Chương 6: Bộ Nhớ Thực

- Các kiểu địa chỉ nhớ
- Chuyển đổi địa chỉ nhớ
- Overlay và swapping
- Mô hình quản lý bộ nhớ đơn giản
  - Fixed partitioning
  - Dynamic partitioning

Quản lý bộ nhớ

- Phân phối và sắp xếp các process trong bộ nhớ sao cho hệ thống hoạt động hiệu quả.
  - Vd: “nạp càng nhiều process vào bộ nhớ càng tốt (gia tăng mức độ đa chương)”
- Thông thường, kernel chiếm một vùng cố định của bộ nhớ, vùng còn lại phân phối cho các process.
- Yêu cầu đối với việc quản lý bộ nhớ
  - Cấp phát vùng nhớ cho các process
  - Tái định vị (relocation): khi swapping,…
  - Bảo vệ: phải kiểm tra truy xuất bộ nhớ có hợp lệ không
  - Chia sẻ: cho phép các process chia sẻ vùng nhớ chung
  - Kết gán địa chỉ nhớ luận lý của process vào địa chỉ thực

Các kiểu địa chỉ nhớ (1/2)

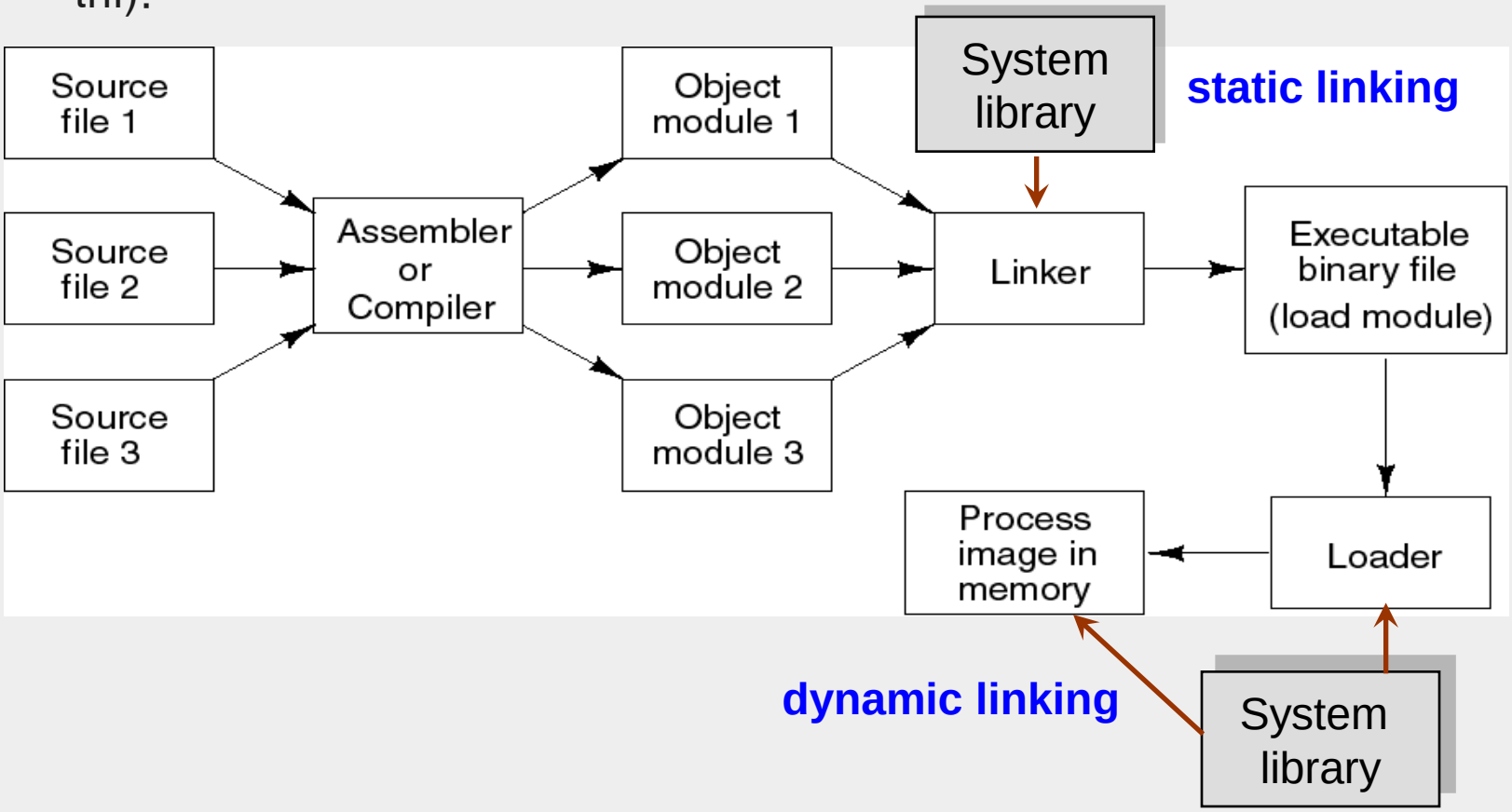
- *Địa chỉ vật lý* -- physical (memory) address -- là địa chỉ mà CPU, hay MMU (Memory Management Unit) nếu có, gửi đến bộ nhớ chính.
- *Địa chỉ luận lý* (logical address) là địa chỉ một ô nhớ mà một quá trình sinh ra
- Trình biên dịch (compiler) tạo ra mã lệnh chương trình mà trong đó một tham chiếu bộ nhớ là
  - *tương đối*: cần được diễn dịch tương đối so với một vị trí xác định nào đó trong chương trình.
    - Ví dụ: 12 byte so với vị trí bắt đầu chương trình,…
  - *tuyệt đối*: cần được diễn dịch tương đương địa chỉ thực.

Các kiểu địa chỉ nhớ (2/2)

