** Nếu vị trí bộ nhớ được biết trước, mã chính xác (absolute code) có thể được sinh ra; phải biên dịch lại mã nếu vị trí bắt đầu thay đổi.
  - 2. Load time:** Phải sinh ra mã *có thể tái định vị (relocatable code)* nếu không biết vị trí bộ nhớ ở giai đoạn biên dịch.
  - 3. Execution time:** Sự liên kết bị hoãn lại đến giai đoạn chạy nếu trong quá trình thực hiện, tiến trình có thể bị chuyển từ một đoạn bộ nhớ đến đoạn khác. Cần có sự hỗ trợ phần cứng để ánh xạ địa chỉ (ví dụ, *base* và *limit registers*).

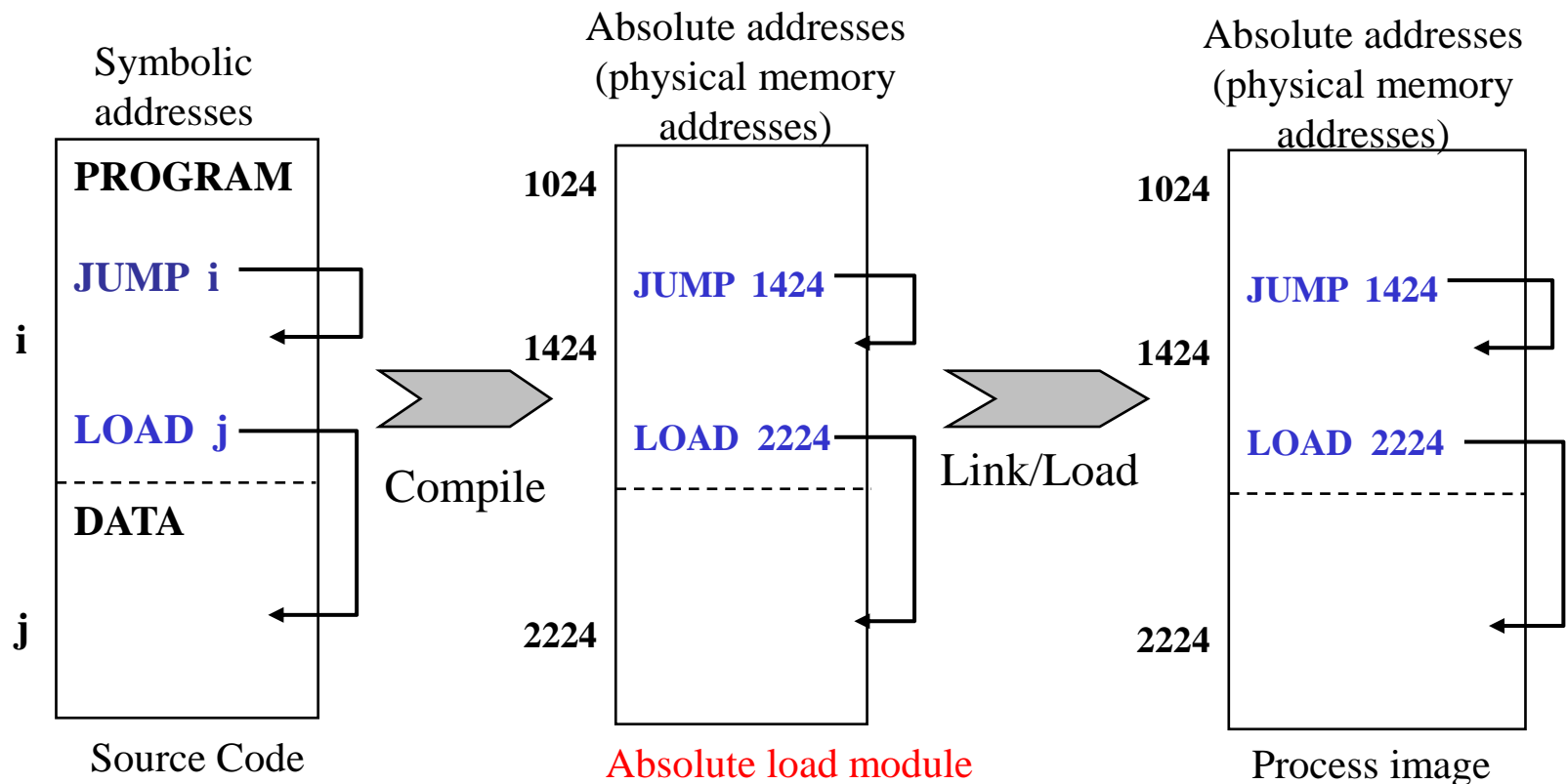
# 1. Khái niệm(3): *Liên kết địa chỉ*

## ■ Minh họa:



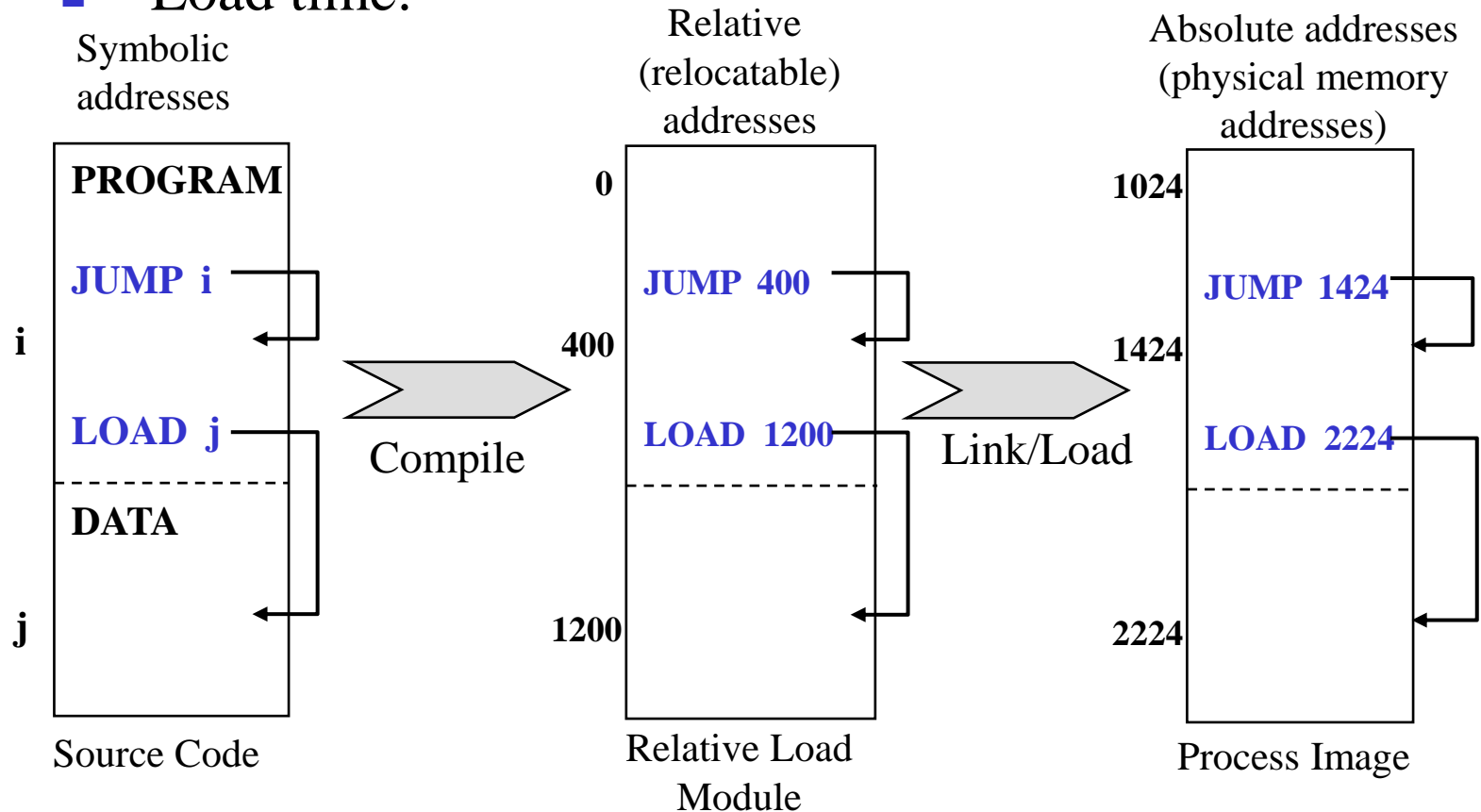
# 1. Khái niệm(3): *Liên kết địa chỉ*

- Compile time:



# 1. Khái niệm(3): *Liên kết địa chỉ*

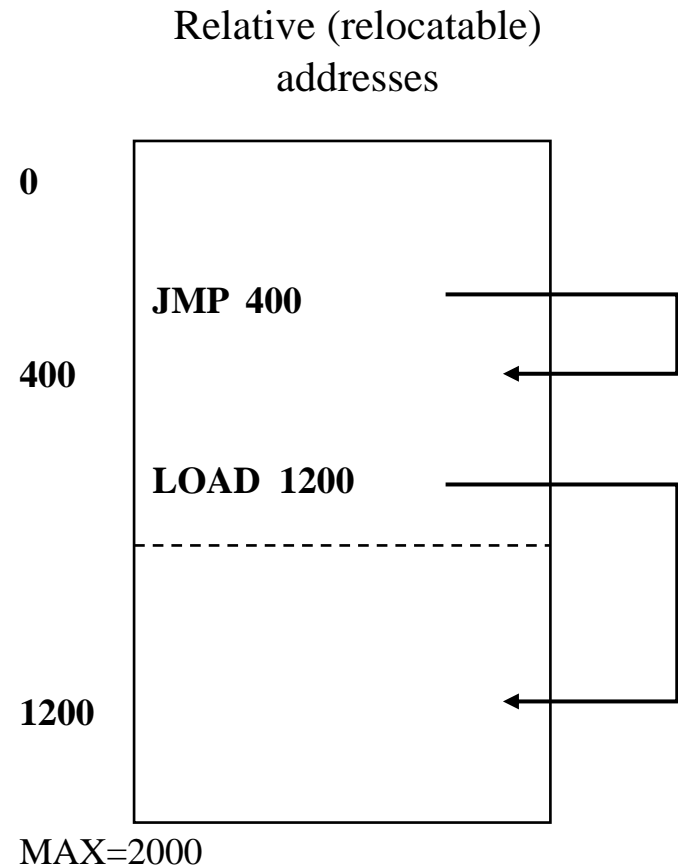
## ■ Load time:

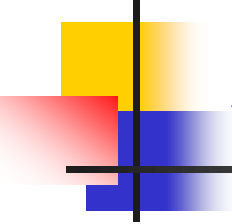




# 1. Khái niệm(3): *Liên kết địa chỉ*

- Execution time:

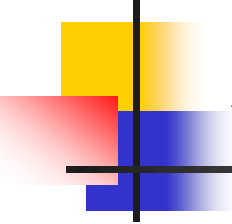




# 1. Khái niệm(4): *Địa chỉ logic & Địa chỉ vật lý*

---

- Khái niệm *không gian địa chỉ logic (logical address space)* được tách riêng với *không gian địa chỉ vật lý (physical address space)* để quản lý bộ nhớ thích hợp.
  - *Logical address* – được tạo ra bởi CPU, còn gọi là địa chỉ ảo (*virtual address*).
  - *Physical address* – địa chỉ được nhận biết bởi đơn vị bộ nhớ (*memory unit*).
- Các địa chỉ logic (ảo) và vật lý là như nhau trong các giai đoạn gắn kết địa chỉ *compile-time* và *load-time*; chúng khác nhau trong *execution-time*.



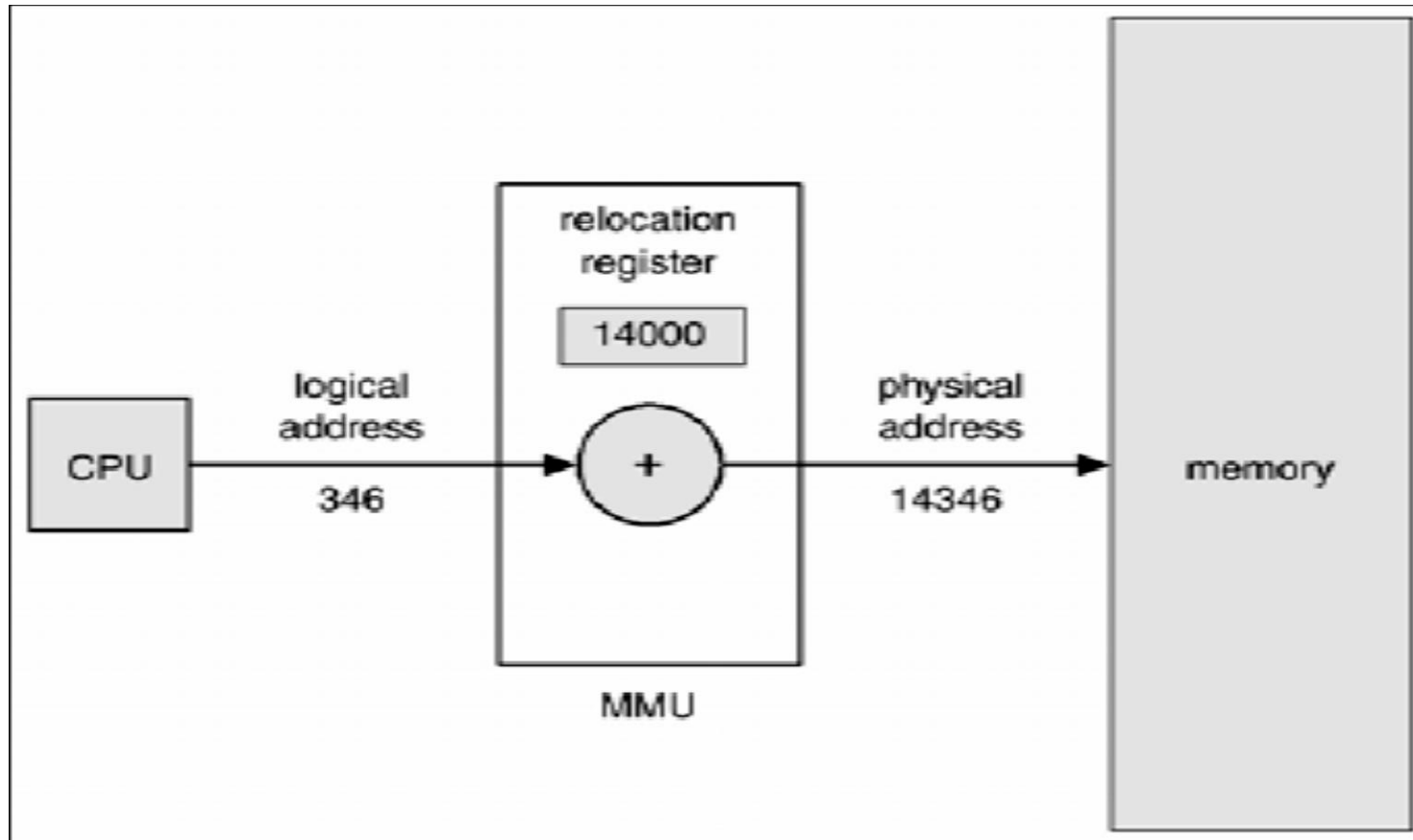
# 1. Khái niệm(5): *Địa chỉ logic & Địa chỉ vật lý*

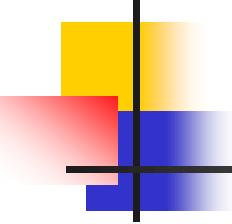
---

## ■ Memory-Management Unit (MMU)

- Là thiết bị phần cứng ánh xạ địa chỉ ảo tới địa chỉ vật lý.
- Trong lược đồ MMU, giá trị trong thanh ghi định vị (relocation register) được cộng với tất cả địa chỉ được sinh ra bởi tiến trình của người dùng tại thời điểm nó được gửi tới bộ nhớ.
- Chương trình của người dùng làm việc với các địa chỉ logic; nó không bao giờ nhận ra các địa chỉ vật lý thực.

# 1. Khái niệm(6): *Địa chỉ logic & Địa chỉ vật lý*





# 1. Khái niệm(7): *Dynamic Loading-Tải động*

---

- Chương trình(routine) chỉ được nạp vào bộ nhớ khi nó được gọi.
- Sử dụng không gian bộ nhớ tốt hơn; tiến trình không dùng đến thì không bao giờ được nạp.
- Hữu ích trong trường hợp số lượng lớn mã cần xử lý hiếm khi xuất hiện.
- Không yêu cầu sự hỗ trợ đặc biệt từ HĐH, được thực hiện thông qua thiết kế chương trình.



# 1. Khái niệm(8): *Dynamic Linking-Liên kết động*

---

- Việc liên kết hoãn lại đến execution time.
- Stub: đoạn mã nhỏ
  - Sử dụng để định vị các chương trình thư viện cư trú trong bộ nhớ (memory-resident library routine) thích hợp.
- Khi được thực hiện, stub kiểm tra routine cần đến có trong bộ nhớ của tiến trình:
  - **nếu chưa** thì chương trình nạp routine vào bộ nhớ
  - **nếu rồi**: stub tự thay thế chính nó bởi địa chỉ của thường trình rồi thực hiện thường trình đó.
- Liên kết động đặc biệt hữu dụng đối với các thư viện chương trình, nhất là trong việc cập nhật thư viện (vd sửa lỗi)



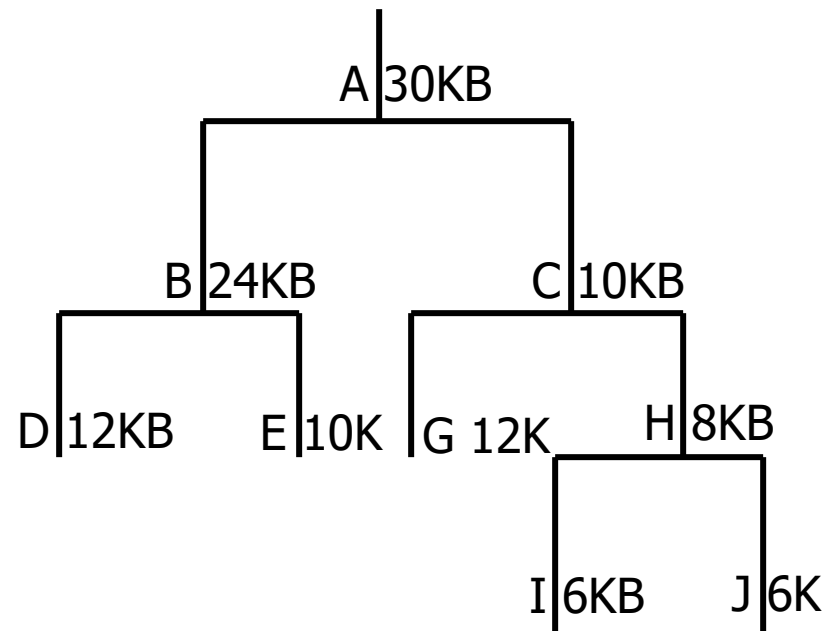
# 1. Khái niệm(9): cơ chế *Overlays*

---

- *Ý tưởng*: chỉ giữ trong bộ nhớ những lệnh và dữ liệu cần đến tại mọi thời điểm( thường là các module tải) -> giảm không gian nhớ liên tục dành cho ctr
- Cơ chế Overlay:
  - Cho phép tổ chức ctr thành các đơn vị ctr(module):
  - Module luôn tồn tại trong quá trình thực hiện -> module chương trình chính.
  - Quan hệ độc lập/phụ thuộc chỉ sự có mặt của 1 nhóm module trong bộ nhớ đòi hỏi/không đòi hỏi sự có mặt của một nhóm module khác
    - ⇒ Các module độc lập không cần thiết phải có mặt đồng thời trong bộ nhớ
- Cần đến khi tiến trình có dung lượng lớn hơn bộ nhớ được cấp phát cho nó.

