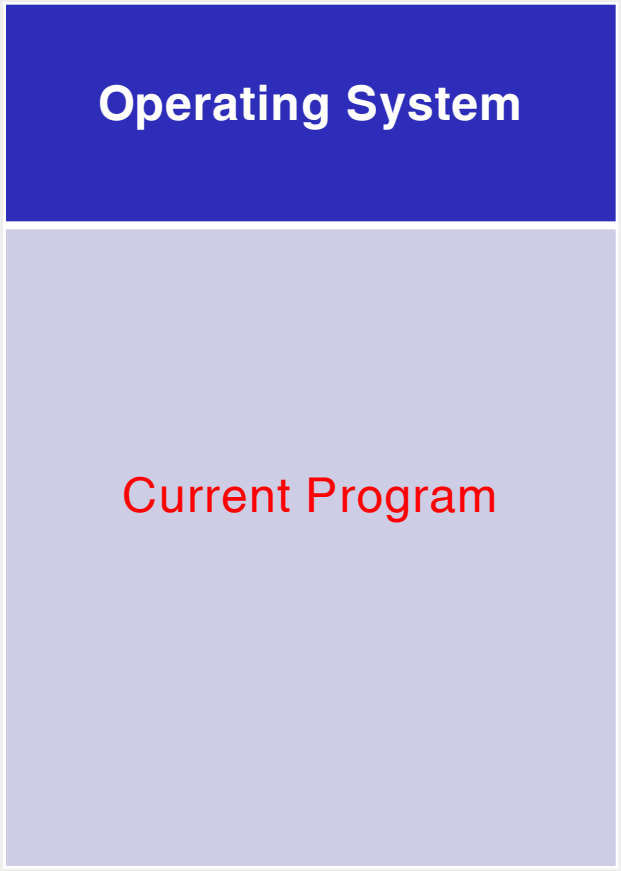
Bộ Nhớ Thực

- Các kiểu địa chỉ nhớ
- Chuyển đổi địa chỉ nhớ
- Overlay và swapping
- Vấn đề cấp phát bộ nhớ liên tục (contiguous memory allocation)
  - Giải pháp fixed partitioning
  - Giải pháp dynamic partitioning

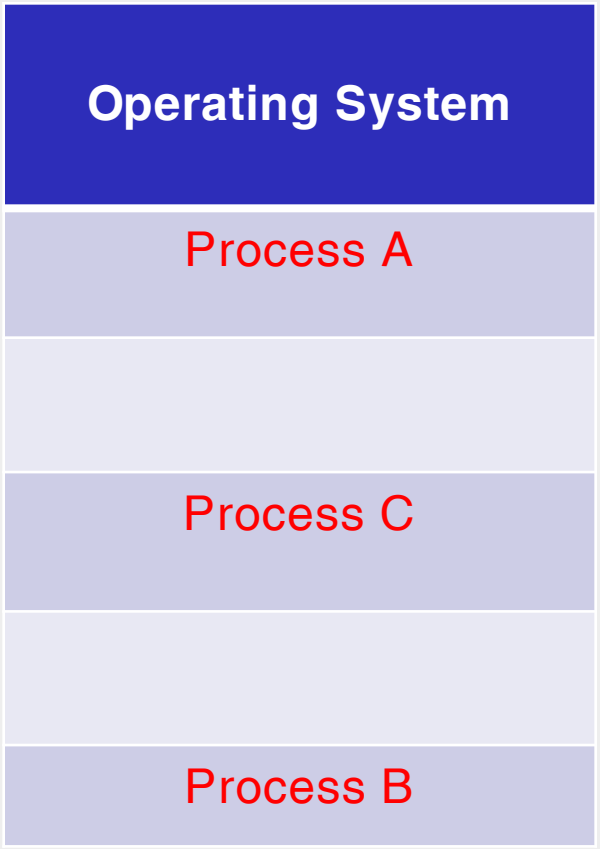
Quản lý bộ nhớ

- Kernel chiếm một vùng cố định của bộ nhớ, vùng còn lại dành để cấp phát cho các process
- Cấp phát vùng nhớ cho các process sao cho hệ thống hoạt động hiệu quả
  - Vd: Nạp càng nhiều process vào bộ nhớ càng tốt để gia tăng mức độ multiprogramming
- Quản lý bộ nhớ
  - Cấp phát vùng nhớ cho các process
  - Bảo vệ: kiểm tra truy xuất bộ nhớ có hợp lệ không
  - Chia sẻ: cho phép các process chia sẻ vùng nhớ chung
  - Chuyển đổi địa chỉ luận lý sang địa chỉ vật lý

Layout bộ nhớ



Uni-programming



Multi-programming

Các kiểu địa chỉ nhớ (1/2)

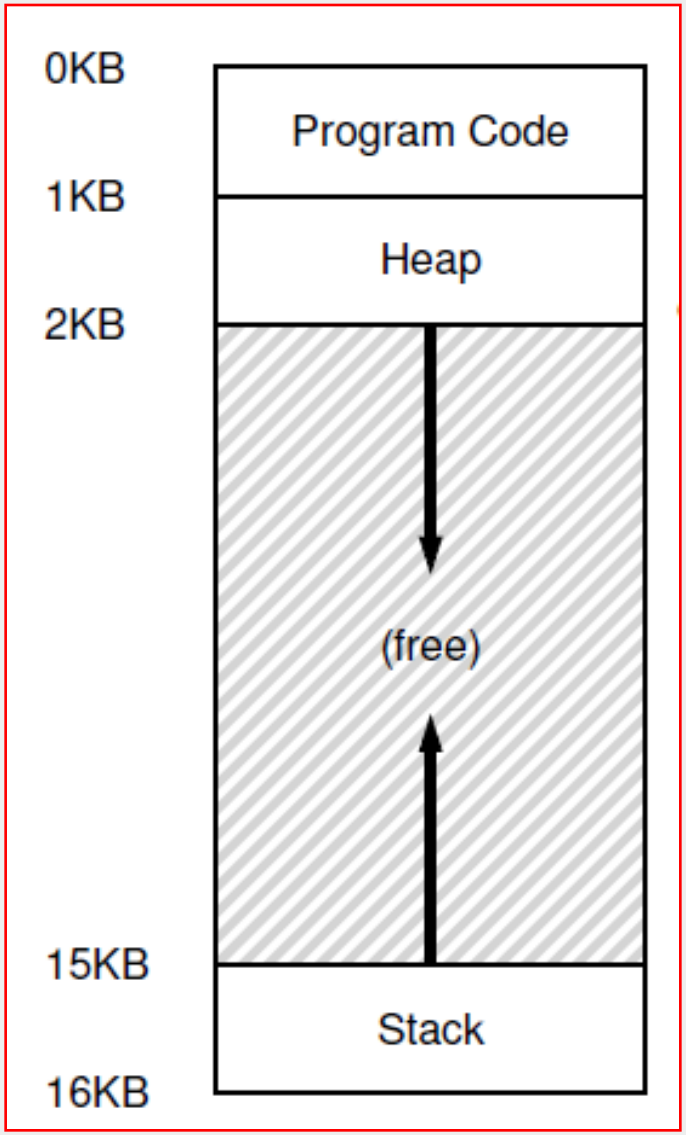
- *Địa chỉ vật lý* -- physical (memory) address -- là địa chỉ mà CPU, hay MMU (nếu có), gửi đến bộ nhớ chính
- *Địa chỉ luận lý* (logical address) là địa chỉ mà một quá trình sinh ra
- Các địa chỉ sinh bởi trình biên dịch (compiler) là
  - *tương đối* hay khả tái định vị (relocatable): compiler giả thiết không gian địa chỉ của *đơn vị biên dịch* (compilation unit) bắt đầu từ địa chỉ 0

  hoặc
  - *tuyệt đối*: kết quả biên dịch có thể nạp được ngay vào bộ nhớ để thực thi; ít được dùng