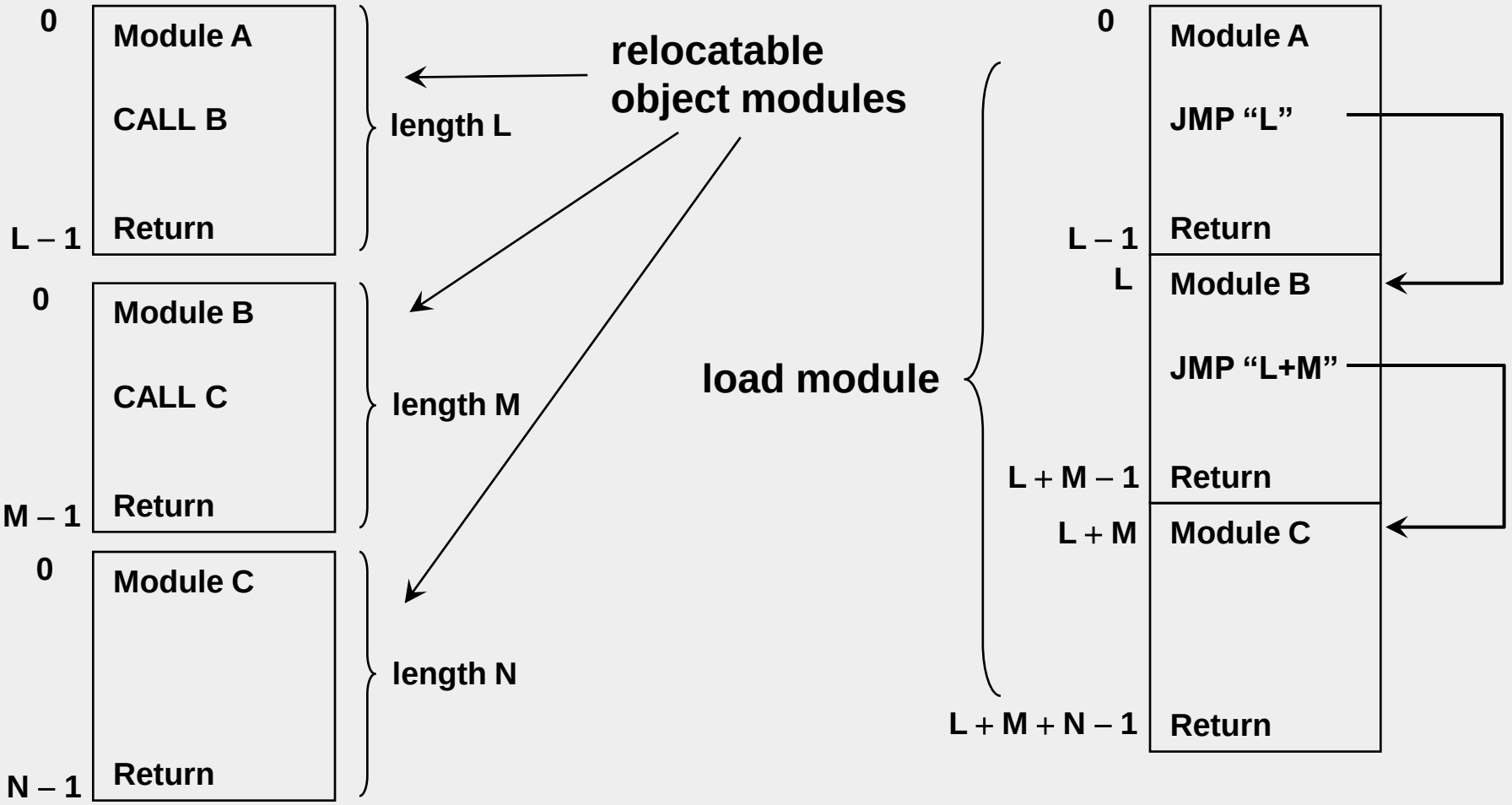
- Khi một lệnh được thực thi, các địa chỉ luận lý phải được chuyển đổi thành địa chỉ vật lý.
  - Sự chuyển đổi này thường có sự hỗ trợ của phần cứng để đạt hiệu suất cao.

Nạp chương trình vào bộ nhớ

- Bộ linker:
  - tái định vị địa chỉ tương đối và phân giải các external reference
  - kết hợp các object module thành một *load module* (file nhị phân khả thực thi).

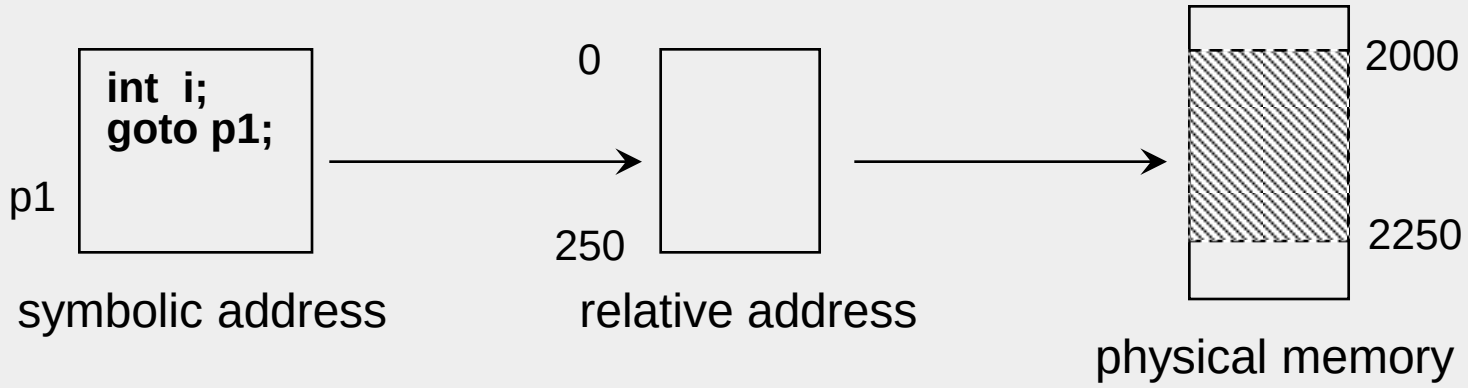


Thực hiện (static) linking



Chuyển đổi địa chỉ (1/3)

- *Chuyển đổi địa chỉ:* quá trình ánh xạ một địa chỉ từ không gian địa chỉ này sang không gian địa chỉ khác.
- Biểu diễn địa chỉ nhớ
  - Trong source code: symbolic (các biến, hằng, pointer,…)
  - Thời điểm biên dịch: thường là địa chỉ tương đối
    - Ví dụ: a ở vị trí 14 byte so với vị trí bắt đầu của module.
  - Thời điểm linking/loading: có thể là địa chỉ tuyệt đối.

