ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Hệ Điều Hành

(Nguyên lý các hệ điều hành)

Người trình bày:
Bộ môn Khoa Học Máy Tính
Viện Công Nghệ Thông Tin và Truyền Thông

Chương 3 Quản lý bộ nhớ

- Mục đích của hệ thống máy tính: thực hiện chương trình
  - Chương trình và dữ liệu (toàn bộ hoặc một phần) phải nằm trong bộ nhớ chính trong khi thực hiện
  - **Byte tích cực**:Những byte nội dung đang được thực hiện tại thời điểm quan sát
  - Phần chương trình chưa đưa vào bộ nhớ chính được lưu trên bộ nhớ thứ cấp (VD: đĩa cứng)⇒ *Bộ nhớ ảo*
    - Cho phép lập trình viên không lo lắng về giới hạn bộ nhớ vật lý

Chương 3 Quản lý bộ nhớ

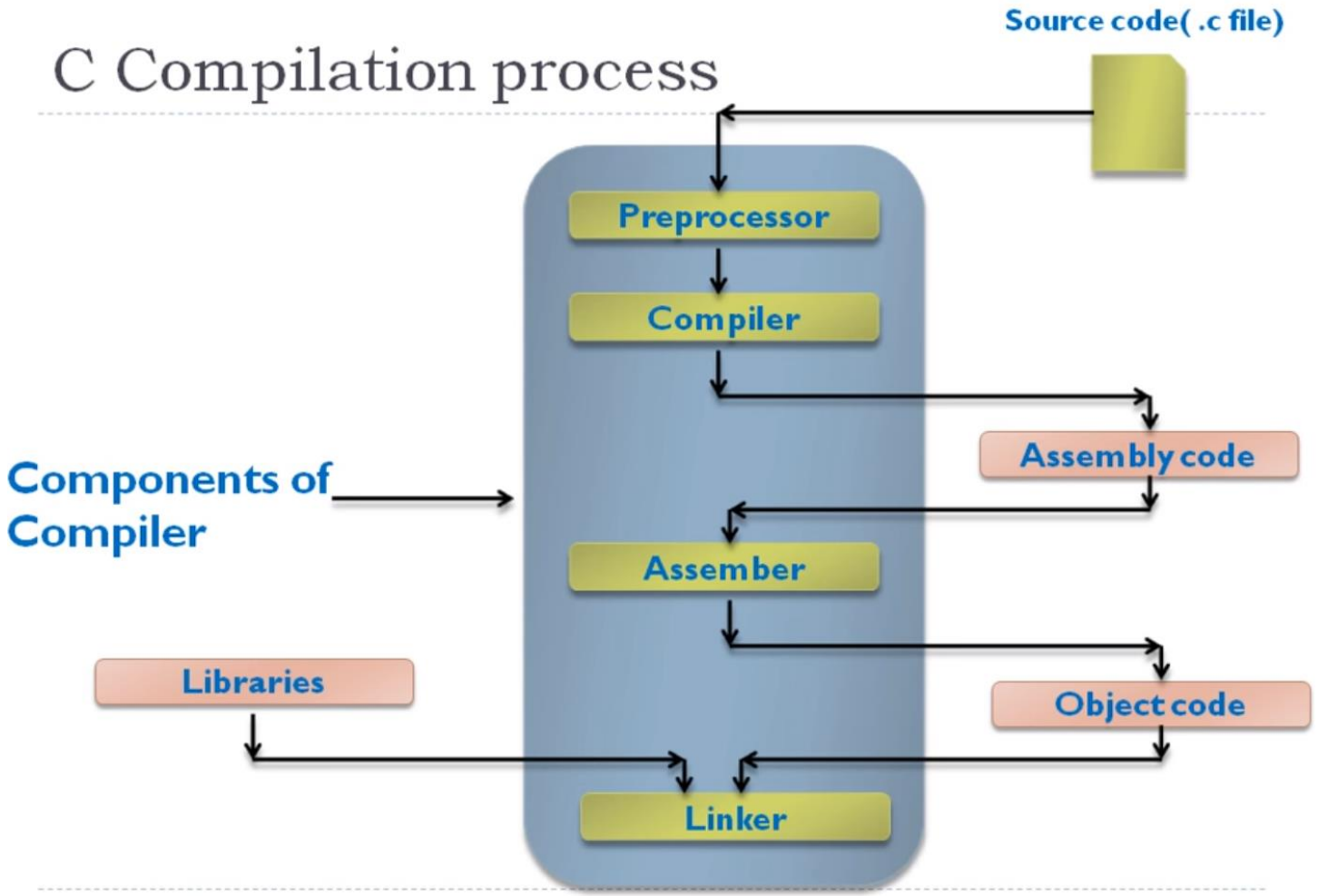
- Để s/d CPU hiệu quả và tăng tốc độ đáp ứng của hệ thống:
  - Cần luân chuyển CPU thường xuyên giữa các tiến trình
  - *Điều phối CPU (Phần 3- Chương 2)*
  - Cần nhiều tiến trình sẵn sàng trong bộ nhớ
    - Hệ số song song của hệ thống: Số tiến trình đồng thời tồn tại trong hệ thống
- Tồn tại nhiều chiến lược quản lý bộ nhớ khác nhau
  - Nhiều chiến lược đòi hỏi trợ giúp từ phần cứng
  - Thiết kế phần cứng có thể được tích hợp chặt chẽ với HDH

Chương 3 Quản lý bộ nhớ

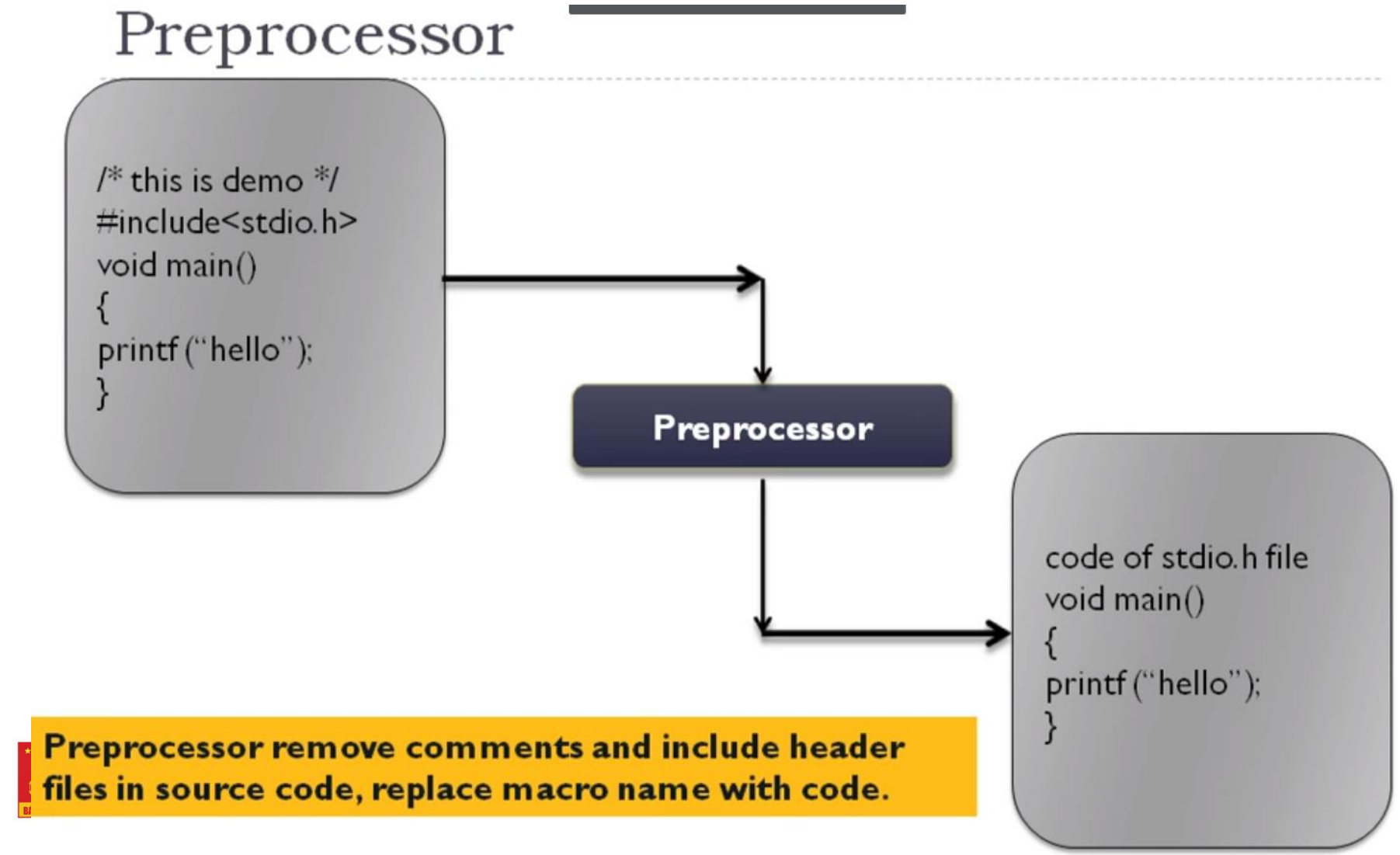
① Tổng quan
② Các chiến lược quản lý bộ nhớ
③ Bộ nhớ ảo
④ Quản lý bộ nhớ trong VXL họ Intel

- Ví dụ
- Bộ nhớ và chương trình
- Liên kết địa chỉ
- Các cấu trúc chương trình

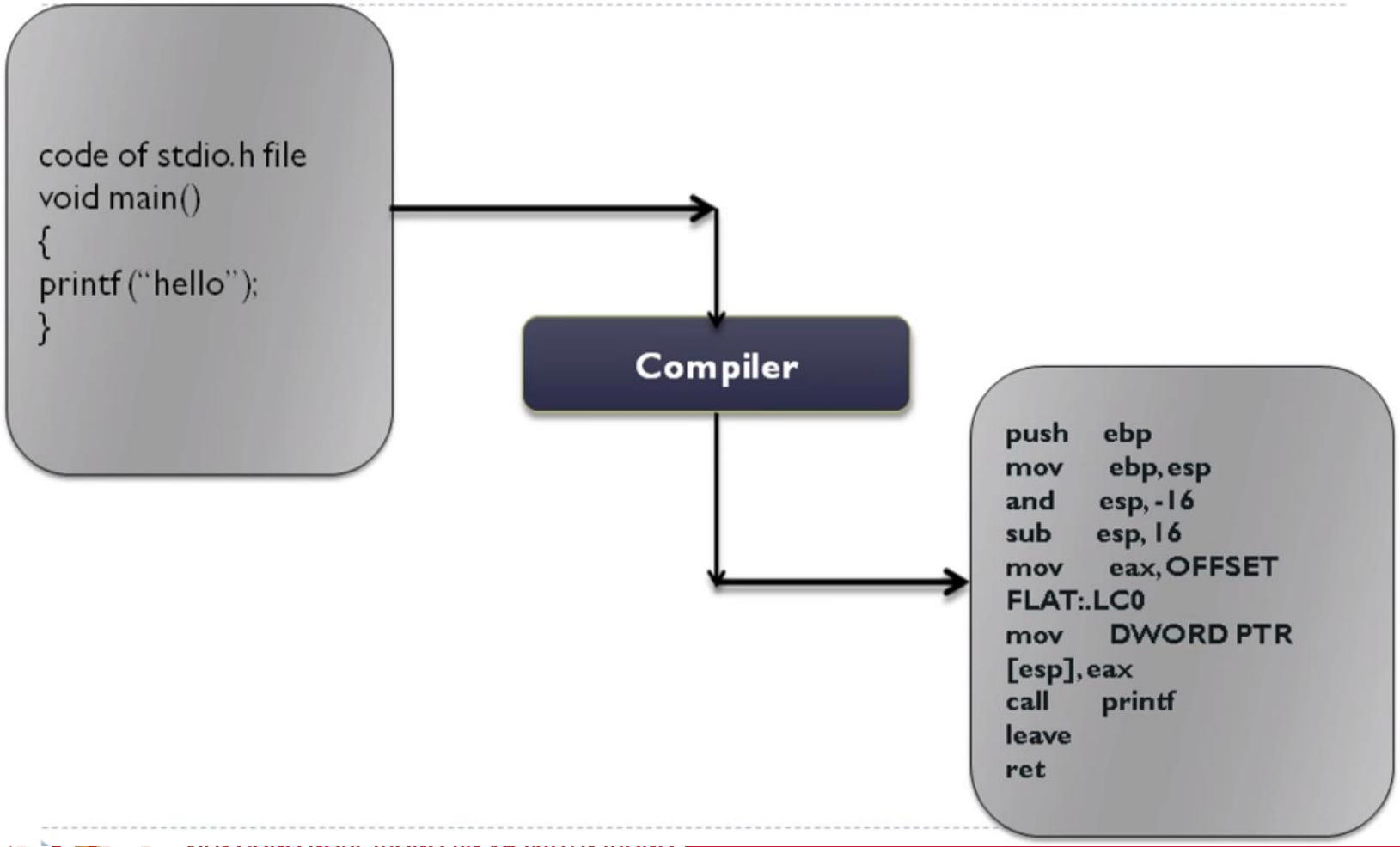
Quá trình biên dịch file C



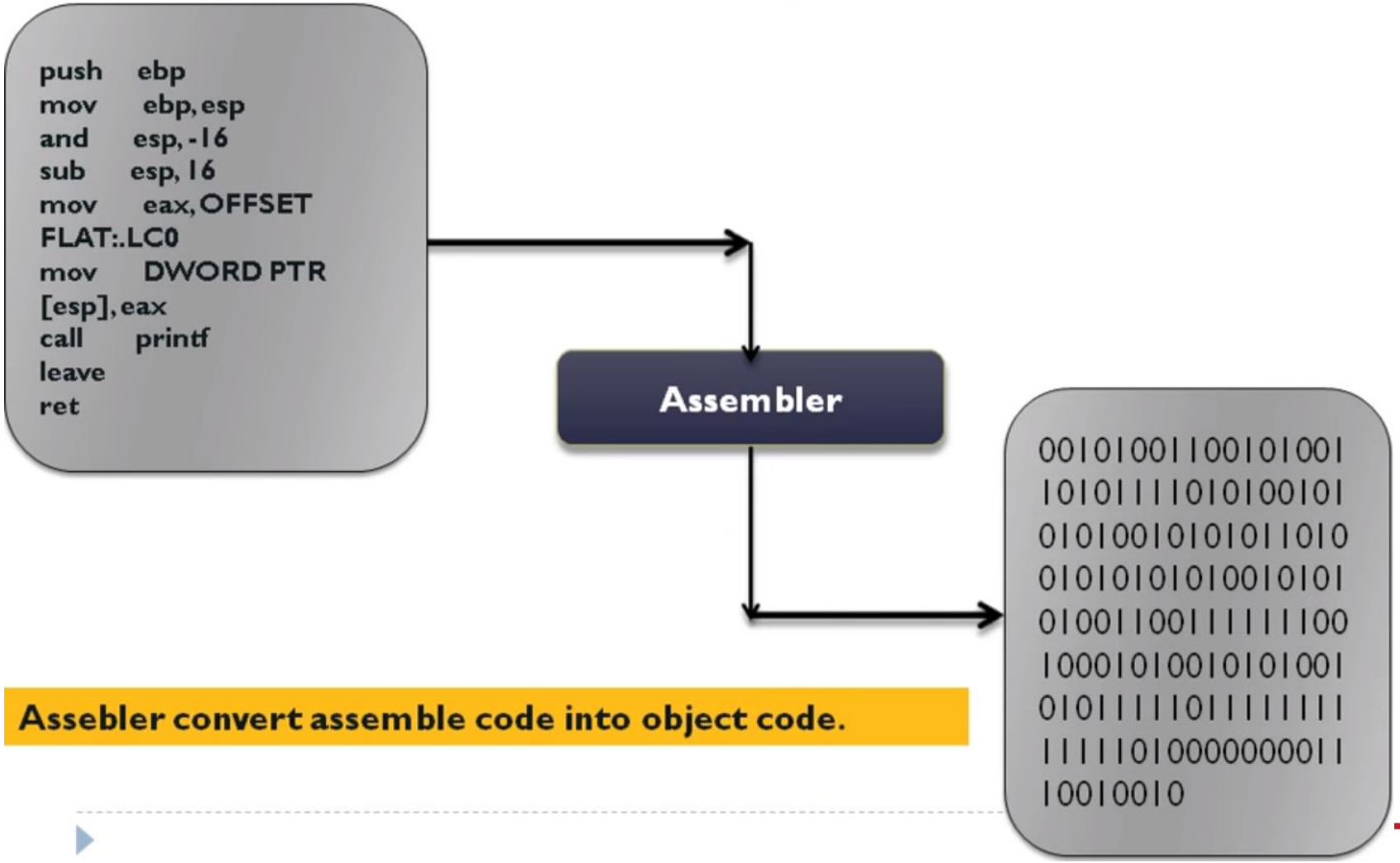
Quá trình biên dịch file C



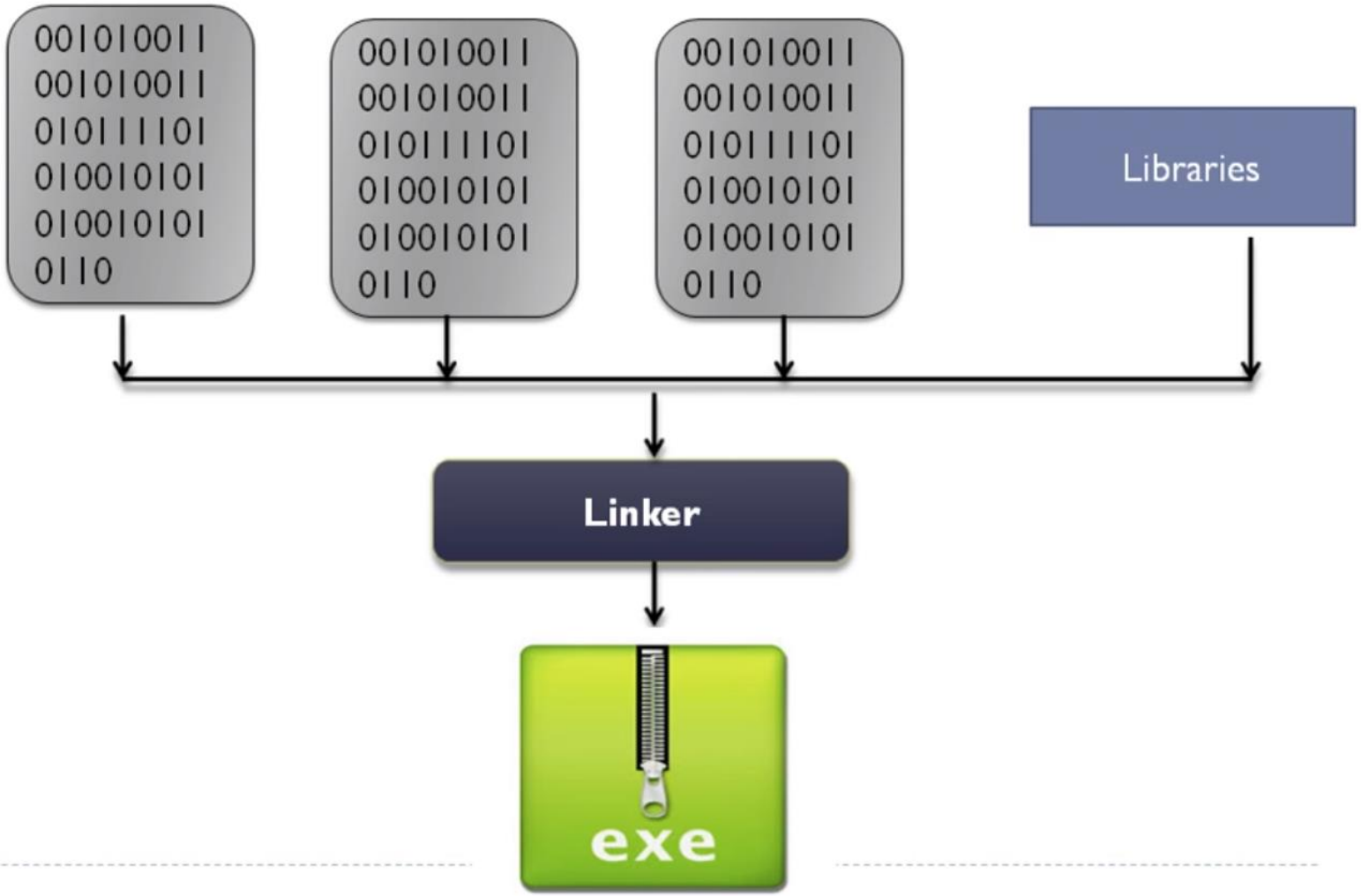
Quá trình biên dịch file C



Quá trình biên dịch file C



Quá trình biên dịch file C



Ví dụ : Tạo chương trình thực thi từ nhiều modul

Toto project

file **main.c**

```
#include <stdio.h>
extern int x, y;
extern void toto();
int main(int argc, char *argv[]){
    toto();
    printf("KQ: %d \n",x * y);
    return 0;
}
```

file **M1.c**

```
int y = 10;
```

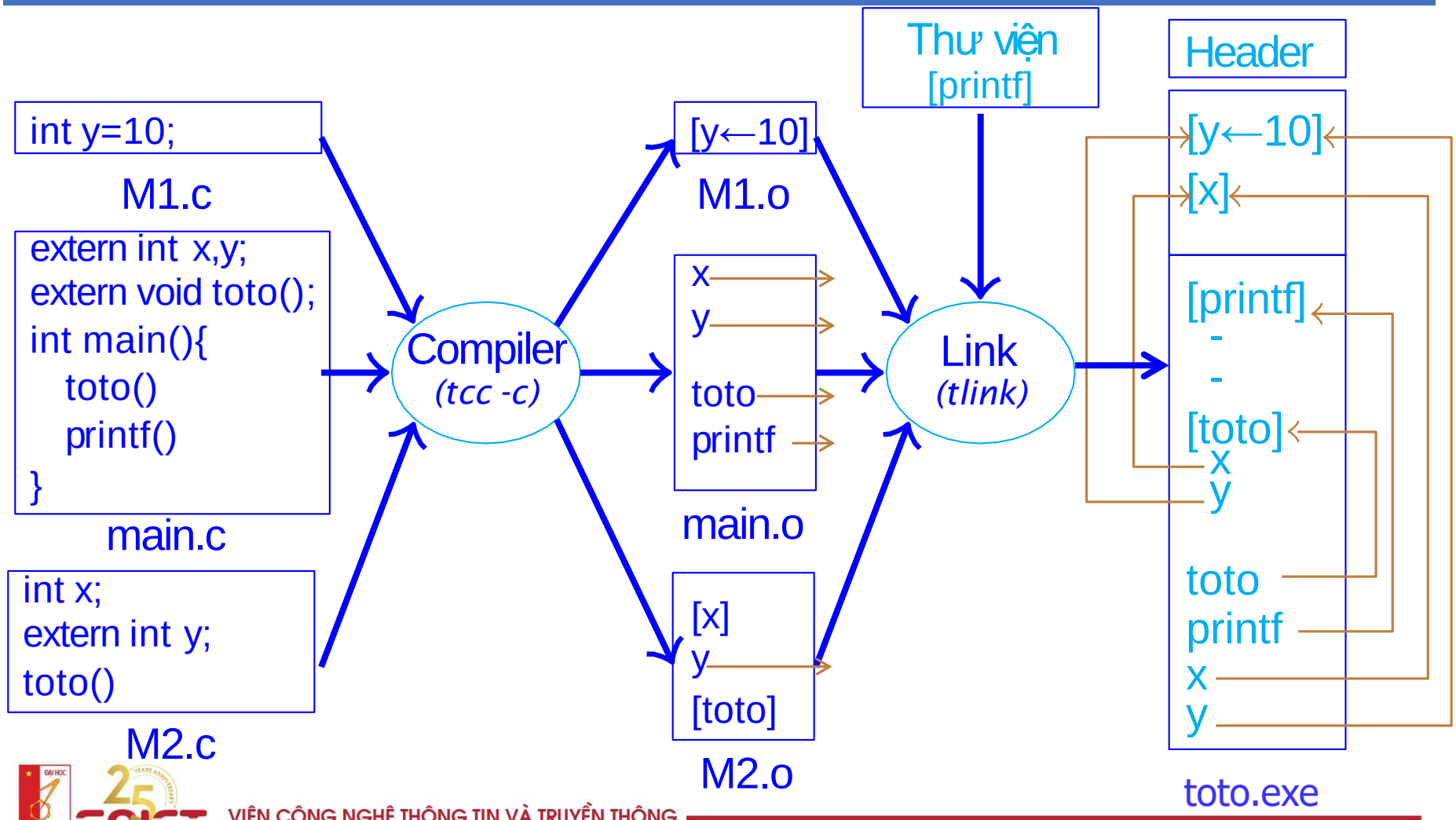
file **M2.c**

```
int x;
extern int y;
void toto(){
  x = 10 * y;
}
```

Kết quả

```
KQ: 1000
```

Quá trình biên dịch toto project



- Ví dụ
- **Bộ nhớ và chương trình**
- Liên kết địa chỉ
- Các cấu trúc chương trình

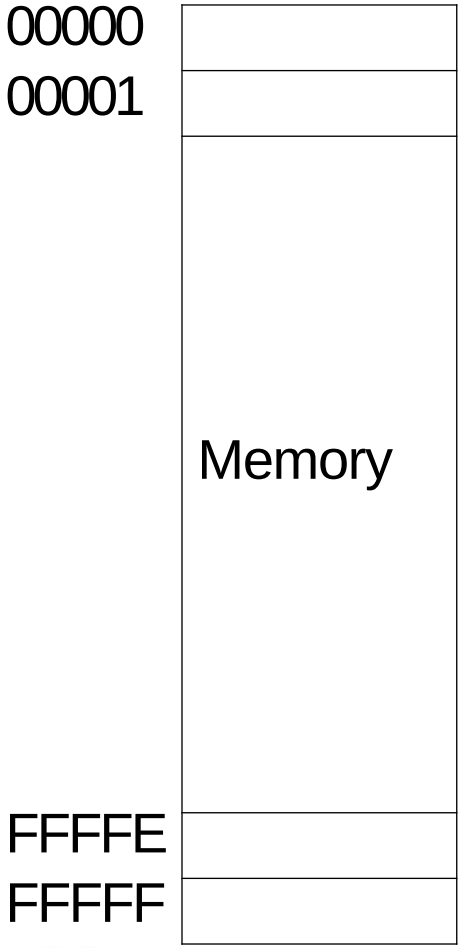
Phân cấp bộ nhớ

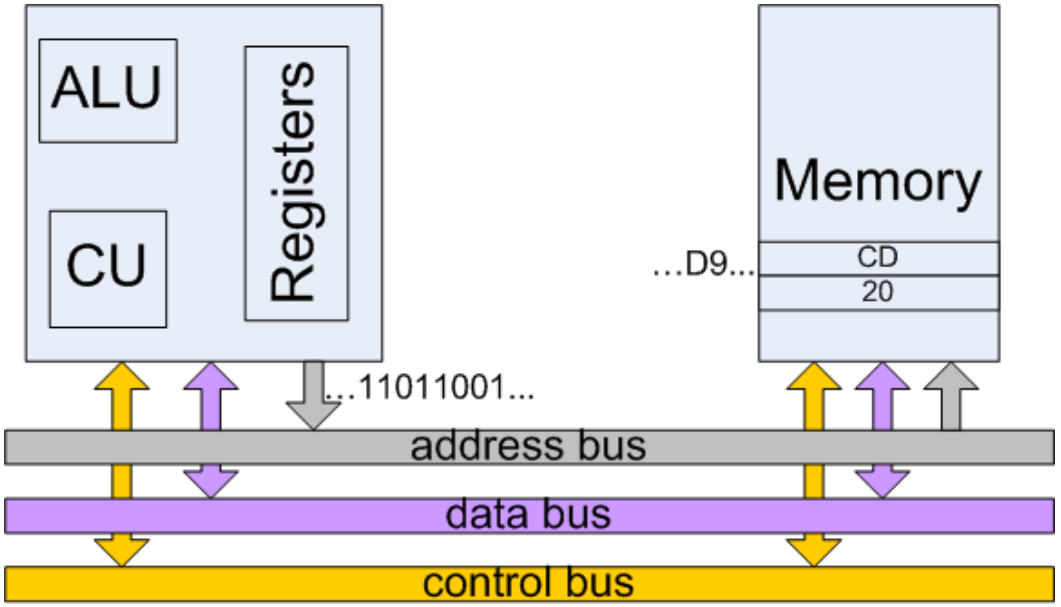
- Bộ nhớ là tài nguyên quan trọng của hệ thống
  - Chương trình phải nằm trong bộ nhớ trong để thực hiện

- Bộ nhớ được đặc trưng bởi kích thước và tốc độ truy nhập
- Bộ nhớ được phân cấp theo tốc độ truy nhập

Loại bộ nhớ	Kích thước	Tốc độ
Thanh ghi (Registers) Cache trên VXL Cache mức 2 Bộ nhớ chính Bộ nhớ lưu trữ (Disk) Băng từ, đĩa quang	bytes Kilo Bytes KiloByte-MegaByte MegaByte-GigaByte GigaByte-Terabytes Không giới hạn	Tốc độ CPU(ηs) 10 nano seconds 100 nanoseconds e Micro-seconds Mili-Seconds 10 Seconds

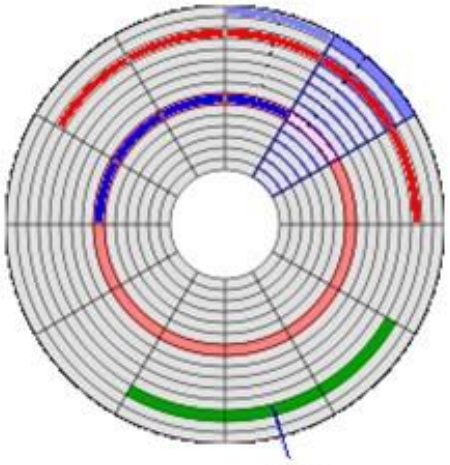
Bộ nhớ chính



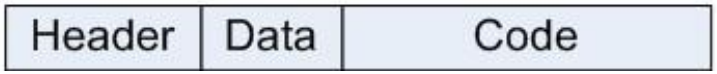


- Dùng lưu trữ dữ liệu và chương trình
- Là mảng các ô nhớ kiểu bytes, words
- Mỗi ô nhớ có một địa chỉ riêng
  - **Địa chỉ vật lý**: địa chỉ x/hiện ở chân VXL

Chương trình



File



- Tồn tại trên thiết bị lưu trữ ngoài
- Là các file nhị phân thực thi được
  - Vùng tham số file
  - Lệnh máy (mã nhị phân),
  - Vùng dữ liệu (biến toàn cục), . .
- Phải được đưa vào bộ nhớ trong và được đặt trong một tiến trình để thực hiện (tiến trình thực hiện chương trình)
- **Hàng đợi vào** (input queue)
  - Tập các tiến trình ở bộ nhớ ngoài (*thông thường: disk*)
  - Đợi để được đưa vào bộ nhớ trong và thực hiên

Các bước thực hiện chương trình

- Nạp chương trình vào bộ nhớ
  - Đọc và phân tích (dịch) file thực thi (VD file *.com, file *.exe)
  - Xin vùng nhớ để nạp chương trình từ file trên đĩa
  - Thiết lập các tham số, các thanh ghi tới giá trị thích hợp

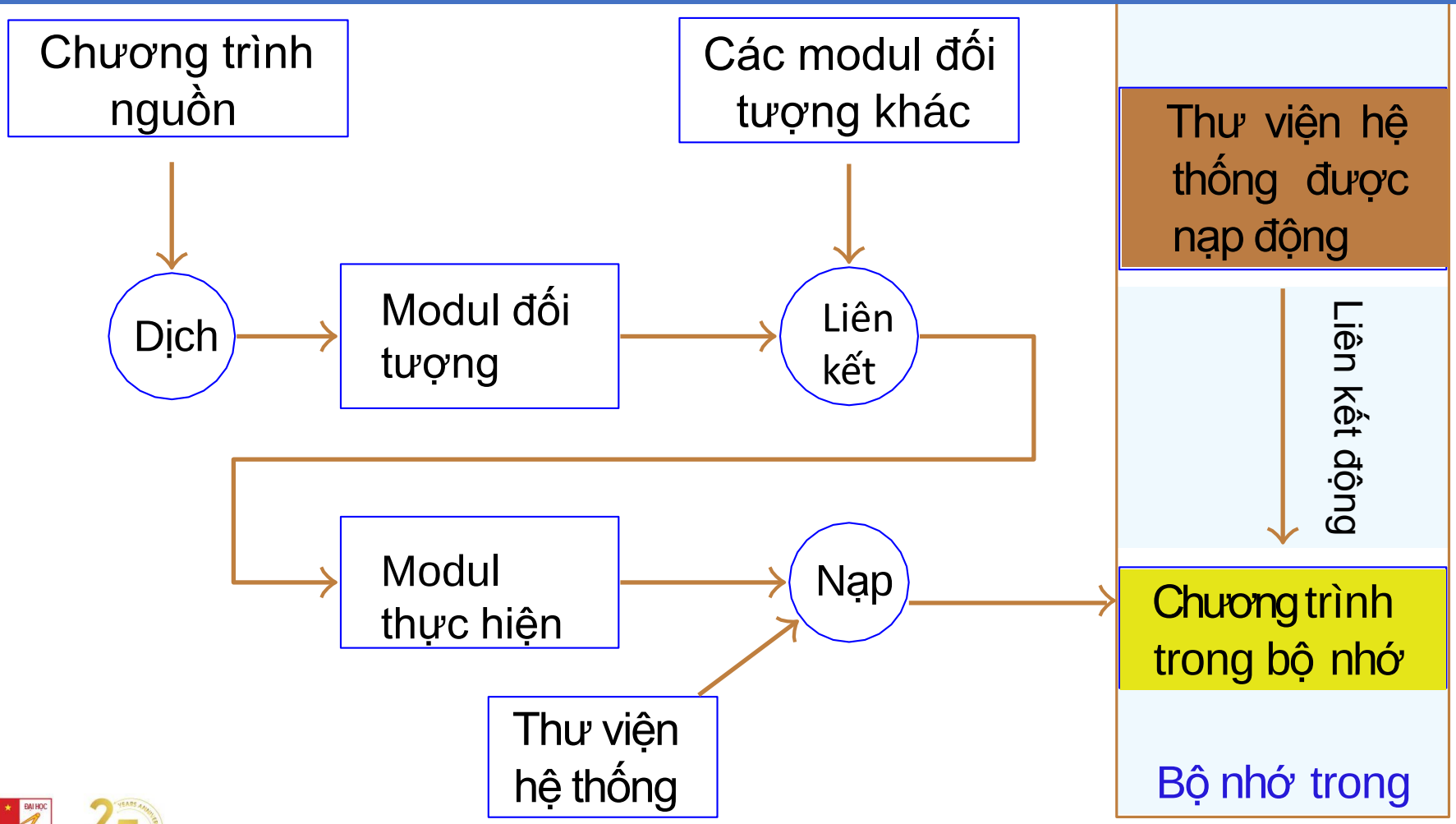
- Thực thi chương trình
  - CPU lấy các lệnh trong bộ nhớ tại vị trí được xác định bởi bộ đếm chương trình (Program counter)
    - Cặp thanh ghi CS:IP với VXL họ Intel (Ví dụ : 80x86)
  - CPU giải mã lệnh
    - Có thể lấy thêm toán hạng từ bộ nhớ
  - Thực hiện lệnh với toán hạng
  - Nếu cần thiết, lưu kết quả vào bộ nhớ tại một địa chỉ xác định

Các bước thực hiện chương trình (tiếp)

- Thực hiện xong
  - Giải phóng vùng không gian nhớ dành cho chương trình
- Vấn đề
  - Chương trình có thể được nạp vào vị trí bất kỳ trong bộ nhớ
  - Khi thực hiện chương trình sinh ra chuỗi địa chỉ bộ nhớ
  - Truy nhập địa chỉ bộ nhớ như thế nào?