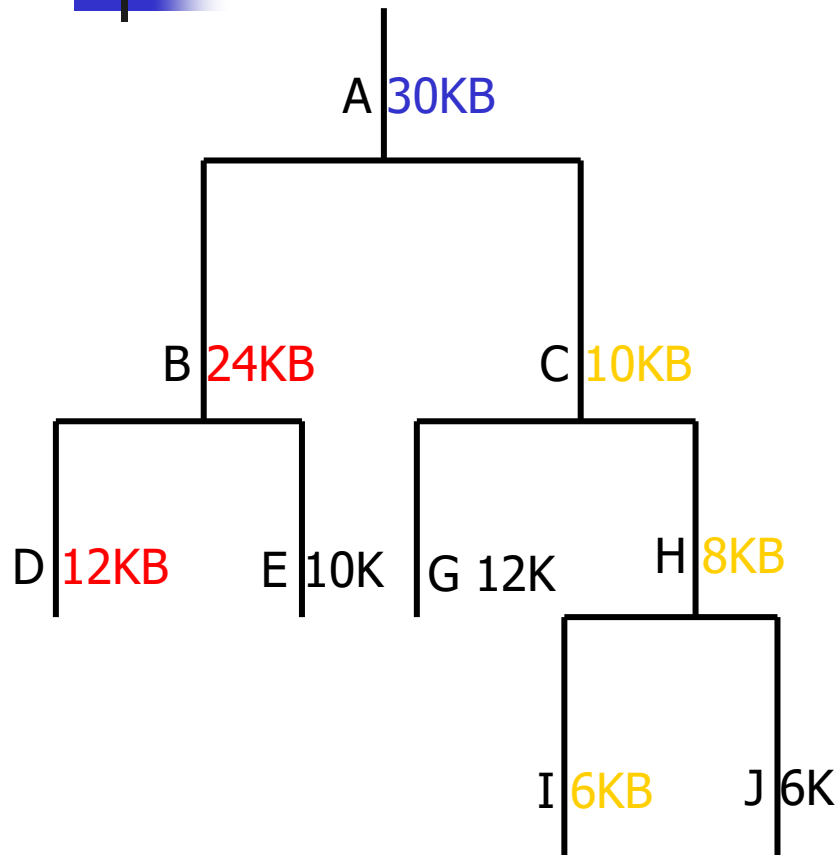
# 1. Khái niệm(10): cơ chế *Overlays(tt)*

- Ex: chương trình gồm 1 module ctr chính A(đòi hỏi bộ nhớ 30k); mọi module ctr khác đều phụ thuộc vào nó
- Module sử dụng 2 module độc lập B(24KB), C(10KB)
- B sử dụng 2 module độc lập D(12KB), E(10K)
- C sử dụng 2 module độc lập G(12KB), H(8KB)
- H sử dụng 2 module độc lập I(6KB), J(6KB)





# 1. Khái niệm(10): cơ chế *Overlays(tt)*



## ■ Overlay:

- Cây chương trình gốc B cần:  $24+12=36K$
- Cây chương trình gốc C cần:  $10+8+6=24K$
- B, C độc lập không có mặt đồng thời  
 $\Rightarrow$  cần 36K(lấy module lớn)
- $\Rightarrow$  Cả chương trình cần:  $30 \text{ (cho A)} + 36 = 66K$

## ■ Không sử dụng overlay cần: $30+24+10+12+10+12+8+6+6=118K$

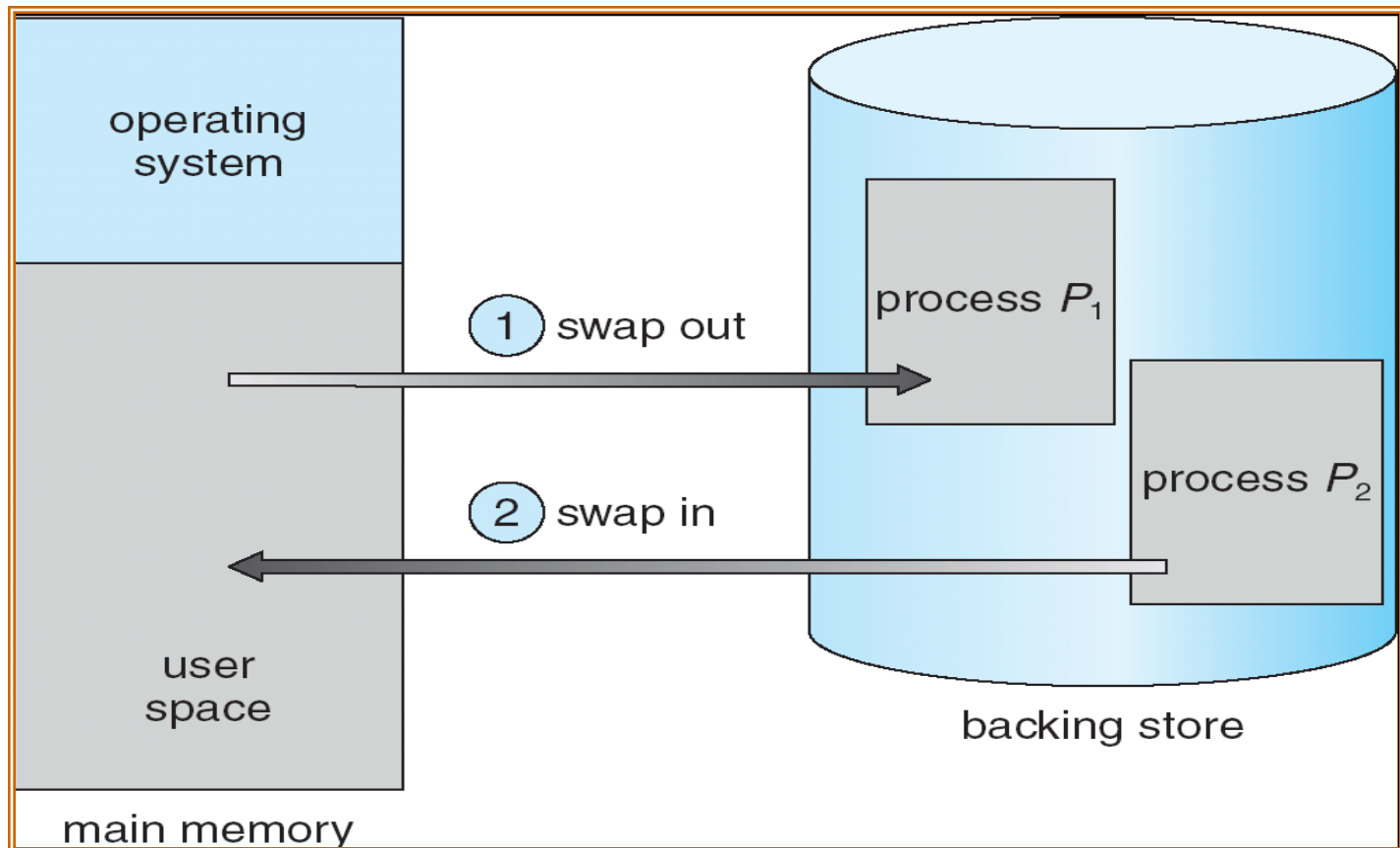


# 1. Khái niệm(10): *Swapping*

---

- Một tiến trình có thể được tạm thời hoán đổi ra khỏi bộ nhớ tới *backing store*, và rồi được đưa trở lại bộ nhớ để thực hiện tiếp.
- Backing store – thiết bị nhớ thứ cấp đủ lớn( đĩa từ) để cung cấp bản sao của tất cả hình ảnh bộ nhớ cho tất cả người sử dụng; phải cung cấp sự truy nhập trực tiếp tới các hình ảnh bộ nhớ này.
- *Roll out, roll in* – biến thể hoán đổi được sử dụng cho thuật giải lập lịch dựa trên mức ưu tiên (priority-based scheduling); tiến trình có mức ưu tiên thấp hơn bị thay ra để tiến trình có mức ưu tiên cao hơn có thể được nạp và thực hiện.
- Phần lớn thời gian hoán đổi là thời gian chuyển dữ liệu; tổng thời gian chuyển tỷ lệ thuận với dung lượng bộ nhớ hoán đổi.
- *Swap out*: chọn tiến trình để đưa ra backing store
- *Swap in*: chọn tiến trình từ backing store để đưa vào bộ nhớ trong
- Trong các hệ điều hành sử dụng swapping, tồn tại module hệ thống ***swapper*** có chức năng: chọn tiến trình swap out, chọn tiến trình swap in, định vị & quản lý không gian chuyển

# 1. Khái niệm(10): *Swapping*



# 1. Khái niệm(1 1): Sự phân mảnh - Fragmentation

- **External Fragmentation** – tổng không gian bộ nhớ thực tế đủ đáp ứng yêu cầu, nhưng nó không nằm kề nhau.
- **Internal Fragmentation** – bộ nhớ được phân phối có thể lớn hơn không đáng kể so với bộ nhớ được yêu cầu; sự khác biệt kích thước này là bộ nhớ bên trong một phân vùng, nhưng không được sử dụng.
- Làm giảm external fragmentation bằng cách nén lại (compaction)
  - Di chuyển các nội dung bộ nhớ để đặt tất cả các vùng nhớ tự do lại với nhau thành một khối lớn.
  - Kết khối chỉ có thể tiến hành nếu sự tái định vị là động, và nó được thực hiện trong execution time.



## 2. Các yêu cầu đối với quản lý bộ nhớ(1)

---

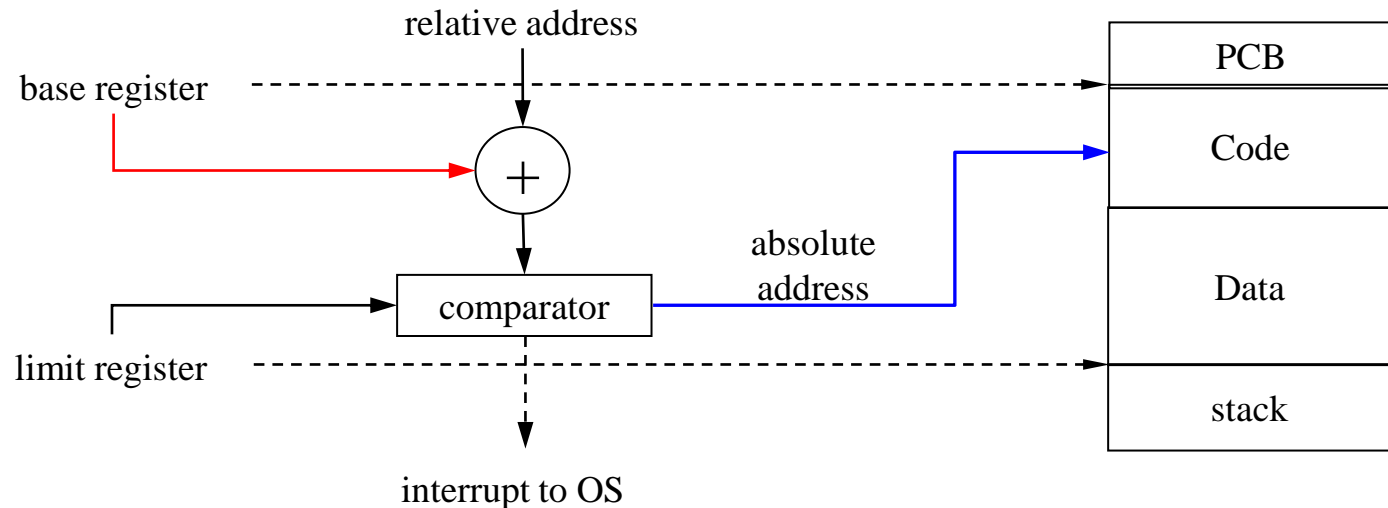
- Tái định vị (relocation)
- Bảo vệ (protection)
- Chia sẻ (sharing)
- Tổ chức lôgic (logical organization)
- Tổ chức vật lý (physical organization)

## 2. Các yêu cầu đối với quản lý bộ nhớ(2): Tái định vị (relocation)

### ■ Vấn đề:

- Khi một tiến trình được đưa ra khỏi bộ nhớ, sau đó lại được đưa vào ở một địa chỉ khác
- Khi tiến trình có thể lại ở vị trí khác
- ⇒ Yêu cầu tái định vị

⇒ Giải pháp: dùng địa chỉ tương đối và cơ chế chuyển địa chỉ tương đối thành địa chỉ thực( ví dụ dùng thanh ghi *base*, *limit*)





# 3. Các mô hình quản lý bộ nhớ

---

- Mô hình đơn giản
  - Mô hình phân phối liên tục: không có cơ chế swapping & bộ nhớ ảo
    - Mono-programming
    - Multi-programming with fixed partitions
    - Multi-programming with variant partitions
  - Các mô hình phân phối gián đoạn
    - Simple paging
    - Simple segmentation
- Mô hình bộ nhớ ảo:
  - Paging
  - Segmentation
  - Hybrid (segmentation + paging)



## 3.1. Mô hình đơn giản

---

- Các mô hình phân phối liên tục: không có cơ chế swapping & bộ nhớ ảo
  - Mono-programming
  - Multi-programming with fixed partitions
  - Multi-programming with variant partitions
- Các mô hình phân phối không liên tục
  - Simple paging
  - Simple segmentation





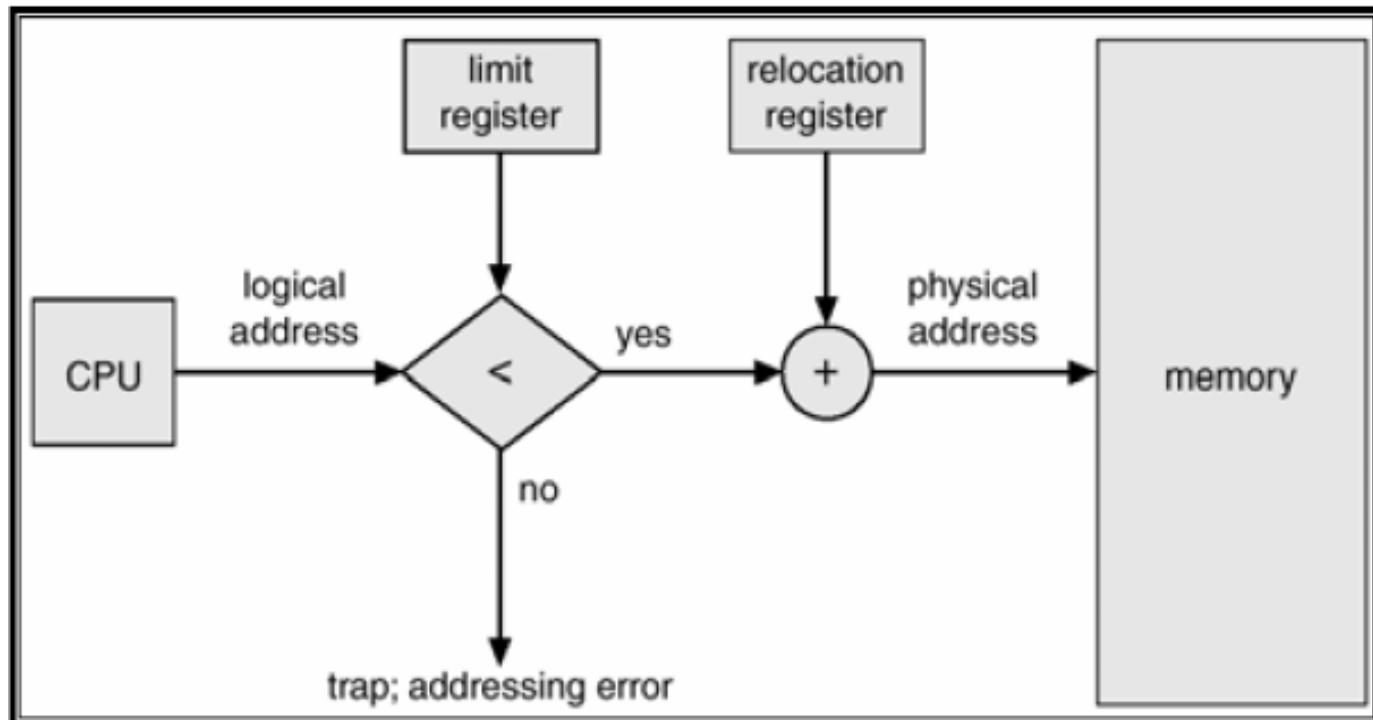
## 3.1.1. Phân phối liên tục(1)

---

- Bộ nhớ chính được chia thành 2 phần:
  - Nơi HĐH cư trú, thường ở vùng nhớ thấp, chứa bảng vector ngắt.
  - Các tiến trình của người dùng được chứa trong vùng nhớ cao.
- Phân phối phân vùng đơn (Single-partition allocation)
  - Mỗi tiến trình chỉ làm việc trong vùng nhớ đã được phân
- Lược đồ thanh ghi định vị được sử dụng để bảo vệ các tiến trình của người sử dụng từ các tiến trình khác và từ sự thay đổi dữ liệu và mã HĐH.
- **Relocation register** chứa giá trị **địa chỉ vật lý nhỏ nhất**; **limit register** chứa **dải địa chỉ logic** - mỗi địa chỉ logic phải nhỏ hơn limit register.

