Hệ Điều Hành

Thời gian:

- Lý thuyết: 45 tiết
- Thực hành: 30 tiết

Điểm số:

- Điểm thi giữa kỳ: 20%
- Điểm làm bài tiểu luận: 30%
- Điểm thi cuối HK: 50%

- **Khoa Kỹ thuật máy tính**
- GV: TS. **Vũ Đức Lung**
- Email: lungvd@uit.edu.vn

Nội dung môn học

Chương 1: Tổng quan về hệ điều hành
Chương 2: Cấu trúc Hệ điều hành
Chương 3: Quản lý tiến trình (Processes)
Chương 4: Định thời CPU
Chương 5: Đồng bộ hóa tiến trình
Chương 6: Tắc nghẽn (Deadlocks)
Chương 7: Quản lý bộ nhớ
Chương 8: Bộ nhớ ảo

Tài liệu tham khảo

1. Trần Hạnh Nhi, Lê Khắc Nhiên Ân. Giáo trình hệ điều hành. Trung tâm phát triển công nghệ thông tin-ĐHQG.HCM, 2005.
2. Nguyễn Phú Trường. Giáo trình hệ điều hành. ĐH Cần Thơ, 2005.
3. Silberschatz, Galvin, Gagne. Operating System Concepts. Sixth edition, John Wiley & Sons,2003
4. Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.

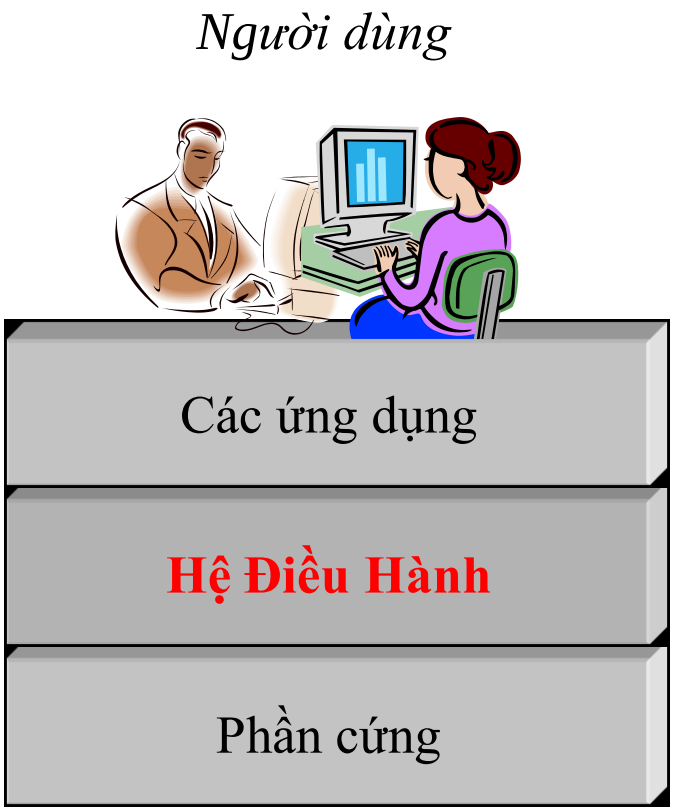
Chương I:
Tổng quan hệ điều hành

1.1. Tổng quan

- **Giới thiệu**
  - Định nghĩa hệ điều hành
  - Cấu trúc hệ thống máy tính
  - Các chức năng chính của hệ điều hành

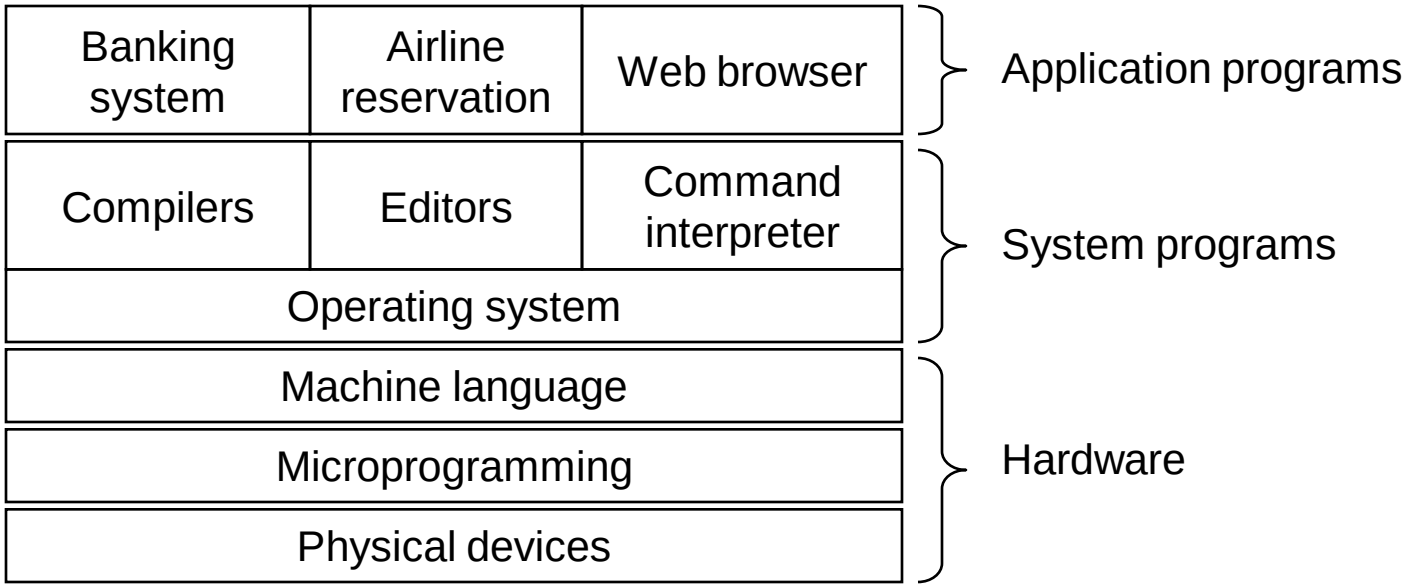
Định nghĩa

- Hệ điều hành là gì?
  - *Chương trình* trung gian giữa phần cứng máy tính và người sử dụng, có chức năng *điều khiển và phối hợp vệc sử dụng phần cứng* và cung cấp các *dịch vụ cơ bản* cho các ứng dụng.
- Mục tiêu
  - Giúp người dùng dễ dàng sử dụng hệ thống.
  - Quản lý và cấp phát tài nguyên hệ thống một cách hiệu quả.



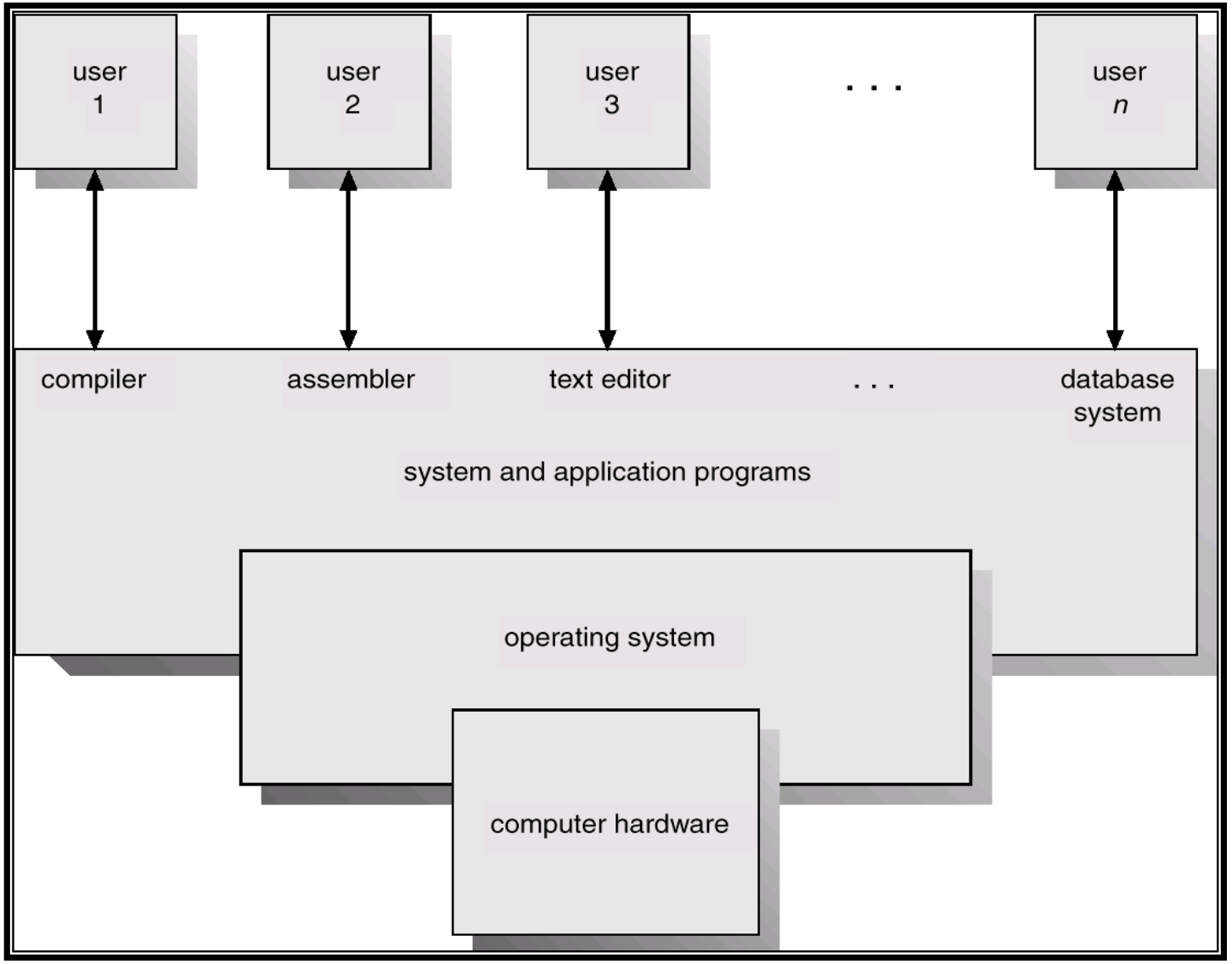
Định nghĩa (tt)

Hình chính xác hơn



Hình của Dror G. Feitelson

Các thành phần của hệ thống



Các thành phần của hệ thống (tt)

- Phần cứng (hardware)

  Bao gồm các tài nguyên cơ bản của máy tính như CPU, bộ nhớ, các thiết bị I/O,...

- Hệ điều hành (operating system)

  Phân phối tài nguyên, điều khiển và phối hợp các hoạt động của các chương trình trong hệ thống.

- Chương trình ứng dụng (application programs)

  Sử dụng tài nguyên hệ thống để giải quyết một vấn đề tính toán nào đó của người sử dụng, ví dụ: compilers, database systems, video games, business programs.

- Dữ liệu

Các chức năng chính của OS

- Phân chia thời gian xử lý và định thời CPU
- Phối hợp và đồng bộ hoạt động giữa các processes (coordination & synchronization)
- Quản lý tài nguyên hệ thống (thiết bị I/O, bộ nhớ, file chứa dữ liệu,…)
- Thực hiện và kiểm soát access control, protection
- Duy trì sự nhất quán (integrity) của hệ thống, kiểm soát lỗi và phục hồi hệ thống khi có lỗi (error recovery)
- Cung cấp giao diện làm việc cho users

Các dạng HĐH

- Same machine, different operating systems:
  - IBM PC: DOS, Linux, NeXTSTEP, Windows, SCO Unix
  - DEC VAX: VMS, Ultrix-32, 4.3 BSD UNIX
- Same OS, different machines: UNIX
  - PC (XENIX 286, APPLE A/UX)
  - CRAY-Y/MP (UNICOS - AT&T Sys V)
  - IBM 360/370 (Amdahl UNIX UTS/580, IBM UNIX AIX/ESA)
- Windows NT, XP, 2000, 2003
  - Intel i386 (i486 an NT 4.0), Alpha, PowerPC, MIPS, Itanium

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

Dưới gĩc độ loại máy tính

- Hệ điều hành dành cho máy MainFrame
- Hệ điều hành dành cho máy Server
- Hệ điều hành dành cho máy nhiều CPU
- Hệ điều hành dành cho máy tính cá nhân (PC)
- Hệ điều hành dành cho máy PDA (Embedded OS - hệ điều hành nhúng)
- Hệ điều hành dành cho máy chuyên biệt
- Hệ điều hành dành cho thẻ chíp (SmartCard)

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

Dưới gĩc độ số chương trình được sử dụng cùng lúc

- Hệ điều hành đơn nhiệm
- Hệ điều hành đa nhiệm

Dưới gĩc độ người dùng (truy xuất tài nguyên cùng lúc)

- **Một người dùng**
- **Nhiều người dùng**
  - **Mạng ngang hàng**
  - **Mạng cĩ máy chủ: LAN, WAN, ...**

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

Dưới góc độ hình thức xử lý

- –Hệ thống xử lý theo lô
- –Hệ thống chia sẻ
- –Hệ thống song song
- –Hệ thống phân tán
- –Hệ thống xử lý thời gian thực

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

HỆ THỐNG XỬ LÝ ĐƠN CHƯƠNG

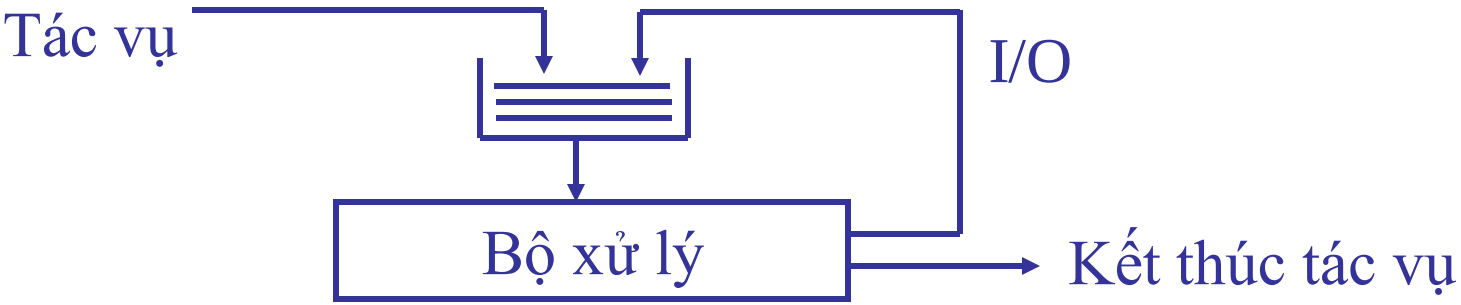
❖ **Đơn chương**

- Tác vụ được thi hành tuần tự.
- Bộ giám sát thường trực,
- CPU và các thao tác nhập xuất,
- Xử lý offline,
- Đồng bộ hóa các thao tác bên ngoài - Spooling (Simultaneous Peripheral Operation On Line)

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

HỆ THỐNG XỬ LÝ ĐA CHƯƠNG

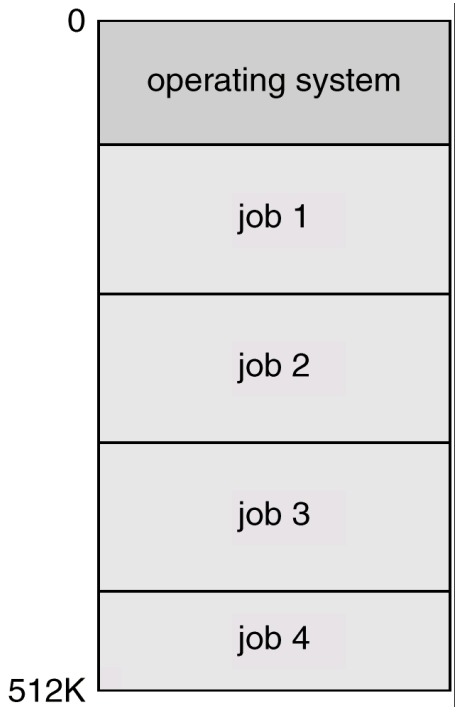
- Nhiều tác vụ sẵn sàng thi hành cùng một thời điểm.
- Khi một tác vụ thực hiện I/O, bắt đầu tác vụ khác.
- Bộ xử lý và thiết bị thi hành toàn thời gian.



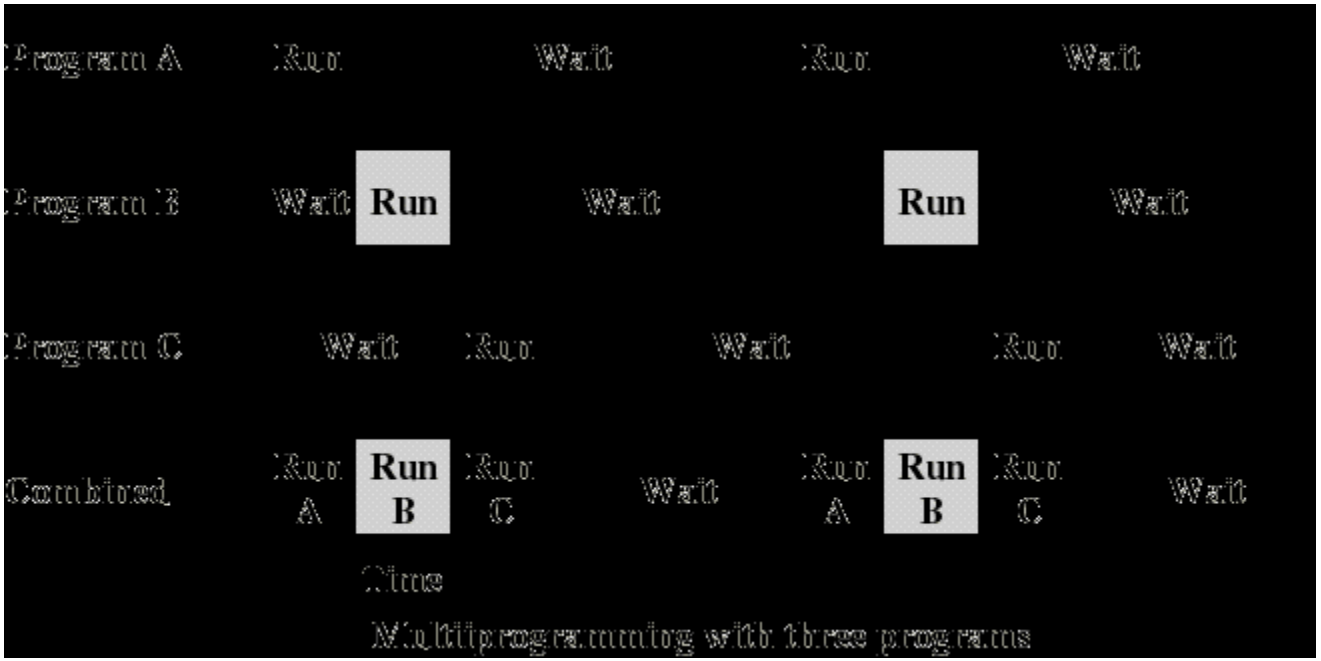
1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

- *Multiprogrammed systems*
  - Nhiều công việc được nạp đồng thời vào bộ nhớ chính
  - Khi một tiến trình thực hiện I/O, một tiến trình khác được thực thi
  - Tận dụng được thời gian rảnh, tăng *hiệu suất sử dụng* CPU (CPU utilization)

  - Yêu cầu đối với hệ điều hành
    - ✓ Định thời công việc (job scheduling): chọn job trong job pool trên đĩa và nạp nó vào bộ nhớ để thực thi.
    - ✓ Quản lý bộ nhớ (memory management)
    - ✓ Định thời CPU (CPU scheduling)
    - ✓ Cấp phát tài nguyên (đĩa, máy in,…)
    - ✓ Bảo vệ



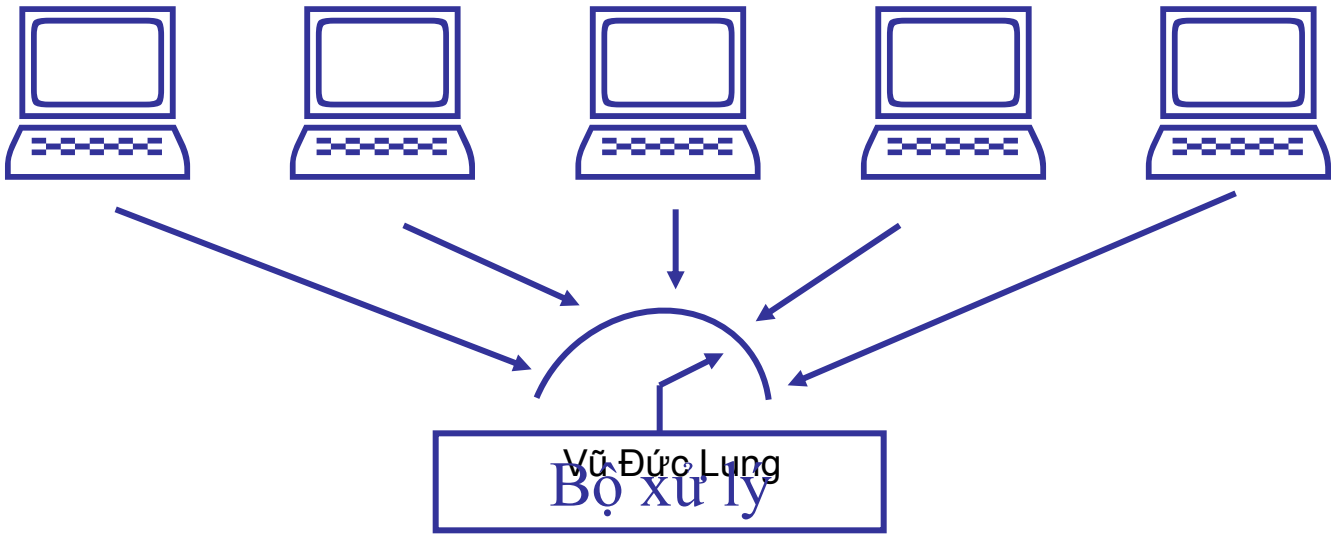
1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH



1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

HỆ THỐNG CHIA XẺ THỜI GIAN

- Hệ thống đa nhiệm (multitasking).
- Lập lịch CPU.
- Thời gian chuyển đổi giữa các tác vụ rất ngắn.



HỆ THỐNG CHIA XẺ THỜI GIAN

- *Time-sharing systems*
  - Multiprogrammed systems không cung cấp khả năng tương tác hiệu quả với users
  - CPU luân phiên thực thi giữa các công việc
    - Mỗi công việc được chia một phần nhỏ thời gian CPU (*time slice*, *quantum time*)
    - Cung cấp tương tác giữa user và hệ thống với *thời gian đáp ứng* (response time) nhỏ (1 s)
  - Một công việc chỉ được chiếm CPU khi nó nằm trong bộ nhớ chính.
  - Khi cần thiết, một công việc nào đó có thể được chuyển từ bộ nhớ chính ra thiết bị lưu trữ (swapping), nhường bộ nhớ chính cho công việc khác.

HỆ THỐNG CHIA XẺ THỜI GIAN

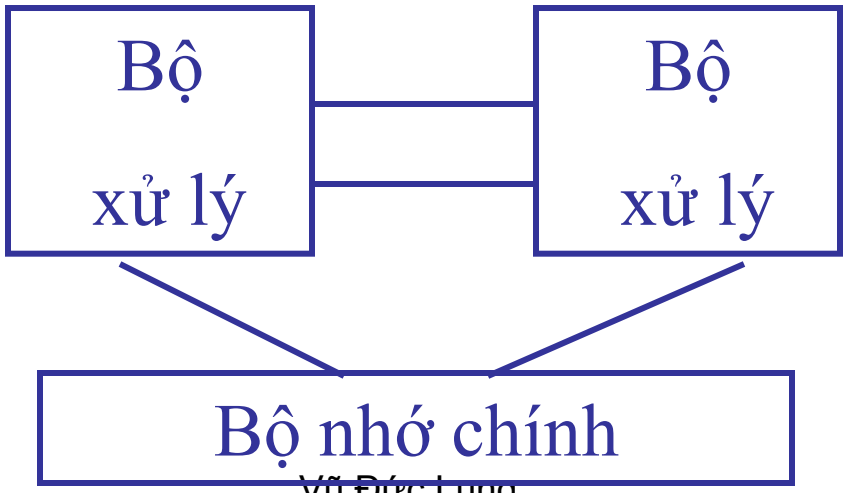
- Yêu cầu đối với OS trong hệ thống time-sharing
  - Định thời công việc (job scheduling)
  - Quản lý bộ nhớ (memory management)
    - Virtual memory
  - Quản lý các quá trình (process management)
    - Định thời CPU
    - Đồng bộ các quá trình (synchronization)
    - Giao tiếp giữa các quá trình (process communication)
    - Tránh deadlock
  - Quản lý hệ thống file, hệ thống lưu trữ
  - Cấp phát hợp lý các tài nguyên
  - Bảo vệ (protection)

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

HỆ THỐNG ĐA XỬ LÝ

❖Hai hoặc nhiều bộ xử lý cùng chia sẻ một bộ nhớ.

❖ Master/Slave : một bộ xử lý chính kiểm soát một số bộ xử lý I/O



HỆ THỐNG ĐA XỬ LÝ

- *Hệ thống song song* (parallel, multiprocessor, hay tightly-coupled system)
  - Nhiều CPU
  - Chia sẻ computer bus, clock
  - Ưu điểm
    - *Năng xuất hệ thống (System throughput)*: càng nhiều processor thì càng nhanh xong công việc
    - Multiprocessor system ít tốn kém hơn multiple single-processor system: vì có thể dùng chung tài nguyên (đĩa,…)
    - *Độ tin cậy*: khi một processor hỏng thì công việc của nó được chia sẻ giữa các processor còn lại

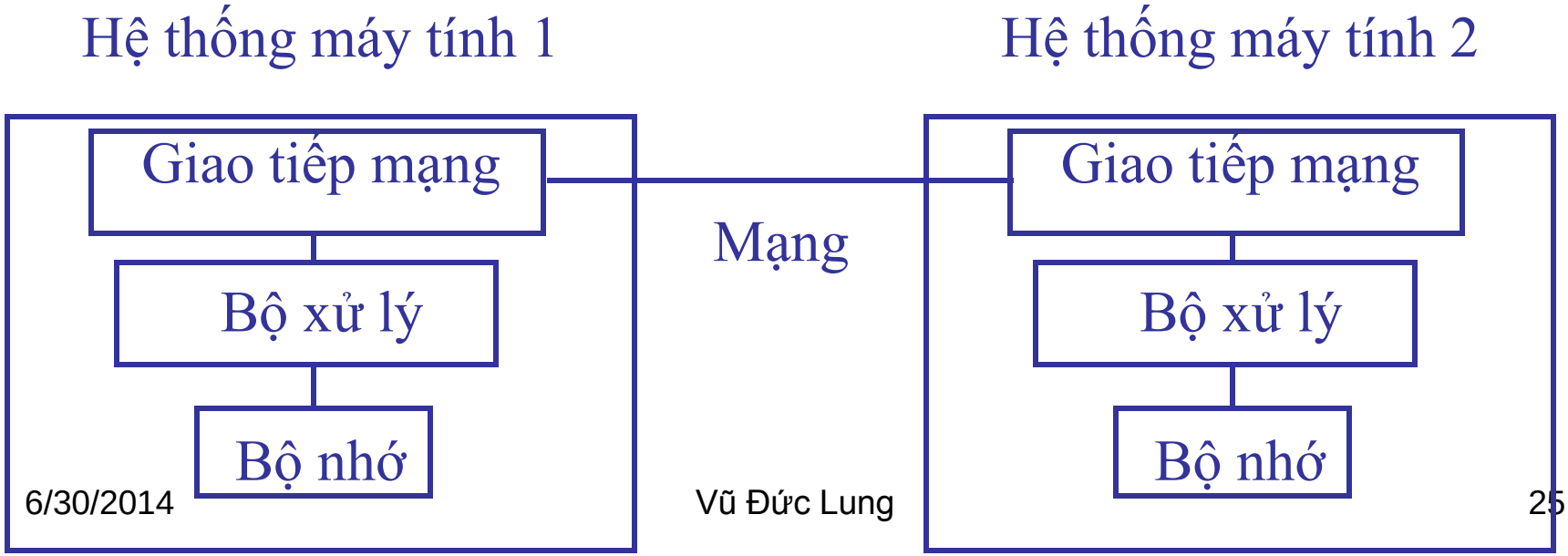
HỆ THỐNG ĐA XỬ LÝ

- Phân loại hệ thống song song
  - *Đa xử lý đối xứng* (symmetric multiprocessor - SMP)
    - Mỗi processor vận hành một identical copy của hệ điều hành
    - Các copy giao tiếp với nhau khi cần
    - (Windows NT, Solaris 5.0, Digital UNIX, OS/2, Linux)
  - *Đa xử lý bất đối xứng* (asymmetric multiprocessor)
    - Mỗi processor thực thi một công việc khác nhau
    - Master processor định thời và phân công việc cho các slave processors
    - (SunOS 4.0)

1.2. PHÂN LOẠI HỆ ĐIỀU HÀNH

HỆ THỐNG PHÂN TÁN

- **Nhiều máy tính liên kết với nhau bằng đường truyền thông đặc biệt.**
- **Tương tự hệ thống đa xử lý nhưng không chia xẻ bộ nhớ.**



HỆ THỐNG PHÂN TÁN

- *Hệ thống phân tán* (distributed system, loosely-coupled system)
  - Mỗi processor có bộ nhớ riêng, các processor giao tiếp qua các kênh nối như mạng, bus tốc độ cao
  - Người dùng chỉ thấy một hệ thống đơn nhất
  - Ưu điểm
    - Chia sẻ tài nguyên (resource sharing)
    - Chia sẻ sức mạnh tính toán (computational sharing)
    - Độ tin cậy cao (high reliability)
    - *Độ sẵn sàng* cao (high availability): các dịch vụ của hệ thống được cung cấp liên tục cho dù một thành phần hardware trở nên hỏng

HỆ THỐNG PHÂN TÁN

- Hệ thống phân tán (tt)

  Các mô hình hệ thống phân tán

  - *Client-server*
    - Server: cung cấp dịch vụ
    - Client: có thể sử dụng dịch vụ của server
  - *Peer-to-peer* (P2P)
    - Các *peer* (máy tính trong hệ thống) đều ngang hàng nhau
    - Không có cơ sở dữ liệu tập trung
    - Các peer là tự trị
    - Vd: Gnutella

Hệ thống thời gian thực (real-time system)

- *Hệ thống thời gian thực* (real-time system)
  - Sử dụng trong các thiết bị chuyên dụng như điều khiển các thử nghiệm khoa học, điều khiển trong y khoa, dây chuyền công nghiệp, thiết bị gia dụng, quân sự
  - Ràng buộc về thời gian: hard và soft real-time

  Phân loại
  - *Hard real-time*
    - Hạn chế (hoặc không có) bộ nhớ phụ, tất cả dữ liệu nằm trong bộ nhớ chính (RAM hoặc ROM)
    - Yêu cầu về thời gian đáp ứng/xử lý rất nghiêm ngặt, thường sử dụng trong điều khiển công nghiệp, robotics,…
  - *Soft real-time*
    - Thường được dùng trong lĩnh vực multimedia, virtual reality với yêu cầu mềm dẻo hơn về thời gian đáp ứng

Thiết bị cầm tay (handheld system)

- *Thiết bị cầm tay* (handheld system)
  - Personal digital assistant (PDA): Palm, Pocket-PC
  - Điện thoại di động (cellular phones)
  - Đặc trưng
    - Bộ nhớ nhỏ (512 KB – 128 MB)
    - Tốc độ processor thấp (để ít tốn pin)
    - Màn hình hiển thị có kích thước nhỏ và độ phân giải thấp.
    - Có thể dùng các công nghệ kết nối như IrDA, Bluetooth, wireless

1.3. LỊCH SỬ PHÁT TRIỂN CỦA HỆ ĐIỀU HÀNH

❖Thế hệ 1 (1945 - 1955)

- **Thiết kế, xây dựng, lập trình, thao tác: đều do 1 nhóm người**
- **Lưu trên phiếu đục lỗ**

❖ Thế hệ 2 (1955 - 1965)

- **Xuất hiện sự phân công công việc**
- **Hệ thống sử lý theo lô ra đời, lưu trên băng từ**
- **Hoạt động dưới sự điều khiển đặc biệt của 1 chương trình**

❖ Thế hệ 3 (1965 - 1980)

-Ra đời hệ điều hành, khái niệm đa chương

- HĐH chia sẻ thời gian như CTSS của MIT
- MULTICS, UNIX

1.3. LỊCH SỬ PHÁT TRIỂN CỦA HỆ ĐIỀU HÀNH

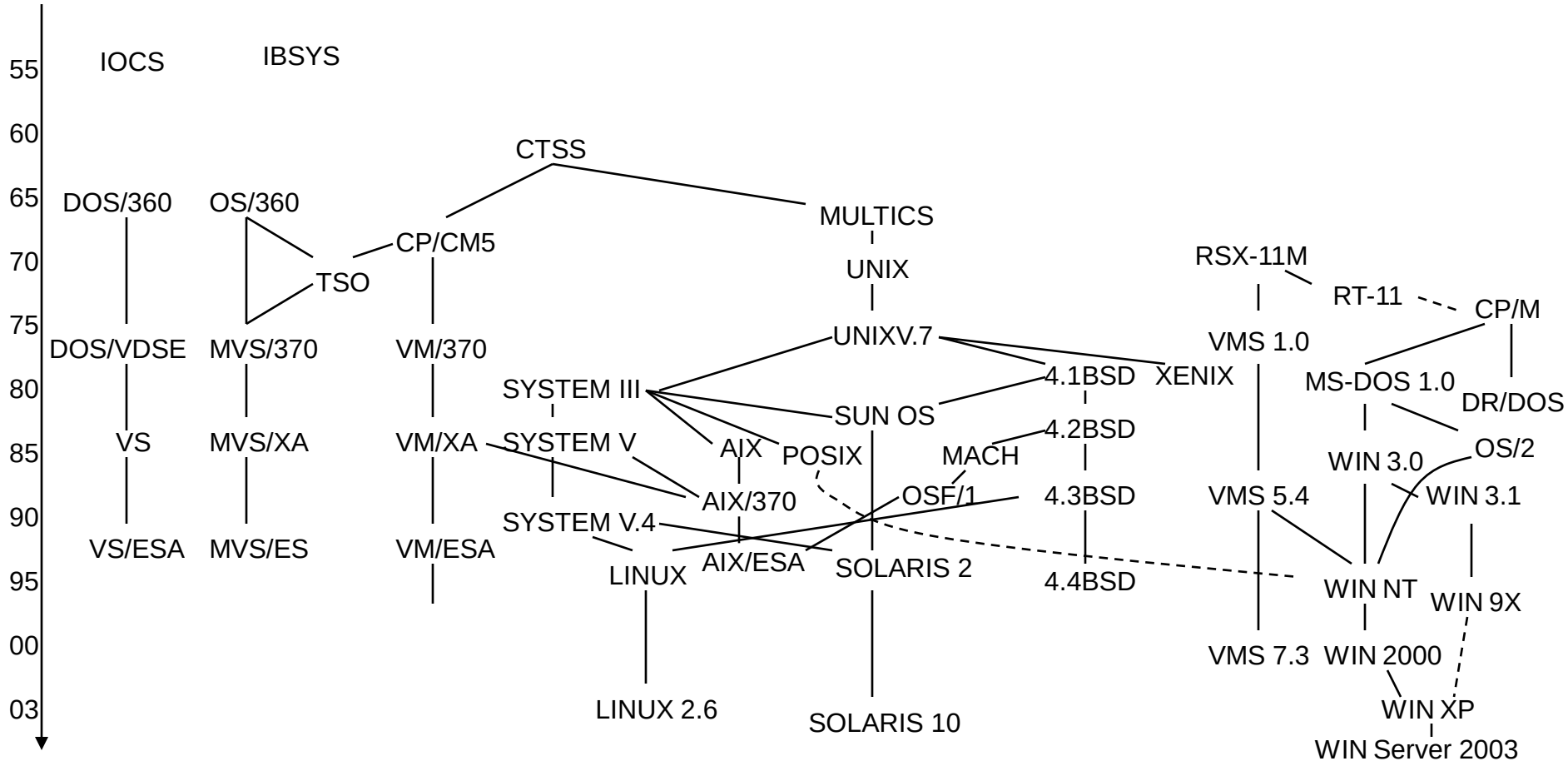
❖Thế hệ 4 (1980 -)

-Ra đời máy tính cá nhân, IBM PC

- HĐH MS-DOS, MacOS (Apple Macintosh), MS Windows, OS/1

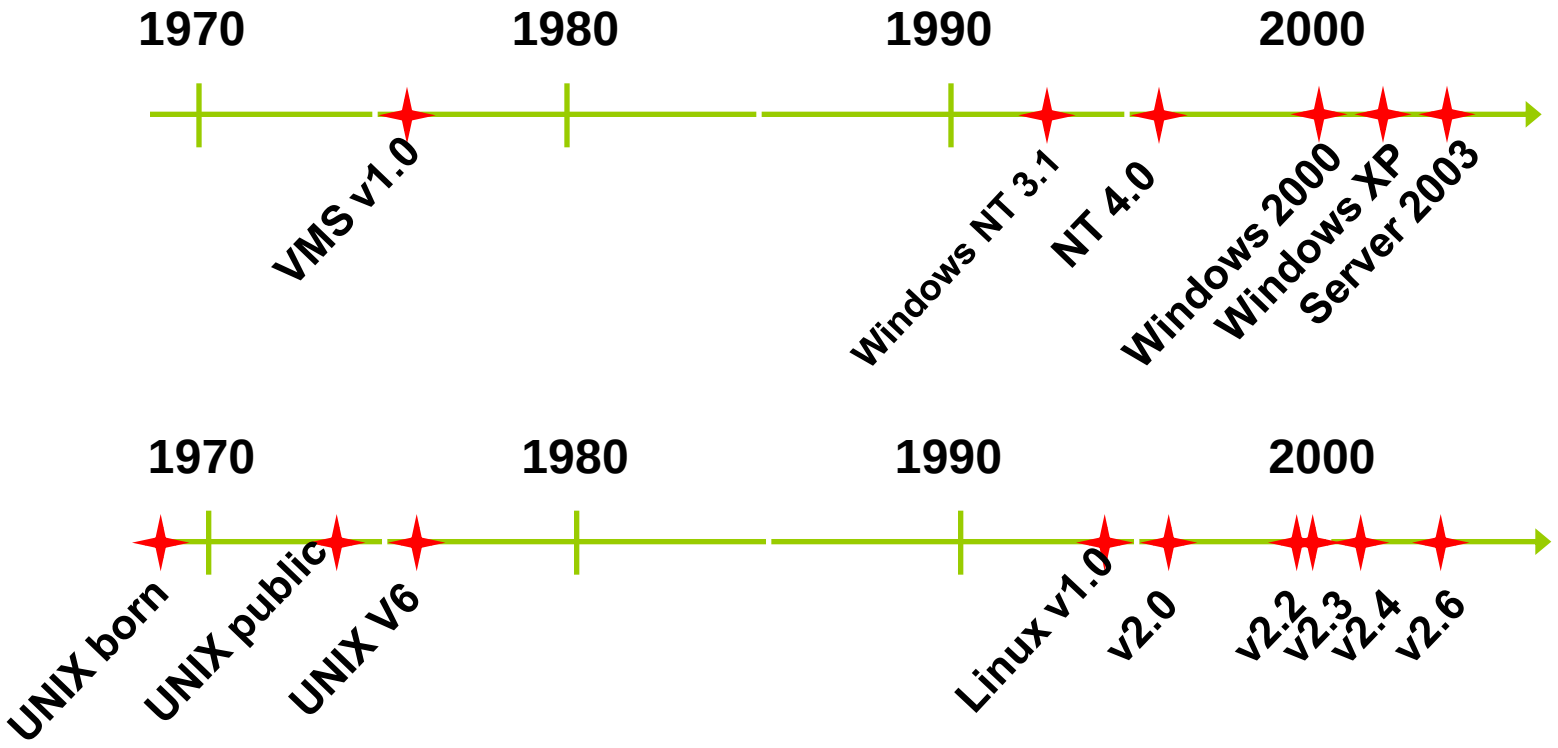
- Linux, QNX, HĐH mạng,…

Operating Systems Evolution



Windows And Linux Evolution

- Windows and Linux kernels are based on foundations developed in the mid-1970s



(see http://www.levenez.com for diagrams showing history of Windows & Unix)

Chương II: Cấu Trúc Hệ Điều Hành

- Các thành phần của hệ điều hành
- Các dịch vụ hệ điều hành cung cấp
- Lời gọi hệ thống (System call)
- Các chương trình hệ thống (system programs)
- Cấu trúc hệ thống
- Máy ảo (virtual machine)

2.1. Các thành phần của hệ điều hành

•2.1.1. Quản lý quá trình (process management)

- Quá trình (hay tiến trình – **process**) là gì?
- Quá trình khác chương trình ở điểm gì?
- Một quá trình cần các tài nguyên của hệ thống như CPU, bộ nhớ, file, thiết bị I/O,… để hoàn thành công việc.
- Các nhiệm vụ của thành phần
  - Tạo và hủy quá trình
  - Tạm dừng/thực thi tiếp (suspend/resume) quá trình
  - Cung cấp các cơ chế
    - đồng bộ hoạt động các quá trình (synchronization)
    - giao tiếp giữa các quá trình (interprocess communication)
    - khống chế tắc nghẽn (deadlock)

2.1. Các thành phần của hệ điều hành

•2.1.2. Quản lý bộ nhớ chính

- *Bộ nhớ chính* là trung tâm của các thao tác, xử lý
- Để nâng caó hiệu suất sử dụng CPU, hệ điều hành cần quản lý bộ nhớ thích hợp
- Các nhiệm vụ của thành phần
  - Theo dõi, quản lý các vùng nhớ trống và đã cấp phát
  - Quyết định sẽ nạp chương trình nào khi có vùng nhớ trống
  - Cấp phát và thu hồi các vùng nhớ khi cần thiết

2.1. Các thành phần của hệ điều hành

- 2.1.3. Quản lý file (file management)
  - Hệ thống file (file system)
    - *File*
    - *Thư mục*
  - Các dịch vụ mà thành phần cung cấp
    - Tạo và xoá file/thư mục.
    - Các thao tác xử lý file/thư mục (mkdir, rename, copy, move, new,…)
    - "Ánh xạ" file/thư mục vào thiết bị lưu trữ thứ cấp tương ứng
    - Sao lưu và phục hồi dữ liệu

2.1. Các thành phần của hệ điều hành

•2.1.4. Quản lý hệ thống I/O (I/O system management)

- Che dấu sự khác biệt của các thiết bị I/O trước người dùng
- Có chức năng
  - Cơ chế: *buffering*, *caching*, *spooling*
  - Cung cấp giao diện chung đến các trình điều khiển thiết bị (*device-driver interface*)
  - Bộ điều khiển các thiết bị (*device driver*) phần cứng.