- Ví dụ
- Bộ nhớ và chương trình
- **Liên kết địa chỉ**
- Các cấu trúc chương trình

Các bước xử lý chương trình ứng dụng



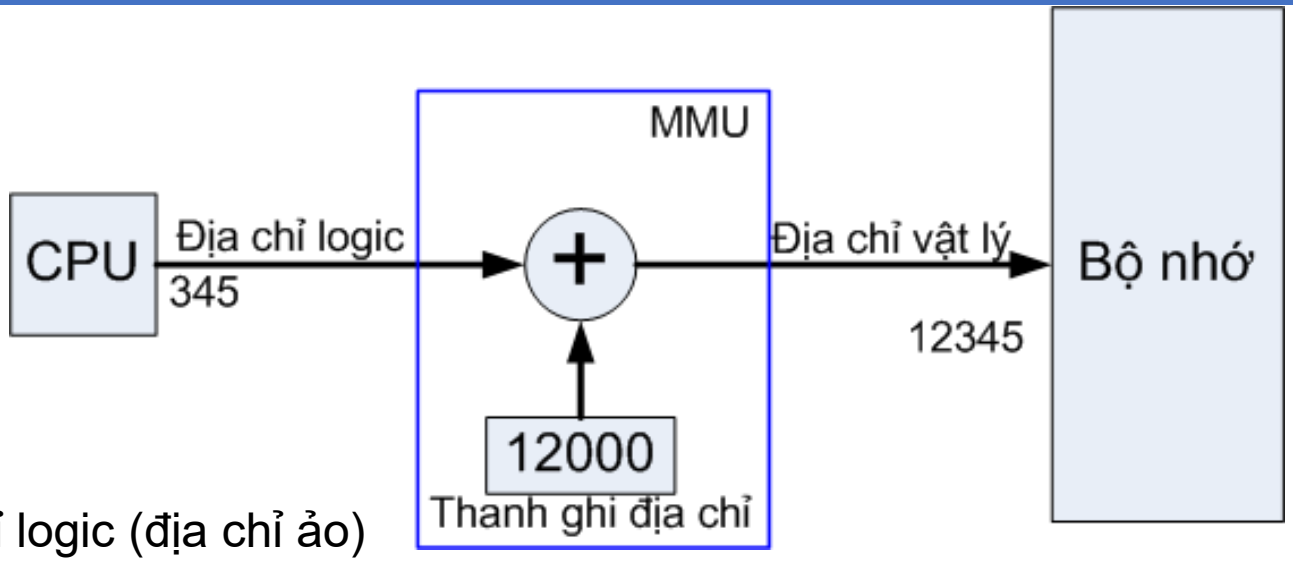
Các kiểu địa chỉ

- Địa chỉ biểu tượng (symbolic)
  - Là tên của đối tượng trong chương trình nguồn
  - Ví dụ: counter, x, y,...
- Địa chỉ tương đối
  - Sinh ra từ địa chỉ biểu tượng trong giai đoạn dịch (compiler)
  - Là vị trí tương đối của đối tượng kể từ đầu modul
    - Byte thứ 10 kể từ đầu modul
    - EB08 ⇒ JMP +08: Nhảy tới vị trí cách vị trí hiện tại 8 ô
- Địa chỉ tuyệt đối
  - Sinh ra từ địa chỉ tương đối trong giai đoạn nạp chương trình thực thi vào bộ nhớ để thực hiện
    - Với PC: địa chỉ tương đối <Seg :Ofs>→ Seg * 16+Ofs
  - Là địa chỉ của đối tượng trong bộ nhớ vật lý-địa chỉ vật lý
  - Ví dụ: JMP 010A⇒ Nhảy tới ô nhớ có vị trí 010Ah tại cùng đoạn mã lệnh (CS)
    - Nếu CS=1555h, sẽ đi tới vị trí: 1555h*10h+010Ah =1560Ah

Xác định địa chỉ

- Xác định địa chỉ câu lệnh và dữ liệu trong bộ nhớ có thể thực hiện tại các giai đoạn khác nhau khi xử lý chương trình ứng dụng
- Giai đoạn dịch:
  - Sử dụng nếu biết chương trình sẽ nằm ở đâu trong bộ nhớ
  - Khi dịch sẽ sinh ra mã (địa chỉ) tuyệt đối
  - Phải dịch lại khi vị trí bắt đầu thay đổi
- Thời điểm nạp:
  - Sử dụng khi không biết c/trình sẽ nằm ở đâu trong bộ nhớ
  - Các đối tượng được dịch ra sẽ mang địa chỉ tương đối
  - Xác định địa chỉ được hoãn lại tới khi khi nạp chương trình vào bộ nhớ
- Trong khi thực hiện:
  - S/dụng khi các tiến trình có thể thay đổi vị trí trong khi t/hiện
  - Xác định địa chỉ được hoãn lại tới khi thực thi chương trình
  - Thường đòi hỏi trợ giúp từ phần cứng
  - Được sử dụng trong nhiều hệ điều hành

Địa chỉ vật lý-địa chỉ logic



- Địa chỉ logic (địa chỉ ảo)
  - Được sinh ra trong tiến trình, (CPU đưa ra)
  - Được khối quản lý bộ nhớ (MMU) chuyển sang địa chỉ vật lý khi truy nhập tới đối tượng trong chương trình
- Địa chỉ vật lý
  - Địa chỉ của một phần tử (byte/word) của bộ nhớ
  - Tương ứng với địa chỉ logic được CPU đưa ra
- Chương trình làm việc với địa chỉ logic

- Ví dụ
- Bộ nhớ và chương trình
- Liên kết địa chỉ
- Các cấu trúc chương trình

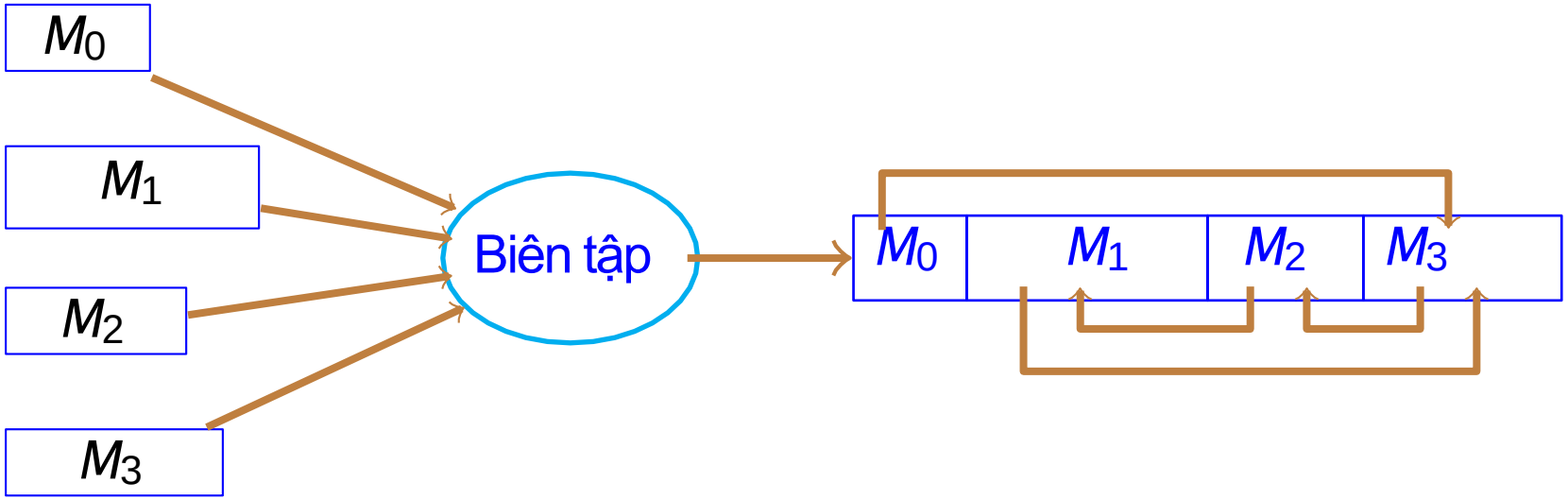
① Cấu trúc tuyến tính

② Cấu trúc nạp động

③ Cấu trúc liên kết động

④ Cấu trúc Overlays

Cấu trúc tuyến tính



- Sau khi biên tập, các modul được tập hợp thành một chương trình hoàn thiện
  - Chứa đầy đủ các thông tin để có thể thực hiện được
  - Các biến trỏ ngoài đã thay bằng giá trị cụ thể
  - Để thực hiện, chỉ cần định vị 1 lần trong bộ nhớ

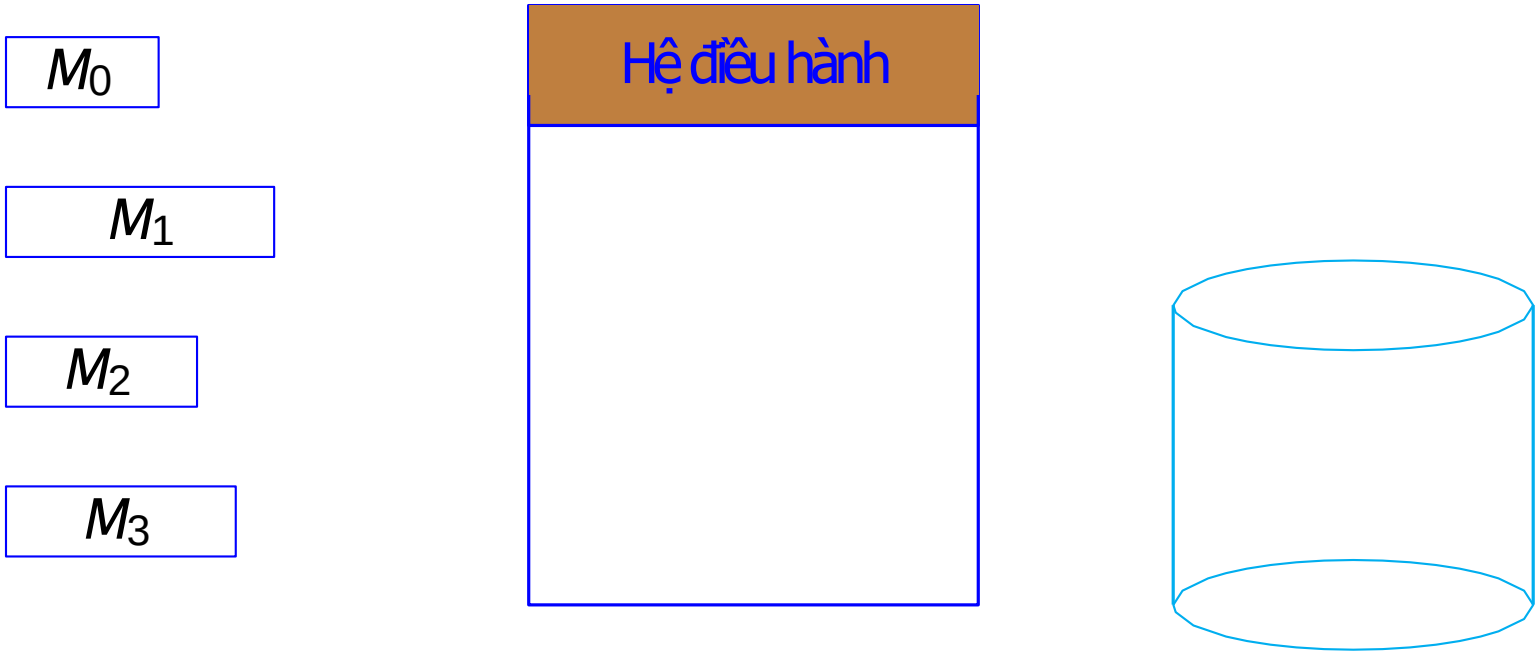
Cấu trúc tuyến tính

- Ưu điểm
  - Đơn giản, dễ tổ chức biên tập và định vị chương trình
  - Thời gian thực hiện nhanh
  - Tính lưu động cao

- Nhược điểm
  - Lãng phí bộ nhớ
    - Không phải toàn bộ chương trình đều cần thiết cho thực hiện chương trình
  - Không thực hiện được chương trình có kích thước lớn hơn kích thước bộ nhớ vật lý

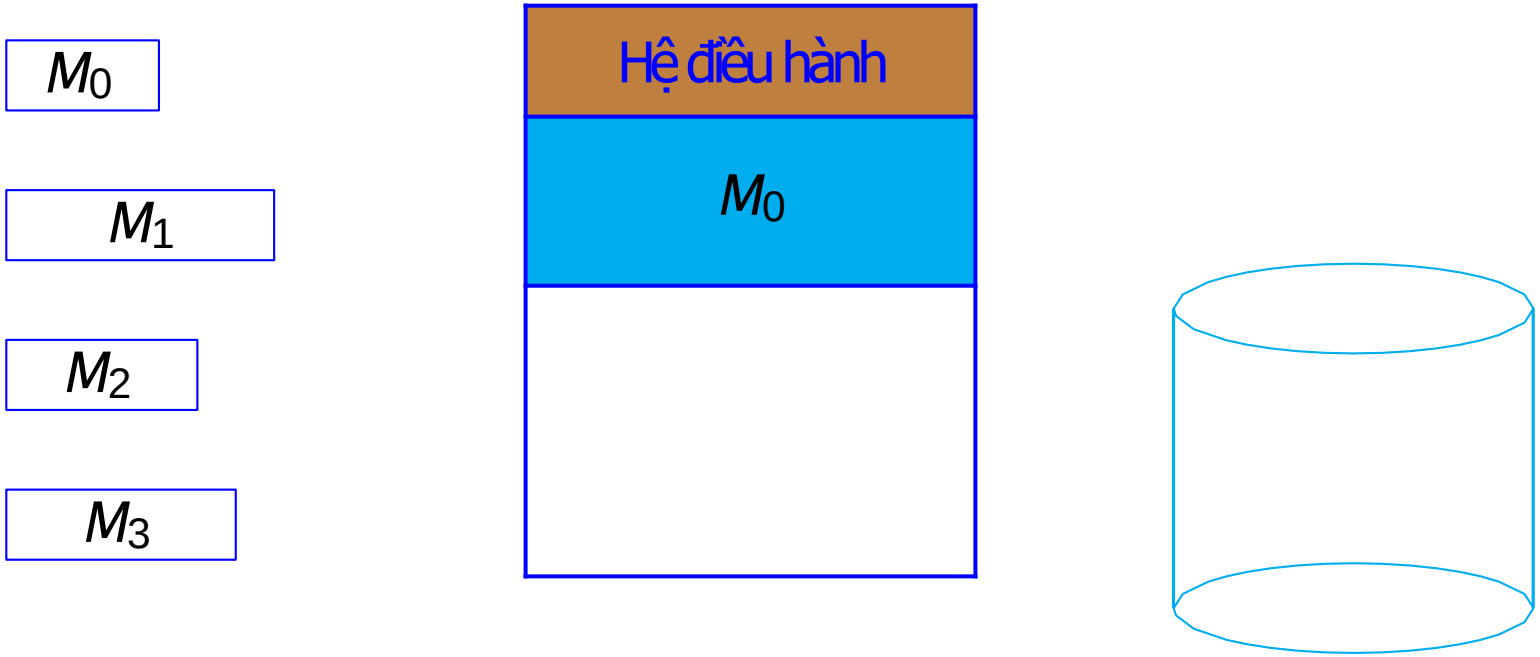
1.4 Các cấu trúc chương trình

Cấu trúc nạp động



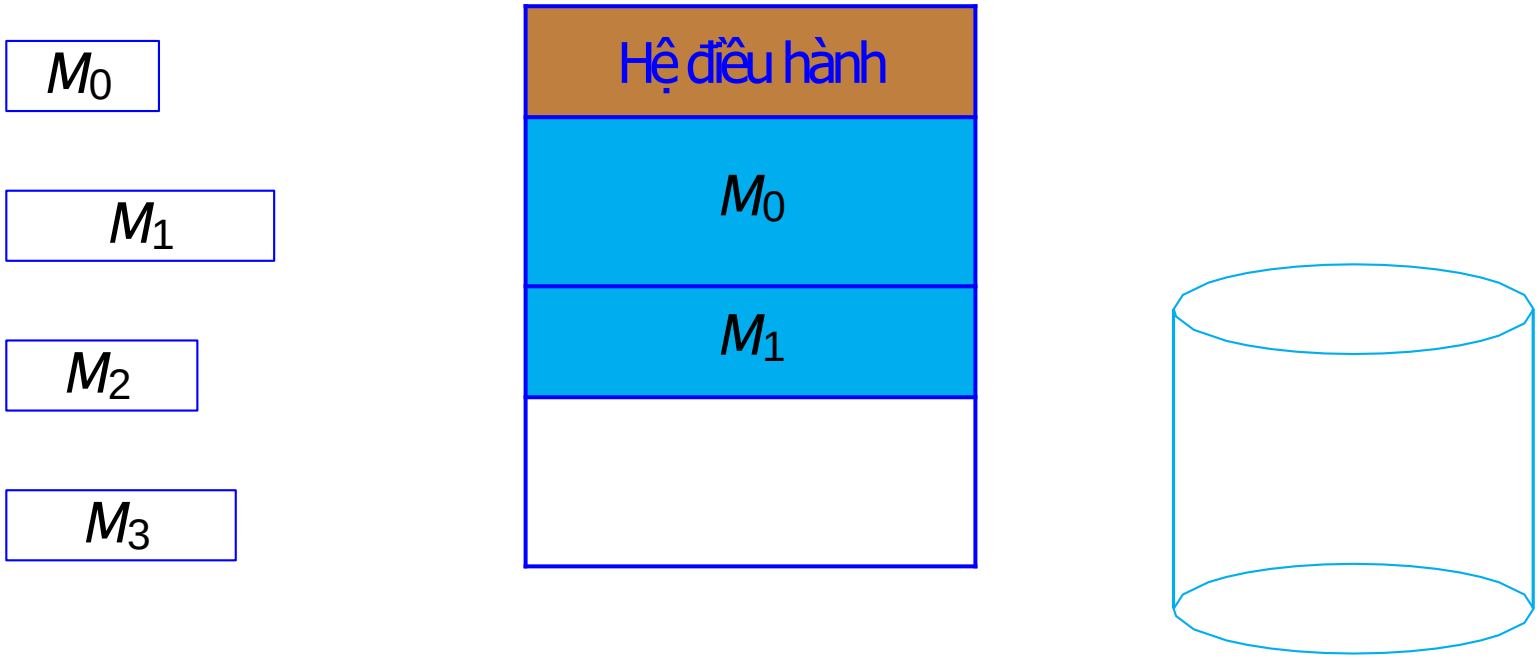
- Mỗi modul được biên tập riêng

Cấu trúc nạp động



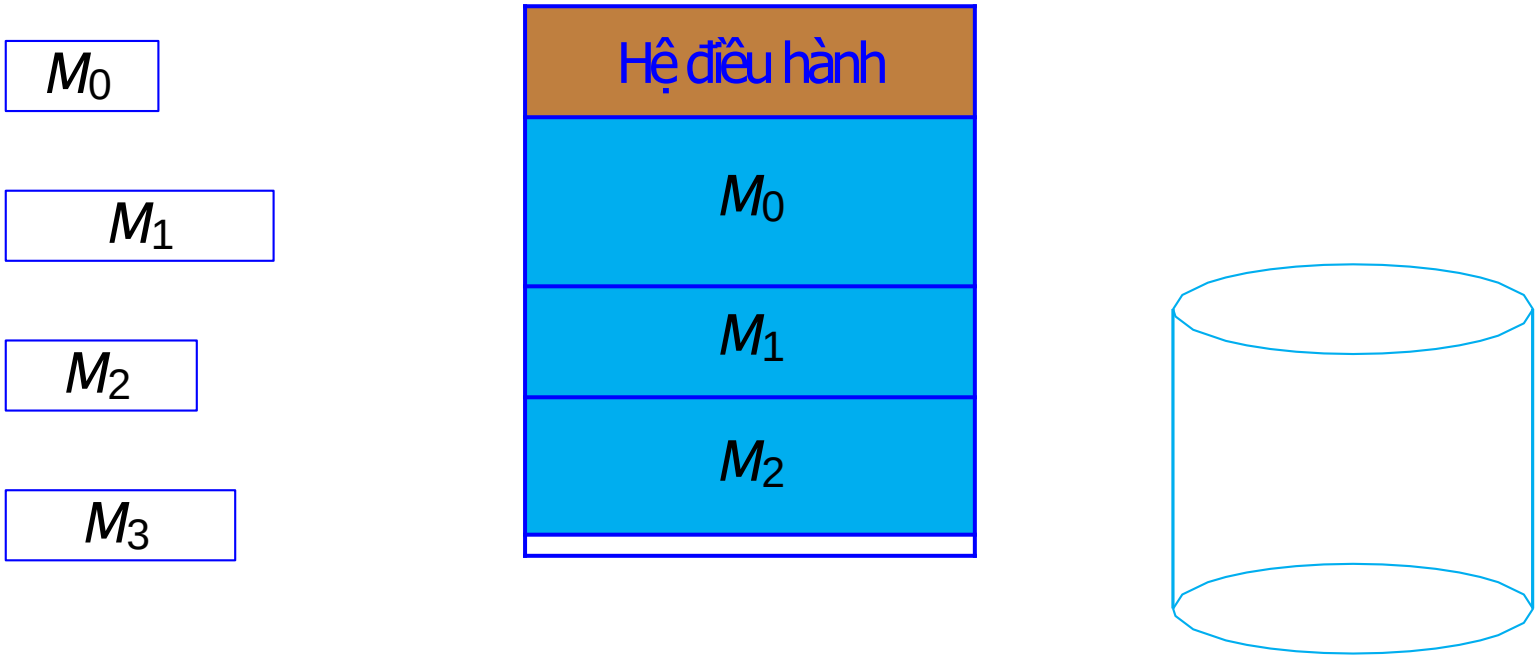
- Mỗi modul được biên tập riêng
- Khi thực hiện, hệ thống sẽ định vị modul gốc

Cấu trúc nạp động



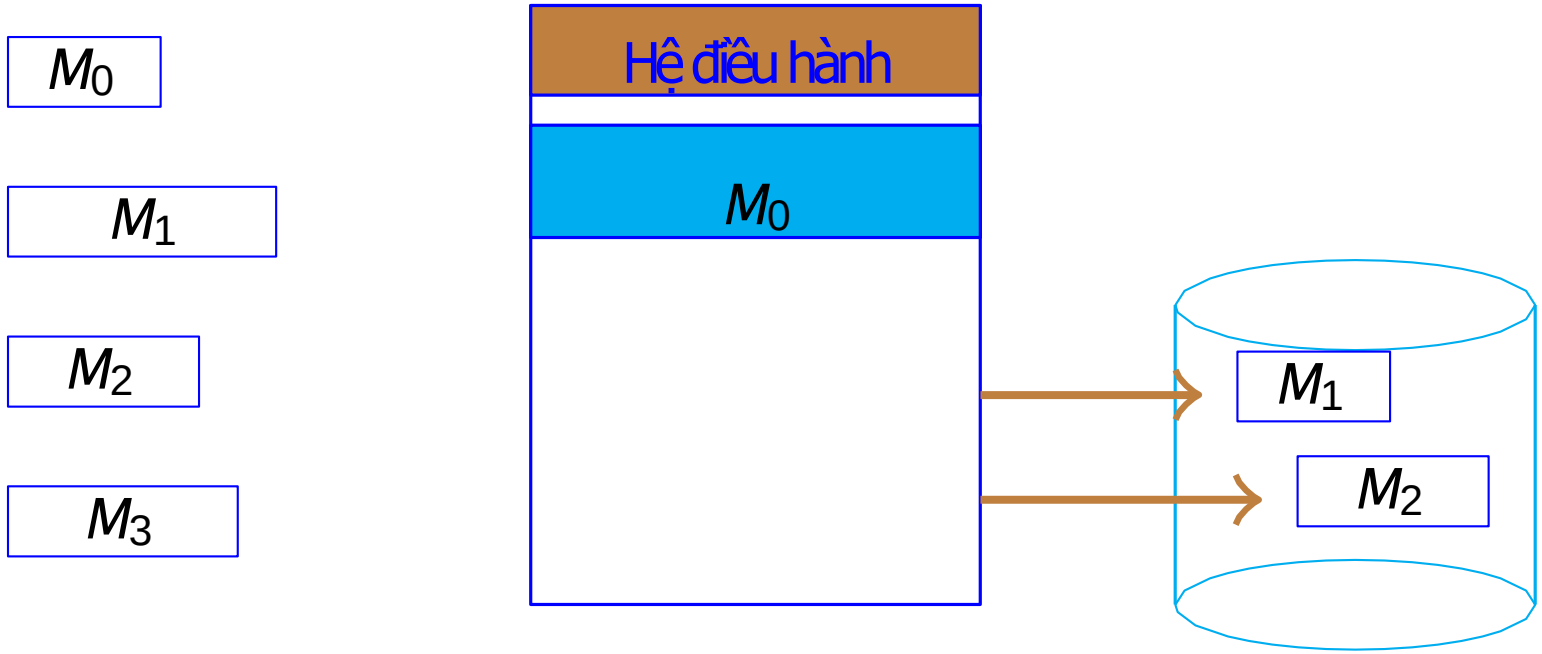
- Mỗi modul được biên tập riêng
- Khi thực hiện, hệ thống sẽ định vị modul gốc
- Cần tới modul nào sẽ xin bộ nhớ và giải nạp modul vào

Cấu trúc nạp động



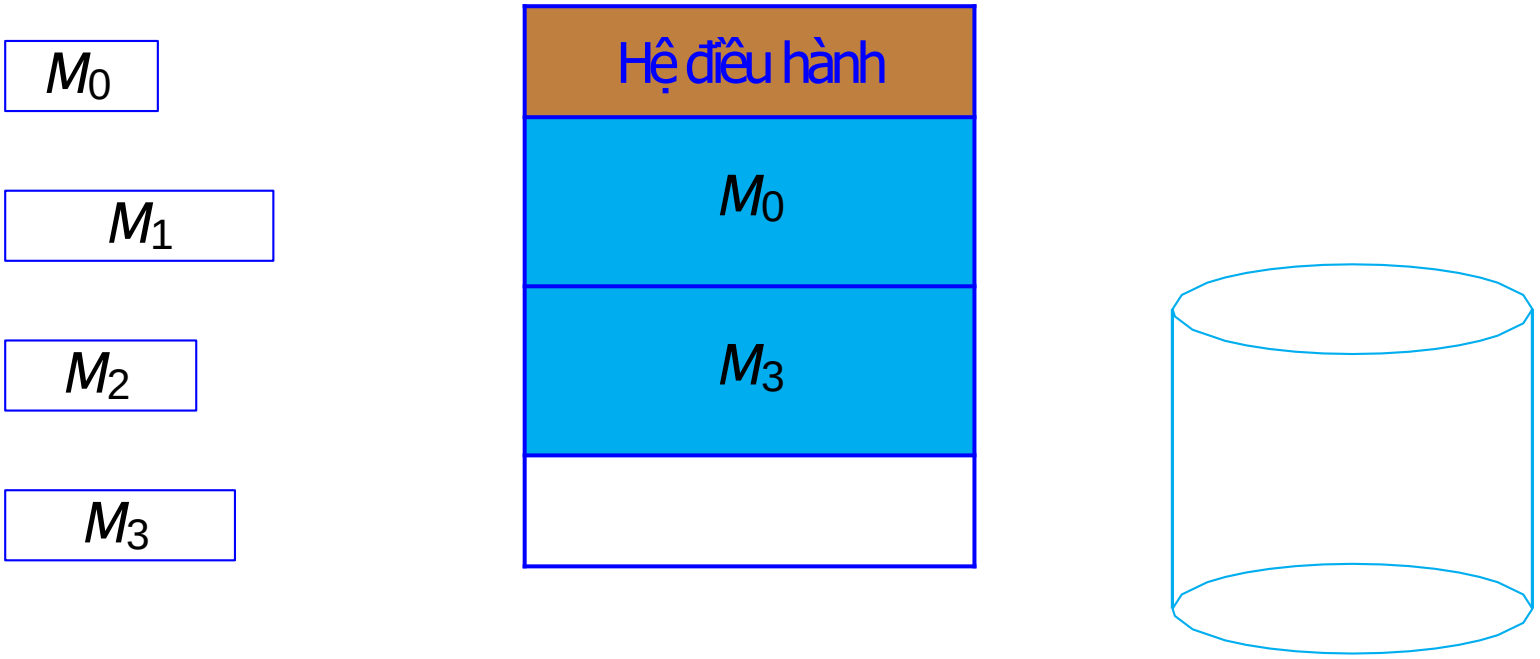
- Mỗi modul được biên tập riêng
- Khi thực hiện, hệ thống sẽ định vị modul gốc
- Cần tới modul nào sẽ xin bộ nhớ và giải nạp modul vào

Cấu trúc nạp động



- Mỗi modul được biên tập riêng
- Khi thực hiện, hệ thống sẽ định vị modul gốc
- Cần tới modul nào sẽ xin bộ nhớ và giải nạp modul vào
- Khi sử dụng xong một modul, hoặc khi thiếu vùng nhớ sẽ đưa nhưng modul không cần thiết ra ngoài

Cấu trúc nạp động



- Mỗi modul được biên tập riêng
- Khi thực hiện, hệ thống sẽ định vị modul gốc
- Cần tới modul nào sẽ xin bộ nhớ và giải nạp modul vào
- Khi sử dụng xong một modul, hoặc khi thiếu vùng nhớ sẽ đưa nhưng modul không cần thiết ra ngoài

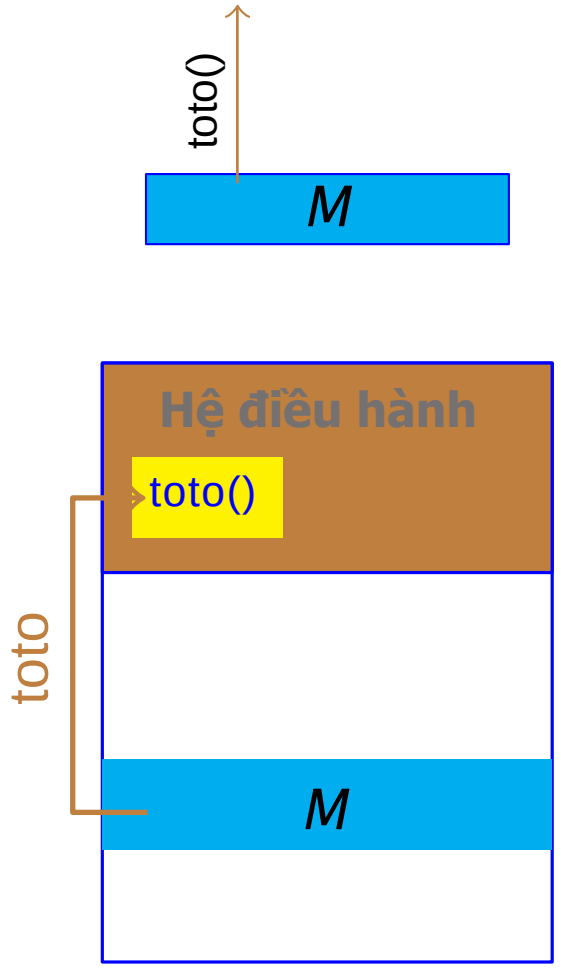
Cấu trúc nạp động

- Ưu điểm
- Có thể sử dụng vùng nhớ nhiều hơn phần dành cho chương trình
- Hiệu quả sử dụng bộ nhớ cao nếu quản lý tốt

- Nhược điểm
- Tốc độ thực hiện chậm
- Sai lầm sẽ dẫn tới lãng phí bộ nhớ và tăng thời gian thực hiện
- Yêu cầu người sử dụng phải nạp và xóa các modul
  - Người dùng phải nắm rõ hệ thống
  - Giảm tính lưu động

Cấu trúc liên kết động (DLL:Dynamic-link library)

- Các liên kết sẽ hoãn lại cho tới khi thực hiện chương trình
- Một phần của đoạn mã (stub) được sử dụng để tìm kiếm thủ tục tương ứng trong thư viện trong bộ nhớ
- Khi tìm thấy, stub sẽ được thay thế với địa chỉ của thủ tục và thực hiện thủ tục
- Hữu ích cho xây dựng thư viện



Cấu trúc Overlays

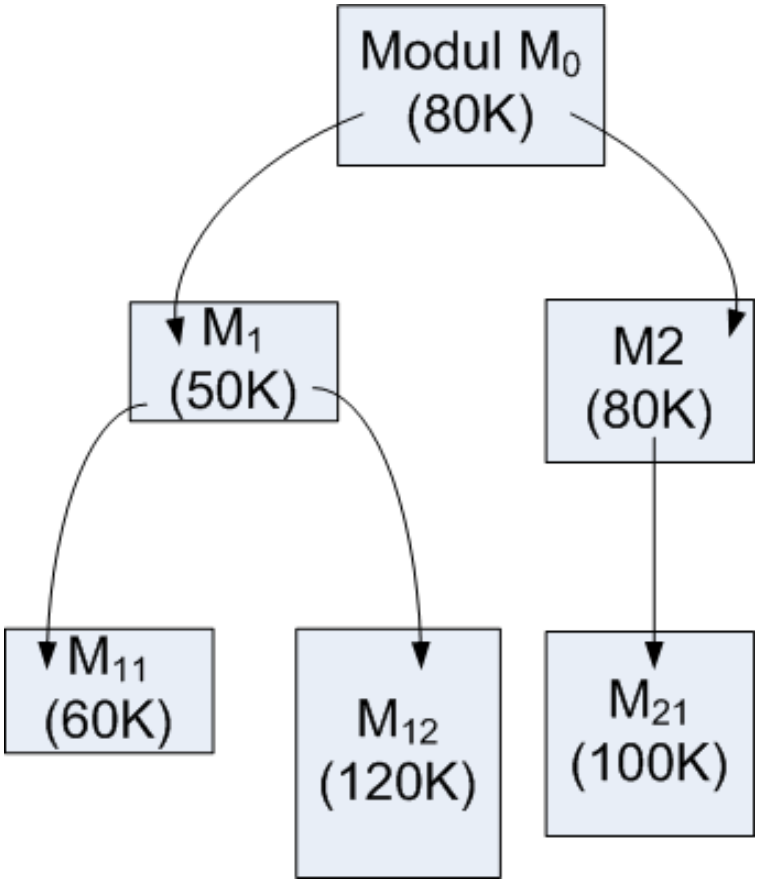
- Modul được chia thành các mức
  - Mức 0 chứa modul gốc, nạp và định vị chương trình
  - Mức 1 chứa các Modul được gọi từ những modul ở mức 0 và không đồng thời tồn tại
  - . . .
- Bộ nhớ cũng được chia thành mức ứng với mức chương trình
  - Kích thước bằng kích thước của modul lớn nhất cùng mức
- Để có cấu trúc Overlay, cần cung cấp thêm các thông tin
  - Chương trình bao nhiêu mức, mỗi mức gồm những modul nào
  - Thông tin cung cấp lưu trong file (sơ đồ overlay)