Các kiểu địa chỉ nhớ (2/2)

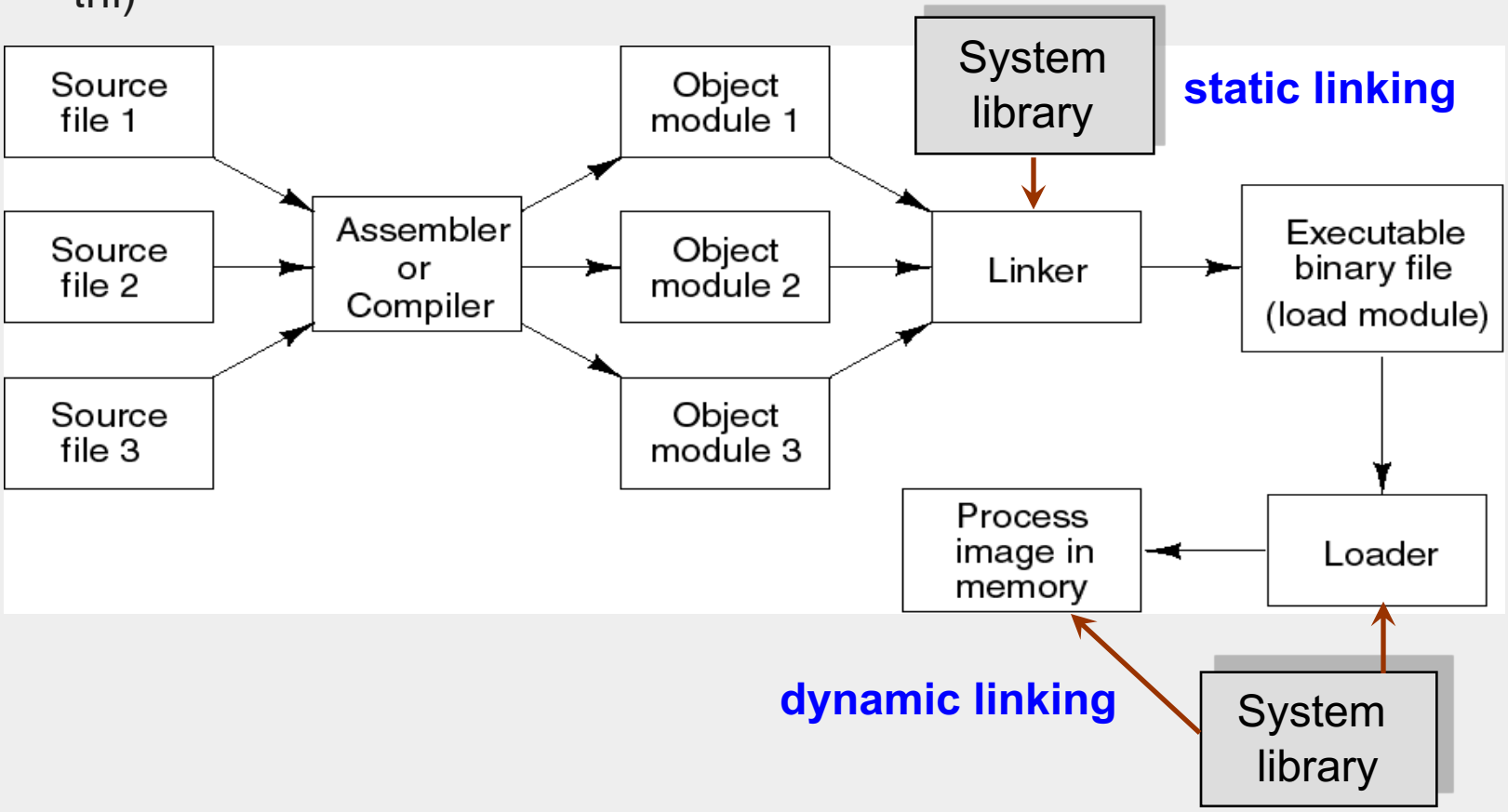
- Khi một lệnh được thực thi, các địa chỉ luận lý phải được chuyển đổi thành địa chỉ vật lý
  - Sự chuyển đổi này thường có sự hỗ trợ của phần cứng để đạt hiệu năng cao