2.1. Các thành phần của hệ điều hành

•2.1.5. Quản lý hệ thống lưu trữ thứ cấp (secondary storage management)

- Bộ nhớ chính: kích thước nhỏ, là môi trường chứa tin không bền vững => cần hệ thống lưu trữ thứ cấp để lưu trữ bền vững các dữ liệu, chương trình
- Phương tiện lưu trữ thông dụng là đĩa từ, đĩa quang
- Nhiệm vụ của hệ điều hành trong quản lý đĩa
  - Quản lý không gian trống trên đĩa(*free space management*)
  - Cấp phát không gian lưu trữ (*storage allocation*)
  - Định thời hoạt động cho đĩa (*disk scheduling*)

➢ *Sử dụng thường xuyên => ảnh hưởng lớn đến tốc độ của cả hệ thống => cần hiệu quả*

2.1. Các thành phần của hệ điều hành

•2.1.6. Hệ thống bảo vệ

Trong hệ thống cho phép nhiều user hay nhiều process diễn ra đồng thời:

- Kiểm soát quá trình người dùng đăng nhập/xuất và sử dụng hệ thống
- Kiểm soát việc truy cập các tài nguyên trong hệ thống
- Bảo đảm những *user/process* chỉ được phép sử dụng các tài nguyên dành cho nó
- Các nhiệm vụ của hệ thống bảo vệ
  - Cung cấp cơ chế kiểm soát đăng nhập/xuất *(login, log out)*
  - Phân định được sự truy cập tài nguyên *hợp pháp* và *bất hợp pháp (authorized/unauthorized)*
  - Phương tiện thi hành các chính sách (*enforcement of policies*)
    Chính sách: cần bảo vệ dữ liệu của ai đối với ai

2.1. Các thành phần của hệ điều hành

•2.1.7. *Hệ thống thông dịch lệnh*

- Là giao diện chủ yếu giữa người dùng và OS
  - Ví dụ: shell, mouse-based window-and-menu
- Khi user login
  - *command line interpreter* (shell) chạy, và chờ nhận lệnh từ người dùng, thực thi lệnh và trả kết quả về.
- Các lệnh ->bộ điều khiển lệnh ->hệ điều hành
- Các lệnh có quan hệ với các việc:
  - Tạo, hủy, và quản lý quá trình, hệ thống
  - Kiểm soát I/O
  - Quản lý bộ lưu trữ thứ cấp
  - Quản lý bộ nhớ chính
  - Truy cập hệ thống file và cơ chế bảo mật

2.2. Các dịch vụ hệ điều hành cung cấp

- Thực thi chương trình
- Thực hiện các thao tácï I/O theo yêu cầu của chương trình
- Các thao tác trên hệ thống file
  - Đọc/ghi hay tạo/xóa file
- Trao đổi thông tin giữa các quá trình qua hai cách:
  - Chia xẻ bộ nhớ (Shared memory)
  - Chuyển thông điệp (Message passing)
- Phát hiện lỗi
  - Trong CPU, bộ nhớ, trên thiết bị I/O (dữ liệu hư, hết giấy,…)
  - Do chương trình: chia cho 0, truy cập đến địa chỉ bộ nhớ không cho phép.

2.2. Các dịch vụ hệ điều hành cung cấp

Ngoài ra còn các dịch vụ giúp tăng hiệu suất của hệ thống:

- Cấp phát tài nguyên (resource allocation)
  - Tài nguyên: CPU, bộ nhớ chính, tape drives,…
  - OS có các routine tương ứng
- Kế toán (accounting)
  - Nhằm lưu vết user để tính phí hoặc đơn giản để thống kê.
- Bảo vệ (protection)
  - Hai quá trình khác nhau không được ảnh hưởng nhau
  - Kiểm soát được các truy xuất tài nguyên của hệ thống
- An ninh (security)
  - Chỉ các user được phép sử dụng hệ thống mới truy cập được tài nguyên của hệ thống (vd: thông qua username và password)

2.3. Lời gọi hệ thống (*System call)*

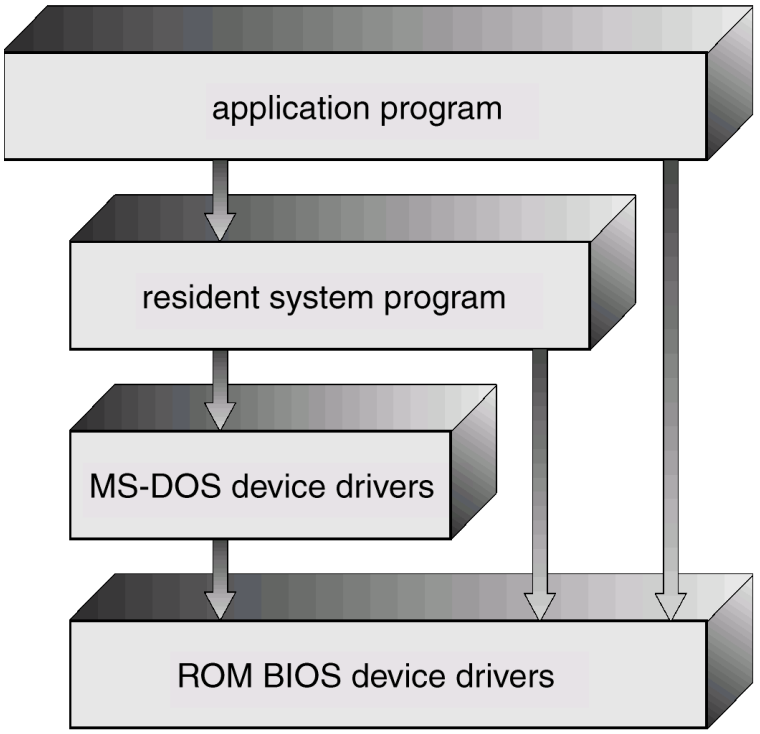
- *Dùng để giao tiếp giữa quá trình và hệ điều hành*
  - Cung cấp giao diện giữa quá trình và hệ điều hành
    - Vd: open, read, write file
  - Thông thường ở dạng thư viện nhị phân (binary libraries) hay giống như các lệnh hợp ngữ.
  - Trong các ngôn ngữ lập trình cấp cao, một số thư viện lập trình được xây dựng dựa trên các thư viện hệ thống (ví dụ Windows API, thư viện GNU C/C++ như glibc, glibc++,…)
  - Ba phương pháp truyền tham số khi sử dụng system call
    - Qua thanh ghi
    - Qua một vùng nhớ, địa chỉ của vùng nhớ được gửi đến hệ điều hành qua thanh ghi
    - Qua stack

2.4. Các chương trình hệ thống

- *Chương trình hệ thống* (*system program*, phân biệt với application program) gồm
  - Quản lý hệ thống file: như *create*, *delete*, *rename*, *list*
  - Thông tin trạng thái: như *date*, *time*, dung lượng bộ nhớ trống
  - Soạn thảo file: như *file editor*
  - Hỗ trợ ngôn ngữ lập trình: như *compiler*, *assembler*, *interpreter*
  - Nạp, thực thi, giúp tìm lỗi chương trình: như *loader*, *debugger*
  - Giao tiếp: như *email*, *talk*, *web browser*
  - …
- Người dùng chủ yếu làm việc thông qua các system program (không làm việc "trực tiếp" với các system call)

2.5. Cấu trúc hệ thống

- Cấu trúc đơn giản (monolithic)
  - MS-DOS: khi thiết kế, do giới hạn về dung lượng bộ nhớ nên không *phân chia thành các module* (modularization) và chưa phân chia rõ chức năng giữa các phần của hệ thống.



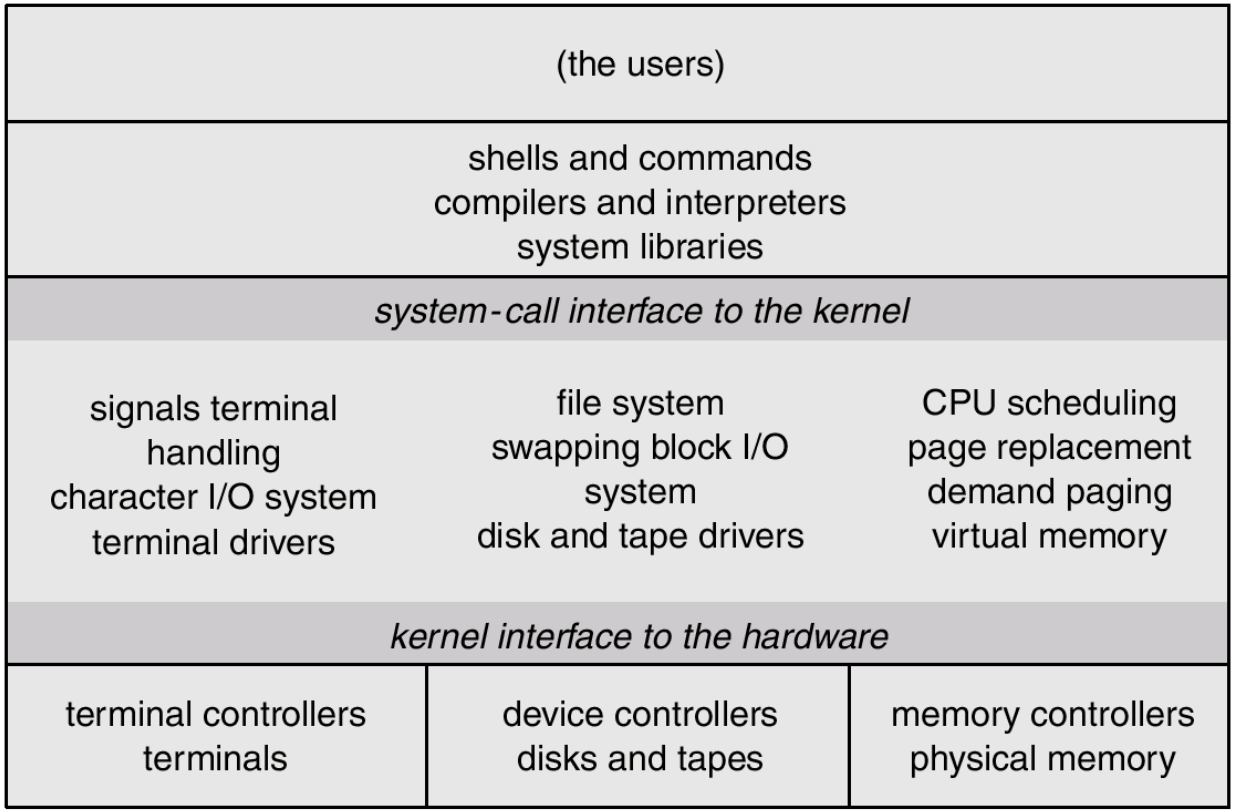
Cấu trúc phân tầng của MS-DOS

2.5. Cấu trúc hệ thống

➢ Cấu trúc đơn giản (monolithic)

UNIX: gồm hai phần có thể tách rời nhau

- Nhân (cung cấp file system, CPU scheduling, memory management, và một số chức năng khác) và system program



2.5. Cấu trúc hệ thống

- **Cấu trúc phân tầng:** HĐH được chi thành nhiều *lớp* (layer).
  - Lớp dưới cùng: hardware
  - Lớp trên cùng là giao tiếp với user
  - Lớp trên chỉ phụ thuộc lớp dưới
  - Một lớp chỉ có thể gọi các hàm của lớp dưới và các hàm của nó được gọi bởi lớp trên
  - Mỗi lớp tương đương một đối tượng trừu tượng: cấu trúc dữ liệu + thao tác
  - Phân lớp có lợi ích gì? Gỡ rối (debugger, kiểm tra hệ thống, thay đổi chức năng)

2.5. Cấu trúc hệ thống

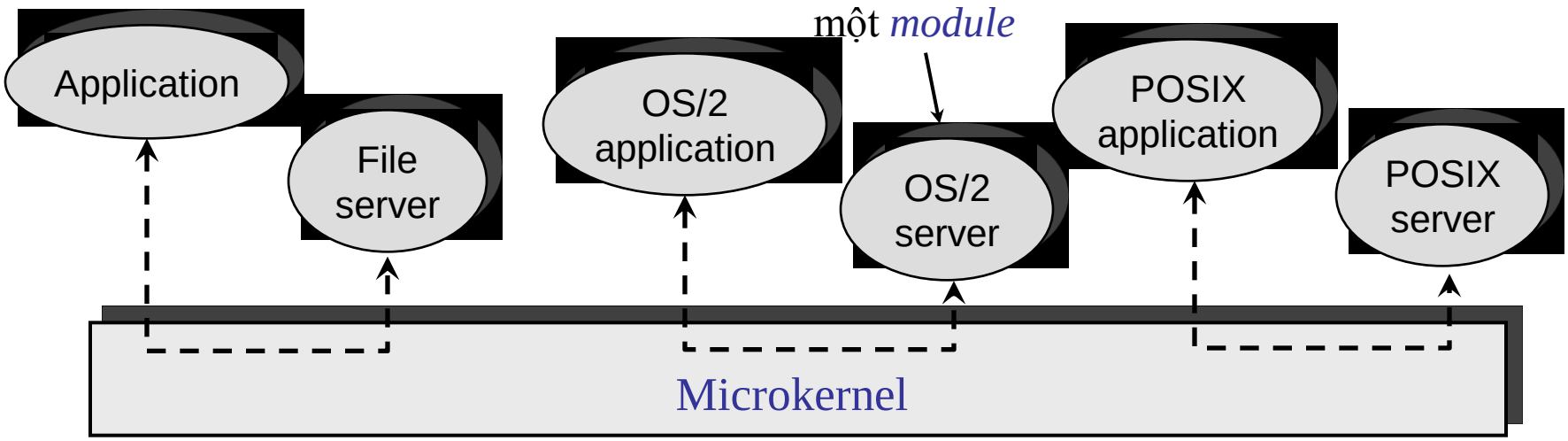
➢ Cấu trúc phân tầng:

- Lần đầu tiên được áp dụng cho HĐH THE (Technische Hogeschool Eindhoven)

Lớp 5	user programm
Lớp 4	Tạo buffer cho thiết bị I/O
Lớp 3	Device driver thao tác màn hình
Lớp 2	Quản lý bộ nhớ
Lớp 1	Lập lịch CPU
Lớp 0	Phần cứng

2.5. Cấu trúc hệ thống

- Vi nhân: phân chia module theo microkernel (CMU Mach OS, 1980)
- Chuyển một số chức năng của OS từ kernel space sang user space
- Thu gọn kernel => microkernel, microkernel chỉ bao gồm các chức năng tối thiểu như quản lý quá trình, bộ nhớ và cơ chế giao tiếp giữa các quá trình
- Giao tiếp giữa các module qua cơ chế truyền thông điệp



2.5. Cấu trúc hệ thống

➢ Vi nhân:

- Lợi ích: dễ mở rộng HĐH
- Một số HĐH hiện đại sử dụng vi nhân:

  + Tru64 UNIX (Digital UNIX trước đây): nhân Mach

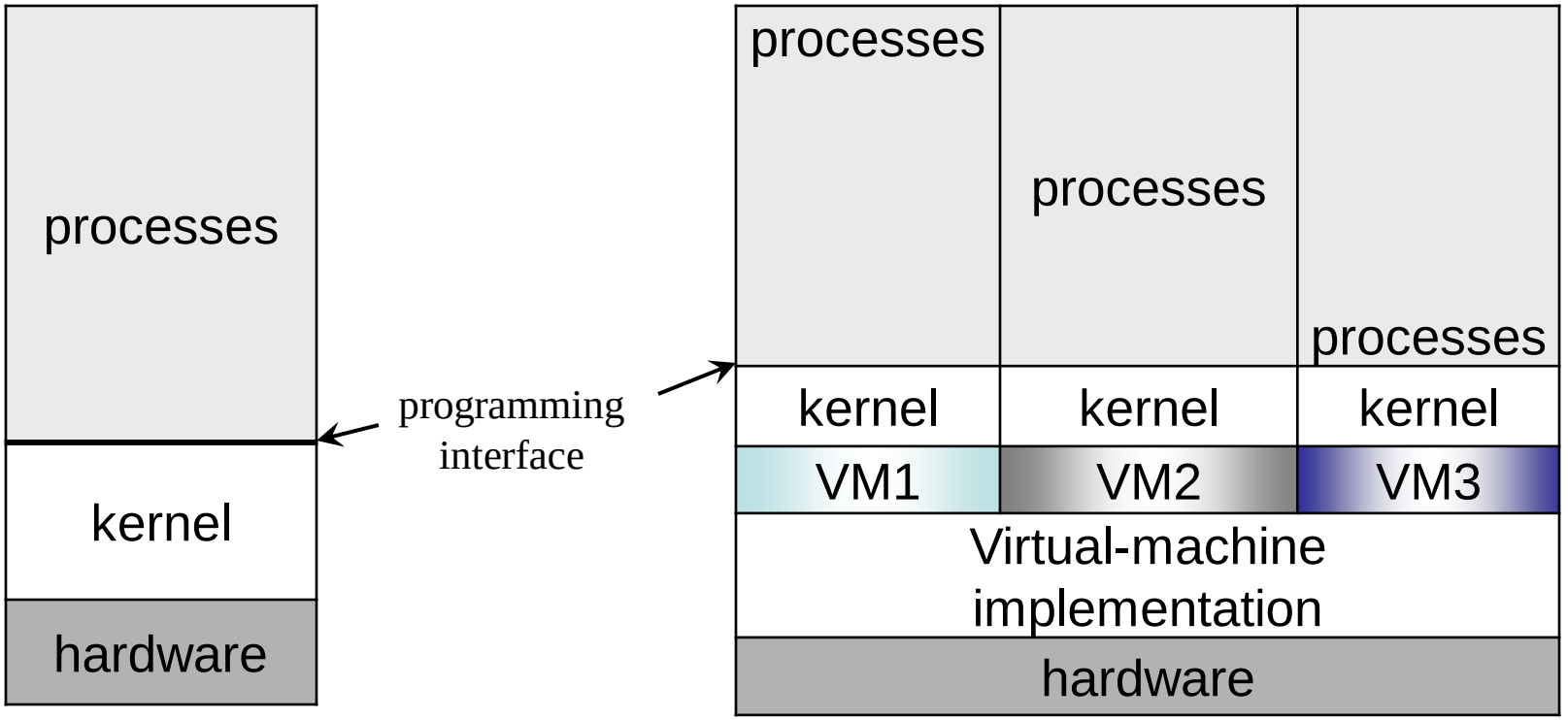
  + Apple MacOS Server : nhân Mach

  + QNX – vi nhân cung cấp: truyền thông điệp, định thời CPU, giao tiếp mạng cấp thấp và ngắt phần cứng

  + Windows NT: chạy các ứng dụng khác nhau win32, OS/2, POSIX (Portable OS for uniX)

2.6. Máy ảo

- Từ OS layer đến *máy ảo* (virtual machine)



Non-virtual machine system model

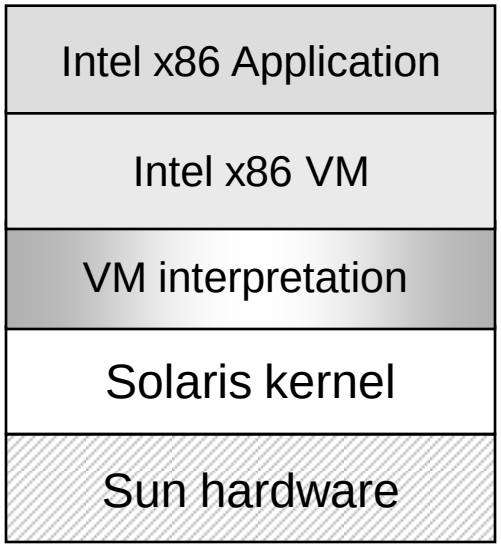
Virtual machine system model

2.6. Máy ảo

- Hiện thực ý niệm VM
  - Làm thế nào để thực thi một chương trình MS-DOS trên một hệ thống Sun với hệ điều hành Solaris ?

1. Tạo một *máy ảo Intel* bên trên hệ điều hành Solaris và hệ thống Sun

2. Các lệnh Intel (x86) được máy ảo Intel chuyển thành lệnh tương ứng của hệ thống Sun.



Chương III: Tiến trình (Process)

- Khái niệm cơ bản
- Trạng thái quá trình
- Khối điều khiển quá trình (Process control block)
- Định thời quá trình (Process Scheduling)
- Các tác vụ đối với quá trình
- Sự cộng tác giữa các quá trình
- Giao tiếp giữa các quá trình

3.1. Khái niệm cơ bản

- Cái gì gọi các hoạt động của CPU?
  - Hệ thống bó (Batch system): jobs
  - Time-shared systems: user programs, tasks
  - Các hoạt động là tương tự => gọi là process

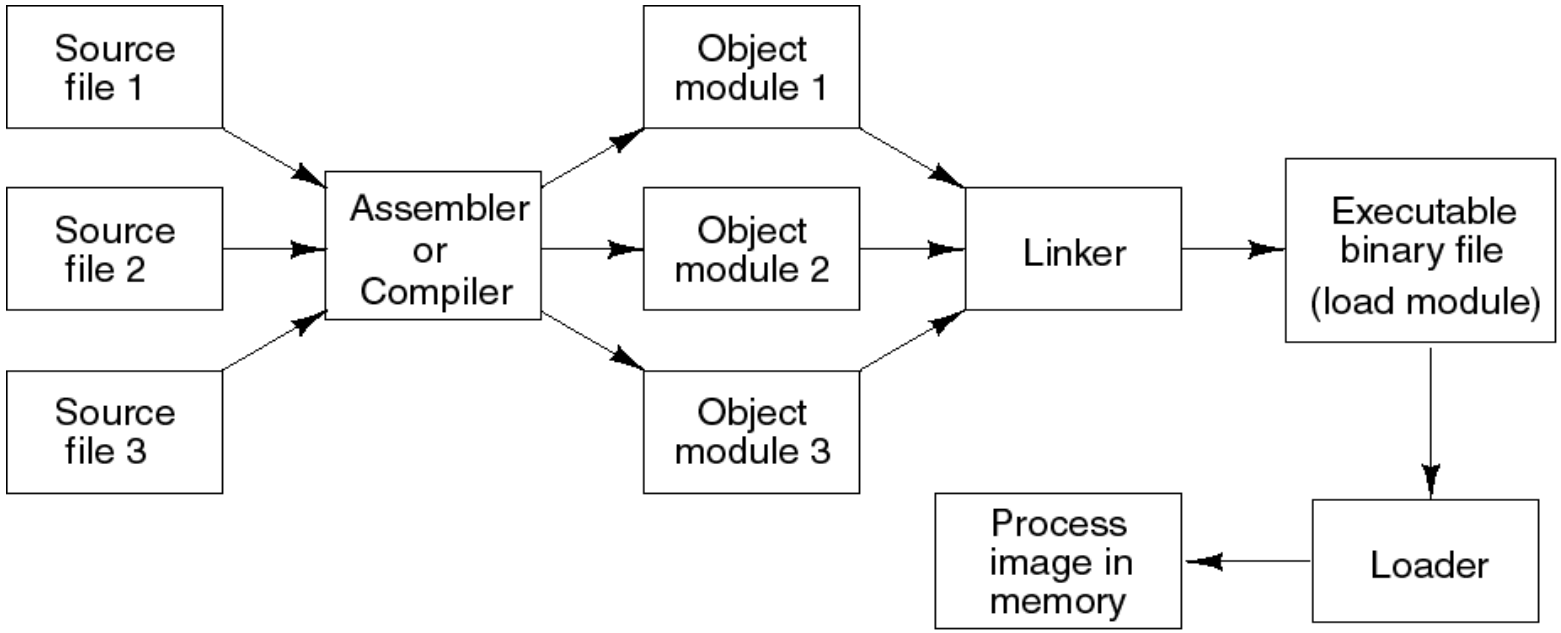
- *Quá trình* (process)
  - một chương trình đang thực thi

Một quá trình bao gồm

- *Text section* (program code), *data section* (chứa global variables)
- *program counter (PC), process status word (PSW), stack pointer (SP), memory management registers,...*

3.1. Khái niệm cơ bản

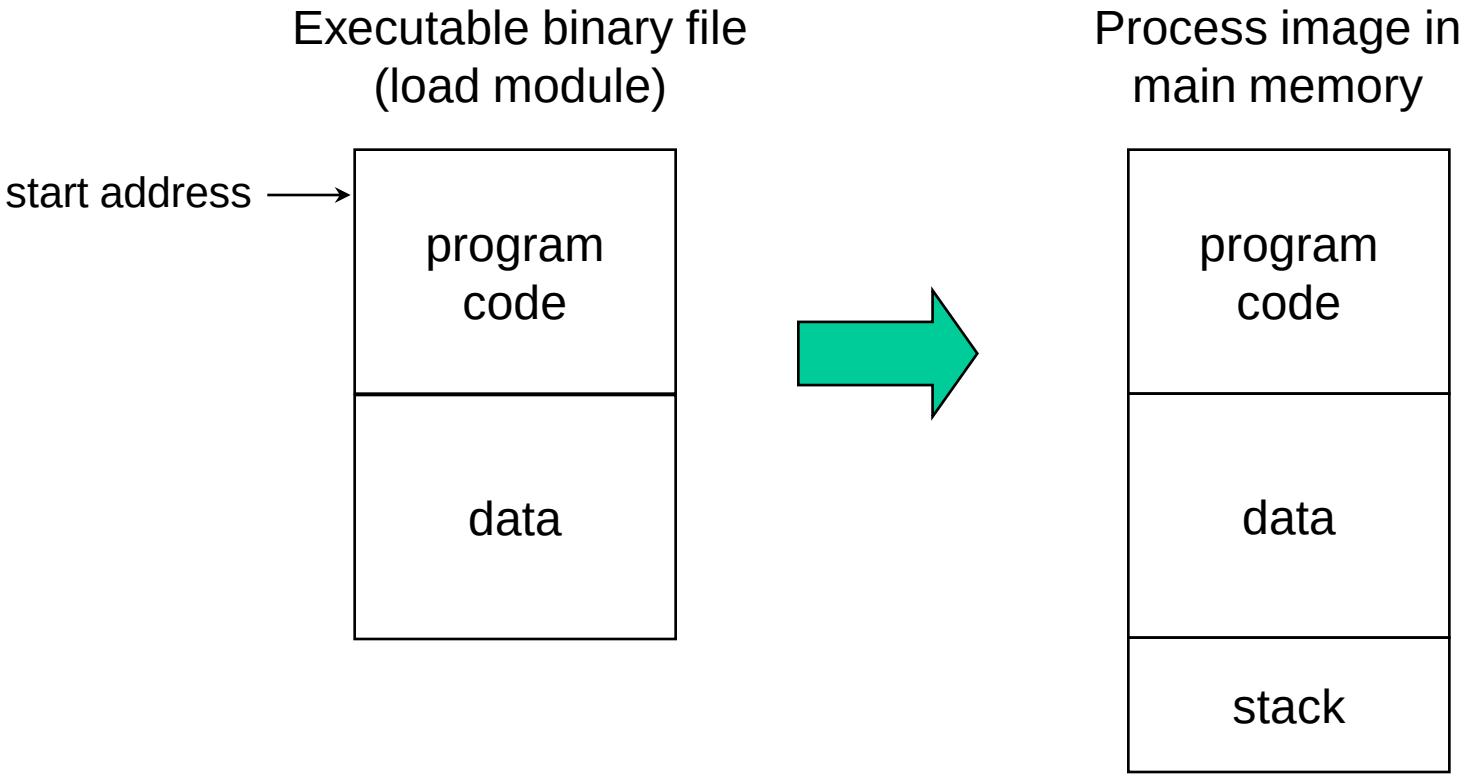
Các bước nạp chương trình vào bộ nhớ



3.1. Khái niệm cơ bản

chương trình => quá trình

- Dùng *load module* để biểu diễn chương trình thực thi được
- Layout luận lý của *process image*



3.1. Khái niệm cơ bản

Khởi tạo quá trình

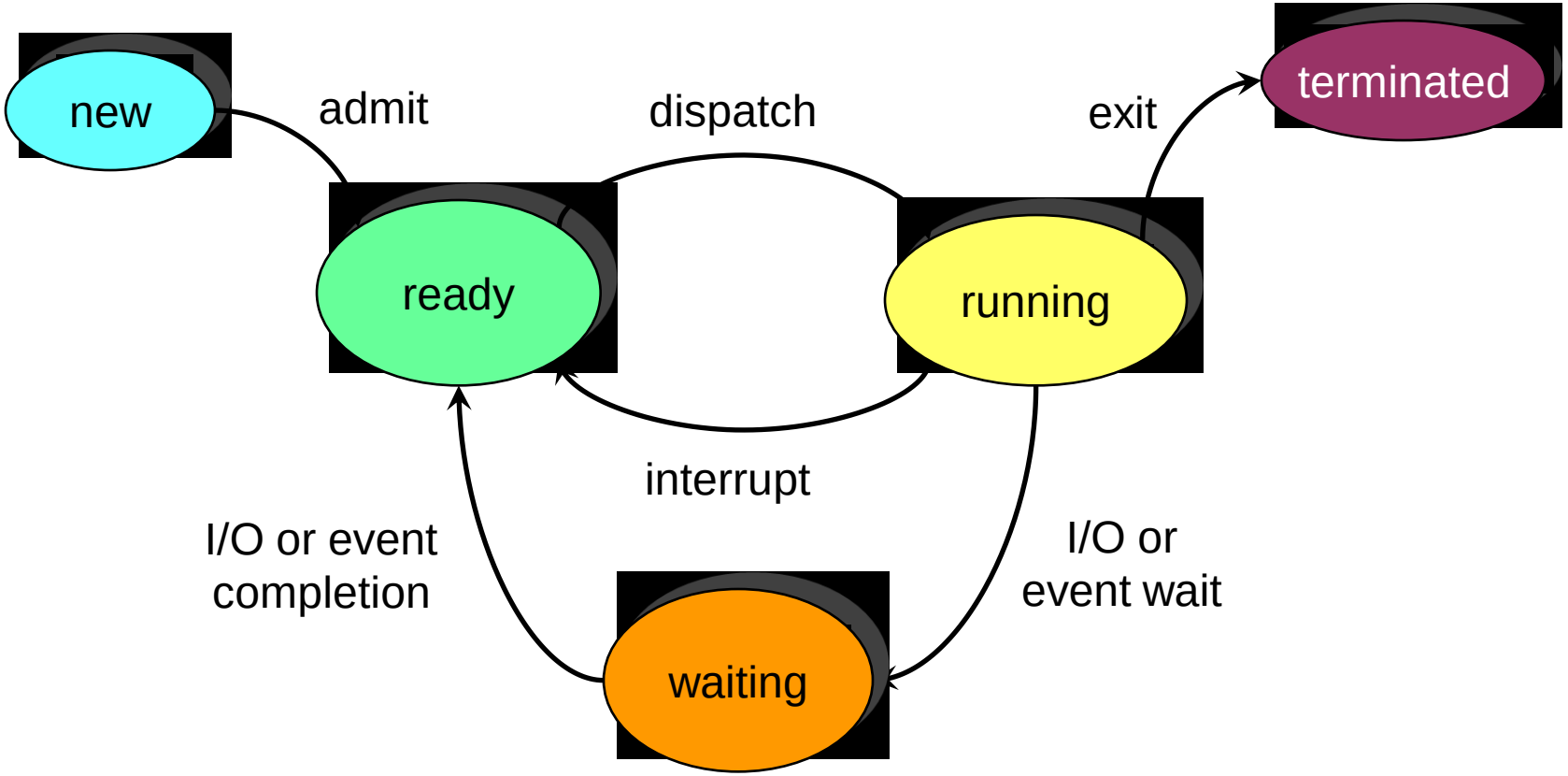
- Các bước hệ điều hành khởi tạo quá trình
  - Cấp phát một *định danh* duy nhất (process number hay process identifier, pid) cho quá trình
  - Cấp phát không gian nhớ để nạp quá trình
  - Khởi tạo khối dữ liệu *Process Control Block* (PCB) cho quá trình
    - PCB là nơi hệ điều hành lưu các thông tin về quá trình
  - Thiết lập các mối liên hệ cần thiết (vd: sắp PCB vào hàng đợi định thời,…)

3.2.Trạng thái quá trình

- Các *trạng thái của quá trình* (process states):
  - *new*: quá trình vừa được tạo
  - *ready*: quá trình đã có đủ tài nguyên, chỉ còn cần CPU
  - *running*: các lệnh của quá trình đang được thực thi
  - *waiting*: hay là *blocked*, quá trình đợi I/O hoàn tất, tín hiệu.
  - *terminated*: quá trình đã kết thúc.

3.2.Trạng thái quá trình

- Chuyển đổi giữa các trạng thái của quá trình



3.2.Trạng thái quá trình

Ví dụ

```
/* test.c */
int main(int argc, char** argv)
{
   printf("Hello world\n");
   exit(0);
}
```

Biên dịch chương trình trong Linux
`gcc test.c -o test`

Thực thi chương trình test
`./test`

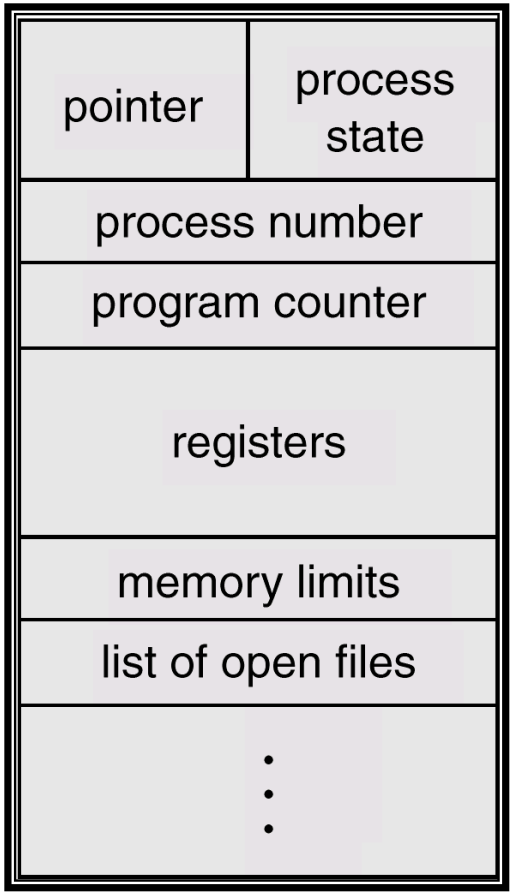
Trong hệ thống sẽ có một quá trình *test* được tạo ra, thực thi và kết thúc.