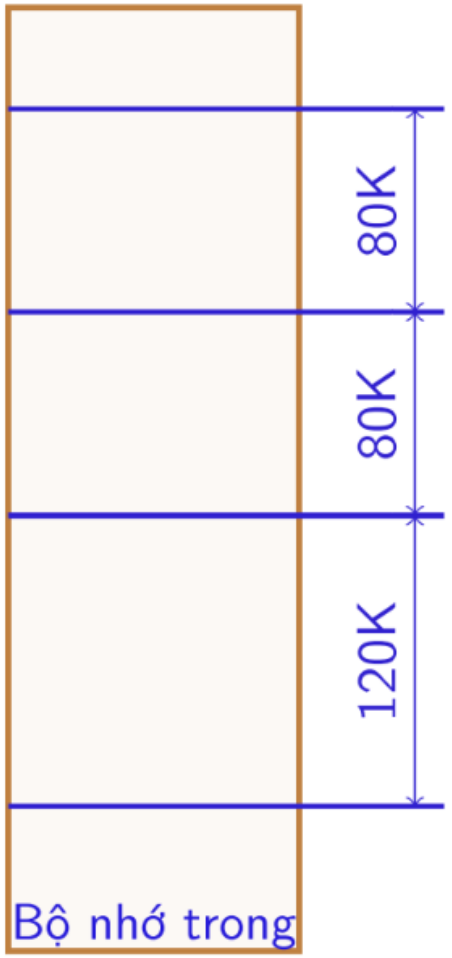
Cấu trúc Overlays

- Modul mức 0 được biên tập thành file thực thi riêng
- Khi thực hiện chương trình
  - Nạp modul mức 0 như chương trình tuyến tính
  - Cần tới modul khác, sẽ nạp modul vào mức bộ nhớ tương ứng
    - Nếu có modul đồng mức tồn tại, đưa ra bên ngoài

1.4 Các cấu trúc chương trình

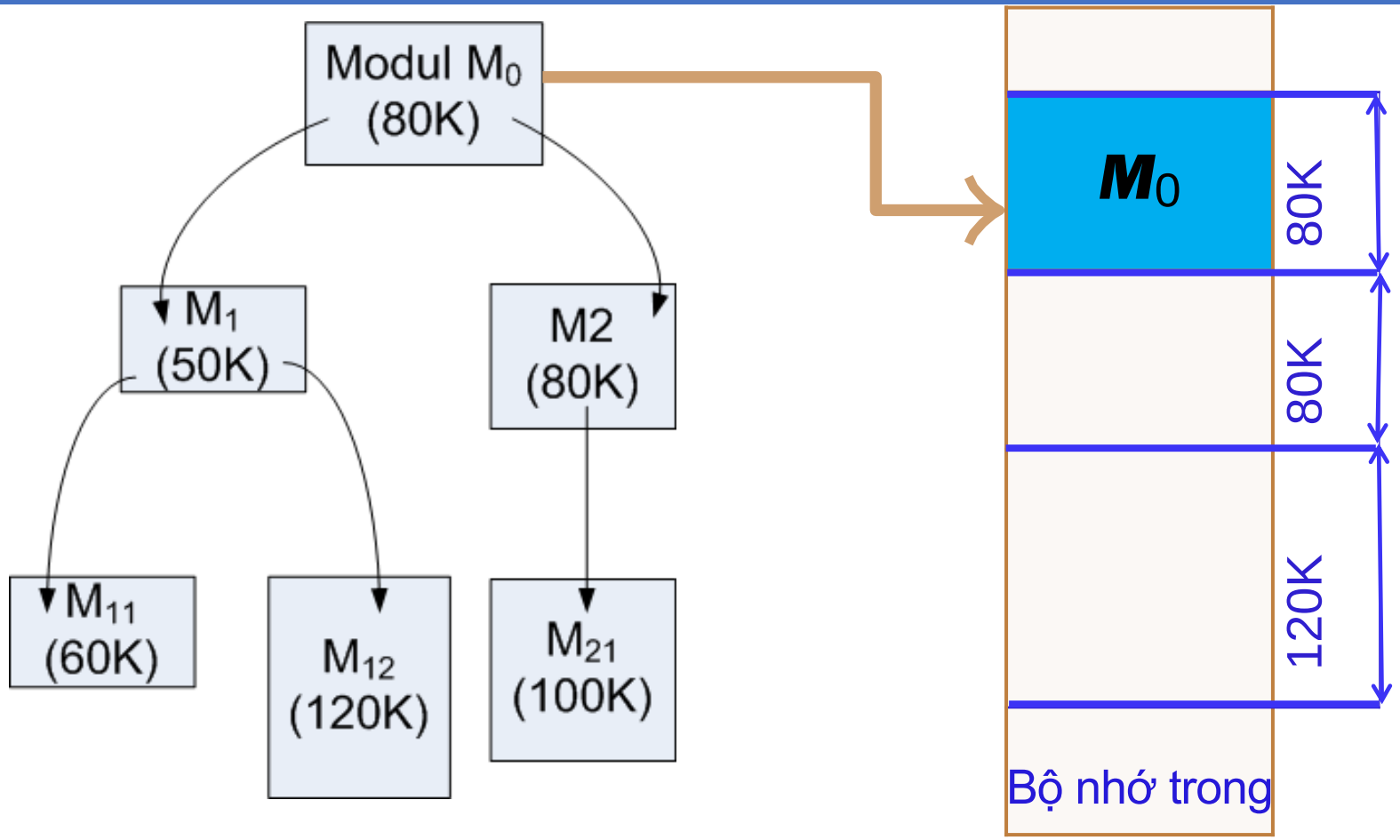
Cấu trúc Overlays



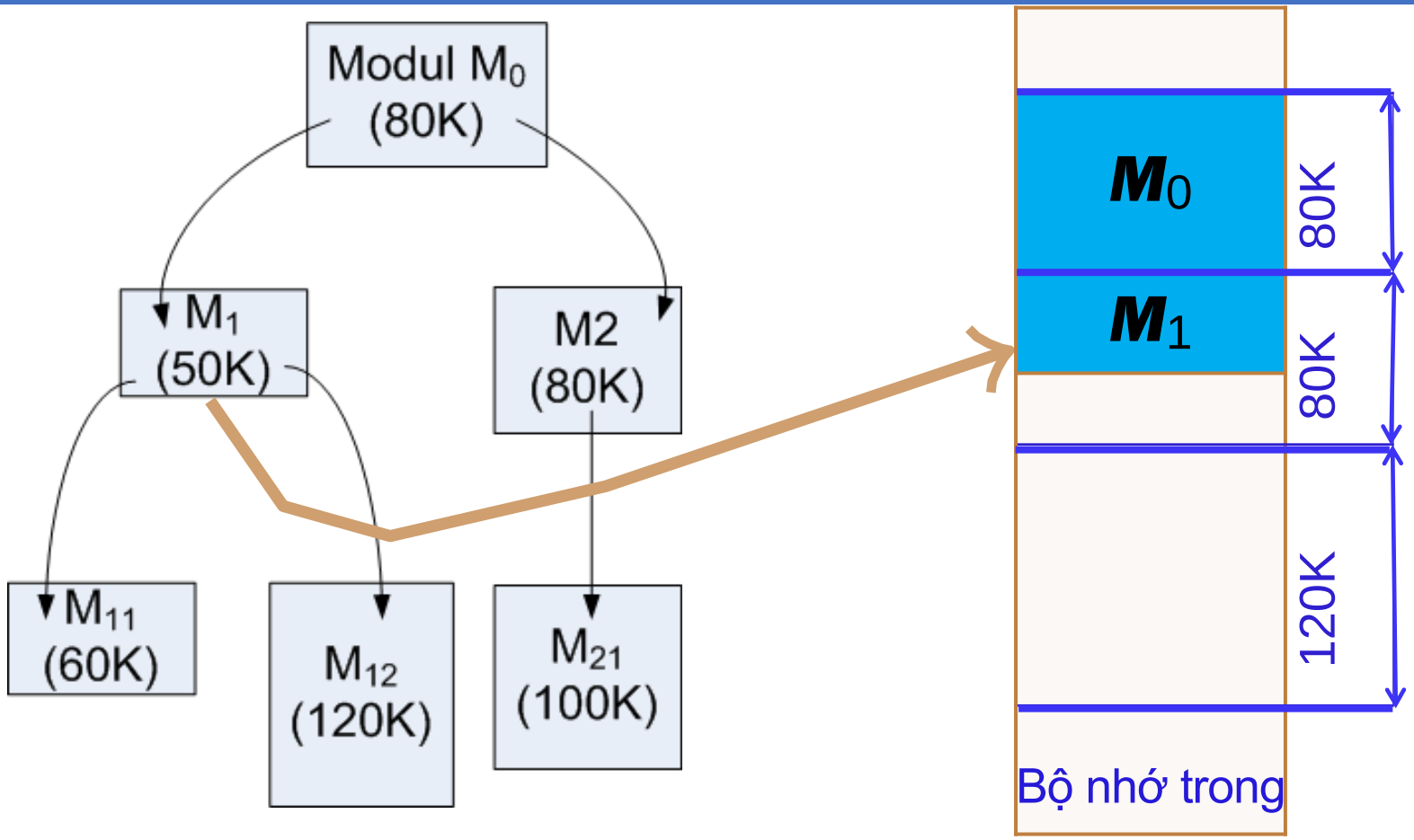


1.4 Các cấu trúc chương trình

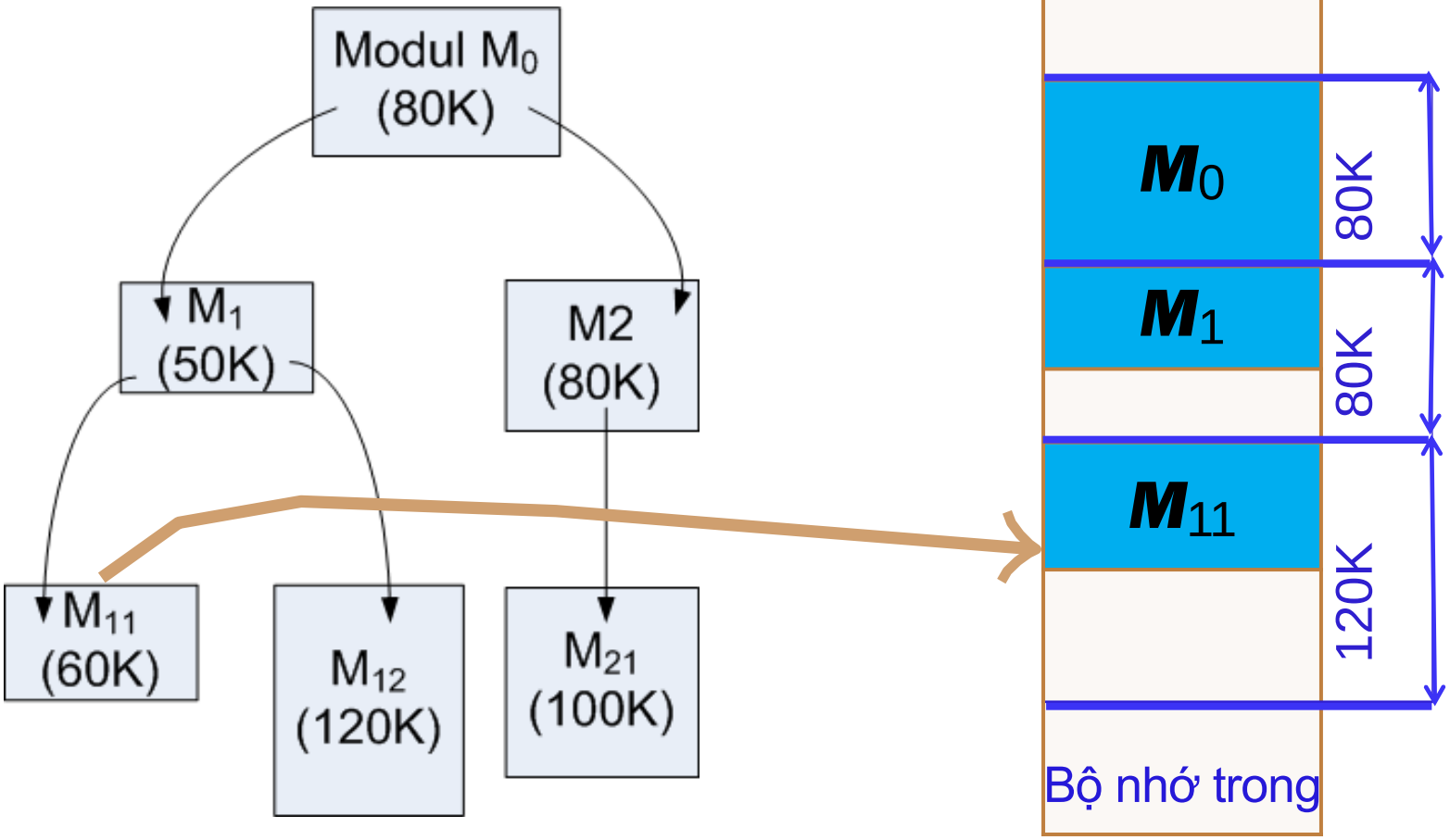
Cấu trúc Overlays



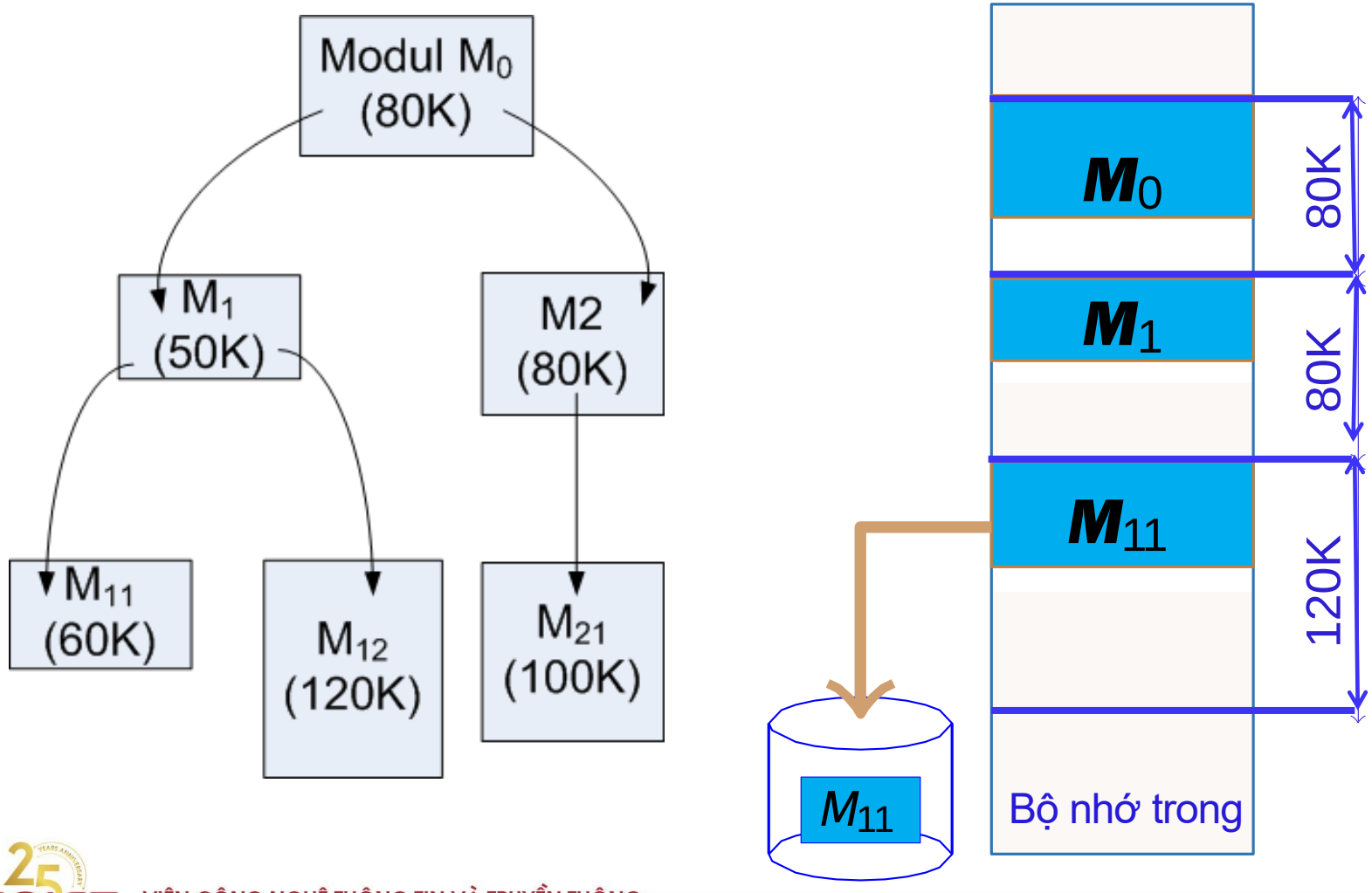
Cấu trúc Overlays



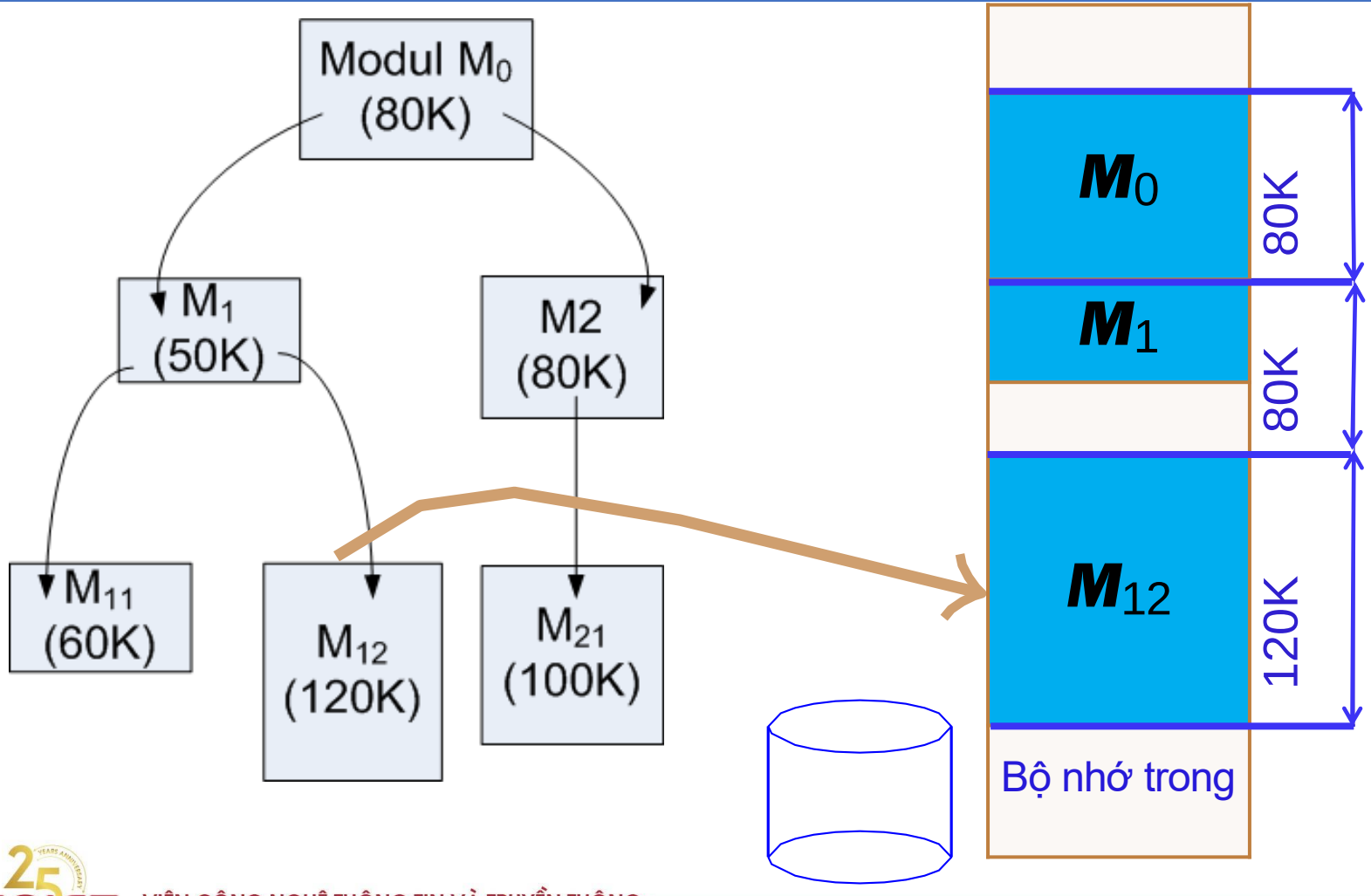
Cấu trúc Overlays



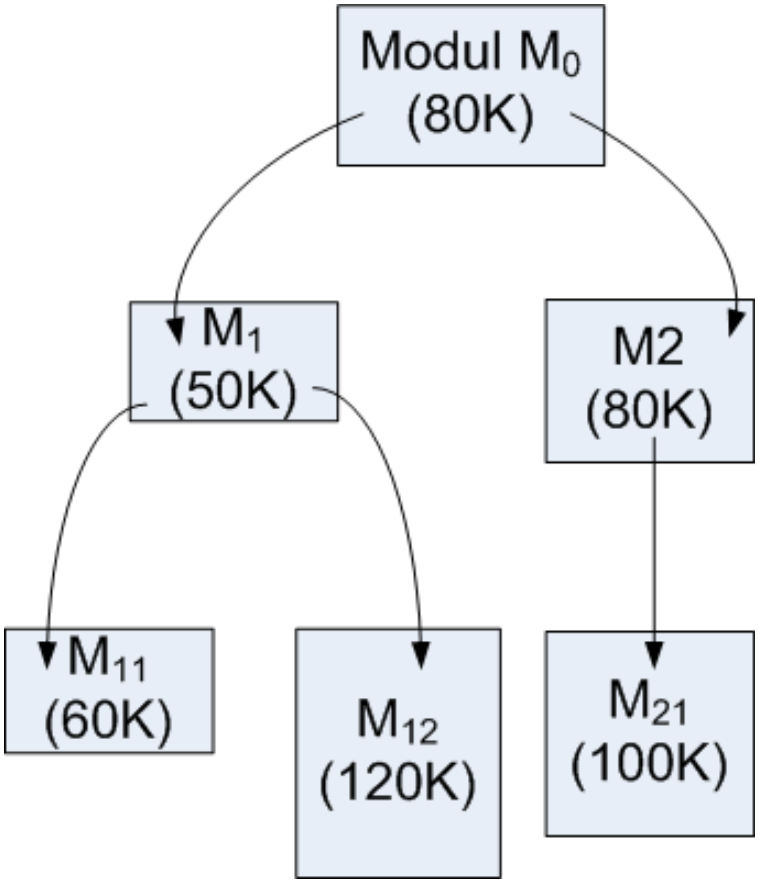
Cấu trúc Overlays

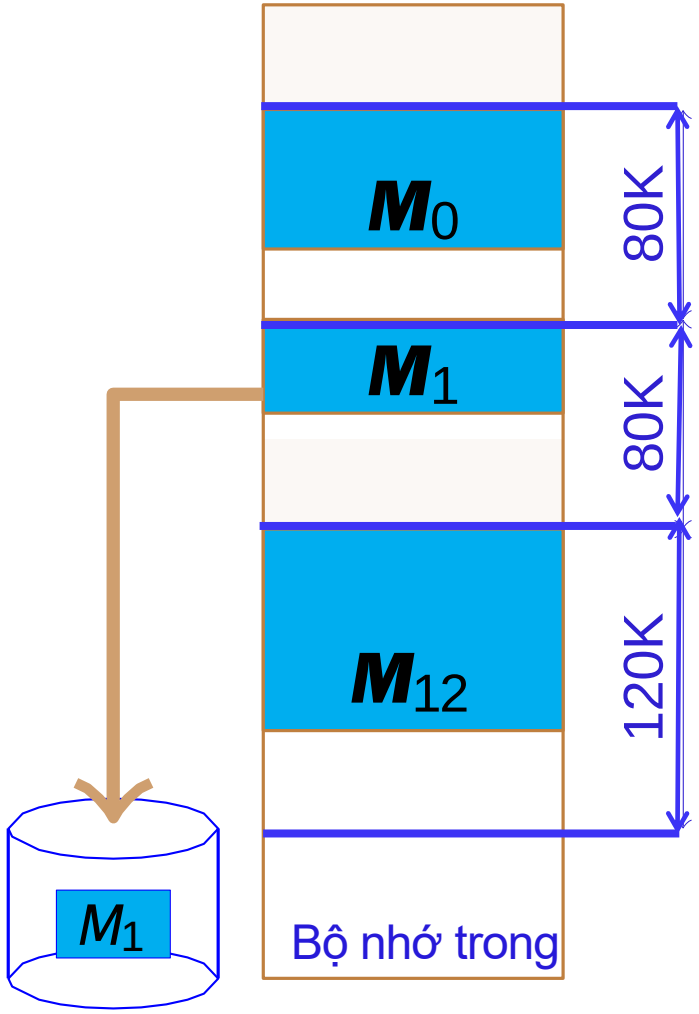


Cấu trúc Overlays

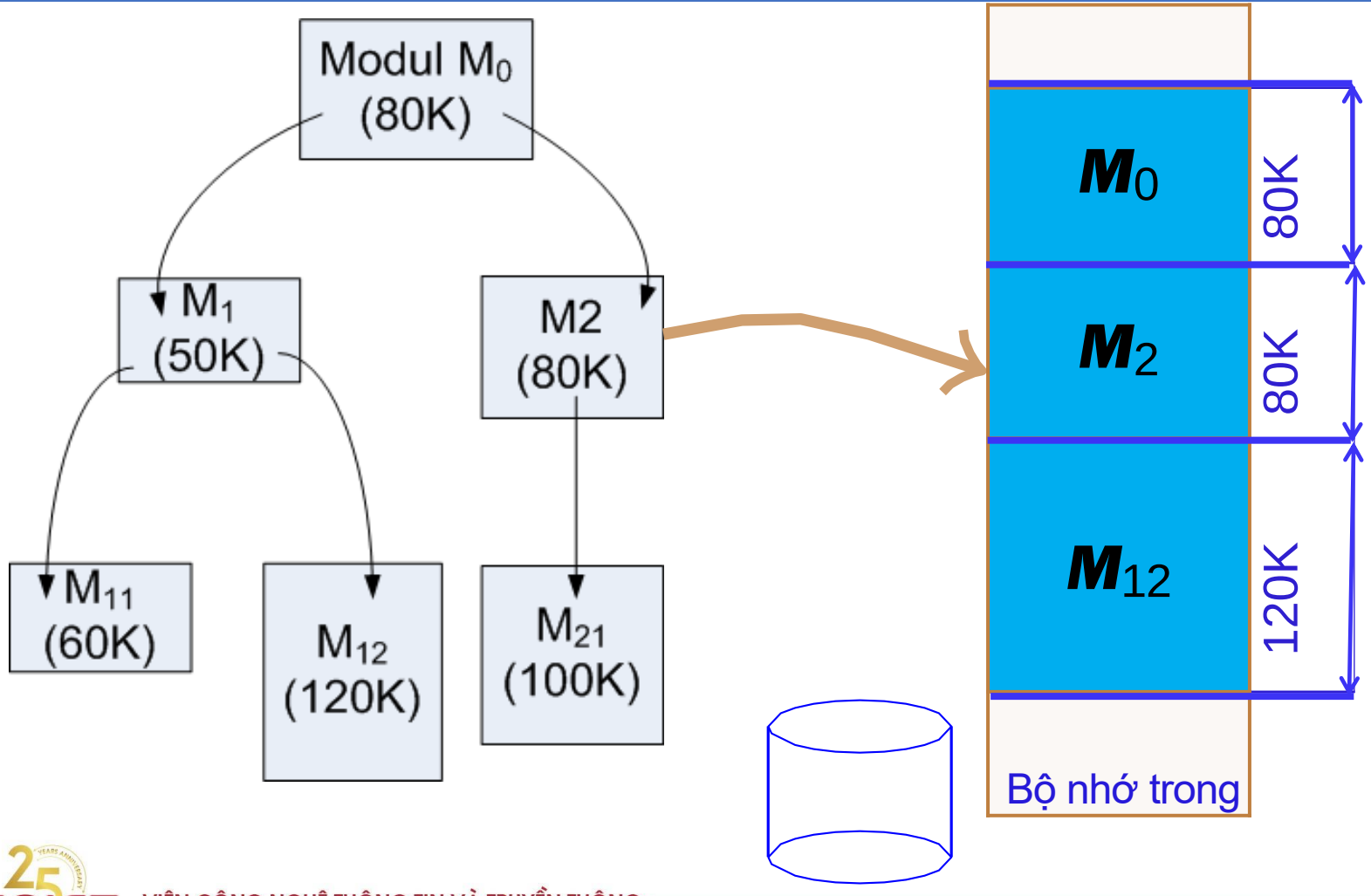


Cấu trúc Overlays





Cấu trúc Overlays



Cấu trúc Overlays:Nhận xét

- Cho phép chạy chương trình có kích thước lớn hơn kích thước HĐH dành cho
- Yêu cầu người sử dụng cung cấp các thông tin phụ
  - Hiệu quả sử dụng phụ thuộc vào các thông tin được cung cấp
- Hiệu quả sử dụng bộ nhớ phụ thuộc cách tổ chức các modul trong chương trình
  - Nếu tồn tại một modul có kích thước lớn hơn các modul khác cùng mức rất nhiều ⇒Hiệu quả giảm rõ rệt
- Quá trình nạp các modul là động, nhưng chương trình có tính chất tĩnh ⇒Không thay đổi trong các lần thực hiện
  - Cung cấp thêm bộ nhớ tự do, hiệu quả vẫn không đổi

Kết luận

Chương 3 Quản lý bộ nhớ

① Tổng quan

② Các chiến lược quản lý bộ nhớ

③ Bộ nhớ ảo

④ Quản lý bộ nhớ trong VXL họ Intel

- Chiến lược phân chương cố định
- Chiến lược phân chương động
- Chiến lược phân đoạn
- Chiến lược phân trang
- Chiến lược kết hợp phân đoạn-phân trang

Nguyên tắc

- Bộ nhớ được chia thành ***n*** phần
- Mỗi phần gọi là một *chương* (partition)
  - kích thước: không nhất thiết phải bằng nhau
  - được sử dụng như một vùng nhớ độc lập
    - Tại 1 thời điểm chỉ cho phép 1 chương trình tồn tại
    - Các chương trình nằm trong vùng nhớ cho tới khi kết thúc
- Ví dụ: Xét hệ thống:

Process	Size	time
P_1	120	20
P_2	80	15
P_3	70	5
P_4	50	5
P_5	140	12
Hàng đợi		

Nhận xét

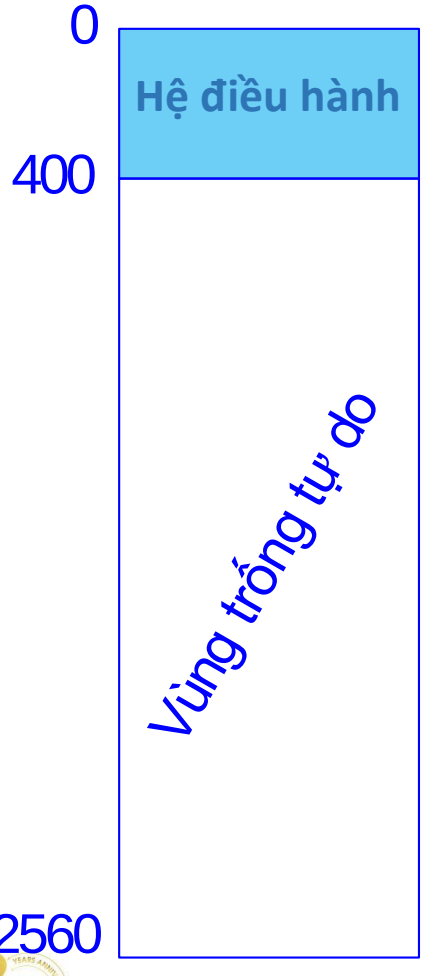
- Đơn giản, dễ tổ chức bảo vệ
  - Chương trình và vùng nhớ có một khóa bảo vệ
  - So sánh 2 khóa với nhau khi nạp chương trình
- Giảm thời gian tìm kiếm
- Phải sao các modul điều khiển ra làm nhiều bản và lưu ở nhiều nơi
- Hệ số song song không thể vượt quá *n*
- Bị phân đoạn bộ nhớ
  - Kích thước chương trình lớn hơn kích thước chương lớn nhất
  - Tổng bộ nhớ tự do còn lớn, nhưng không dùng để nạp các chương trình khác
  - ⇒Sửa lại cấu trúc chương, kết hợp một số chương kề nhau
- Áp dụng
  - Thường dùng cho quản lý các đĩa dung lượng lớn
  - HĐH OS/360 của IBM

- Chiến lược phân chương cố định
- **Chiến lược phân chương động**
- Chiến lược phân đoạn
- Chiến lược phân trang
- Chiến lược kết hợp phân đoạn-phân trang

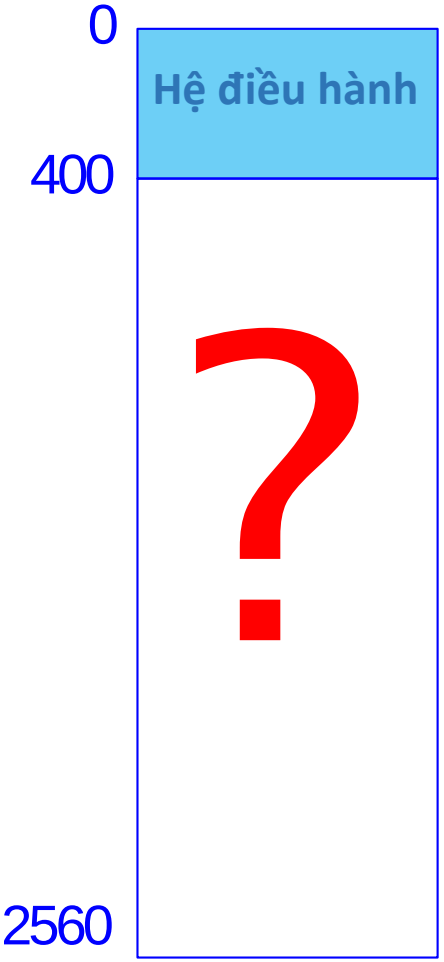
Nguyên tắc

- Chỉ có 1 danh sách quản lý bộ nhớ tự do
- Thời điểm ban đầu toàn bộ bộ nhớ là tự do với các tiến trình ⇒ vùng trống lớn nhất (*hole*)
- Khi 1 tiến trình yêu cầu bộ nhớ
  - Tìm trong DS vùng trống một phần tử đủ lớn cho yêu cầu
  - Nếu tìm thấy
    - Vùng trống được chia thành 2 phần
    - 1 phần cung cấp theo yêu cầu
    - 1 phần trả lại danh sách vùng trống tự do
  - Nếu không tìm thấy
    - Phải chờ tới khi có được 1 vùng trống thỏa mãn
    - Cho phép tiến trình khác trong hàng đợi thực hiện (nếu độ ưu tiên đảm bảo)
- Khi tiến trình kết thúc
  - Vùng nhớ chiếm được trả về DS quản lý vùng trống tự do
  - Kết hợp với các vùng trống khác liên kề nếu cần thiết

Bộ nhớ chính



Process	Size	time
P_1	600	10
P_2	1000	5
P_3	300	20
P_4	700	8
P_5	500	15
File đợi		



Chiến lược lựa chọn vùng trống tự do

Có nhiều chiến lược lựa chọn vùng trống cho yêu cầu

- First Fit : Vùng trống đầu tiên thỏa mãn
- Best Fit : Vùng trống vừa vặn nhất
- Worst Fit : Vùng trống kích thước lớn nhất

Buddy Allocation: Cung cấp nhớ

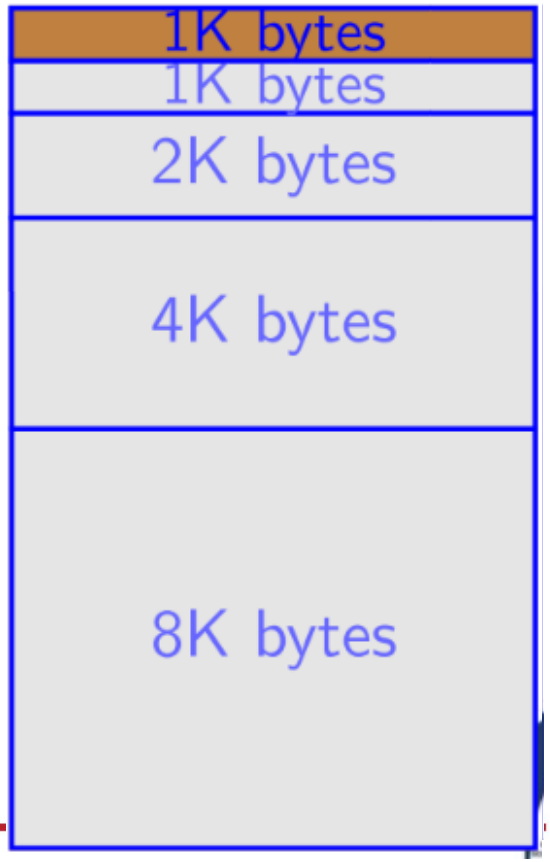
Nguyên tắc: Chia đôi liên tiếp vùng trống tự do cho tới khi thu được vùng trống nhỏ nhất thỏa mãn