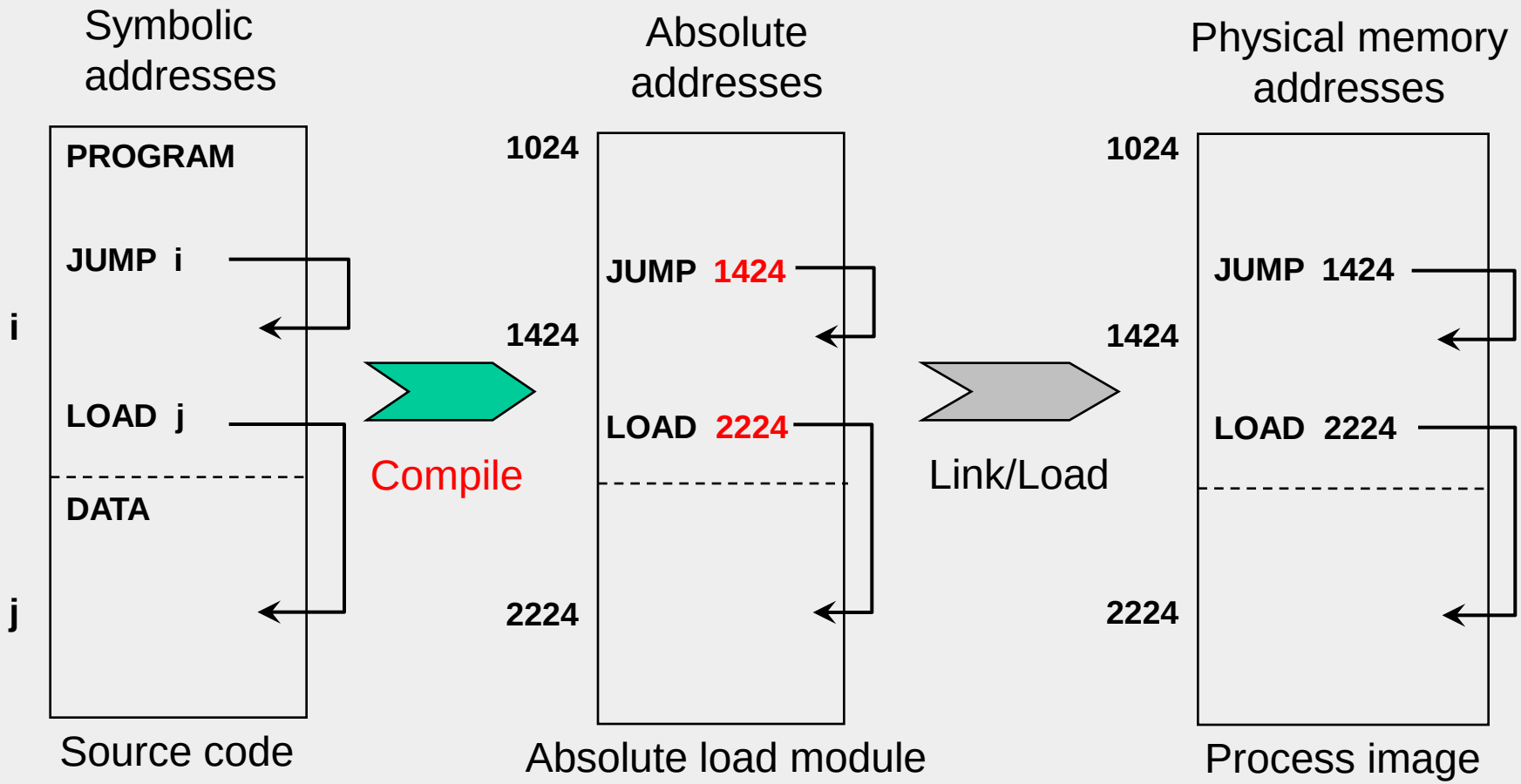


Chuyển đổi địa chỉ (2/3)

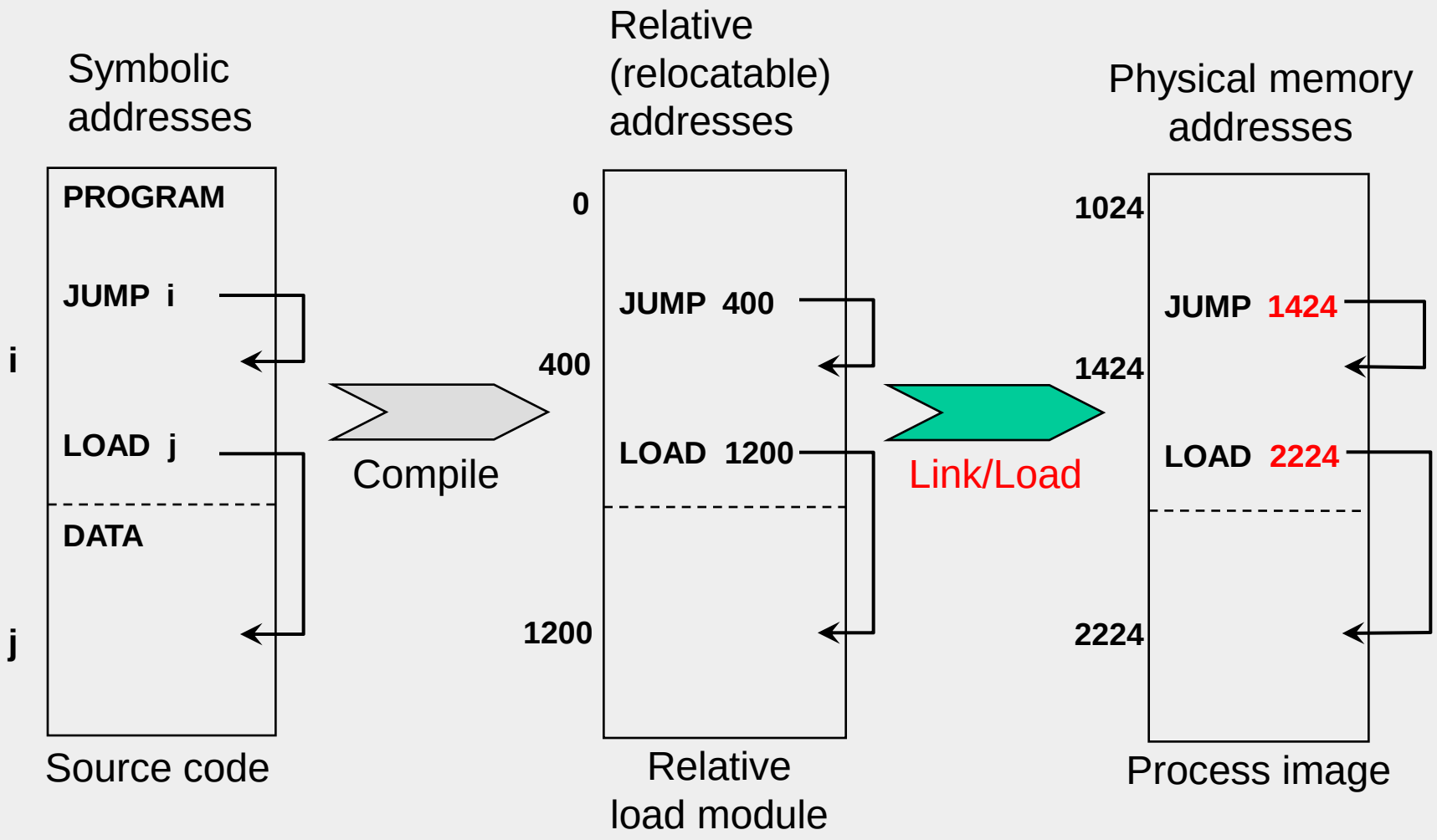
- Địa chỉ lệnh (instruction) và dữ liệu (data) có thể được chuyển đổi thành địa chỉ tuyệt đối tại các thời điểm
  - Compile time: nếu biết trước địa chỉ bộ nhớ của chương trình thì có thể kết gán địa chỉ tuyệt đối lúc biên dịch.
    - Ví dụ: chương trình .COM của MS-DOS, phát biểu assembly

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

Sinh địa chỉ tuyệt đối vào thời điểm dịch



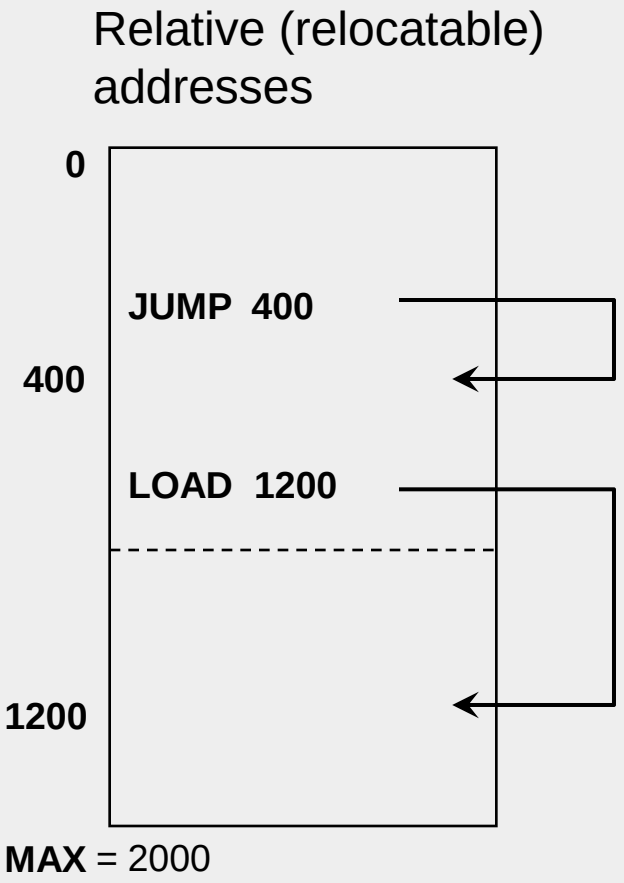
Sinh địa chỉ thực vào thời điểm nạp



Chuyển đổi địa chỉ (3/3)