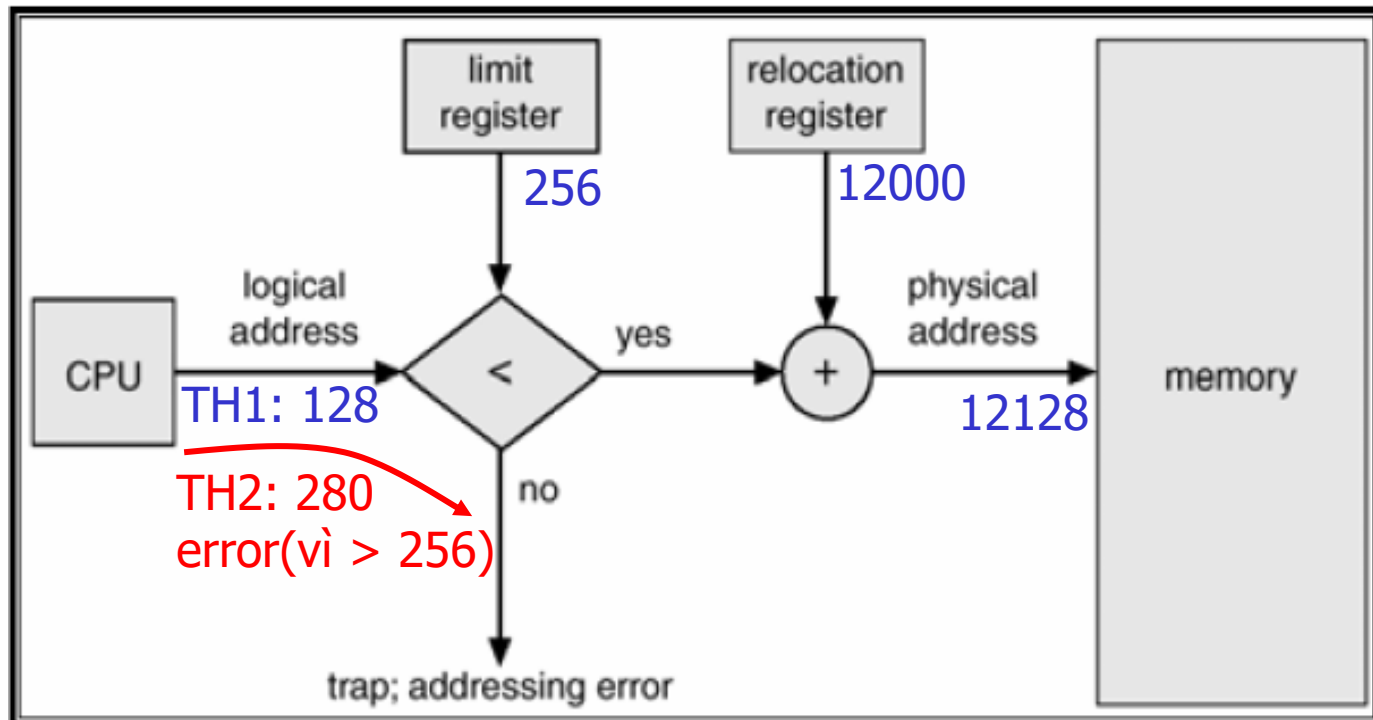
## 3.1.1. Phân phối liên tục(2)

- Ví dụ các thanh ghi Relocation và Limit



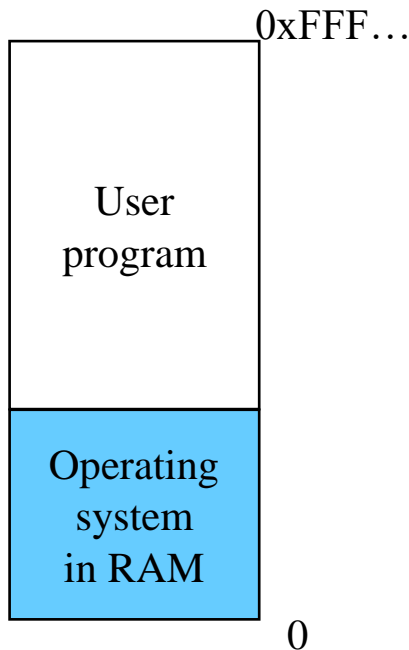
## 3.1.1. Phân phối liên tục(2)

- Ví dụ các thanh ghi Relocation và Limit(tt)

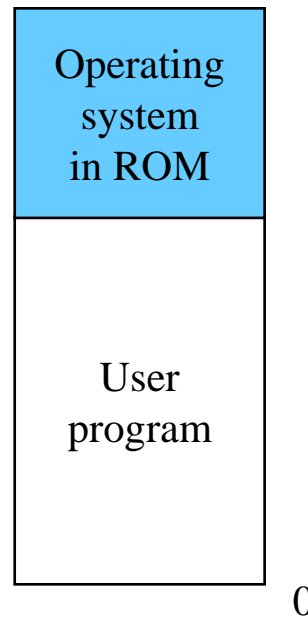


# 3.1.1. Phân phối liên tục(3): Mono-programming

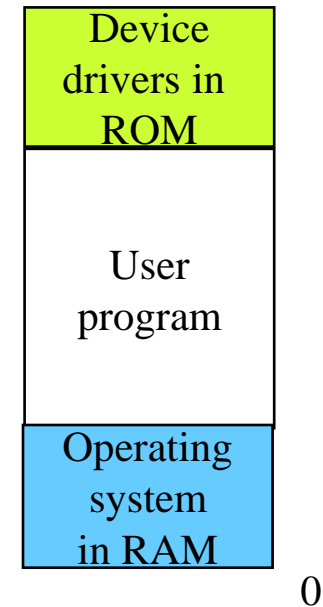
- Operation system + 1 user program



Old mainframes,  
minicomputers



Palmtop, embedded  
systems



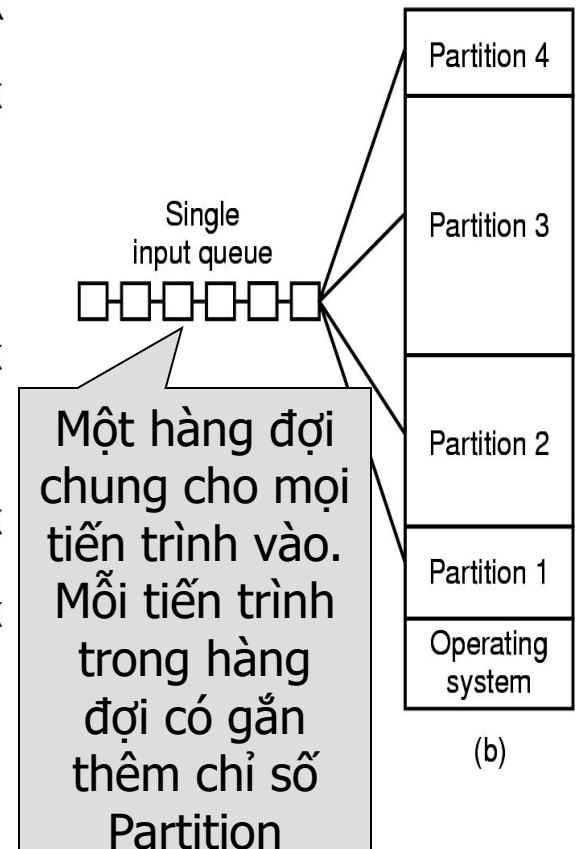
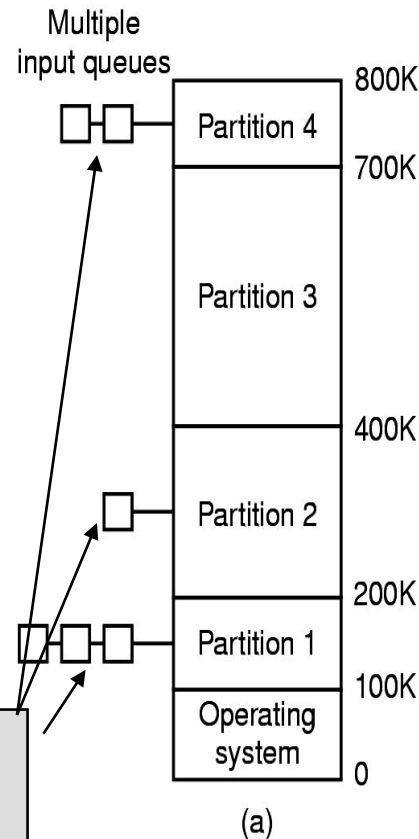
PC system  
(MSDOS)

# 3.1.1. Cấp phát liên tục(4):

## MultiProg – Fixed Partitions(Đa ctr với phân vùng cố định)

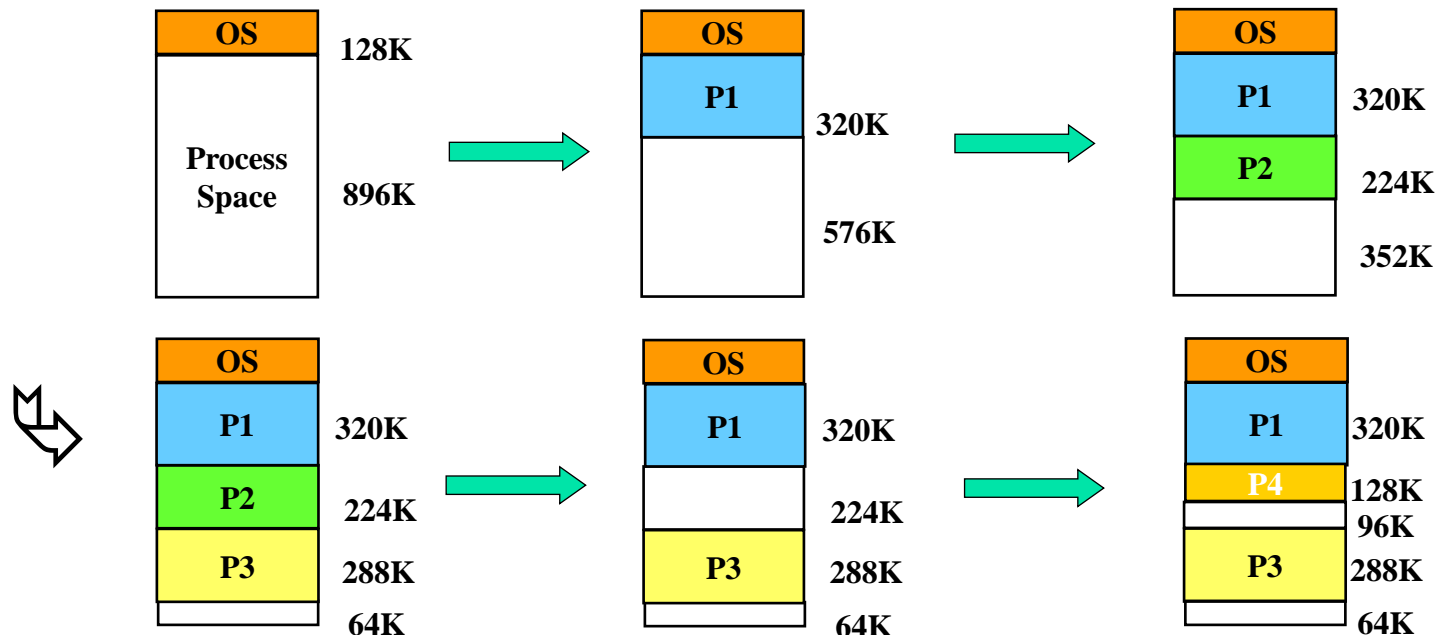
- Bộ nhớ được chia thành các partition (phân vùng or miền or chương..) cố định; tên partition, địa chỉ, dung lượng được gán trong quá trình khởi tạo hệ điều hành
- Partition 0 dành cho nhân hệ điều hành
- Mỗi tiến trình được tải vào một partition nhất định và chỉ hoạt động trong partition chứa nó.
- Số tiến trình đồng thời trong bộ nhớ xác định trước

Mỗi phân vùng có một hàng đợi tiến trình gắn với nó



# 3.1.1. Phân phối liên tục(5): MultiProgmg-Variant partitions( Đa ctr với phân vùng thay đổi)

- Tương ứng với chế độ MVT(Multiprogramming w/ a Variable number of Tasks) của hệ điều hành
- Các tiến trình được nạp liên tục vào bộ nhớ đến khi còn đủ dung lượng.
- Số lượng tiến trình đồng thời trong bộ nhớ không định trước





## 3.1.2. Phân phối không liên tục

---

- Phân trang đơn giản (simple paging)
- Phân đoạn đơn giản (simple segmenting)



## 3.1.2.1. Phân trang đơn giản(1)

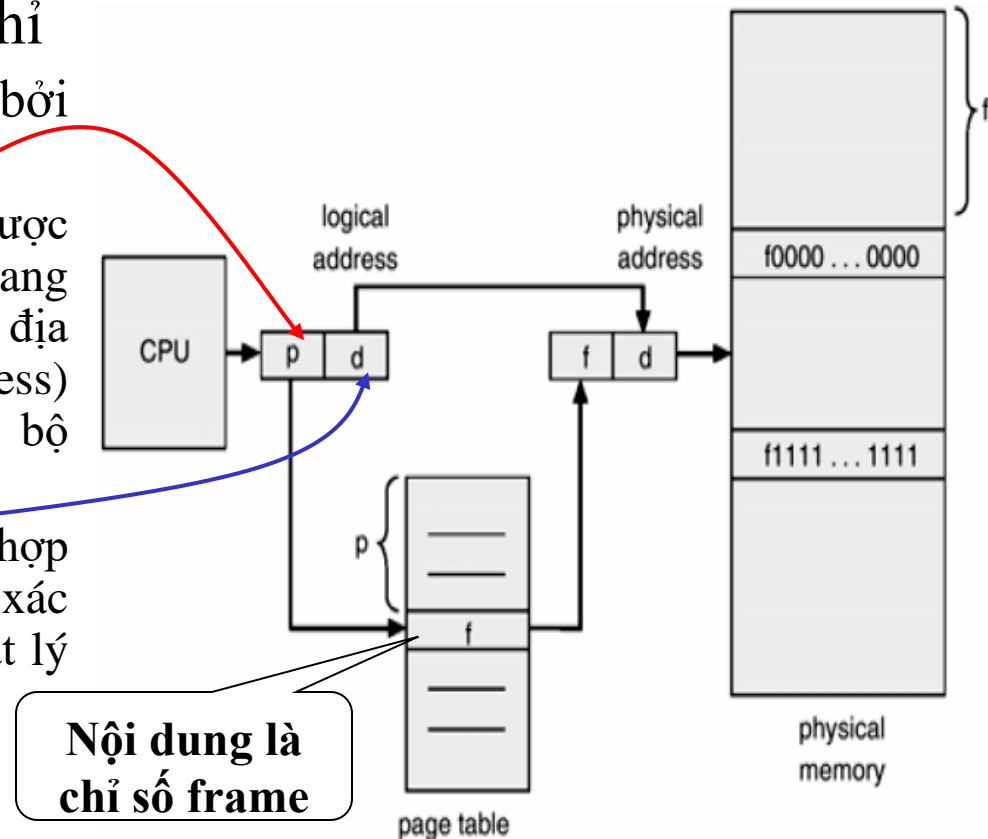
---

- Không gian địa chỉ logic của một tiến trình có thể không kề nhau; tiến trình được phân phối bộ nhớ vật lý bất kỳ lúc nào khi bộ nhớ sẵn có.
- Chia bộ nhớ vật lý thành những khối có kích thước cố định là lũy thừa của 2 (512 bytes - 16 MB), được gọi là các **frame(page vật lý)**.
- Chia bộ nhớ logic( dành cho các tiến trình) thành các khối cùng kích thước - các **page**. Mỗi **page** có **kích thước = 1 frame**
- Luôn theo dõi tất cả các frame còn trống.
- Để chạy một chương trình có kích thước  $n$  pages, cần phải tìm  $n$  frames còn trống và nạp chương trình.
- Thiết lập một bảng phân trang (page table) để biên dịch (translate) các địa chỉ logic thành địa chỉ vật lý.
- Nội dung mỗi phần tử trong **page table** cho biết chỉ số **frame**( địa chỉ cơ sở) của bộ nhớ vật lý



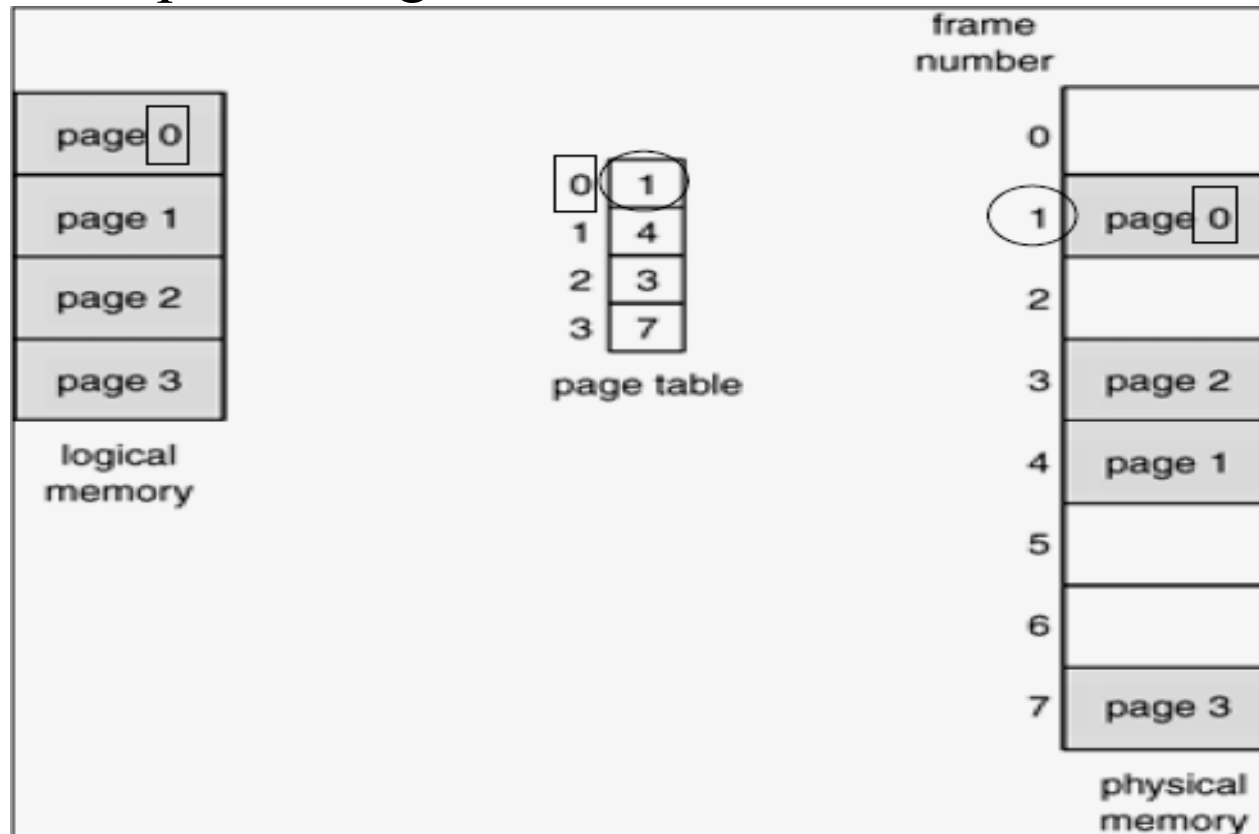
## 3.1.2.1. Phân trang đơn giản(2)

- Lược đồ biên dịch địa chỉ
  - Địa chỉ được tạo ra bởi CPU được chia thành:
    - *Page number (p)* – được sử dụng làm chỉ số trang trong *page table*, chứa địa chỉ cơ sở (base address) của mỗi trang trong bộ nhớ vật lý.
    - *Page offset (d)* – kết hợp với địa chỉ cơ sở để xác định địa chỉ bộ nhớ vật lý được gửi đến bộ nhớ.