Cung cấp cho yêu cầu n bytes

- Chia vùng trống tìm được thành 2 khối bằng nhau (gọi là buddies)
- Tiếp tục chia vùng trống phía trên thành 2 phần cho tới khi đạt vùng trống nhỏ nhất kích thước lớn hơn *n*

Ví dụ

- Vùng trống 16K Bytes
- Yêu cầu 735 Bytes



Buddy Allocation: Cung cấp nhớ

- Hệ thống duy trì các danh sách vùng trống kích thước 1, 2, . . . , 2^n bytes
- Với yêu cầu K, tìm phần tử nhỏ nhất kích thước lớn hơn K
- Nếu phần tử nhỏ nhất lớn hơn 2K, chia liên tiếp tới khi được vùng nhỏ nhất kích thước lớn hơn K
- Nhận xét: Với bộ nhớ kích thước *n*, cần duyệt $\log_2 n$ danh sách ⇒ Nhanh

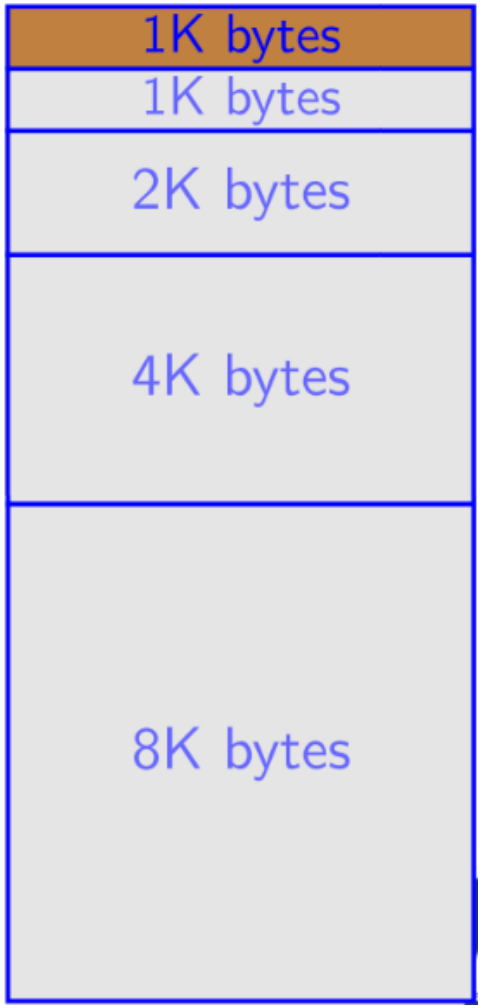
Ví dụ bộ nhớ 16K bytes

Buddy Allocation: Cung cấp nhớ

- Hệ thống duy trì các danh sách vùng trống kích thước 1, 2, . . . , 2^n bytes
- Với yêu cầu K, tìm phần tử nhỏ nhất kích thước lớn hơn K
- Nếu phần tử nhỏ nhất lớn hơn 2K, chia liên tiếp tới khi được vùng nhỏ nhất kích thước lớn hơn K
- Nhận xét: Với bộ nhớ kích thước *n*, cần duyệt $\log_2 n$ danh sách ⇒ Nhanh

Ví dụ bộ nhớ 16K bytes

- Yêu cầu 735 bytes

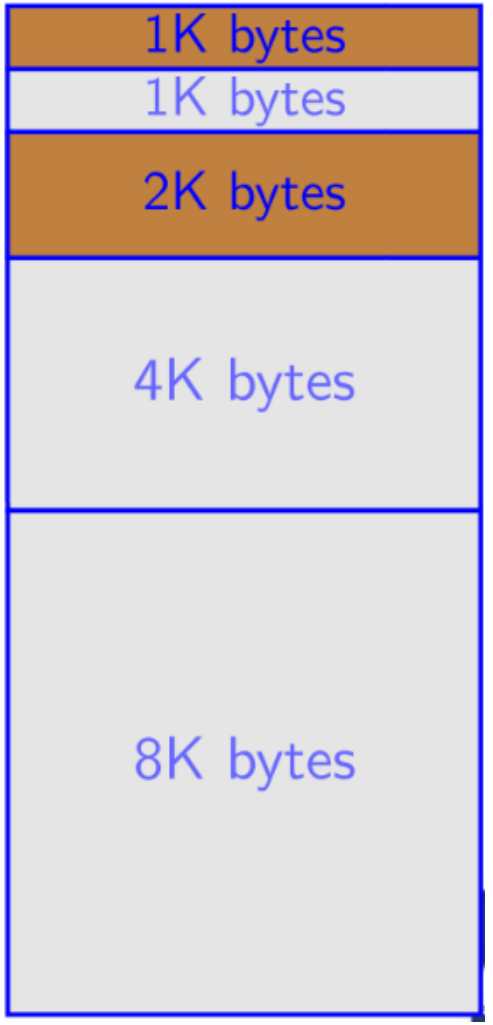


Buddy Allocation: Cung cấp nhớ

- Hệ thống duy trì các danh sách vùng trống kích thước 1, 2, . . . , 2^n bytes
- Với yêu cầu K, tìm phần tử nhỏ nhất kích thước lớn hơn K
- Nếu phần tử nhỏ nhất lớn hơn 2K, chia liên tiếp tới khi được vùng nhỏ nhất kích thước lớn hơn K
- Nhận xét: Với bộ nhớ kích thước *n*, cần duyệt $\log_2 n$ danh sách ⇒ Nhanh

Ví dụ bộ nhớ 16K bytes

- Yêu cầu 735 bytes
- Yêu cầu 1205 bytes

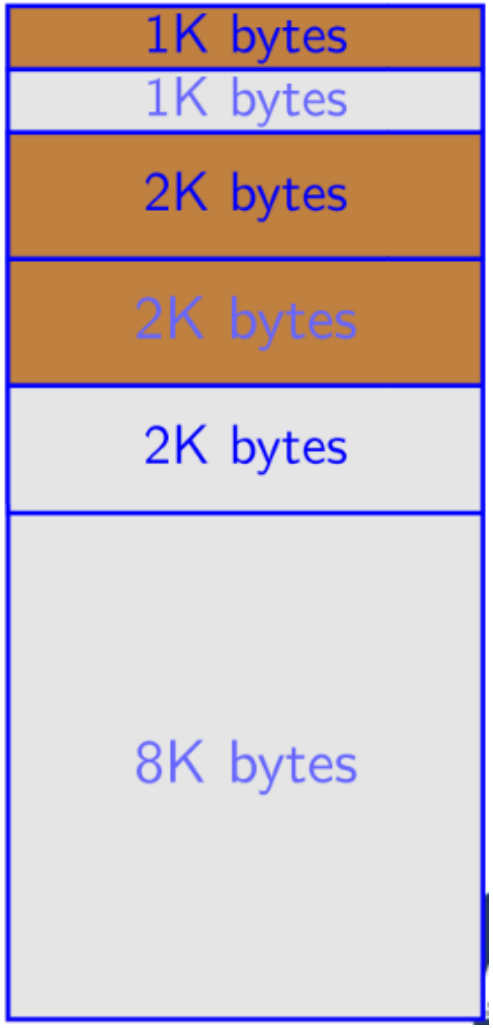


Buddy Allocation: Cung cấp nhớ

- Hệ thống duy trì các danh sách vùng trống kích thước 1, 2, . . . , 2^n bytes
- Với yêu cầu K, tìm phần tử nhỏ nhất kích thước lớn hơn K
- Nếu phần tử nhỏ nhất lớn hơn 2K, chia liên tiếp tới khi được vùng nhỏ nhất kích thước lớn hơn K
- Nhận xét: Với bộ nhớ kích thước *n*, cần duyệt $\log_2 n$ danh sách ⇒ Nhanh

Ví dụ bộ nhớ 16K bytes

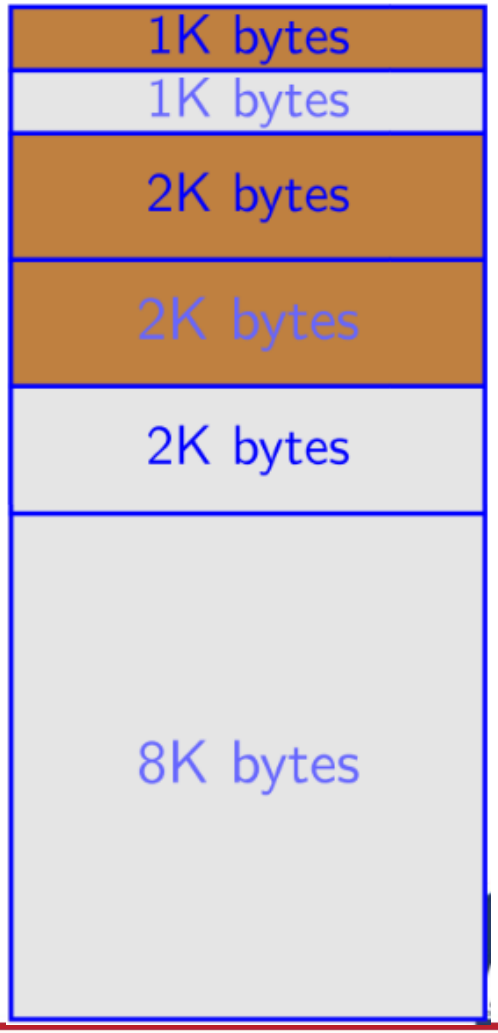
- Yêu cầu 735 bytes
- Yêu cầu 1205 bytes
- Yêu cầu 2010 bytes



Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

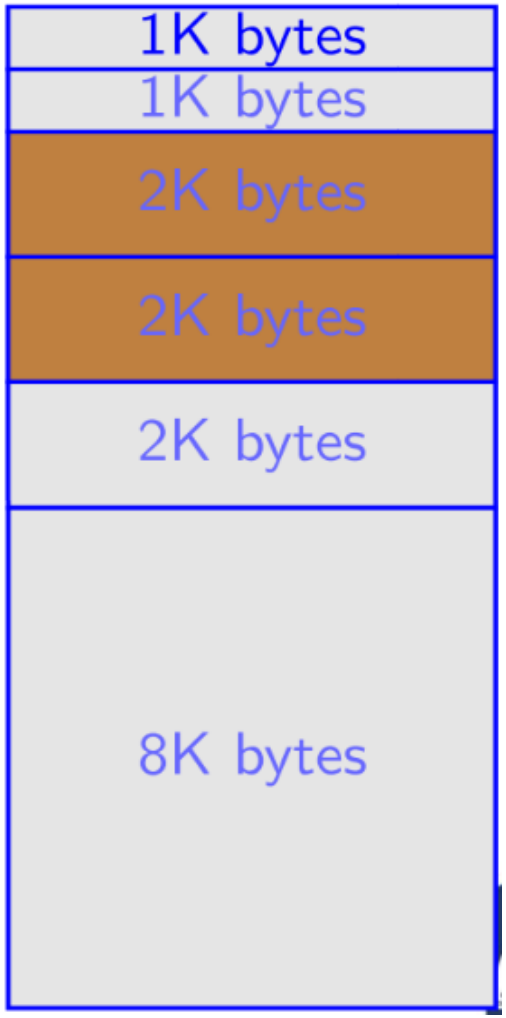


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)

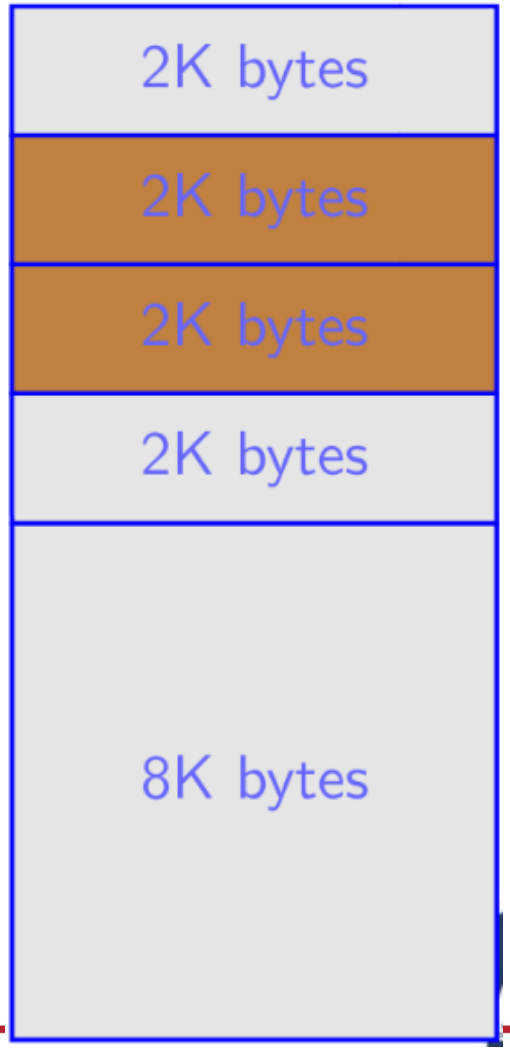


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K

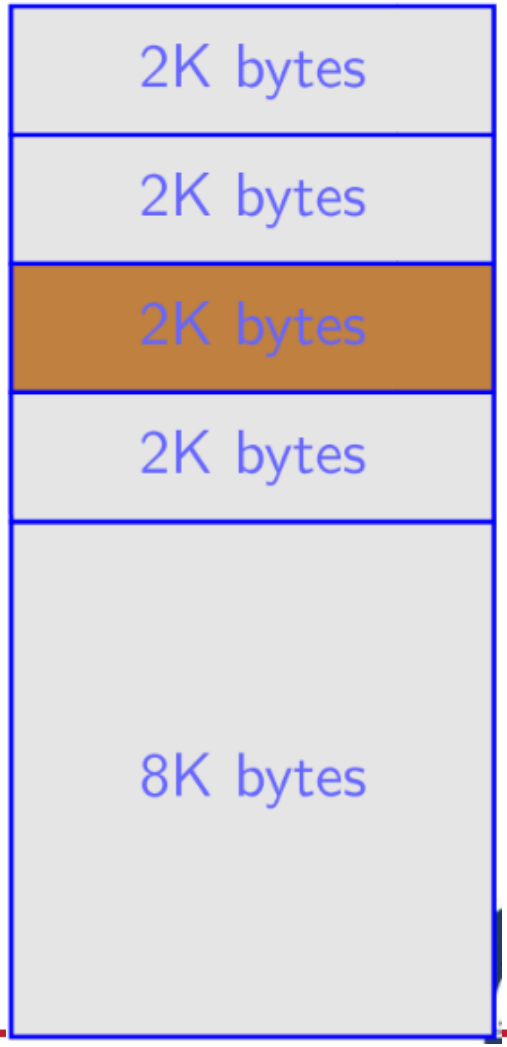


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)

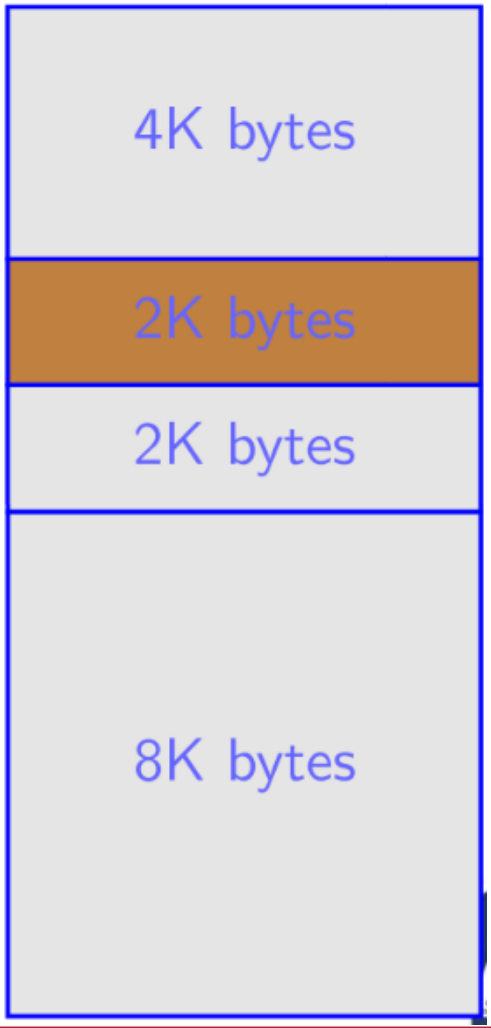


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K

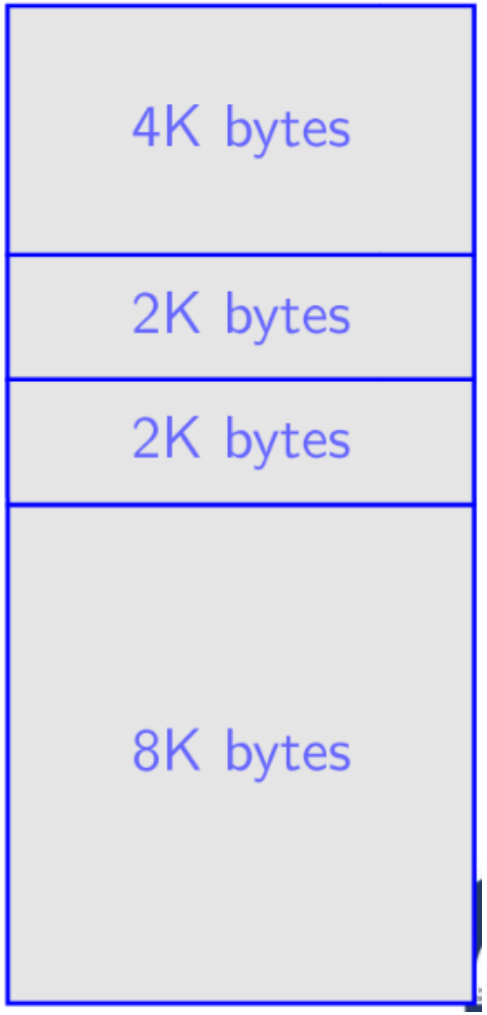


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
- Giải phóng vùng nhớ thứ ba (2K)

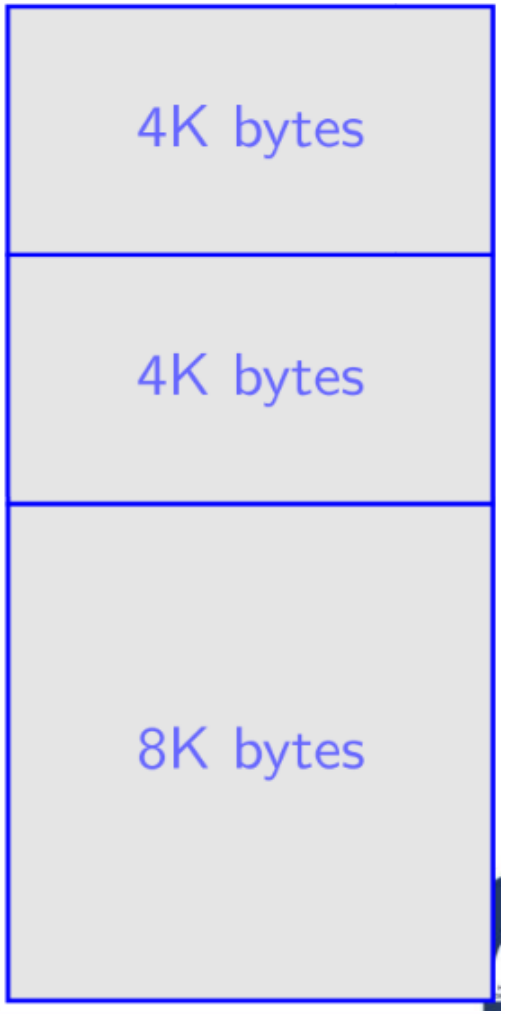


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
- Giải phóng vùng nhớ thứ ba (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K

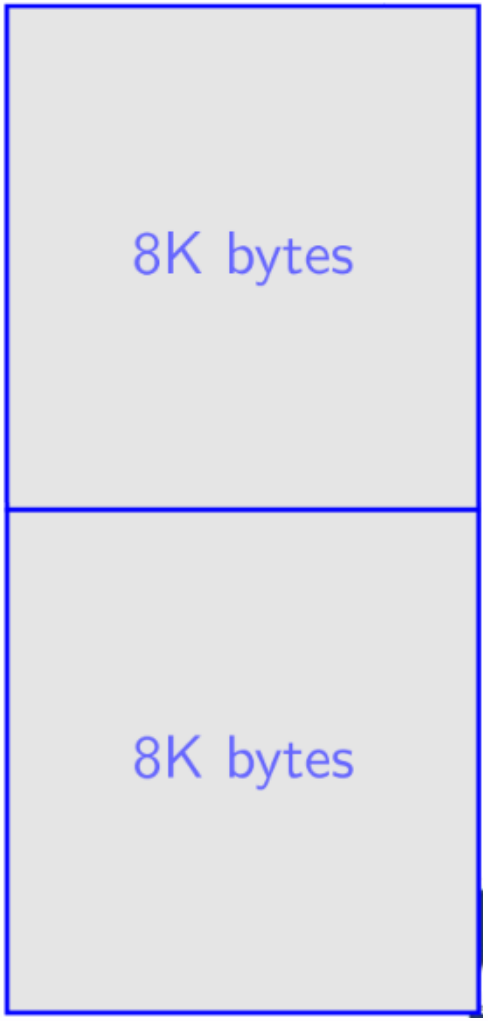


Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
- Giải phóng vùng nhớ thứ ba (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
  - Kết hợp 2 vùng 4K thành vùng 8K



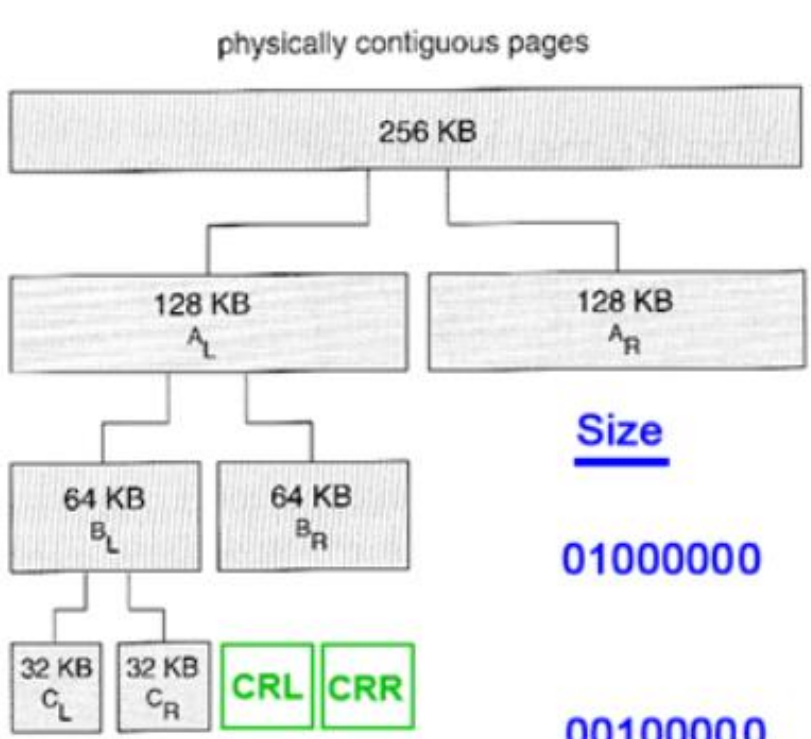
Buddy Allocation : Thu hồi vùng nhớ

- Có thể kết hợp 2 vùng kề nhau có cùng kích thước
- Tiếp tục kết hợp liên tiếp cho tới khi tạo ra vùng trống lớn nhất có thể

Ví dụ

- Giải phóng vùng nhớ thứ nhất (1K)
  - Kết hợp 2 vùng 1K thành vùng 2K
- Giải phóng vùng nhớ thứ hai (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
- Giải phóng vùng nhớ thứ ba (2K)
  - Kết hợp 2 vùng 2K thành vùng 4K
  - Kết hợp 2 vùng 4K thành vùng 8K
  - Kết hợp 2 vùng 8K thành vùng 16K

Buddy Allocation : Thu hồi vùng nhớ



Buddy system allocation.

Buddy Addresses

00000000

00000000 10000000

Size

01000000 00000000 01000000

00100000 00000000 00100000

0100000 0110000

Địa chỉ của hai khối nhớ chỉ khác nhau ở bit thứ *k*.

Vấn đề bố trí lại bộ nhớ

Sau một thời gian hoạt động, các vùng trống nằm rải rác khắp nơi gây ra hiện tượng thiếu bộ nhớ.
⇒Cần phải bố trí lại bộ nhớ

- Dịch chuyển các tiến trình

- Phương pháp tráo đổi (swapping)

Vấn đề bố trí lại bộ nhớ

- Dịch chuyển các tiến trình
  - Vấn đề không đơn giản vì các đối tượng bên trong khi chuyển sang vị trí mới sẽ có địa chỉ khác
    - Sử dụng thanh ghi dịch chuyển (relocation register) chứa giá trị bằng độ dịch chuyển của tiến trình
  - Vấn đề lựa chọn phương pháp để chi phí nhỏ nhất
    - tất cả về một phía ⇒ vùng trống lớn nhất
    - Dịch chuyển để tạo ra ngay lập tức một vùng trống vừa vặn
- Phương pháp tráo đổi (swapping)
  - Lựa chọn thời điểm dừng tiến trình đang thực hiện
  - Đưa tiến trình và trạng thái tương ứng ra bên ngoài
    - Giải phóng vùng nhớ để kết hợp với các phần tử liền kề
  - Tái định vị vào vị trí cũ và khôi phục trạng thái cũ
    - Dùng thanh ghi dịch chuyển nếu đưa vào vị trí khác

Nhận xét

- Không phải sao lưu modul điều khiển ra nhiều nơi
- Tăng/giảm hệ số song song tùy theo số lượng và kích thước chương trình
- Không thực hiện được chương trình có kích thước lớn hơn kích thước bộ nhớ vật lý
- Gây ra hiện tượng rác
  - Bộ nhớ không được sử dụng, nhưng cũng không nằm trong DS quản lý bộ nhớ tự do
    - Do lỗi hệ điều hành
    - Do phần mềm phá hoại
- Gây ra hiện tượng phân đoạn ngoài
  - Vùng nhớ tự do được quản lý đầy đủ, nhưng nằm rải rác nên không sử dụng được
- Gây ra hiện tượng phân đoạn trong
  - Vùng nhớ dành cho chương trình nhưng không được chương trình sử dụng tới

- Chiến lược phân chương cố định
- Chiến lược phân chương động
- **Chiến lược phân đoạn**
- Chiến lược phân trang
- Chiến lược kết hợp phân đoạn-phân trang

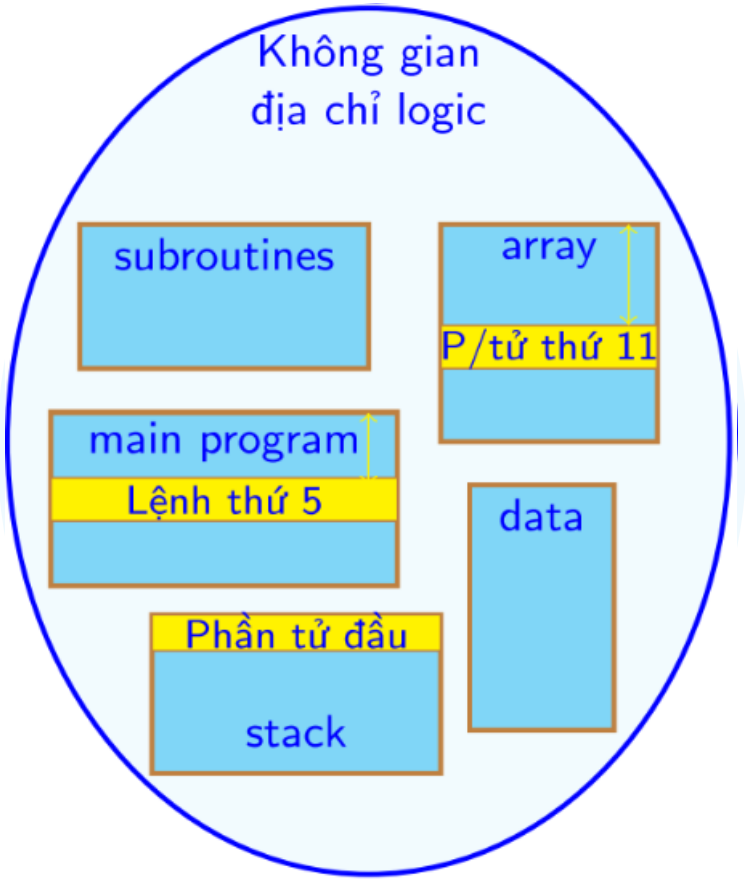
Chương trình

- Chương trình thường gồm các modul
  - 1 chương trình chính (main program)
  - Tập các chương trình con
  - Các biến, các cấu trúc dữ liệu,. . .
- Các modul, đối tượng trong c/trình được xác định bằng tên
  - Hàm sqrt(), thủ tục printf() . . .
  - x, y, counter, Buffer. . .
- Các p/tử trong modul được x/định theo độ lệch với vị trí đầu
  - Câu lệnh thứ 10 của hàm sqrt(). . .
  - Phần tử thứ 2 của mảng Buffer. . .

Chương trình được tổ chức như thế nào trong bộ nhớ?
- Stack nằm trên hay Data nằm trên trong bộ nhớ?
- Địa chỉ vật lý các đối tượng . . .?

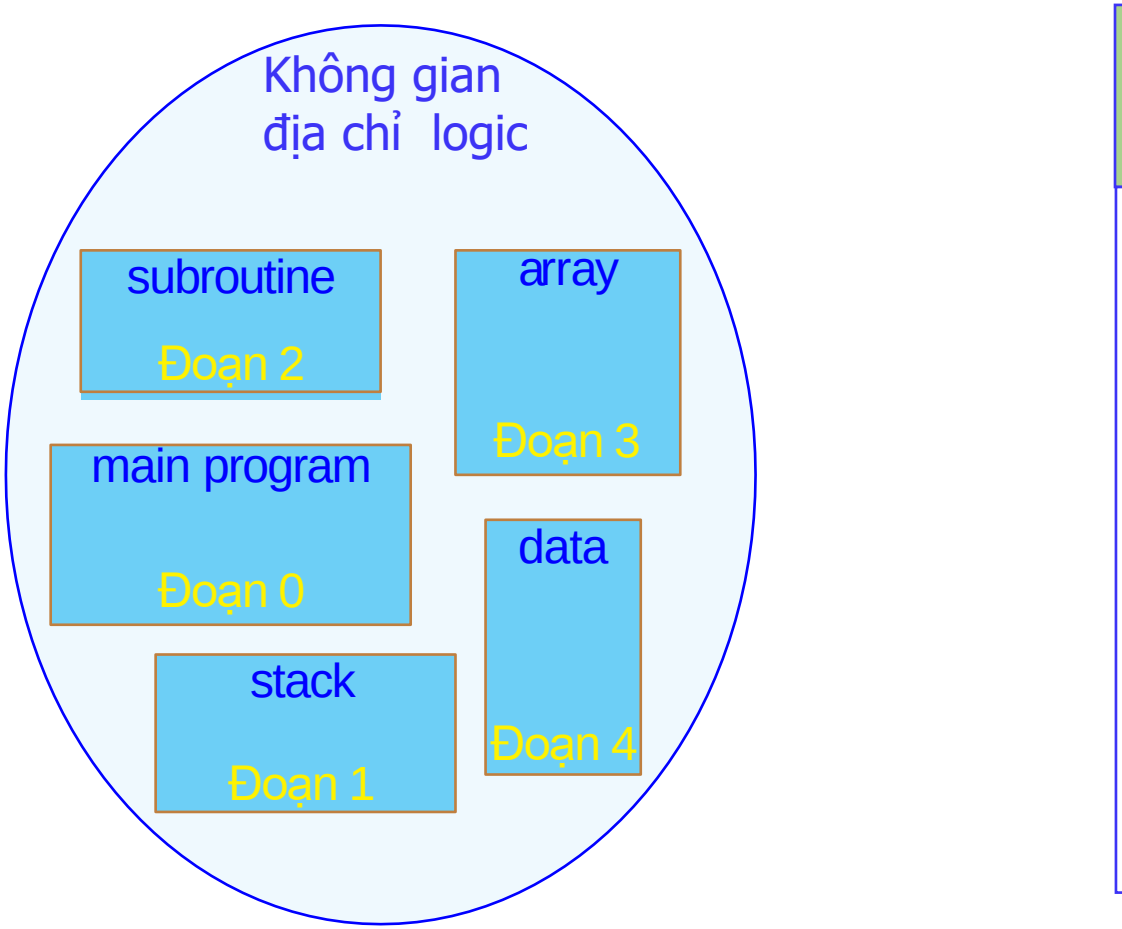
⇒Không quan tâm

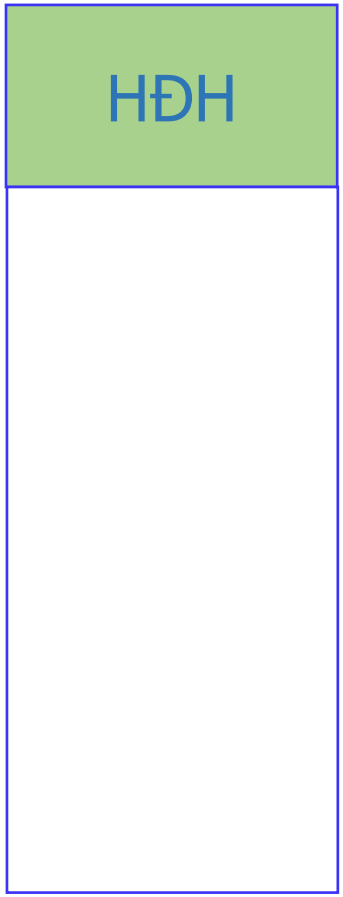
Quan điểm người dùng



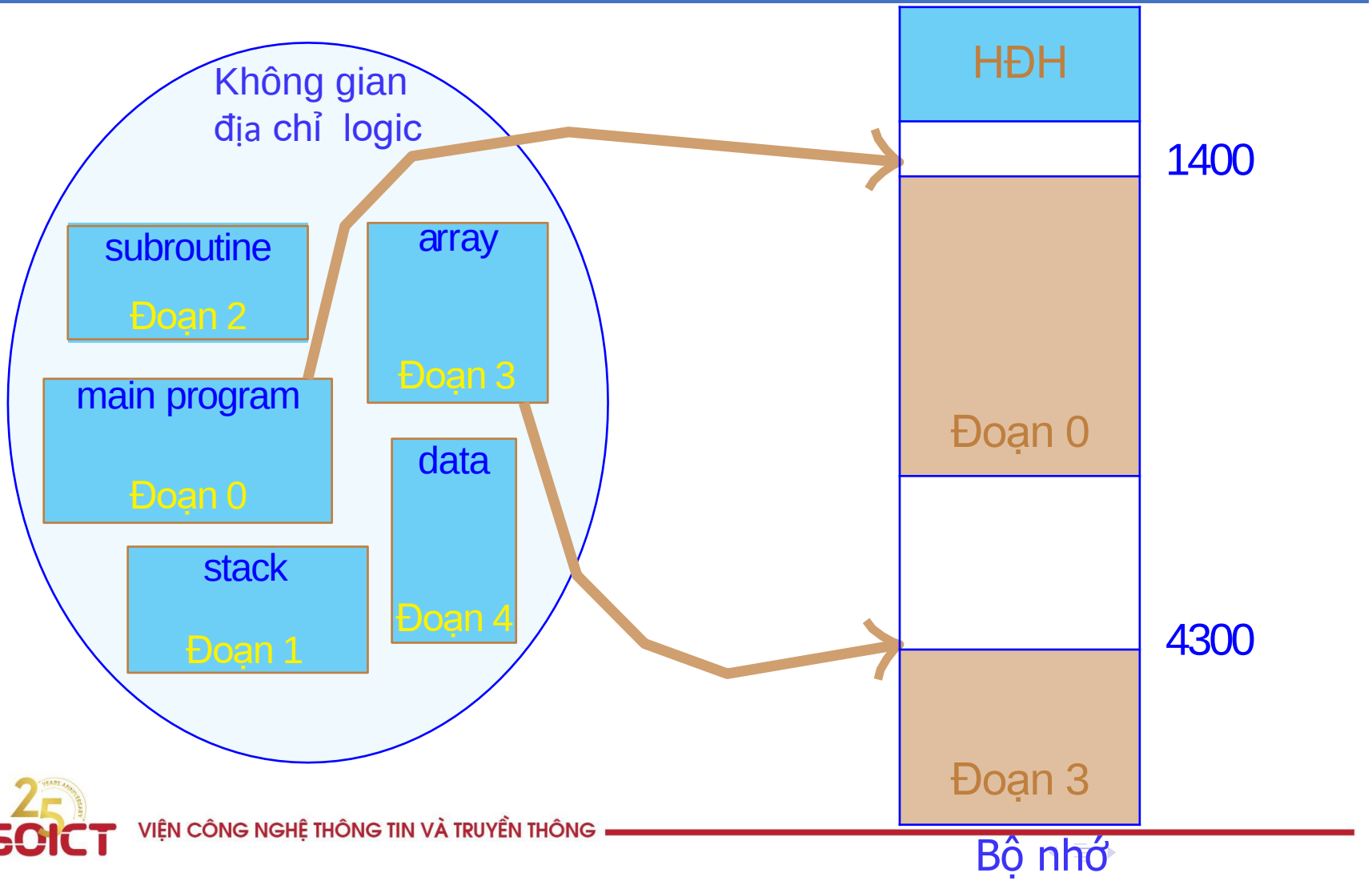
- Khi đưa c/trình vào bộ nhớ để thực hiện C/trình gồm nhiều đoạn khác nhau
- Mỗi đoạn là một khối logic, ứng với một modul
  - Mã lệnh: main(), thủ tục, hàm. . .
  - Dữ liệu: Đối tượng toàn cục, cục bộ
  - Các đoạn khác: stack, mảng. . .
- Mỗi đoạn chiếm một vùng liên tục
  - Có vị trí bắt đầu và kích thước
  - Có thể nằm tại bất cứ đâu trong bộ nhớ
- Đối tượng trong đoạn được xác định bởi vị trí tương đối so với đầu đoạn
  - Lệnh thứ 5 của chương trình chính
  - Phần tử đầu tiên của stack. . .
- Vị trí các đối tượng trong bộ nhớ?