- Execution time: nếu trong khi thực thi, process có thể được di chuyển từ vùng nhớ này sang vùng nhớ khác thì việc chuyển đổi địa chỉ được trì hoãn đến thời điểm thực thi
  - Cần sự hỗ trợ của phần cứng cho việc chuyển đổi địa chỉ.

    Ví dụ
    - Dùng thanh ghi base và limit
    - Paging (Chương tới)
    - Segmentation (Chương tới)



Tiết kiệm bộ nhớ từ ứng dụng

- Các kỹ thuật mà ứng dụng có thể sử dụng để hỗ trợ tiết kiệm bộ nhớ:
  - Dynamic linking
  - Dynamic loading
  - Overlay
  - Swapping

Dynamic linking (1)

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

Dynamic linking (2)

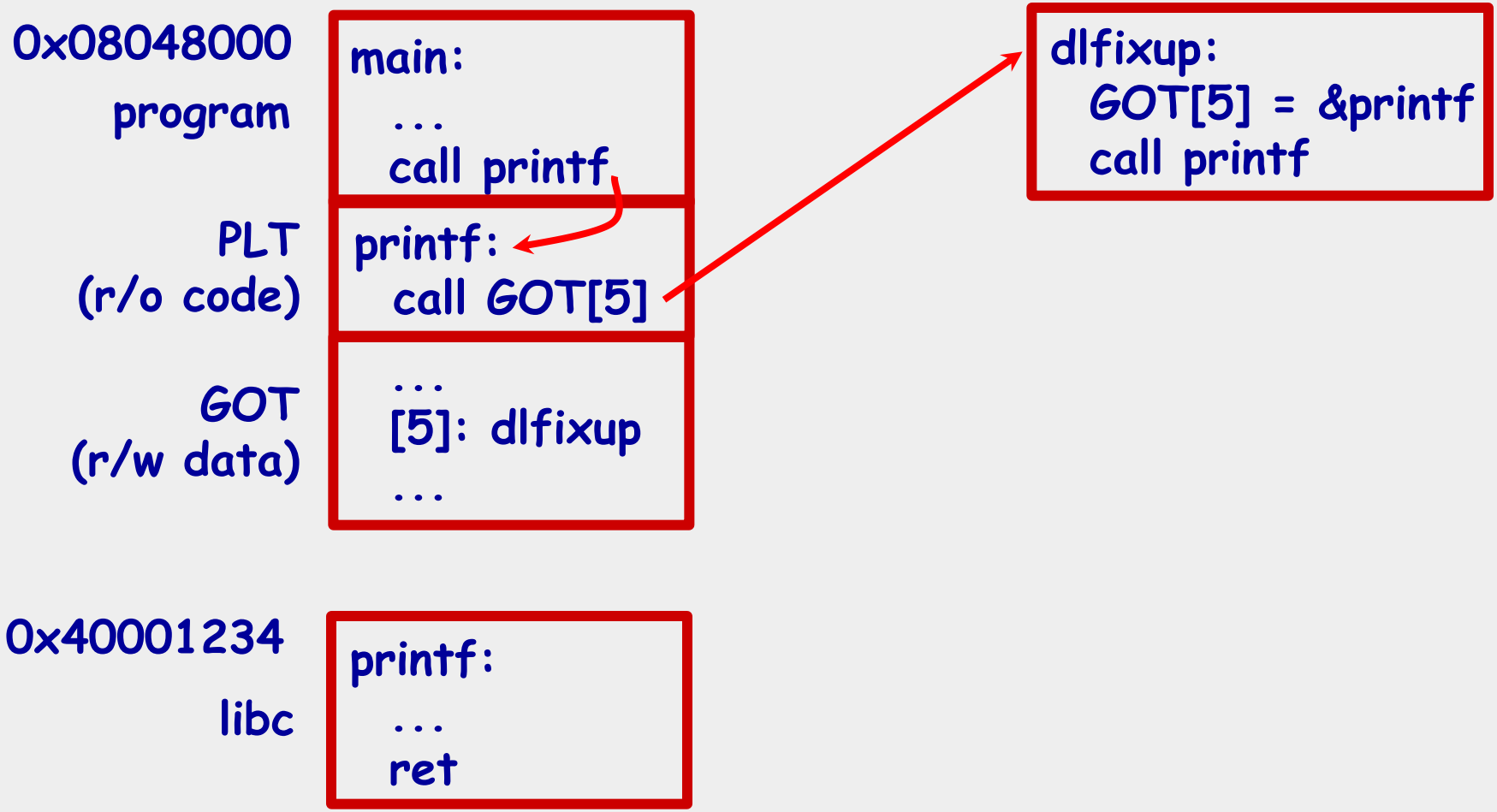


Fig from M. Rosenblum

Ưu điểm của dynamic linking

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

Dynamic loading

- Chỉ khi nào cần được gọi đến thì một thủ tục mới được nạp vào bộ nhớ chính $\Rightarrow$ tăng độ hiệu dụng của bộ nhớ (memory utilization) bởi vì các thủ tục không được gọi đến sẽ không chiếm chỗ trong bộ nhớ