- Chuỗi trạng thái của quá trình test như sau (trường hợp tốt nhất):
  - new
  - ready
  - running
  - waiting (do chờ I/O khi gọi printf)
  - ready
  - running
  - terminated

3.3.Process control block

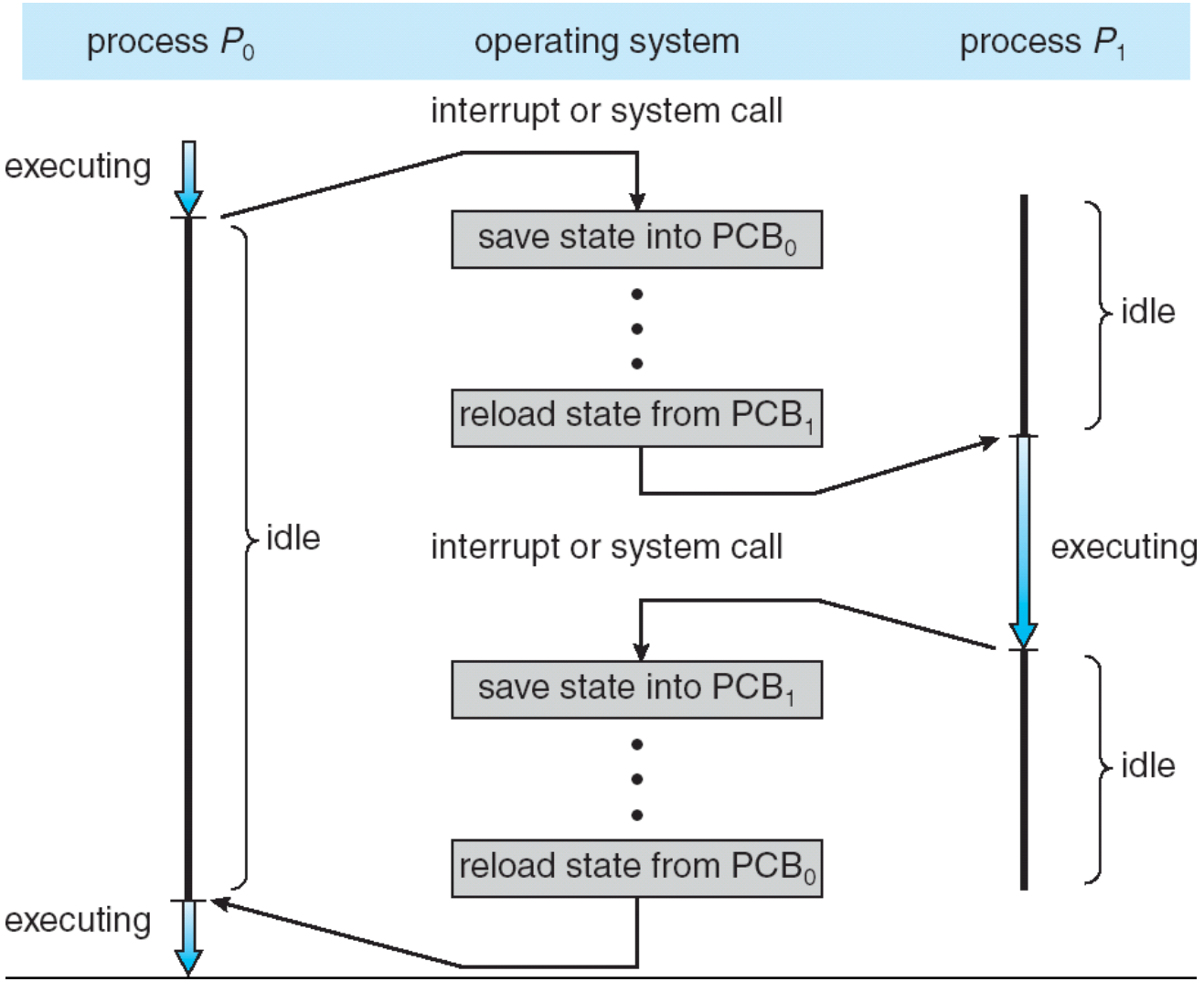
- Đã thấy là mỗi quá trình trong hệ thống đều được cấp phát một *Process Control Block* (PCB)
- PCB là một trong các cấu trúc dữ liệu quan trọng nhất của hệ điều hành và gồm:

  - Trạng thái quá trình: new, ready, running,…
  - Bộ đếm chương trình
  - Các thanh ghi
  - Thông tin lập thời biểu CPU: độ ưu tiên, …
  - Thông tin quản lý bộ nhớ
  - Thông tin tài khoản: lượng CPU, thời gian sử dụng,
  - Thông tin trạng thái I/O



3.3.Process control block

Lưu đồ chuyển CPU từ quá trình này đến quá trình khác

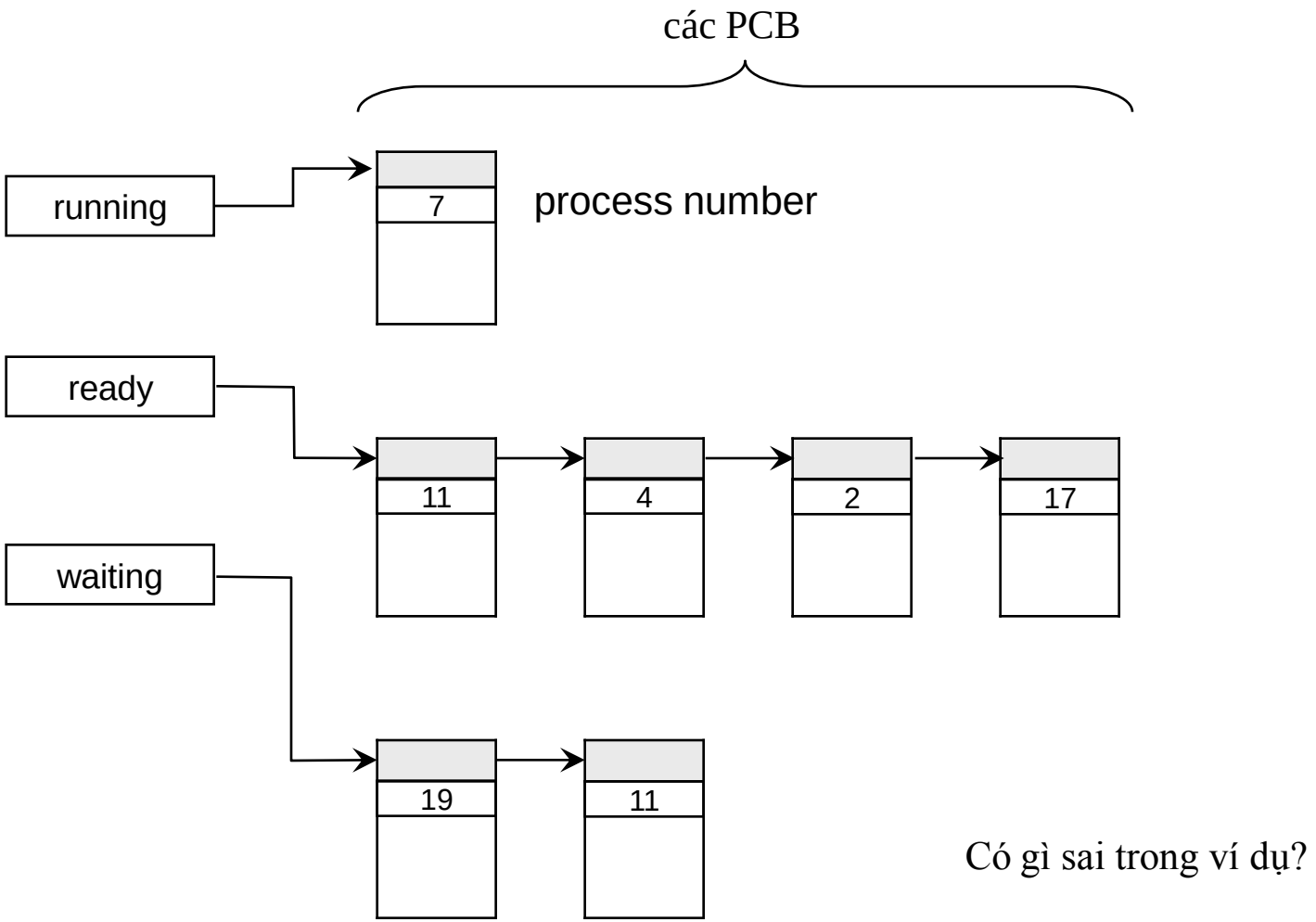


Yêu cầu đối với hệ điều hành về quản lý quá trình

- Hỗ trợ sự thực thi luân phiên giữa nhiều quá trình
  - Hiệu suất sử dụng CPU
  - Thời gian đáp ứng
- Phân phối tài nguyên hệ thống hợp lý
  - tránh deadlock, trì hoãn vô hạn định,…
- Cung cấp cơ chế giao tiếp và đồng bộ hoạt động các quá trình
- Cung cấp cơ chế hỗ trợ user tạo/kết thúc quá trình

Quản lý các quá trình: các hàng đợi

- Ví dụ



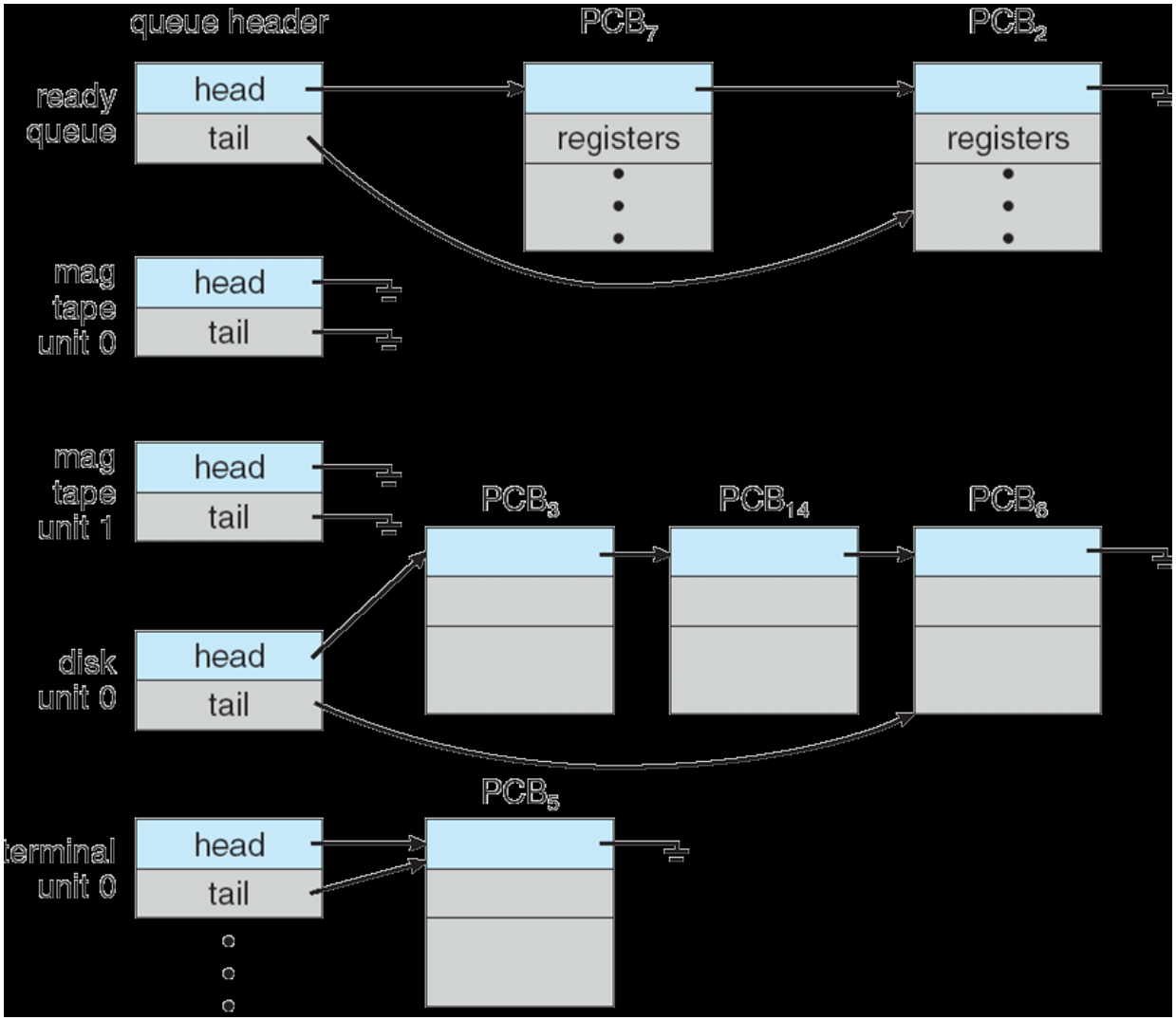
Có gì sai trong ví dụ?

3.4. Định thời quá trình (Process Scheduling)

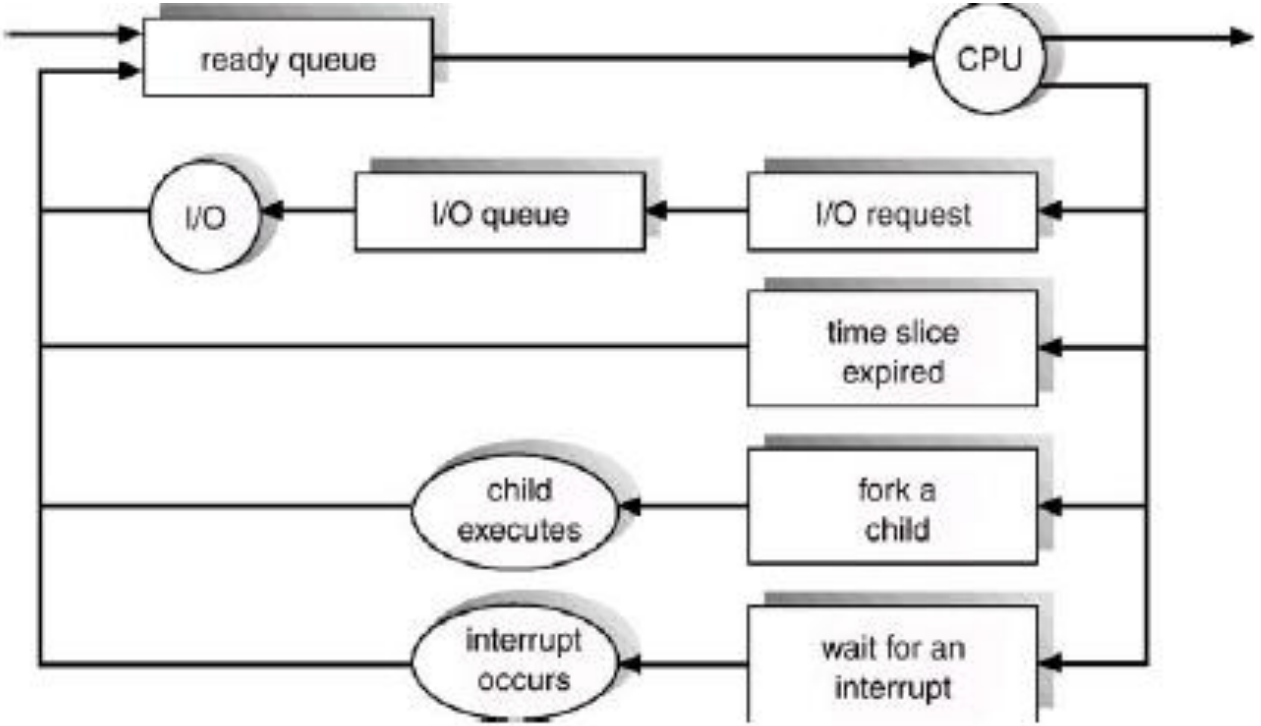
- Tại sao phải định thời?
  - Đa chương (Multiprogramming)
    - Có vài quá trình chạy tại các thời điểm
    - Mục tiêu: tận dụng tối đa CPU
  - Chia thời(Time-sharing)
    - Users tương tác với mỗi chương trình đang thực thi
    - Mục tiêu: tối thiểu thời gian đáp ứng
- Một số khái niệm cơ bản
  - Các *bộ định thời* (scheduler)
  - Các *hàng đợi định thời* (scheduling queue)

Các hàng đợi định thời (Scheduling queues)

- Hàng đợi công việc-Job queue
- Hàng đợi sẵn sàng-Ready queue
- Hàng đợi thiết bị-Device queues
- …



Các hàng đợi định thời (Scheduling queues)



Lưu đồ hàng đợi của định thời quá trình

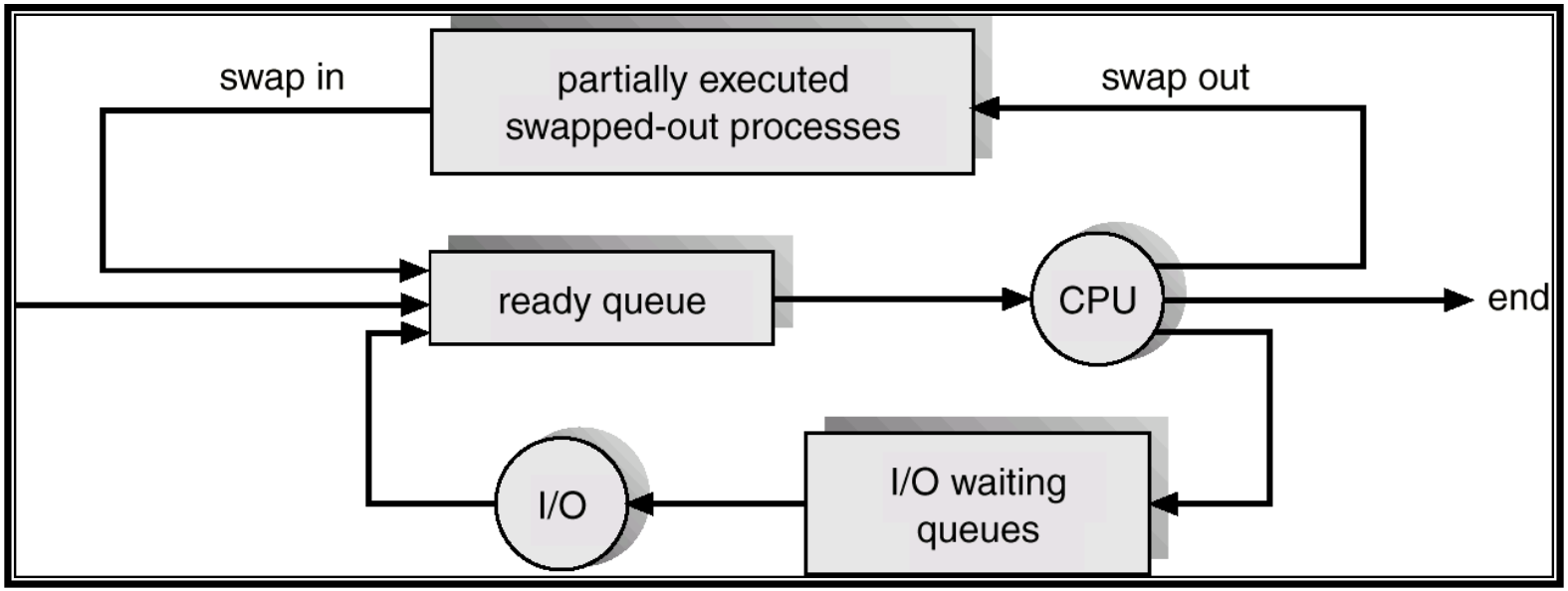
3.5. Bộ định thời (Scheduler)

- Bộ định thời công việc (Job scheduler) hay bộ định thời dài (long-term scheduler)
- Bộ định thời CPU hay bộ định thời ngắn
- Các quá trình có thể mô tả như:
  - Quá trình hướng I/O (I/O bound process)
  - Quá trình hướng CPU (CPU bound process)

  Thời gian thực hiện khác nhau => kết hợp hài hòa giữa chúng

Bộ định thời trung gian(medium-term scheduling)

- Đôi khi hệ điều hành (như time-sharing system) có thêm medium-term scheduling để điều chỉnh mức độ đa chương của hệ thống
- *Medium-term scheduler*
  - chuyển quá trình từ bộ nhớ sang đĩa (swap out)
  - chuyển quá trình từ đĩa vào bộ nhớ (swap in)

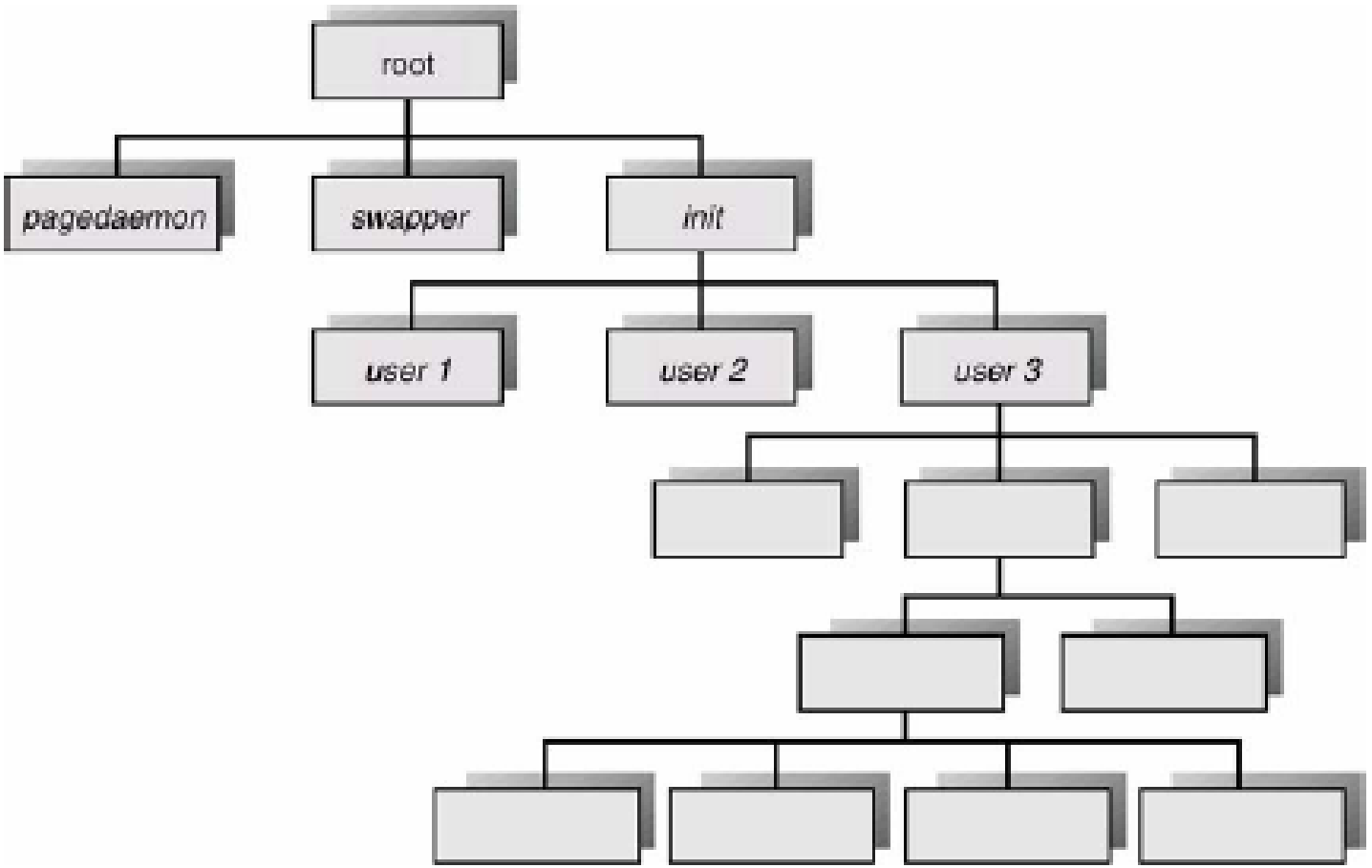


3.6. Các tác vụ đối với quá trình

- Tạo quá trình mới (process creation)
  - Một quá trình có thể tạo nhiều quá trình mới thông qua một lời gọi hệ thống ***create-process*** (vd: hàm fork trong Unix)
    - Ví dụ: (Unix) Khi user đăng nhập hệ thống, một command interpreter (shell) sẽ được tạo ra cho user
  - Quá trình được tạo là quá trình *con* của quá trình tạo (quá trình *cha*). Quan hệ cha-con định nghĩa một *cây quá trình.*

Cây quá trình trong Linux/Unix

- Ví dụ

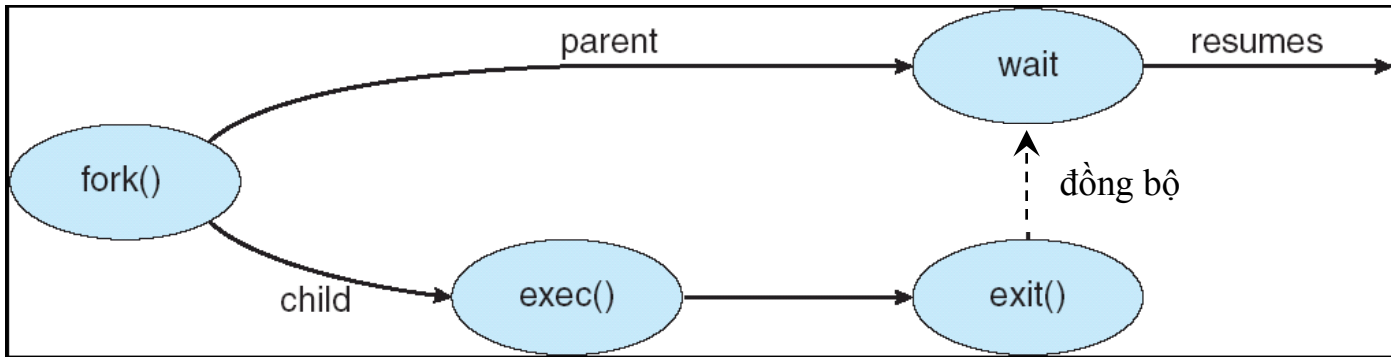


3.6.Các tác vụ đối với quá trình

- Tạo quá trình mới
  - Quá trình con nhận tài nguyên: từ HĐH hoặc từ P cha
  - Chia sẻ tài nguyên của quá trình cha
    - Quá trình cha và con chia sẻ mọi tài nguyên
    - Quá trình con chia sẻ một phần tài nguyên của cha
  - Trình tự thực thi
    - Quá trình cha và con thực thi đồng thời (concurrently)
    - Quá trình cha đợi đến khi các quá trình con kết thúc.

Về quan hệ cha/con

- Không gian địa chỉ (address space)
  - Không gian địa chỉ của quá trình con được nhân bản từ cha
  - Không gian địa chỉ của quá trình con được khởi tạo từ template.
- Ví dụ trong UNIX/Linux
  - System call fork() tạo một quá trình mới
  - System call exec() dùng sau fork() để nạp một chương trình mới vào không gian nhớ của quá trình mới



Ví dụ tạo process với fork()

```
#include <stdio.h>
#include <unistd.h>
int main (int argc, char *argv[]){
   int  pid;
   /* create a new process */
   pid = fork();

   if (pid > 0){
      printf("This is parent process");
      wait(NULL);
      exit(0);
   }
   else if (pid == 0)
   {
      printf("This is child process");
      execlp("/bin/ls", "ls", NULL);
      exit(0);
   }
   else {
        printf("Fork error\n");
        exit(-1);
   }
}
```

3.6.Các tác vụ đối với quá trình (tt)

- Tạo quá trình mới ✓
- Kết thúc quá trình
  - Quá trình tự kết thúc
    - Quá trình kết thúc khi thực thi lệnh cuối và gọi system routine exit
  - Quá trình kết thúc do quá trình khác (có đủ quyền, vd: quá trình cha của nó)
    - Gọi system routine abort với tham số là pid (process identifier) của quá trình cần được kết thúc
  - Hệ điều hành thu hồi tất cả các tài nguyên của quá trình kết thúc (vùng nhớ, I/O buffer,…)

3.7. Cộng tác giữa các quá trình

- Trong quá trình thực thi, các quá trình có thể *cộng tác* (cooperate) để hoàn thành công việc
- Các quá trình cộng tác để
  - Chia sẻ dữ liệu (information sharing)
  - Tăng tốc tính toán (computational speedup)
    - Nếu hệ thống có nhiều CPU, chia công việc tính toán thành nhiều công việc tính toán nhỏ chạy song song
  - Thực hiện một công việc chung
    - Xây dựng một phần mềm phức tạp bằng cách chia thành các module/process hợp tác nhau
- Sự cộng tác giữa các quá trình yêu cầu hệ điều hành hỗ trợ cơ chế giao tiếp và cơ chế đồng bộ hoạt động của các quá trình

Bài toán người sản xuất-người tiêu thụ (producer-consumer)

- Ví dụ cộng tác giữa các quá trình: *bài toán producer-consumer*
  - *Producer* tạo ra các dữ liệu và *consumer* tiêu thụ, sử dụng các dữ liệu đó. Sự trao đổi thông tin thực hiện qua buffer
    - *unbounded buffer:* kích thước buffer vô hạn (không thực tế).
    - *bounded buffer:* kích thước buffer có hạn.
  - Producer và consumer phải hoạt động đồng bộ vì
    - Consumer không được tiêu thụ khi producer chưa sản xuất
    - Producer không được tạo thêm sản phẩm khi buffer đầy.

3.8.Giao tiếp liên quá trình (Interprocess communication-IPC)

- *IPC* là cơ chế cung cấp bởi hệ điều hành nhằm giúp các quá trình
  - giao tiếp với nhau
  - và đồng bộ hoạt động

  mà không cần chia sẻ không gian địa chỉ

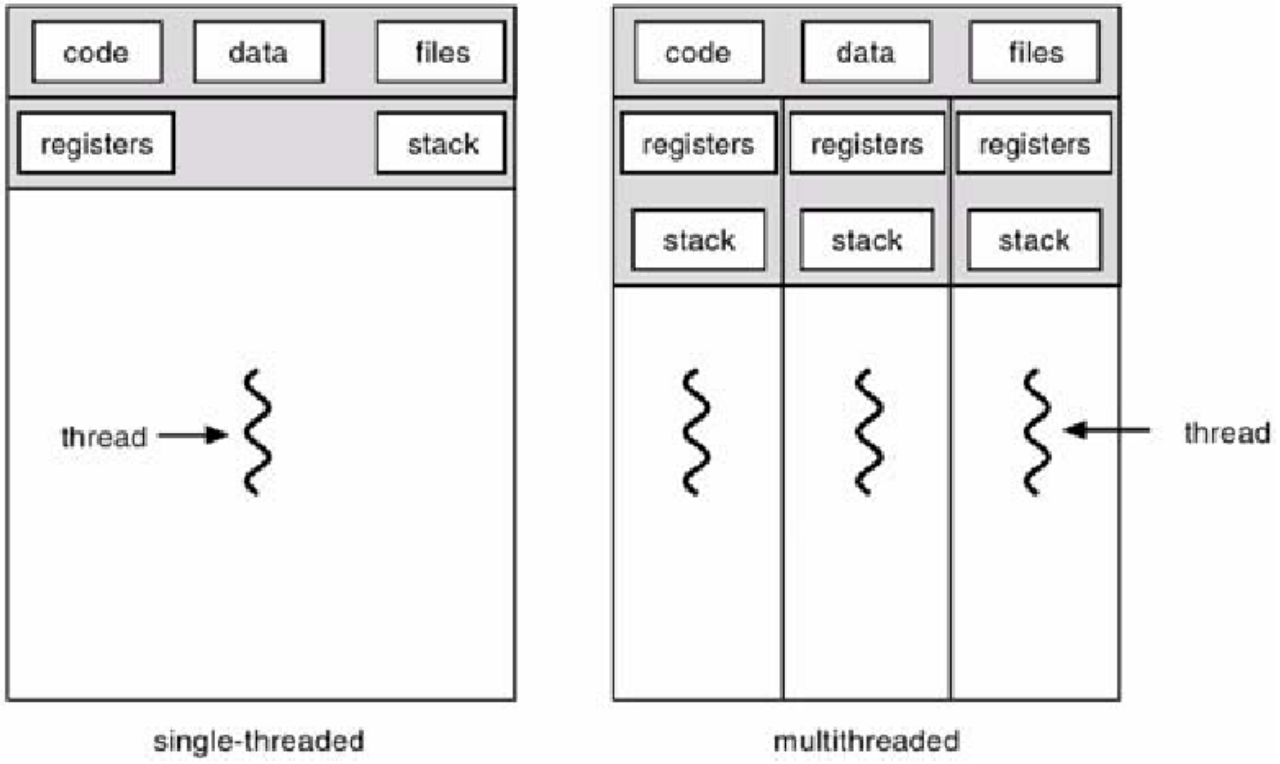
- IPC có thể được cung cấp bởi message passing system

Hệ thống truyền thông điệp
Message passing system

- Làm thế nào để các quá trình giao tiếp nhau? Các vấn đề:
  - *Đặt tên (Naming)*
    - Giao tiếp trực tiếp
      - send(P, msg): gửi thông điệp đến quá trình P
      - receive(Q, msg): nhận thông điệp đến từ quá trình Q
    - Giao tiếp gián tiếp: thông qua *mailbox* hay *port*
      - send(A, msg): gửi thông điệp đến mailbox A
      - receive(B, msg): nhận thông điệp từ mailbox B
  - *Đồng bộ hóa (Synchronization)*: blocking send, nonblocking send, blocking receive, nonblocking receive
  - *Tạo vùng đệm (Buffering)*: dùng queue để tạm chứa các message
    - Khả năng chứa là 0(Zero capacity hay no buffering)
    - Bounded capacity: độ dài của queue là giới hạn
    - Unbounded capacity: độ dài của queue là không giới hạn

Tiểu trình (luồng) - Thread

- Tiểu trình(tiến trình nhẹ-lightweight process): là một đơn vị cơ bản sử dụng CPU, gồm:
  - Thread ID, PC, Registers, stack và chia sẻ chung code, data, resources (files)



PCB vàTCB trong mô hình multithreads

PCB

pid
Threads list
Context (Mem, global ressources…)
Relatives (Dad, children)
Scheduling statistic

Thread Control Block
TCB

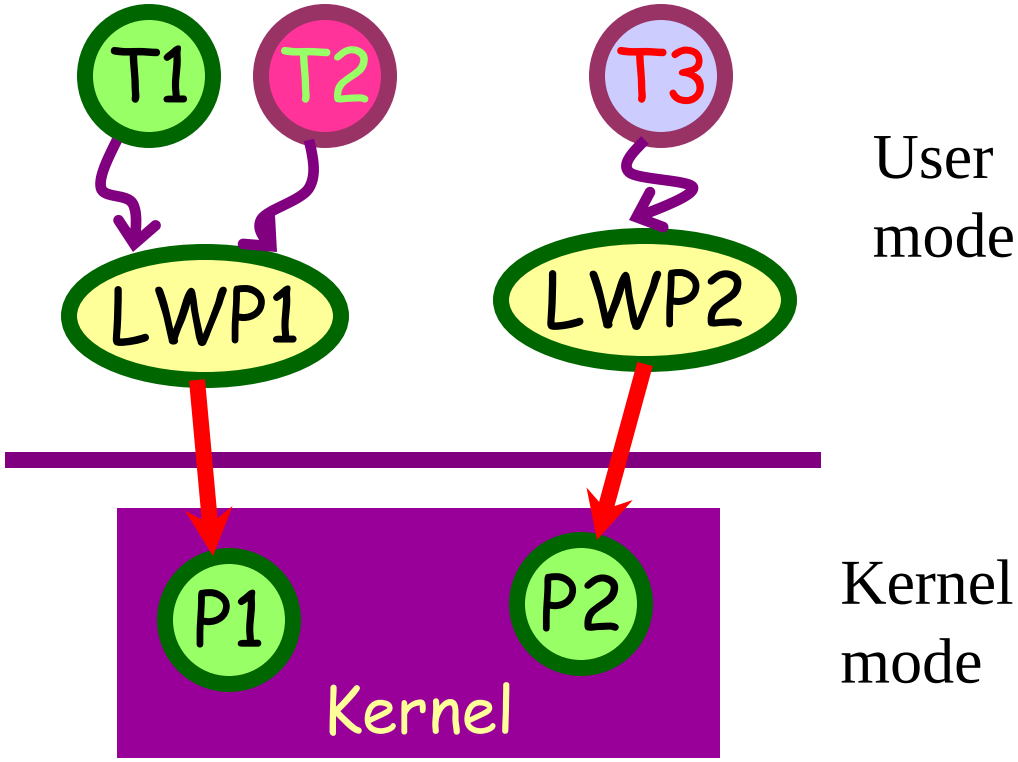
tid
State (State, details)
Context (IP, local stack…)

Lợi ích của tiến trình đa luồng

- Đáp ứng nhanh: Cho phép chương trình tiếp tục thực thi khi một bộ phận bị khóa hoặc một hoạt động dài
- Chia sẻ tài nguyên: tiết kiệm không gian nhớ
- Kinh tế: tạo và chuyển ngữ cảnh nhanh hơn tiến trình

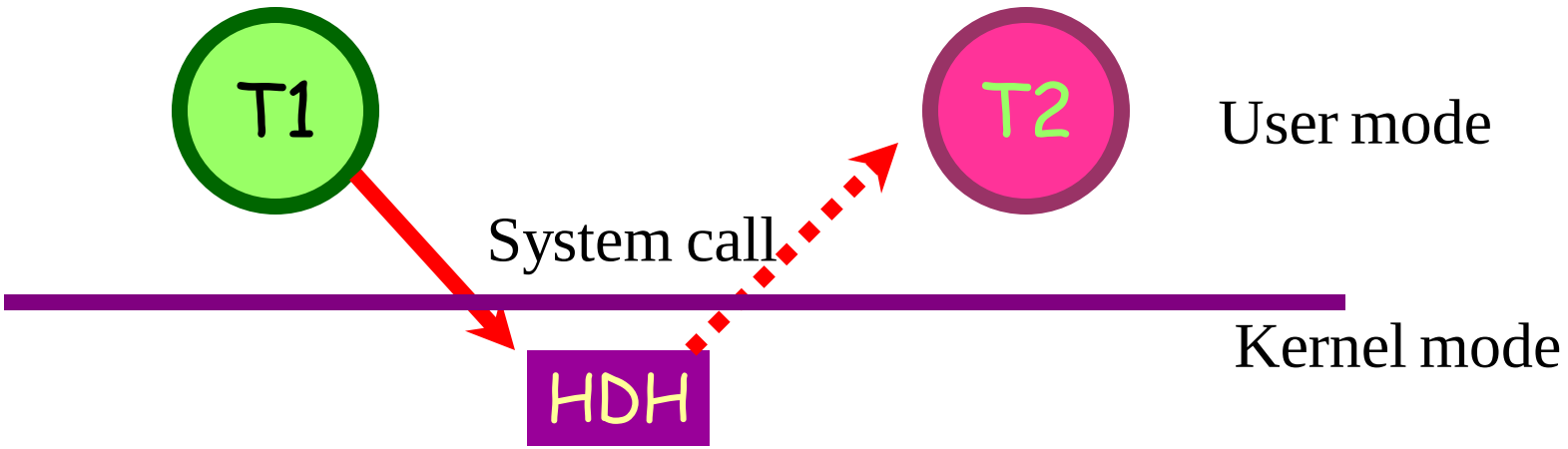
  VD: trong Solaris 2, tạo process chậm hơn 30 lần, chuyển chậm hơn 5 lần
- Trong multiprocessor: có thể thực hiện song song

Tiểu trình người dùng (User thread)



Khái niệm tiểu trình được hỗ trợ bởi một thư viện hoạt động trong user mode

Tiểu trình hạt nhân(Kernel thread)



Khái niệm tiểu trình được xây dựng bên trong hạt nhân

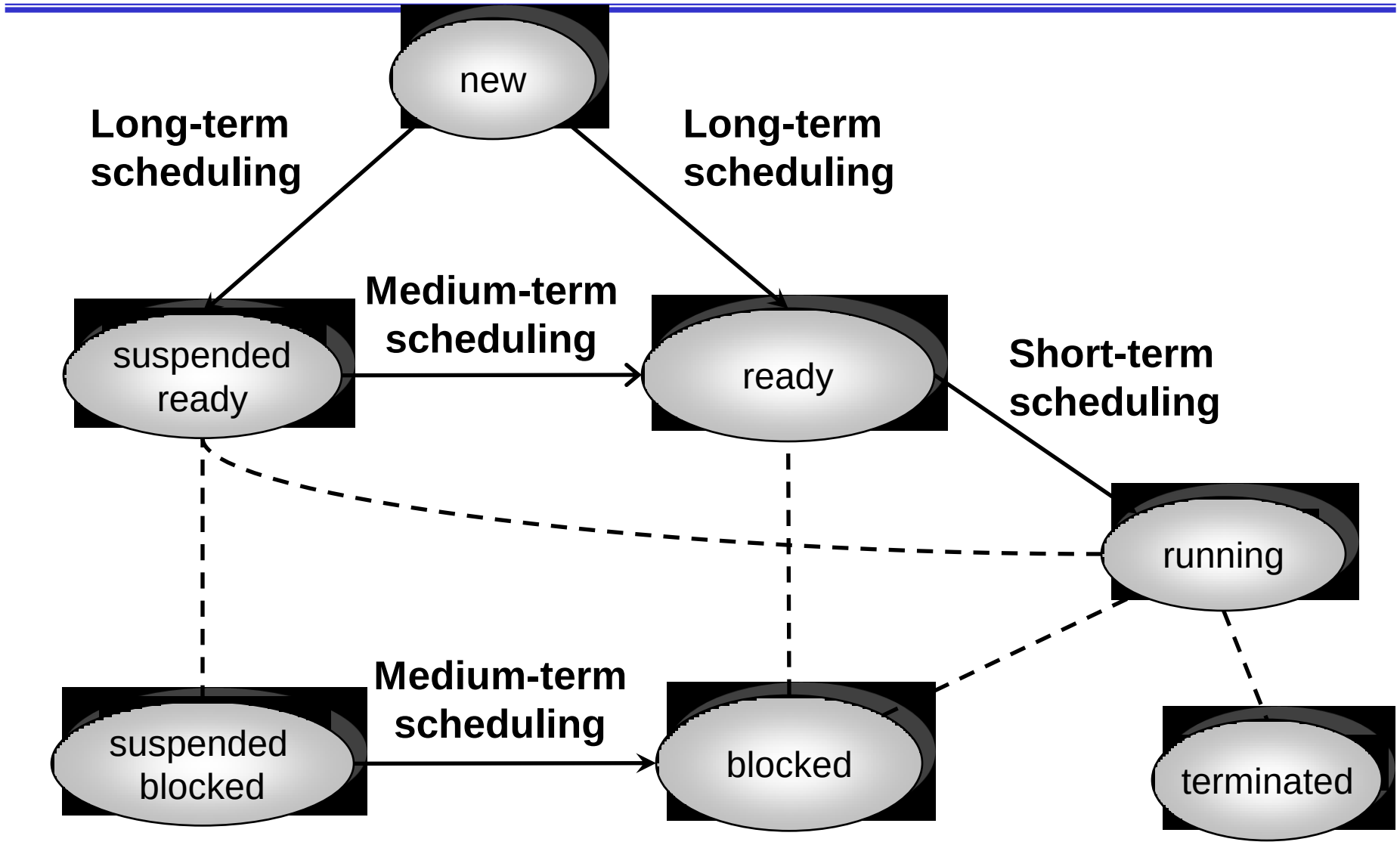
Chương IV: Định thời CPU

- Khái niệm cơ bản
- Các bộ định thời
  - long-term, mid-term, short-term
- Các tiêu chuẩn định thời CPU
- Các giải thuật định thời
  - First-Come, First-Served (FCFS)
  - Round-Robin (RR)
  - Shortest Job First (SJF) và Shortest Remaining Time First (SRTF)
  - Priority Scheduling
  - Highest Response Ratio Next (HRRN)
  - Multilevel Queue
  - Multilevel Feedback Queue

Khái niệm cơ bản

- Trong các hệ thống multitasking
  - Thực thi nhiều chương trình đồng thời làm tăng hiệu suất hệ thống.
  - Tại mỗi thời điểm, chỉ có một process được thực thi. Do đó, cần phải giải quyết vấn đề phân chia, lựa chọn process thực thi sao cho được hiệu quả nhất [redacted]iến lược định thời CPU.
- *Định thời CPU*
  - Chọn một process (từ ready queue) thực thi.
  - Với một multithreaded kernel, việc định thời CPU là do OS chọn kernel thread được chiếm CPU.

Các bộ định thời



Các bộ định thời

- *Long-term scheduling*
  - Xác định chương trình nào được chấp nhận nạp vào hệ thống để thực thi
  - Điều khiển mức độ multiprogramming của hệ thống
  - Long term scheduler thường cố gắng duy trì xen lẫn CPU-bound và I/O-bound process
- *Medium-term scheduling*
  - Process nào được đưa vào (swap in), đưa ra khỏi (swap out) bộ nhớ chính
  - Được thực hiện bởi phần quản lý bộ nhớ và được thảo luận ở phần quản lý bộ nhớ.