Ví dụ

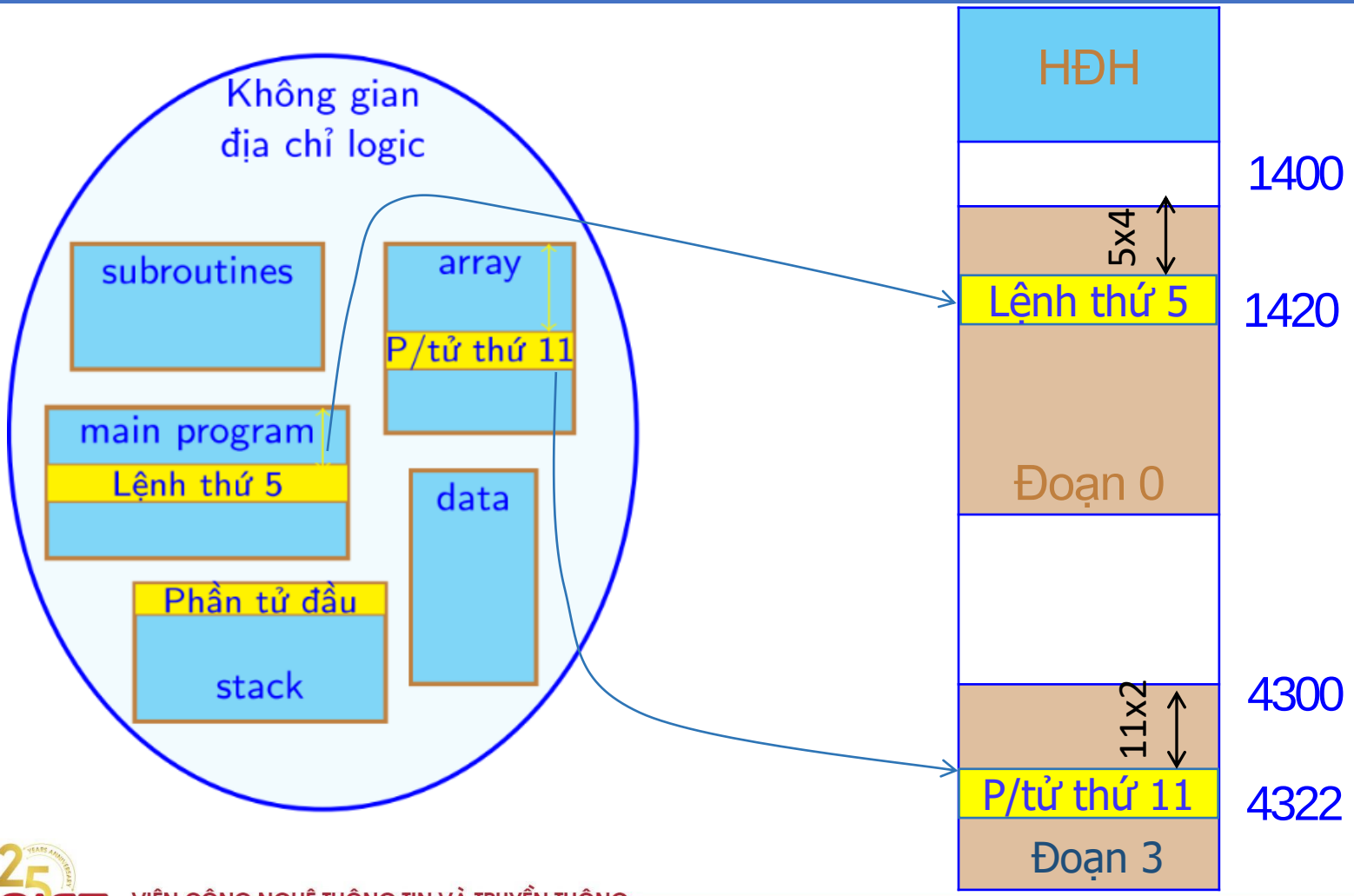

Không gian
địa chỉ logic
subroutine
Đoạn 2
array
Đoạn 3
main program
Đoạn 0
data
Đoạn 4
stack
Đoạn 1


HĐH
Bộ nhớ

Ví dụ


Không gian
địa chỉ logic
subroutine
Đoạn 2
array
Đoạn 3
main program
Đoạn 0
data
Đoạn 4
stack
Đoạn 1
HĐH
1400
Đoạn 0
4300
Đoạn 3
Bộ nhớ

Ví dụ


Không gian
địa chỉ logic
subroutines
array
P/tử thứ 11
main program
Lệnh thứ 5
data
Phần tử đầu
stack
HĐH
1400
5x4
Lệnh thứ 5
1420
Đoạn 0
4300
11x2
P/tử thứ 11
4322
Đoạn 3

Cấu trúc phân đoạn

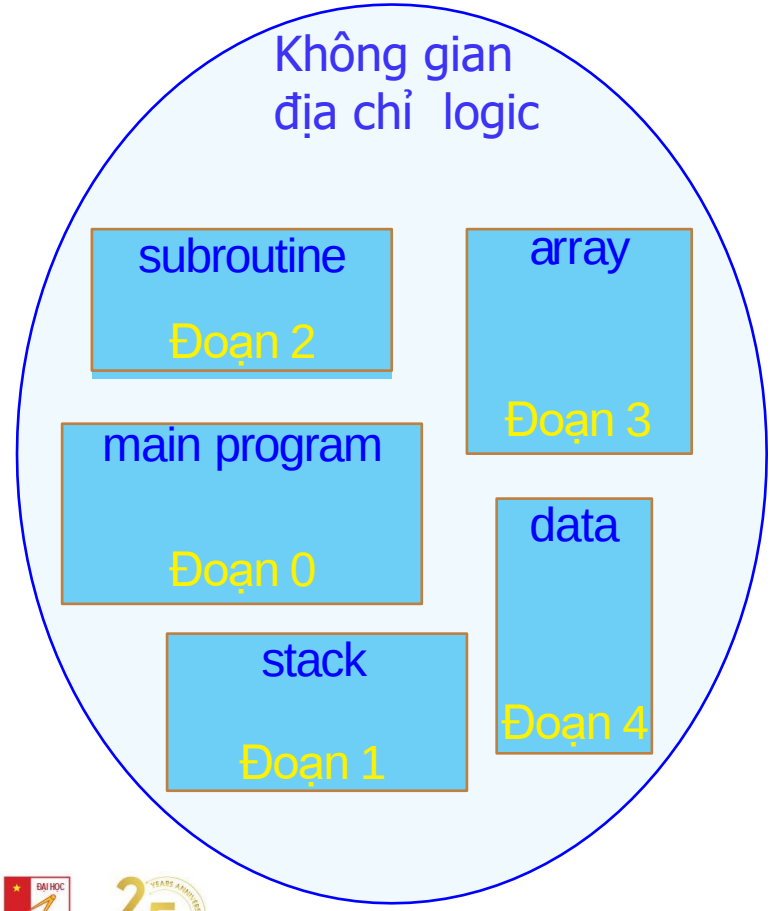
- Chương trình là tập hợp các đoạn (modul,segment)
  - Tên đoạn (*số hiệu đoạn*), độ dài của đoạn
  - Mỗi đoạn có thể được biên tập riêng.
- Dịch và biên tập chương trình tạo ra bảng quản lý đoạn
- (SCB: Segement Control Block)
- Mỗi phần tử của bảng ứng với một đoạn của chương trình

	Mark	Address	Length
0			
...			
n	⋮	⋮	⋮

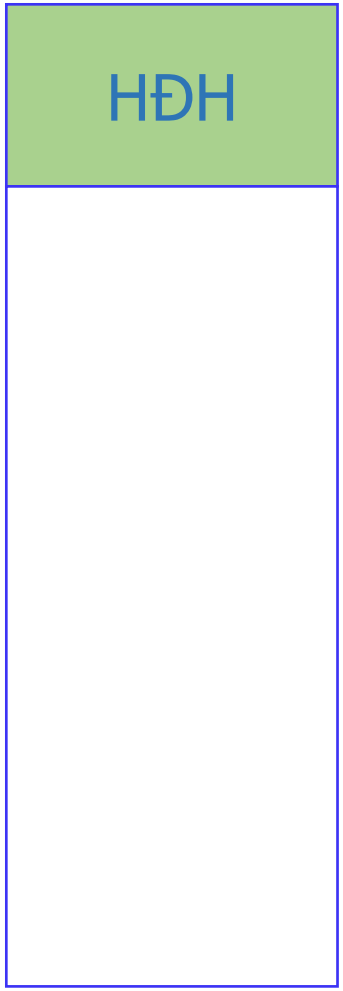
  - Dấu hiệu (Mark(0/1)): Đoạn đã tồn tại trong bộ nhớ
  - Địa chỉ (Address): Vị trí cơ sở (base) của đoạn trong bộ nhớ
  - Độ dài (Length): Độ dài của đoạn
- Địa chỉ truy nhập: tên (số hiệu) đoạn và độ lệch trong đoạn

Vấn đề: Chuyển đổi từ địa chỉ 2 chiều ⇒ địa chỉ một chiều

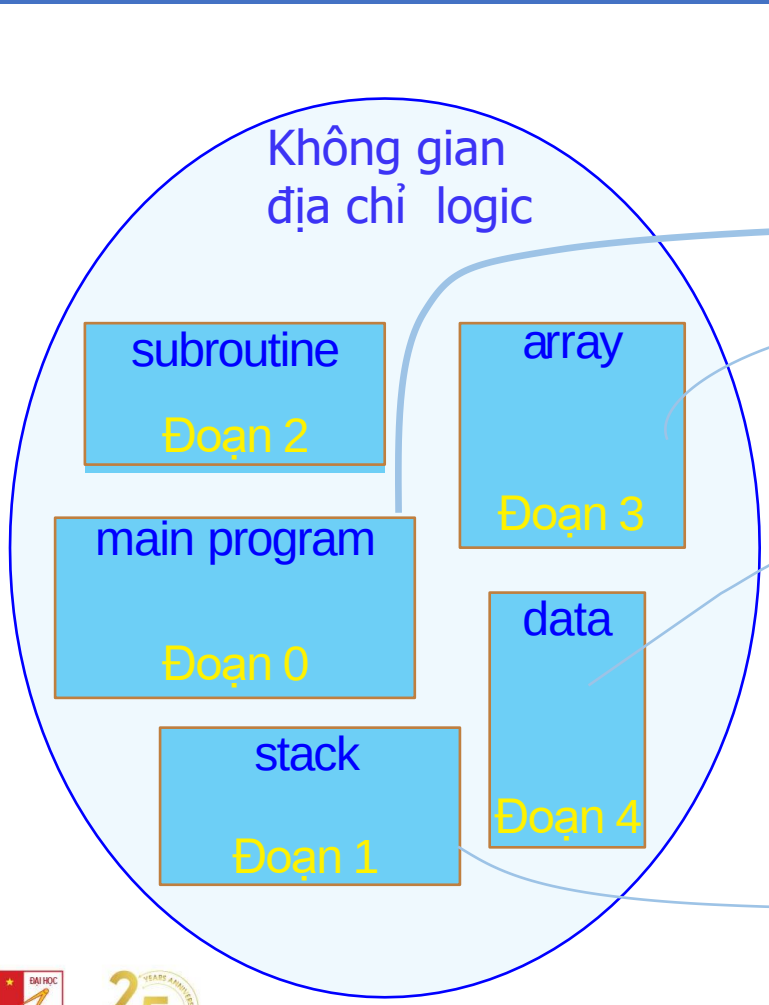
Ví dụ



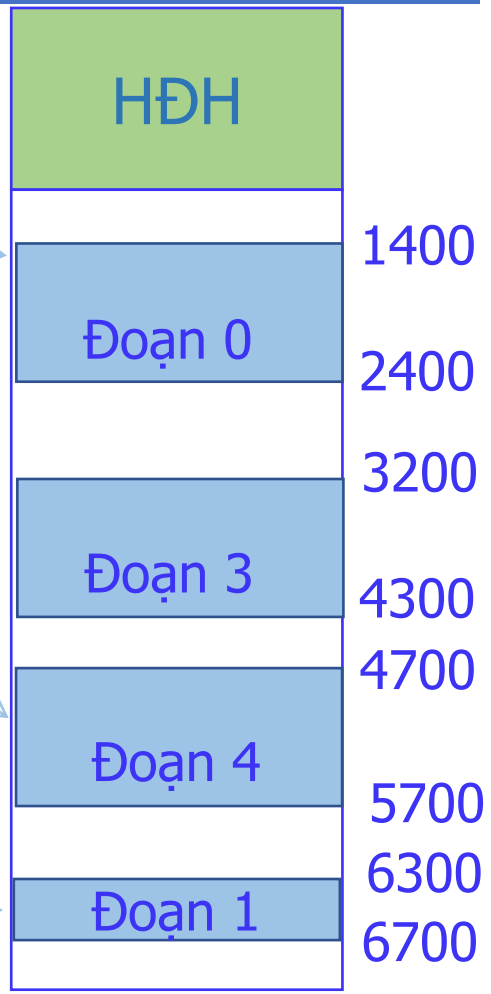
M	*A*	*L*
0	-	1000
0	-	400
0	-	400
0	-	1100
0	-	1000
SCB		



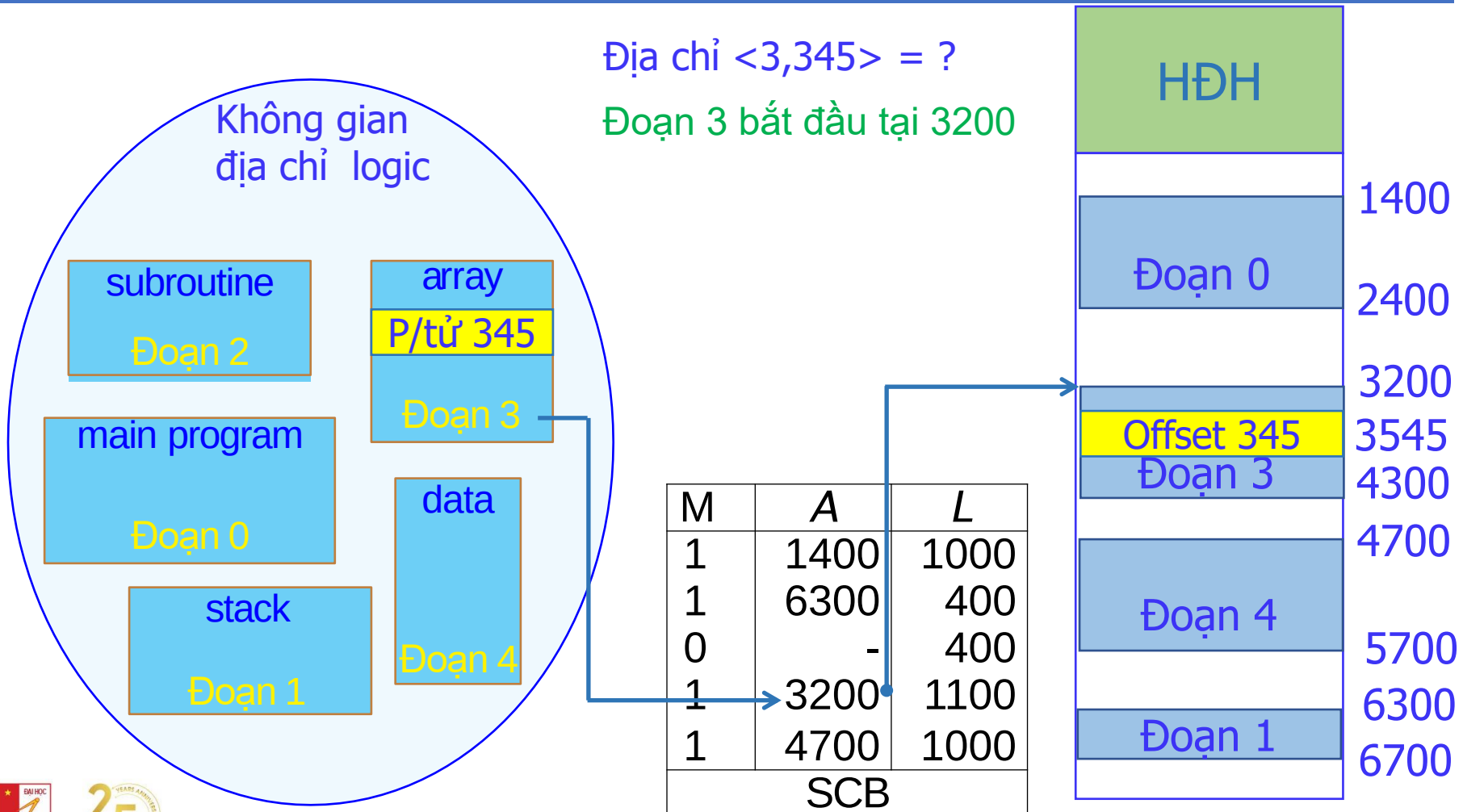
Ví dụ



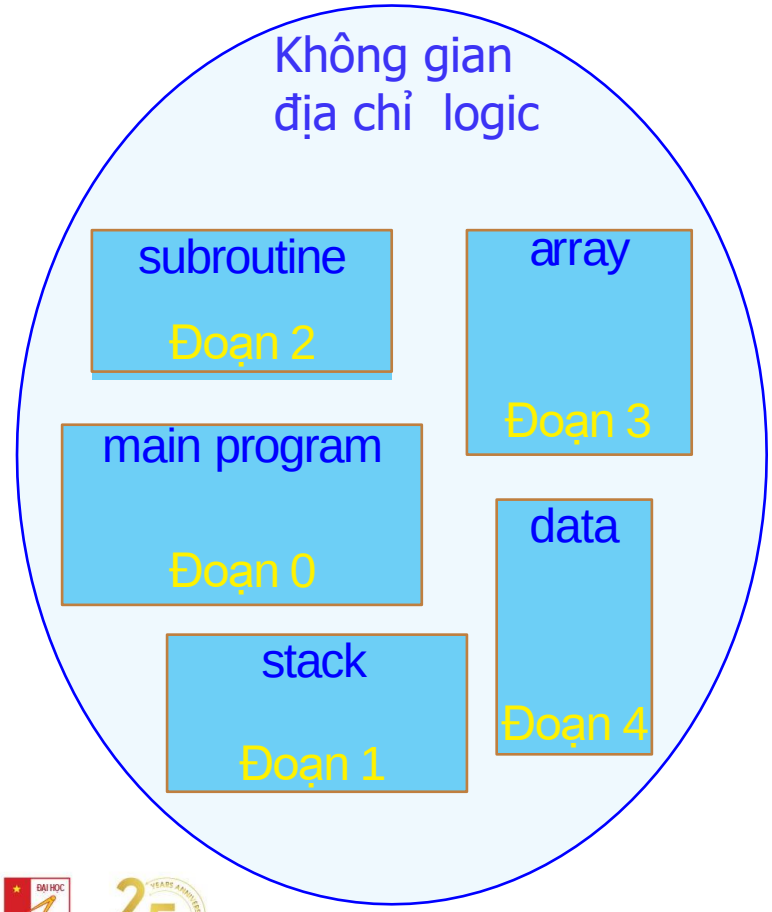
M	*A*	*L*
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		



Ví dụ

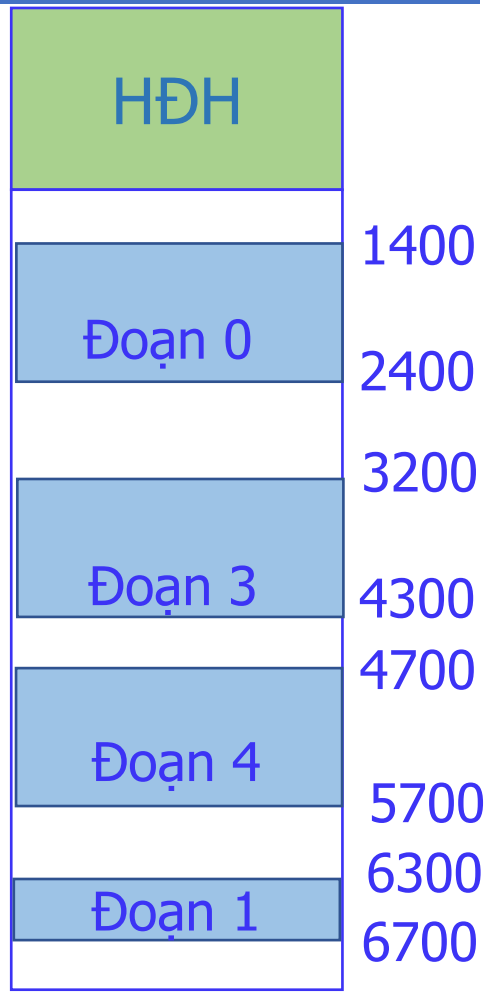

Địa chỉ <3,345> = ?
Đoạn 3 bắt đầu tại 3200
Không gian địa chỉ logic
subroutine
Đoạn 2
array
P/tử 345
Đoạn 3
main program
Đoạn 0
data
Đoạn 4
stack
Đoạn 1
M A L
1 1400 1000
1 6300 400
0 - 400
1 3200 1100
1 4700 1000
SCB
HĐH
1400
Đoạn 0
2400
3200
Offset 345
3545
Đoạn 3
4300
4700
Đoạn 4
5700
6300
Đoạn 1
6700
Bộ nhớ

Ví dụ

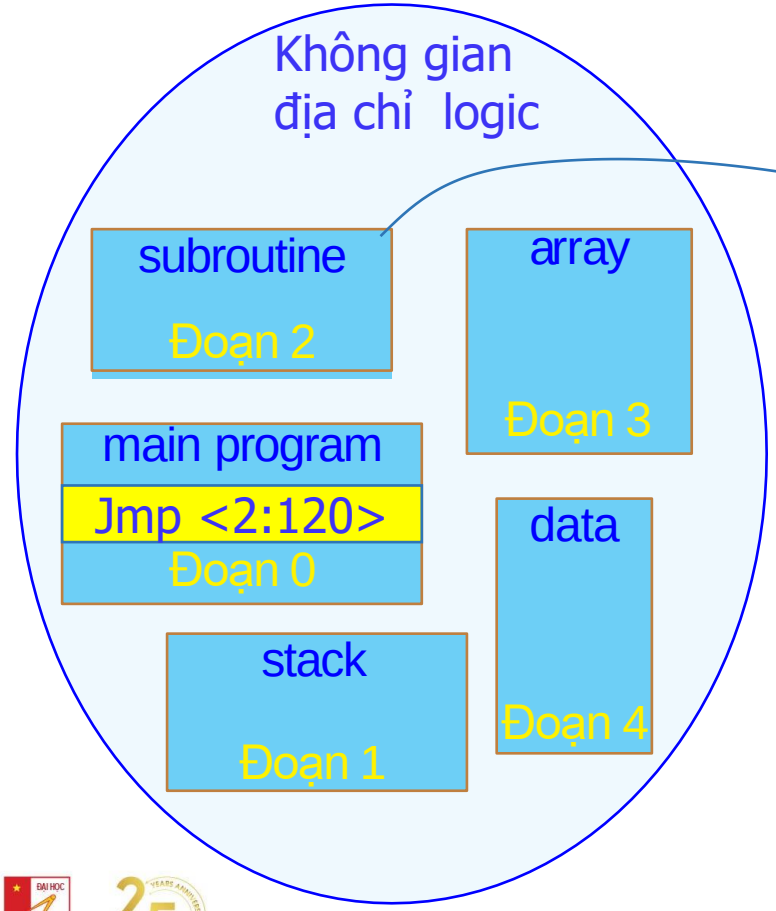


Địa chỉ <4,185> = 4885

M	*A*	*L*
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

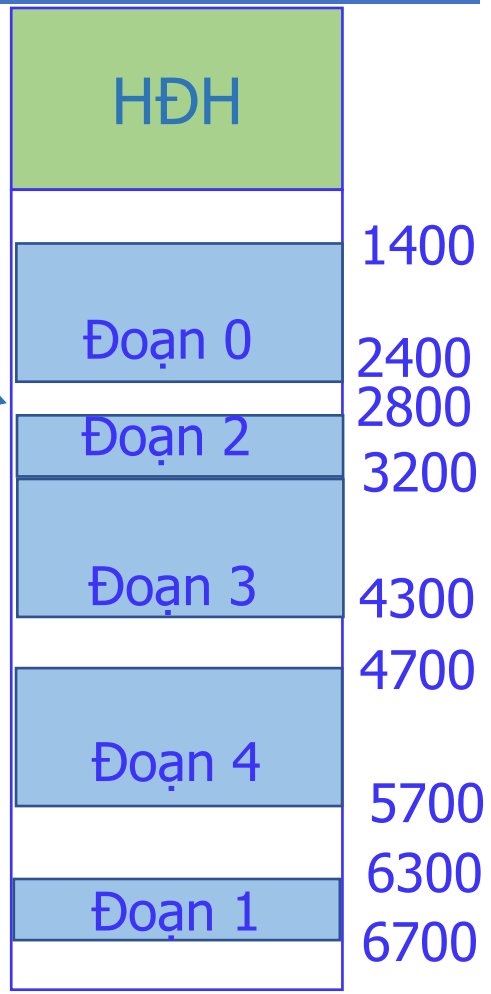


Ví dụ

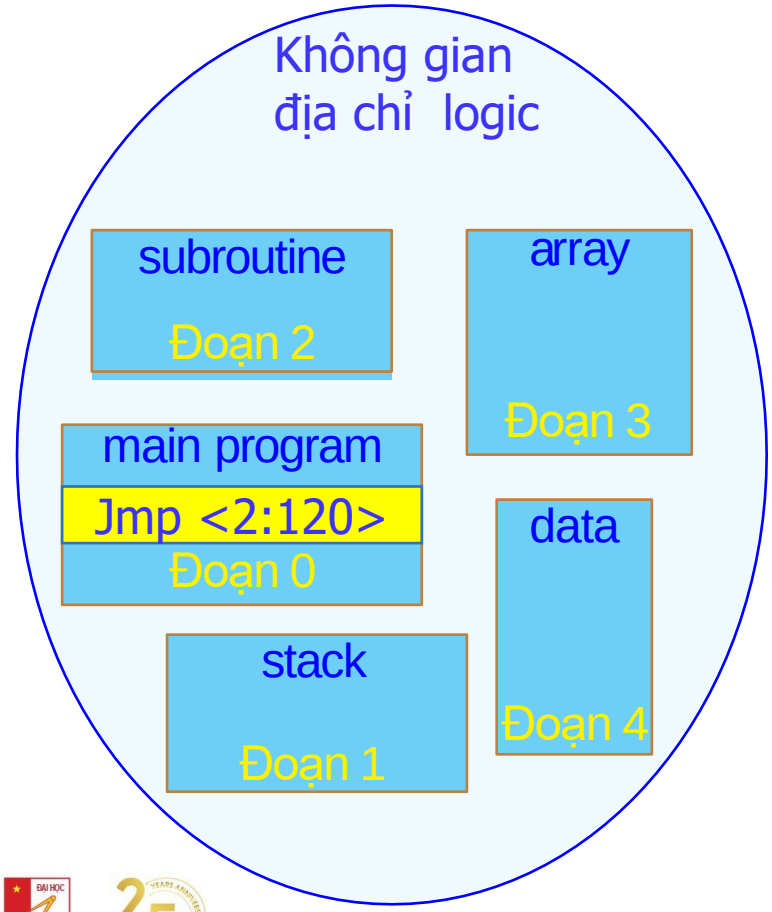


Địa chỉ <2,120> = ?

M	*A*	*L*
1	1400	1000
1	6300	400
0	-	400
1	3200	1100
1	4700	1000
SCB		

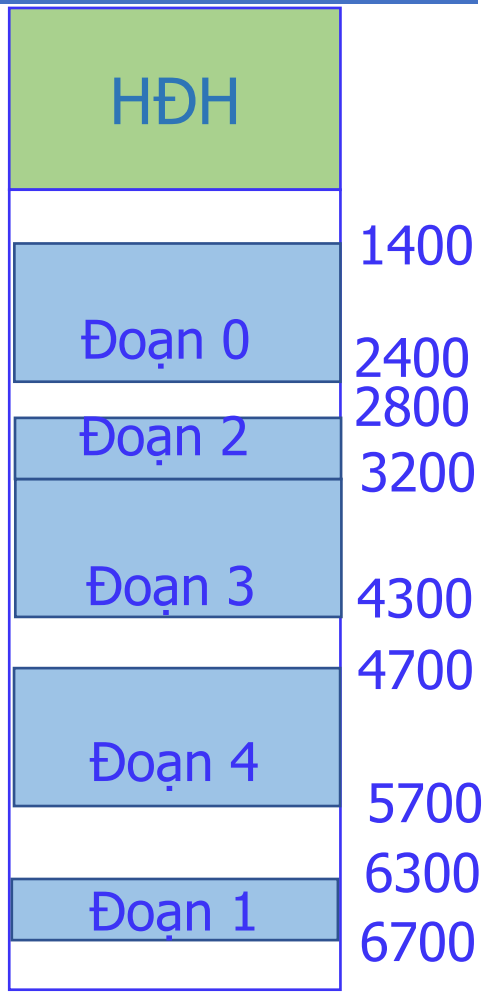


Ví dụ

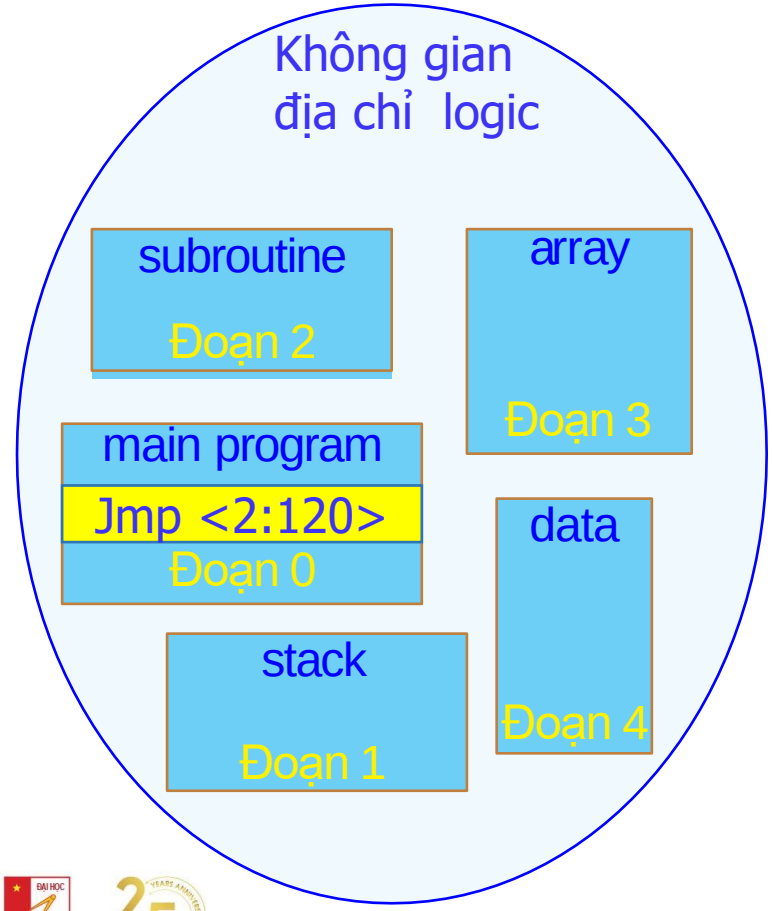


Địa chỉ <2,120> = ?