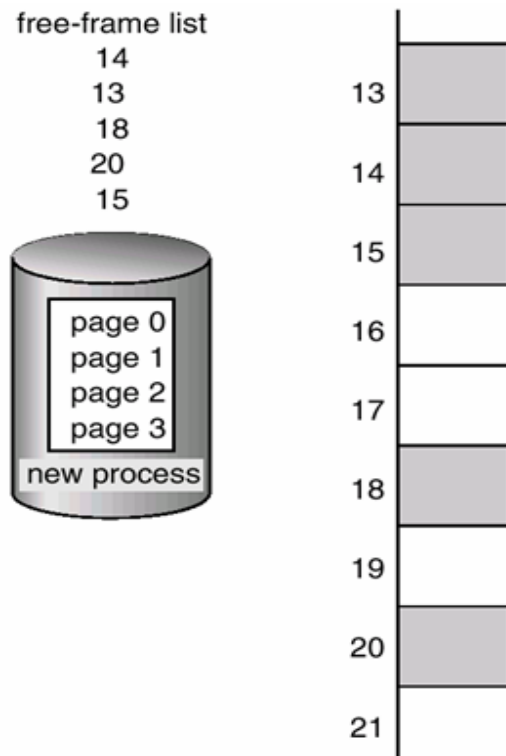
## 3.1.2.1. Phân trang đơn giản(3)

- Ví dụ phân trang

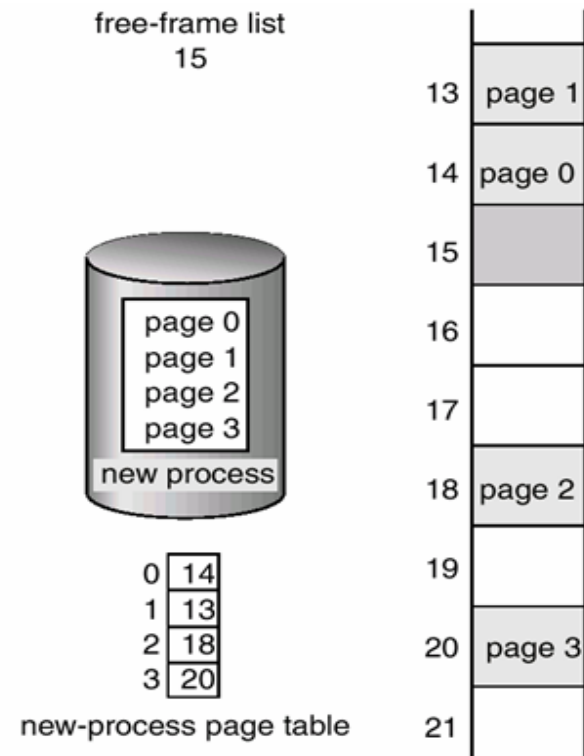


## 3.1.2.1. Phân trang đơn giản(4)

- Ví dụ phân trang(tiếp): Các Frame rỗi



(a)



(b)



## 3.1.2.1. Phân trang đơn giản(5)

---

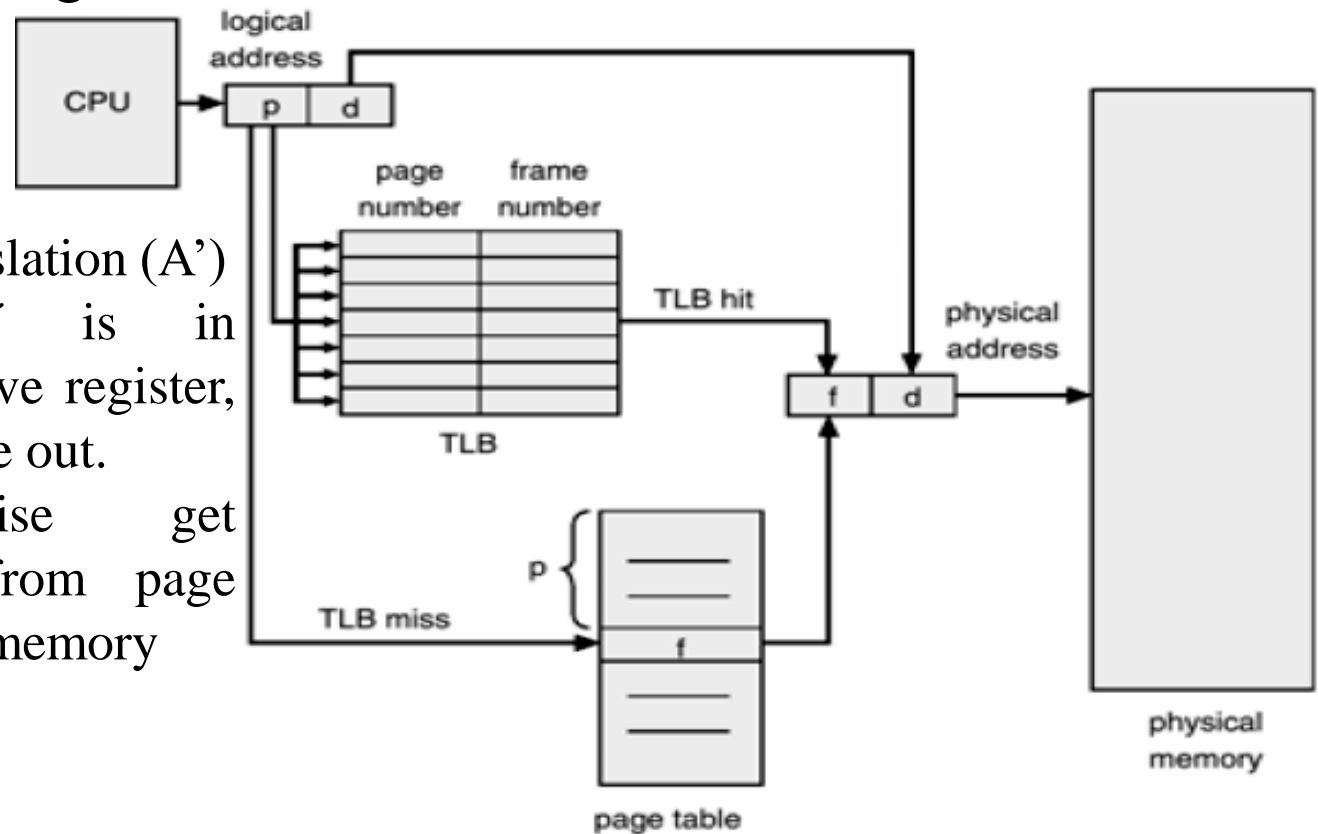
- Hoạt động của Page Table
  - Page table được lưu trong main memory.
  - *Page-table base register* (PTBR) chỉ tới page table.
  - *Page-table length register* (PRLR) cho biết kích thước của page table.
  - Trong lược đồ phân trang, mọi sự truy nhập lệnh/dữ liệu yêu cầu 2 lần truy nhập bộ nhớ: một cho page table và một cho lệnh/dữ liệu(địa chỉ ô nhớ trong page). → truy nhập chậm hơn.
  - Vấn đề 2 lần truy nhập bộ nhớ có thể được giải quyết bằng cách sử dụng một bộ nhớ cache tra cứu nhanh đặc biệt gọi là *bộ nhớ liên kết - associative memory or translation look-aside buffers (TLB)*

## 3.1.2.1. Phân trang đơn giản(6)

### ■ Phân trang với TLB

Address translation (A')

- If A' is in associative register, get frame out.
- Otherwise get frame from page table in memory



# 3.1.2.1. Phân trang đơn giản(7): Effective Access Time

- Associative Lookup =  $\epsilon$  time unit
- Assume memory cycle time is 1 microsecond
- Hit ratio – percentage of times that a page number is found in the associative registers; rati~~on~~ related to number of associative registers(*Phần trăm số lần tìm thấy trang trong cache*). Suppose **Hit ratio** =  $\alpha$ . In case **Not found** is  $1 - \alpha$
- Effective Access Time (EAT)(*Thời gian truy cập hiệu quả*)

$$\begin{aligned} \text{EAT} &= (1 + \epsilon) \alpha + (2 + \epsilon)(1 - \alpha) \\ &= 2 + \epsilon - \alpha \end{aligned}$$

Vì:  $(1 + \epsilon)$  là thời gian khi tìm thấy trang trong cache và lấy dữ liệu trong bộ nhớ( chu kỳ truy nhập bộ nhớ là 1)

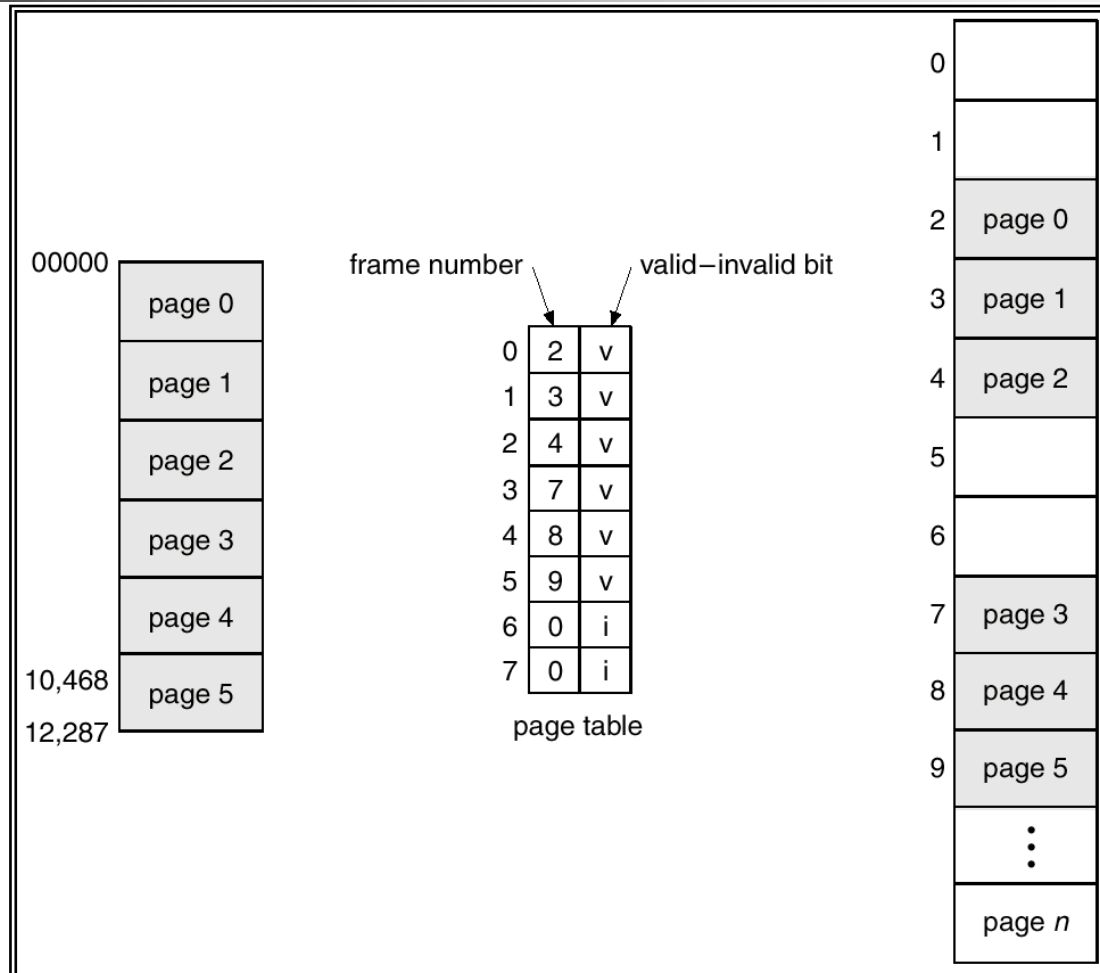
$(2 + \epsilon)$  là thời gian khi không tìm thấy trang trong cache + thời gian truy cập bảng trang trong bộ nhớ + thời gian truy cập dữ liệu trong bộ nhớ

## 3.1.2.1. Phân trang đơn giản(8): Memory Protection

---

- Memory protection implemented by associating *protection bit* with each frame.
- *Valid-invalid* bit attached to each entry in the page table:
  - “valid” indicates that the associated page is in the process’ logical address space, and is thus a legal page.
  - “invalid” indicates that the page is not in the process’ logical address space.

# 3.1.2.1. Phân trang đơn giản(9): Memory Protection(cont.)





# 3.1.2.1. Phân trang đơn giản(10): Page Table Structure



---

- Thường dùng 3 loại cấu trúc bảng trang:
  - Hierarchical Paging (bảng trang phân cấp)
  - Hashed Page Tables (bảng trang băm)
  - Inverted Page Tables (bảng trang nghịch đảo)



### 3.1.2.1. Phân trang đơn giản(11): Hierarchical Page Tables

---

- Break up **the logical address space** into **multiple page tables**.
- A simple technique is a two-level page table.

# 3.1.2.1. Phân trang đơn giản(12): Two-Level Paging Example

- A logical address (on 32-bit machine with 4K page size) is divided into:
  - a **page number** consisting of 20 bits.
  - a **page offset** consisting of 12 bits.

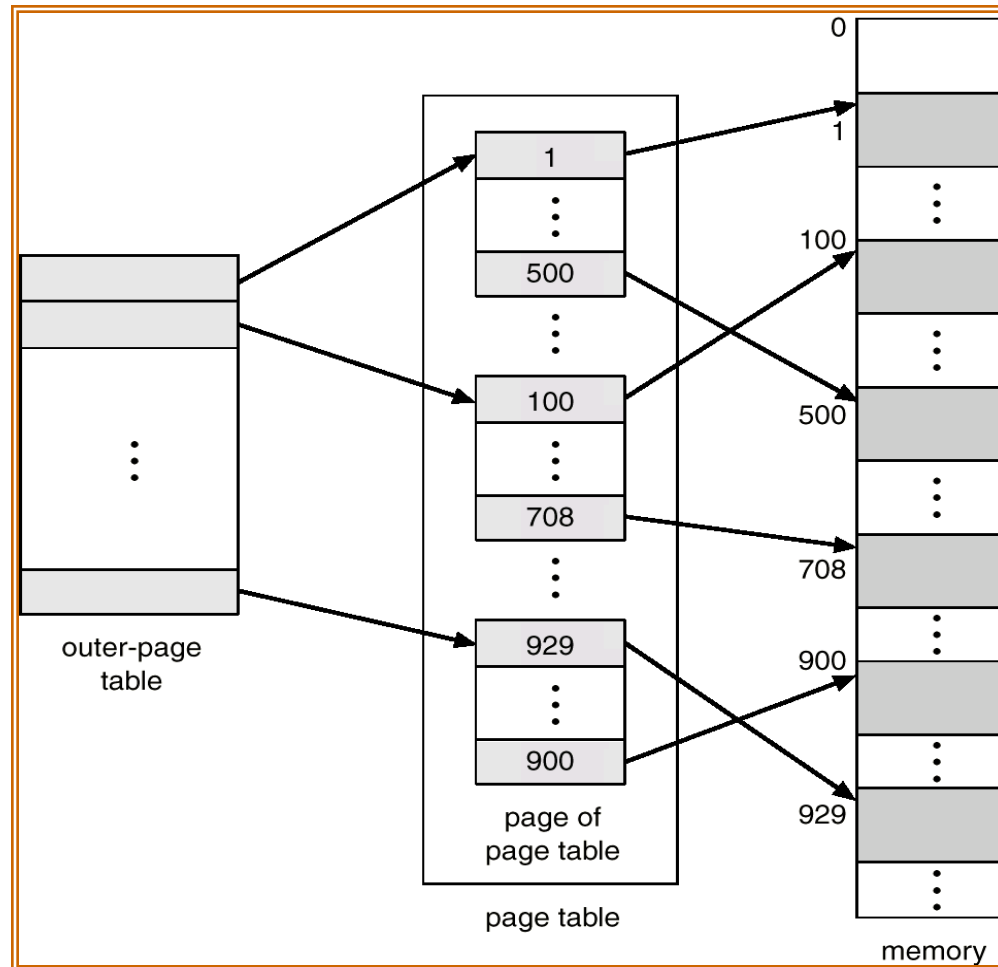
- Since the page table is paged, the page number is further divided into:

- a 10-bit page number.
- a 10-bit page offset.

page number		page offset
$p_1$	$p_2$	$d$
10	10	12

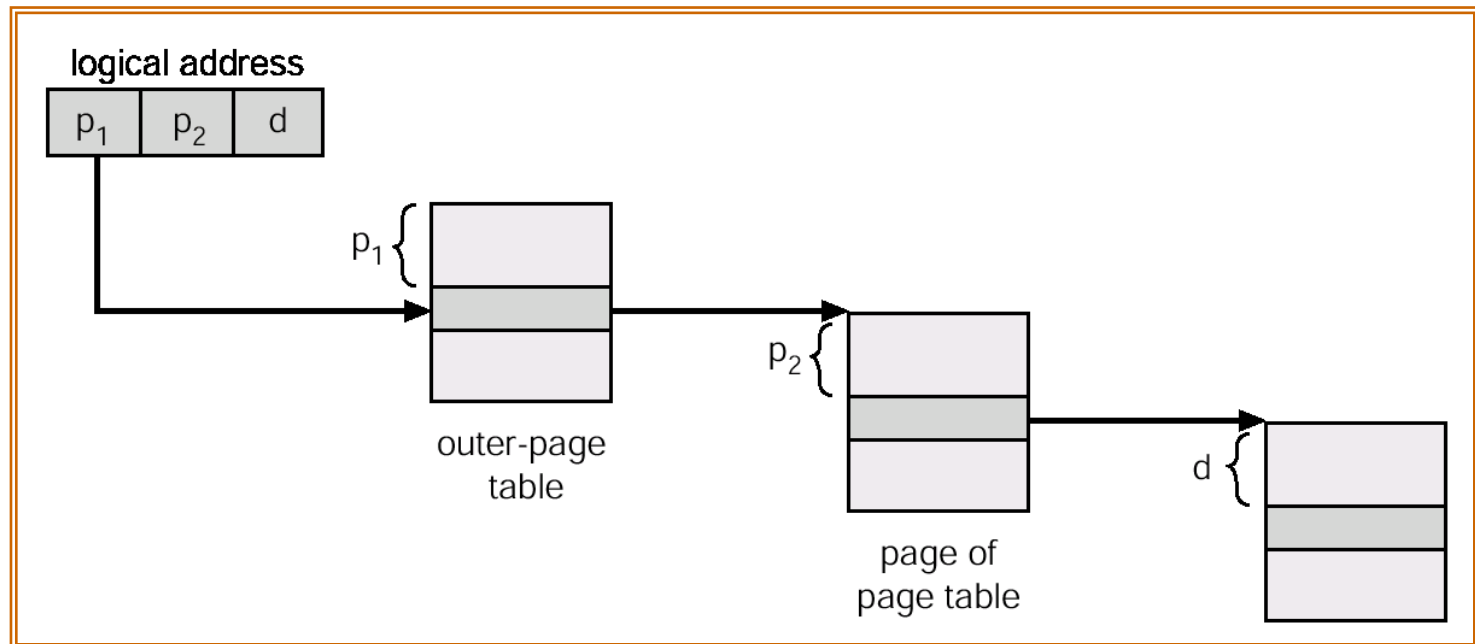
- Thus, a logical address is as follows:  
where  $p_1$  is an index into the outer page table, and  $p_2$  is the displacement(độ rời) within the page of the outer page table.