Các bộ định thời (tt)

- *Short term scheduling*

- Xác định process nào trong ready queue sẽ được chiếm CPU để thực thi kế tiếp (còn được gọi là định thời CPU, CPU scheduling)
- Short term scheduler còn được gọi với tên khác là *dispatcher*
- Bộ định thời short-term được gọi mỗi khi có một trong các sự kiện/interrupt sau xảy ra:

  - t thời gian (clock interrupt)
  - Ngắt ngoại vi (I/O interrupt)
  - Lời gọi hệ thống (operating system call)
  - Signal

Chương này sẽ tập trung vào định thời ngắn hạn

Dispatcher

- Dispatcher sẽ chuyển quyền điều khiển CPU về cho process được chọn bởi bộ định thời ngắn hạn
- Bao gồm:
  - Chuyển ngữ cảnh (sử dụng thông tin ngữ cảnh trong PCB)
  - Chuyển chế độ người dùng
  - Nhảy đến vị trí thích hợp trong chương trình ứng dụng để khởi động lại chương trình (chính là program counter trong PCB)
- Công việc này gây ra phí tổn
  - *Dispatch latency*: thời gian mà dispatcher dừng một process và khởi động một process khác

Các tiêu chuẩn định thời CPU

- User-oriented
  - *Thời gian đáp ứng (Response time)*: khoảng thời gian process nhận yêu cầu đến khi yêu cầu đầu tiên được đáp ứng (time-sharing, interactive system) [redacted] cực tiểu
  - *Thời gian quay vòng (hoàn thành) (Turnaround time)*: khoảng thời gian từ lúc một process được nạp vào hệ thống đến khi process đó kết thúc [redacted]rc tiểu
  - *Thời gian chờ (Waiting time)*: tổng thời gian một process đợi trong ready queue [redacted]rc tiểu
- System-oriented
  - *Sử dụng CPU (processor utilization)*: định thời sao cho CPU càng bận càng tốt [redacted]rc đại
  - *Công bằng (fairness)*: tất cả process phải được đối xử như nhau
  - *Thông lượng (throughput)*: số process hoàn tất công việc trong một đơn vị thời gian [redacted]rc đại.

Hai yếu tố của giải thuật định thời

- *Hàm chọn lựa* (selection function): dùng để chọn process nào trong ready queue được thực thi (thường dựa trên độ ưu tiên, yêu cầu về tài nguyên, đặc điểm thực thi của process,…), ví dụ
  - w = tổng thời gian đợi trong hệ thống
  - e = thời gian đã được phục vụ
  - s = tổng thời gian thực thi của process (bao gồm cả "e")

Hai yếu tố của giải thuật định thời (tt)

- *Chế độ quyết định* (decision mode): chọn thời điểm thực hiện hàm chọn lựa để định thời. Có hai chế độ
  - Không trưng dụng (Non-preemptive)
    - Khi ở trạng thái running, process sẽ thực thi cho đến khi kết thúc hoặc bị blocked do yêu cầu I/O
  - Trưng dụng (Preemptive)
    - Process đang thực thi (trạng thái running) có thể bị ngắt nửa chừng và chuyển về trạng thái ready bởi hệ điều hành
    - Chi phí cao hơn non-preemptive nhưng đánh đổi lại bằng thời gian đáp ứng tốt hơn vì không có trường hợp một process độc chiếm CPU quá lâu.

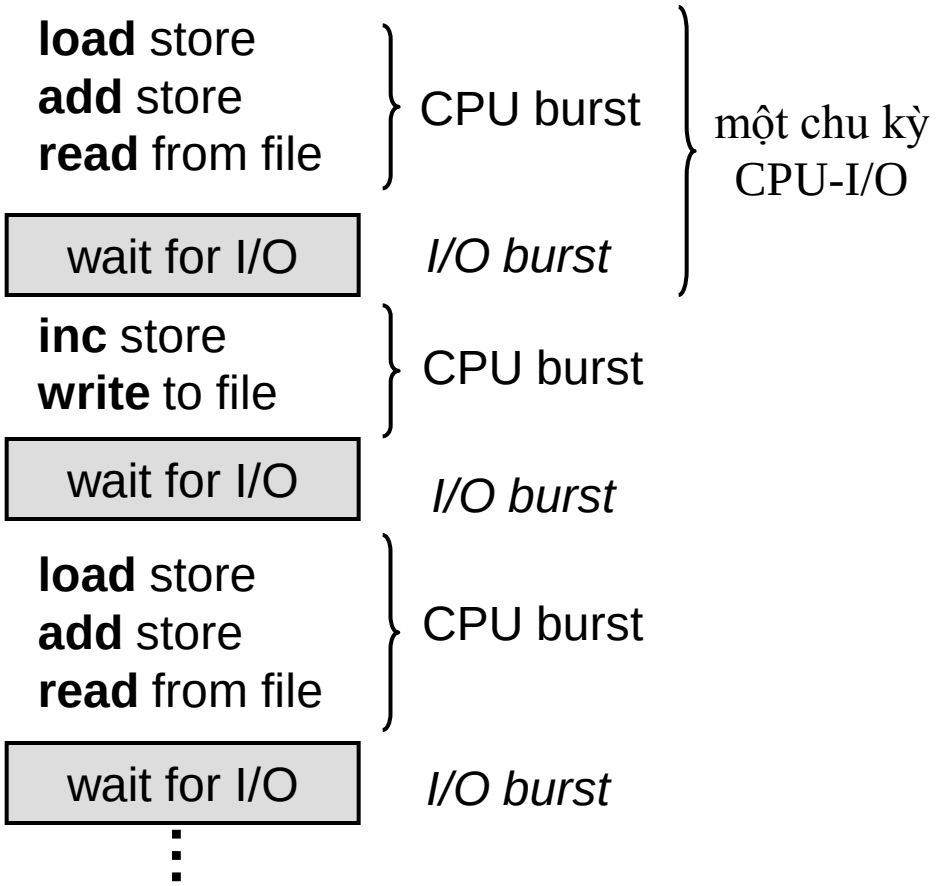
Preemptive và Non-preemptive

- Hàm định thời được thực hiện khi
  - i running sang waiting
  - i running sang ready
  - i waiting, new sang ready
  - c thi

  1 và 4 không cần lựa chọn loại định thời biểu, 2 và 3 cần
- i *nonpreemptive*
- i *preemptive*

i sao?

Khảo sát giải thuật định thời

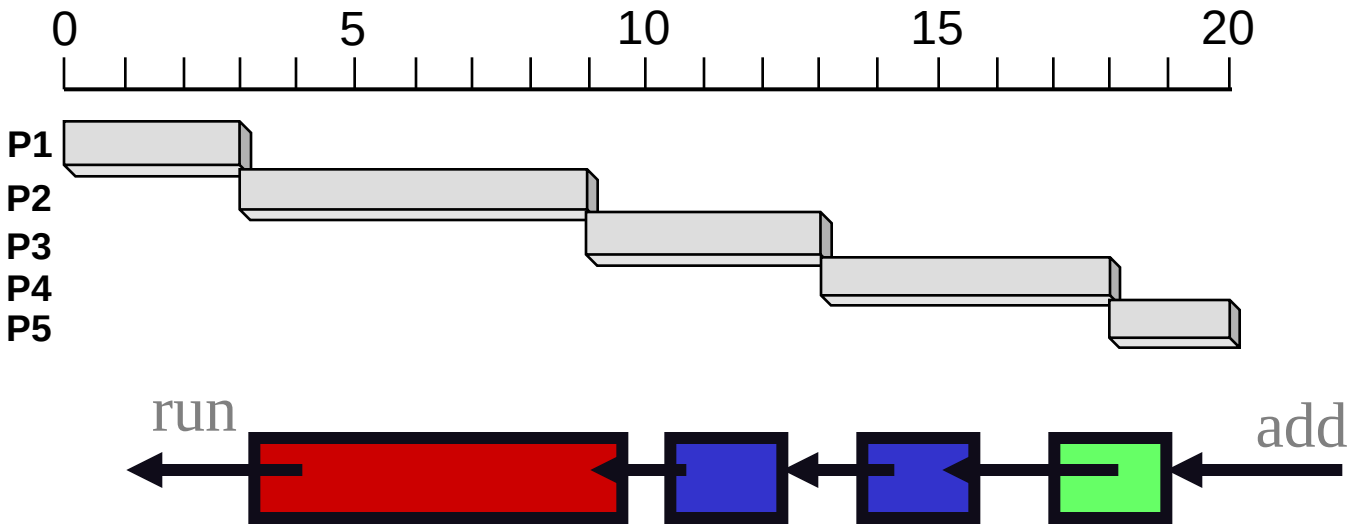


Process	*Arrival Time*	*Service Time*
1	0	3
2	2	6
3	4	4
4	6	5
5	8	2

- *Service time* = thời gian process cần CPU trong một chu kỳ CPU-I/O
- Process có service time lớn là các CPU-bound process

1. First-Come First-Served (FCFS)

- Hàm lựa chọn: Tiến trình nào yêu cầu CPU trước sẽ được cấp phát CPU trước; Process sẽ thực thi đến khi kết thúc hoặc bị blocked do I/O
- Chế độ quyết định: non-preemptive algorithm
- Hiện thực : sử dụng hàng đợi FIFO (FIFO queues)
  - Tiến trình đi vào được thêm vào cuối hàng đợi
  - Tiến trình được lựa chọn để xử lý được lấy từ đầu của queues

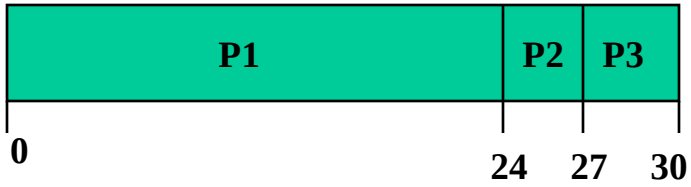


FCFS Scheduling

- Ví dụ :

Process	Burst Time
P1	24
P2	3
P3	3

Gantt Chart for Schedule

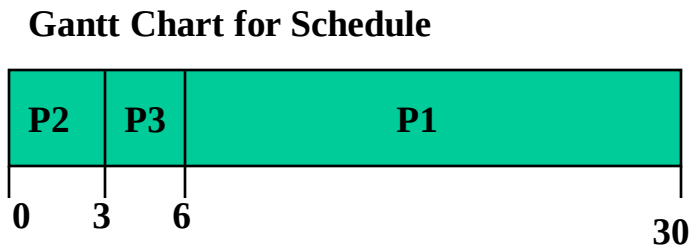


- Giả sử thứ tự vào của các tiến trình là
  - P1, P2, P3
- Thời gian chờ
  - P1 = 0;
  - P2 = 24;
  - P3 = 27;
- Thời gian chờ trung bình
  - (0+24+27)/3 = 17

FCFS Scheduling

- Ví dụ:

Process	Burst Time
P1	24
P2	3
P3	3



- Giả sử thời gian vào của các tiến trình là
  - P2, P3, P1
- Thời gian chờ :
  - P1 = 6; P2 = 0; P3 = 3;
- Thời gian chờ trung bình
  - (6+0+3)/3 = 3 , tốt hơn..

2. Shortest-Job-First(SJF) Scheduling

- Định thời biểu công việc ngắn nhất trước
- Khi CPU được tự do, nó sẽ cấp phát cho tiến trình yêu cầu ít thời gian nhất để kết thúc (tiến trình ngắn nhất)
- Liên quan đến chiều dài thời gian sử dụng CPU cho lần tiếp theo của mỗi tiến trình. Sử dụng những chiều dài này để lập lịch cho tiến trình với thời gian ngắn nhất.

2. Shortest-Job-First(SJF) Scheduling

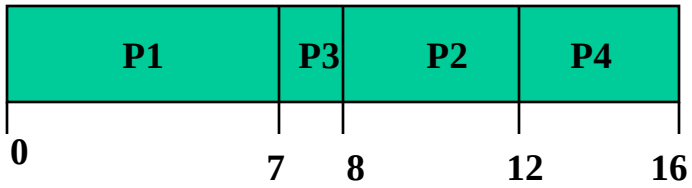
- Hai hình thức (Schemes):
  - *Scheme 1: Non-preemptive*(tiến trình độc quyền CPU)
    - Khi CPU được trao cho quá trình nó không nhường cho đến khi nó kết thúc chu kỳ xử lý của nó
  - *Scheme 2: Preemptive*(tiến trình không độc quyền)
    - Nếu một tiến trình CPU mới được đưa vào danh sách với chiều dài sử dụng CPU cho lần tiếp theo nhỏ hơn thời gian còn lại của tiến trình đang xử lý nó sẽ dừng hoạt động tiến trình hiện hành (hình thức này còn gọi là Shortest-Remaining-Time-First (SRTF).)
  - SJF là tối ưu – cho thời gian chờ đợi trung bình tối thiểu với một tập tiến trình cho trước

Non-Preemptive SJF Scheduling

- Ví dụ :

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



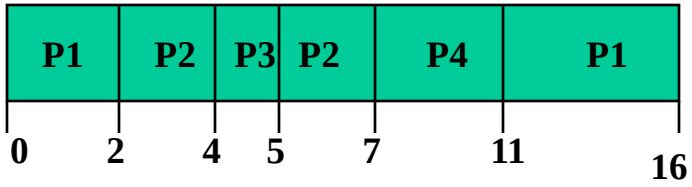
Average waiting time =
(0+6+3+7)/4 = 4

Preemptive SJF Scheduling(SRTF)

- Ví dụ 1:

Process	Arrival Time	Burst Time
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Gantt Chart for Schedule



Average waiting time =
(9+1+0+2)/4 = 3

VD2:

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5

Nhận xét về giải thuật SJF

- Có thể xảy ra tình trạng "đói" (starvation) đối với các process có CPU-burst lớn khi có nhiều process với CPU-burst nhỏ đến hệ thống.
- Cơ chế non-preemptive không phù hợp cho hệ thống time sharing (interactive)
- Giải thuật SJF ngầm định ra độ ưu tiên theo burst time
- Các CPU-bound process có độ ưu tiên thấp hơn so với I/O-bound process, nhưng khi một process không thực hiện I/O được thực thi thì nó độc chiếm CPU cho đến khi kết thúc

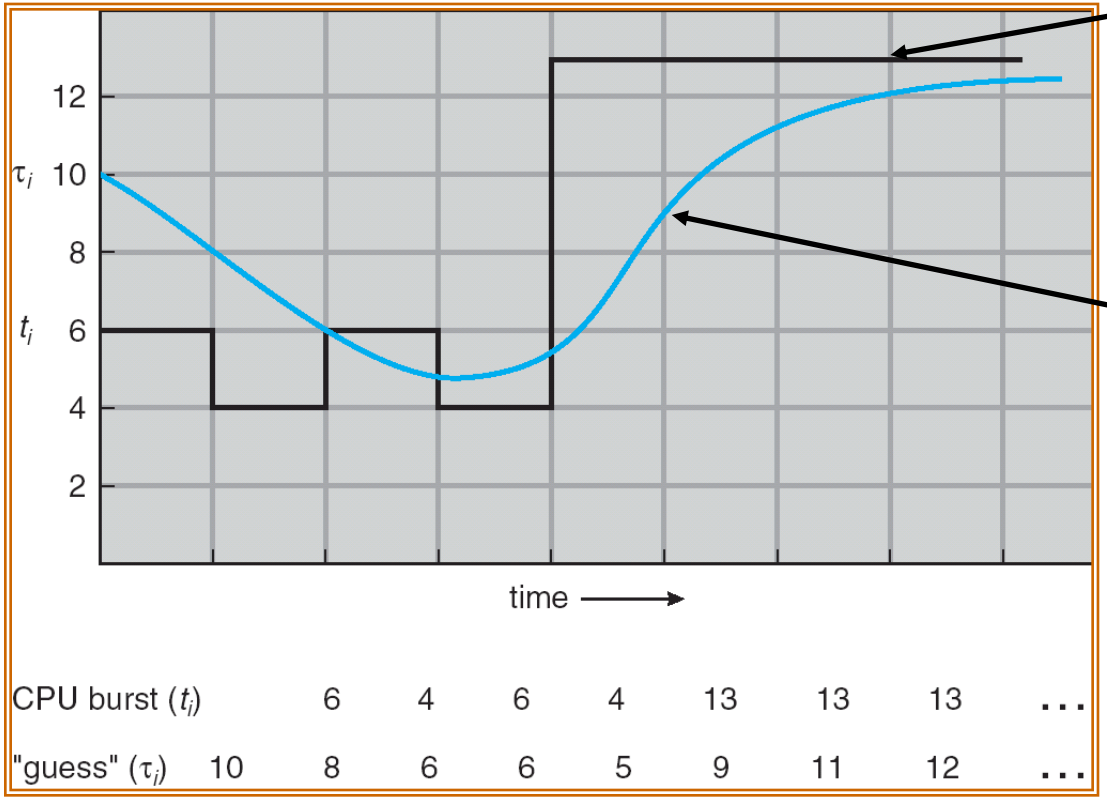
Nhận xét về giải thuật SJF

- Tương ứng với mỗi process cần có độ dài của CPU burst tiếp theo
- Hàm lựa chọn: chọn process có độ dài CPU burst nhỏ nhất
- Chứng minh được: SJF tối ưu trong việc giảm thời gian đợi trung bình
- Nhược điểm: Cần phải ước lượng thời gian cần CPU tiếp theo của process
- Giải pháp cho vấn đề này?

Nhận xét về giải thuật SJF

- (Thời gian sử dụng CPU chính là độ dài của CPU burst)
- Trung bình tất cả các CPU burst đo được trong quá khứ
- Nhưng thông thường những CPU burst càng mới càng phản ánh đúng hành vi của process trong tương lai
- Một kỹ thuật thường dùng là sử dụng *trung bình hàm mũ* (exponential averaging)
  - $\tau_{n+1} = a\, t_n + (1 - a)\, \tau_n$, $0 \leq a \leq 1$
  - $\tau_{n+1} = a\, t_n + (1- a)\, a\, t_{n-1} + \ldots + (1- a)^j a \tau_{n-j} + \ldots + (1- a)^{n+1} a \tau_0$
  - Nếu chọn $a = ½$ thì có nghĩa là trị đo được t_n và trị dự đoán τ_n được xem quan trọng như nhau.

Dự đoán thời gian sử dụng CPU



Độ dài CPU burst đo được

Độ dài CPU burst dự đoán, với a = ½ và [redacted] = 10

3. Priority Scheduling

- Mỗi process sẽ được gán một độ ưu tiên
- CPU sẽ được cấp cho process có độ ưu tiên cao nhất
- Định thời sử dụng độ ưu tiên có thể:
  - Preemptive hoặc
  - Nonpreemptive

Gán độ ưu tiên

- SJF là một giải thuật định thời sử dụng độ ưu tiên với độ ưu tiên là thời-gian-sử-dụng-CPU-dự-đoán
- Gán độ ưu tiên còn dựa vào:
  - Yêu cầu về bộ nhớ
  - Số lượng file được mở
  - Tỉ lệ thời gian dùng cho I/O trên thời gian sử dụng CPU
  - Các yêu cầu bên ngoài ví dụ như: số tiền người dùng trả khi thực thi công việc

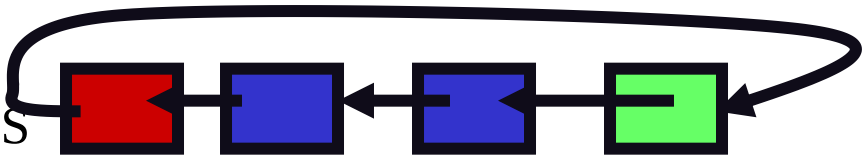
3. Priority Scheduling

- Vấn đề: trì hoãn vô hạn định – process có độ ưu tiên thấp có thể không bao giờ được thực thi
- Giải pháp: làm mới (*aging)* – độ ưu tiên của process sẽ tăng theo thời gian
- Vd:

4. Định thời luân phiên (Round Robin -RR)

- Mỗi process nhận được một đơn vị nhỏ thời gian CPU (time slice, quantum time), thông thường từ 10-100 msec để thực thi. Sau khoảng thời gian đó, process bị đoạt quyền và trở về cuối hàng đợi ready.
- Nếu có n process trong hàng đợi ready và quantum time = q thì không có process nào phải chờ đợi quá (n [redacted])q đơn vị thời gian.

- Hiệu suất
  - Nếu q lớn: RR [redacted] CFS



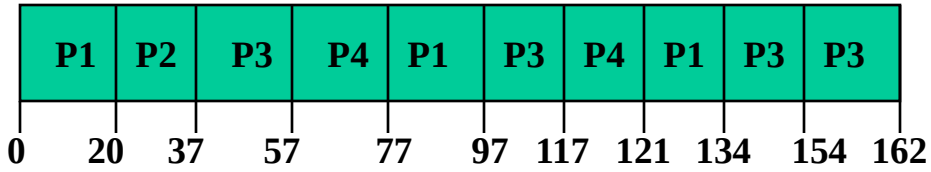
  - Nếu q nhỏ (q không được quá nhỏ bởi vì phải tốn chi phí chuyển ngữ cảnh)
- Thời gian chờ đợi trung bình của giải thuật RR thường khá lớn nhưng thời gian đáp ứng nhỏ

Ví dụ Round Robin

- Time Quantum = 20

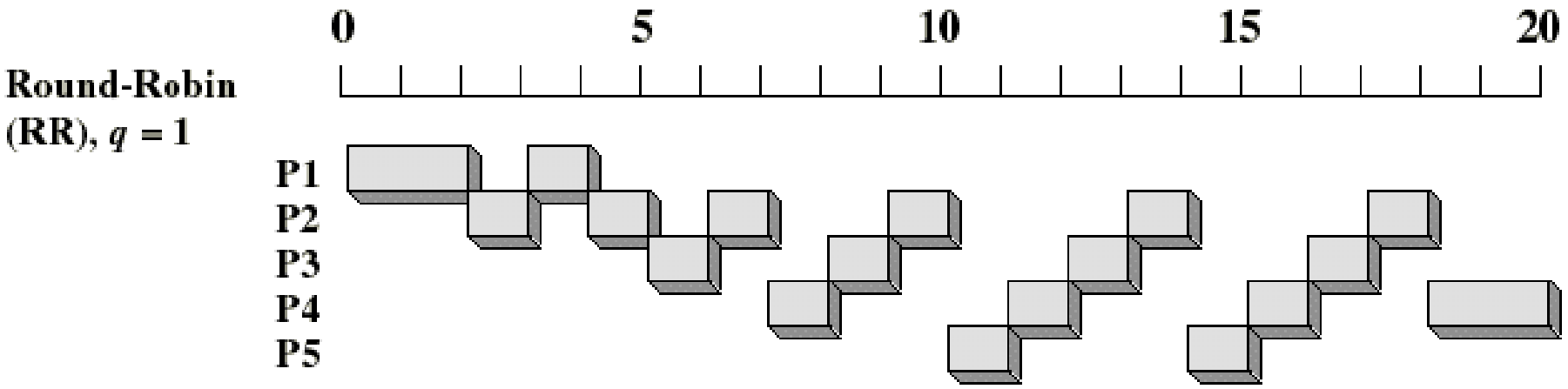
Process	Burst Time
P1	53
P2	17
P3	68
P4	24

Gantt Chart for Schedule



turnaround time trung bình lớn hơn SJF, nhưng đáp ứng tốt hơn

RR với time quantum = 1

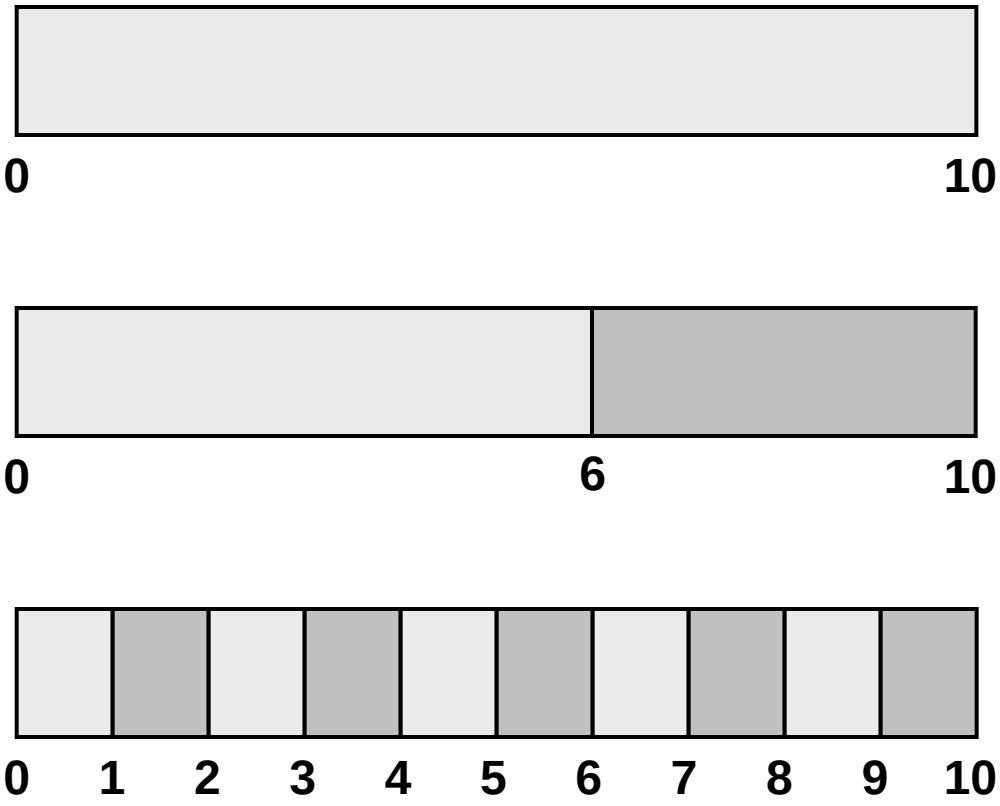


- Thời gian turn-around trung bình cao hơn so với SJF nhưng có thời gian đáp ứng trung bình tốt hơn.
- Ưu tiên CPU-bound process
  - I/O-bound process thường sử dụng rất ít thời gian của CPU, sau đó phải blocked đợi I/O
  - CPU-bound process tận dụng hết quantum time, sau đó quay về ready queue [redacted]ợc xếp trước các process bị blocked

Time quantum và context switch

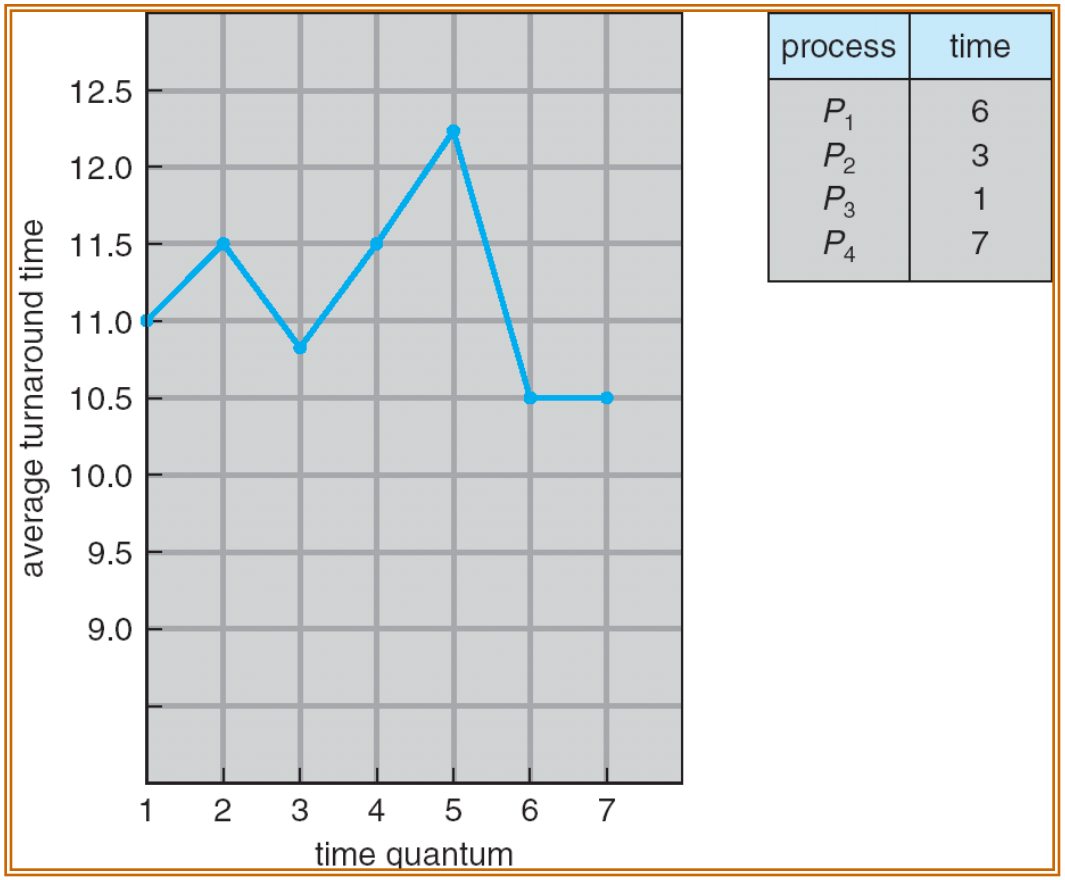
Process time = 10

quantum	context switch
12	0
6	1
1	9



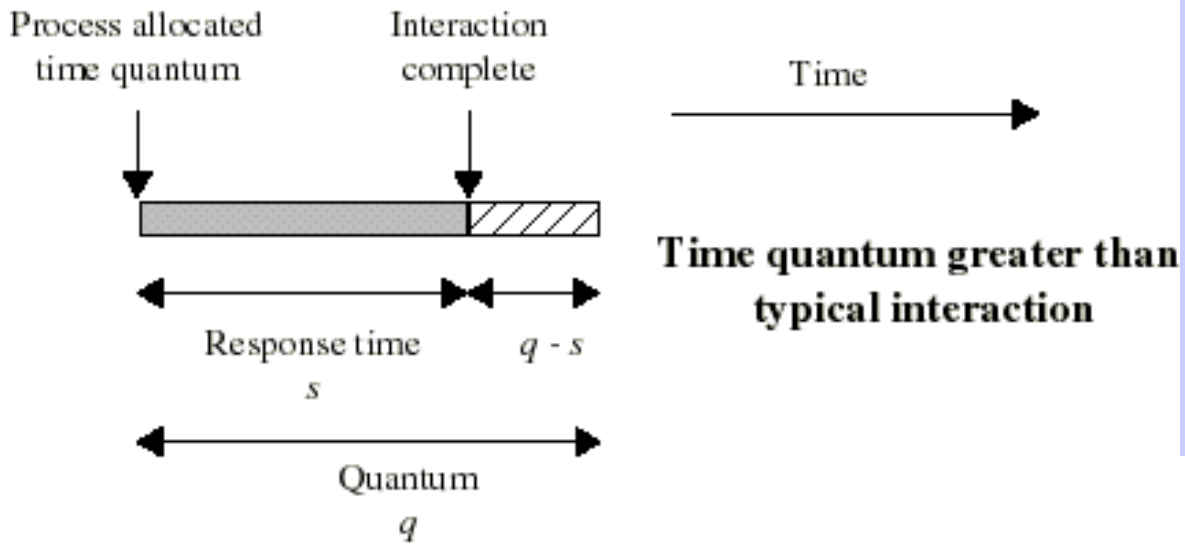
Thời gian hoàn thành và quantum time

- Thời gian hoàn thành trung bình (average turnaround time) không chắc sẽ được cải thiện khi quantum lớn

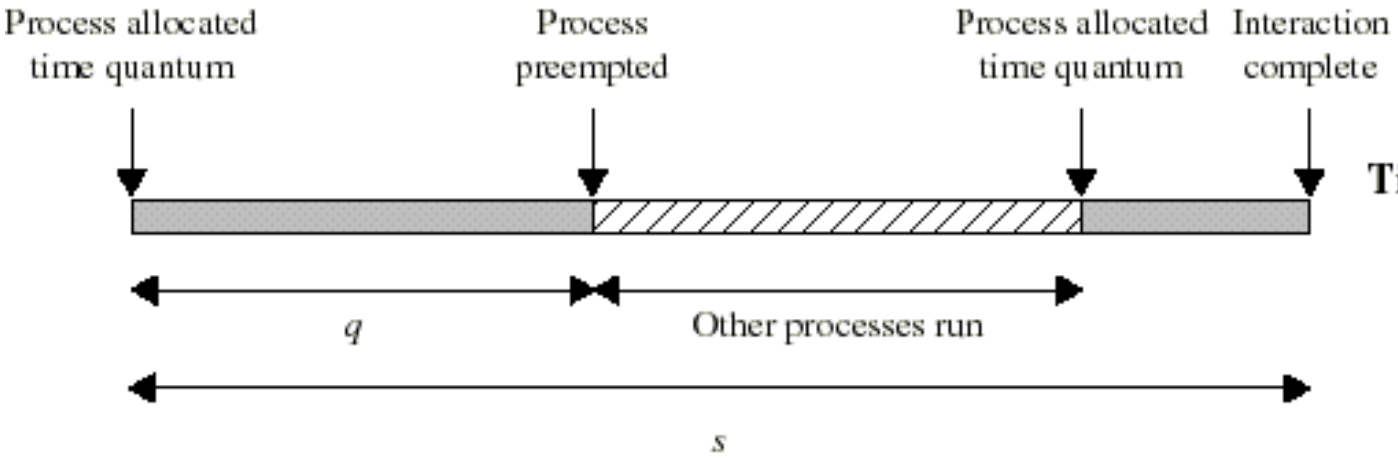


process	time
P_1	6
P_2	3
P_3	1
P_4	7

Quantum và response time



Time quantum greater than typical interaction



Time quantum less than typical interaction

- Quantum time phải lớn hơn thời gian dùng để xử lý clock interrupt (timer) và thời gian dispatching
- Nên lớn hơn thời gian tương tác trung bình (typical interaction)

Quantum time cho Round Robin*

- Khi thực hiện process switch thì OS sẽ sử dụng CPU chứ không phải process của người dùng (*OS overhead*)
  - Dừng thực thi, lưu tất cả thông tin, nạp thông tin của process sắp thực thi
- Performance tùy thuộc vào kích thước của quantum time (còn gọi là time slice), và hàm phụ thuộc này không đơn giản
- Time slice ngắn thì đáp ứng nhanh
  - Vấn đề: có nhiều chuyển ngữ cảnh. Phí tổn sẽ cao.
- Time slice dài hơn thì throughput tốt hơn (do giảm phí tổn OS overhead) nhưng thời gian đáp ứng lớn
  - Nếu time slice quá lớn, RR trở thành FCFS.

Quantum time cho Round Robin

- Quantum time và thời gian cho process switch:
  - Nếu quantum time = 20 ms và thời gian cho process switch = 5 ms, như vậy phí tổn OS overhead chiếm 5/25 = 20%
  - Nếu quantum = 500 ms, thì phí tổn chỉ còn [redacted]%
    - Nhưng nếu có nhiều người sử dụng trên hệ thống và thuộc loại interactive thì sẽ thấy đáp ứng rất chậm
  - Tùy thuộc vào tập công việc mà lựa chọn quantum time
  - Time slice nên lớn trong tương quan so sánh với thời gian cho process switch
  - Ví dụ với 4.3 BSD UNIX, time slice là 1 giây

Round Robin

- Nếu có *n* process trong hàng đợi ready, và quantum time là *q*, như vậy mỗi process sẽ lấy 1/*n* thời gian CPU theo từng khối có kích thước lớn nhất là *q*
  - Sẽ không có process nào chờ lâu hơn (*n* - 1)*q* đơn vị thời gian
- RR sử dụng một giả thiết ngầm là tất cả các process đều có tầm quan trọng ngang nhau
  - Không thể sử dụng RR nếu muốn các process khác nhau có độ ưu tiên khác nhau

Round Robin: nhược điểm

- Các process dạng CPU-bound vẫn còn được "ưu tiên"
  - Ví dụ:
    - Một I/O-bound process sử dụng CPU trong thời gian ngắn hơn quantum time và bị blocked để đợi I/O. Và
    - Một CPU-bound process chạy hết time slice và lại quay trở về hàng đợi ready queue (ở phía trước các process đã bị blocked)

5. Highest Response Ratio Next

$$RR = \frac{\text{time spent waiting} + \text{expected service time}}{\text{expected service time}}$$

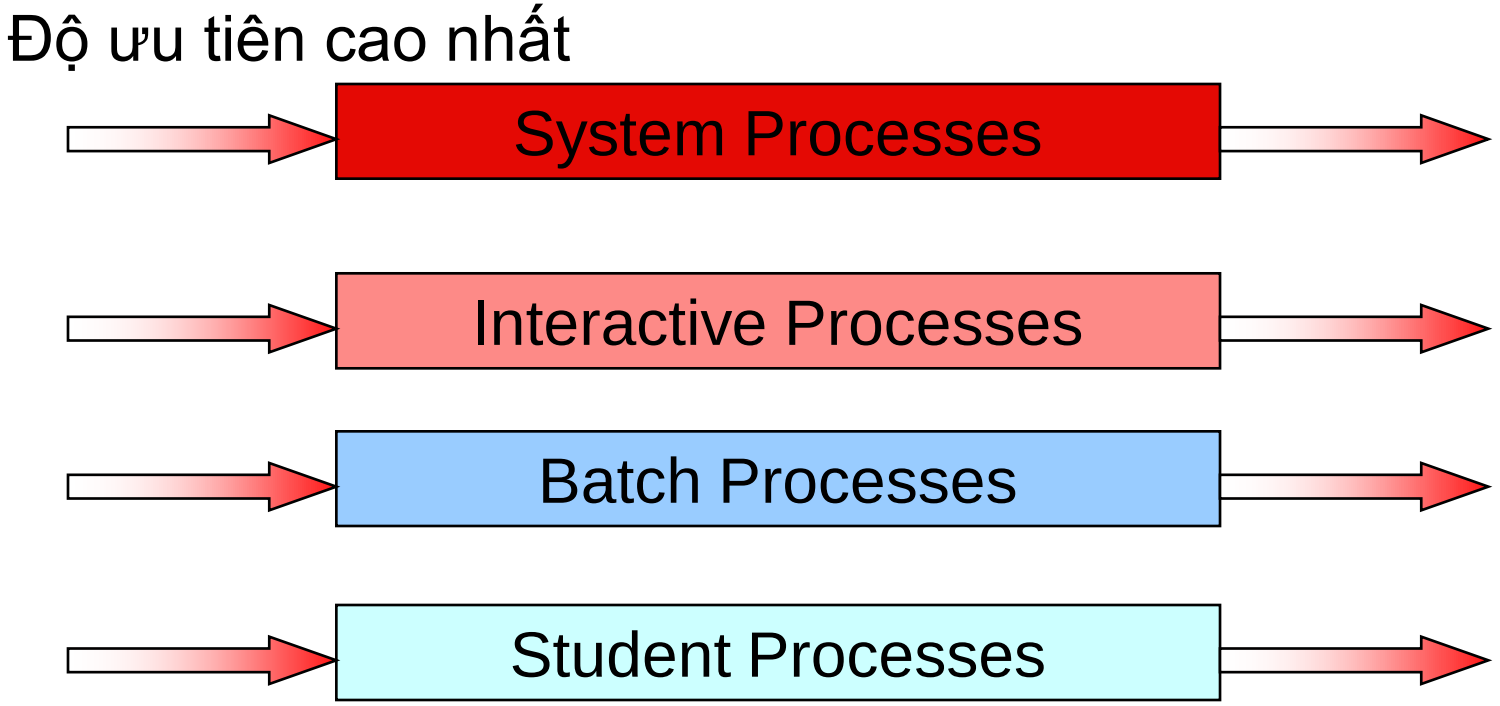
- Chọn process kế tiếp có giá trị *RR* (Response ratio) lớn nhất
- Các process ngắn được ưu tiên hơn (vì service time nhỏ)

6. Multilevel Queue Scheduling

- Hàng đợi ready được chia thành nhiều hàng đợi riêng biệt theo một số tiêu chuẩn như
  - Đặc điểm và yêu cầu định thời của process
  - Foreground (interactive) và background process,…
- Process được gán cố định vào một hàng đợi, mỗi hàng đợi sử dụng giải thuật định thời riêng
- Hệ điều hành cần phải định thời cho các hàng đợi.
  - *Fixed priority scheduling*: phục vụ từ hàng đợi có độ ưu tiên cao đến thấp. Vấn đề: có thể có starvation.
  - *Time slice*: mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó. Ví dụ: 80% cho hàng đợi foreground định thời bằng RR và 20% cho hàng đợi background định thời bằng giải thuật FCFS.

Multilevel Queue Scheduling*

- Ví dụ phân nhóm các quá trình

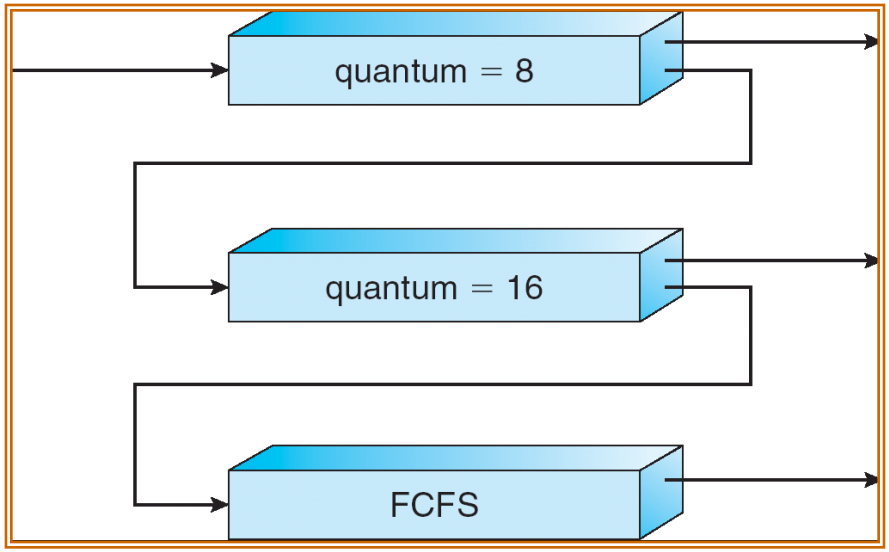


7. Hàng đợi phản hồi đa cấp
Multilevel Feedback Queue

- Vấn đề của multilevel queue
  - process không thể chuyển từ hàng đợi này sang hàng đợi khác [redacted] khắc phục bằng cơ chế feedback: cho phép process di chuyển một cách thích hợp giữa các hàng đợi khác nhau.
- *Multilevel Feedback Queue*
  - Phân loại processes dựa trên các đặc tính về CPU-burst
  - Sử dụng decision mode preemptive
  - Sau một khoảng thời gian nào đó, các I/O-bound process và interactive process sẽ ở các hàng đợi có độ ưu tiên cao hơn còn CPU-bound process sẽ ở các queue có độ ưu tiên thấp hơn.
  - Một process đã chờ quá lâu ở một hàng đợi có độ ưu tiên thấp có thể được chuyển đến hàng đợi có độ ưu tiên cao hơn (cơ chế *niên hạn*, aging).