M	*A*	*L*
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		

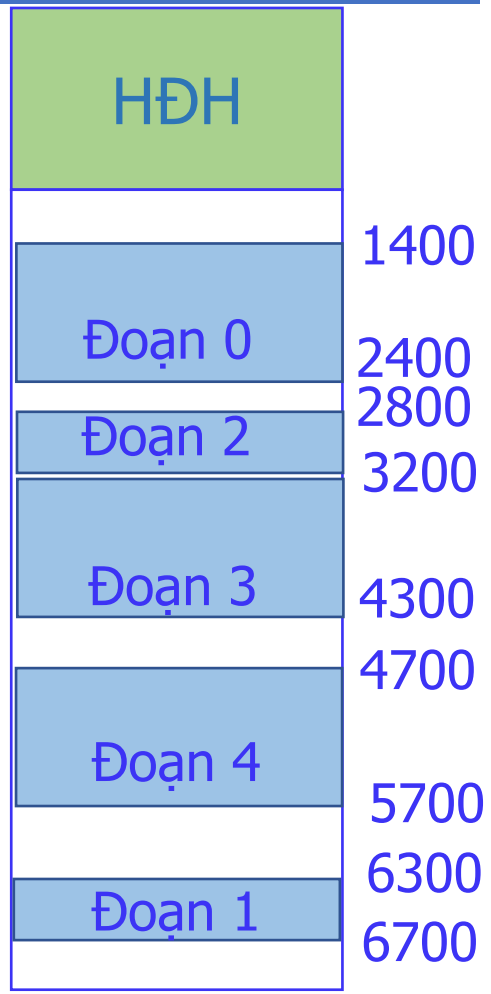


Ví dụ

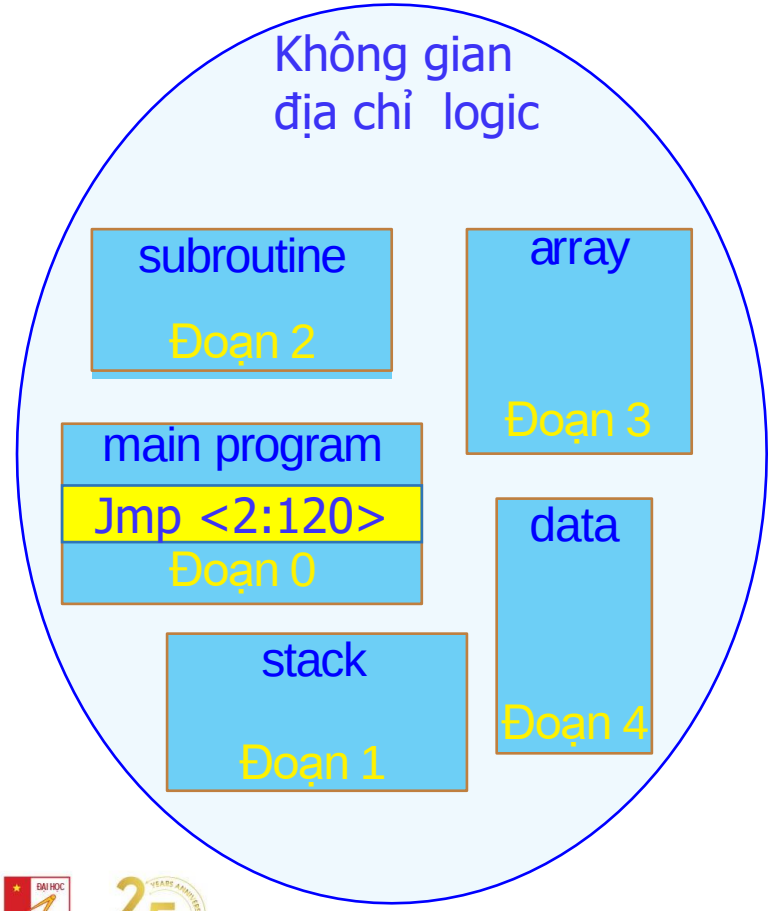


Địa chỉ <2,120> = 2920

M	*A*	*L*
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



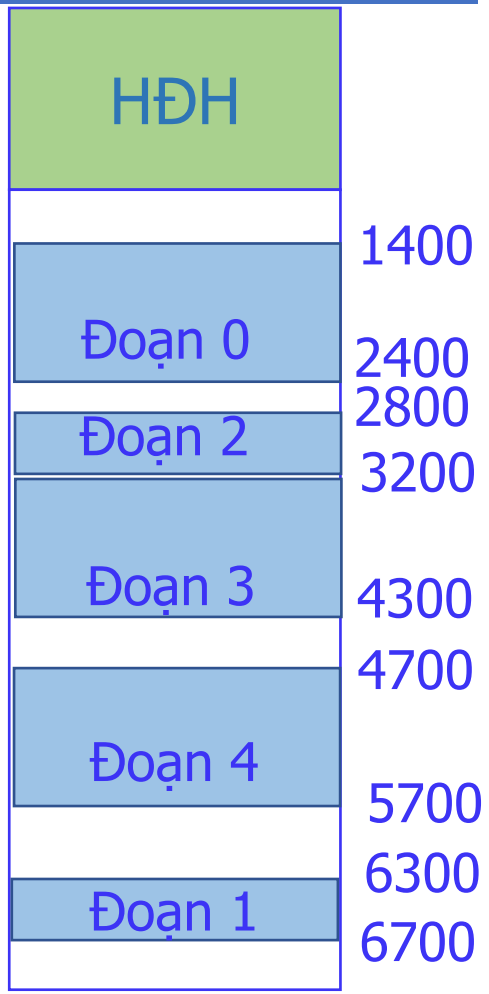
Ví dụ



Địa chỉ <2,450> = ?

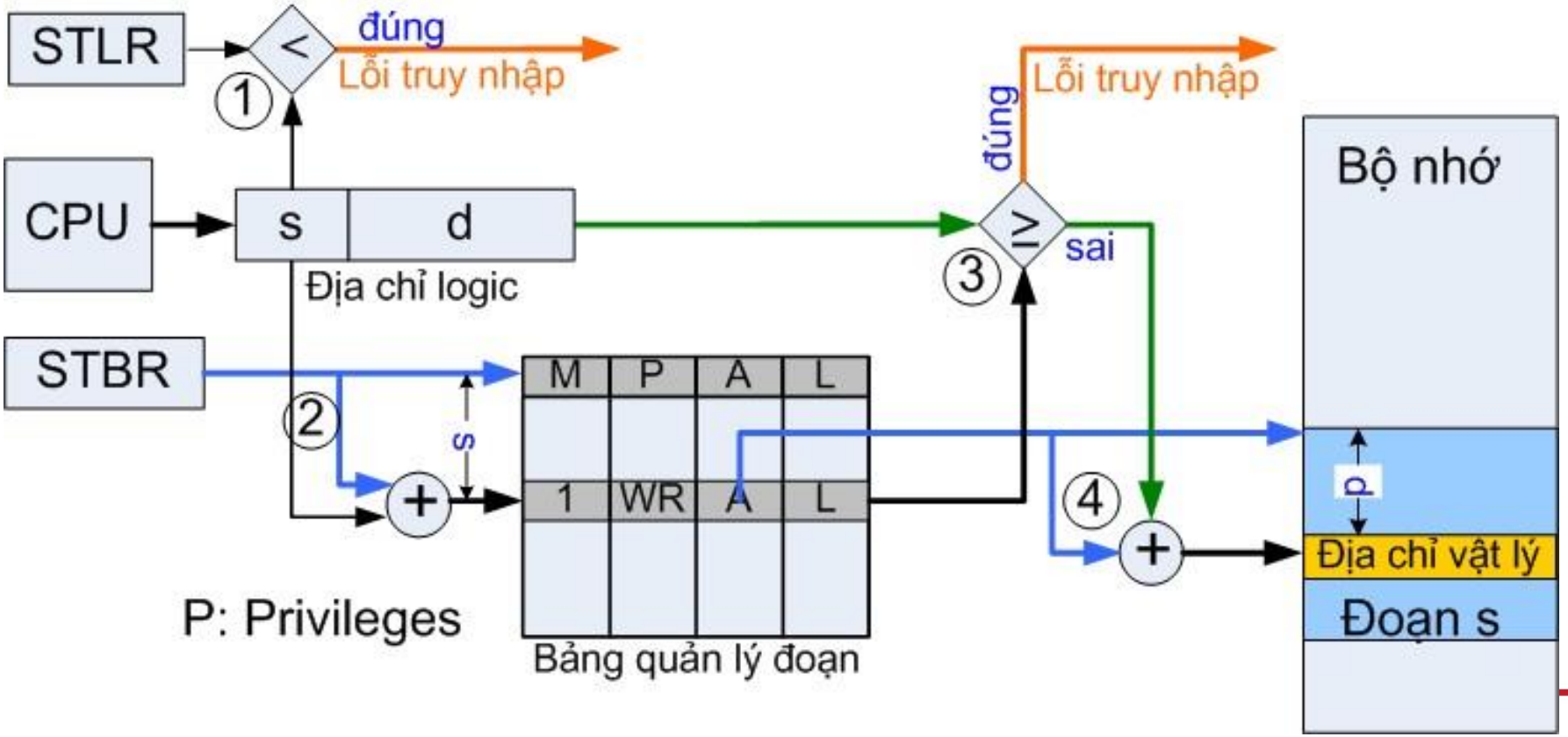
Lỗi truy nhập!

M	*A*	*L*
1	1400	1000
1	6300	400
1	2800	400
1	3200	1100
1	4700	1000
SCB		



Chuyển đổi địa chỉ: Sơ đồ truy nhập

- Khi thực hiện chương trình
  - Bảng quản lý đoạn được nạp vào bộ nhớ
    - STBR (Segment-table base register): Vị trí SCB trong bộ nhớ
    - STLR (Segment-table length register): Số phần tử của SCB



Chuyển đổi địa chỉ

- Truy nhập tới địa chỉ logic < s, d >

① s ≥ STLR : Lỗi

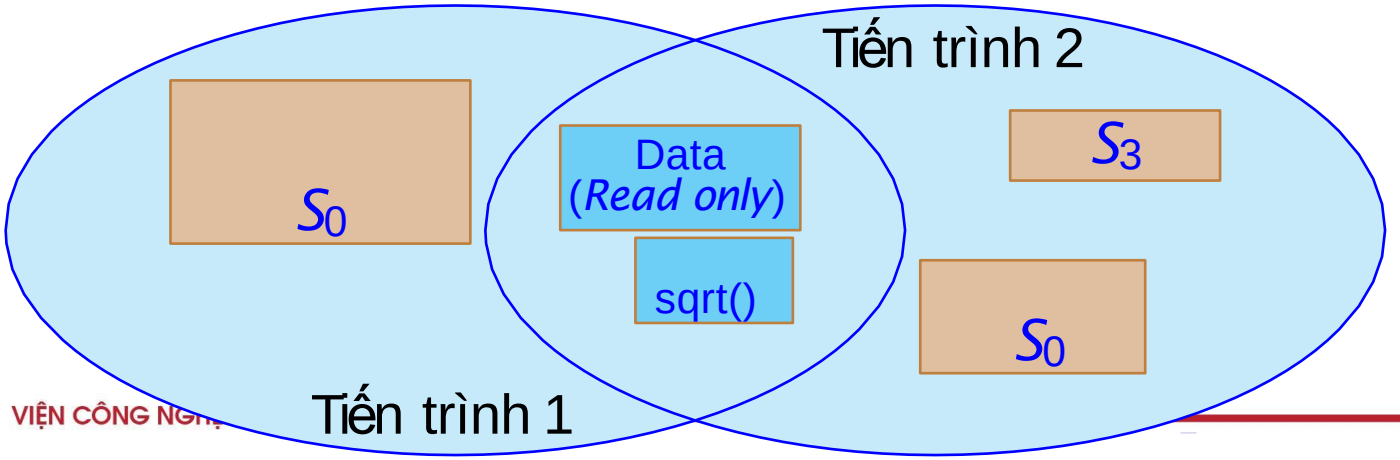
② STBR + s * K : Vị trí phần tử s trong SCB

③ Kiểm tra trường dấu hiệu M của phần tử SCBs

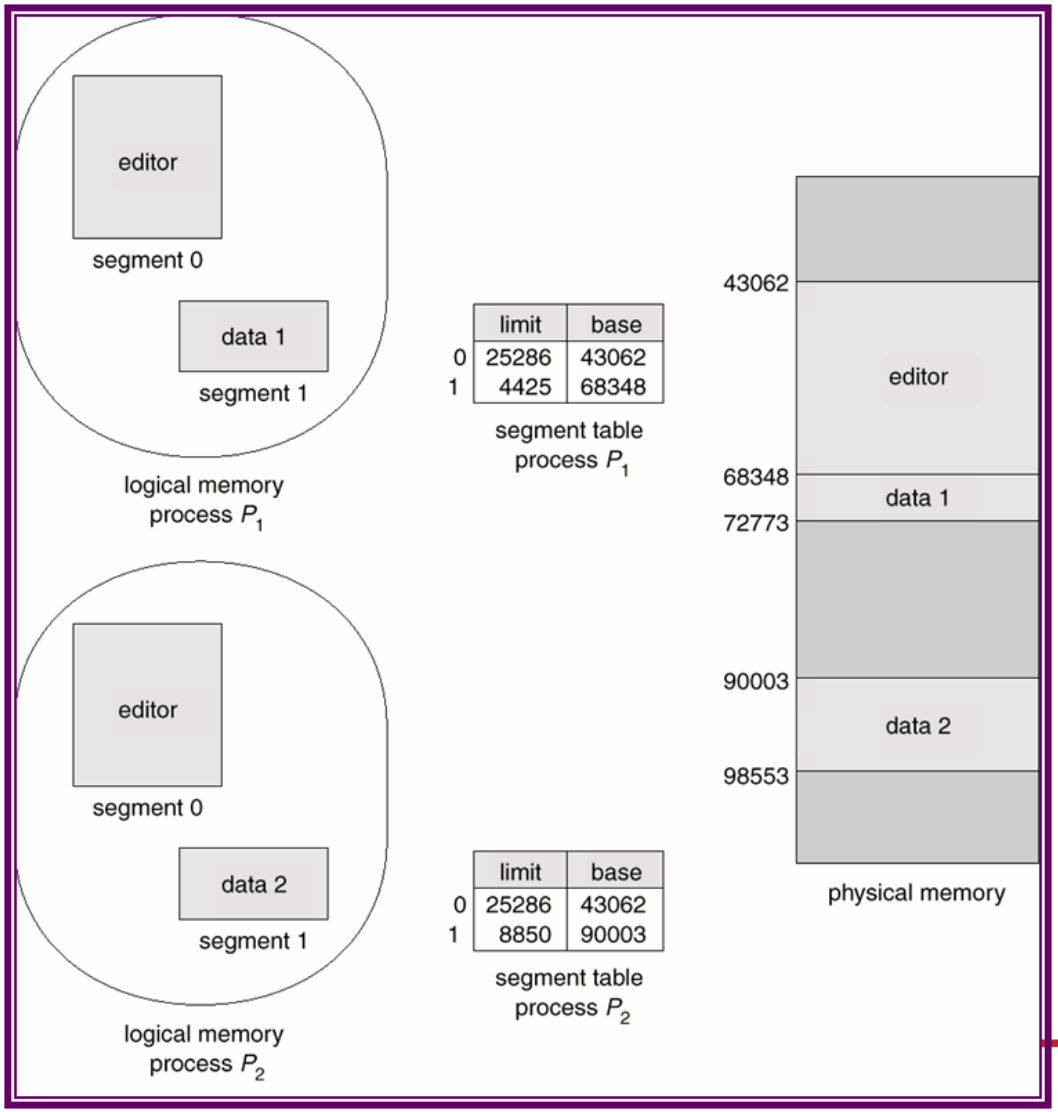
- M = 0: Đoạn s chưa tồn tại trong bộ nhớ ⇒ Lỗi truy nhập ⇒ HĐH phải nạp đoạn
  - Xin vùng nhớ có kích thước được ghi trong trường L
  - Tìm modul tương ứng ở bộ nhớ ngoài và nạp và định vị vào vùng nhớ xin được
  - Sửa lại trường địa chỉ A và trường dấu hiệu M(M = 1)
  - Truy nhập bộ nhớ như trường hợp không gặp lỗi truy nhập
- M = 1 :Đoạn s đã tồn tại trong bộ nhớ
  - d ≥ Ls: Lỗi truy nhập (vượt quá kích thước đoạn)
  - d + As: Địa chỉ vật lý cần tìm

Nhận xét: ưu điểm

- Sơ đồ nạp modul không cần sự tham gia của người sử dụng
- Dễ dàng thực hiện nhiệm vụ bảo vệ đoạn
  - Kiểm tra lỗi truy nhập bộ nhớ
    - Địa chỉ không hợp lệ :vươt quá kích thước đoạn
  - Kiểm tra tính chất truy nhập
    - Đoạn mã: chỉ đọc -> Viết vào đoạn mã: lỗi truy nhập
  - Kiểm tra quyền truy nhập modul
    - Thêm trường quyền truy nhập(user/system) vào SCB
- Cho phép sử dụng chung đoạn (VD Soạn thảo văn bản)



Dùng chung đoạn : Vấn đề chính



- Đoạn dùng chung phải cùng
- số hiệu trong SCB
  - Call (0, 120) ?
  - Read (1, 245) ?
- Giải quyết bằng cách truy nhập
- gián tiếp
  - JMP + 08
  - Thanh ghi đoạn chứa số hiệu đoạn (ES:BX)

Nhận xét : Nhược điểm

- Hiệu quả sử dụng phụ thuộc vào cấu trúc chương trình
- Bị phân mảnh bộ nhớ
  - Phân phối vùng nhớ theo các chiến lược first fit /best fit...
  - Cần phải bố trí lại bộ nhớ (dịch chuyển, swapping)
    - Có thể dựa vào bảng SCB
      - M ← 0 : Đoạn chưa được nạp vào
      - Vùng nhớ được xác định bởi A và L được trả về DS tự do
    - Vấn đề lựa chọn modul cần đưa ra
      - Đưa ra modul tồn tại lâu nhất
      - Đưa ra modul có lần sử dụng cuối cách xa nhất
      - Đưa ra modul có tần xuất sử dụng thấp nhất ⇒Cần phương tiên ghi lại số lần và thời điểm truy nhập đoạn
- Giải pháp: phân phối bộ nhớ theo các đoạn bằng nhau (page)?

- Chiến lược phân chương cố định
- Chiến lược phân chương động
- Chiến lược phân đoạn
- **Chiến lược phân trang**
- Chiến lược kết hợp phân đoạn-phân trang

Nguyên tắc

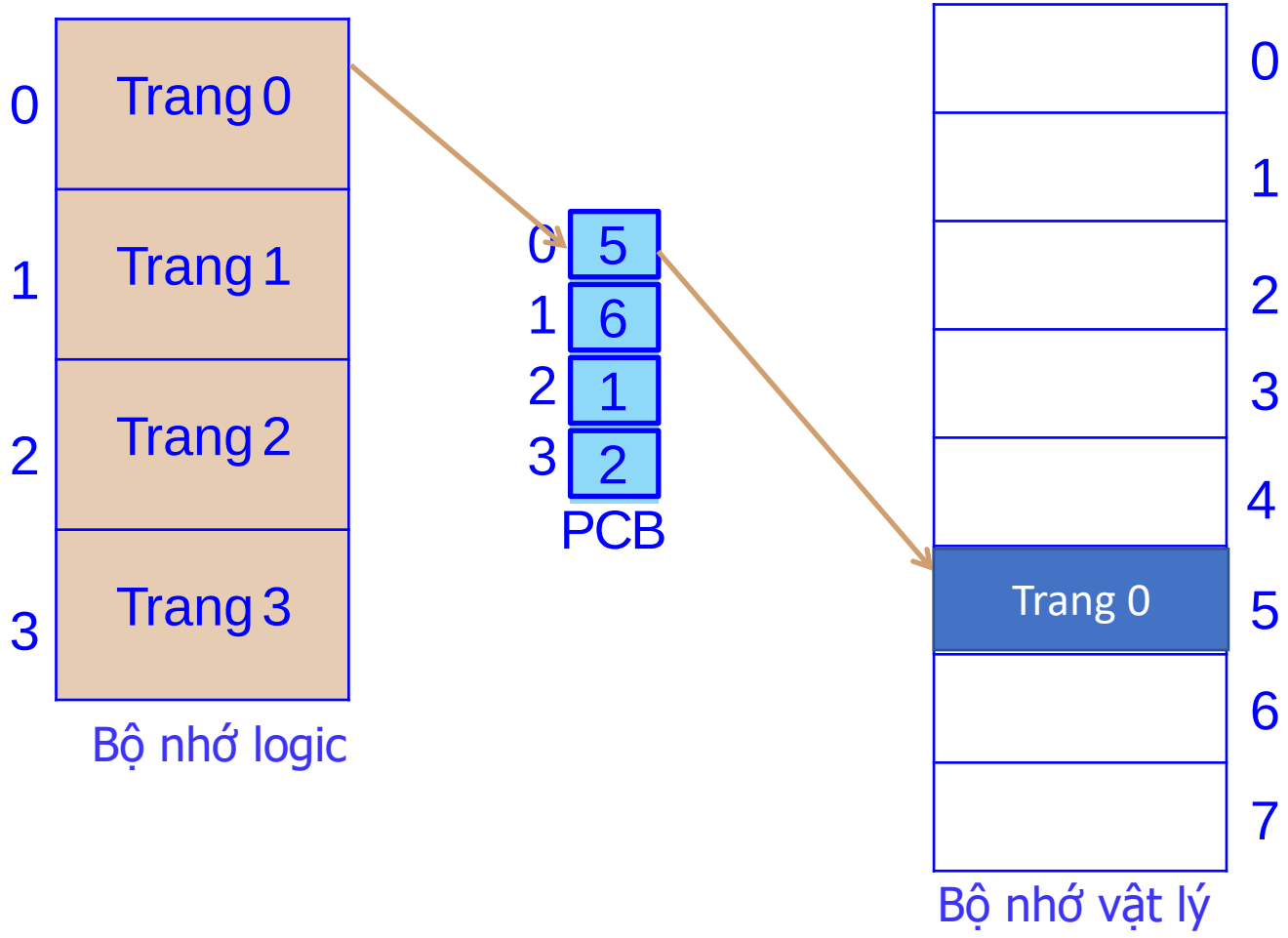
- Bộ nhớ vật lý được chia thành từng khối có kích thước bằng nhau: trang vật lý (frames – khung trang)
  - Trang vật lý được đánh số 0, 1, 2, . . . : địa chỉ vật lý của trang
  - Trang được dùng làm đơn vị phân phối nhớ

- Bộ nhớ logic (chương trình) được chia thành từng trang có kích thước bằng trang vật lý: trang logic (pages)

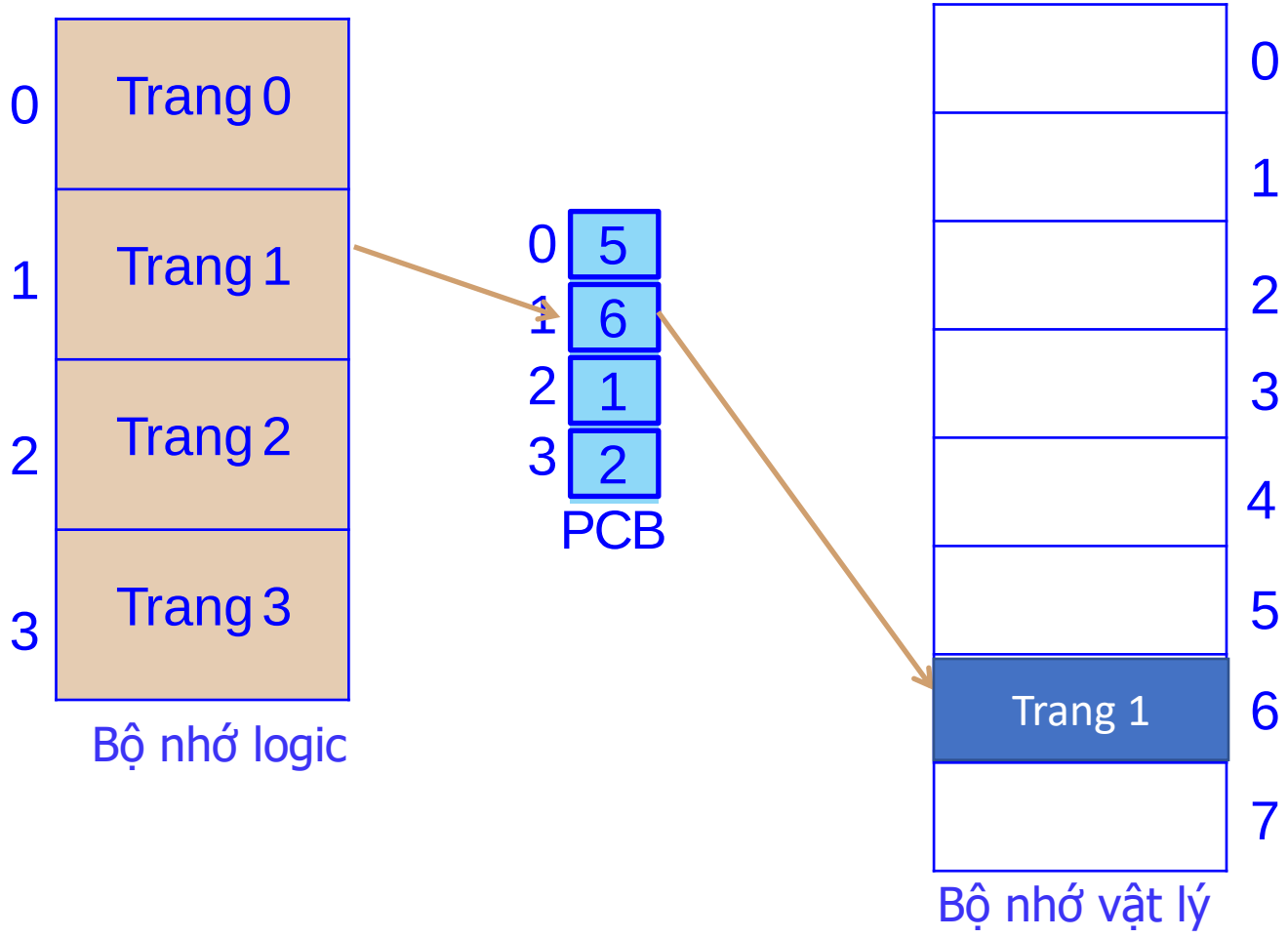
Nguyên tắc

- Khi thực hiện chương trình
  - Nạp trang logic (từ bộ nhớ ngoài) vào trang vật lý
  - Xây dựng một bảng quản lý trang (PCB: Page Control Block) dùng để xác định mối quan hệ giữa trang vật lý và trang logic
  - Mỗi phần tử của PCB ứng với một trang chương trình
    - Cho biết trang vật lý chứa trang logic tương ứng
    - Ví dụ PCB[8] = 4 ⇒ ?
  - Địa chỉ truy nhập được chia thành
    - Số hiệu trang (p) : Chỉ số trong PCB để tìm đ/chỉ cơ sở trang
    - Độ lệch trong trang (d): Kết hợp địa chỉ cơ sở của trang để tìm ra đ/chỉ vật lý

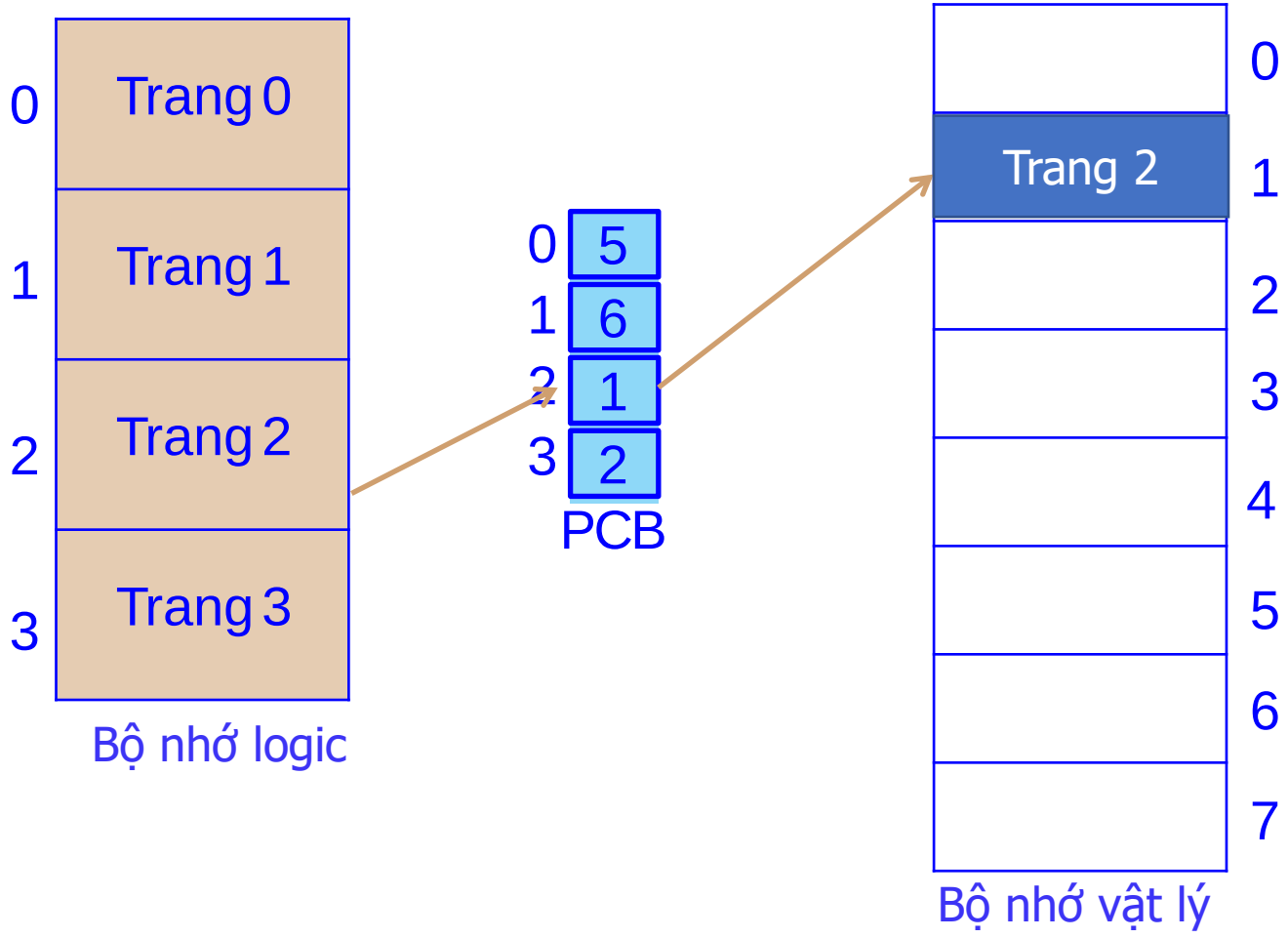
Ví dụ


0
Trang 0
1
Trang 1
2
Trang 2
3
Trang 3
Bộ nhớ logic
0 5
1 6
2 1
3 2
PCB
0
1
2
3
4
Trang 0
5
6
7
Bộ nhớ vật lý

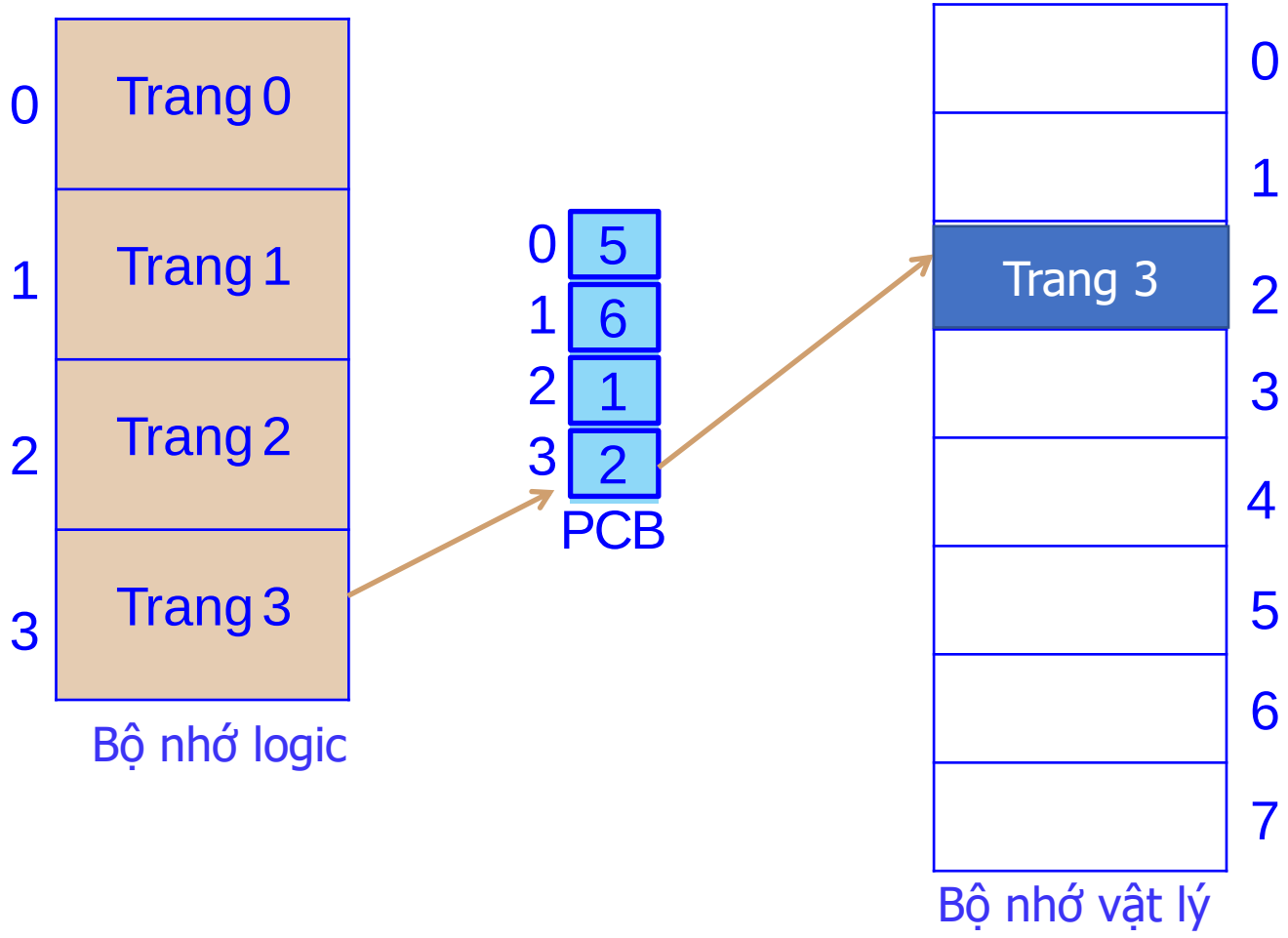
Ví dụ


0
Trang 0
1
Trang 1
2
Trang 2
3
Trang 3
Bộ nhớ logic
0 5
1 6
2 1
3 2
PCB
0
1
2
3
4
5
Trang 1
6
7
Bộ nhớ vật lý

Ví dụ

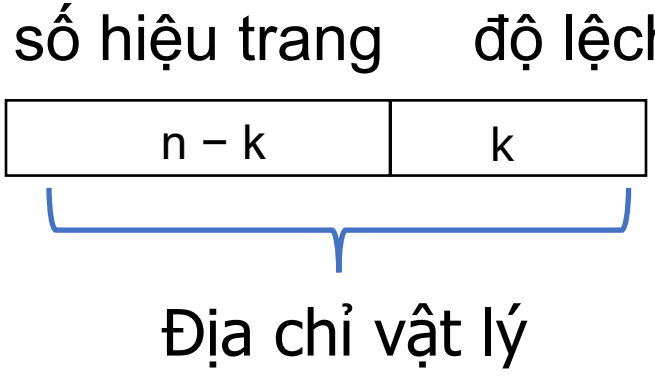

0
Trang 0
1
Trang 1
2
Trang 2
3
Trang 3
Bộ nhớ logic
0 5
1 6
2 1
3 2
PCB
0
Trang 2
1
2
3
4
5
6
7
Bộ nhớ vật lý

Ví dụ


0
Trang 0
1
Trang 1
2
Trang 2
3
Trang 3
Bộ nhớ logic
0 5
1 6
2 1
3 2
PCB
Trang 3
0
1
2
3
4
5
6
7
Bộ nhớ vật lý

Ghi chú

- Dung lượng trang luôn là lũy thừa của 2
  - Cho phép ghép giữa số hiệu trang vật lý và độ lệch trong trang
  - Ví dụ: Bộ nhớ n bit, kích thước trang 2^k