# 3.1.2.1. Phân trang đơn giản(13): Two-Level Page-Table Scheme



# 3.1.2.1. Phân trang đơn giản(14): Address-Translation Scheme

- Address-translation scheme for a two-level 32-bit paging architecture

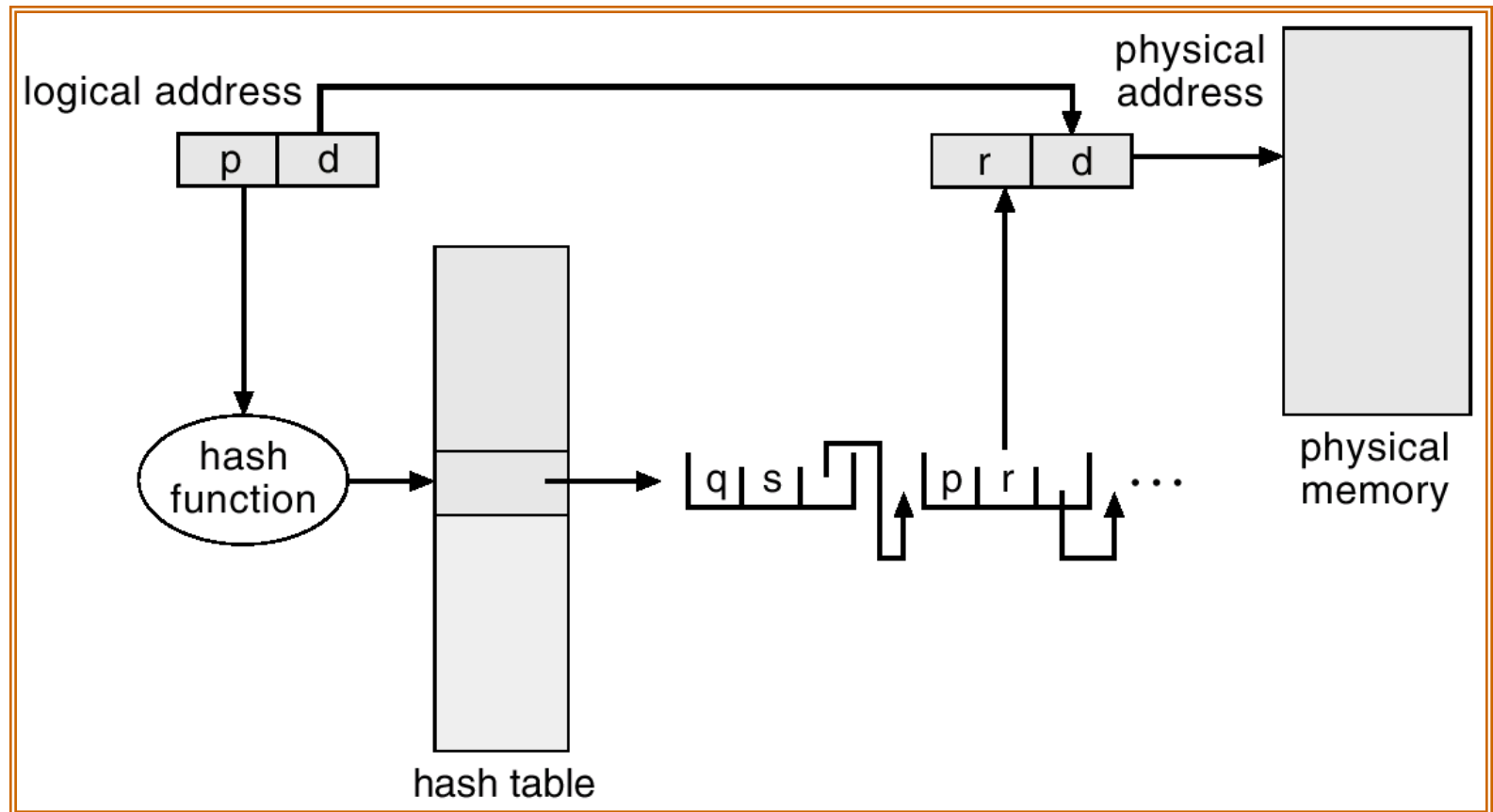


### 3.1.2.1. Phân trang đơn giản(15): Hashed Page Tables

---

- Common in address spaces  $> 32$  bits.
- The virtual page number is hashed into a page table. This page table contains a chain of elements hashing to the same location.
- Virtual page numbers are compared in this chain **searching** for a match. If a match is found, the corresponding physical **frame** is extracted.

# 3.1.2.1. Phân trang đơn giản(16): Hashed Page Table

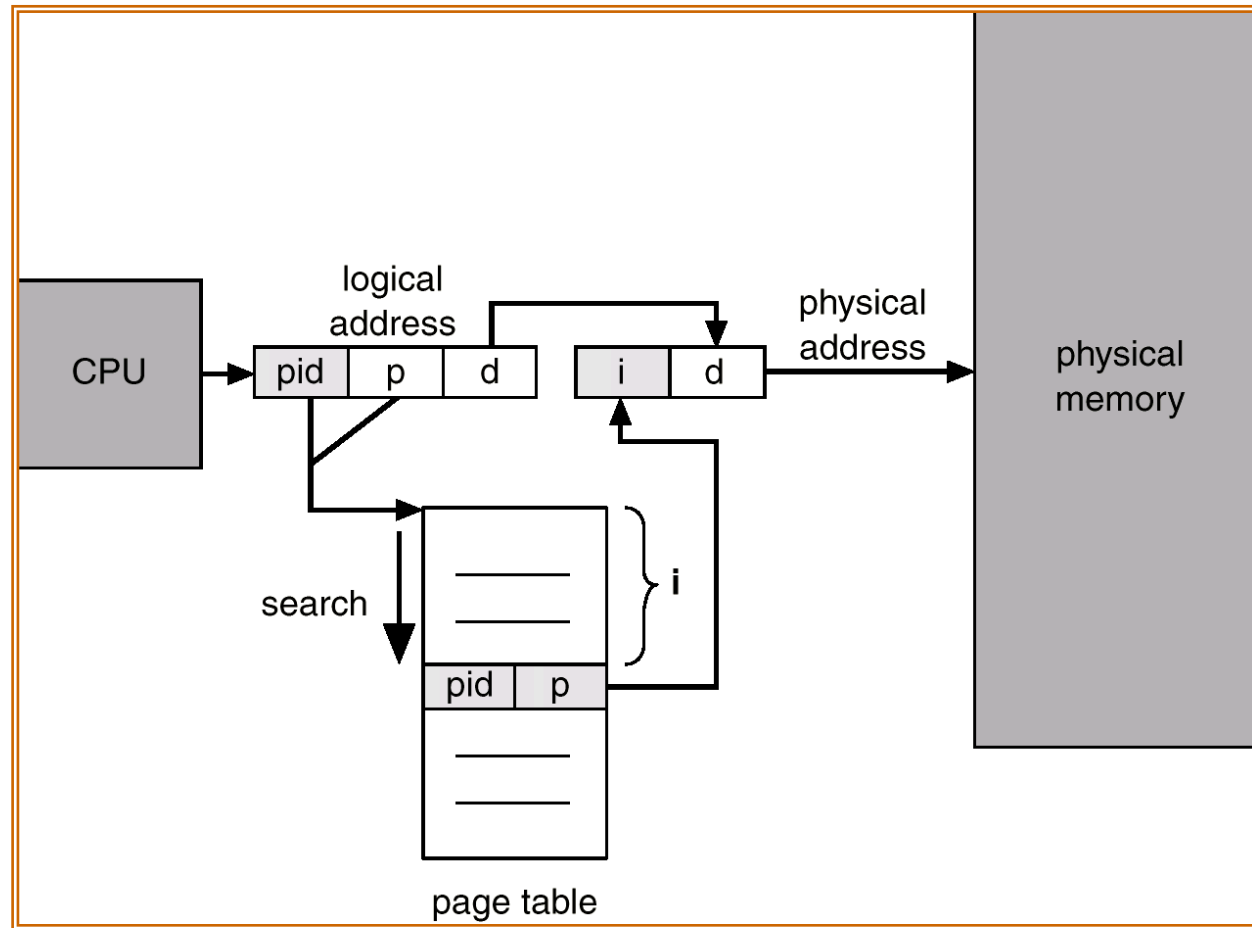


# 3.1.2.1. Phân trang đơn giản(17): Inverted Page Table

- One entry(điểm vào) for each real page of memory.
- **Entry** consists of the **virtual address** of the page stored in that real memory location, with **information about the process** that owns that page.
- **Decreases memory** needed to **store** each page table, but **increases time** needed to **search** the table when a page reference occurs.
- Use hash table to limit the search to one — or at most a few — page-table entries.



### 3.1.2.1. Phân trang đơn giản(17): Inverted Page Table Architecture



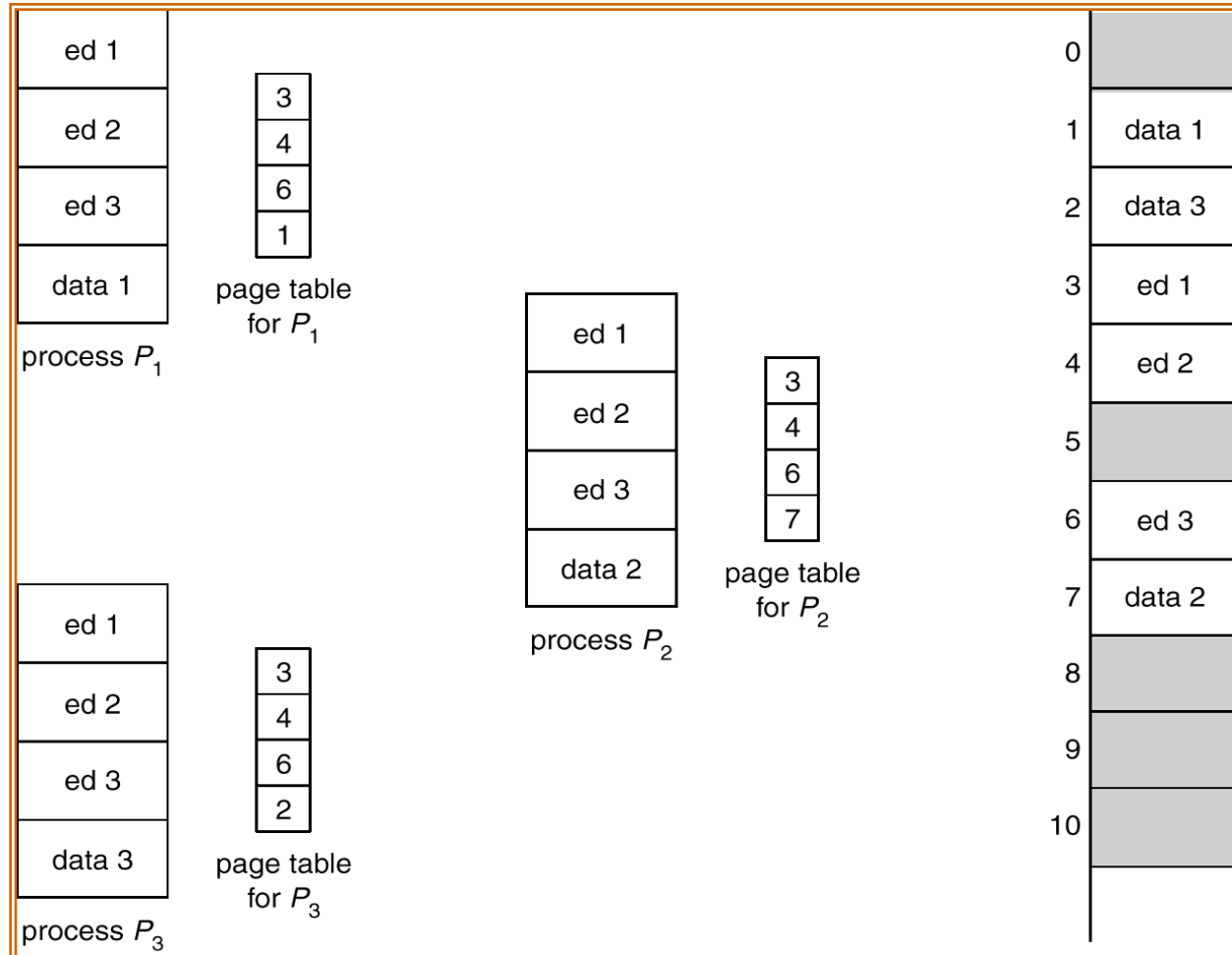
# 3.1.2.1. Phân trang đơn giản(18): Shared Pages



---

- Shared code
  - One copy of read-only (reentrant) code shared among processes (i.e., text editors, compilers, window systems).
  - **Shared code** must appear in **same location in the logical address space of all processes.**
- Private code and data
  - Each process keeps a separate(tách rời, riêng) copy of the code and data.
  - The pages for the private code and data can appear **anywhere in the logical address space.**

# 3.1.2.1. Phân trang đơn giản(19): Shared Pages Example





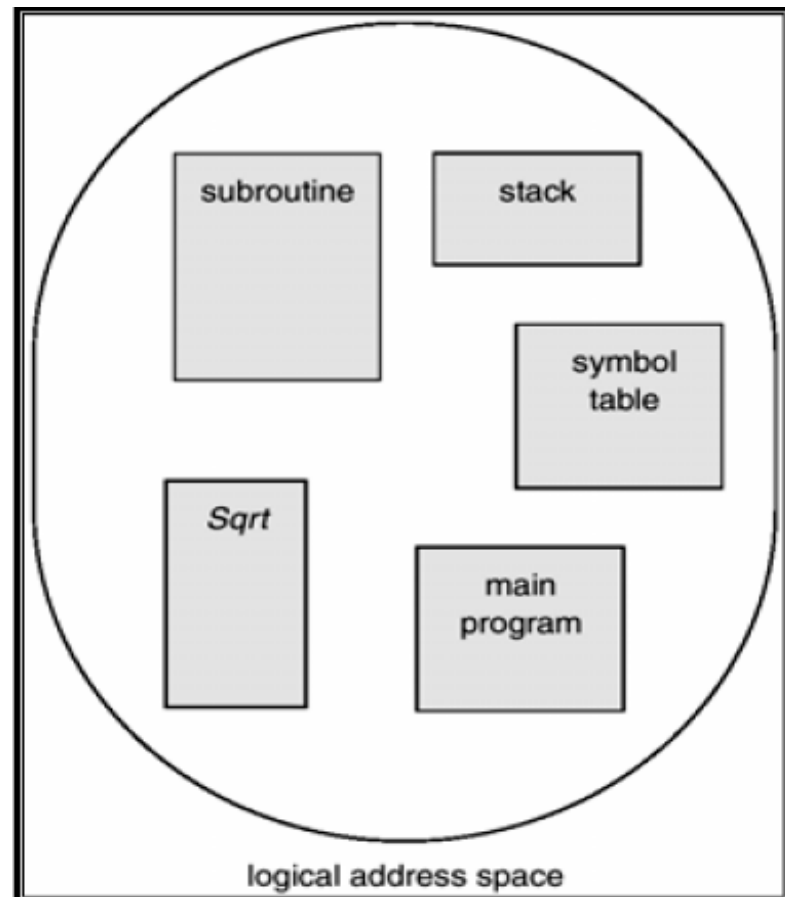
## 3.1.2.2. Phân đoạn đơn giản(1)

---

- Lược đồ quản lý bộ nhớ giúp người sử dụng ‘nhìn thấy’ bộ nhớ.
- Một chương trình là một tập hợp các đoạn. Mỗi đoạn là một đơn vị logic như là:
  - main program,
  - procedure,
  - function,
  - method,
  - object,
  - local variables, global variables,
  - common block,
  - stack

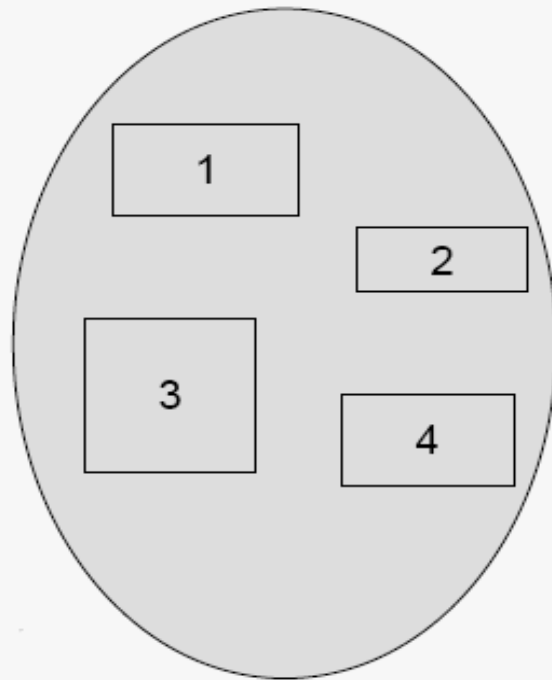
## 3.1.2.2. Phân đoạn đơn giản(2)

- Chương trình dưới góc nhìn của người sử dụng

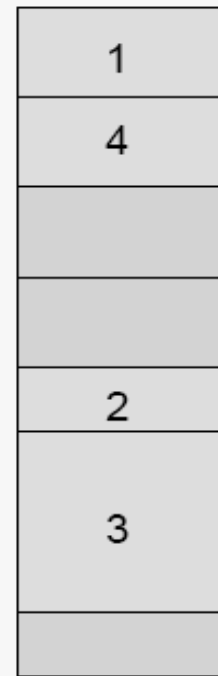


## 3.1.2.2. Phân đoạn đơn giản(3)

- Góc nhìn logic của sự phân đoạn



user space



physical memory space



## 3.1.2.2. Phân đoạn đơn giản(4)

---

- Kiến trúc phân đoạn
  - Địa chỉ logic gồm 2 thành phần:  
<**segment-number**, **offset**>,
  - *Segment table* – tương tự bảng phân trang, nội dung mỗi mục trong *Segment table* gồm có:
    - *Base* – chứa địa chỉ vật lý đầu tiên của đoạn trong bộ nhớ.
    - *Limit* – xác định độ dài của đoạn.
  - *Segment-table base register (STBR)*: trỏ tới vị trí của *Segment table*( bảng phân đoạn) trong bộ nhớ.
  - *Segment-table length register (STLR)*: xác định số đoạn mà một chương trình sử dụng;
  - Segment number  $s$  là hợp lệ nếu  $s < \text{STLR}$ .