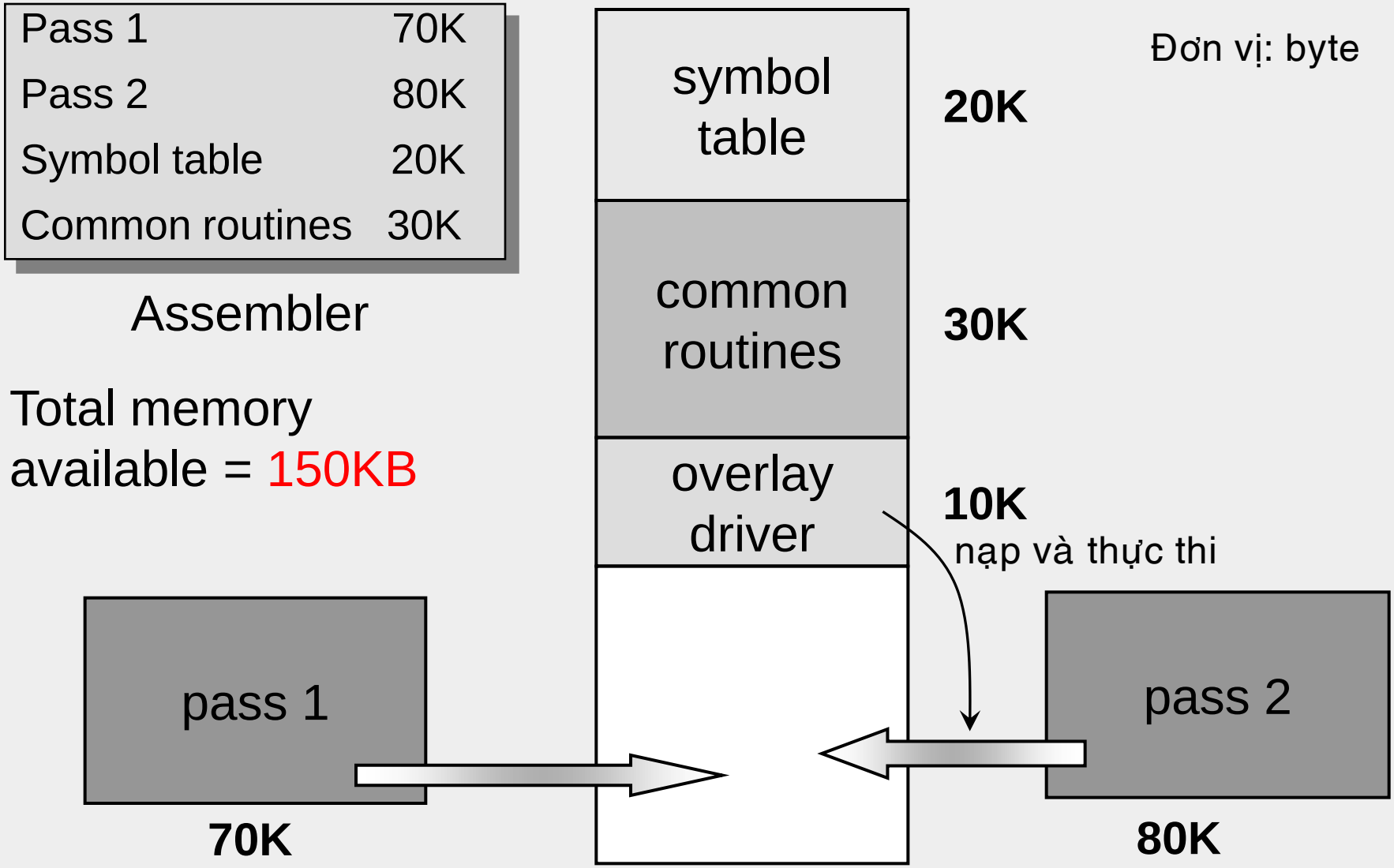
- Rất hiệu quả trong trường hợp tồn tại khối lượng lớn mã chương trình có tần suất sử dụng thấp (ví dụ các thủ tục xử lý lỗi)

- Quá trình tự điều khiển dynamic loading.
  - Hệ điều hành cung cấp một số thủ tục thư viện hỗ trợ.

Kỹ thuật overlay (1/2)

- Chỉ giữ trong bộ nhớ những lệnh hoặc dữ liệu cần thiết, giải phóng các lệnh/dữ liệu chưa hoặc không cần dùng đến.
- Kỹ thuật này rất hữu dụng khi kích thước một process lớn hơn kích thước vùng nhớ cấp cho nó.
- Quá trình tự điều khiển việc overlay (có sự hỗ trợ của thư viện lập trình)

Kỹ thuật overlay (2/2)



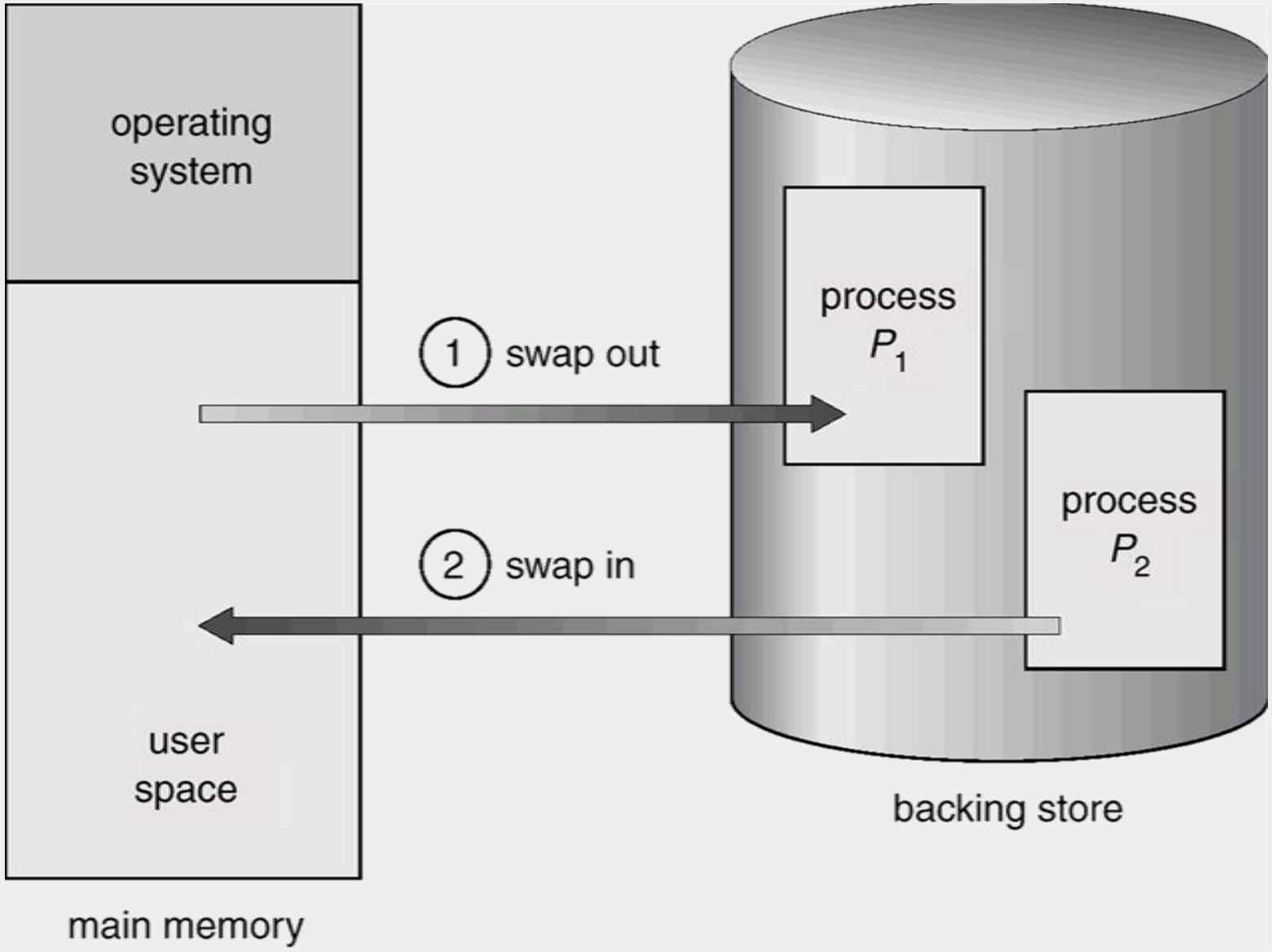
Cơ chế swapping

- Một process có thể bị swap ra khỏi bộ nhớ chính và lưu trên bộ nhớ phụ. Khi thích hợp, process được nạp lại vào bộ nhớ để tiếp tục thực thi.