Không gian nhớ quá trình



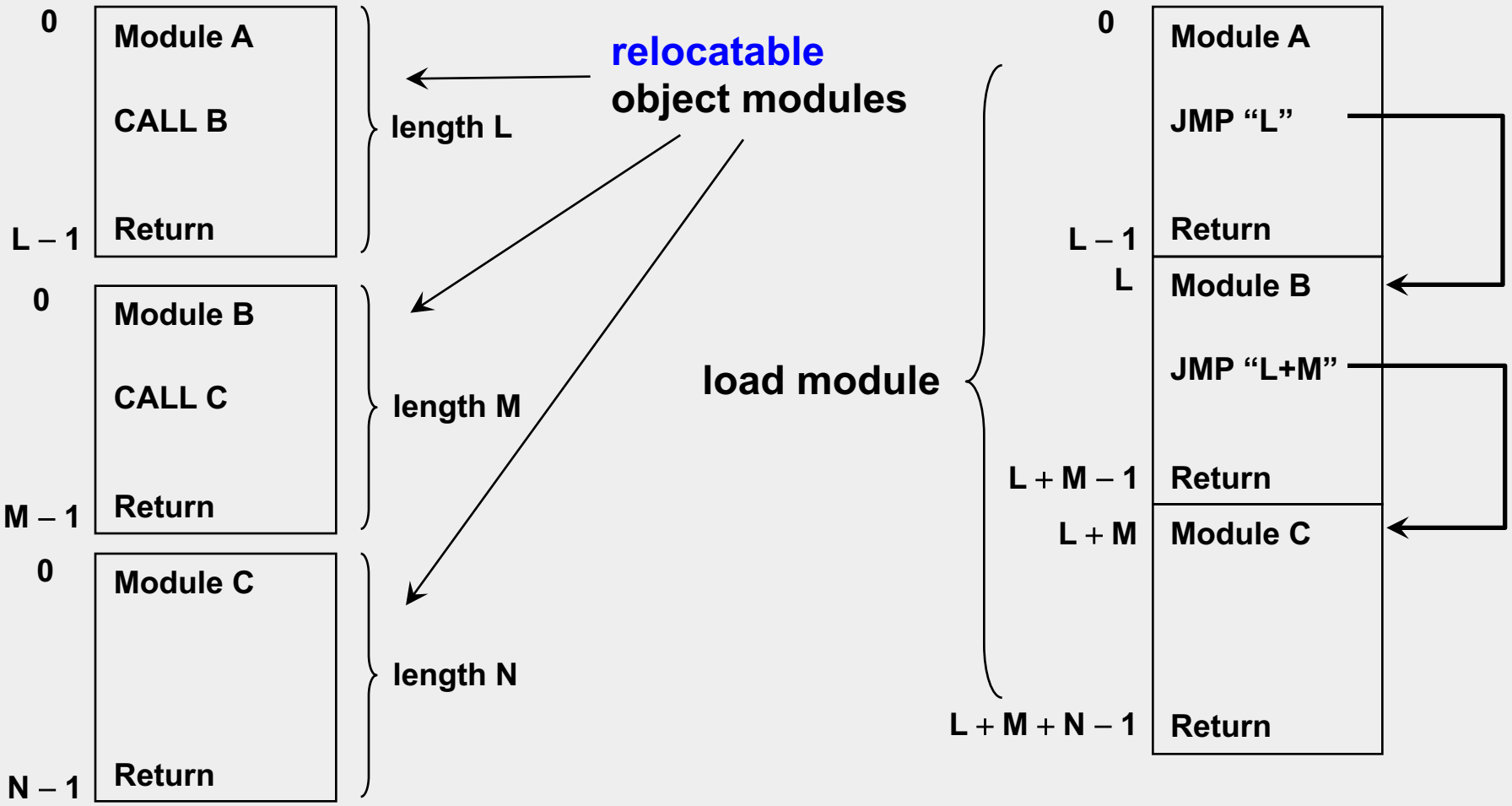
Từ mã nguồn đến file thực thi được

- Linker: kết hợp các object module thành một file thực thi được
  - 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)



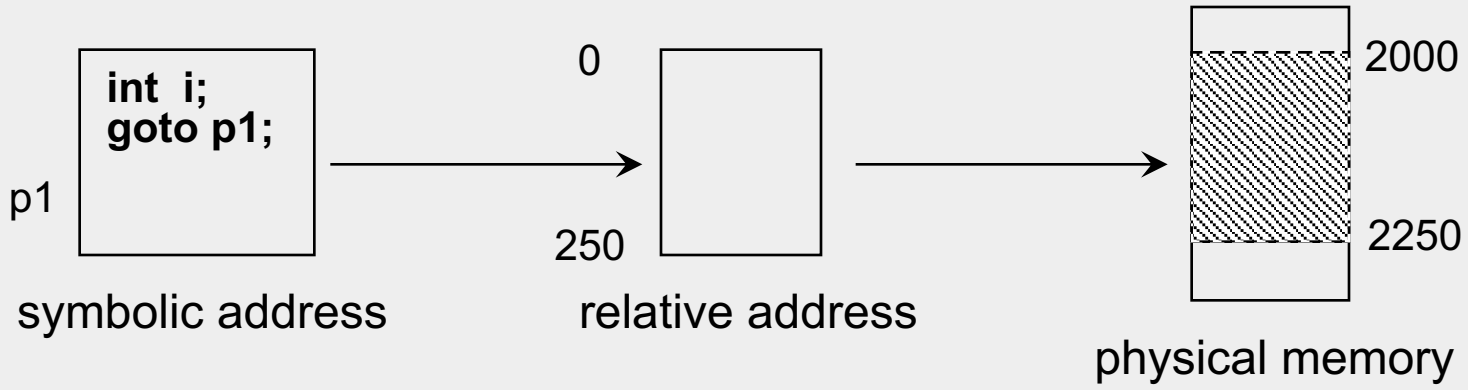
Thực hiện (static) linking

- Linker chuyển đổi địa chỉ tương đối sang địa chỉ tuyệt đối



Chuyển đổi địa chỉ

- *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…)
  - Vào 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



Sinh địa chỉ vật lý

- Trong khi thực thi
  - Địa chỉ được chuyển đổi động trong khi thực thi
  - Không gian địa chỉ vật lý có thể noncontiguous
  - Cần có phần cứng để chuyển đổi địa chỉ ảo sang địa chỉ vật lý được nhanh
    - “Phân trang” (“paging”)
    - “Phân đoạn” (“segmentation”)
  - Rất phổ biến hiện nay

Tiết kiệm vùng nhớ

- Các kỹ thuật
  - Dynamic linking
  - Dynamic loading
  - Overlay
  - Swapping

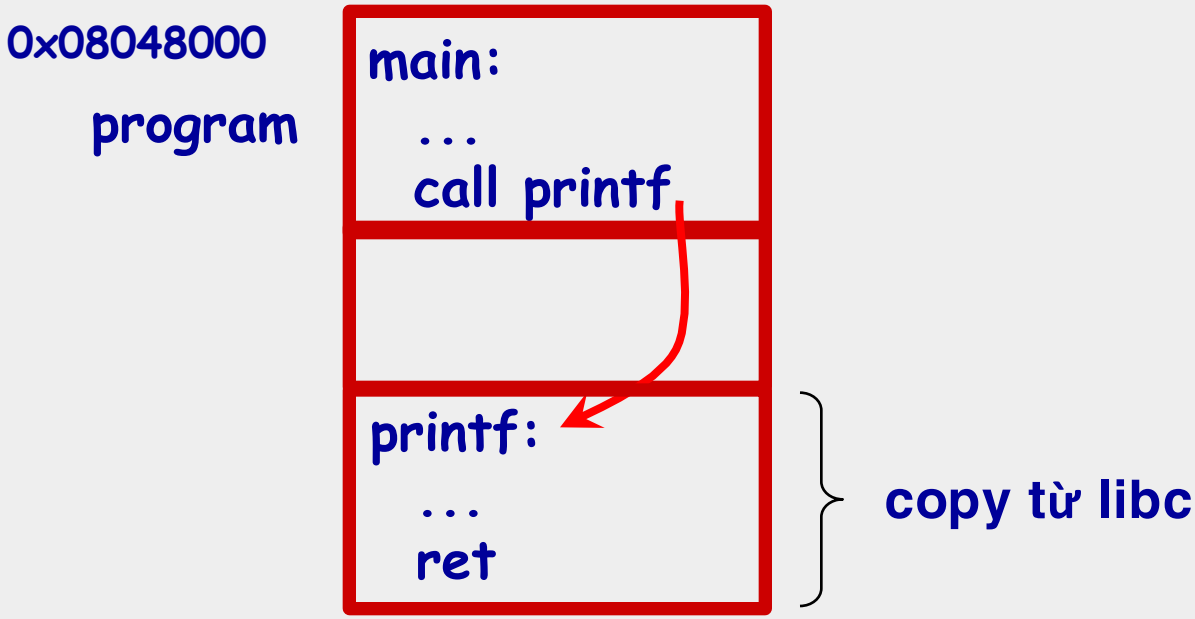
Dynamic linking (1)

Trong dynamic linking

- Việc link một load module *L* đến một *module ngoài* (external module) được thực hiện sau khi đã tạo xong *L*
  - 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 các routine của external module
  - Khi process gọi routine lần đầu, stub sẽ kích hoạt nạp routine vào bộ nhớ (nếu routine chưa được nạp trước đó), thay thế địa chỉ mình bằng địa chỉ routine, và gọi routine để thực thi
  - Các lần gọi routine sau sẽ xảy ra bình thường, không tốn overhead

Dynamic linking (2)

- Nhắc lại static linking



Dynamic linking (3)