7. Multilevel Feedback Queue

- Ví dụ: Có 3 hàng đợi
  - *Q*0 , dùng RR với quantum 8 ms
  - *Q*1 , dùng RR với quantum 16 ms
  - *Q*2 , dùng FCFS



7. Multilevel Feedback Queue (tt)

- Định thời dùng multilevel feedback queue đòi hỏi phải giải quyết các vấn đề sau
  - Số lượng hàng đợi bao nhiêu là thích hợp?
  - Dùng giải thuật định thời nào ở mỗi hàng đợi?
  - Làm sao để xác định thời điểm cần chuyển một process đến hàng đợi cao hơn hoặc thấp hơn?
  - Khi process yêu cầu được xử lý thì đưa vào hàng đợi nào là hợp lý nhất?

So sánh các giải thuật

- Giải thuật định thời nào là tốt nhất?
- Câu trả lời phụ thuộc các yếu tố sau:
  - Tần xuất tải việc (System workload)
  - Sự hỗ trợ của phần cứng đối với dispatcher
  - Sự tương quan về trọng số của các tiêu chuẩn định thời như response time, hiệu suất CPU, throughput,…
  - Phương pháp định lượng so sánh

Đọc thêm

- Policy và Mechanism
- Định thời trên hệ thống multiprocessor
- Đánh giá giải thuật định thời CPU
- Định thời trong một số hệ điều hành thông dụng

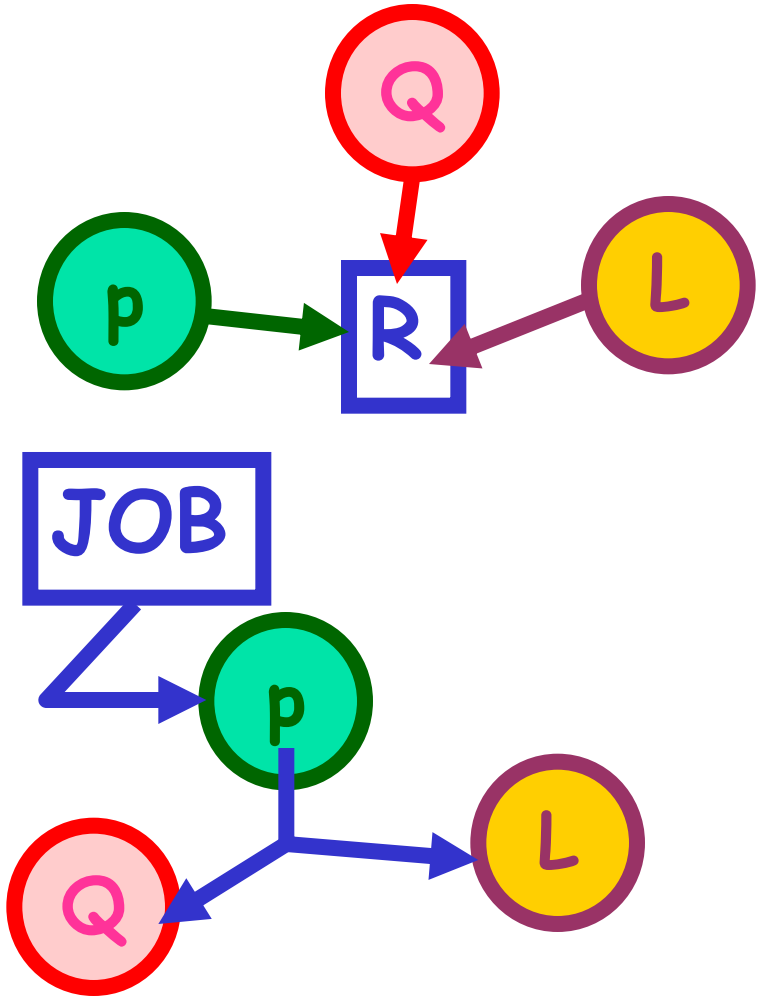
- Nguồn:

Operating System Concepts. Sixth Edition. John Wiley & Sons, Inc. 2002. Silberschatz, Galvin, Gagne

Chương V-I: Liên lạc giữa các Tiến Trình

Nhu Cầu Liên Lạc

- **Chia sẻ thơng tin**
- **Phối hợp tăng tốc độ xử lý**



Các Cơ Chế Liên Lạc

Signal : Khơng truyền được dữ liệu

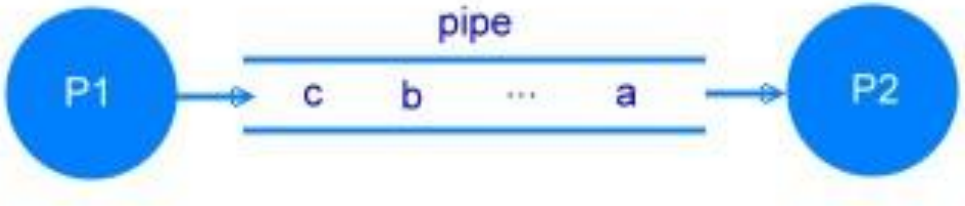
Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím DEL để ngắt xử lý tiến trình
SIGQUIT	Yêu cầu thoát xử lý
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi floating – point xảy ra (chia cho 0)
SIGPIPE	Tiến trình ghi dữ liệu vào pipe mà không có reader
SIGSEGV	Tiến trình truy xuất đến một địa chỉ bất hợp lệ
SIGCLD	Tiến trình con kết thúc
SIGUSR1	Tín hiệu 1 do người dùng định nghĩa
SIGUSR2	Tín hiệu 2 do người dùng định nghĩa

Các tín hiệu được gửi đi bởi?khi nhận thì xử lý ra sao?

Các Cơ Chế Liên Lạc

- **Pipe**

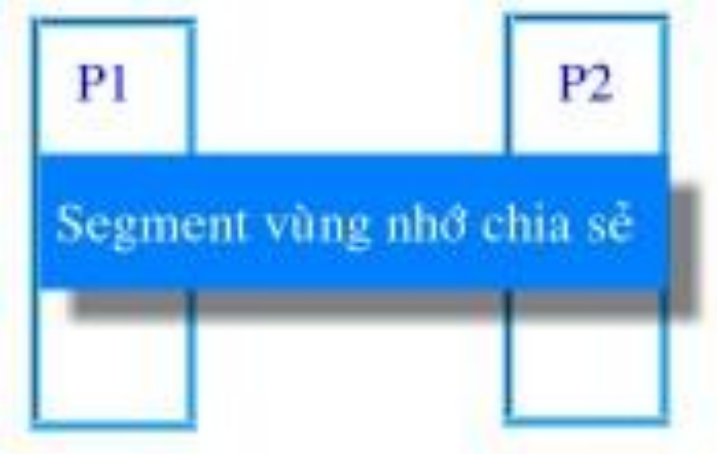
  Truyền dữ liệu không cấu trúc



Các Cơ Chế Liên Lạc

- **Shared Memory**

  Mâu thuẫn truy xuất => nhu cầu đồng bộ hoá



Các Cơ Chế Liên Lạc

- **Message**

  Liên lạc trên môi trường phân tán

- **Liên kết tiềm ẩn**
  - Send(message) : gởi một thông điệp
  - Receive(message) : nhận một thông điệp
- **Liên kết tường minh**
  - Send(destination, message) : gởi một thông điệp đến *destination*
  - Receive(source,message) : nhận một thông điệp từ *source*

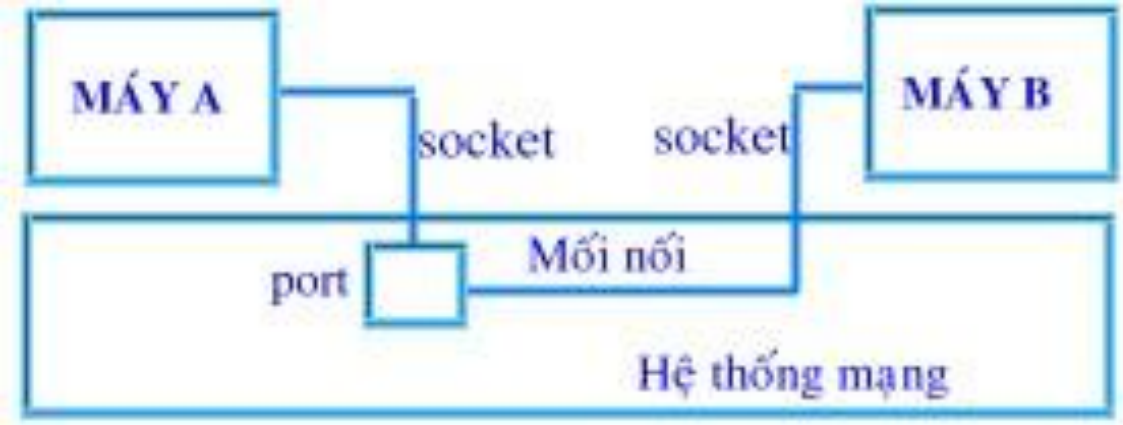
Các Cơ Chế Liên Lạc

- **Socket:** là một thiết bị truyền thông hai chiều như tập tin
- Mỗi Socket là một thành phần trong một mối nối giữa các máy trong mạng
- Các thuộc tính của socket:
  - Domaine: định nghĩa dạng thức địa chỉ và các nghi thức sử dụng. Có nhiều domaines, ví dụ UNIX, INTERNET, XEROX_NS, ...
  - Type: định nghĩa các đặc điểm liên lạc
    - a) độ tin cậy
    - b) độ bảo toàn thứ tự dữ liệu
    - c) Lặp lại dữ liệu
    - d) Chế độ nối kết
    - e) Bảo toàn giới hạn thông điệp
    - f) Khả năng gởi thông điệp khẩn

Các Cơ Chế Liên Lạc

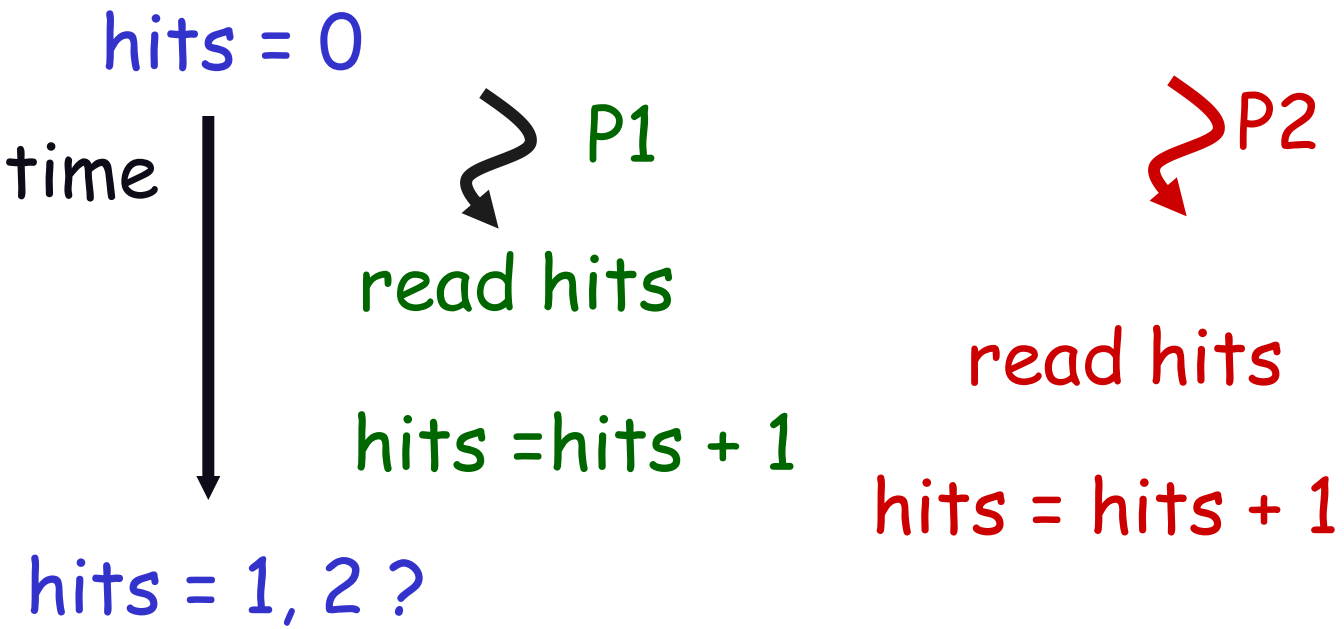
- Để thực hiện liên lạc bằng socket, cần tiến hành các thao tác ::
  - Tạo lập hay mở một socket
  - Gắn kết một socket với một địa chỉ
  - Liên lạc : có hai kiểu liên lạc tùy thuộc vào chế độ nối kết:
    - Liên lạc trong chế độ không liên kết
    - Liên lạc trong chế độ nối kết
  - Hủy một socket

VD: Giao tiếp trong TCP



Race condition

- **P1 và P2 chia sẻ biến chung hits**

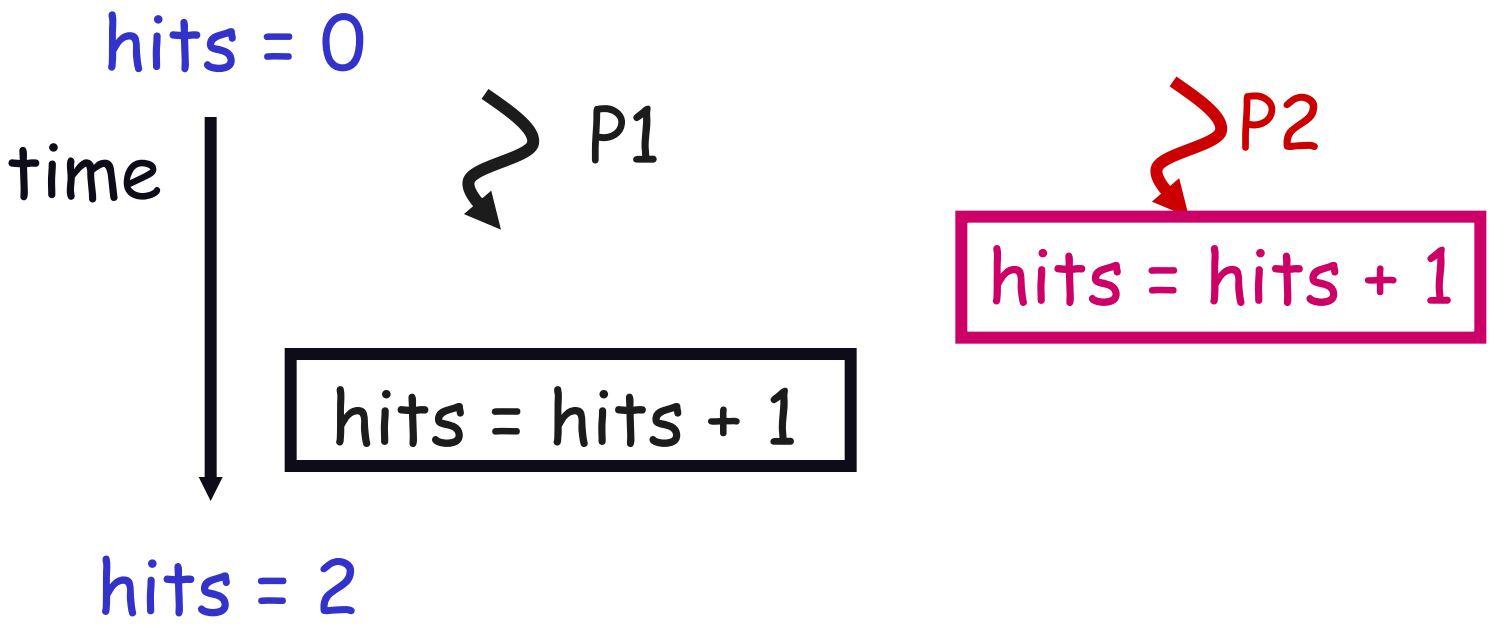


Kết quả cuối cùng khơng dự đốn được !

Vùng tranh chấp (Miền găng - critical section)

CS là đoạn chương trình có khả năng gây ra hiện tượng race condition

Giải pháp tổng quát



Bảo đảm tính "độc quyền truy xuất" miền găng tại một thời điểm

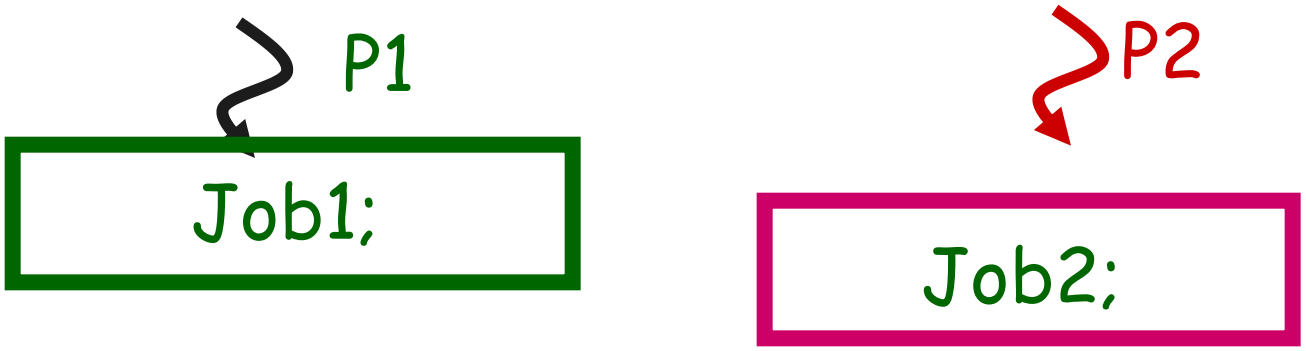
Mô hình đảm bảo độc quyền truy xuất

Kiểm tra và dành quyền vào CS

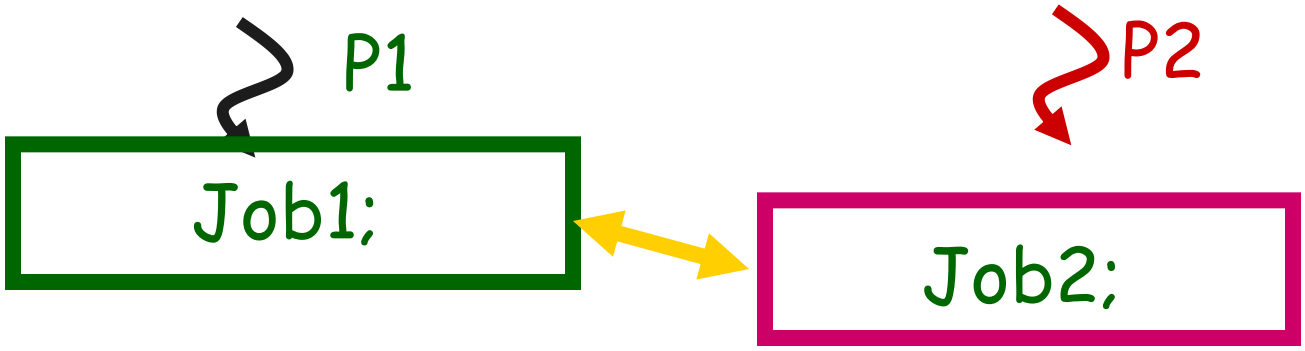
CS;

Từ bỏ quyền sử dụng CS

Hẹn hò

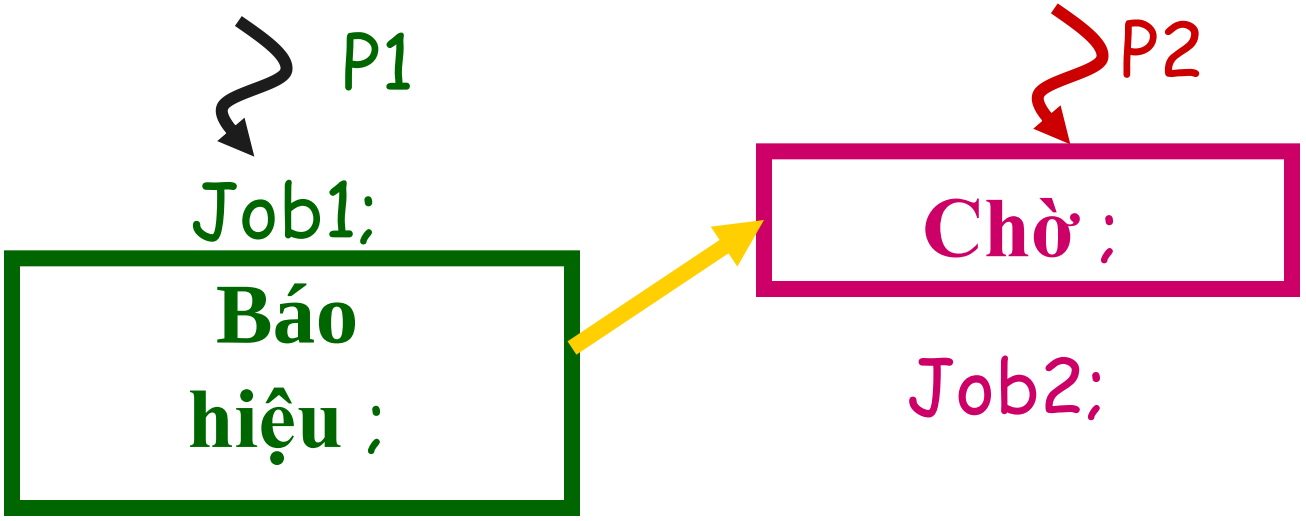


Làm thế nào bảo đảm trình tự thực hiện Job1 - Job2 ?



Hai tiến trình cần trao đổi thông tin về diễn tiến xử lý

Mô hình tổ chức phối hợp hoạt động giữa hai tiến trình



Chương V - Phần II

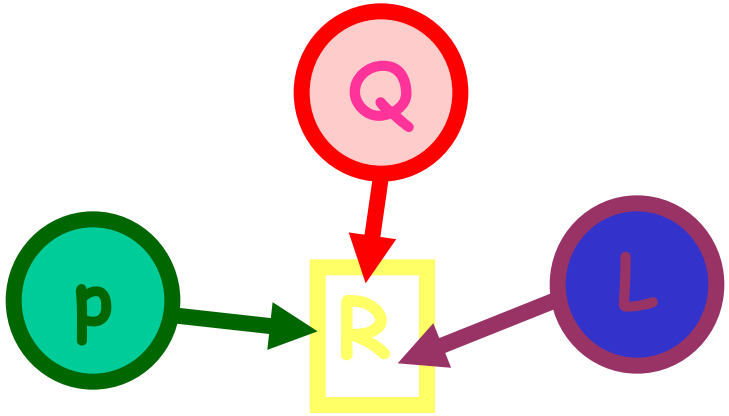
Đồng Bộ và Giải Quyết Tranh Chấp (Process Synchronization)

Nội dung

- Đặt vấn đề (tại sao phải đồng bộ và giải quyết tranh chấp ?)
- Vấn đề Critical section
- Các giải pháp phần mềm
  - Giải thuật Peterson, và giải thuật bakery
- Đồng bộ bằng hardware
- Semaphore
- Các bài toán đồng bộ
- Critical region
- Monitor

Đặt vấn đề

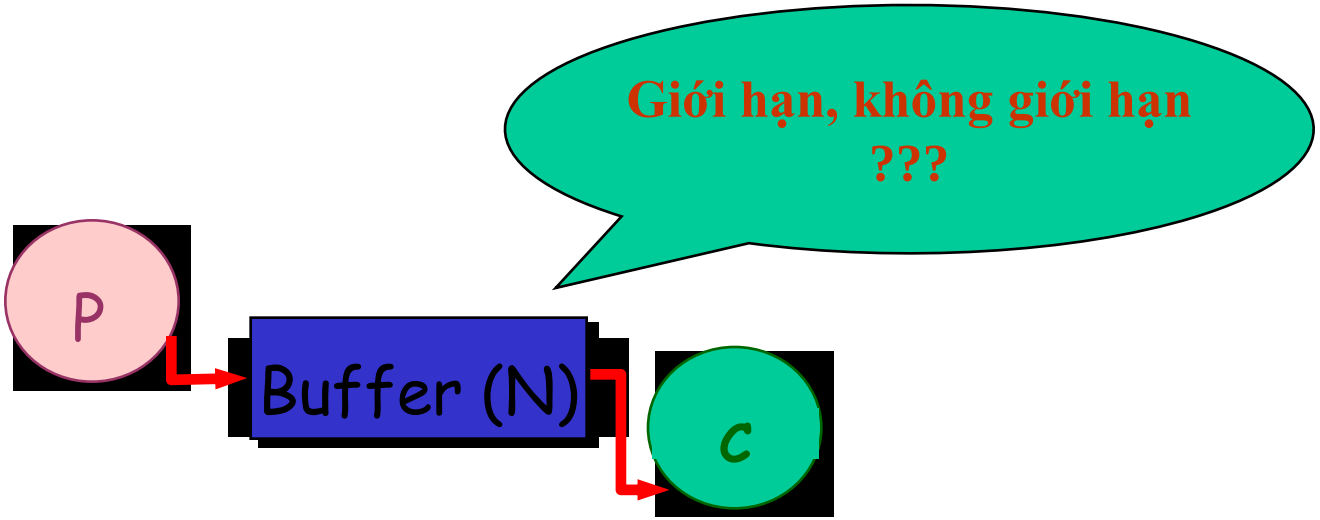
- Khảo sát các process/thread thực thi đồng thời và chia sẻ dữ liệu (qua shared memory, file).
- Nếu không có sự kiểm soát khi truy cập các dữ liệu chia sẻ thì có thể đưa đến ra trường hợp *không nhất quán dữ liệu* (data inconsistency).
- Để duy trì sự nhất quán dữ liệu, hệ thống cần có cơ chế bảo đảm sự thực thi có trật tự của các process đồng thời.



Bài toán Producer-Consumer

Producer-Consumer

- **P không được ghi dữ liệu vào buffer đã đầy**
- **C không được đọc dữ liệu từ buffer đang trống**
- **P và C không được thao tác trên buffer cùng lúc**



Đặt vấn đề

- Xét bài toán Producer-Consumer với bounded buffer
- Bounded buffer, thêm biến đếm ***count***

```
#define BUFFER_SIZE 10     /* 10 buffers */
typedef struct {
        . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0, out = 0, count = 0;
```

Bounded buffer (tt)

- Quá trình Producer

```
item nextProduced;
while(1) {
        while (count == BUFFER_SIZE); /* do nothing */
        buffer[in] = nextProduced;
        count++;
        in = (in + 1) % BUFFER_SIZE;
}
```

- Quá trình Consumer

```
item nextConsumed;
while(1) {
        while (count == 0); /* do nothing */
        nextConsumed = buffer[out] ;
        count--;
        out = (out + 1) % BUFFER_SIZE;
}
```

biến ***count*** được chia sẻ giữa producer và consumer

Bounded buffer (tt)

- Các lệnh tăng, giảm biến count tương đương trong ngôn ngữ máy là:

- (Producer) count++:
  - $register_1$ = count
  - $register_1$ = $register_1$ + 1
  - count = $register_1$

- (Consumer) count--:
  - $register_2$ = count
  - $register_2$ = $register_2$ - 1
  - count = $register_2$

- Trong đó, các $register_i$ là các thanh ghi của CPU.

Bounded buffer (tt)

- Mã máy của các lệnh tăng và giảm biến count có thể bị thực thi xen kẽ
- Giả sử count đang bằng 5. Chuỗi thực thi sau có thể xảy ra:

0:	producer	$register_1$:= count	{$register_1$ = 5}
1:	producer	$register_1$:= $register_1$ + 1	{$register_1$ = 6}
2:	consumer	$register_2$:= count	{$register_2$ = 5}
3:	consumer	$register_2$:= $register_2$ - 1	{$register_2$ = 4}
4:	producer	count := $register_1$	**{count = 6}**
5:	consumer	count := $register_2$	**{count = 4}**

Các lệnh count++, count-- phải là *đơn nguyên* (atomic), nghĩa là thực hiện như một lệnh đơn, không bị ngắt nửa chừng.

Bounded buffer (tt)

- *Race condition*: nhiều process truy xuất và thao tác đồng thời lên dữ liệu chia sẻ (như biến count)
  - Kết quả cuối cùng của việc truy xuất đồng thời này phụ thuộc thứ tự thực thi của các lệnh thao tác dữ liệu.

- Để dữ liệu chia sẻ được nhất quán, cần bảo đảm sao cho tại mỗi thời điểm chỉ có một process được thao tác lên dữ liệu chia sẻ. Do đó, cần có cơ chế đồng bộ hoạt động của các process này.

Vấn đề Critical Section

- Giả sử có *n* process cùng truy xuất đồng thời dữ liệu chia sẻ
- Cấu trúc của mỗi process Pi- Mỗi process có đoạn code như sau :

```
Do {
    entry section     /* vào critical section */
    critical section  /* truy xuất dữ liệu chia xẻ */
exit  section     /* rời critical section */
    remainder section  /* làm những việc khác */
} While (1)
```

- Trong mỗi process có những đoạn code có chứa các thao tác lên dữ liệu chia sẻ. Đoạn code này được gọi là *vùng tranh chấp* (critical section, *CS*).

Vấn đề Critical Section

- Vấn đề Critical Section: phải bảo đảm sự *loại trừ tương hỗ* (MUTual EXclusion, mutex), tức là khi một process đang thực thi trong vùng tranh chấp, không có process nào khác đồng thời thực thi các lệnh trong vùng tranh chấp.

Yêu cầu của lời giải cho Critical Section Problem

- *Lời giải* phải thỏa bốn tính chất:

  (1) Độc quyền truy xuất (*Mutual exclusion)*: Khi một process P đang thực thi trong vùng tranh chấp (CS) của nó thì không có process Q nào khác đang thực thi trong CS của Q.

  (2) Progress: Một tiến trình tạm dừng bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng và việc lựa chọn P nào vào CS phải có hạn định

- (3) Chờ đợi giới hạn (*Bounded waiting)*: Mỗi process chỉ phải chờ để được vào vùng tranh chấp trong một khoảng thời gian có hạn định nào đó. Không xảy ra tình trạng *đói tài nguyên* (starvation).

  (4)Không có giả thiết nào đặt ra cho sự liên hệ về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý trong hệ thống

Phân loại giải pháp

- Nhóm giải pháp Busy Waiting
  - Sử dụng các biến cờ hiệu
  - Sử dụng việc kiểm tra luân phiên
  - Giải pháp của Peterson
  - Cấm ngắt
  - Chỉ thị TSL
- Nhóm giải pháp Sleep & Wakeup
  - Semaphore
  - Monitor
  - Message

Các giải pháp "Busy waiting"

While (chưa có quyền) donothing() ;

CS;

Từ bỏ quyền sử dụng CS

- **Tiếp tục tiêu thụ CPU trong khi chờ đợi vào miền găng**
- **Khơng địi hỏi sự trợ giúp của Hệ điều hành**

Các giải pháp "Sleep & Wake up"

if (chưa có quyền) Sleep() ;

CS;

Wakeup(somebody);

- **Từ bỏ CPU khi chưa được vào miền găng**
- **Cần được Hệ điều hành hỗ trợ**

Các giải pháp "Busy waiting"

Giải thuật 1

- Biến chia sẻ
  - **int turn**; /* khởi đầu **turn = 0** */
  - nếu **turn** = i thì P_i được phép vào critical section, với i = 0 hay 1
- Process P_i

```
do {
        while (turn != i);
            critical section
        turn = j;
            remainder section
} while (1);
```

- Thoả mãn mutual exclusion (1)
- Nhưng không thoả mãn yêu cầu về progress (2) và bounded waiting (3) vì tính chất strict alternation của giải thuật

Giải thuật 1 (tt)

```
Process P0:
do
   while (turn != 0);
       critical section
   turn := 1;
       remainder section
while (1);
```

```
Process P1:
do
   while (turn != 1);
       critical section
   turn := 0;
       remainder section
while (1);
```

Ví dụ:
P0 có RS (remainder section) rất lớn còn P1 có RS nhỏ???

Giải thuật 2

- Biến chia sẻ
  - **boolean flag[2]**; /* khởi đầu **flag[0] = flag[1] = false** */
  - Nếu **flag[** i **] = true** thì P_i "sẵn sàng" vào critical section.
- Process P_i

```
do {
        flag[ i ] = true;       /* Pi "sẵn sàng" vào CS */
        while ( flag[ j ] );    /* Pi "nhường" Pj       */
            critical section
        flag[ i ] = false;
            remainder section
} while (1);
```

- Bảo đảm được mutual exclusion. Chứng minh?
- Không thỏa mãn progress. Vì sao?

Giải thuật 3 (Peterson)

- Biến chia sẻ: kết hợp cả giải thuật 1 và 2
- Process P_i , với i = 0 hay 1

```
do {
        flag[ i ] = true;        /*  Process i sẵn sàng  */
        turn = j;                /*  Nhường process j    */
        while (flag[ j ] and turn == j);
                critical section
        flag[ i ] = false;
                remainder section
} while (1);
```

- Thoả mãn được cả 3 yêu cầu (chứng minh?) [redacted]ải quyết bài toán critical section cho 2 process.

Giải thuật Peterson-2 process

Process P_0

```
do {
        /*  0 wants in  */
       flag[0] = true;
       /*  0 gives a chance to 1  */
       turn = 1;
       while (flag[1] && turn == 1);
          critical section;
        /*  0 no longer wants in  */
       flag[0] = false;
          remainder section;
} while(1);
```

Process P_1

```
do {
       /*  1 wants in  */
       flag[1] = true;
       /*  1 gives a chance to 0  */
       turn = 0;
       while (flag[0] && turn == 0);
          critical section;
        /*  1 no longer wants in  */
       flag[1] = false;
          remainder section;
} while(1);
```

Giải thuật 3: Tính đúng đắn

- Giải thuật 3 thỏa mutual exclusion, progress, và bounded waiting
- Mutual exclusion được bảo đảm bởi vì
  - P0 và P1 đều ở trong CS nếu và chỉ nếu flag[0] = flag[1] = true và turn = i cho mỗi Pi (không thể xảy ra)
- Chứng minh thỏa yêu cầu về progress và bounded waiting
  - Pi không thể vào CS nếu và chỉ nếu bị kẹt tại vòng lặp while() với điều kiện flag[j] = true và turn = j .
  - Nếu Pj không muốn vào CS thì flag[j] = false và do đó Pi có thể vào CS.

Giải thuật 3: Tính đúng đắn (tt)

- Nếu Pj đã bật flag[j] = true và đang chờ tại while() thì có chỉ hai trường hợp là turn = i hoặc turn = j
- Nếu turn = i thì Pi vào CS. Nếu turn = j thì Pj vào CS nhưng sẽ bật flag[j] = false khi thoát ra [redacted]ho phép Pi vào CS
- Nhưng nếu Pj có đủ thời gian bật flag[j] = true thì Pj cũng phải gán turn = i
- Vì Pi không thay đổi trị của biến turn khi đang kẹt trong vòng lặp while(), Pi sẽ chờ để vào CS nhiều nhất là sau một lần Pj vào CS (bounded waiting)

Giải thuật bakery: *n* process

- Trước khi vào CS, process Pi nhận một con số. Process nào giữ con số nhỏ nhất thì được vào CS
- Trường hợp Pi và Pj cùng nhận được một chỉ số:
  - Nếu i < j thì Pi được vào trước. (Đối xứng)
- Khi ra khỏi CS, Pi đặt lại số của mình bằng 0
- Cơ chế cấp số cho các process thường tạo các số theo cơ chế tăng dần, ví dụ 1, 2, 3, 3, 3, 3, 4, 5,…

- Kí hiệu
  - (a,b) < (c,d) nếu a < c hoặc if a = c và b < d
  - max(a_0,…,a_k) là con số b sao cho b [redacted]$_i$ với mọi i = 0,…, k

Giải thuật bakery: *n* process (tt)

```
/*  shared variable  */
boolean     choosing[ n ];  /*  initially, choosing[ i ] = false  */
int             num[ n ];                   /*  initially, num[ i ] = 0             */

do {
            choosing[ i ] = true;
            num[ i ]       = max(num[0], num[1],…, num[n ]) + 1;
            choosing[ i ] = false;
            for (j = 0; j < n; j++)
            {
                while (choosing[ j ]);
                while ((num[ j ] != 0) && (num[ j ], j) < (num[ i ], i));
            }
               critical section
            num[ i ] = 0;
               remainder section
} while (1);
```

Từ software đến hardware

- Khuyết điểm của các giải pháp software
  - Các process khi yêu cầu được vào vùng tranh chấp đều phải liên tục kiểm tra điều kiện (busy waiting), tốn nhiều thời gian xử lý của CPU
  - Nếu thời gian xử lý trong vùng tranh chấp lớn, một giải pháp hiệu quả nên có cơ chế block các process cần đợi.

- Các giải pháp phần cứng (hardware)
  - Cấm ngắt (disable interrupts)
  - Dùng các lệnh đặc biệt

Cấm ngắt

- Trong hệ thống uniprocessor: mutual exclusion được bảo đảm.
  - Nhưng nếu system clock được cập nhật do interrupt thì sao?
- Trên hệ thống multiprocessor: mutual exclusion không được đảm bảo
  - Chỉ cấm ngắt tại CPU thực thi lệnh disable_interrupts
  - Các CPU khác vẫn có thể truy cập bộ nhớ chia sẻ

```
Process Pi:
do {
        disable_interrupts();
            critical section
        enable_interrupts();
            remainder section
} while (1);
```

Lệnh *TestAndSet*

- Đọc và ghi một biến trong một thao tác *atomic* (không chia cắt được).

```
boolean TestAndSet(boolean &target)
{
    boolean rv = target;
    target = true;
    return rv;
}
```

- Shared data:
  boolean lock = false;

- Process P_i :

```
do {
    while (TestAndSet(lock));
        critical section
    lock = false;
        remainder section
} while (1);
```

Lệnh TestAndSet (tt)

- Mutual exclusion được bảo đảm: nếu P_i vào CS, các process P_j khác đều đang busy waiting

- Khi P_i ra khỏi CS, quá trình chọn lựa process P_j vào CS kế tiếp là tùy ý [redacted]hông bảo đảm điều kiện bounded waiting. Do đó có thể xảy ra *starvation* (bị bỏ đói)

- Các processor (ví dụ Pentium) thông thường cung cấp một lệnh đơn là Swap(a, b) có tác dụng hoán chuyển nội dung của a và b.
  - Swap(a, b) cũng có ưu nhược điểm như TestAndSet

Swap và mutual exclusion

- Biến chia sẻ **lock** được khởi tạo giá trị false
- Mỗi process P_i có biến cục bộ **key**
- Process P_i nào thấy giá trị **lock = false** thì được vào CS.
  - Process P_i sẽ loại trừ các process P_j khác khi thiết lập **lock = true**

```
void Swap(boolean &a,
                boolean &b) {
   boolean temp = a;
   a = b;
   b = temp;
}
```

- Biến chia sẻ (khởi tạo là **false**)

  bool lock;

  bool key;

- Process P_i

```
do {
        key = true;
        while (key == true)
                Swap(lock, key);
            critical section
        lock = false;
            remainder section
} while (1)
```

Không thỏa mãn bounded waiting

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu (1)

- Cấu trúc dữ liệu dùng chung (khởi tạo là false)

  bool waiting[n];

  bool lock;

- Mutual exclusion: Pi chỉ có thể vào CS nếu và chỉ nếu hoặc waiting[i] = false, hoặc key = false
  - key = false chỉ khi TestAndSet (hay Swap) được thực thi
    - Process đầu tiên thực thi TestAndSet mới có key == false; các process khác đều phải đợi
  - waiting[i] = false chỉ khi process khác rời khỏi CS
    - Chỉ có một waiting[i] có giá trị false
- Progress: chứng minh tương tự như mutual exclusion
- Bounded waiting: waiting in the cyclic order

Giải thuật dùng TestAndSet thoả mãn 3 yêu cầu (2)

```
do {
        waiting[ i ] = true;
        key = true;
        while (waiting[ i ] && key)
            key = TestAndSet(lock);
        waiting[ i ] = false;

            critical section

        j = (i + 1) % n;
        while ( (j != i)  &&  !waiting[ j ] )
            j = (j + 1) % n;
        if (j == i)
            lock = false;
        else
            waiting[ j ] = false;

            remainder section
} while (1)
```

Các giải pháp "Sleep & Wake up"

```
int busy; // =1 nếu CS đang bị chiếm
Int blocked;          // số P đang bị khóa
do
{
    if (busy==1){
                                  blocked = blocked +1;
                                  sleep();
                    }
    else busy =1;
          CS;
    busy = 0;
    if(blocked !=0){
                                  wakeup(process);
                                  blocked = blocked -1;
                        }
          RS;
} while(1);
```

Trường hợp:
-A vào CS
-B kích hoạt và tăng blocked
-A kích hoạt lại
-B kích hoạt lại
-?????