  Swapping policy:

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

Minh họa cơ chế swapping



Mô hình quản lý bộ nhớ

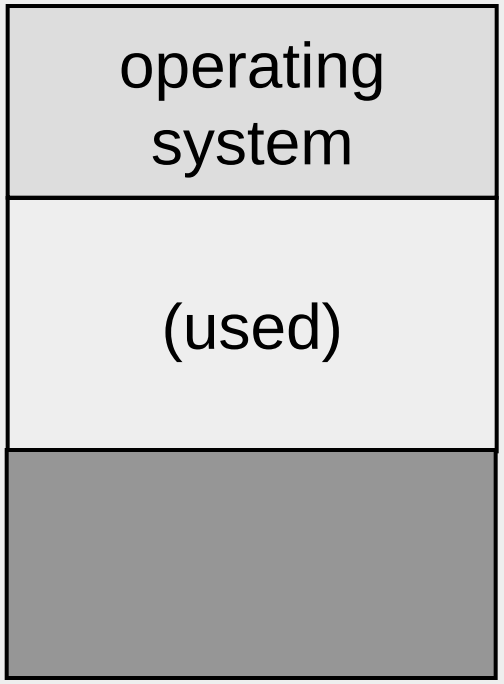
- Trong phần còn lại của chương này, mô hình quản lý bộ nhớ là một mô hình đơn giản [không có bộ nhớ ảo].
- Một process phải được nạp hoàn toàn và liên tục vào bộ nhớ thì mới thực thi được (ngoại trừ khi dùng kỹ thuật overlay).
- Các cơ chế quản lý bộ nhớ sau đây hầu như không còn được dùng trong các hệ thống hiện đại
  - Phân chia cố định (fixed partitioning)
  - Phân chia động (dynamic partitioning)

Phân mảnh

- *Phân mảnh ngoại* (external fragmentation)
  - Vùng nhớ còn trống đủ lớn để thỏa mãn một yêu cầu cấp phát, nhưng lại không liên tục
  - Có thể dùng *kết khối* (compacting) để gom lại thành vùng nhớ liên tục.

- *Phân mảnh nội* (internal fragmentation)
  - Vùng nhớ được cấp phát lớn hơn vùng nhớ yêu cầu.
    - Ví dụ: cấp một khoảng trống 18.464 byte cho một process yêu cầu 18.462 byte.
  - Thường xảy ra khi bộ nhớ thực được chia thành các khối kích thước cố định (fixed-sized block) và các process được cấp phát theo đơn vị khối.

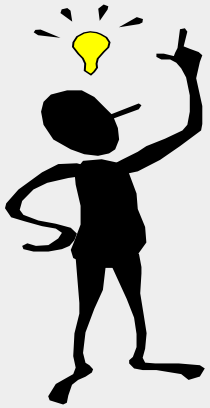
Phân mảnh nội



hole kích thước 18.464 byte

yêu cầu kế tiếp là 18.462 byte

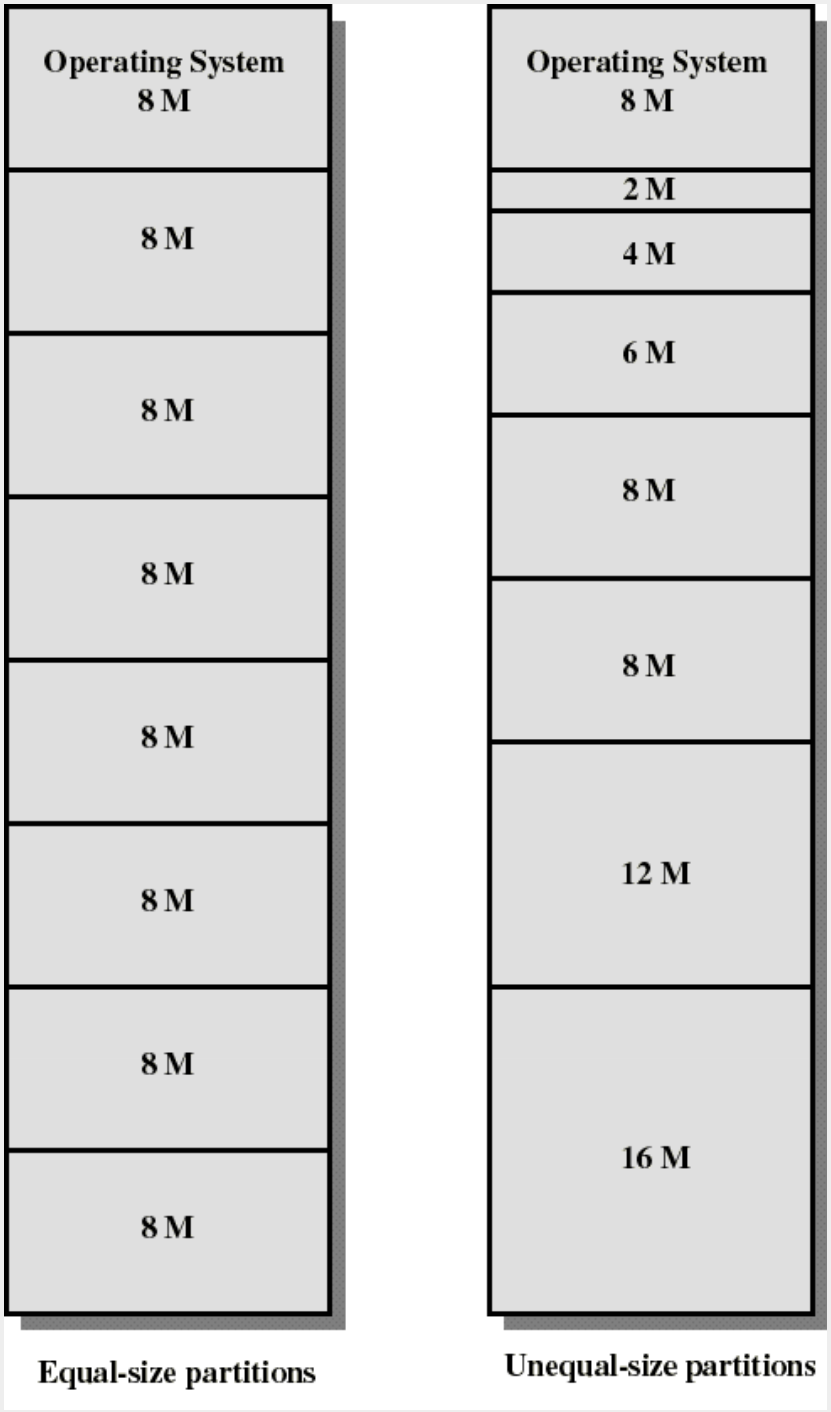
cần quản lý khoảng trống 2 byte



OS sẽ cấp phát hẳn khối 18.464 byte cho process ⇒ dư ra 2 byte không dùng

Fixed partitioning (1)

- Khi khởi động hệ thống, bộ nhớ chính được chia thành nhiều phần cố định rời nhau, gọi là các *partition*, có kích thước bằng nhau hoặc khác nhau
- Process nào có kích thước nhỏ hơn hoặc bằng kích thước partition thì có thể được nạp vào partition đó.