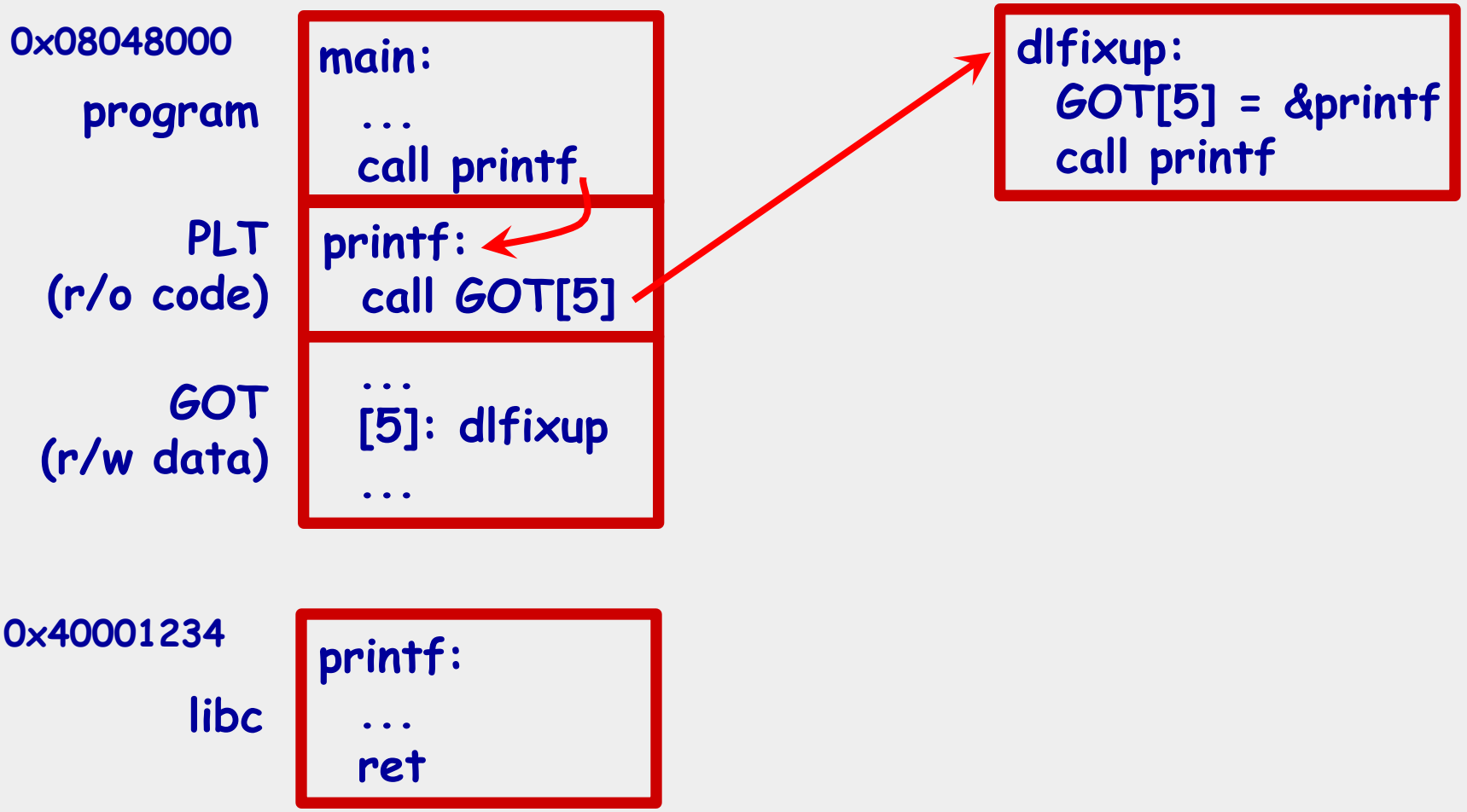


Fig from M. Rosenblum

Ưu điểm của dynamic linking

- Chương trình thực thi có thể gọi phiên bản mới (ví dụ phiên bản đã sửa lỗi) của external module mà không cần được sửa đổi và/hay biên dịch lại
- *Chia sẻ mã* (code sharing): chỉ cần nạp external module vào bộ nhớ một lần
  - Các process sử dụng dynamic link với external module này chia sẻ vùng mã của external module $\Rightarrow$ tiết kiệm không gian nhớ và không gian đĩa

Dynamic linking

- Các external module thường là thư viện cung cấp các tiện ích (như libc)
- Stub cần sự hỗ trợ của OS
  - Kiểm tra xem routine đã được nạp vào bộ nhớ chưa

Dynamic loading (1)

- Chỉ khi nào cần được gọi đến thì một thủ tục mới được nạp vào bộ nhớ chính
  - Các thủ tục không được gọi đến sẽ không chiếm chỗ trong bộ nhớ
- Rất hiệu quả khi chương trình có khối lượng lớn mã có tần suất sử dụng thấp (ví dụ các thủ tục xử lý lỗi)
- Chính quá trình tự điều khiển dynamic loading
  - Hệ điều hành cung cấp một số thủ tục thư viện hỗ trợ

Dynamic loading (2)

- Các thủ tục để người dùng thực hiện dynamic loading trong UNIX:
  - dlopen() – Open một file thư viện
  - dlsym() – Dò tìm một ký hiệu (symbol) trong file thư viện
  - dlclose() – Close một file thư viện