Semaphore

- Là công cụ đồng bộ cung cấp bởi OS mà không đòi hỏi busy waiting
- *Semaphore* S là một biến số nguyên.
- Ngoài thao tác khởi động biến thì chỉ có thể được truy xuất qua hai tác vụ có tính đơn nguyên (atomic) và loại trừ (mutual exclusion)
  - wait(S) hay còn gọi là P(S): giảm giá trị semaphore (S=S-1) . Kế đó nếu giá trị này âm thì process thực hiện lệnh wait() bị blocked.
  - signal(S) hay còn gọi là V(S): tăng giá trị semaphore (S=S+1) . Kế đó nếu giá trị này không dương, một process đang blocked bởi một lệnh wait() sẽ được hồi phục để thực thi.
- Tránh busy waiting: khi phải đợi thì process sẽ được đặt vào một blocked queue, trong đó chứa các process đang chờ đợi cùng một sự kiện.

Semaphore

- P(S) hay wait(S) sử dụng để giành tài nguyên và giảm biến đếm S=S-1
- V(S) hay signal(S) sẽ giải phóng tài nguyên và tăng biến đếm S= S+1
- Nếu P được thực hiện trên biến đếm <= 0 , tiến trình phải đợi V hay chờ đợi sự giải phóng tài nguyên

Hiện thực semaphore

- Định nghĩa semaphore là một record

```
typedef  struct {
        int  value;
        struct  process *L;    /* process queue */
} semaphore;
```

- Giả sử hệ điều hành cung cấp hai tác vụ (system call):
  - **block()**: tạm treo process nào thực thi lệnh này
  - **wakeup(**P**)**: hồi phục quá trình thực thi của process P đang blocked

Hiện thực semaphore (tt)

- Các tác vụ semaphore được hiện thực như sau

```
void wait(semaphore S) {
        S.value--;
        if (S.value < 0) {
                add this process to S.L;
                block();
          }
}

void signal(semaphore S) {
        S.value++;
        if (S.value <= 0) {
                remove a process P from S.L;
                wakeup(P);
          }
}
```

Hiện thực semaphore (tt)

- Khi một process phải chờ trên semaphore S, nó sẽ bị blocked và được đặt trong hàng đợi semaphore
  - Hàng đợi này là danh sách liên kết các PCB
- Tác vụ signal() thường sử dụng cơ chế FIFO khi chọn một process từ hàng đợi và đưa vào hàng đợi ready
- block() và wakeup() thay đổi trạng thái của process
  - block: chuyển từ running sang waiting
  - wakeup: chuyển từ waiting sang ready

Ví dụ sử dụng semaphore 1 : Hiện thực mutex với semaphore

- Dùng cho *n* process
- Khởi tạo S.value = 1
- Chỉ duy nhất một process được vào CS (mutual exclusion)
- Để cho phép *k* process vào CS, khởi tạo S.value = k

- Shared data:

```
semaphore  mutex;
/* initially mutex.value = 1 */
```

- Process *Pi:*

```
do {
        wait(mutex);
          critical section
        signal(mutex);
          remainder section
} while (1);
```

Ví dụ sử dụng semaphore 2 :Đồng bộ process bằng semaphore

- Hai process: P1 và P2
- Yêu cầu: lệnh S1 trong P1 cần được thực thi trước lệnh S2 trong P2
- Định nghĩa semaphore synch để đồng bộ
- Khởi động semaphore:

  synch.value = 0

- Để đồng bộ hoạt động theo yêu cầu, P1 phải định nghĩa như sau:

  S1;

  signal(synch);

- Và P2 định nghĩa như sau:

  wait(synch);

  S2;

Nhận xét

- Khi S.value [redacted]: số process có thể thực thi wait(S) mà không bị blocked = S.value

- Khi S.value < 0: số process đang đợi trên S là [redacted].value[redacted]

- Atomic và mutual exclusion: không được xảy ra trường hợp 2 process cùng đang ở trong thân lệnh wait(S) và signal(S) (cùng semaphore S) tại một thời điểm (ngay cả với hệ thống multiprocessor)

  [redacted] đó, đoạn mã định nghĩa các lệnh wait(S) và signal(S) cũng chính là vùng tranh chấp

Nhận xét (tt)

- Vùng tranh chấp của các tác vụ wait(S) và signal(S) thông thường rất nhỏ: khoảng 10 lệnh.

- Giải pháp cho vùng tranh chấp wait(S) và signal(S)
  - Uniprocessor: có thể dùng cơ chế cấm ngắt (disable interrupt). Nhưng phương pháp này không làm việc trên hệ thống multiprocessor.
  - Multiprocessor: có thể dùng các giải pháp software (như giải thuật Dekker, Peterson) hoặc giải pháp hardware (TestAndSet, Swap).
    - Vì CS rất nhỏ nên chi phí cho busy waiting sẽ rất thấp.

Deadlock và starvation

- *Deadlock*: hai hay nhiều process đang chờ đợi vô hạn định một sự kiện không bao giờ xảy ra (vd: sự kiện do một trong các process đang đợi tạo ra).
- Gọi S và Q là hai biến semaphore được khởi tạo = 1

```
P0                          P1

wait(S);            wait(Q);
wait(Q);            wait(S);
   ⋮                   ⋮
signal(S);          signal(Q);
signal(Q);          signal(S);
```

P0 thực thi wait(S), rồi P1 thực thi wait(Q), rồi P0 thực thi wait(Q) bị blocked, P1 thực thi wait(S) bị blocked.

- *Starvation* (indefinite blocking) Một tiến trình có thể không bao giờ được lấy ra khỏi hàng đợi mà nó bị treo trong hàng đợi đó.

Các loại semaphore

- *Counting semaphore*: một số nguyên có giá trị không hạn chế.
- *Binary semaphore*: có trị là 0 hay 1. Binary semaphore rất dễ hiện thực.
- Có thể hiện thực counting semaphore bằng binary semaphore.

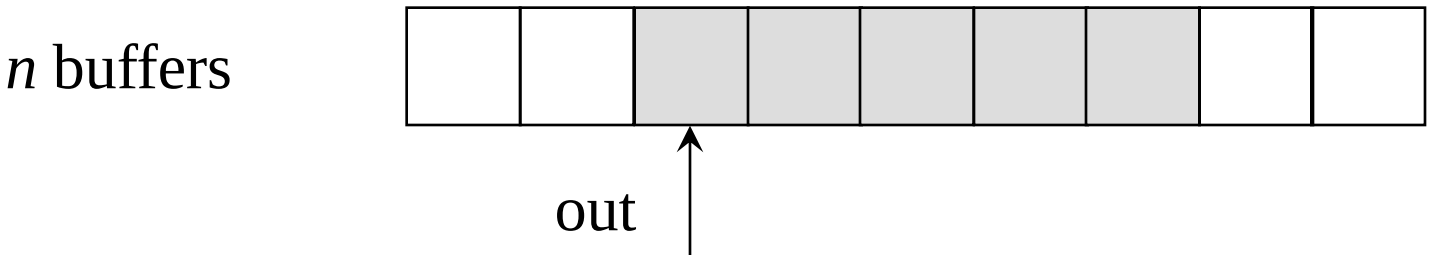
Các bài toán đồng bộ (kinh điển)

- Bounded Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Các bài toán đồng bộ

- Bài toán bounded buffer
  - Dữ liệu chia sẻ:

    semaphore full, empty, mutex;
  - Khởi tạo:
    - full = 0;/* số buffers đầy */
    - empty = n; /* số buffers trống */
    - mutex = 1;



Bounded buffer

producer

```
do {
        …
        nextp = new_item();
        …
        wait(empty);
        wait(mutex);
        …
        insert_to_buffer(nextp);
        …
        signal(mutex);
        signal(full);
} while (1);
```

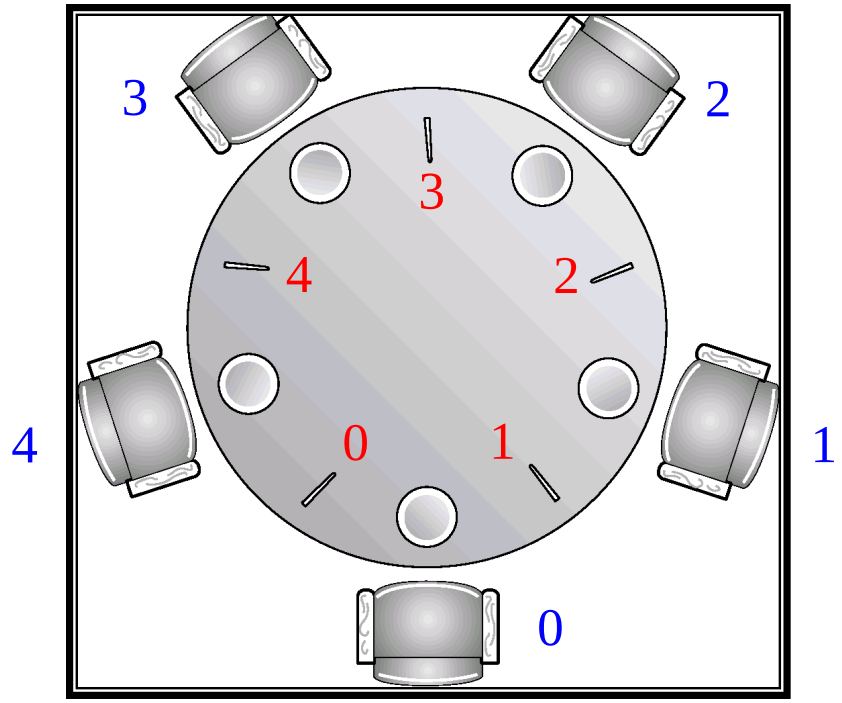
consumer

```
do {
        wait(full)
        wait(mutex);
        …
        nextc = get_buffer_item(out);
        …
        signal(mutex);
        signal(empty);
        …
        consume_item(nextc);
        …
} while (1);
```

Bài toán "Dining Philosophers" (1)

- 5 triết gia ngồi ăn và suy nghĩ
- Mỗi người cần 2 chiếc đũa (chopstick) để ăn
- Trên bàn chỉ có 5 đũa

- Bài toán này minh họa sự khó khăn trong việc phân phối tài nguyên giữa các process sao cho không xảy ra deadlock và starvation



- Dữ liệu chia sẻ:
  semaphore chopstick[5];

- Khởi đầu các biến đều là 1

Bài toán "Dining Philosophers" (2)

Triết gia thứ *i:*

```
do {
    wait(chopstick [ i ])
    wait(chopstick [ (i + 1) % 5 ])
     …
    eat
     …
    signal(chopstick [ i ]);
    signal(chopstick [ (i + 1) % 5 ]);
     …
    think
     …
} while (1);
```

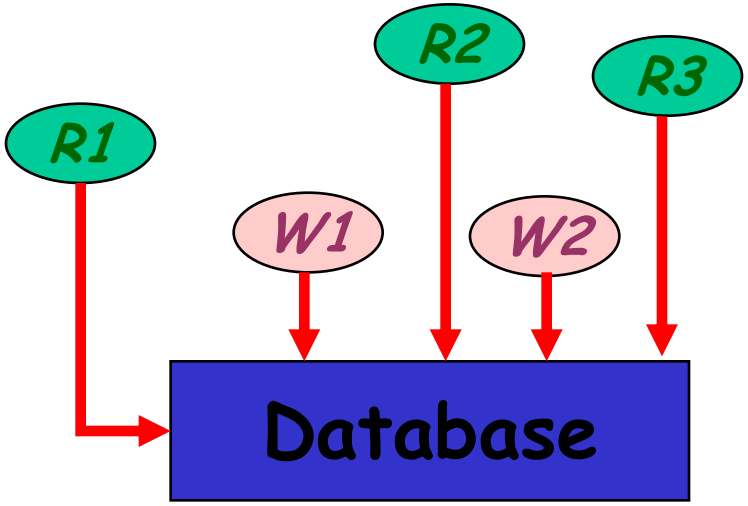
Bài toán "Dining Philosophers" (3)

- Giải pháp trên có thể gây ra deadlock
  - Khi tất cả triết gia đói bụng cùng lúc và đồng thời cầm chiếc đũa bên tay trái [redacted]deadlock
- Một số giải pháp khác giải quyết được deadlock
  - Cho phép nhiều nhất 4 triết gia ngồi vào cùng một lúc
  - Cho phép triết gia cầm các đũa chỉ khi cả hai chiếc đũa đều sẵn sàng (nghĩa là tác vụ cầm các đũa phải xảy ra trong CS)
  - Triết gia ngồi ở vị trí lẻ cầm đũa bên trái trước, sau đó mới đến đũa bên phải, trong khi đó triết gia ở vị trí chẵn cầm đũa bên phải trước, sau đó mới đến đũa bên trái
- Starvation?

Bài toán Readers-Writers (1)

Readers - Writers

- **W không được cập nhật dữ liệu khi có một R đang truy xuất CSDL .**
- **Tại một thời điểm , chỉ cho phép một W được sửa đổi nội dung CSDL.**



Bài toán Readers-Writers (2)

- Bộ đọc trước bộ ghi (first reader-writer)
- Dữ liệu chia sẻ

```
semaphore  mutex = 1;
semaphore  wrt   = 1;
int        readcount  = 0;
```

- Writer process

```
wait(wrt);
...
writing is performed
...
signal(wrt);
```

- Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(wrt);
signal(mutex);
...
reading is performed
...
wait(mutex);
readcount--;
if (readcount == 0)
        signal(wrt);
signal(mutex);
```

Bài toán Readers-Writers (3)

- mutex: "bảo vệ" biến readcount
- wrt
  - Bảo đảm mutual exclusion đối với các writer
  - Được sử dụng bởi reader đầu tiên hoặc cuối cùng vào hay ra khỏi vùng tranh chấp.
- Nếu một writer đang ở trong CS và có *n* reader đang đợi thì một reader được xếp trong hàng đợi của wrt và *n* - 1 reader kia trong hàng đợi của mutex
- Khi writer thực thi signal(wrt), hệ thống có thể phục hồi thực thi của một trong các reader đang đợi hoặc writer đang đợi.

Các vấn đề với semaphore

- Semaphore cung cấp một công cụ mạnh mẽ để bảo đảm mutual exclusion và phối hợp đồng bộ các process
- Tuy nhiên, nếu các tác vụ wait(S) và signal(S) nằm rải rác ở rất nhiều processes [redacted]hó nắm bắt được hiệu ứng của các tác vụ này. Nếu không sử dụng đúng [redacted]ể xảy ra tình trạng deadlock hoặc starvation.
- Một process bị "die" có thể kéo theo các process khác cùng sử dụng biến semaphore.

```
signal(mutex)
…
critical section
…
wait(mutex)
```

```
wait(mutex)
…
critical section
…
wait(mutex)
```

```
signal(mutex)
…
critical section
…
signal(mutex)
```

Critical Region (CR)

- Là một cấu trúc ngôn ngữ cấp cao (high-level language construct, được dịch sang mã máy bởi một compiler), thuận tiện hơn cho người lập trình.

- Một biến chia sẻ *v* kiểu dữ liệu *T*, khai báo như sau

  v: shared T;

- Biến chia sẻ *v* chỉ có thể được truy xuất qua phát biểu sau

  region v when B do S; /* B là một biểu thức Boolean */

- Ý nghĩa: trong khi S được thực thi, không có quá trình khác có thể truy xuất biến v.

CR và bài toán bounded buffer

Dữ liệu chia sẻ:

```
struct  buffer
{
        int  pool[n];
        int  count,
             in,
             out;
}
```

Producer

```
region buffer when (count < n) {
        pool[in] = nextp;
        in = (in + 1) % n;
        count++;
}
```

Consumer

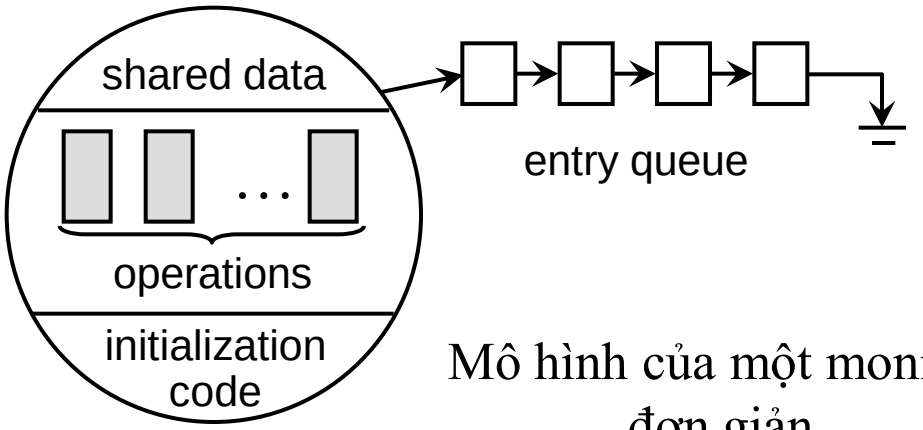
```
region buffer when (count > 0){
        nextc = pool[out];
        out = (out + 1) % n;
        count--;
}
```

Monitor (1)

- Cũng là một cấu trúc ngôn ngữ cấp cao tương tự CR, có chức năng như semaphore nhưng dễ điều khiển hơn
- Xuất hiện trong nhiều ngôn ngữ lập trình đồng thời như
  - Concurrent Pascal, Modula-3, Java,…
- Có thể hiện thực bằng semaphore

Monitor (2)

- Là một module phần mềm, bao gồm
  - Một hoặc nhiều *thủ tục* (procedure)
  - Một đoạn *code khởi tạo* (initialization code)
  - Các *biến dữ liệu cục bộ* (local data variable)



Mô hình của một monitor đơn giản

- Đặc tính của monitor
  - Local variable chỉ có thể truy xuất bởi các thủ tục của monitor
  - Process "vào monitor" bằng cách gọi một trong các thủ tục đó
  - Chỉ có một process có thể vào monitor tại một thời điểm [redacted] mutual exclusion được bảo đảm

Cấu trúc của monitor

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (…) {
                . . .
        }
        procedure body P2 (…) {
                . . .
        }
        procedure body Pn (…) {
                . . .
        }
        {
                initialization code
        }
}
```

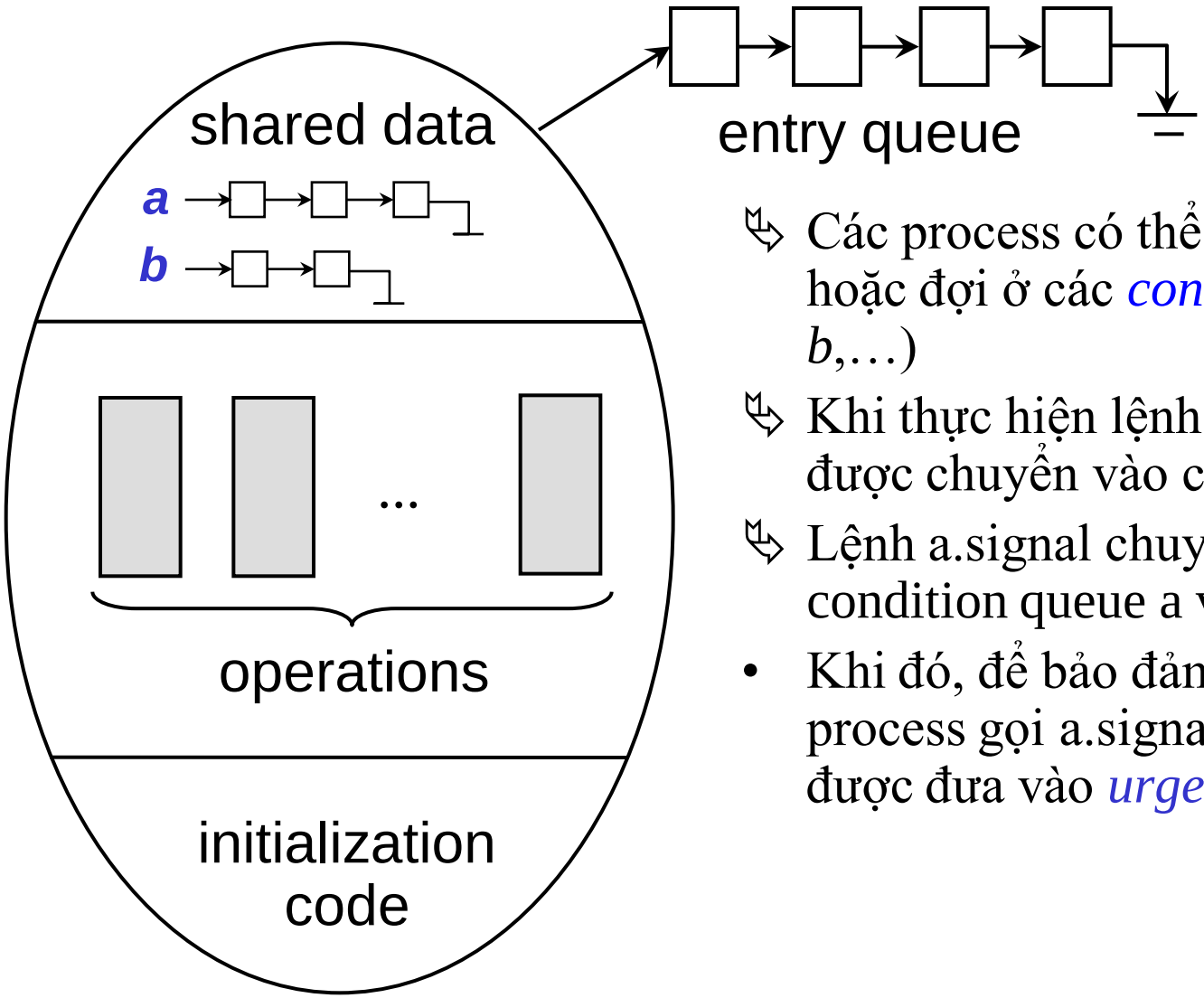
Condition variable

- Nhằm cho phép một process đợi "trong monitor", phải khai báo *biến điều kiện* (condition variable)

  condition a, b;

- Các biến điều kiện đều cục bộ và chỉ được truy cập bên trong monitor.
- Chỉ có thể thao tác lên biến điều kiện bằng hai thủ tục:
  - a.**wait**: process gọi tác vụ này sẽ bị "block trên biến điều kiện" a
    - process này chỉ có thể tiếp tục thực thi khi có process khác thực hiện tác vụ a.**signal**
  - a.**signal**: phục hồi quá trình thực thi của process bị block trên biến điều kiện a.
    - Nếu có nhiều process: chỉ chọn một
    - Nếu không có process: không có tác dụng

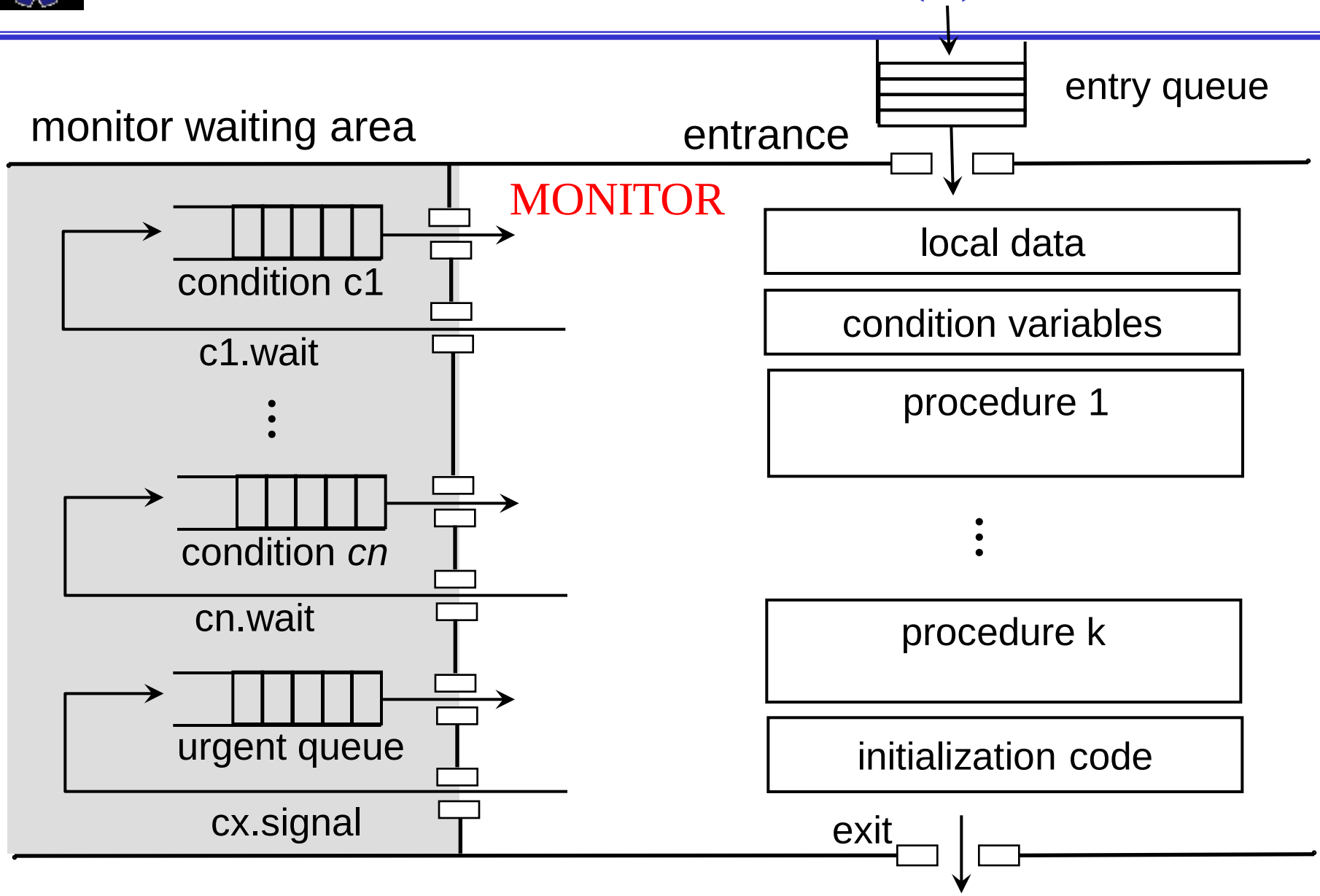
Monitor có condition variable



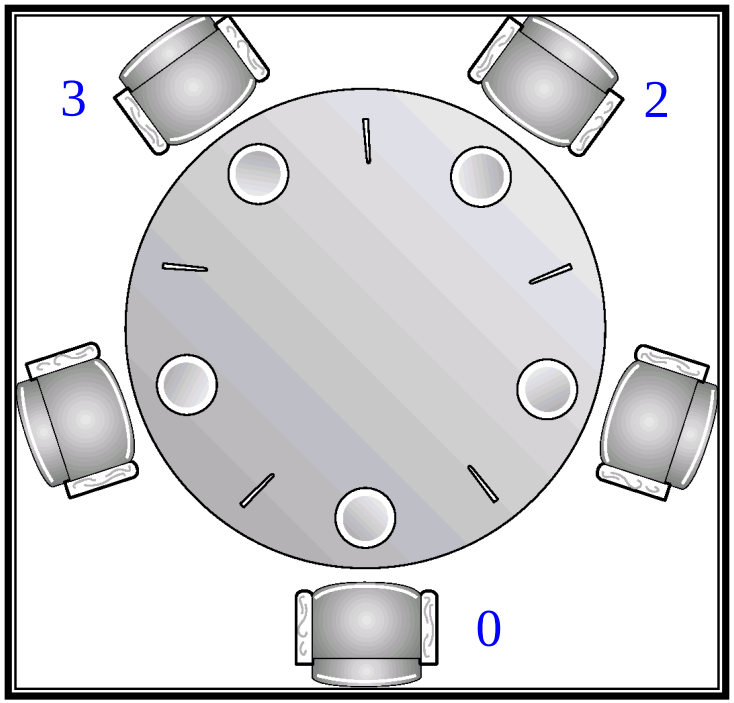
- Các process có thể đợi ở *entry queue* hoặc đợi ở các *condition queue* (*a*, *b*,…)
- Khi thực hiện lệnh a.wait, process sẽ được chuyển vào condition queue a
- Lệnh a.signal chuyển một process từ condition queue a vào monitor

• Khi đó, để bảo đảm mutual exclusion, process gọi a.signal sẽ bị blocked và được đưa vào *urgent queue*

Monitor có condition variable (tt)



Monitor và dining philosophers




```
monitor dp
{
        enum {thinking, hungry, eating} state[5];
        condition self[5];
```

Dining philosophers (tt)

```
void pickup(int i) {
        state[ i ] = hungry;
        test[ i ];
        if (state[ i ] != eating)
                self[ i ].wait();
}
void putdown(int i) {
        state[ i ] = thinking;
        // test left and right neighbors
        test((i + 4) % 5);    // left neighbor
        test((i + 1) % 5);    // right …
}
```

Dining philosophers (tt)

```
void test(int i) {
        if ( (state[(i + 4) % 5] != eating) &&
             (state[ i ] == hungry) &&
             (state[(i + 1) % 5] != eating) ) {
                 state[ i ] = eating;
                 self[ i ].signal();
        }
void init() {
        for (int i = 0; i < 5; i++)
            state[ i ] = thinking;
        }
}
```

Dining philosophers (tt)

- Trước khi ăn, mỗi triết gia phải gọi hàm pickup(), ăn xong rồi thì phải gọi hàm putdown()

```
dp.pickup(i);
ăn
dp.putdown(i);
```

- Giải thuật không deadlock nhưng có thể gây starvation.

Chương 6 : Tắc nghẽn(Deadlock)

- Mô hình hệ thống
- Định nghĩa
- Điều kiện cần của deadlock
- Resource Allocation Graph (RAG)
- Phương pháp giải quyết deadlock
- Deadlock prevention
- Deadlock avoidance
- Deadlock detection
- Deadlock recovery
- Phương pháp kết hợp để giải quyết Deadlock

Vấn đề deadlock trong hệ thống

- *Tình huống*: một tập các process bị blocked, mỗi process giữ tài nguyên và đang chờ tài nguyên mà process khác trong tập đang giữ.

- Ví dụ 1
  - Giả sử hệ thống có 2 file trên đĩa.
  - P1 và P2 mỗi process đang mở một file và yêu cầu mở file kia.

- Ví dụ 2
  - Semaphore A và B, khởi tạo bằng 1

P0	P1
wait(A);	wait(B);
wait(B);	wait(A);

Mô hình hóa hệ thống

- Hệ thống gồm các loại *tài nguyên*, kí hiệu R_1, R_2,…, R_m , bao gồm:
  - CPU cycle, không gian bộ nhớ, thiết bị I/O, file, semaphore,…
- Mỗi loại tài nguyên R_i có W_i thực thể (instance).

- Giả sử tài nguyên tái sử dụng theo kỳ (Serially Reusable Resources)
  - *Yêu cầu* (request): process phải chờ nếu yêu cầu không được đáp ứng ngay
  - *Sử dụng* (use): process sử dụng tài nguyên
  - *Hoàn trả* (release): process hoàn trả tài nguyên

- Các tác vụ yêu cầu (request) và hoàn trả (release) đều là system call. Ví dụ
  - request/release device
  - open/close file
  - allocate/free memory
  - wait/signal

Định nghĩa

- Một tiến trình gọi là *deadlocked* nếu nó đang đợi một sự kiện mà sẽ không bao giờ sảy ra.

  Thông thường, có nhiều hơn một tiến trình bị liên quan trong một deadlock.

- Một tiến trình gọi là trì hoãn vô hạn định (*indefinitely postponed)* nếu nó bị trì hoãn một khoảng thời gian dài lặp đi lặp lại trong khi hệ thống đáp ứng cho những tiến trình khác .
  - i.e. Một tiến trình sẵn sàng để xử lý nhưng nó không bao giờ nhận được CPU.

Điều kiện cần để xảy ra deadlock

Bốn điều kiện cần (necessary condition) để xảy ra deadlock

1. *Loại trừ hỗ tương (Mutual exclusion)*: ít nhất một tài nguyên được giữ theo nonsharable mode (ví dụ: printer; ví dụ sharable resource: read-only files).

2. *Giữ và chờ cấp thêm tài nguyên (Hold and wait)*: một process đang giữ ít nhất một tài nguyên và đợi thêm tài nguyên do quá trình khác đang giữ.

Điều kiện cần để xảy ra deadlock (tt)

3. *Không trưng dụng (No preemption)*: (= no resource preemption) tài nguyên không thể bị lấy lại, mà chỉ có thể được trả lại từ process đang giữ tài nguyên đó khi nó muốn.

4. *Chu trình đợi (Circular wait)*: tồn tại một tập $\{P_0,\ldots,P_n\}$ các quá trình đang đợi sao cho

   P_0 đợi một tài nguyên mà P_1 đang giữ

   P_1 đợi một tài nguyên mà P_2 đang giữ

   …

   P_n đợi một tài nguyên mà P_0 đang giữ

Đồ thị cấp phát tài nguyên
Resource Allocation Graph

- *Resource allocation graph* (RAG) là đồ thị có hướng, với tập đỉnh V và tập cạnh E

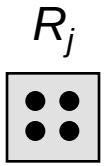
  - Tập đỉnh V gồm 2 loại:
    - $P = \{P_1, P_2, \ldots, P_n\}$ (Tất cả process trong hệ thống)
    - $R = \{R_1, R_2, \ldots, R_m\}$ (Tất cả các loại tài nguyên trong hệ thống)

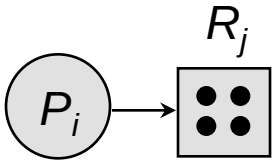
  - Tập cạnh E gồm 2 loại:
    - *Cạnh yêu cầu (Request edge)*: ø $P_i \rightarrow R_j$
    - *Cạnh cấp phát (Assignment edge)*: $R_j \rightarrow P_i$

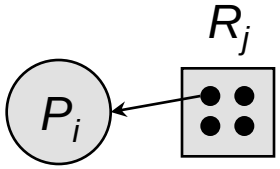
Resource Allocation Graph (tt)

Ký hiệu

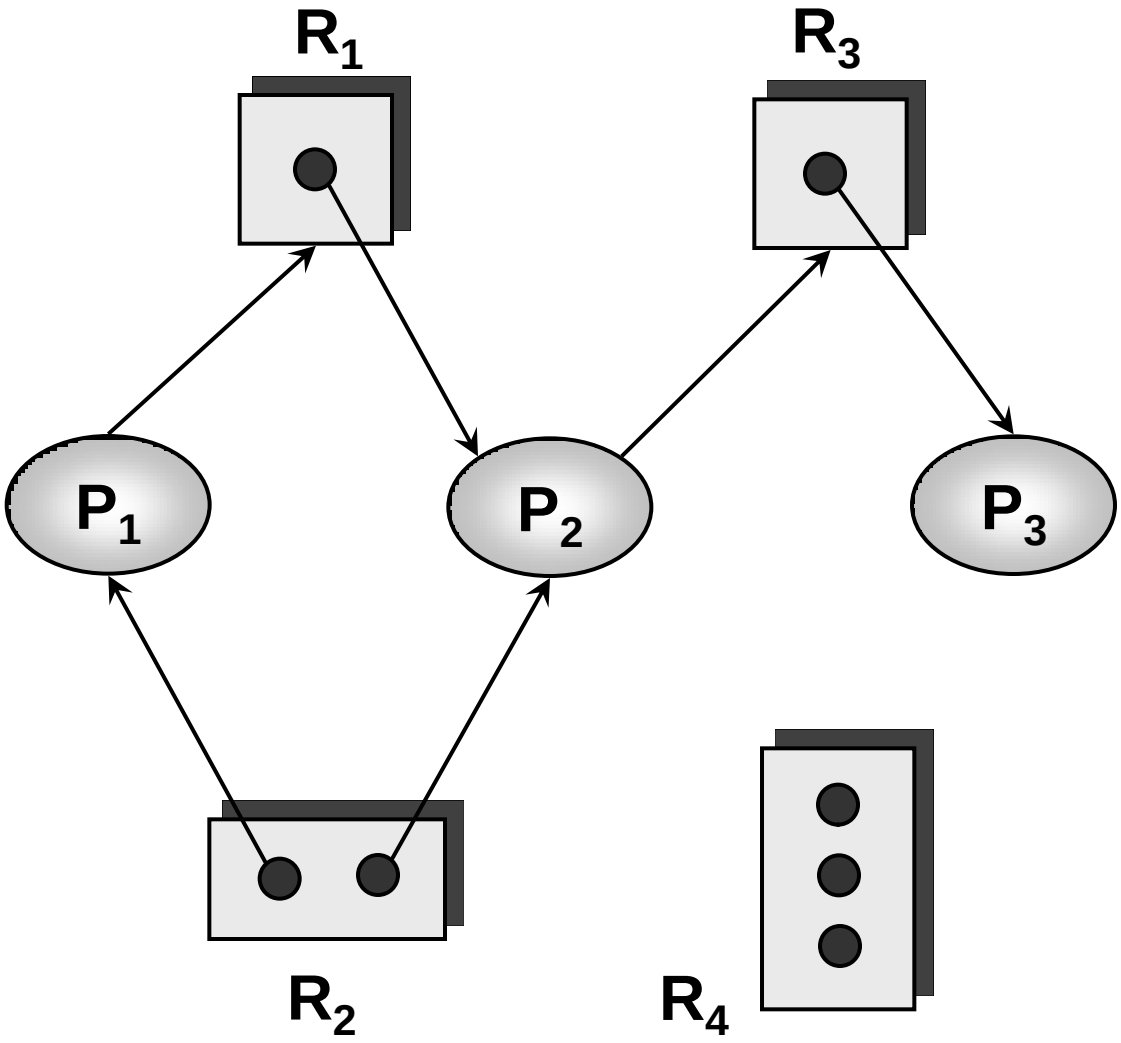
- Process: 

- Loại tài nguyên với 4 thực thể: 

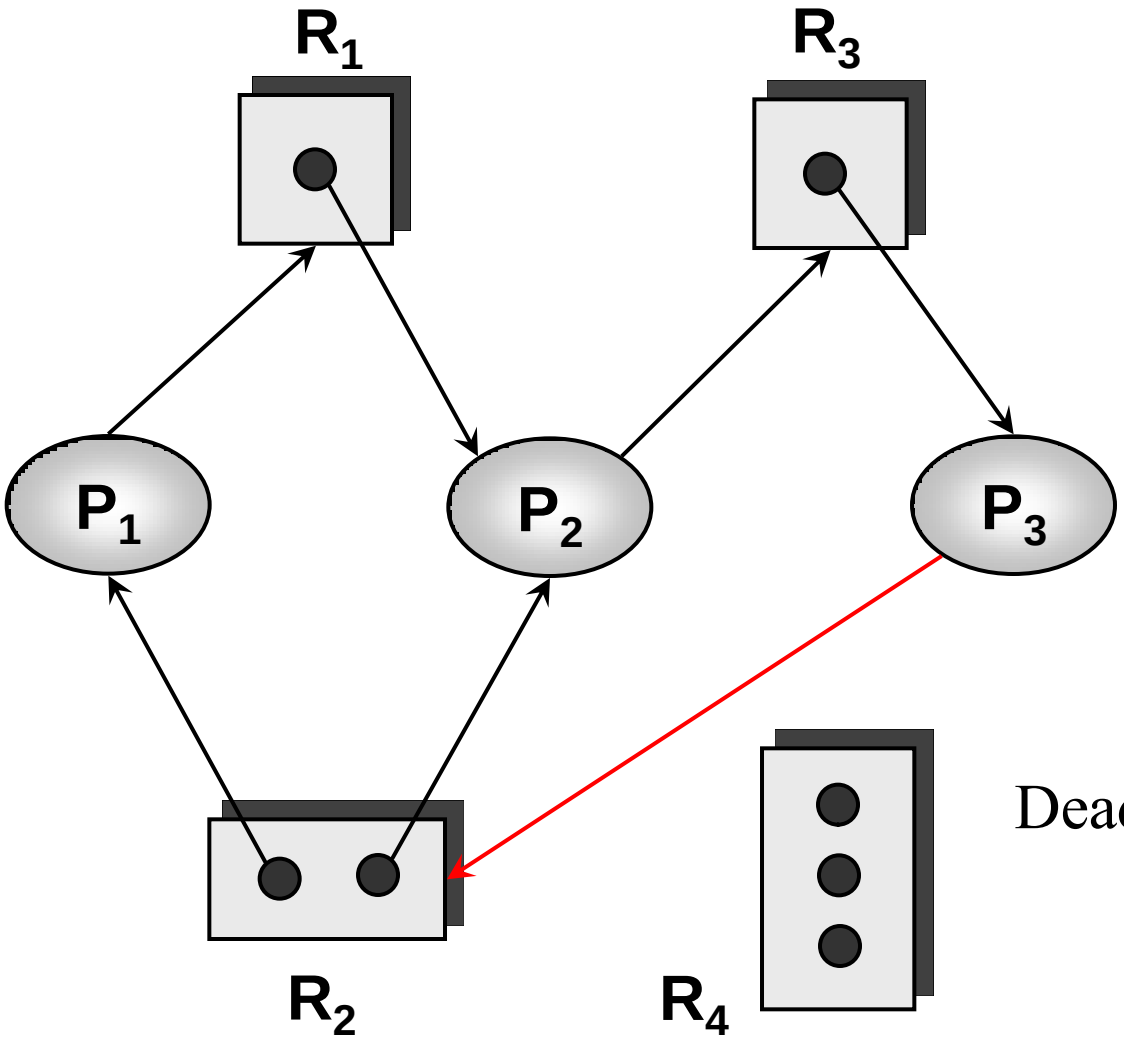
- P_i yêu cầu một thực thể của R_j : 

- P_i đang giữ một thực thể của R_j : 

Ví dụ về RAG



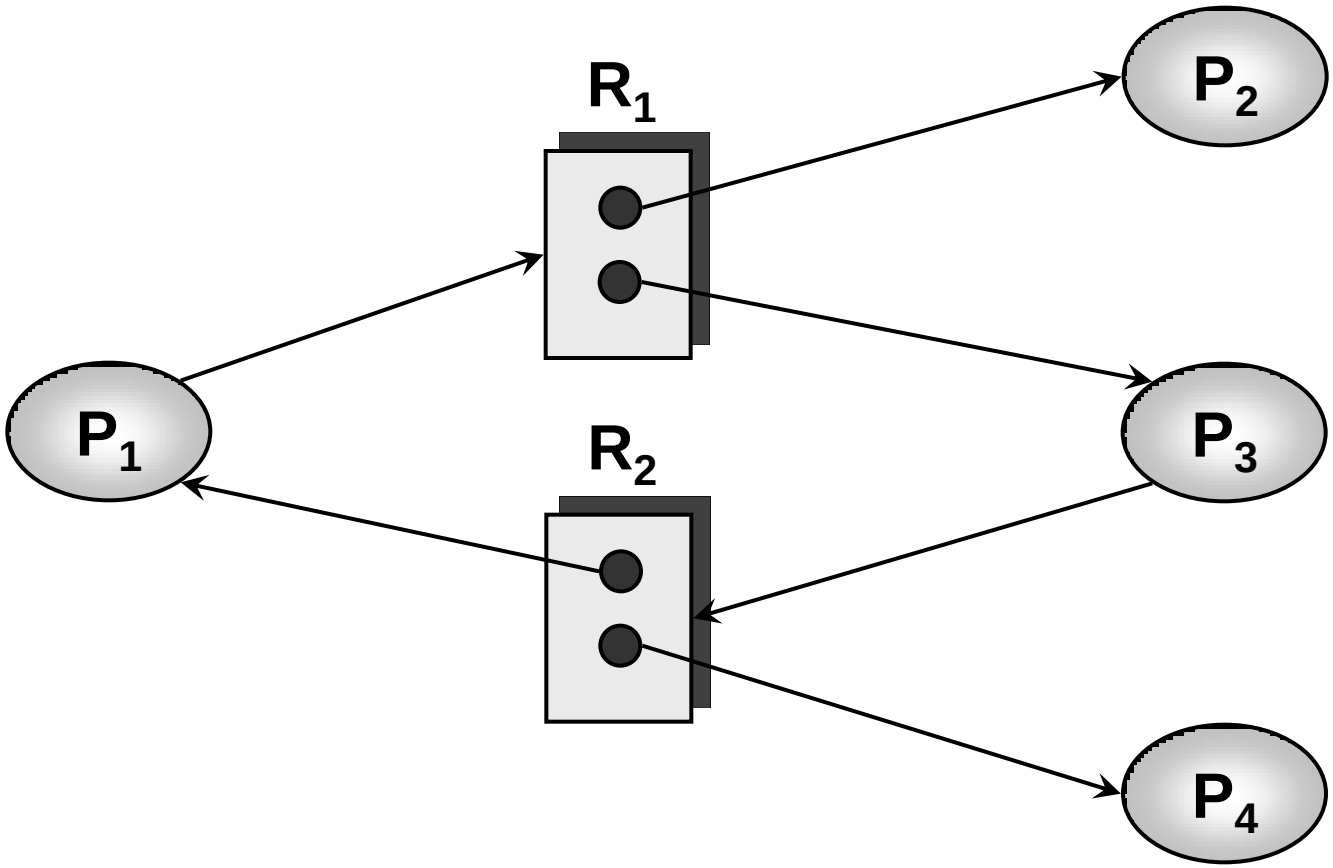
Ví dụ về RAG (tt)



Deadlock xảy ra!