## 3.1.2.2. Phân đoạn đơn giản(5)

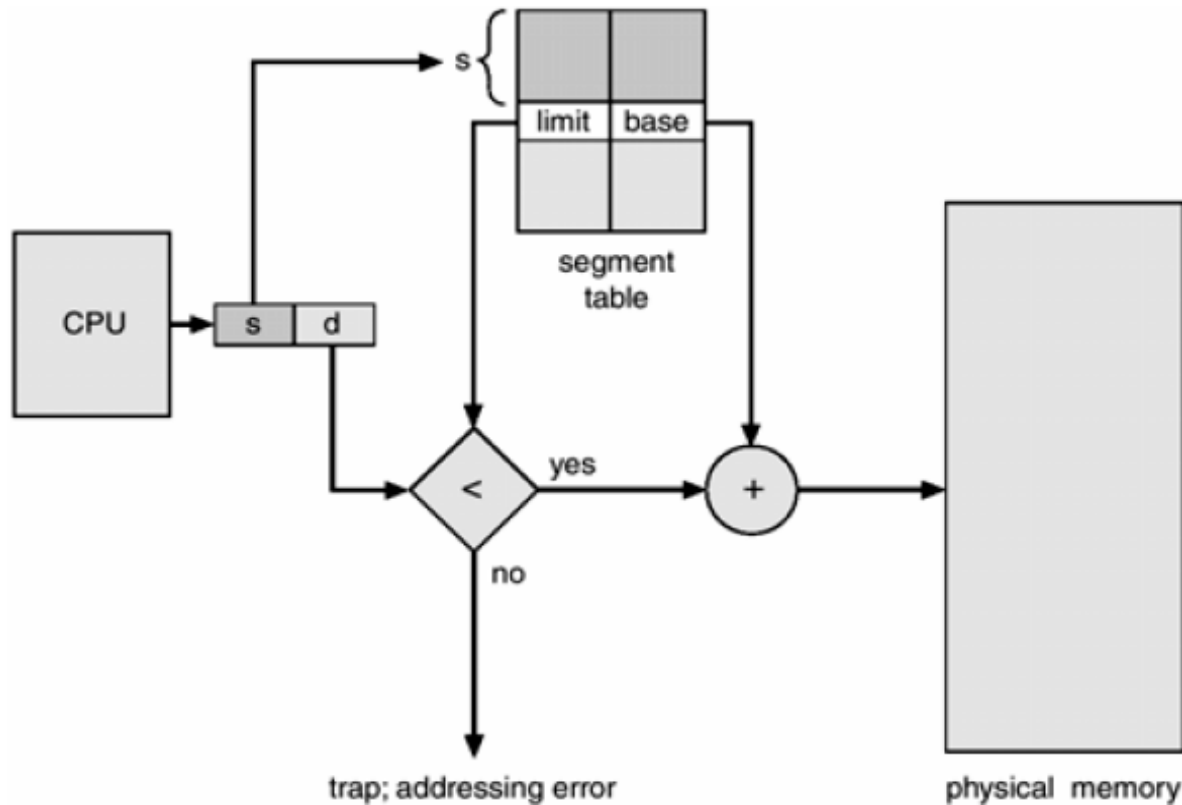
---

- Kiến trúc phân đoạn(tiếp)
  - Phân đoạn.
    - Các đoạn có kích thước khác nhau (khác với phân trang)
  - Định vị.
    - Động
    - Được thực hiện bởi bảng phân đoạn
  - Phân phối bộ nhớ.
    - Giải quyết bài toán phân phối bộ nhớ động
    - First fit/best fit/worst fit
    - Có sự phân mảnh ngoài



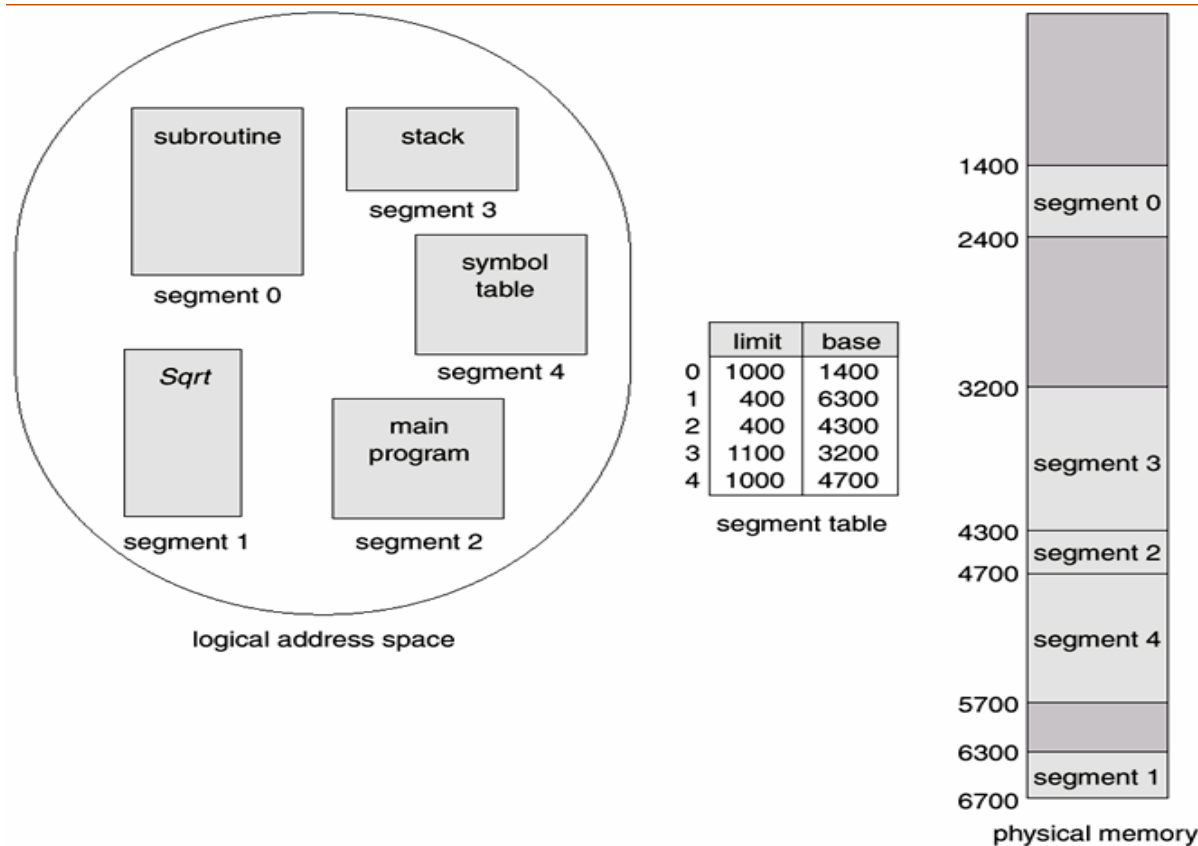
## 3.1.2.2. Phân đoạn đơn giản(6)

- Lược đồ phân đoạn

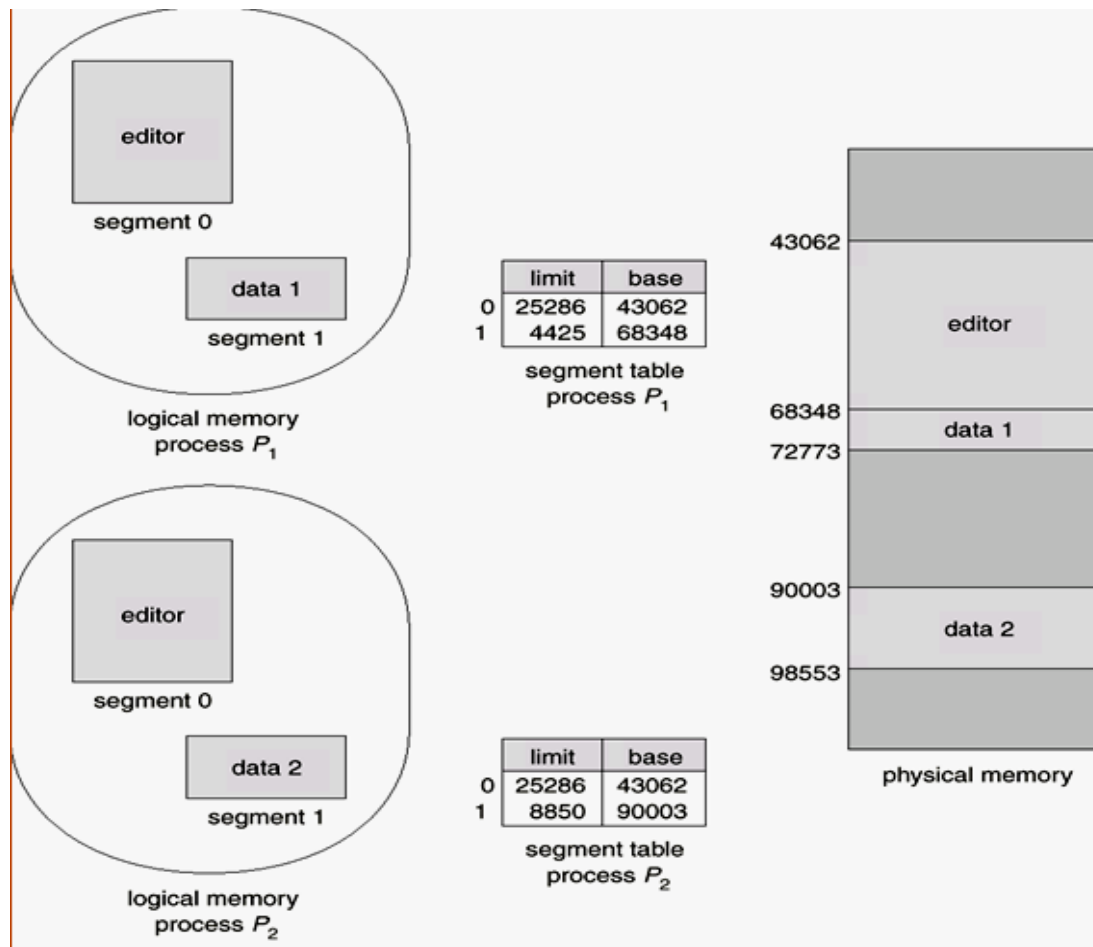


## 3.1.2.2. Phân đoạn đơn giản(7)

- Ví dụ:



## 3.1.2.2. Phân đoạn đơn giản(8): Sharing of Segments





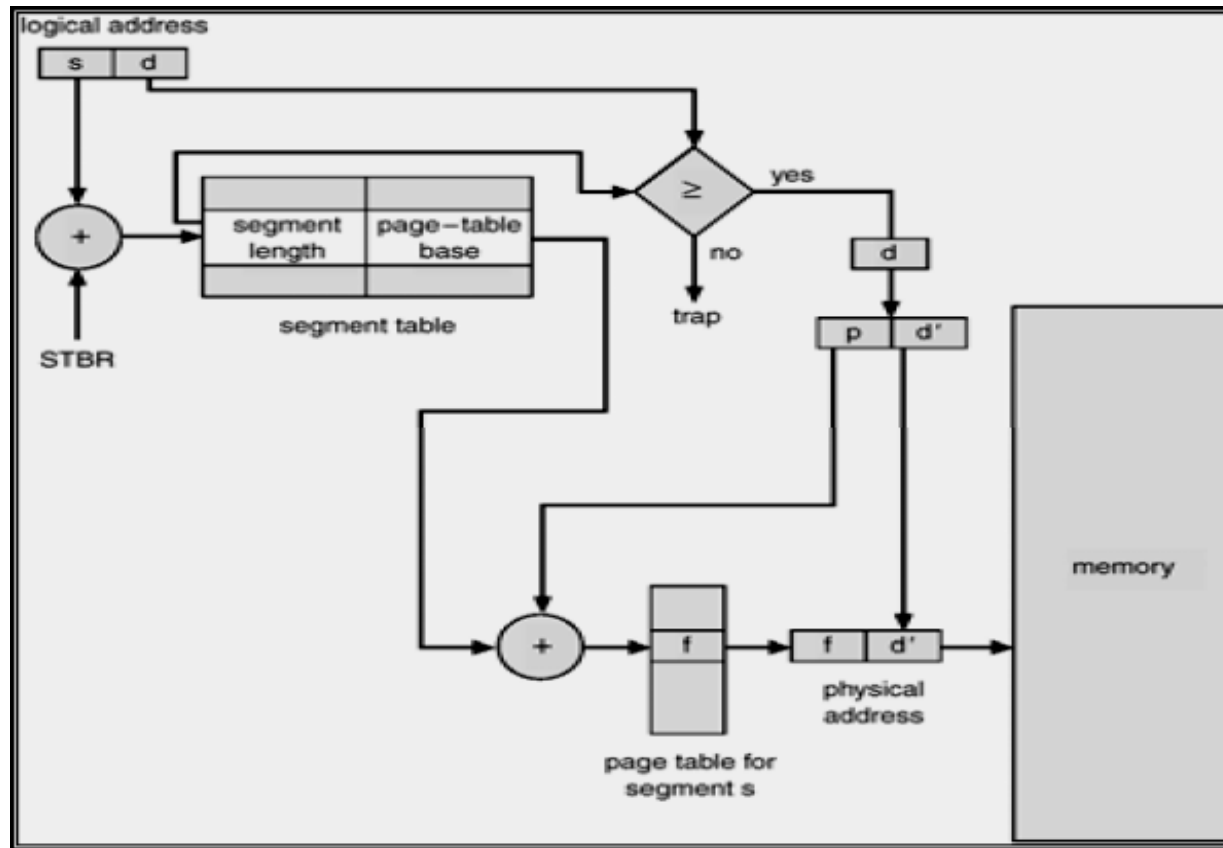
### 3.1.2.3. Kết hợp phân đoạn - phân trang – MULTICS(1)

---

- Bộ nhớ được phân thành các đoạn, sau đó mỗi đoạn lại được phân trang.
- Hệ thống **MULTICS** giải quyết được vấn đề phân mảnh ngoài và thời gian tìm kiếm dài.
- Giải pháp khác so với phân đoạn ở chỗ mỗi mục của bảng phân đoạn không chứa địa chỉ cơ sở của đoạn mà chứa địa chỉ cơ sở của bảng phân trang của đoạn đó.

### 3.1.2.3. Kết hợp phân đoạn - phân trang – MULTICS(2)

- Lược đồ MULTICS





## 3.2. Mô hình Bộ nhớ ảo

---

- Thông tin cơ bản - Background
- Phân trang theo yêu cầu - Demand Paging
- Thay trang - Page Replacement
- Phân phối các Frames - Allocation of Frames
- Thrashing
- Phân đoạn theo yêu cầu - Demand Segmentation



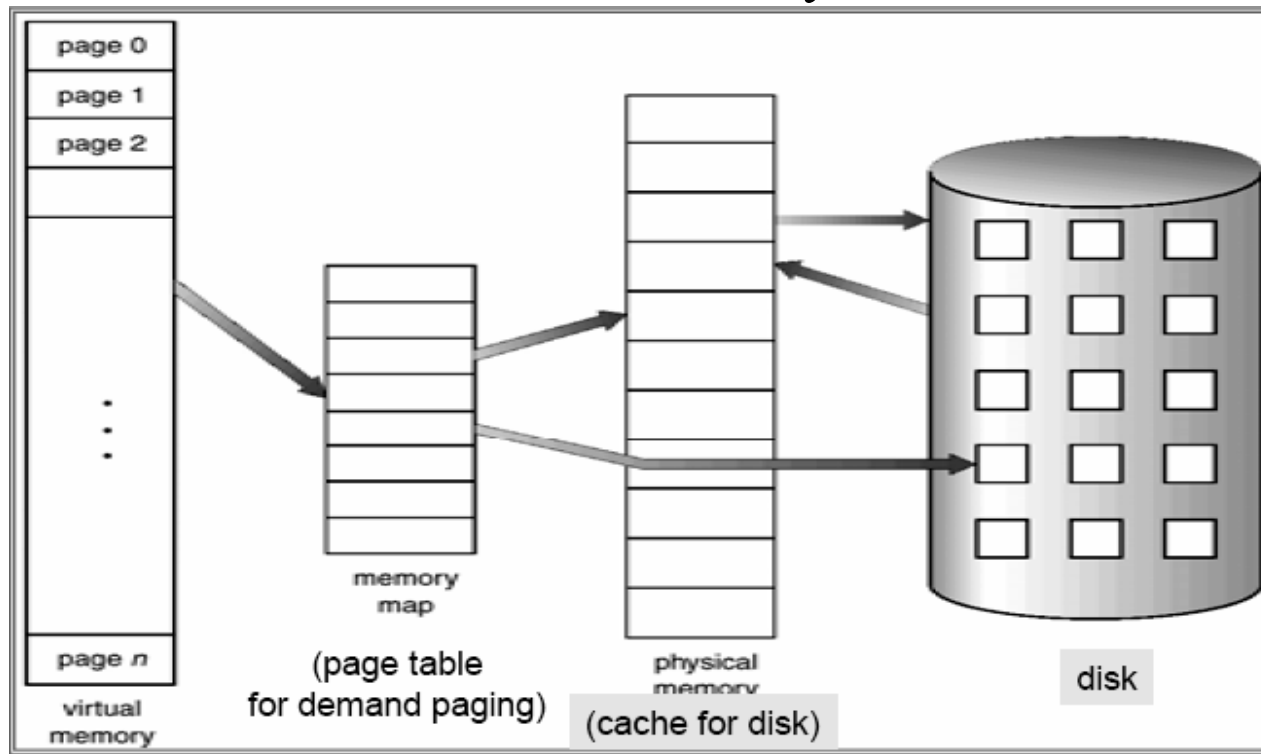
## 3.2.1. Hiểu biết cơ bản(1)

---

- **Virtual memory** – sự tách riêng của không gian địa chỉ logic ra khỏi bộ nhớ vật lý. Không gian địa chỉ logic gọi là bộ nhớ ảo
  - Kích thước bộ nhớ vật lý có hạn -> nó giới hạn kích thước chương trình
  - Thực tế, chỉ một phần của chương trình cần phải đưa vào bộ nhớ (vật lý) để thực hiện -> có thể chứa chương trình ở đâu? – VIRTUAL MEMORY
  - Do đó không gian địa chỉ logic có thể lớn hơn không gian địa chỉ vật lý rất nhiều -> cung cấp bộ nhớ rất lớn cho người lập trình
  - Cho phép một số tiến trình chia sẻ không gian địa chỉ.
  - Cho phép tạo tiến trình hiệu quả hơn.
- Bộ nhớ ảo có thể được thực hiện thông qua:
  - Demand paging (Windows, Linux...)(Phân trang theo yêu cầu)
  - Demand segmentation (IBM OS/2)(Phân đoạn theo yêu cầu)