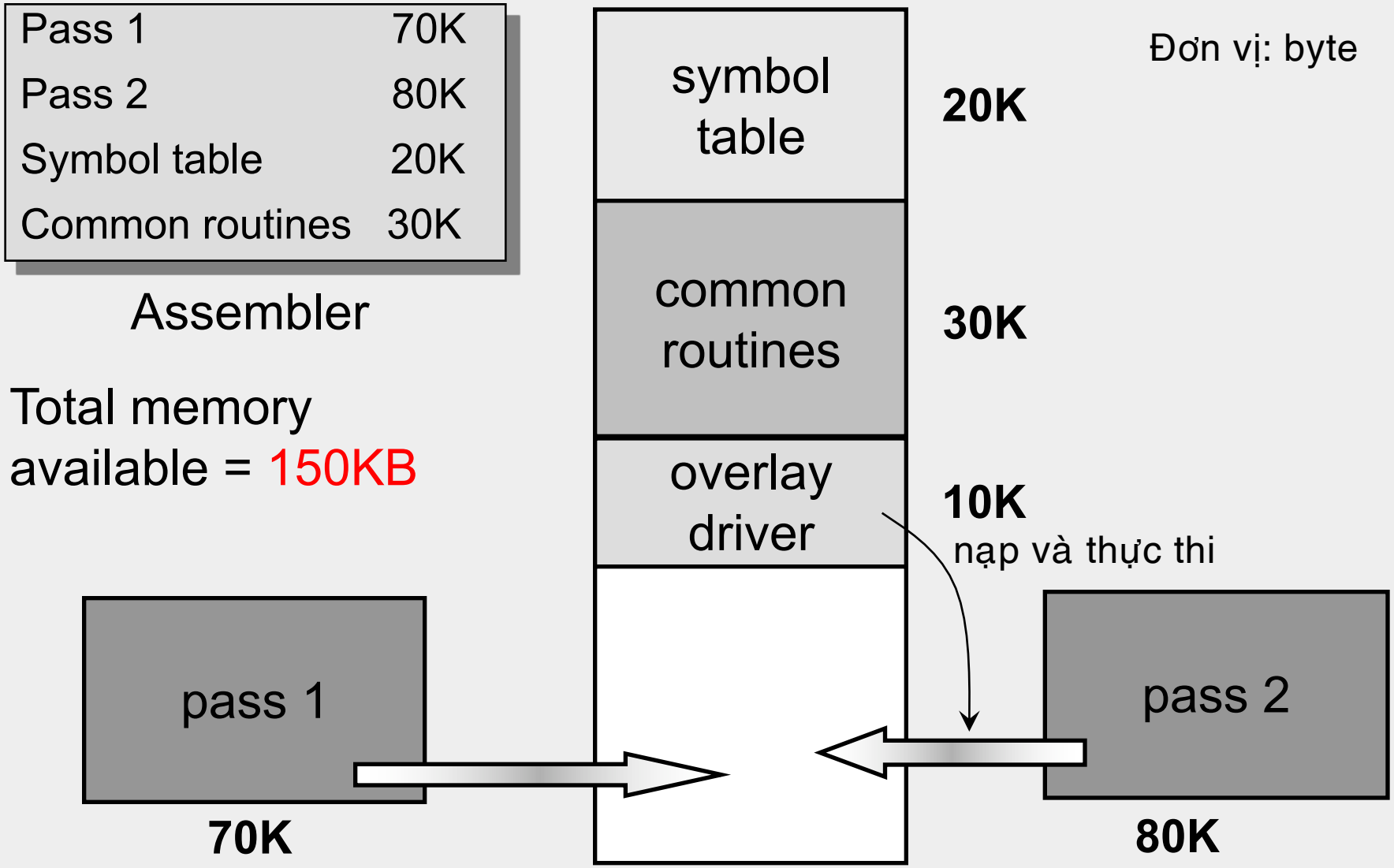
Dynamic loading – Ví dụ

```
#include <stdio.h>
#include <dlfcn.h>

int main() {
    void *handle;
    void (*somefunc)(int, int);
    char *error;

    /* dynamically load the shared lib that contains somefunc() */
    handle = dlopen("./libvector.so", RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    /* get a pointer to the somefunc() function we just loaded */
    somefunc = dlsym(handle, "somefunc");
    if ((error = dlerror()) != NULL) {
        fprintf(stderr, "%s\n", error);
        exit(1);
    }

    /* Now we can call somefunc() just like any other function */
    somefunc(42, 38);

    /* unload the shared library */
    if (dlclose(handle) < 0) {
        fprintf(stderr, "%s\n", dlerror());
        exit(1);
    }
    return 0;
}
```

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)
- Có thể được xem là tiền thân của kỹ thuật “bộ nhớ ảo”

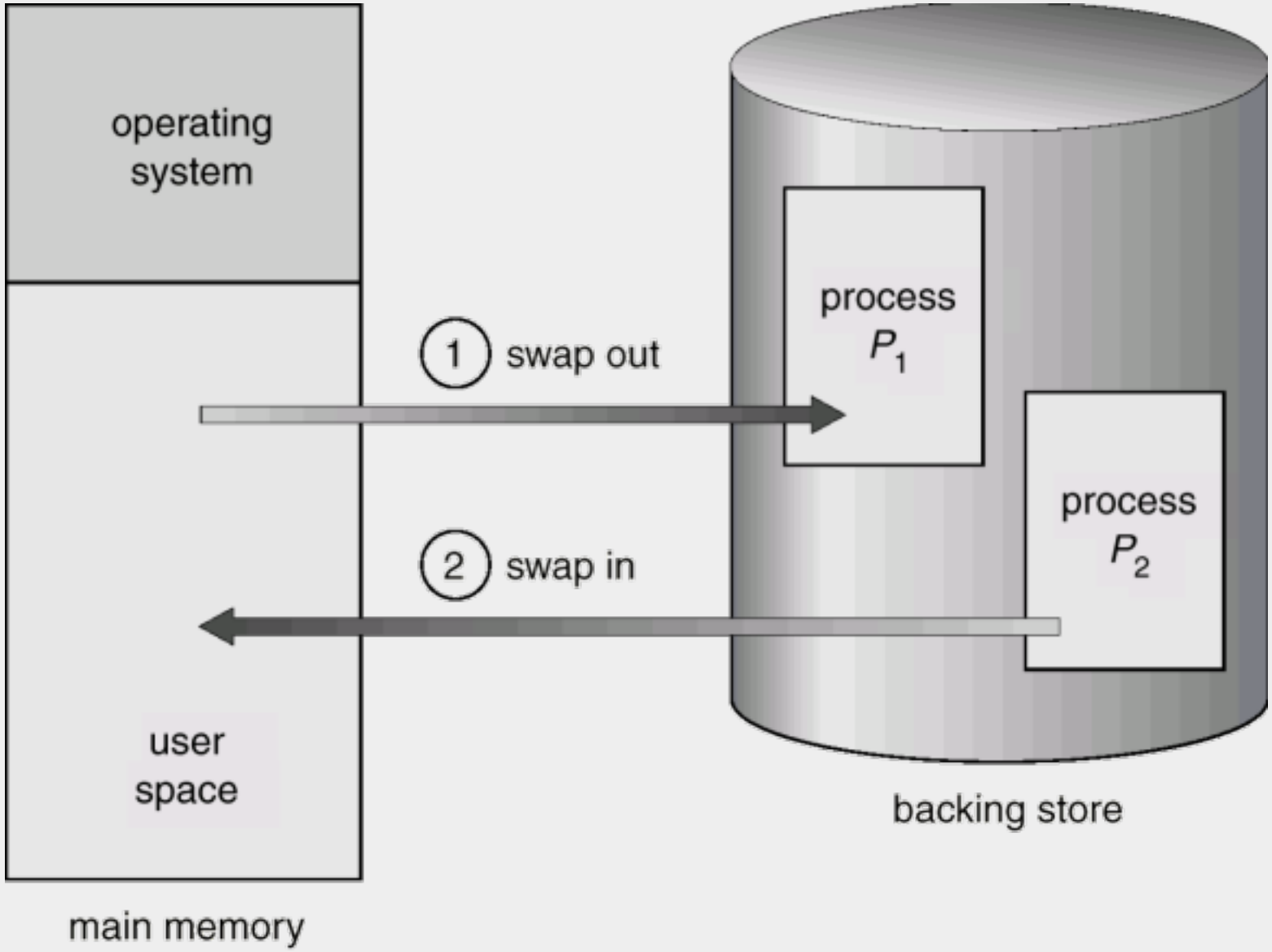
Kỹ thuật overlay (2/2)



Swapping

- Cơ chế: di chuyển một process khỏi bộ nhớ chính và lưu trên bộ nhớ phụ (swap out). Khi thích hợp, nạp process vào bộ nhớ (swap in) để có thể tiếp tục thực thi
- Chính sách:
  - *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