RAG và deadlock

- Ví dụ một RAG chứa chu trình nhưng không xảy ra deadlock: P_4 có thể trả lại instance của R_2.



RAG và deadlock (tt)

- RAG không chứa chu trình (cycle) [redacted]hông có deadlock
- RAG chứa một (hay nhiều) chu trình
  - Nếu mỗi loại tài nguyên chỉ có một thực thể [redacted]eadlock
  - Nếu mỗi loại tài nguyên có nhiều thực thể [redacted] có thể xảy ra deadlock

Các phương pháp giải quyết deadlock (1)

- Ba phương pháp
- 1) Bảo đảm rằng hệ thống không rơi vào tình trạng deadlock bằng cách *ngăn* (preventing) hoặc *tránh* (avoiding) deadlock.
- Khác biệt
  - Ngăn deadlock: không cho phép (ít nhất) một trong 4 điều kiện cần cho deadlock
  - Tránh deadlock: các quá trình cần cung cấp thông tin về tài nguyên nó cần để hệ thống cấp phát tài nguyên một cách thích hợp

Các phương pháp giải quyết deadlock (2)

- 2) Cho phép hệ thống vào trạng thái deadlock, nhưng sau đó phát hiện deadlock và phục hồi hệ thống.

- 3) Bỏ qua mọi vấn đề, xem như deadlock không bao giờ xảy ra trong hệ thống.

  ☺Khá nhiều hệ điều hành sử dụng phương pháp này.

  – Deadlock không được phát hiện, dẫn đến việc giảm hiệu suất của hệ thống. Cuối cùng, hệ thống có thể ngưng hoạt động và phải được khởi động lại.

1. Ngăn deadlock (deadlock prevention)

Ngăn deadlock bằng cách ngăn một trong 4 điều kiện cần của deadlock

1. Ngăn mutual exclusion
   - đối với nonsharable resource (vd: printer): không làm được
   - đối với sharable resource (vd: read-only file): không cần thiết

Ngăn deadlock (tt)

2. Ngăn Hold and Wait

- Cách 1: mỗi process yêu cầu toàn bộ tài nguyên cần thiết một lần. Nếu có đủ tài nguyên thì hệ thống sẽ cấp phát, nếu không đủ tài nguyên thì process phải bị blocked.

- Cách 2: khi yêu cầu tài nguyên, process không được giữ bất kỳ tài nguyên nào. Nếu đang có thì phải trả lại trước khi yêu cầu.

- Ví dụ để so sánh hai cách trên: một quá trình copy dữ liệu từ tape drive sang disk file, sắp xếp disk file, rồi in kết quả ra printer.

- Khuyết điểm của các cách trên:
  - Hiệu suất sử dụng tài nguyên (resource utilization) thấp
  - Quá trình có thể bị starvation

Ngăn deadlock (tt)

3. Ngăn No Preemption: nếu process A có giữ tài nguyên và đang yêu cầu tài nguyên khác nhưng tài nguyên này chưa cấp phát ngay được thì

   – Cách 1: Hệ thống lấy lại mọi tài nguyên mà A đang giữ
     - A chỉ bắt đầu lại được khi có được các tài nguyên đã bị lấy lại cùng với tài nguyên đang yêu cầu

   – Cách 2: Hệ thống sẽ xem tài nguyên mà A yêu cầu
     - Nếu tài nguyên được giữ bởi một process khác đang đợi thêm tài nguyên, tài nguyên này được hệ thống lấy lại và cấp phát cho A.
     - Nếu tài nguyên được giữ bởi process không đợi tài nguyên, A phải đợi và tài nguyên của A bị lấy lại. Tuy nhiên hệ thống chỉ lấy lại các tài nguyên mà process khác yêu cầu

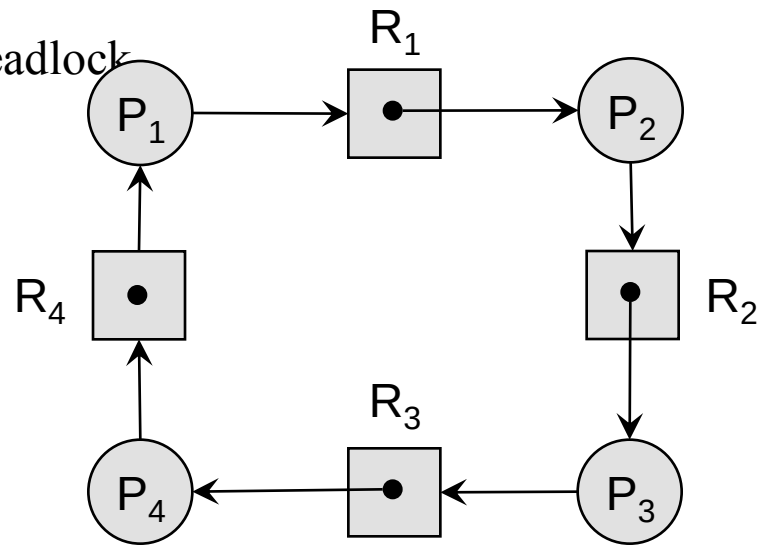
Ngăn deadlock (tt)

4. Ngăn Circular Wait: gán một thứ tự cho tất cả các tài nguyên trong hệ thống.
   - Tập hợp loại tài nguyên: $R=\{R_1, R_2, \ldots, R_m\}$

     Hàm ánh xạ: F: R->N
   - Ví dụ: F(tape drive) = 1, F(disk drive) = 5, F(printer) = 12
     - F là hàm định nghĩa thứ tự trên tập các loại tài nguyên.

Ngăn deadlock (tt)

4. Ngăn Circular Wait (tt)
   - Mỗi process chỉ có thể yêu cầu thực thể của một loại tài nguyên theo thứ tự tăng dần (định nghĩa bởi hàm F) của loại tài nguyên. Ví dụ
     - Chuỗi yêu cầu thực thể hợp lệ: tape drive [redacted]sk drive [redacted]rinter
     - Chuỗi yêu cầu thực thể không hợp lệ: disk drive [redacted]pe drive
   - Khi một process yêu cầu một thực thể của loại tài nguyên R_j thì nó phải trả lại các tài nguyên R_i với $F(R_i) > F(R_j)$.
   - "Chứng minh" giả sử tồn tại một chu trình deadlock
     - $F(R_4) < F(R_1)$
     - $F(R_1) < F(R_2)$
     - $F(R_2) < F(R_3)$
     - $F(R_3) < F(R_4)$
   - Vậy $F(R_4) < F(R_4)$, mâu thuẩn!



2. **Tránh tắc nghẽn**
Deadlock avoidance

- Deadlock prevention sử dụng tài nguyên không hiệu quả.
- Deadlock avoidance vẫn đảm bảo hiệu suất sử dụng tài nguyên tối đa đến mức có thể.
- Yêu cầu mỗi process khai báo số lượng tài nguyên tối đa cần để thực hiện công việc
- Giải thuật deadlock-avoidance sẽ kiểm tra *trạng thái cấp phát tài nguyên* (resource-allocation state) để bảo đảm hệ thống không rơi vào deadlock.
- Trạng thái cấp phát tài nguyên được định nghĩa dựa trên số tài nguyên còn lại, số tài nguyên đã được cấp phát và yêu cầu tối đa của các process.

Trạng thái safe và unsafe

- Một trạng thái của hệ thống được gọi là *an toàn* (safe) nếu tồn tại một chuỗi (thứ tư)ï *an toàn* (safe sequence).
- Một chuỗi quá trình <P_1, P_2,…, P_n > là một *chuỗi an toàn* nếu
  - Với mọi i = 1,…,n, yêu cầu tối đa về tài nguyên của P_i có thể được thỏa bởi
    - tài nguyên mà hệ thống đang có sẵn sàng (available)
    - cùng với tài nguyên mà tất cả P_j , j < i, đang giữ.

- Một trạng thái của hệ thống được gọi là *không an toàn* (unsafe) nếu không tồn tại một chuỗi an toàn.

Chuỗi an toàn (tt)

Ví dụ: Hệ thống có 12 tape drives và 3 quá trình P_0, P_1, P_2

- Tại thời điểm t_0

	Maximum needs	Current needs
P_0	10	5
P_1	4	2
P_2	9	2

- Còn 3 tape drive sẵn sàng.
- Chuỗi <P_1, P_0, P_2> là chuỗi an toàn [redacted]ệ thống là an toàn

Chuỗi an toàn (tt)

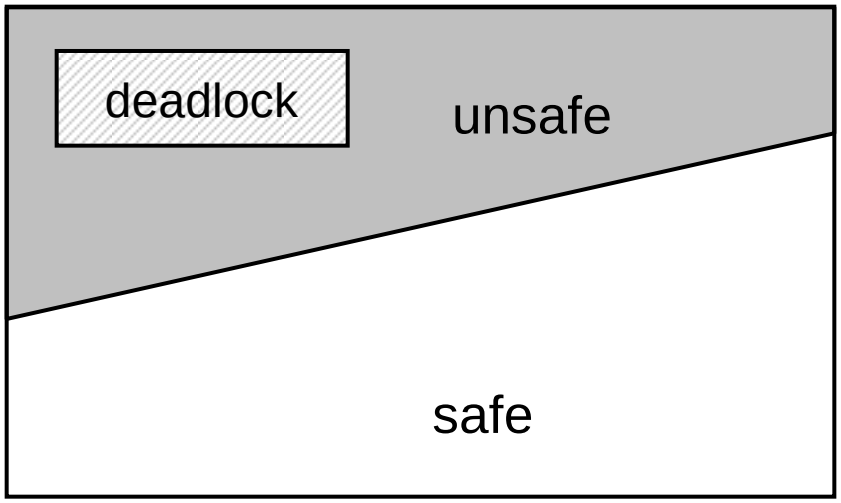
- Giả sử tại thời điểm t_1, P_2 yêu cầu và được cấp phát 1 tape drive
  - còn 2 tape drive sẵn sàng

	cần tối đa	đang giữ
P_0	10	5
P_1	4	2
P_2	9	3

- Hệ thống còn an toàn không?

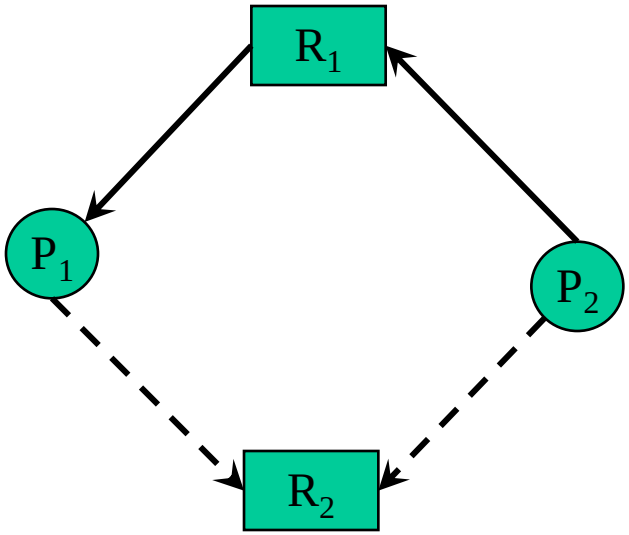
Trạng thái safe/unsafe và deadlock

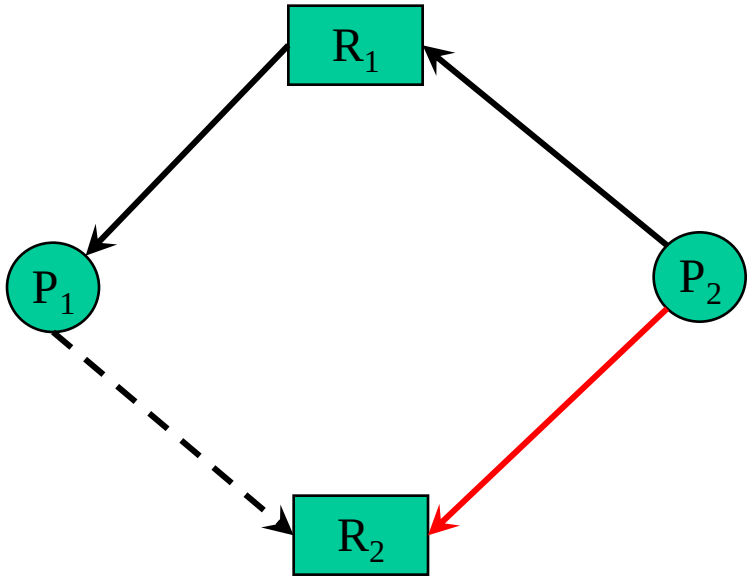
- Nếu hệ thống đang ở trạng thái safe [redacted]hông deadlock.
- Nếu hệ thống đang ở trạng thái unsafe [redacted] có thể dẫn đến deadlock.
- Tránh deadlock bằng cách bảo đảm hệ thống không đi đến trạng thái unsafe.



Giải thuật đồ thị cấp phát tài nguyên

- Khái niệm cạnh thỉnh cầu





Giải thuật banker

- Áp dụng cho hệ thống cấp phát tài nguyên trong đó mỗi loại tài nguyên có thể có nhiều instance.
- Bắt chước nghiệp vụ ngân hàng (banking)
- Điều kiện
  - Mỗi process phải khai báo số lượng thực thể (instance) tối đa của mỗi loại tài nguyên mà nó cần
  - Khi process yêu cầu tài nguyên thì có thể phải đợi mặc dù tài nguyên được yêu cầu đang có sẵn
  - Khi process đã có được đầy đủ tài nguyên thì phải hoàn trả trong một khoảng thời gian hữu hạn nào đó.

Giải thuật banker (tt)

n: số process, *m*: số loại tài nguyên

Các cấu trúc dữ liệu

Available: vector độ dài *m*

Available[j] = *k* [redacted]ại tài nguyên R_j có *k* instance sẵn sàng

Max: ma trận n [redacted]m

Max[i, j] = k [redacted]á trình P_i yêu cầu tối đa k instance của loại tài nguyên R_j

Allocation: ma trận n [redacted]m

Allocation[i, j] = k [redacted] đã được cấp phát k instance của Rj

Need: ma trận n [redacted]m

Need[i, j] = k [redacted] cần thêm k instance của Rj

Nhận xét: Need[i, j] = Max[i, j] – Allocation[i, j]

Ký hiệu Y [redacted]X [redacted][i] [redacted]X[i], ví dụ (0, 3, 2, 1) [redacted]1, 7, 3, 2)

Giải thuật banker (tt)
Giải thuật an toàn

Tìm một chuỗi an toàn

1. Gọi Work và Finish là hai vector độ dài là m và n. Khởi tạo

   Work := Available

   Finish[i] := false, i = 1,…, n

2. Tìm i thỏa

   (a) Finish[i] = false

   (b) $Need_i$ ≤ Work (hàng thứ i của Need)

   Nếu không tồn tại i như vậy, đến bước 4.

3. Work := Work + $Allocation_i$

   Finish[i] := true

   quay về bước 2.

4. Nếu Finish[i] = true, i = 1,…, n, thì hệ thống đang ở trạng thái safe

Thời gian chạy của giải thuật là $O(m \cdot n^2)$

Giải thuật banker (tt)
Giải thuật yêu cầu (cấp phát) tài nguyên

Gọi $Request_i$ là request vector của process P_i .

$Request_i [j] = k$ cần k instance của tài nguyên R_j .

1. Nếu $Request_i$ $Need_i$ thì đến bước 2. Nếu không, báo lỗi vì process đã vượt yêu cầu tối đa.
2. Nếu $Request_i$ Available thì qua bước 3. Nếu không, P_i phải chờ vì tài nguyên không còn đủ để cấp phát.
3. Giả định cấp phát tài nguyên đáp ứng yêu cầu của P_i bằng cách cập nhật trạng thái hệ thống như sau:

$Available := Available – Request_i$

$Allocation_i := Allocation_i + Request_i$

$Need_i := Need_i – Request_i$

Giải thuật banker (tt)
Giải thuật yêu cầu tài nguyên

Áp dụng giải thuật kiểm tra trạng thái an toàn lên trạng thái trên

- Nếu trạng thái là safe thì tài nguyên được cấp thực sự cho P_i .
- Nếu trạng thái là unsafe thì P_i phải đợi, và
- phục hồi trạng thái:

  $Available := Available + Request_i$

  $Allocation_i := Allocation_i – Request_i$

  $Need_i := Need_i + Request_i$

Giải thuật kiểm tra trạng thái an toàn – Ví dụ

- Có 5 process P_0 ,…, P_4
- Có 3 loại tài nguyên: *A* (có 10 instance), *B* (5 instance) và *C* (7 instance).
- Sơ đồ cấp phát trong hệ thống tại thời điểm T_0

	Allocation			Max			Available			Need			
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*	
P_0	0	1	0	7	5	3	3	3	2	7	4	3	❺
P_1	2	0	0	3	2	2				1	2	2	❶
P_2	3	0	2	9	0	2				6	0	0	❹
P_3	2	1	1	2	2	2				0	1	1	❷
P_4	0	0	2	4	3	3				4	3	1	❸

GT (kiểm tra trạng thái)an toàn – Vd (tt)

Chuỗi an toàn <P_1, P_3, P_4, P_2, P_0>

	Allocation	Need	Work
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2
P_1	2 0 0	1 2 2	5 3 2
P_2	3 0 2	6 0 0	7 4 3
P_3	2 1 1	0 1 1	7 4 5
P_4	0 0 2	4 3 1	10 4 7 → 10 5 7

GT cấp phát tài nguyên – Ví dụ

- Yêu cầu (1, 0, 2) của P_1 có thỏa được không?
  - Kiểm tra điều kiện $Request_1$ ≤ Available:
    - (1, 0, 2) ≤ (3, 3, 2) là đúng
  - Giả định thỏa yêu cầu, kiểm tra trạng thái mới có phải là safe hay không.

	Allocation			Need			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	7	4	3	2	3	0
P_1	3	0	2	0	2	0			
P_2	3	0	2	6	0	0			
P_3	2	1	1	0	1	1			
P_4	0	0	2	4	3	1			

P4 (3, 3, 0) ?
P0 (0, 2, 0) ?
P3 (0, 2, 1)?

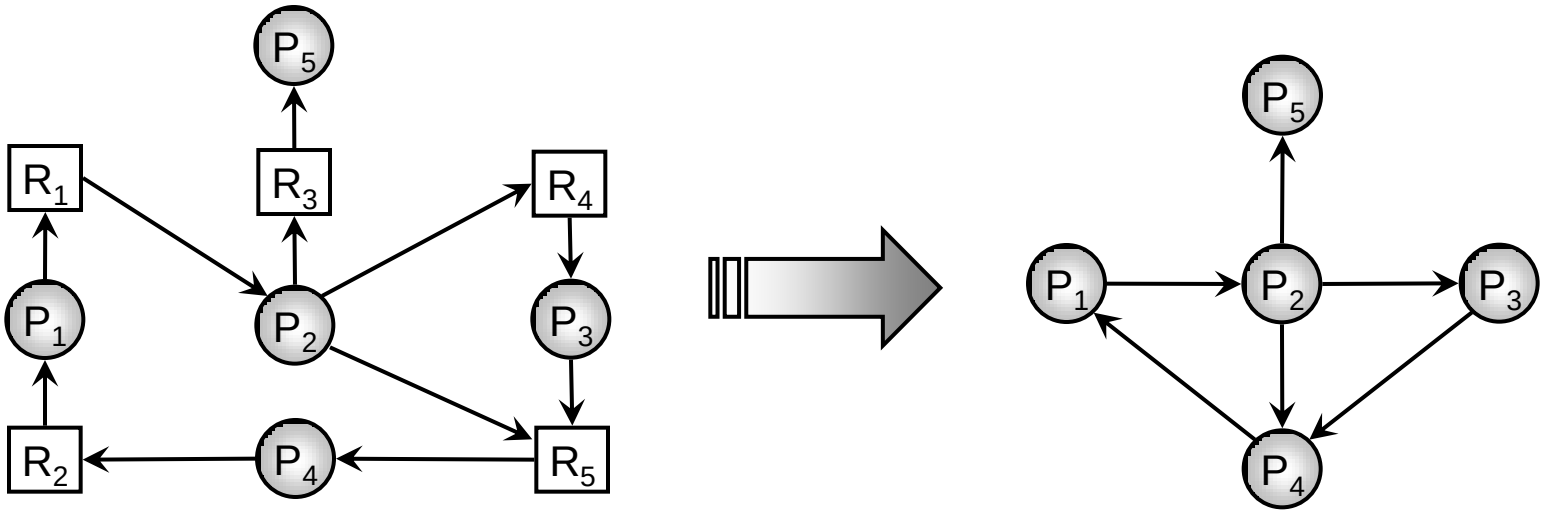
  - Trạng thái mới là safe (chuỗi an toàn là <P_1, P_3, P_4, P_0, P_2>), vậy có thể cấp phát tài nguyên cho P_1.

3. Phát hiện deadlock (Deadlock detection)

- Chấp nhận xảy ra deadlock trong hệ thống, kiểm tra trạng thái hệ thống bằng giải thuật phát hiện deadlock.
- Nếu có deadlock thì tiến hành phục hồi hệ thống
- Các giải thuật phát hiện deadlock thường sử dụng mô hình RAG.
- Hệ thống cấp phát tài nguyên được khảo sát trong mỗi trường hợp sau
  1. Mỗi loại tài nguyên chỉ có một thực thể (instance)
  2. Mỗi loại tài nguyên có thể có nhiều thực thể

Mỗi loại tài nguyên chỉ có một thực thể

- Sử dụng *wait-for graph*
  - Wait-for graph được dẫn xuất từ RAG bằng cách bỏ các node biểu diễn tài nguyên và ghép các cạnh tương ứng.
    - Có cạnh từ P_i đến P_j [redacted] đang chờ tài nguyên từ P_j



- Một giải thuật kiểm tra có tồn tại chu trình trong wait-for graph hay không sẽ được gọi định kỳ. Giải thuật phát hiện chu trình có thời gian chạy là $O(n^2)$, với n là số đỉnh của graph.

Mỗi loại tài nguyên có nhiều thực thể

- Phương pháp dùng wait-for graph không áp dụng được cho trường hợp mỗi loại tài nguyên có nhiều instance.
- Các cấu trúc dữ liệu dùng trong giải thuật phát hiện deadlock

  Available: vector độ dài m
  - số instance sẵn sàng của mỗi loại tài nguyên
- *Allocation:* ma trận n [redacted] m
  - số instance của mỗi loại tài nguyên đã cấp phát cho mỗi process
- *Request:* ma trận n [redacted] m
  - yêu cầu hiện tại của mỗi process.
  - Request [i, j] = k [redacted] đang yêu cầu thêm k instance của Rj

Giải thuật phát hiện deadlock

1. Gọi *Work* và *Finish* là vector kích thước *m* và *n*. Khởi tạo:

   Work := Available

   i = 1, 2,…, n, nếu $Allocation_i$ ≠ 0 thì Finish[i] := false

   còn không thì Finish[i] := true

2. Tìm i thỏa mãn:

   Finish[i] := false và

   $Request_i$ ≤ Work

   thời gian chạy của giải thuật $O(m \cdot n^2)$

- Nếu không tồn tại i như thế, đến bước 4.

3. Work := Work + $Allocation_i$

   Finish[i] := true

   quay về bước 2.

4. Nếu Finish[i] = false, với một i = 1,…, n, thì hệ thống đang ở trạng thái deadlock. Hơn thế nữa, Finish[i] = false thì P_i bị deadlocked.

Giải thuật phát hiện deadlock – Ví dụ

- Hệ thống có 5 quá trình P_0 ,…, P_4
- 3 loại tài nguyên: *A* (7 instance), *B* (2 instance), *C* (6 instance).

	Allocation			Request			Available		
	A	*B*	*C*	*A*	*B*	*C*	*A*	*B*	*C*
P_0	0	1	0	0	0	0	0	0	0
P_1	2	0	0	2	0	2			
P_2	3	0	3	0	0	0			
P_3	2	1	1	1	0	0			
P_4	0	0	2	0	0	2			

Chạy giải thuật, tìm được chuỗi <P_0, P_2, P_3, P_1, P_4> với Finish[i] = true, i = 1,…, n, vậy hệ thống không bị deadlocked.

Giải thuật phát hiện deadlock – Ví dụ (tt)

- P_2 yêu cầu thêm một instance của C. Ma trận Request như sau:

	Request		
	A	B	C
P_0	0	0	0
P_1	2	0	2
P_2	0	0	1
P_3	1	0	0
P_4	0	0	2

  - Trạng thái của hệ thống là gì?
    - Có thể thu hồi tài nguyên đang sở hữu bởi process P_0 nhưng vẫn không đủ đáp ứng yêu cầu của các process khác.
    - Vậy tồn tại deadlock, bao gồm các process P_1, P_2, P_3, và P_4.

Phục hồi deadlock (Deadlock Recovery)

- Khi deadlock xảy ra, để phục hồi
  - báo người vận hành (operator)

  hoặc
  - hệ thống tự động phục hồi bằng cách bẻ gãy chu trình deadlock:
    - chấm dứt một hay nhiều quá trình
    - lấy lại tài nguyên từ một hay nhiều quá trình

Deadlock Recovery: Chấm dứt quá trình

- Phục hồi hệ thống bị deadlock bằng cách chấm dứt quá trình
  - Chấm dứt tất cả process bị deadlocked, hoặc
  - Chấm dứt lần lượt từng process cho đến khi không còn deadlock
    - Sử dụng giải thuật phát hiện deadlock để xác định còn deadlock hay không

- Dựa trên yếu tố nào để chọn process cần được chấm dứt?
  - Độ ưu tiên của process
  - Thời gian đã thực thi của process và thời gian còn lại
  - Loại tài nguyên mà process đã sử dụng
  - Tài nguyên mà process cần thêm để hoàn tất công việc
  - Số lượng process cần được chấm dứt
  - Process là interactive process hay batch process

Deadlock recovery: Lấy lại tài nguyên

- Lấy lại tài nguyên từ một process, cấp phát cho process khác cho đến khi không còn deadlock nữa.

- Các vấn đề trong chiến lược thu hồi tài nguyên:
  - Chọn "nạn nhân" để tối thiểu chi phí (có thể dựa trên số tài nguyên sở hữu, thời gian CPU đã tiêu tốn,...)
  - Trở lại trạng thái trước deadlock (Rollback): rollback process bị lấy lại tài nguyên trở về trạng thái safe, tiếp tục process từ trạng thái đó. Hệ thống cần lưu giữ một số thông tin về trạng thái các process đang thực thi.
  - Đói tài nguyên (Starvation): để tránh starvation, phải bảo đảm không có process sẽ luôn luôn bị lấy lại tài nguyên mỗi khi deadlock xảy ra.

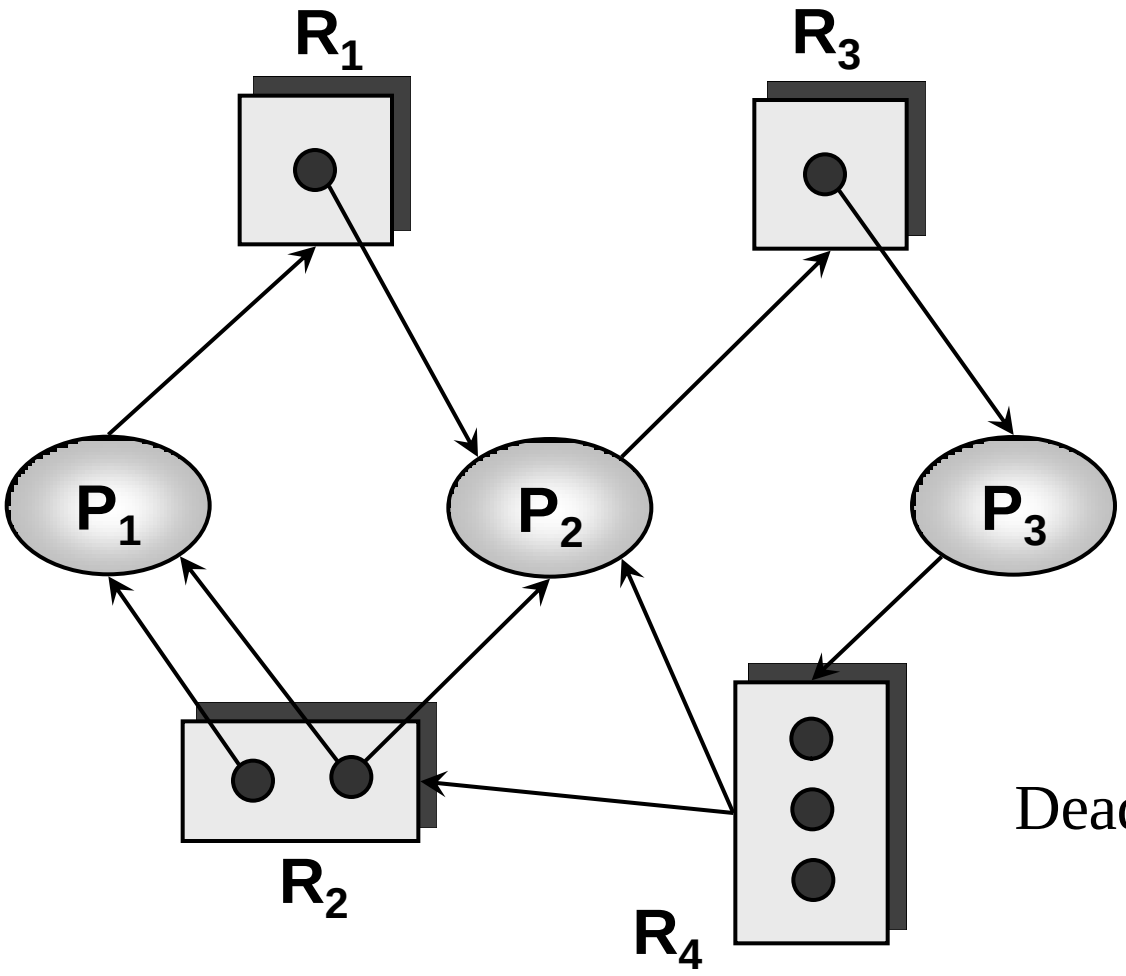
Phương pháp kết hợp để giải quyết Deadlock

- Kết hợp 3 phương pháp cơ bản
  - Ngăn chặn (Prevention)
  - Tránh (Avoidance)
  - Phát hiện (Detection)

  Cho phép sử dụng cách giải quyết tối ưu cho mỗi lớp tài nguyên trong hệ thống.
- Phân chia tài nguyên thành các lớp theo thứ bậc.
  - Sử dụng kỹ thuật thích hợp nhất cho việc quản lý deadlock trong mỗi lớp này.

Bài tập

- Bài 01: Liệt kê 3 trường hợp xảy ra deadlock trong đời sống
- Bài 02:



Deadlock ?

Bài tập

- Bài 03:

	Allocation	*Max*	*Available*
	A B C D	*A B C D*	*A B C D*
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

- A) Tìm Need
- B) Hệ thống có an toàn không
- C)Nếu P_1 yêu cầu (0,4,2,0) thì có thể cấp phát cho nó ngay không?

Chương 7. Quản lý bộ nhớ

- Khái niệm cơ sở
- Các kiểu địa chỉ nhớ (physical address , logical address)
- Chuyển đổi địa chỉ nhớ
- Overlay và swapping
- Mô hình quản lý bộ nhớ đơn giản
  - Fixed partitioning
  - Dynamic partitioning
  - Cơ chế phân trang (paging)
  - Cơ chế phân đoạn (segmentation)
  - Segmentation with paging

Khái niệm cơ sở

- Chương trình phải được mang vào trong bộ nhớ và đặt nó trong một tiến trình để được xử lý
- Input Queue – Một tập hợp của những tiến trình trên đĩa mà đang chờ để được mang vào trong bộ nhớ để thực thi.
- User programs trải qua nhiều bước trước khi được xử lý.

Khái niệm cơ sở

- Quản lý bộ nhớ là công việc của hệ điều hành với sự hỗ trợ của phần cứng nhằm phân phối, sắp xếp các process trong bộ nhớ sao cho hiệu quả.
- Mục tiêu cần đạt được là nạp càng nhiều process vào bộ nhớ càng tốt (gia tăng mức độ đa chương)
- Trong hầu hết các hệ thống, kernel sẽ chiếm một phần cố định của bộ nhớ; phần còn lại phân phối cho các process.
- Các yêu cầu đối với việc quản lý bộ nhớ
  - Cấp phát bộ 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 user vào địa chỉ thực

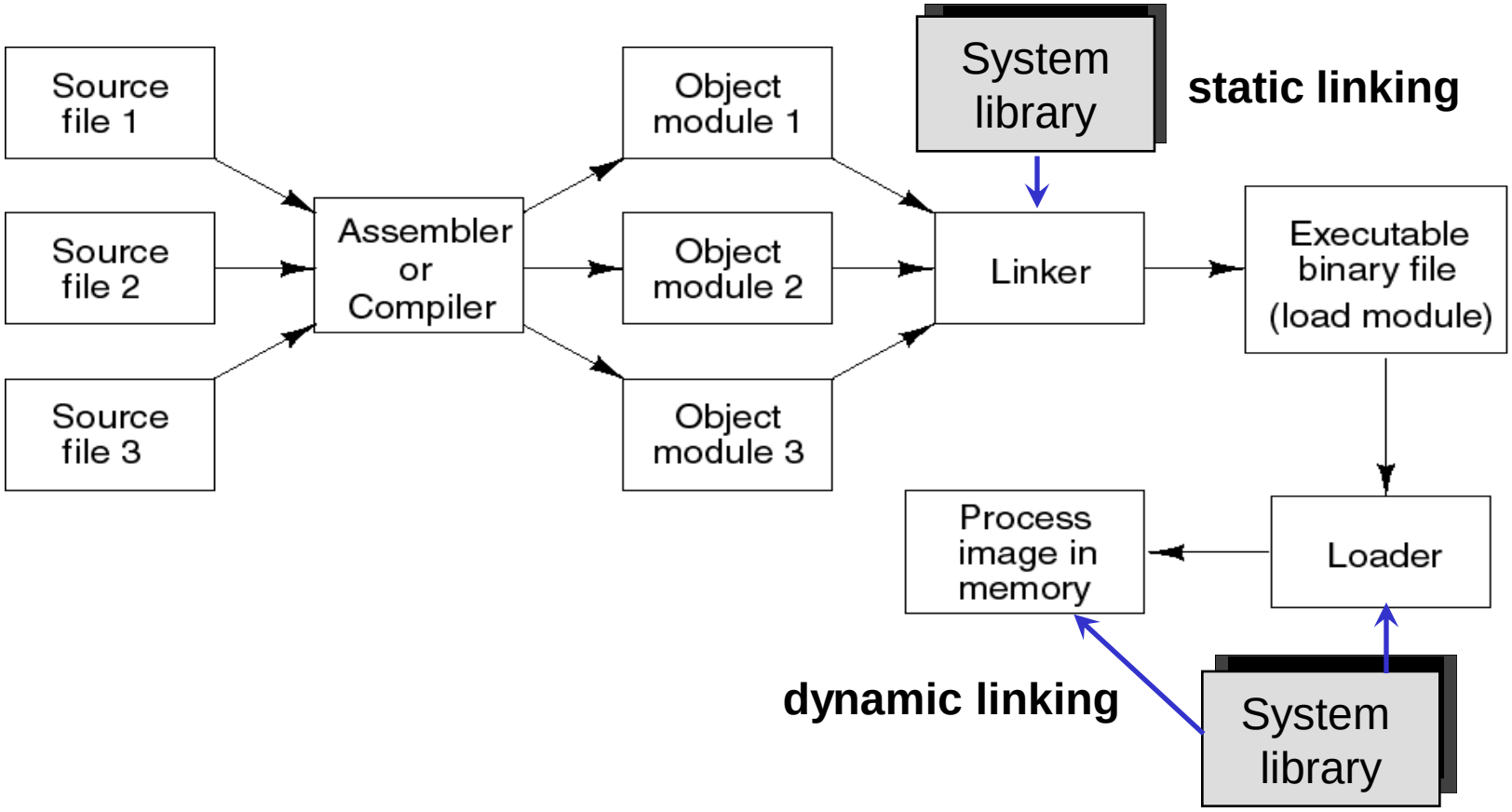
Các kiểu địa chỉ nhớ

- *Địa chỉ vật lý* (physical address) (địa chỉ *thực*) là một vị trí thực trong bộ nhớ chính.