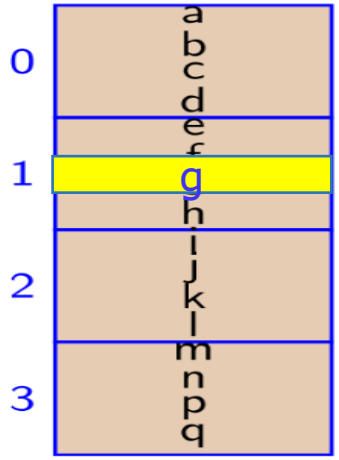


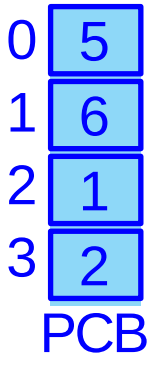
Ghi chú

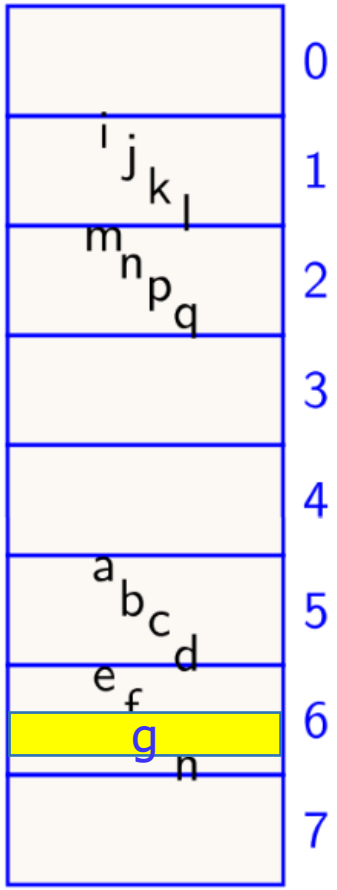
- Không cần thiết nạp toàn bộ trang logic vào
  - Số trang vật lý phụ thuộc k/thước bộ nhớ, số trang logic tùy ý
  - PCB cần trường dấu hiệu (Mark) cho biết trang đã được nạp vào bộ nhớ chưa
    - M = 0 Trang chưa tồn tại
    - M = 1 Trang đã được đưa vào bộ nhớ vật lý

- Phân biệt chiến lược phân trang - phân đoạn
  - Chiến lược phân đoạn
    - Các modul phụ thuộc cấu trúc logic của chương trình
  - Chiến lược phân trang
    - Các khối có kích thước độc lập kích thước chương trình
    - Kích thước khối phụ thuộc phần cứng (VD: $2^9 \rightarrow 2^{13}$ bytes)

Ví dụ





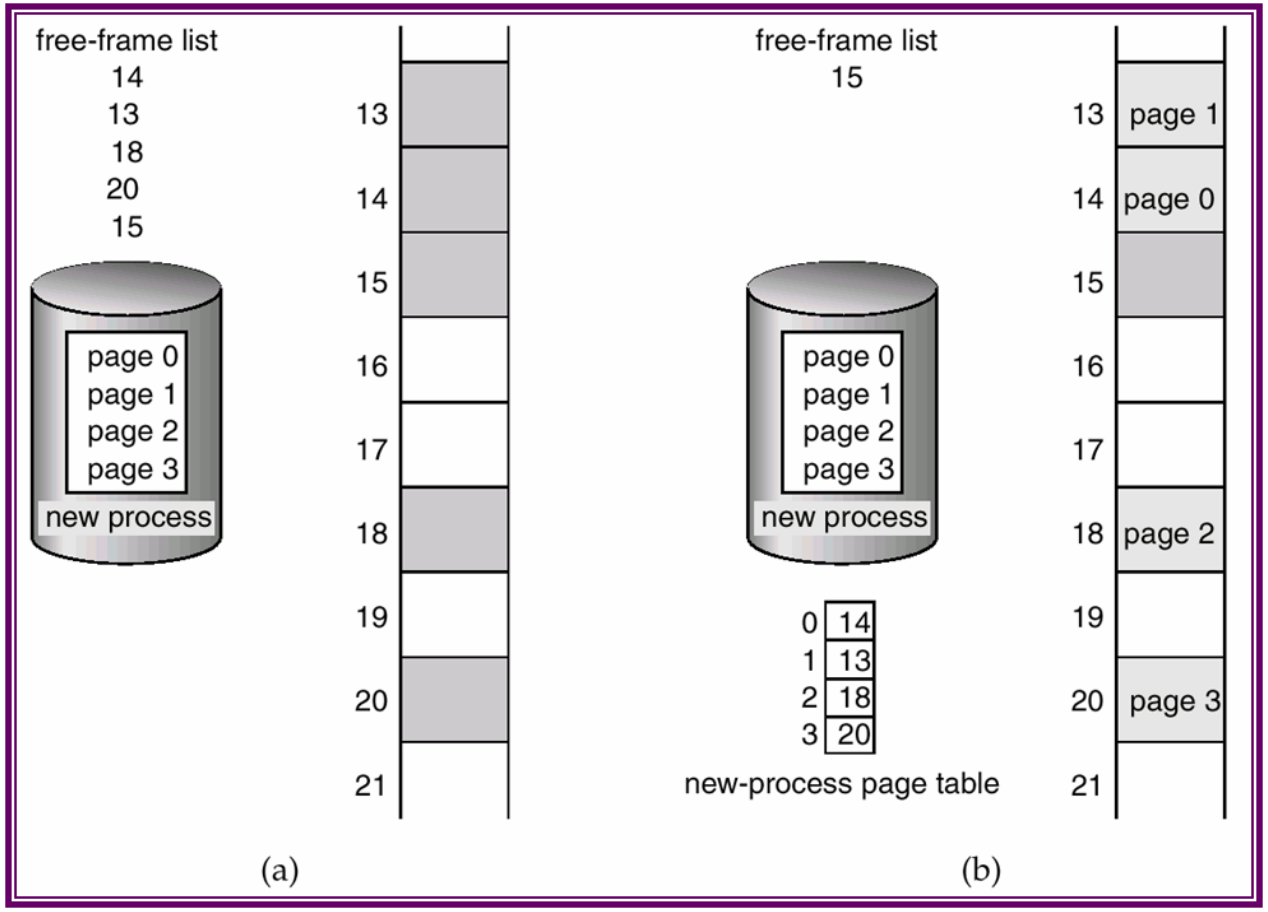


Bộ nhớ logic

Truy nhập địa chỉ logic [6] ?
Địa chỉ [6]: Trang 1, độ lệch 2
Địa chỉ <1,2> = 6*4 + 2 = 26

Bộ nhớ vật lý

Thực hiện chương trình → Nạp chương trình vào bộ nhớ

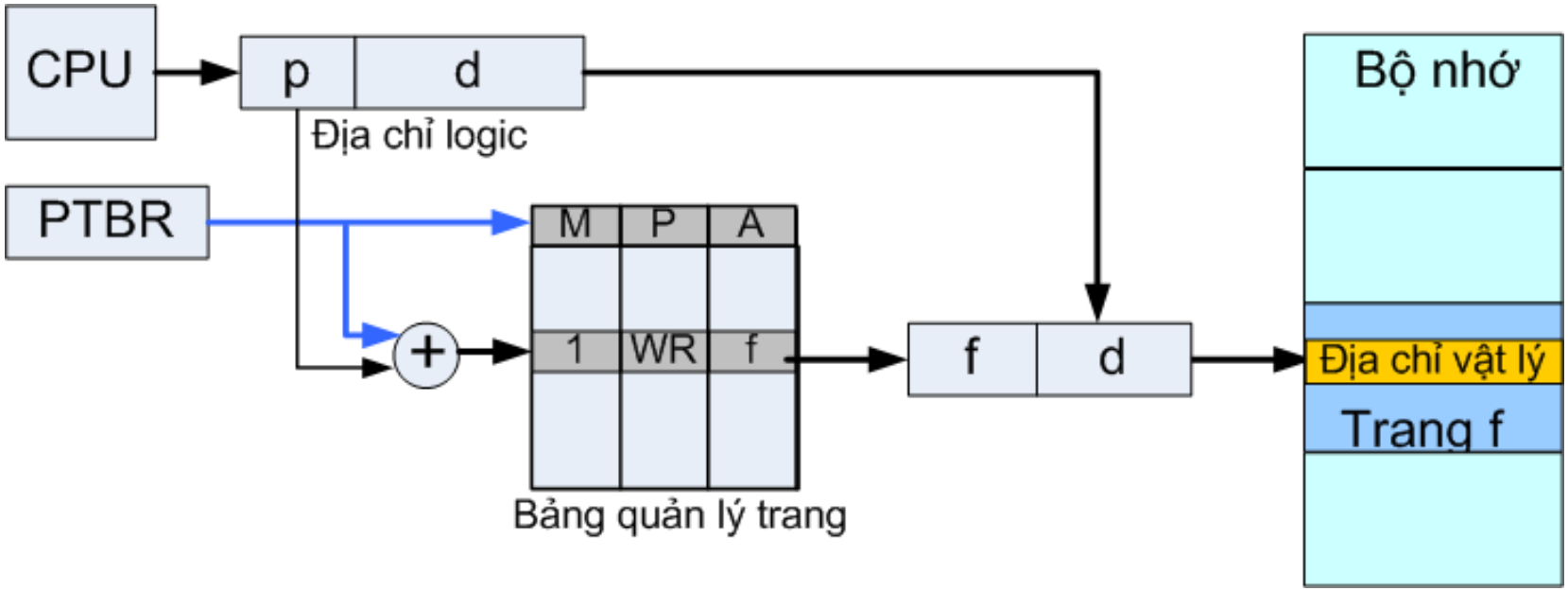


- Nếu đủ trang vật lý tự do ⇒ nạp toàn bộ
- Nếu không đủ trang vật lý tự do ⇒ nạp từng phần

Thực hiện chương trình → Truy nhập bộ nhớ

- Nạp chương trình
  - Xây dựng bảng quản lý trang và luôn giữ trong bộ nhớ
    - PTBR (Page-table base register) trỏ tới PCB.
    - PTLR(Page-table length register) kích thước PCB.
- Thực hiện truy nhập
  - Địa chỉ truy nhập được chia thành dạng <p, d>
  - PTBR + p ∗ K : Địa chỉ phần tử p của PCB trong bộ nhớ
    - K Kích thước 1 phần tử của PCB
  - Kiểm tra Mp
    - Mp = 0 : Lỗi trang, sinh một ngắt để tiến hành nạp trang
      - Xin trang vật lý tự do (Hết trang tự do?)
      - Tìm kiếm trang logic ở bộ nhớ ngoài và nạp trang
      - Sửa lại trường địa chỉ A và dấu hiệu M
    - Mp = 1 : Trang đã tồn tại,
      - Lấy Ap ghép với d ra địa chỉ cần tìm

Chuyển đổi địa chỉ: Sơ đồ truy nhập



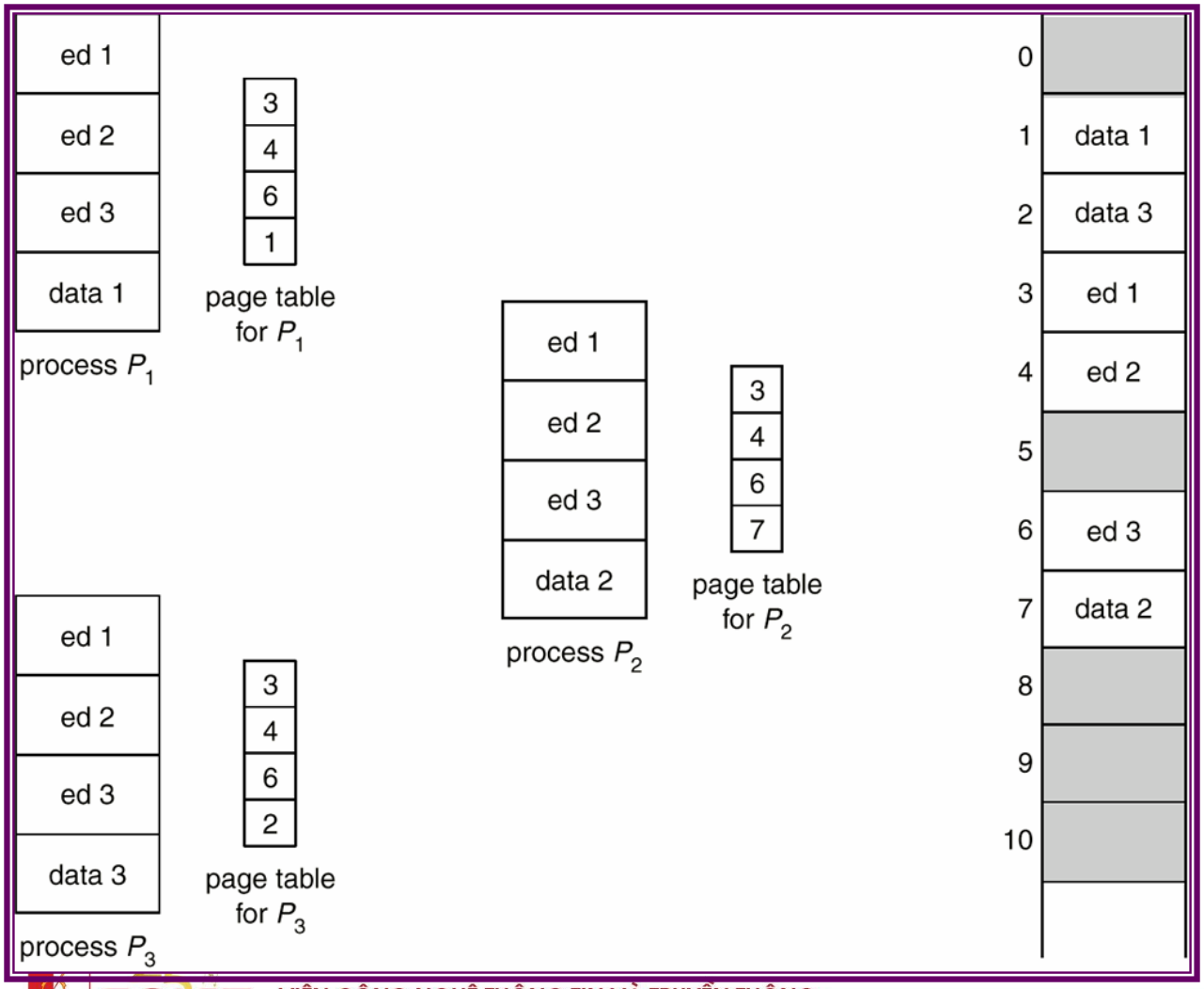
Nạp trang và thay thế trang

- Nhận xét
  - Số trang vật lý dành cho chương trình lớn
    - Thực hiện nhanh nhưng hệ số song song giảm
  - Số trang vật lý dành cho chương trình bé
    - Hệ số song song cao nhưng thực hiện chậm do hay thiếu trang
  - ⇒ Hiệu quả phụ thuộc các chiến lược nạp trang và thay thế trang
- Các chiến lược nạp trang
  - Nạp tất cả: Nạp toàn bộ chương trình
  - Nạp trước: Dự báo trang cần thiết tiếp theo
  - Nạp theo yêu cầu: Chỉ nạp khi cần thiết
- Các chiến lược thay thế trang
  - FIFO First In First Out
  - LRU Least Recently Used
  - LFU Least Frequently Used
  - . . .

Ưu điểm

- Tăng tốc độ truy nhập
  - Hai lần truy nhập bộ nhớ (vào PCB và vào địa chỉ cần tìm)
  - Thực hiện phép ghép thay vì phép cộng
- Không tồn tại hiện tượng phân đoạn ngoài
- Hệ số song song cao
  - Chỉ cần một vài trang của chương trình trong bộ nhớ
  - Cho phép viết chương trình lớn tùy ý
- Dễ dàng thực hiện nhiệm vụ bảo vệ
  - Địa chỉ truy nhập hợp lệ (vượt quá kích thước)
  - Tính chất truy nhập (đọc/ghi)
  - Quyền truy nhập (user/system)
- Cho phép sử dụng chung trang

Dùng chung trang : Soạn thảo văn bản



- Mỗi trang 50K
- 3 trang mã
- 1 trang dữ liệu
-
- 40 người dùng
- Không dùng chung
  - Cần 8000K
- Dùng chung
  - Chỉ cần 2150K

Dùng chung trang : Nguyên tắc

- Cần thiết trong môi trường hoạt động có sự chia sẻ
  - Giảm kích thước vùng nhớ cho tất cả các TT
- Phần mã dùng chung
  - Chỉ 1 phiên bản chia sẻ giữa các TT trong bộ nhớ
    - Ví dụ: Soạn thảo văn bản, chương trình dịch....
  - Vấn đề: Mã dùng chung không đổi
    - Trang dùng chung phải cùng vị trí trong không gian logic của tất cả TT⇒ Cùng số hiệu trong bảng quản lý trang
- Phần mã và dữ liệu riêng biệt
  - Riêng biệt cho các TT
  - Có thể nằm ở vị trí bất kỳ trong bộ nhớ logic của TT

Nhược điểm

- Tồn tại hiện tượng phân đoạn trong
  - Luôn xuất hiện ở trang cuối cùng
  - Giảm hiện tượng phân đoạn trang bởi giảm kích thước trang ?
    - Hay gặp lỗi trang
    - Bảng quản lý trang lớn
- Đòi hỏi hỗ trợ của phần cứng
  - Chi phí cho chiến lược phân trang lớn
- Khi chương trình lớn, bảng quản lý trang nhiều phần tử
  - Chương trình 2^{32}, k/thước trang 2^{12} -> PCB có 2^{20} phần tử
  - Tốn bộ nhớ lưu trữ PCB
  - Giải quyết: Trang nhiều mức

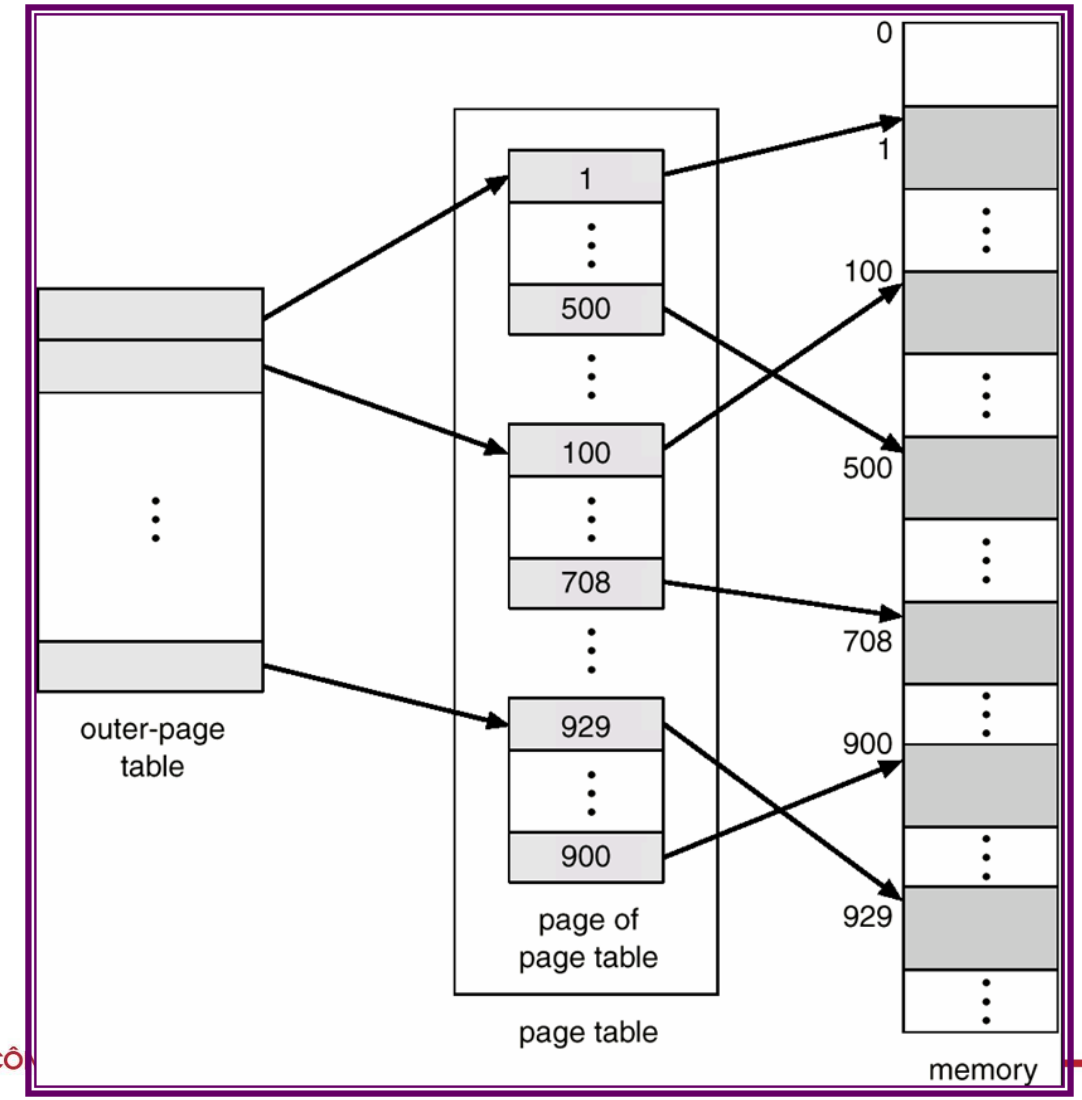
Trang nhiều mức

Nguyên tắc: Bảng quản lý trang được phân trang

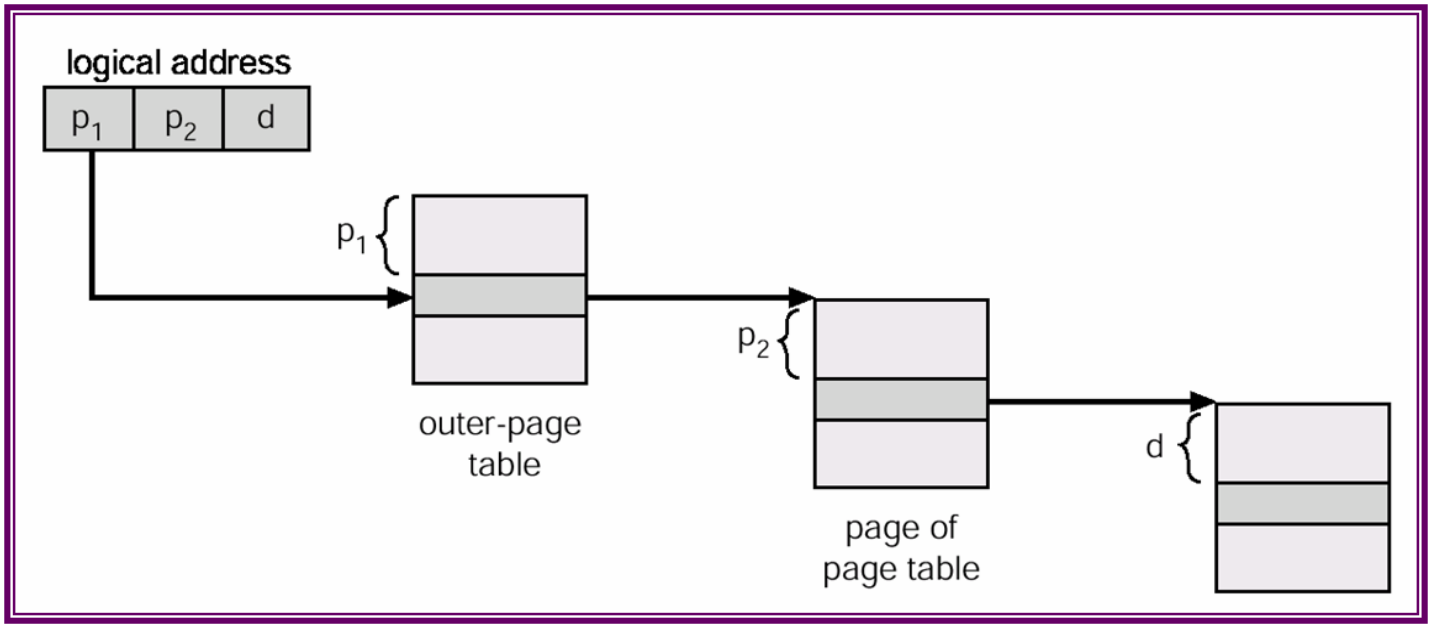
Ví dụ trang 2 mức

- Máy 32 bít địa chỉ (2^{32}); trang kích thước 4K (2^{12}) được chia
  - Số hiệu trang -20 bit
  - Độ lệch trong trang -12 bit
- Bảng trang được phân trang. Số hiệu trang được chia thành
  - Bảng trang ngoài (thư mục trang) - 10 bit
  - Độ lệch trong một thư mục trang – 10 bit
- Địa chỉ truy nhập có dạng <p_1, p_2, d >

Trang nhiều mức: Ví dụ trang 2 mức



Trang nhiều mức: Truy nhập bộ nhớ

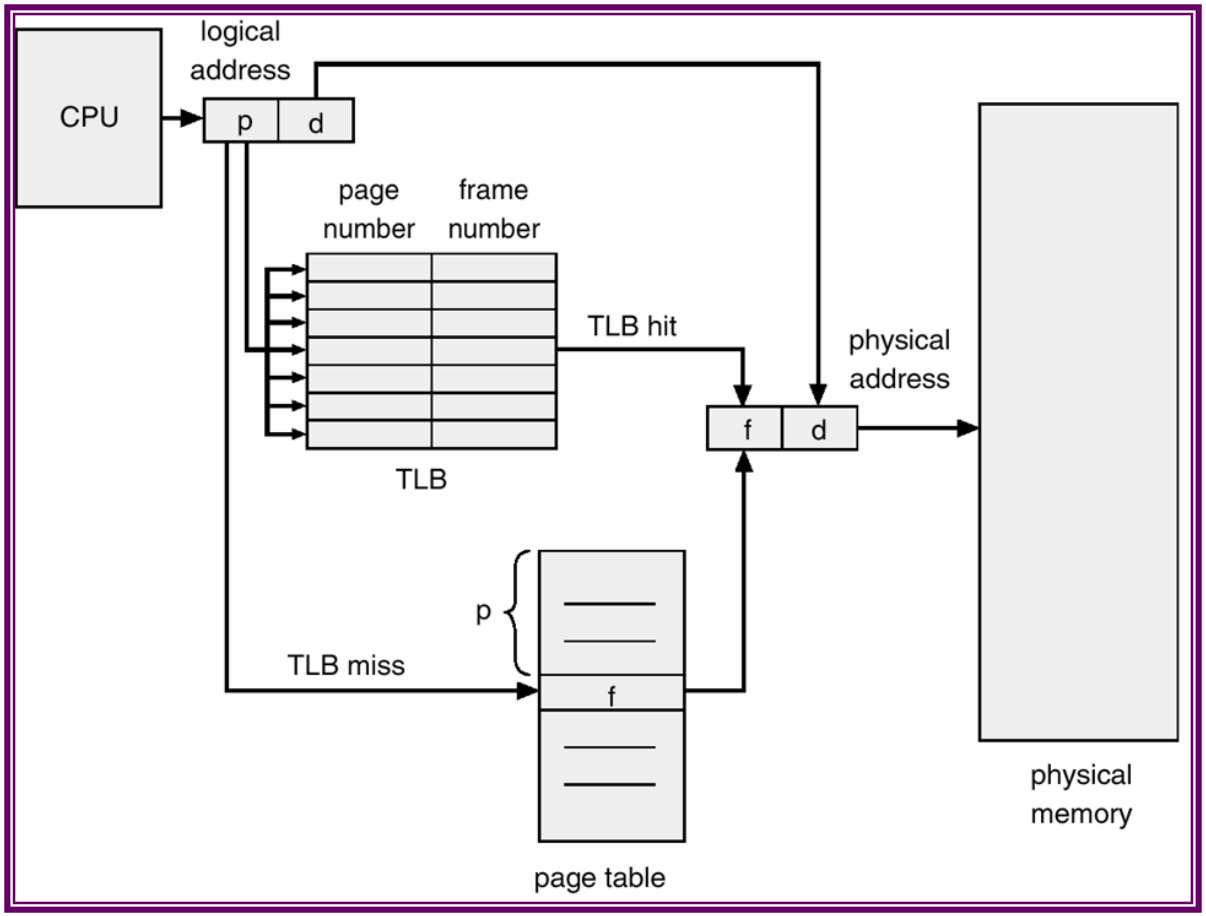


- Khi thực hiện : Hệ thống nạp thư mục trang vào bộ nhớ
- Bảng trang và trang không sử dụng không cần nạp vào bộ nhớ
- Cần 3 lần truy nhập tới bộ nhớ

Bộ đệm chuyển hóa địa chỉ

- Vấn đề: Với hệ thống 64 bit
  - Trang 3, 4,... mức
  - Cần 4, 5,... lần truy nhập bô nhớ ⇒ chậm
  - Giải quyết: Bộ đệm chuyển hóa địa chỉ
    - (TLB: translation look-aside buffers)
    - 98% truy nhập bộ nhớ được thực hiện qua TLB

Bộ đệm chuyển hóa địa chỉ



- Tập thanh ghi liên kết (associative registers)
- Truy nhập song song
- Mỗi phần tử gồm
  - Khóa: Page number
  - Giá trị: Frame number
- TLB chứa đ/chỉ những trang mới truy nhập
- Khi có y/cầu <p,d>
  - Tìm p trong TLB
  - Không có, tìm p trong PCB rồi đưa < p, f > vào TLB

- Chiến lược phân chương cố định
- Chiến lược phân chương động
- Chiến lược phân đoạn
- Chiến lược phân trang
- Chiến lược kết hợp phân đoạn-phân trang

Nguyên tắc I

- Chương trình được biên tập theo chế độ phân đoạn
  - Tạo ra bảng quản lý đoạn SCB
  - Mỗi phần tử của bảng quản lý đoạn ứng với 1 đoạn, gồm 3 trường M, A, L
- Mỗi đoạn được biên tập riêng theo chế độ phân trang
  - Tạo ra bảng quản lý trang cho từng đoạn
- Địa chỉ truy nhập: bộ 3 < s, p, d >

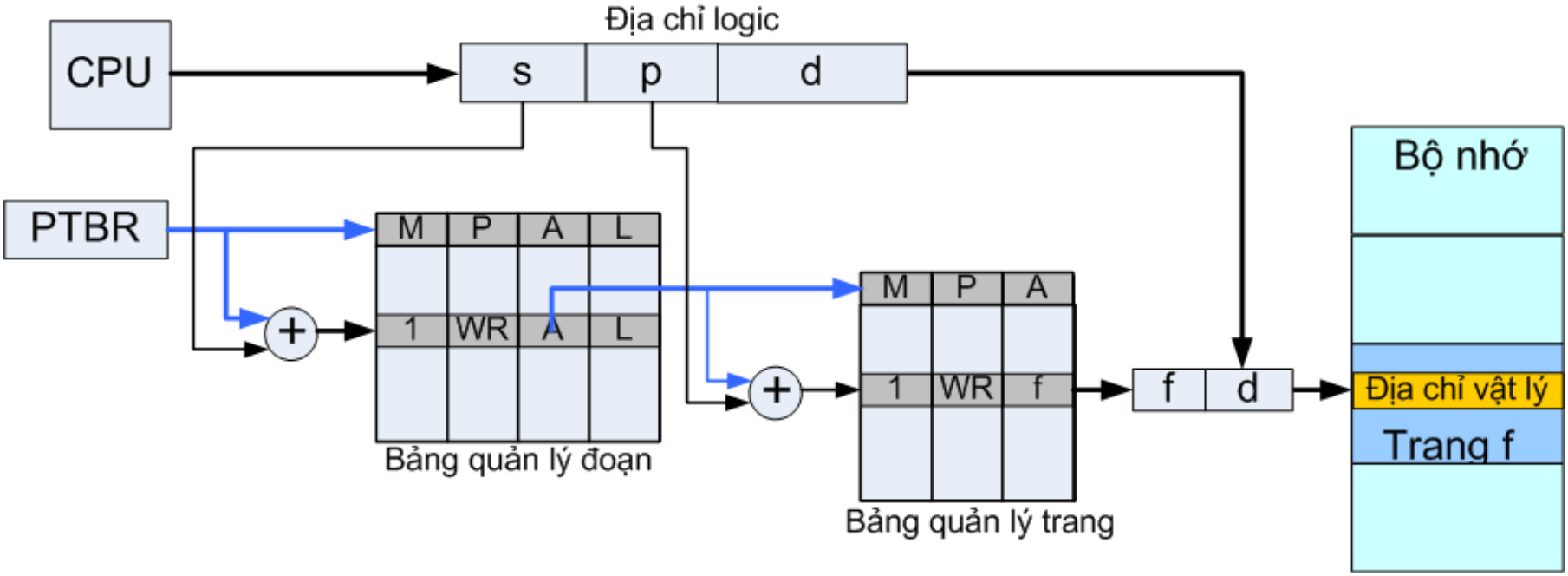
Nguyên tắc II

- Địa chỉ truy nhập: bộ 3 < s, p, d >

- Thực hiện truy nhập địa chỉ
  - STBR + s ⇒: địa chỉ phần tử s
  - Kiểm tra trường dấu hiệu Ms, nạp PCBs nếu cần
  - As + p ⇒ Địa chỉ phần tử p của PCBs
  - Kiểm tra trường dấu hiệu Mp, nạp trang p nếu cần
  - Ghép Ap với d ra được địa chỉ cần tìm

- Được sử dụng trong VXL Intel 80386, MULTICS . . .
  - Quản lý bộ nhớ của VXL họ intel
    - Chế độ thực
    - Chế độ bảo vệ

Sơ đồ truy nhập bộ nhớ



2. 5 Chiến lược kết hợp phân đoạn-phân trang

Tổng kết

M_0 2340B

M_1 5730 B

M_2 4264 B

M_3 1766 B

Phân đoạn

M	A	L
0	–	2340
1	2140	5730
0	–	4264
0	–	1766

Bảng: SCB

Kết hợp Phân đoạn – Phân trang

M	A	L
0	–	3
0	5	6
0	–	5
0	–	2

Bảng: SCB

M	A
0	–
1	8
0	–
0	–
0	–
0	–

Bảng: PCB_2

Chương 3 Quản lý bộ nhớ

1. Tổng quan
2. Các chiến lược quản lý bộ nhớ
3. Bộ nhớ ảo
4. Quản lý bộ nhớ trong VXL họ Intel

3. Bộ nhớ ảo

3.1 Giới thiệu

① Giới thiệu

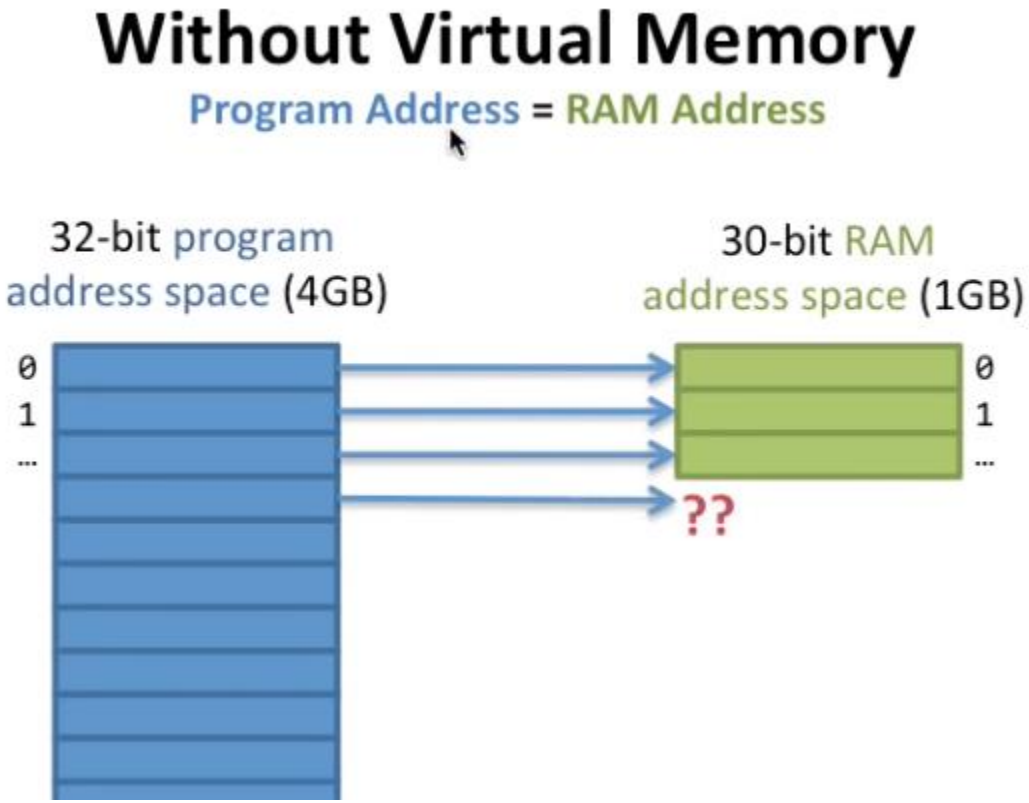
② Các chiến lược đổi trang

Đặt vấn đề

- Câu lệnh phải nằm trong bộ nhớ khi thực hiện !
- Toàn bộ chương trình phải nằm trong bộ nhớ ?
  - Cấu trúc động; cấu trúc Overlays... : Nạp từng phần
    - Đòi hỏi sự chú ý đặc biệt từ lập trình viên