Swapping -- Cơ chế



Vấn đề cấp phát bộ nhớ liên tục

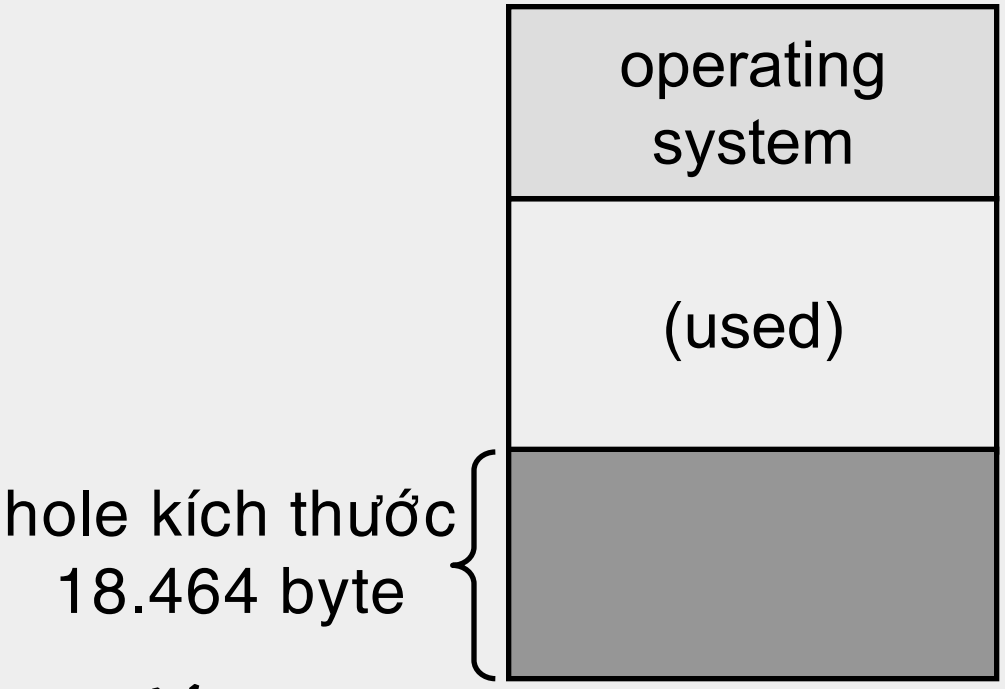
- Trong phần còn lại của chương này, mô hình quản lý bộ nhớ là một mô hình đơn giản [không dùng "bộ nhớ ảo"!]
  - Một process phải được nạp hoàn toàn vào bộ nhớ (ngoại trừ khi dùng kỹ thuật overlay) và nằm liên tục (contiguous)
- Sẽ thảo luận các giải pháp cấp phát bộ nhớ sau
  - *Phân chia cố định* (fixed partitioning)
  - *Phân chia động* (dynamic partitioning)

Hiện tượng phân mảnh

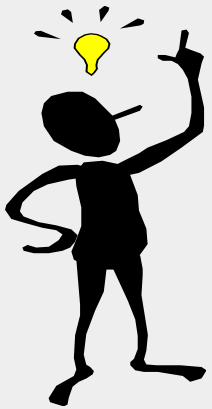
- *Phân mảnh ngoại* (external fragmentation)
  - Vùng nhớ còn trống đủ lớn để thỏa mãn một yêu cầu cấp phát, nhưng lại không liên tục
  - Dùng *kết khối* (compacting), nếu có thể, để gom lại thành vùng nhớ liên tục

- *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



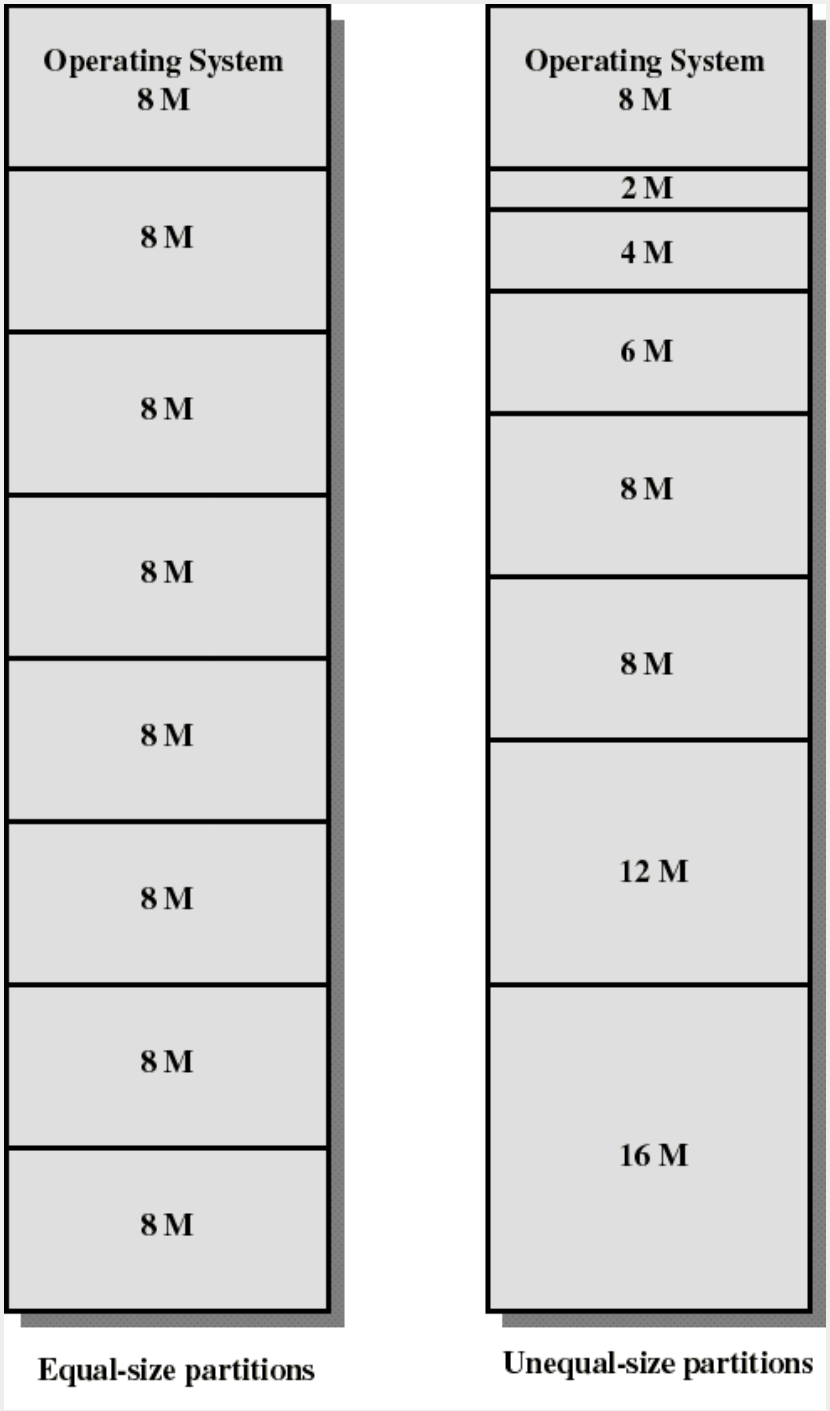
yêu cầu kế tiếp là 18.462 byte



Để tránh overhead quản lý chỉ 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



Giải pháp fixed partitioning (2)