Fixed partitioning (2)

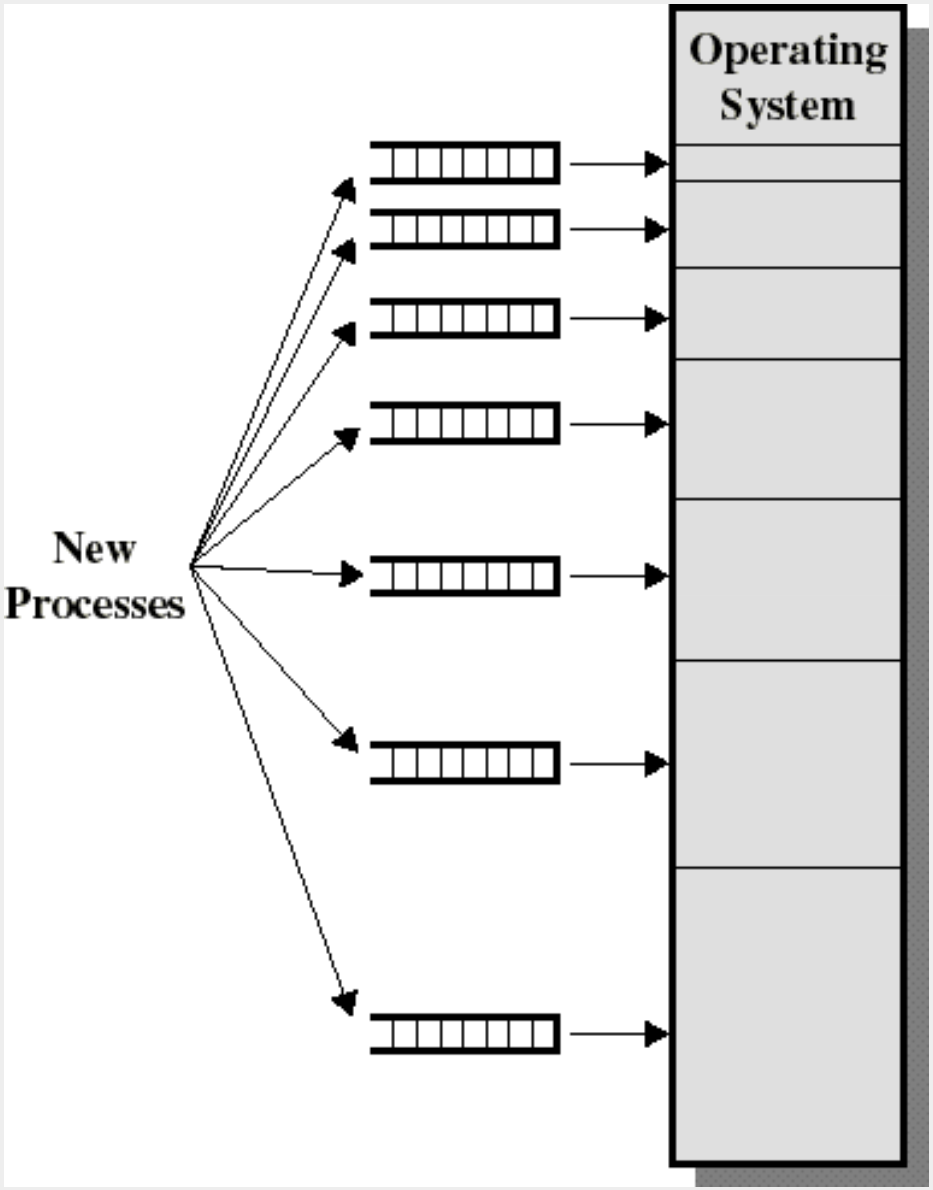
- Nếu process có kích thước lớn hơn partition thì phải dùng kỹ thuật overlay.
- Không hiệu quả do bị phân mảnh nội: một chương trình dù lớn hay nhỏ đều được cấp phát trọn một partition.

Chiến lược placement khi fixed partitioning (1/3)

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

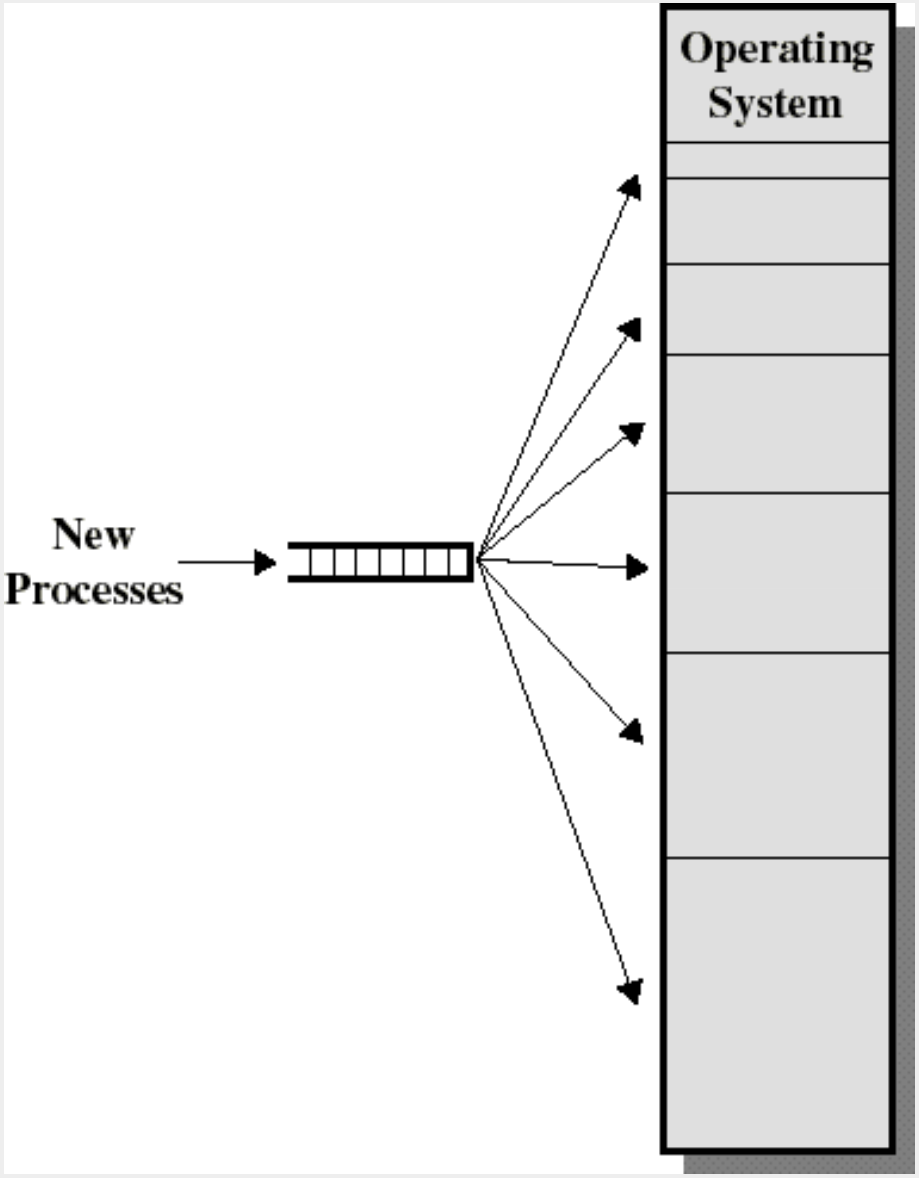
Chiến lược placement khi fixed partitioning (2/3)