## 3.2.1. Hiểu biết cơ bản(2)

- Bộ nhớ ảo lớn hơn bộ nhớ vật lý

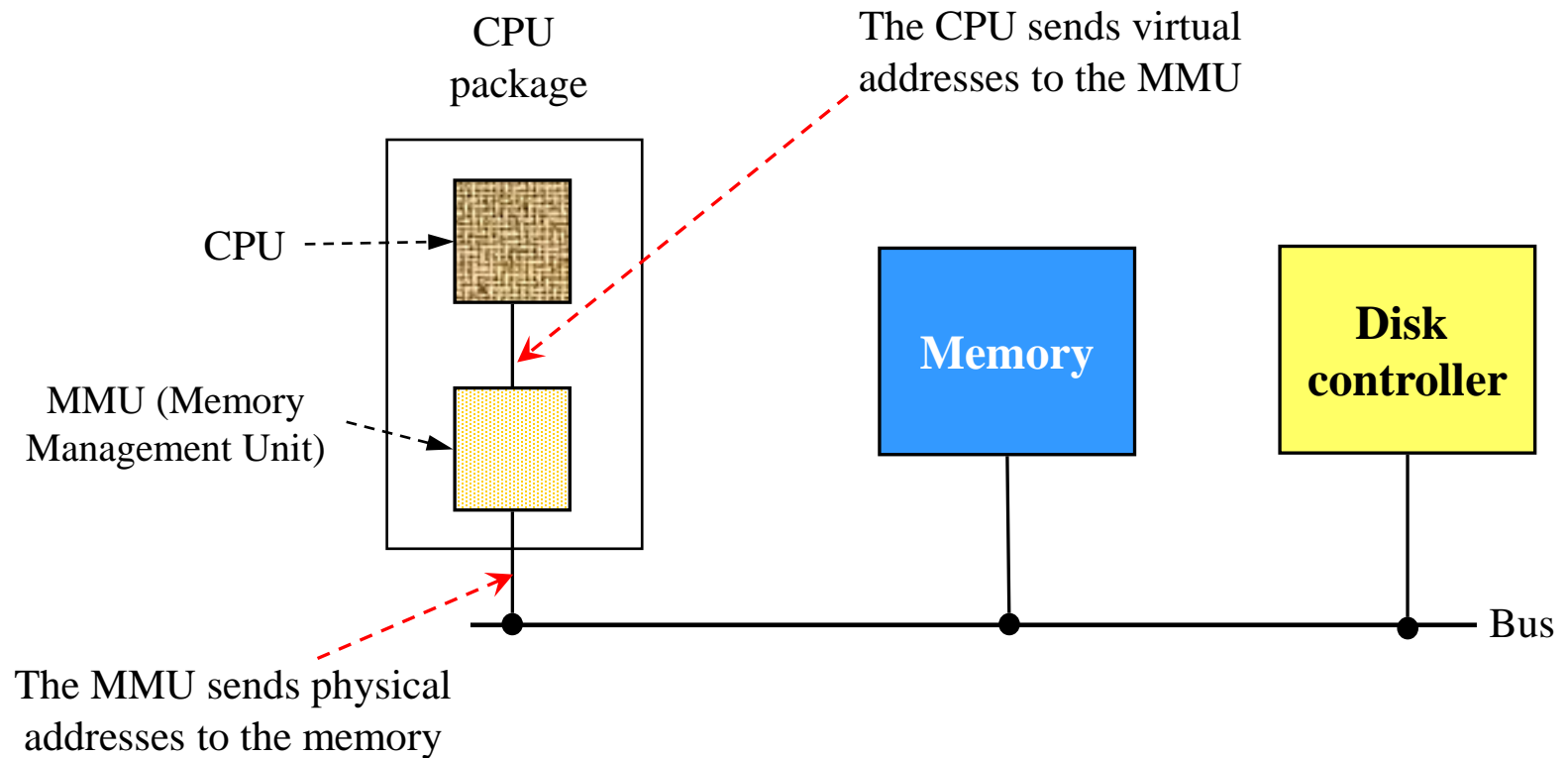


Vì vẫn cần bộ nhớ ngoài để lưu trữ ctr & dữ liệu khi chưa được đưa vào bộ nhớ trong

kích thước bộ nhớ ảo chỉ bị giới hạn bởi dung lượng ổ đĩa



## 3.2.2. Cơ chế bộ nhớ ảo





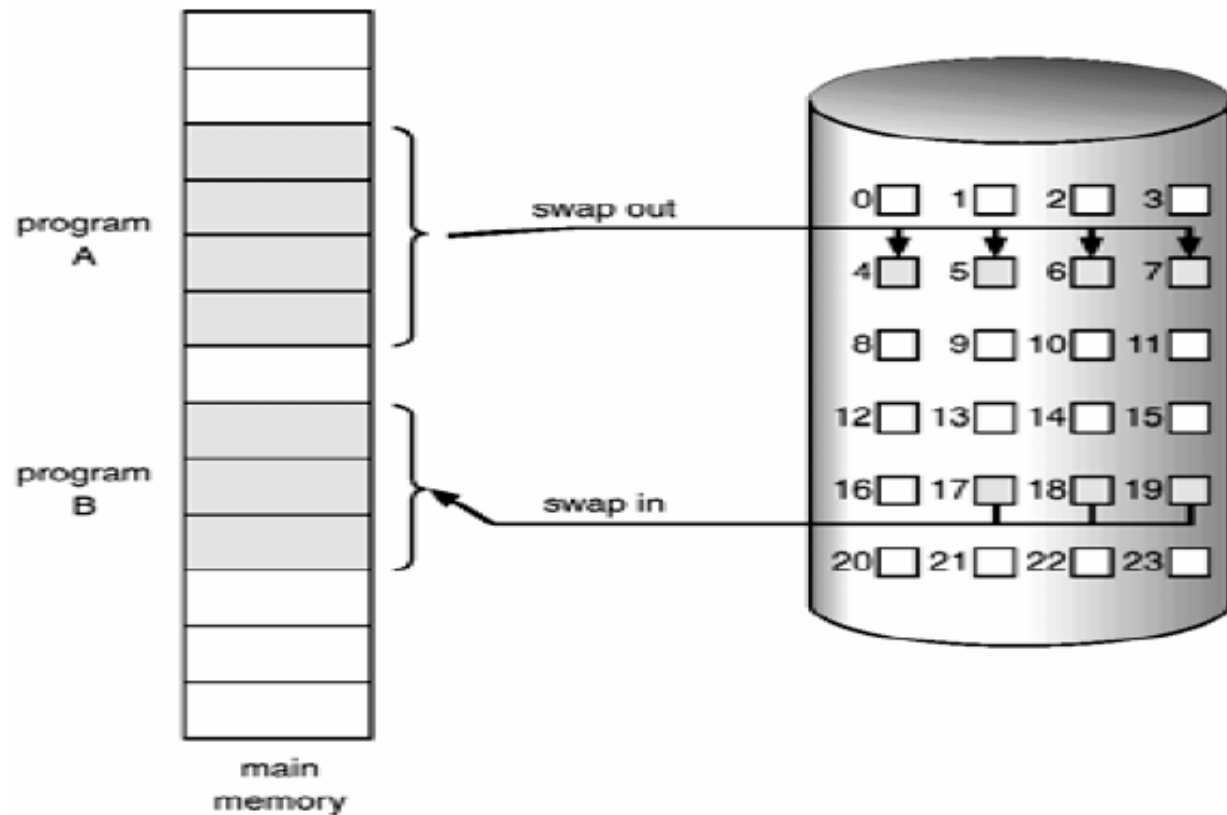
## 3.2.3. Phân trang theo yêu cầu(1)

---

- Đưa một trang vào bộ nhớ chỉ khi nó được cần đến.
  - Cần ít thao tác vào-ra hơn
  - Cần ít bộ nhớ hơn
  - Đáp ứng nhanh hơn: tiến trình bắt đầu ngay sau khi số trang tối thiểu được nạp vào bộ nhớ
  - Nhiều người sử dụng/tiến trình hơn: do mỗi tiến trình dùng ít bộ nhớ hơn
- Khi trang được cần đến (khi tiến trình tham chiếu đến nó)
  - Tham chiếu không hợp lệ -> hủy bỏ
  - Không trong bộ nhớ -> đưa vào bộ nhớ
  - Có trong bộ nhớ -> truy nhập

## 3.2.3. Phân trang theo yêu cầu(2)

- Chuyển một vùng nhớ phân trang tới không gian ổ đĩa



## 3.2.3. Phân trang theo yêu cầu(3)

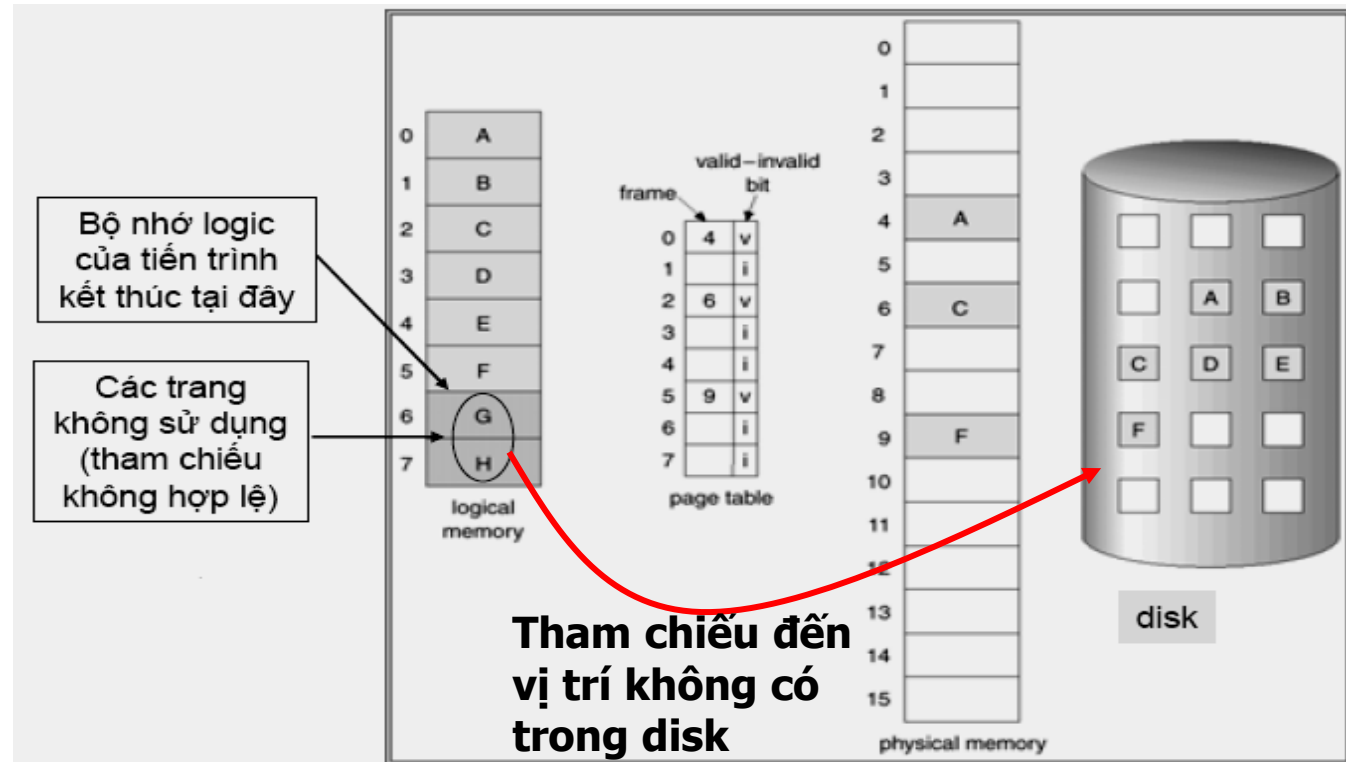
### ■ Valid-Invalid Bit

- Mỗi phần tử trong bảng phân trang được gắn thêm một valid– invalid bit (1 -> có trong bộ nhớ, 0 -> không trong bộ nhớ)
  - Ban đầu khởi tạo tất cả các v–inv bit bằng 0
  - Ví dụ bảng phân trang:
    - Các bước xử lý Page Fault Trong khi thông dịch địa chỉ, nếu v–inv bit bằng 0 -> page fault.
- ⇒ Page Fault có 2 khả năng:
- Tham chiếu(đến bộ nhớ ảo) không hợp lệ
  - Chưa được đưa vào bộ nhớ trong

Frame #	valid-invalid bit
	1
	1
	1
	1
	0
⋮	
	0
	0

## 3.2.3. Phân trang theo yêu cầu(4)

- Bảng phân trang khi một số trang không ở trong bộ nhớ chính





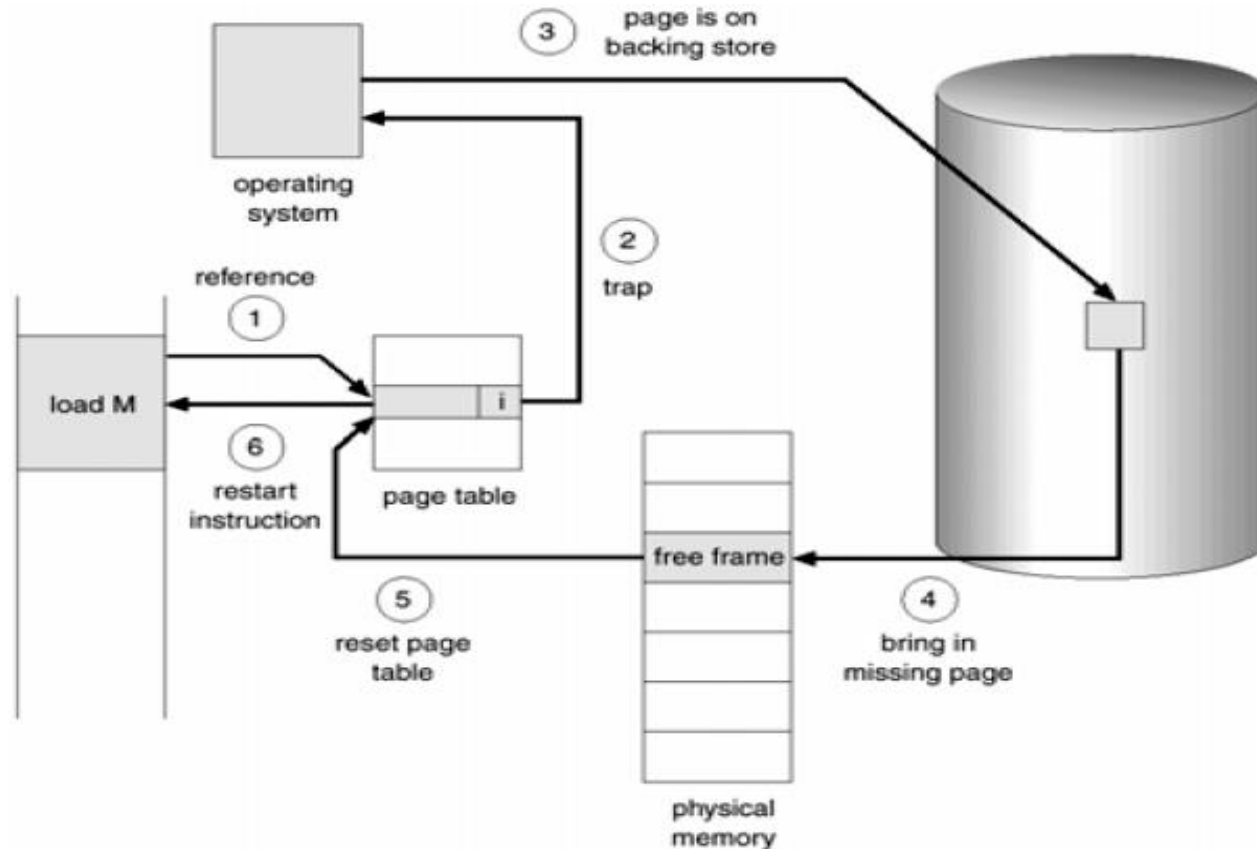
## 3.2.3. Phân trang theo yêu cầu(5)

---

- Các bước xử lý Page Fault
  - 1. Khi có tham chiếu tới trang, đầu tiên tham chiếu sẽ lập bấy tới HÐH -> phát hiện page fault
  - 2. HÐH tìm trong bảng khác để quyết định:
    - Tham chiếu không hợp lệ -> hủy bỏ.
    - Không có trong bộ nhớ -> đưa vào bộ nhớ.
  - 3. Nhận frame rồi
  - 4. Chuyển trang vào frame.
  - 5. Cập nhật lại bảng phân trang (thiết lập v-inv bit = 1), cập nhật danh sách frame rồi.
  - 6. Khởi động lại lệnh

## 3.2.3. Phân trang theo yêu cầu(6)

- Các bước xử lý Page Fault(tiếp)





## 3.2.3. Phân trang theo yêu cầu(7)

---

- Khi không có frame rồi?
  - Thay trang – tìm một số trang trong bộ nhớ nhưng đang không được sử dụng để đưa ra ngoài.
    - Giải thuật thay trang?
    - Hiệu năng? – muốn có một giải thuật giảm tối thiểu số lượng page faults.
  - Một trang có thể được đưa vào bộ nhớ nhiều lần.





### 3.2.3. Phân trang theo yêu cầu(8)

- Hiệu năng của phân trang theo yêu cầu
  - Tỷ lệ Page Fault -  $p : 0 \leq p \leq 1$   
 $p=0$  : không có page faults;  
 $p=1$  : mọi tham chiếu đều là fault.
  - Thời gian truy nhập hiệu quả-Effective Access Time (EAT)  
$$EAT = (1 - p) * ma + p * [thời\ gian\ xử\ lý\ page-fault]$$

Trong đó:

- ***ma***: *memory access* - thời gian truy nhập bộ nhớ
- **Thời gian xử lý page-fault**: gồm 3 vấn đề chính:
  - Phục vụ ngắt page-fault (1-100  $\mu$ s, có thể giảm bằng coding)
  - Đọc trang vào bộ nhớ ( $\approx 25$  ms)
  - Khởi động lại tiến trình (1-100  $\mu$ s, có thể giảm bằng coding)



### 3.2.3. Phân trang theo yêu cầu(9)

---

- Ví dụ về hiệu năng
  - Thời gian xử lý page-fault: 25 ms
  - Thời gian truy nhập bộ nhớ (ma): 100 ns
  - $EAT = (1-p) \times 100 + p \times 25,000,000 \text{ ns}$   
 $= 100 + 24,999,990 \times p \text{ ns}$
  - EAT tỷ lệ trực tiếp với tỷ lệ page-fault, nếu p càng lớn thì EAT càng lớn → máy càng chậm.