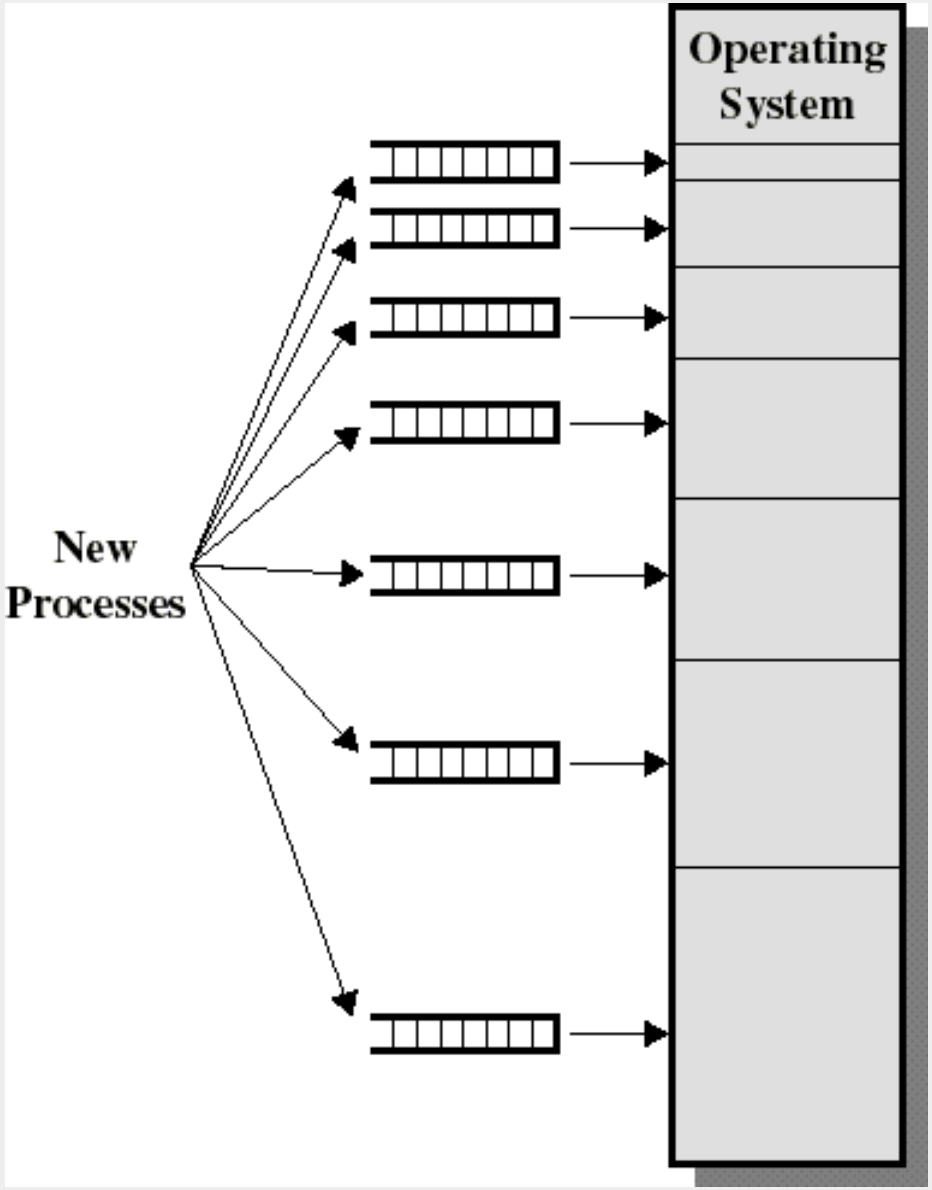
- Nếu process có kích thước lớn hơn partition thì phải dùng kỹ thuật overlay
- Không hiệu quả do bị phân mảnh nội: một quá trình dù lớn hay nhỏ đều được cấp phát trọn một partition

Chiến lược placement khi fixed partitioning (1/3)

- Trường hợp các 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 có process đang bị blocked $\Rightarrow$ swap out process đó ra bộ nhớ phụ, dành partition cho process mới

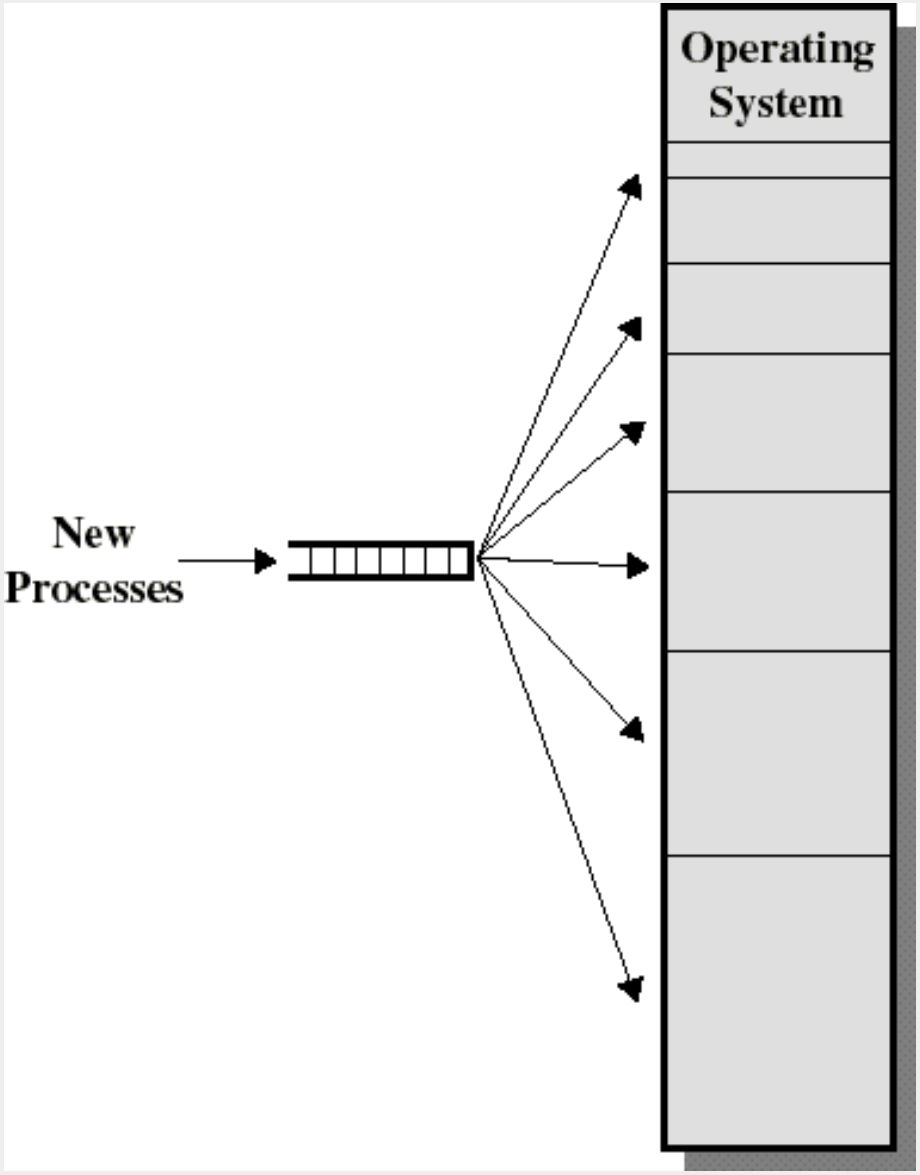
Chiến lược placement khi fixed partitioning (2/3)