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



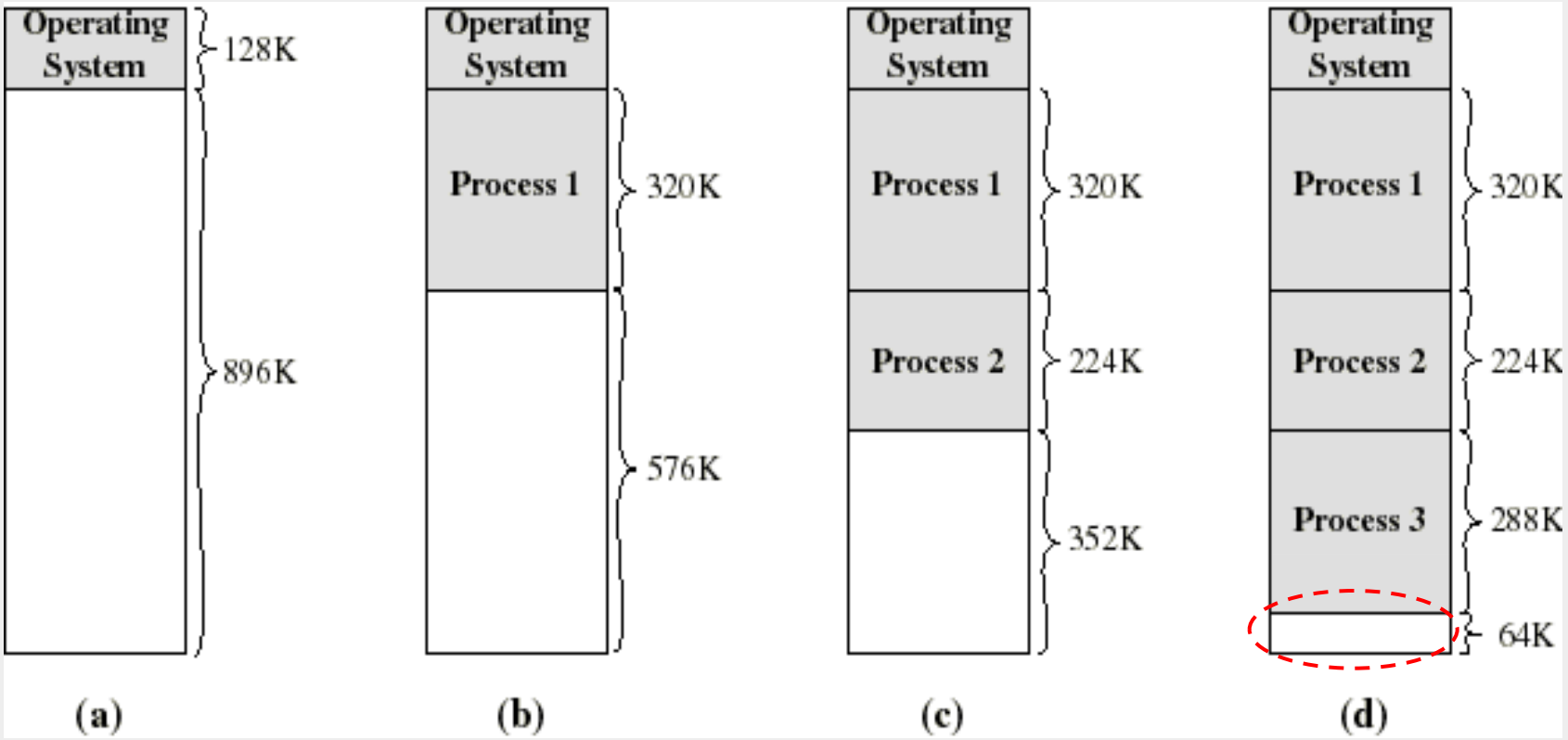
Chiến lược placement khi fixed partitioning (3/3)

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



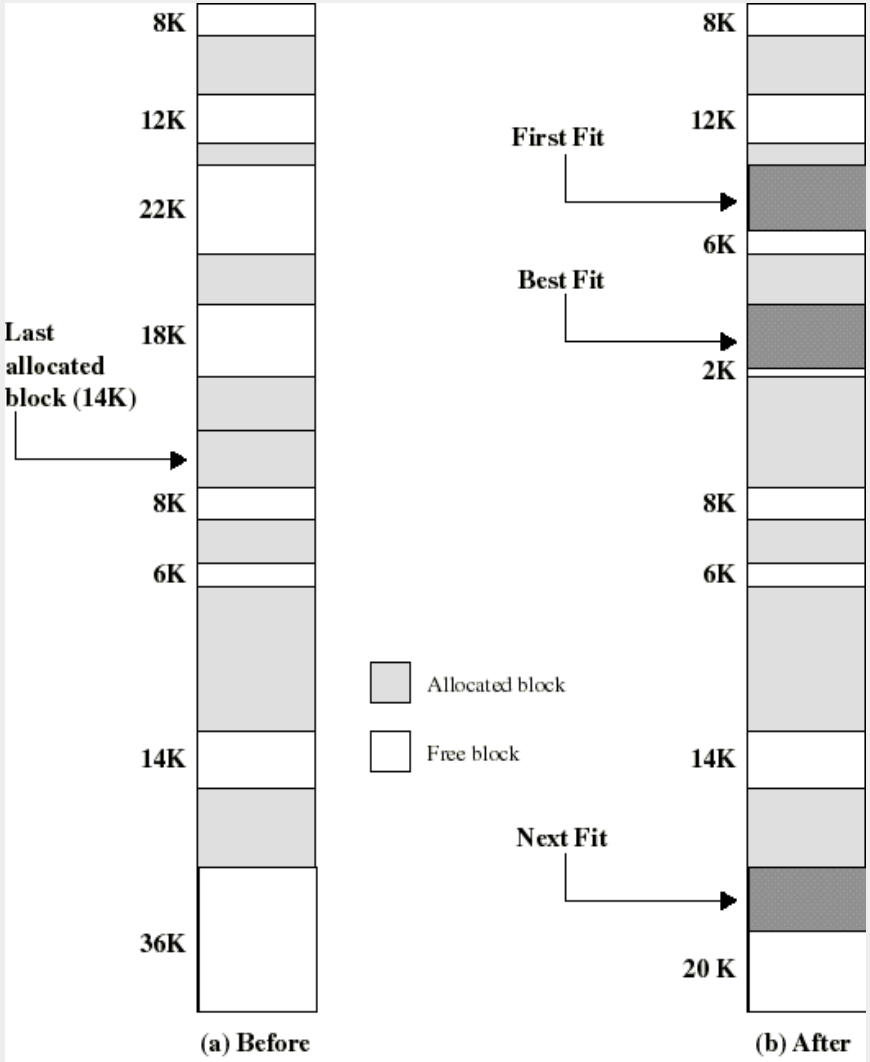
Dynamic partitioning

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



Chiến lược placement khi dynamic partitioning

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



Example Memory Configuration Before and After Allocation of 16 Kbyte Block