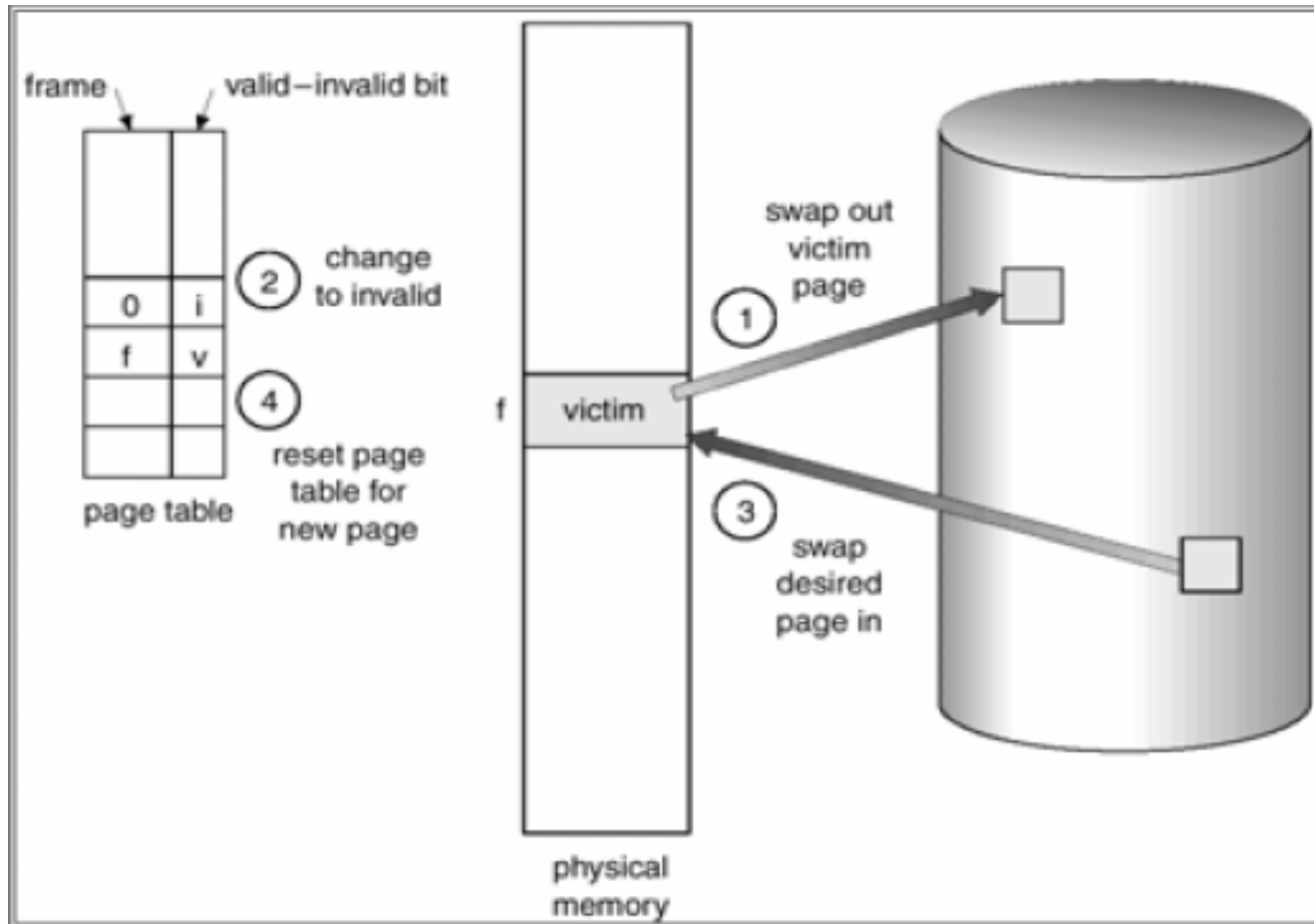


## 3.2.4. Thay trang(1)

---

- Các bước thay trang:
  - 1. Tìm vị trí của trang được yêu cầu trên đĩa.
  - 2. Tìm một frame rồi:
    - Nếu có frame rồi thì sử dụng nó.
    - Nếu không có, sử dụng một giải thuật thay trang để giải phóng một frame *nạn nhân(victim)*.
  - 3. Đọc trang được yêu cầu vào frame rồi. Cập nhật bảng phân trang và bảng quản lý frame rồi.
  - 4. Khởi động lại tiến trình.

## 3.2.4. Thay trang(2):





## 3.2.4. Thay trang(3): Các giải thuật thay trang

---

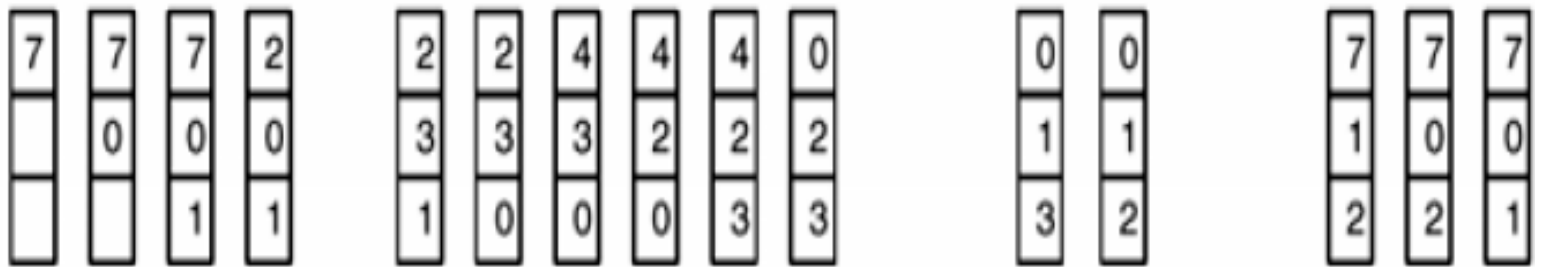
- Mục đích: tỷ lệ page-fault thấp nhất.
- Đánh giá giải thuật bằng cách chạy nó trên một chuỗi riêng biệt các **tham chiếu bộ nhớ** và tính số page faults trên chuỗi đó.
- Một số giải thuật:
  - FIFO(First In First Out)
  - LRU(Least Recently Used)
  - Giải thuật tương tự LRU

## 3.2.4. Thay trang(5): Giải thuật FIFO

- Ý tưởng: trang nào được đưa vào bộ nhớ sớm nhất sẽ được giải phóng để nạp trang khác
- Ví dụ: chuỗi tham chiếu các trang được nạp vào bộ nhớ có 3 frame

reference string

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



page frames



## 3.2.4. Thay trang(6): Least Recently Used (LRU) Algorithm

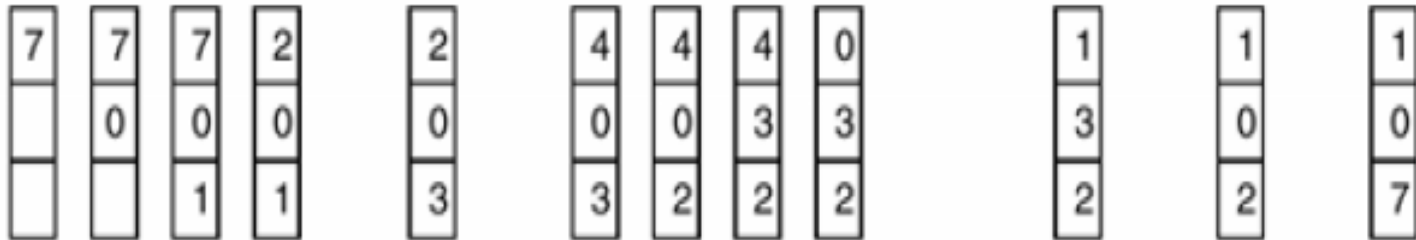
---

- Ý tưởng: Thay trang có khoảng thời gian không dùng lâu nhất
- Sự thực hiện của Bộ đếm (Counter)
  - Mọi phần tử bảng có một bộ đếm, mọi thời điểm **trang được tham chiếu** qua phần tử bảng này, copy clock vào trong bộ đếm.
  - Khi trang cần được hoán đổi, tìm trong bộ đếm để xác định trang nào làm nạn nhân.

## 3.2.4. Thay trang(7): LRU Page Replacement

reference string

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

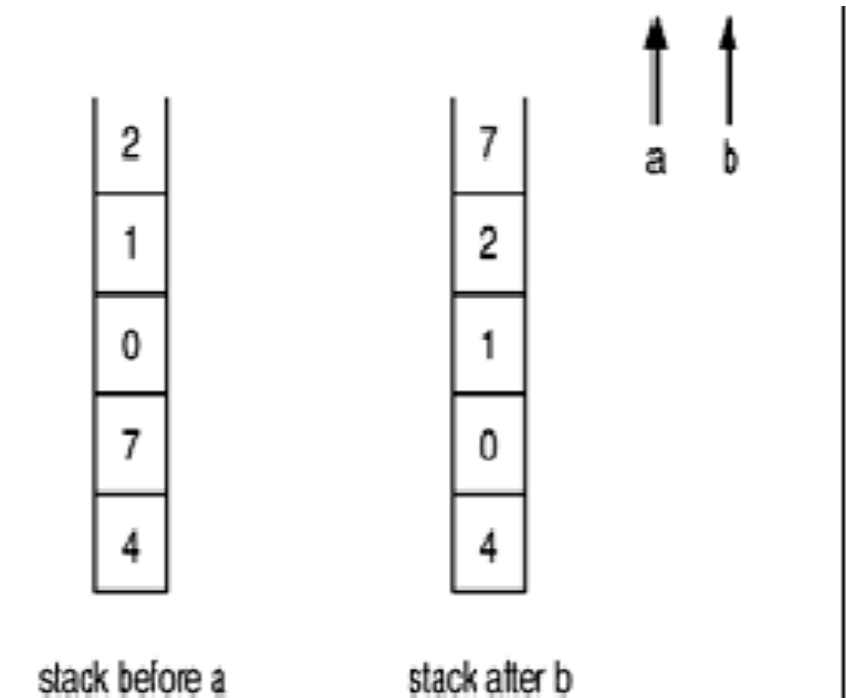


page frames



## 3.2.4. Thay trang(8): LRU Page Replacement

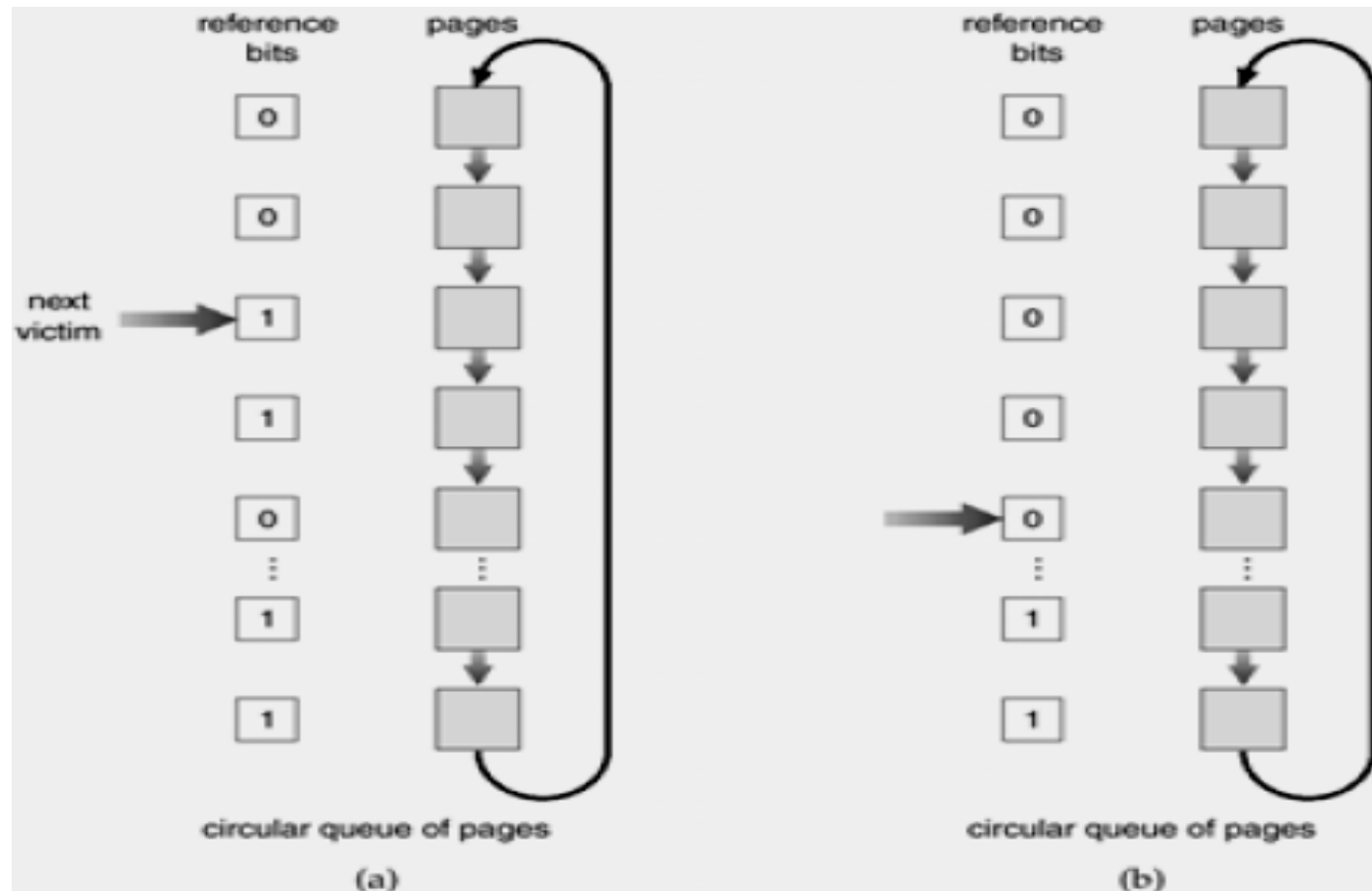
- Sự thực hiện của Stack
  - Dành ra một stack của các số trang trong một danh sách liên kết kép
  - Khi trang được tham chiếu:
    - Chuyển nó lên đỉnh nếu đã có trong bộ nhớ; nếu chưa, chuyển lên đỉnh và loại bỏ trang ở đáy stack
    - Cần 6 con trỏ để đổi trang
  - Không cần tìm kiếm để thay thế

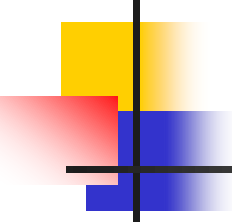


## 3.2.4. Thay trang(9): Các giải thuật tương tự LRU

- Giải thuật dùng thêm bit tham chiếu
  - Gắn một bit vào mỗi trang, khởi tạo = 0
  - Khi trang được tham chiếu, bit đó được thiết lập = 1.
  - Thay trang có bit tham chiếu = 0, nếu có tồn tại trong bộ nhớ
- Giải thuật cơ hội thứ hai (Second chance)
  - Cần có bit tham chiếu. Nếu bit tham chiếu = 0 thì thay trang. Trả lại cho trang đó cơ hội thứ hai và chuyển đến trang tiếp sau
    - Thiết lập bit tham chiếu = 0.
    - Để trang lại trong bộ nhớ.
    - Thay trang kế tiếp (theo FIFO) với luật tương tự.
  - Một cách thực hiện giải thuật là sử dụng queue vòng tròn.

### 3.2.4. Thay trang(12): Giải thuật thay trang "cơ hội thứ hai"





## 3.2.4. Thay trang(13): Các giải thuật dựa trên số liệu thống kê

---

- Dành ra một bộ đếm số tham chiếu đến mỗi trang.
- Least Frequently Used (LFU) Algorithm: thay trang đếm được ít nhất (có tần số truy nhập nhỏ nhất).
- Most Frequently Used (MFU) Algorithm: thay trang đếm được nhiều nhất (có tần số truy nhập cao nhất), dựa trên lý luận rằng trang đếm được ít nhất là có thể vừa được đưa vào bộ nhớ và chưa kịp được sử dụng.



## 3.2.5. Phân phối Frames(1)

---

- Mỗi tiến trình cần số lượng trang tối thiểu để thực hiện.
- Ví dụ: IBM 370 – cần 6 trang để thực hiện lệnh SS MOVE:
  - lệnh độ dài 6 bytes, có thể chứa trong 2 trang.
  - 2 trang để thực hiện **from**.
  - 2 trang để thực hiện **to**.
- Hai cách phân phối chính:
  - phân phối cố định (fixed allocation)
  - phân phối có ưu tiên (priority allocation)

## 3.2.5. Phân phối Frames(2): Phân phối cố định

- Phân phối công bằng – vd nếu có 100 frames và 5 tiến trình, cho mỗi tiến trình 20 trang.
- Phân phối theo kích thước của tiến trình

$s_i$  = size of process  $p_i$

$m = 64$

$S = \sum s_i$

$s_1 = 10$

$m$  = total number of frames

$s_2 = 127$

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

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

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

## 3.2.5. Phân phối Frames(3): Phân phối có ưu tiên

---

- Phân phối theo mức ưu tiên.
  - Nếu tiến trình  $P_i$  phát sinh một page fault, chọn thay thế một trong số các frame của nó.
  - Frame thay vào đó được chọn từ một tiến trình có mức ưu tiên thấp hơn.



## 3.2.5. Phân phối Frames(4): Global vs. Local Allocation

---

- **Global** replacement – tiến trình chọn một frame thay thế từ tập tất cả các frame; một tiến trình có thể lấy một frame từ một tiến trình khác.
- **Local** replacement – mỗi tiến trình chỉ chọn một frame thay thế từ chính tập các frame đã phân phối cho nó.





## 3.2.6. Thrashing (trì trệ)

---

- Nếu một tiến trình không có frame, tỷ lệ page fault là rất cao. Điều này dẫn đến:
  - Sử dụng CPU thấp.
  - HĐH cho rằng nó cần phải tăng mức đa chương trình.
  - Một tiến trình khác được thêm vào hệ thống.
- **Thrashing**  $\equiv$  một tiến trình được gọi là Thrashing nếu nó dành nhiều thời gian (bận rộn) với việc hoán đổi các trang vào và ra hơn là thời gian thực hiện.
- Tại sao thrashing xuất hiện?
  - $\Sigma$  kích thước các vùng  $>$  tổng kích thước bộ nhớ

## 3.2.7. Demand Segmentation- Phân đoạn theo yêu cầu

- Được sử dụng khi thiếu phần cứng thực hiện demand paging.
- OS/2 phân phối bộ nhớ theo đoạn và nó luôn theo dõi qua các trình quản lý đoạn (segment descriptors)
- Segment descriptor chứa một valid bit để cho biết hiện tại đoạn có ở trong bộ nhớ hay không.
  - Nếu đoạn ở trong bộ nhớ -> tiếp tục truy nhập,
  - Nếu đoạn không trong bộ nhớ -> segment fault.



# Q & A

---

- List câu hỏi