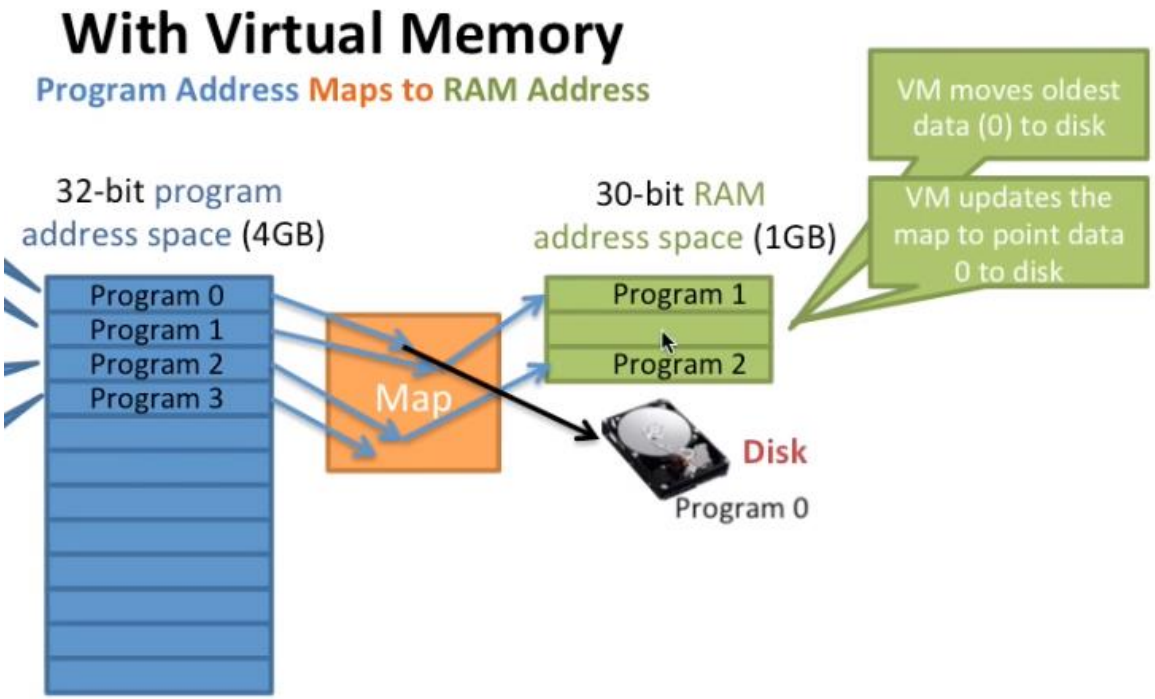
    ⇒Không cần thiết
  - Đoạn chương trình xử lý báo lỗi
    - Lỗi ít xảy tra, ít được thực hiện
  - Phần khai không dùng tới
    - Khai báo ma trận 100x100, sử dụng 10x 10
- Thực hiện c/trình chỉ có 1 phần nằm trong bộ nhớ cho phép
  - Viết chương trình trong không gian địa chỉ áo (virtual address space)
    - ⇒ lớn tùy ý
  - Nhiều chương trình đồng thời tồn tại
    - ⇒ tăng hiệu suất sử dụng CPU
  - Giảm yêu cầu vào/ra cho việc nạp và hoán đổi chương trình
    - Kích thước phần hoán đổi (swap) nhỏ hơn

Khái niệm bộ nhớ ảo



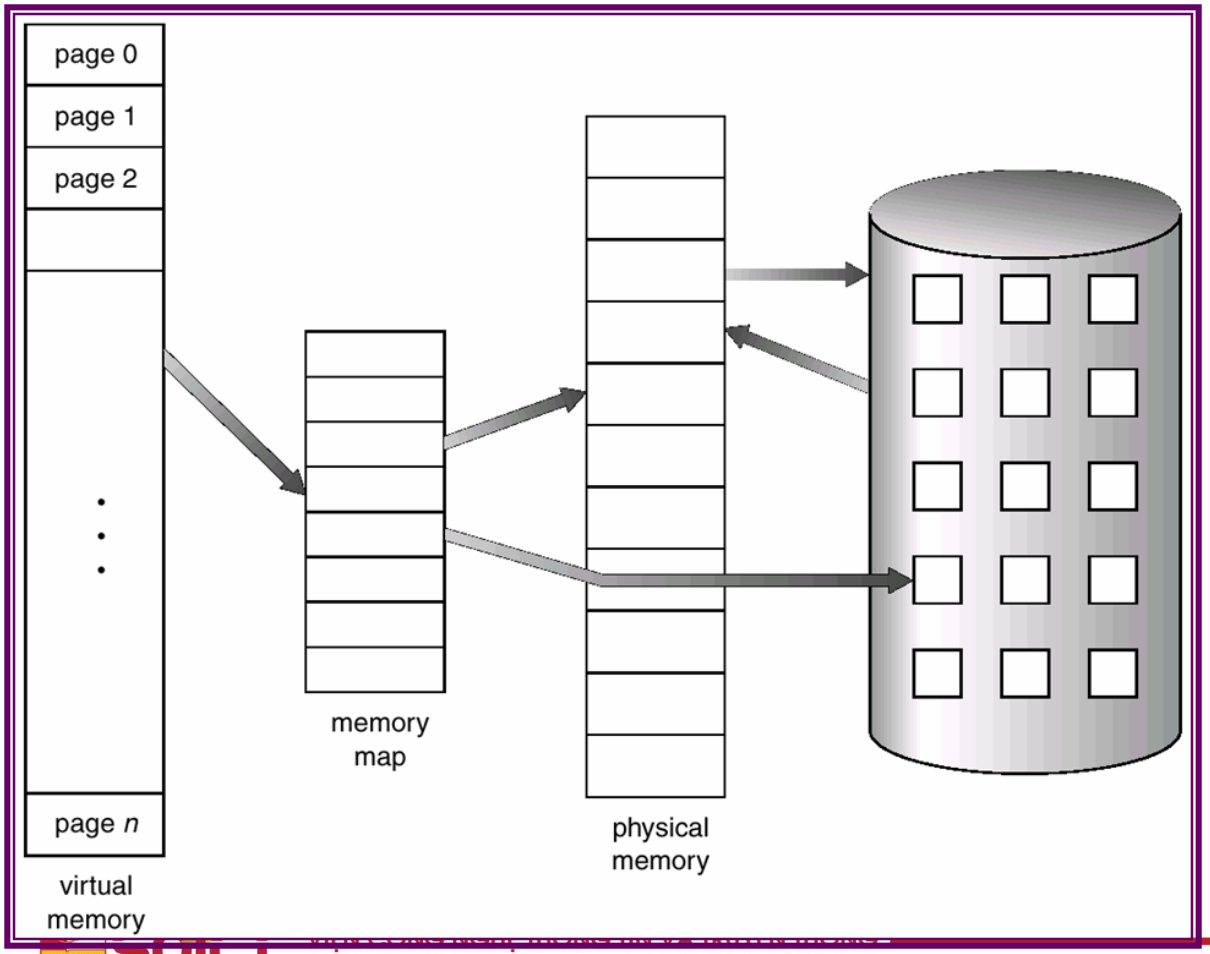
- Không đủ bộ nhớ vật lý ?

Khái niệm bộ nhớ ảo



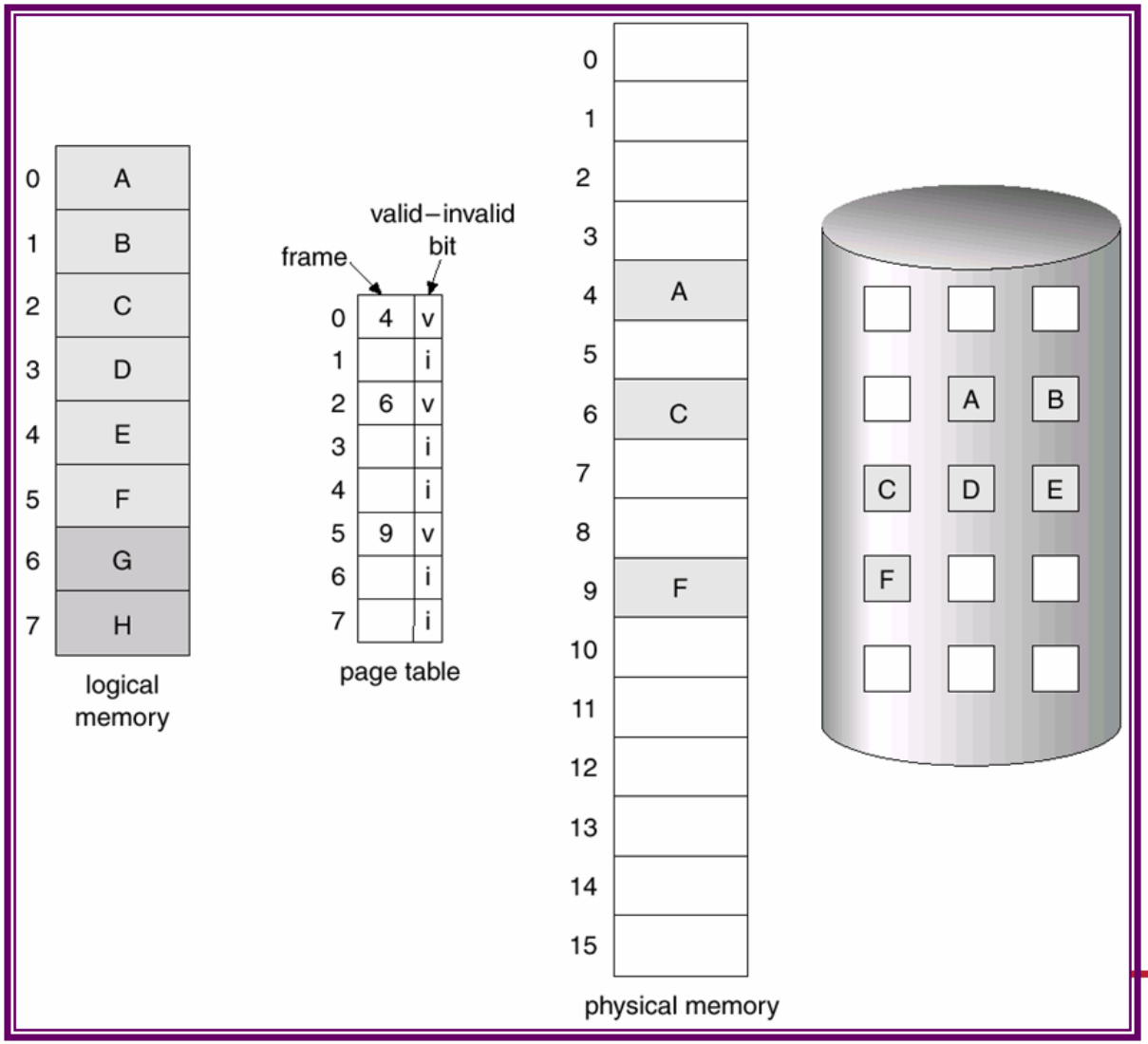
- Dùng bộ nhớ thứ cấp (*HardDisk*) lưu trữ phần chương trình chưa đưa vào bộ nhớ vật lý
- Phân tách bộ nhớ logic (*của người dùng*) với bộ nhớ vật lý

Khái niệm bộ nhớ ảo



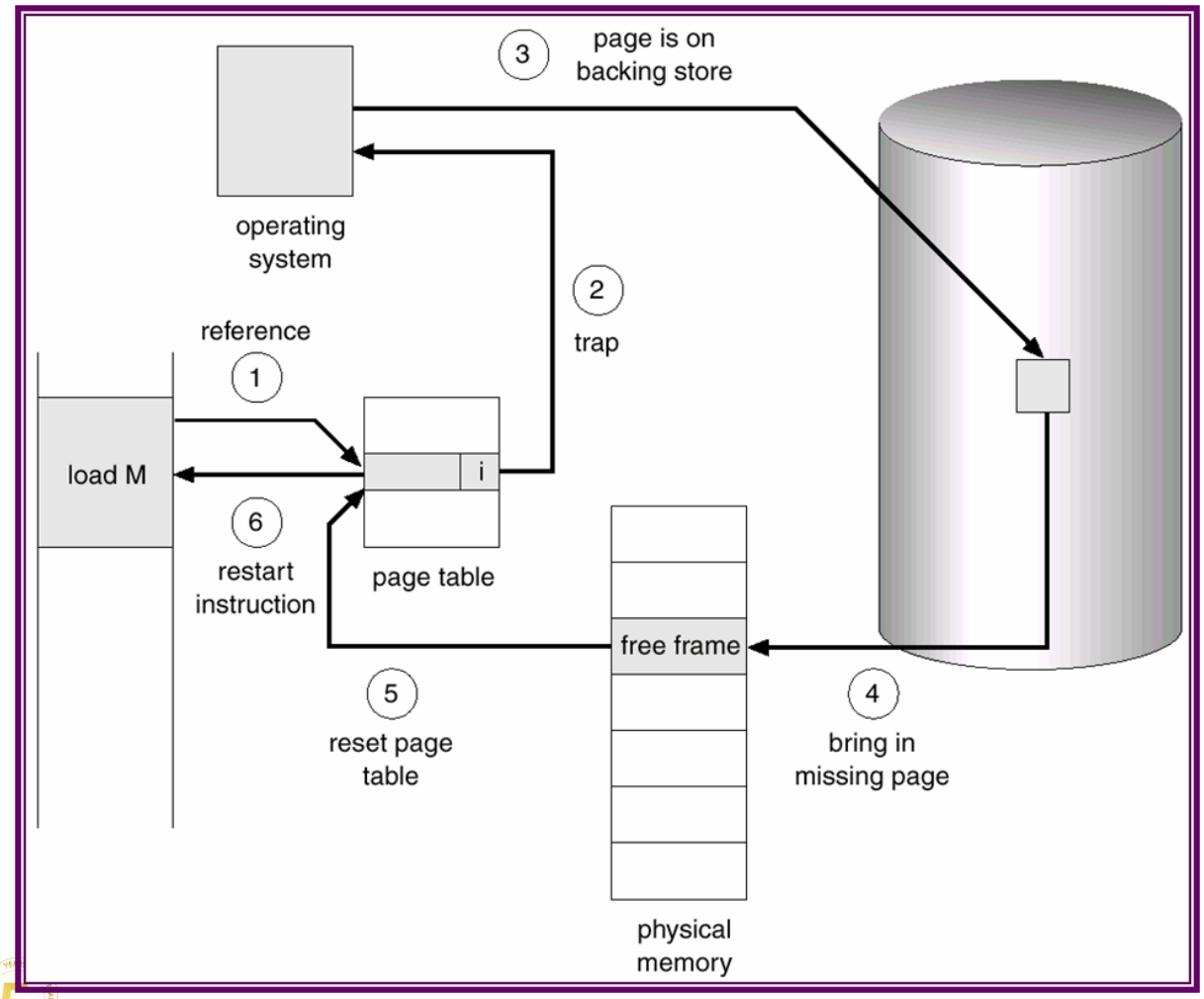
- VM ánh xạ địa chỉ trong chương trình tới địa chỉ vật lý
- Cho phép thể ánh xạ vùng nhớ logic lớn vào bộ nhớ vật lý nhỏ
- Cài đặt theo
  - Phân trang
  - Phân đoạn

Nạp từng phần của trang chương trình vào bộ nhớ



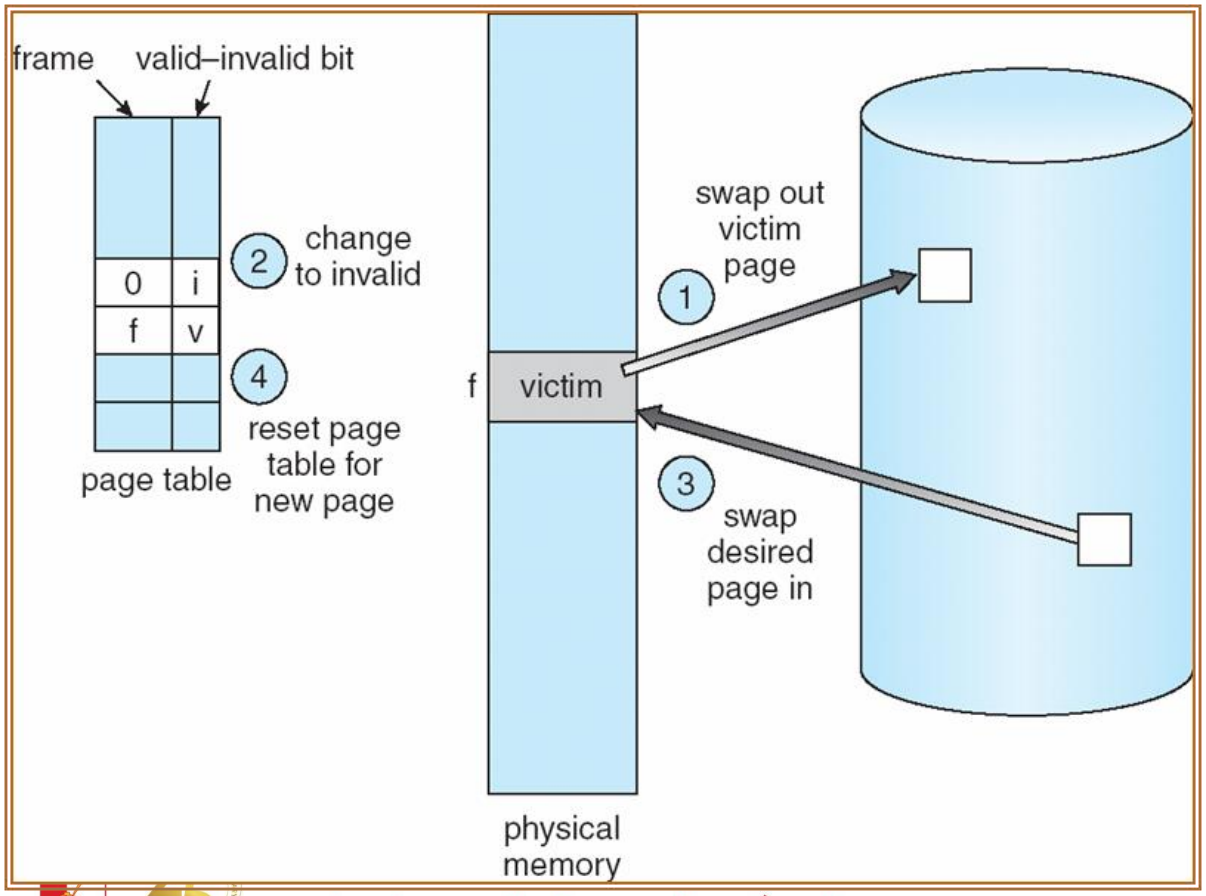
- Trang của tiến trình:
  - bộ nhớ vật lý,
  - một số trang nằm trên đĩa(bộ nhớ ảo)
- Biểu diễn nhờ sử dụng 1 bit trong bảng quản lý trang
- Khi yêu cầu trang, đưa trang từ bộ nhớ thứ cấp -> bộ nhớ vật lý

Xử lý lỗi trang



Nếu không có frames tự do, phải tiến hành đổi trang

Đổi trang



① Xác định vị trí trang logic trên đĩa
② Lựa chọn trang vật lý
- Ghi ra đĩa
- Sửa lại bit valid-invalid

③ Nạp trang logic vào trang vật lý được chọn
④ Restart tiến trình

3. Bộ nhớ ảo

3.2 Các chiến lược đổi trang

① Giới thiệu

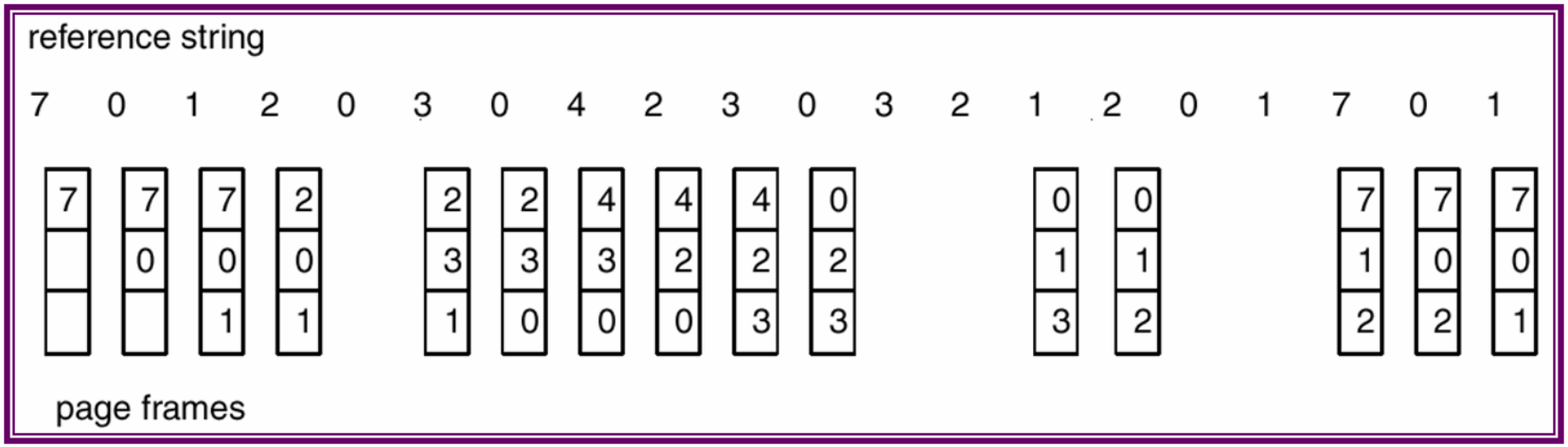
② Các chiến lược đổi trang

Các chiến lược

- FIFO (First In First Out): Vào trước ra trước
- OPT/MIN: Thuật toán thay thế trang tối ưu
- LRU (Least Recently Used): Trang có lần sử dụng cuối cách đây lâu nhất
- LFU (Least Frequently used):Tần xuất sử dụng thấp nhất
- MFU (Most Frequently used): Tần xuất sử dụng cao nhất
- . . .

FIFO

Ví dụ

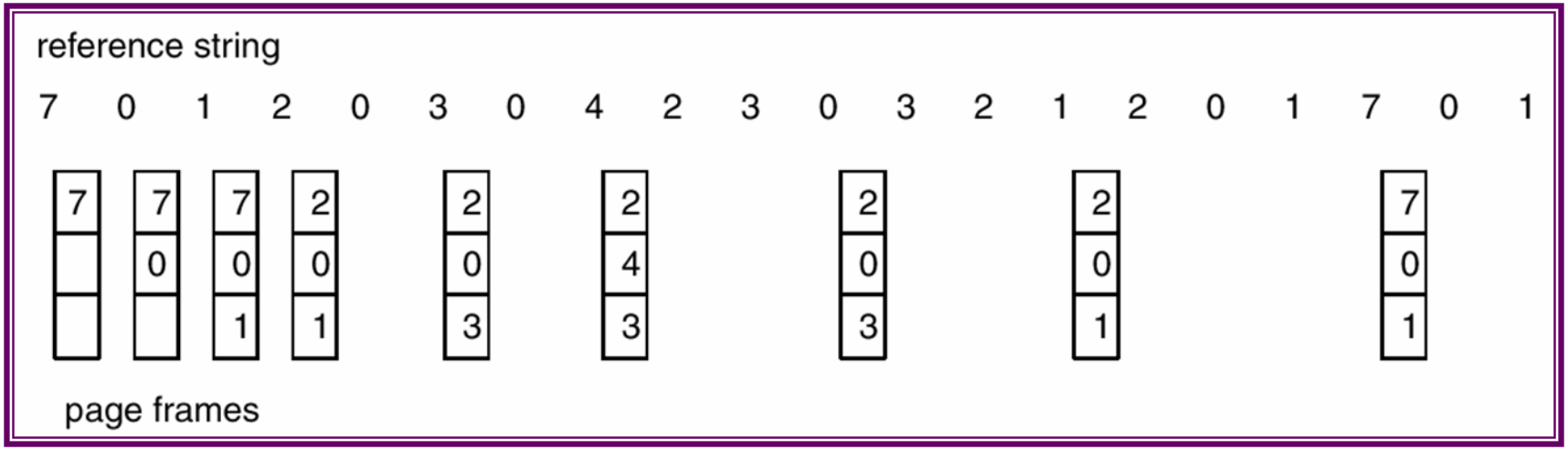


Nhận xét

- Hiệu quả khi c/trình có cấu trúc tuyến tính.
- Kém hiểu quả khi c/trình theo nguyên tắc lập trình cấu trúc
- Đơn giản dễ thực hiện
  - Dùng hàng đợi lưu các trang của chương trình trong bộ nhớ
  - Chèn ở cuối hàng, Thay thế trang ở đầu hàng
- Tăng trang vật lý, không đảm bảo giảm số lần gặp lỗi trang
  - Dãy truy nhập: 1 2 3 4 1 2 5 1 2 3 4 5
  - 3 frames: 9 lỗi trang; 4 frames: 10 lỗi trang

OPT

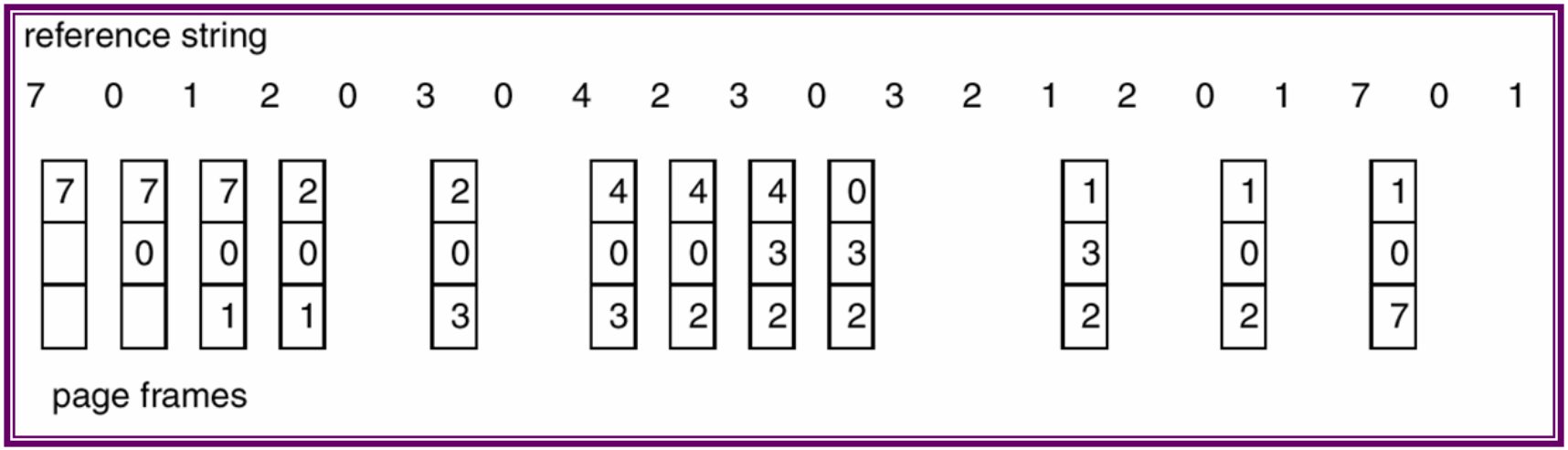
Nguyên tắc: Đưa ra trang có lần sử dụng tiếp theo cách xa nhất



- Số lần gặp lỗi trang ít nhất
- Khó dự báo được diễn biến của chương trình

LRU

Nguyên tắc: Đưa ra trang có lần sử dụng cuối cách xa nhất



- Hiệu quả cho chiến lược thay thế trang
- Đảm bảo giảm số lỗi trang khi tăng số trang vật lý
  - Tập các trang trong bộ nhớ có *n* frames luôn là tập con của các trang trong bộ nhớ có *n + 1* frames
- Y/cầu sự trợ giúp kỹ thuật để chỉ ra thời điểm truy nhập cuối
- Cài đặt như thế nào?

LRU: Cài đặt

- Bộ đếm
  - Thêm 1 trường ghi thời điểm truy nhập vào mỗi phần tử của PCB
  - Thêm vào khối điều khiển (C.U) đồng hồ/bộ đếm
  - Khi có yêu cầu truy nhập trang
    - Tăng bộ đếm
    - Chép nội dung bộ đếm vào trường thời điểm truy nhập tại phần tử tương ứng trong PCB
  - Cần có thủ tục cập nhật PCB (ghi vào trường thời điểm) và thủ tục tìm kiếm trang có giá trị trường thời điểm nhỏ nhất
  - Hiện tượng tràn số !?
- Dãy số (Stack)
  - Dùng dãy số ghi số trang
    - Truy nhập tới một trang, cho phần tử tương ứng lên đầu dãy
  - Thay thế trang: Phần tử cuối dãy
  - Thường cài đặt dưới dạng DSLK 2 chiều
    - 4 phép gán con trỏ ⇒ tốn thời gian

Thuật toán dựa trên bộ đếm

Sử dụng bộ đếm (một trường của PCB) ghi nhận số lần truy nhập tới trang

- LFU: Trang có bộ đếm nhỏ nhất bị thay thế
  - Trang truy nhập nhiều đến
    - Trang quan trọng ⇒ hợp lý
    - Trang khởi tạo, chỉ được dùng ở giai đoạn đầu ⇒ không hợp lý ⇒Dịch bộ đếm một bit (chia đôi) theo thời gian
- MFU: Trang có bộ đếm lớn nhất
  - Trang có bộ đếm nhỏ nhất, vừa mới được nạp vào và vẫn chưa được sử dụng nhiều

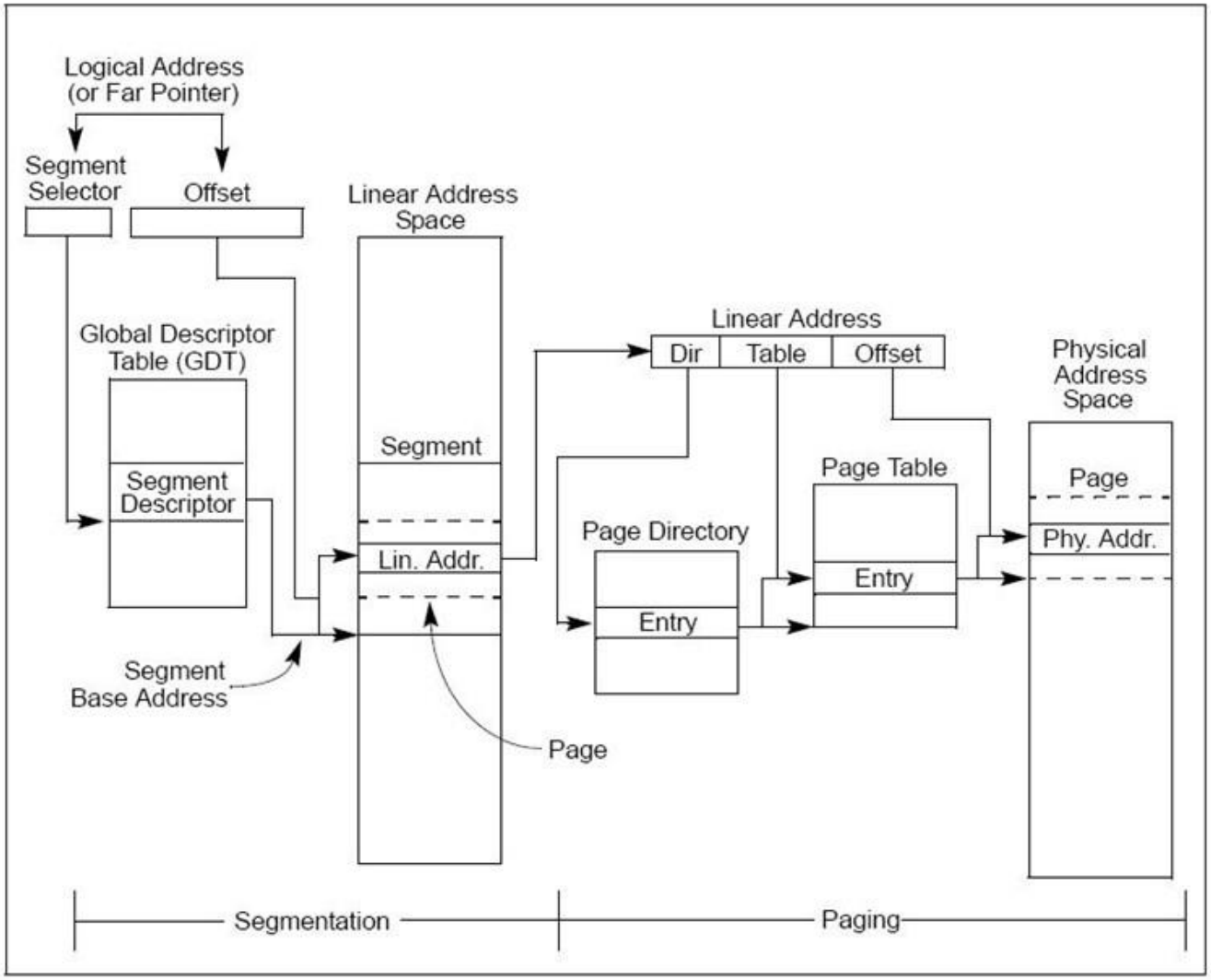
Chương 3 Quản lý bộ nhớ

① Tổng quan
② Các chiến lược quản lý bộ nhớ
③ Bộ nhớ ảo
④ Quản lý bộ nhớ trong VXL họ Intel

Các chế độ

- Intel 8086, 8088
  - Chỉ có một chế độ quản lý: Chế độ thực (Real Mode)
  - Quản lý vùng nhớ lên đến 1MB (20bit)
  - Xác định địa chỉ ô nhớ bằng 2 giá trị 16 bit: Segment, Offset
    - Thanh ghi đoạn: CS, SS, DS, ES,
    - Thanh ghi độ lệch: IP, SP, BP...
    - Địa chỉ vật lý: Seg SHL 4 + Ofs
- Intel 80286
  - Chế độ thực, tương thích với 8086
  - Chế độ bảo vệ (Protected mode),
    - Sử dụng phương pháp phân đoạn
    - Khai thác được bộ nhớ vật lý 16M (24bit)
- Intel 80386, Intel 80486, Pentium,..
  - Chế độ thực, tương thích với 8086
  - Chế độ bảo vệ :Kết hợp phân đoạn, phân trang
  - Chế độ ảo (Virtual mode)
    - Cho phép thực hiện mã 8086 trong chế độ bảo vệ

Chế độ bảo vệ trong Intel 386, 486, Pentium,..



Kết luận

① Tổng quan
   1. Ví dụ
   2. Bộ nhớ và chương trình
   3. Liên kết địa chỉ
   4. Các cấu trúc chương trình

② Các chiến lược quản lý bộ nhớ
   1. Chiến lược phân chương cố định
   2. Chiến lược phân chương động
   3. Chiến lược phân đoạn
   4. Chiến lược phân trang
   5. Chiến lược kết hợp phân đoạn-phân trang

③ Bộ nhớ ảo
   1. Giới thiệu
   2. Các chiến lược đổi trang

④ Quản lý bộ nhớ trong VXL họ Intel