- Trường hợp các partition có kích thước không bằng nhau
  - Giải pháp 1
    - Gán mỗi process vào partition nhỏ nhất (trống hay chưa trống) đủ chứa nó [best fit]
    - Có hàng đợi cho mỗi partition
  - Điểm yếu của giải pháp: có thể có một số hàng đợi trống (vì kích thước partition 'quá lớn' đối với process) và một số hàng đợi dài



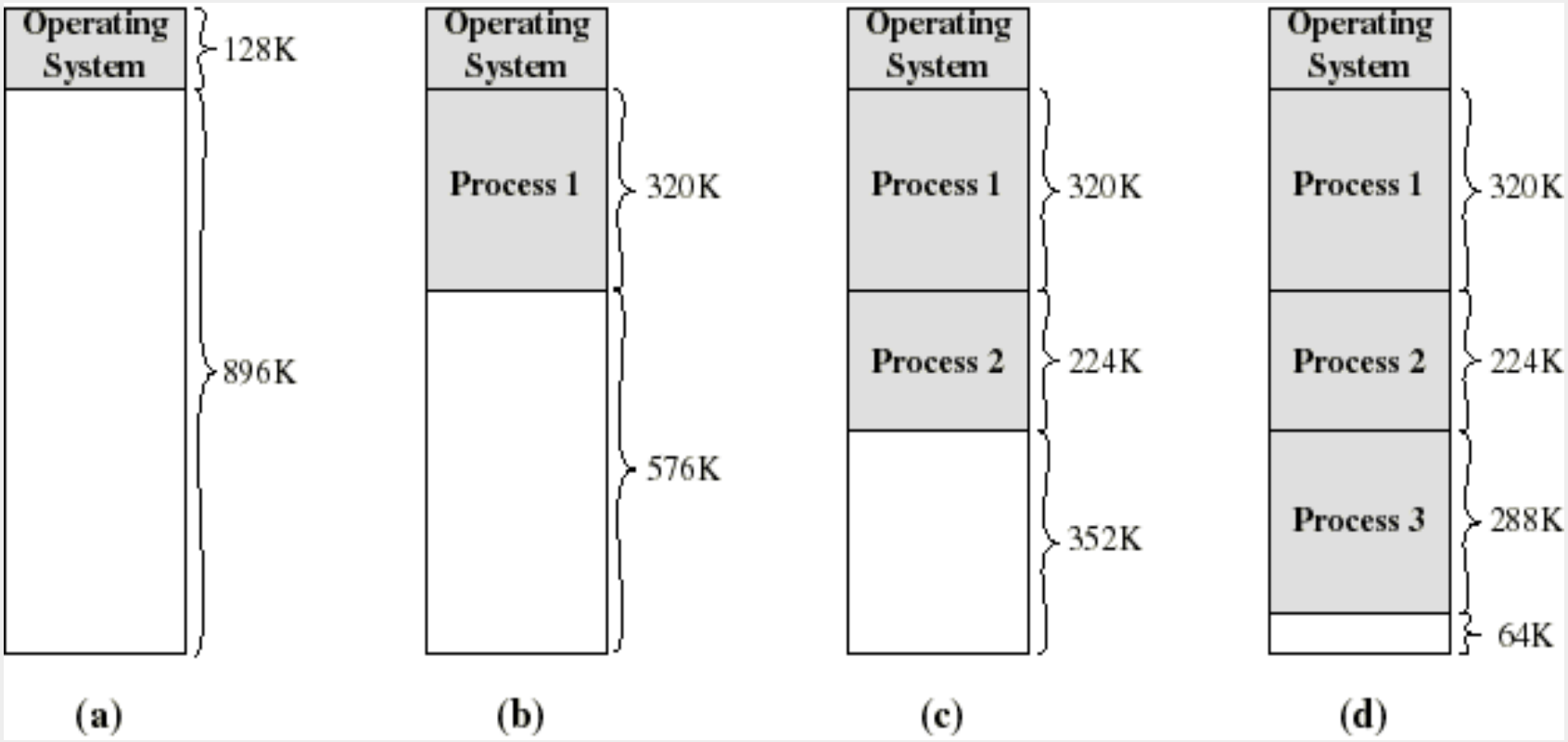
Chiến lược placement khi fixed partitioning (3/3)

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



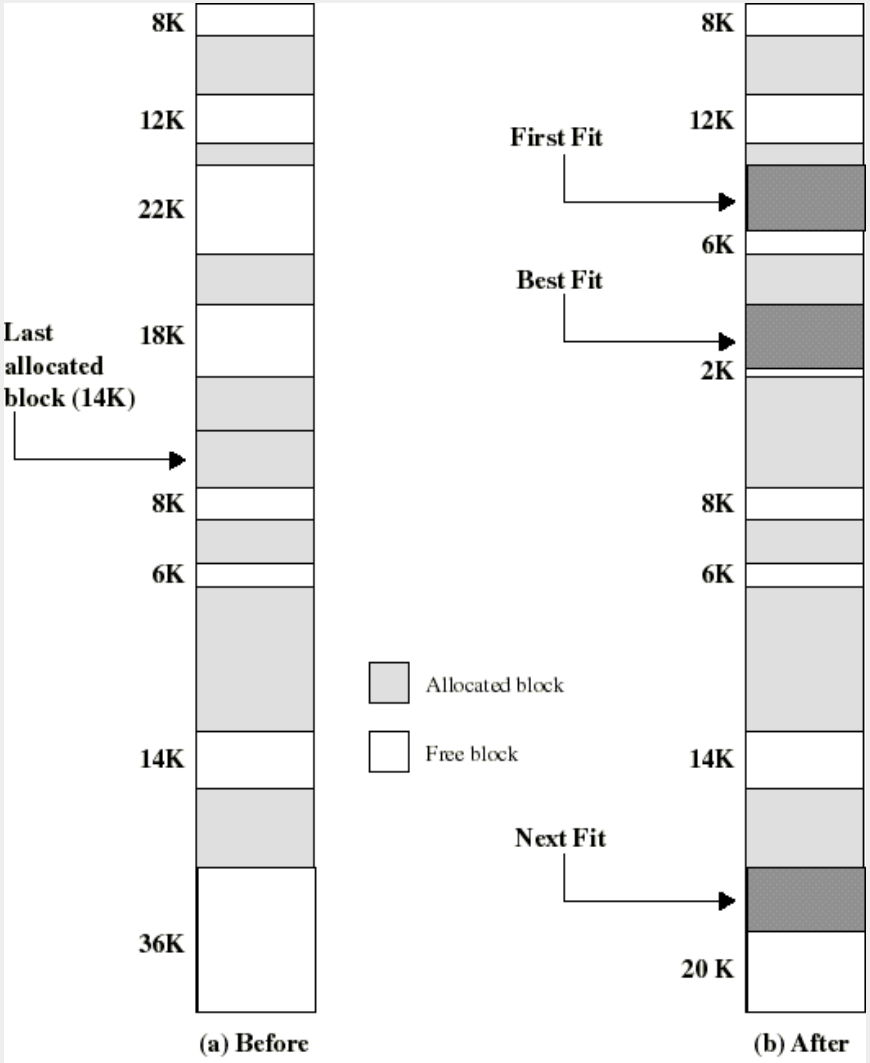
Giải pháp dynamic partitioning

- 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

Nhận xét

- Cả hai giải pháp fixed và dynamic partitioning hầu như không còn được dùng trong các hệ thống hiện đại