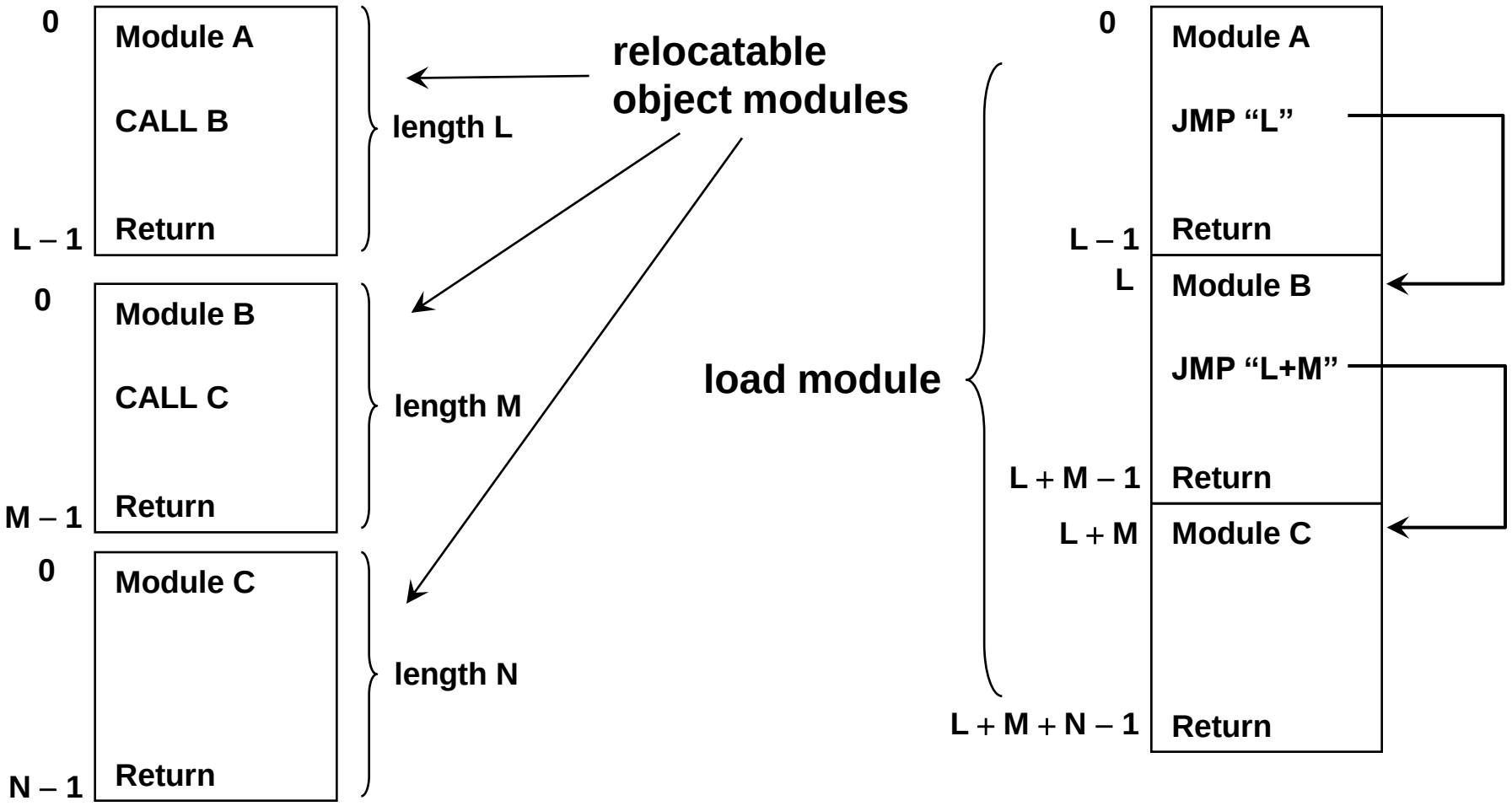
- *Địa chỉ luận lý* (logical address) là một vị trí nhớ được diễn tả trong một chương trình (còn gọi là địa chỉ ảo virtual address)
  - Các trình biên dịch (compiler) tạo ra mã lệnh chương trình mà trong đó mọi tham chiếu bộ nhớ đều là địa chỉ luận lý
  - *Địa chỉ tương đối* (relative address) (địa chỉ *khả tái định vị*, relocatable address) là một kiểu địa chỉ luận lý trong đó các địa chỉ được biểu diễn 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,…
  - *Địa chỉ tuyệt đối* (absolute address): địa chỉ tương đương với địa chỉ thực.

Nạp chương trình vào bộ nhớ

- Bộ linker: kết hợp các object module thành một file nhị phân khả thực thi gọi là *load module*.
- Bộ loader: nạp load module vào bộ nhớ chính

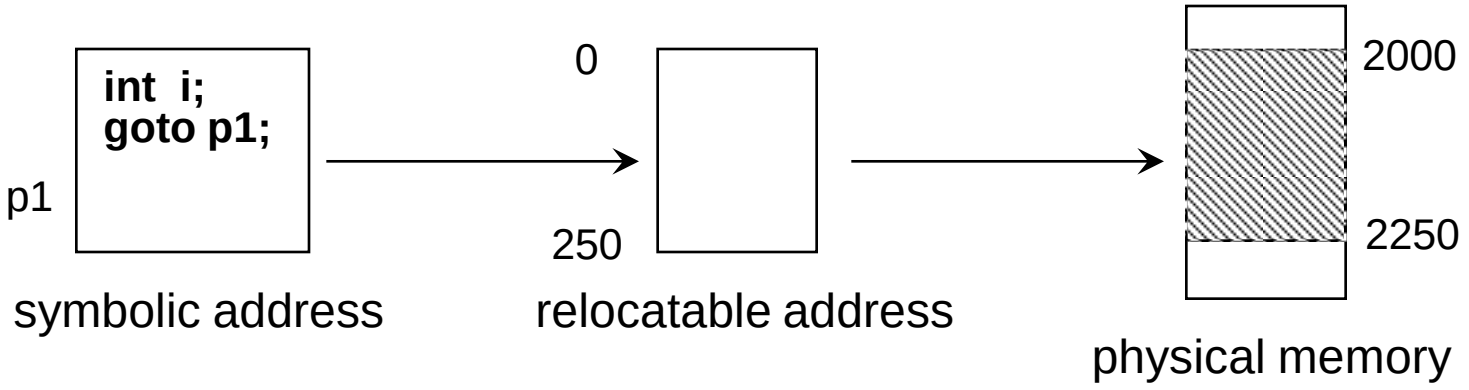


Cơ chế thực hiện linking



Chuyển đổi địa chỉ

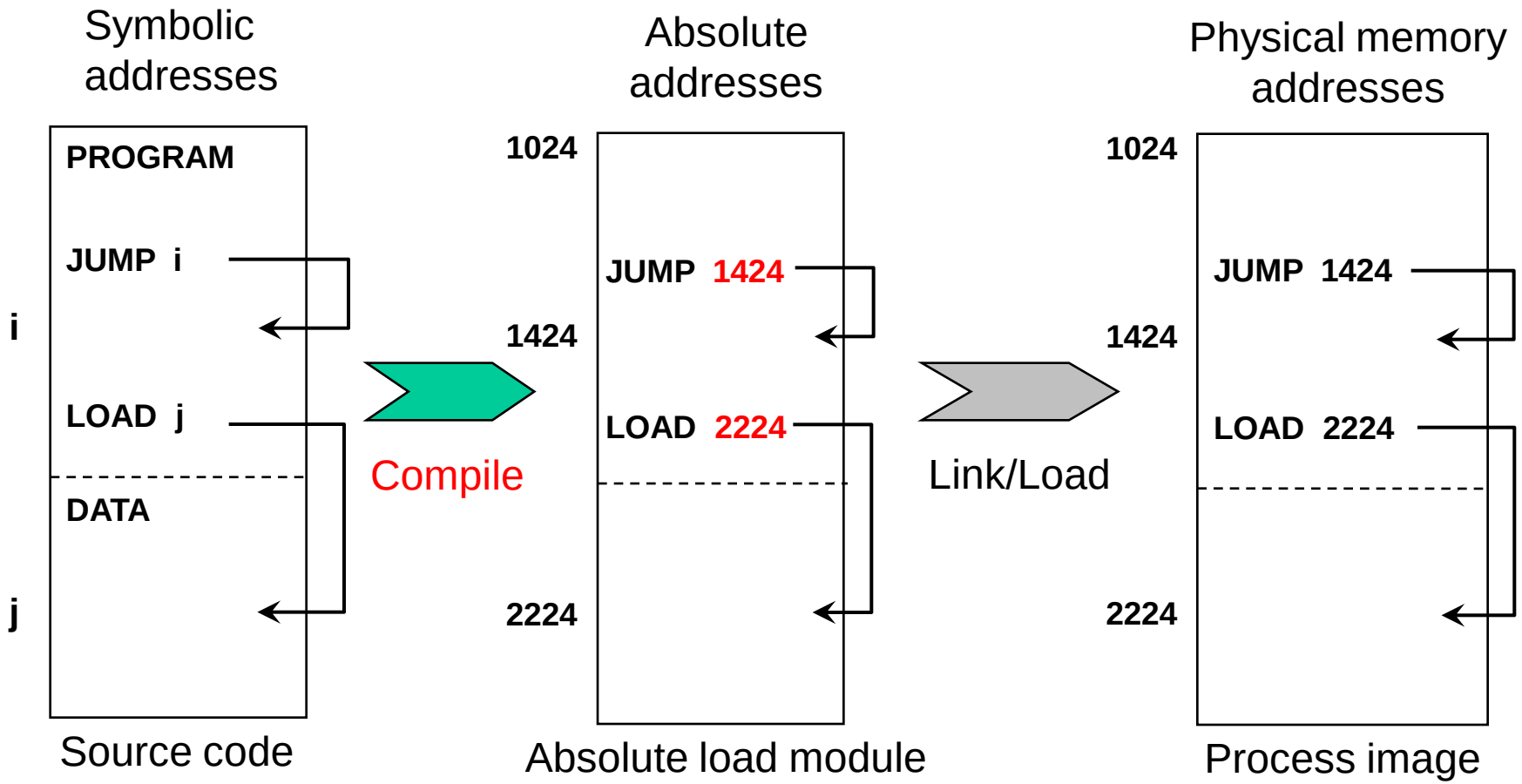
- *Chuyển đổi địa chỉ*: quá trình ánh xạ một địa chỉ từ không gian địa chỉ này sang không gian địa chỉ khác.
- Biểu diễn địa chỉ nhớ
  - Trong source code: symbolic (các biến, hằng, pointer,…)
  - Thời điểm biên dịch: thường là địa chỉ khả tái định vị
    - Ví dụ: a ở vị trí 14 bytes so với vị trí bắt đầu của module.
  - Thời điểm linking/loading: có thể là địa chỉ thực. Ví dụ: dữ liệu nằm tại địa chỉ bộ nhớ thực 2030



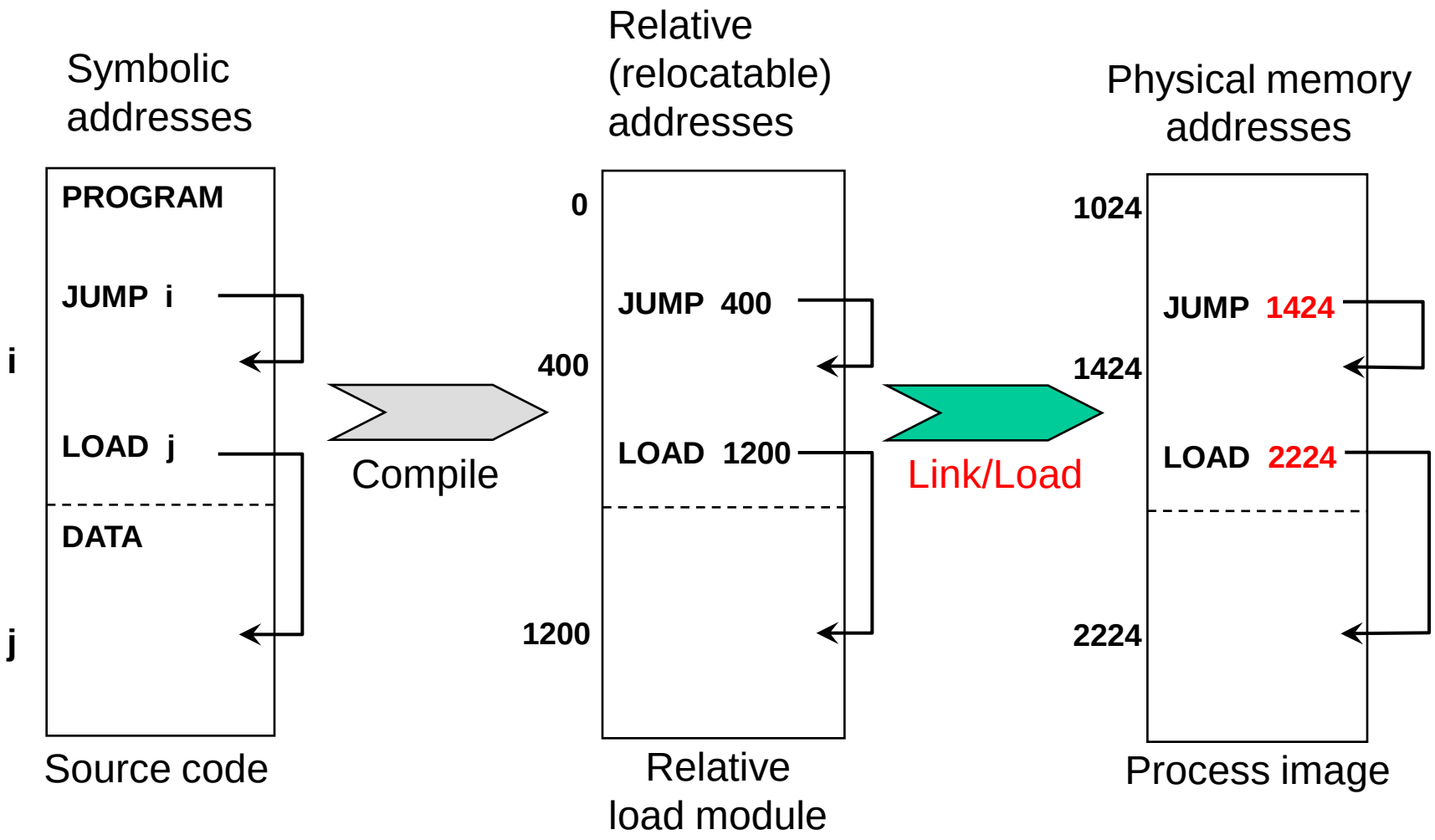
Chuyển đổi địa chỉ (tt)

- Địa chỉ lệnh (instruction) và dữ liệu (data) được chuyển đổi thành địa chỉ thực có thể xảy ra tại ba thời điểm khác nhau
  - 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
    - Khuyết điểm: phải biên dịch lại nếu thay đổi địa chỉ nạp chương trình
  - Load time: Vào thời điểm loading, *loader* phải chuyển đổi địa chỉ khả tái định vị thành địa chỉ thực dựa trên một *địa chỉ nền* (base address).
    - Địa chỉ thực được tính toán vào thời điểm nạp chương trình ⇒ phải tiến hành reload nếu địa chỉ nền thay đổi.

Sinh địa chỉ tuyệt đối vào thời điểm dịch

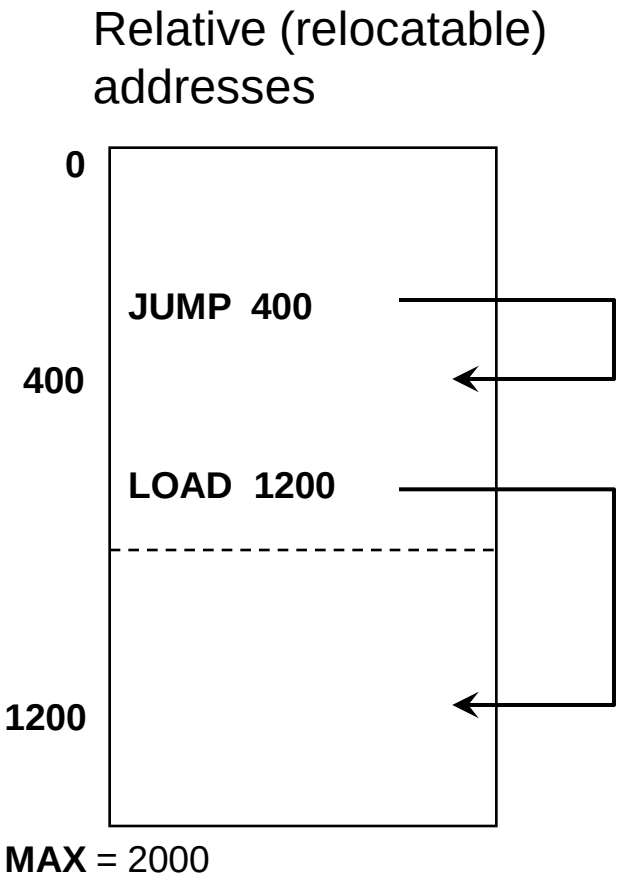


Sinh địa chỉ thực vào thời điểm nạp



Chuyển đổi địa chỉ (tt)

- Execution time: khi trong quá trình thực thi, process có thể được di chuyển từ segment này sang segment khác trong bộ nhớ thì quá trình chuyển đổi địa chỉ được trì hoãn đến thời điểm thực thi
  - Cần sự hỗ trợ của phần cứng cho việc ánh xạ địa chỉ.
    - Ví dụ: trường hợp địa chỉ luận lý là relocatable thì có thể dùng thanh ghi base và limit,…
  - Sử dụng trong đa số các OS đa dụng (general-purpose) trong đó có các cơ chế swapping, paging, segmentation



Dynamic linking

- Quá trình link đến một *module ngoài* (external module) được thực hiện sau khi đã tạo xong load module (i.e. file có thể thực thi, executable)
  - Ví dụ trong Windows: module ngoài là các file .DLL còn trong Unix, các module ngoài là các file .so (shared library)
- Load module chứa các stub tham chiếu (refer) đến routine của external module.
  - Lúc thực thi, 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).

Ưu điểm của dynamic linking

- Thông thường, external module là một thư viện cung cấp các tiện ích của OS. Các chương trình thực thi có thể dùng các phiên bản khác nhau của external module mà không cần sửa đổi, biên dịch lại.
- *Chia sẻ mã* (code sharing): một external module chỉ cần nạp vào bộ nhớ một lần. Các process cần dùng external module này thì cùng chia sẻ đoạn mã của external module ⇒ tiết kiệm không gian nhớ và đĩa.
- Phương pháp dynamic linking cần sự hỗ trợ của OS trong việc 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

- **Cơ chế**: chỉ khi nào cần được gọi đến thì một thủ tục mới được nạp vào bộ nhớ chính ⇒ 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, không được sử dụng thường xuyên (ví dụ các thủ tục xử lý lỗi)

- Hỗ trợ từ hệ điều hành
  - Thông thường, user chịu trách nhiệm thiết kế và hiện thực các chương trình có dynamic loading.
  - Hệ điều hành chủ yếu cung cấp một số thủ tục thư viện hỗ trợ, tạo điều kiện dễ dàng hơn cho lập trình viên.

Cơ chế phủ lắp (overlay)

- Tại mỗi thời điểm, chỉ giữ lại trong bộ nhớ những lệnh hoặc dữ liệu cần thiết, giải phóng các lệnh/dữ liệu chưa hoặc không cần dùng đến.

- Cơ chế này rất hữu dụng khi kích thước một process lớn hơn không gian bộ nhớ cấp cho process đó.

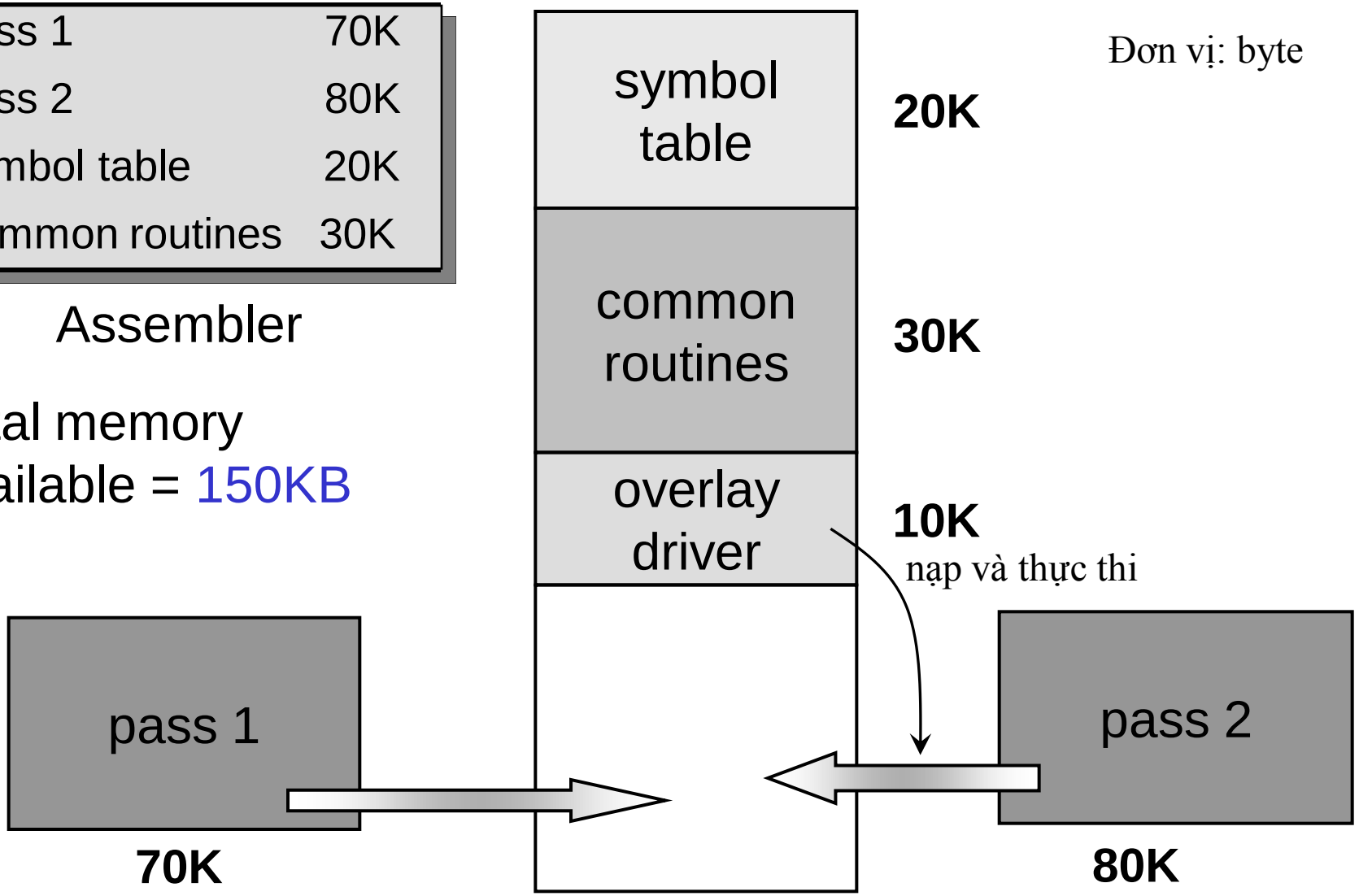
- Cơ chế này được điều khiển bởi người sử dụng (thông qua sự hỗ trợ của các thư viện lập trình) chứ không cần sự hỗ trợ của hệ điều hành

Cơ chế overlay (tt)

Pass 1	70K
Pass 2	80K
Symbol table	20K
Common routines	30K

Assembler

Total memory available = 150KB



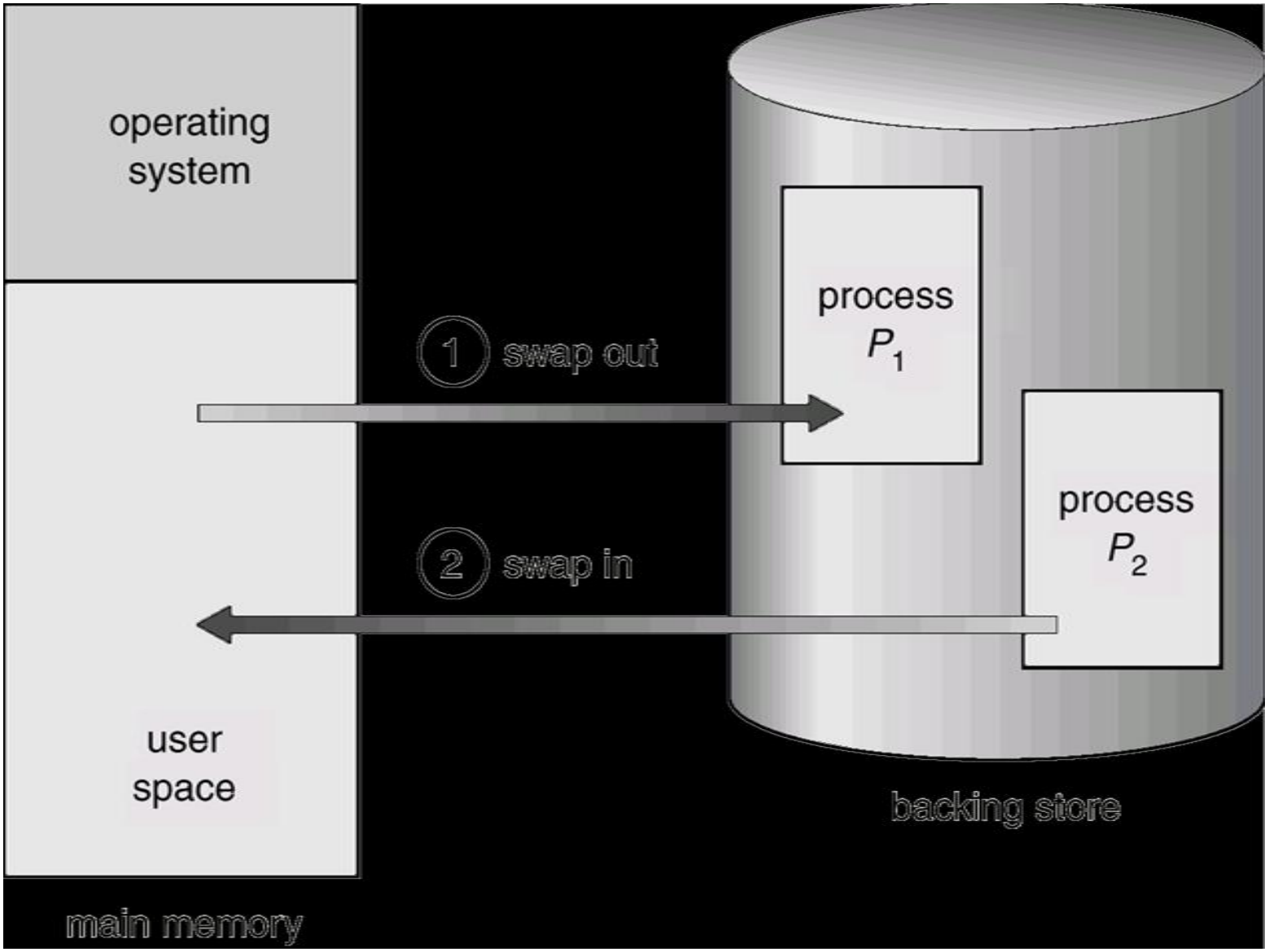
Cơ chế hoán vị (swapping)

- Một process có thể tạm thời bị swap ra khỏi bộ nhớ chính và lưu trên một hệ thống lưu trữ phụ. Sau đó, process có thể được nạp lại vào bộ nhớ để tiếp tục quá trình thực thi.

  Swapping policy: hai ví dụ

  - *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 cơ chế 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 mới đến được nạp vào bộ nhớ để thực thi
- Hiện nay, ít hệ thống sử dụng cơ chế swapping trên

Minh họa cơ chế swapping



Mô hình quản lý bộ nhớ

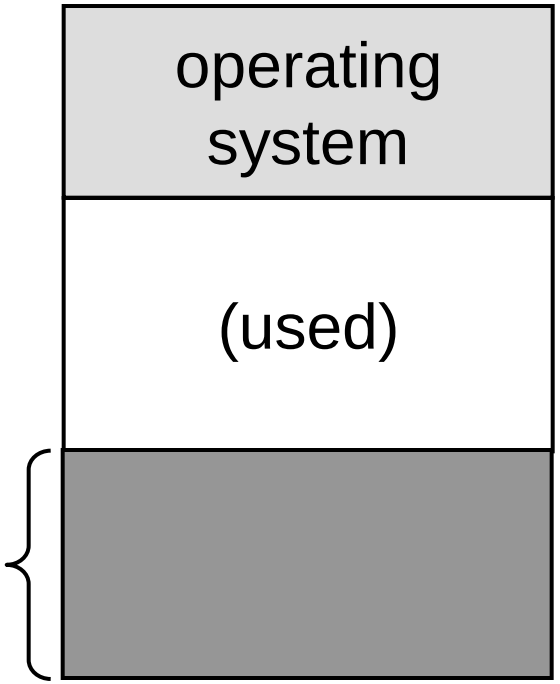
- Trong chương này, mô hình quản lý bộ nhớ là một mô hình đơn giản, không có bộ nhớ ảo.
- Một process phải được nạp hoàn toàn vào bộ nhớ thì mới được thực thi (ngoại trừ khi sử dụng cơ chế overlay).
- Các cơ chế quản lý bộ nhớ sau đây rất ít (hầu như không còn) được dùng trong các hệ thống hiện đại
  - Phân chia cố định (fixed partitioning)
  - Phân chia động (dynamic partitioning)
  - Phân trang đơn giản (simple paging)
  - Phân đoạn đơn giản (simple segmentation)

Phân mảnh (fragmentation)

- *Phân mảnh ngoại* (external fragmentation)
  - Kích thước không gian nhớ còn trống đủ để thỏa mãn một yêu cầu cấp phát, tuy nhiên không gian nhớ này không liên tục ⇒ có thể dùng cơ chế *kết khối* (compaction) để gom lại thành vùng nhớ liên tục.

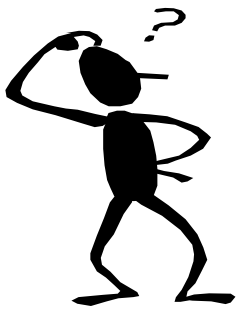
- *Phân mảnh nội* (internal fragmentation)
  - Kích thước vùng nhớ được cấp phát có thể hơi lớn hơn vùng nhớ yêu cầu.
    - Ví dụ: cấp một khoảng trống 18,464 bytes cho một process yêu cầu 18,462 bytes.
  - Hiện tượng phân mảnh nội thường xảy ra khi bộ nhớ thực được chia thành các khối kích thước cố định (fixed-sized block) và các process được cấp phát theo đơn vị khối. Ví dụ: cơ chế phân trang (paging).

Phân mảnh nội

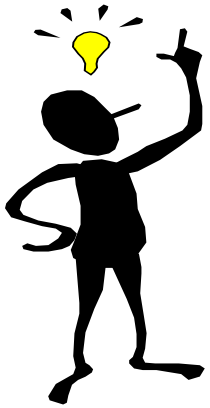


hole kích thước 18,464 bytes

yêu cầu kế tiếp là 18,462 bytes !!!



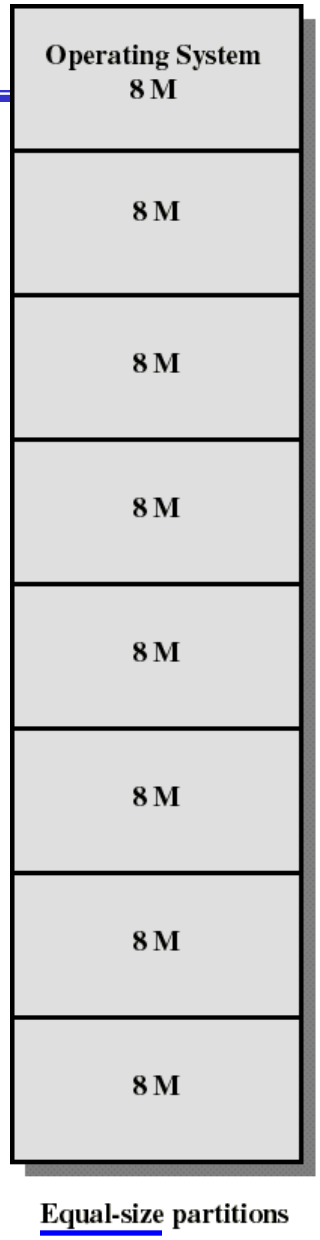
cần quản lý khoảng trống 2 bytes !?!



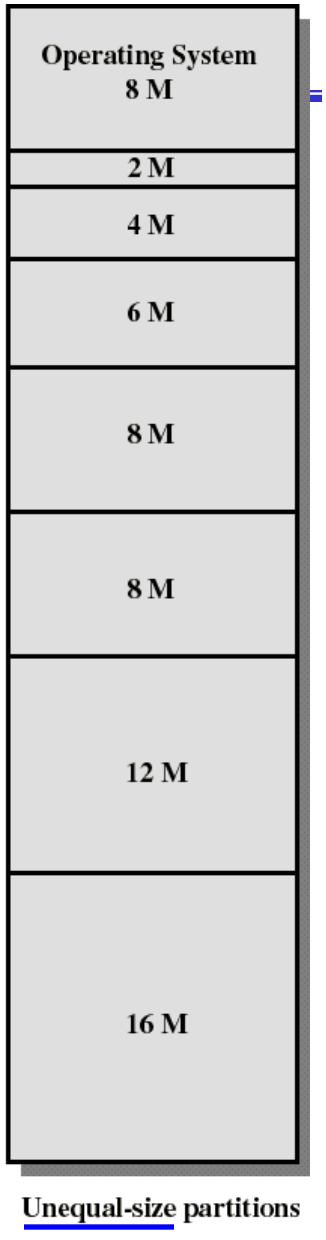
OS sẽ cấp phát hẳn khối 18,464 bytes cho process ⇒ dư ra 2 bytes không dùng!

Fixed partitioning

- Khi khởi động hệ thống, bộ nhớ chính được chia thành nhiều phần 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.
- Nếu chương trình có kích thước lớn hơn partition thì phải dùng cơ chế overlay.
- Nhận xét
  - Không hiệu quả do bị phân mảnh nội: một chương trình dù lớn hay nhỏ đều được cấp phát trọn một partition.



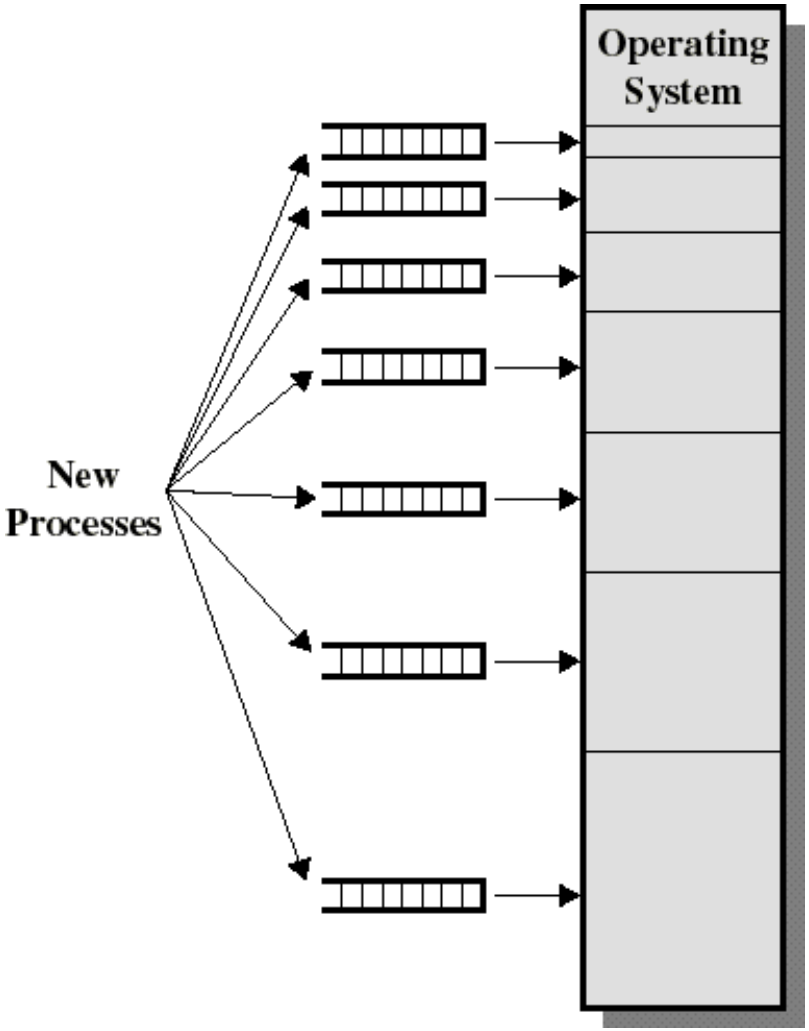
Equal-size partitions



Unequal-size partitions

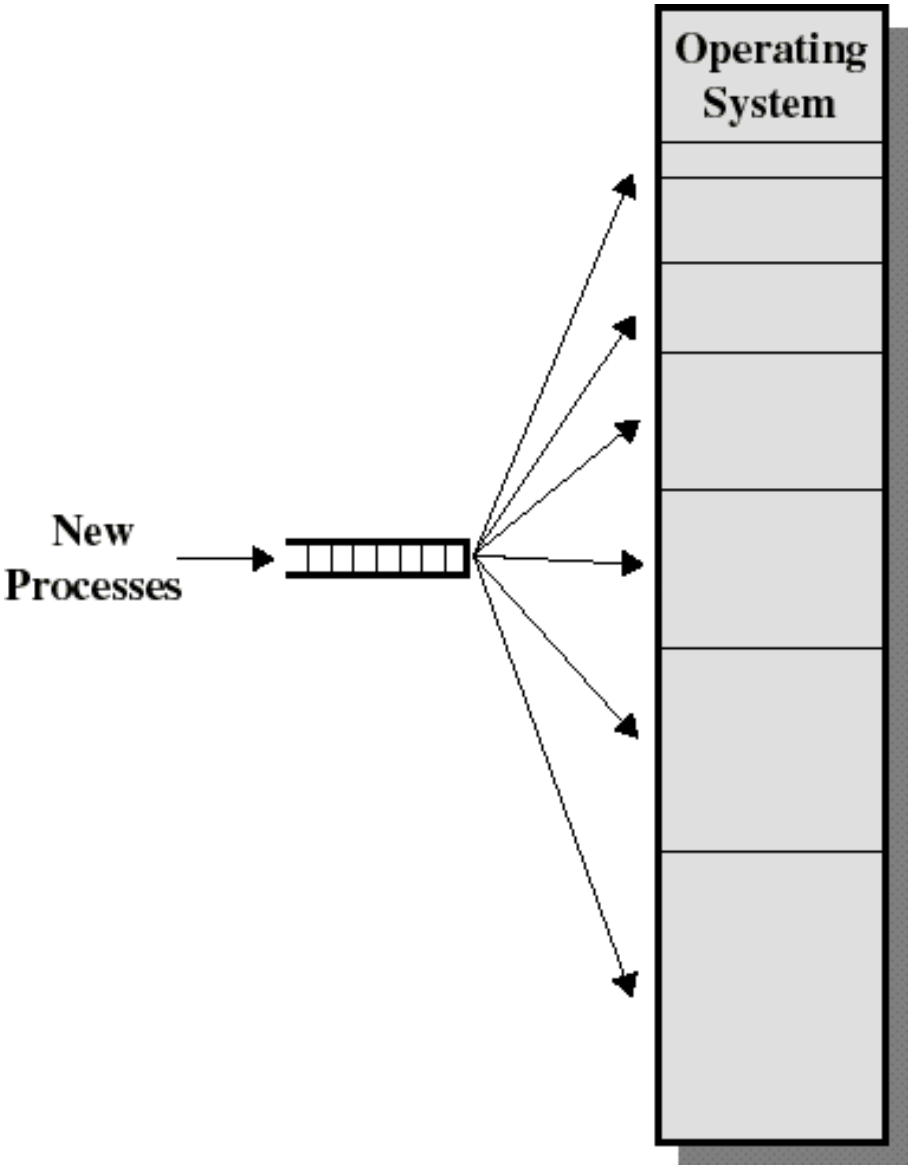
Chiến lược placement (tt)

- Partition có kích thước bằng nhau
  - Nếu còn partition trống ⇒ process mới sẽ được nạp vào partition đó
  - Nếu không còn partition trống, nhưng trong đó có process đang bị blocked ⇒ swap process đó ra bộ nhớ phụ nhường chỗ cho process mới.
- Partition có kích thước không bằng nhau: giải pháp 1
  - Gán mỗi process vào partition nhỏ nhất phù hợp với nó
  - Có hàng đợi cho mỗi partition
  - Giảm thiểu phân mảnh nội
  - Vấn đề: có thể có một số hàng đợi trống không (vì không có process với kích thước tương ứng) và hàng đợi dày đặc



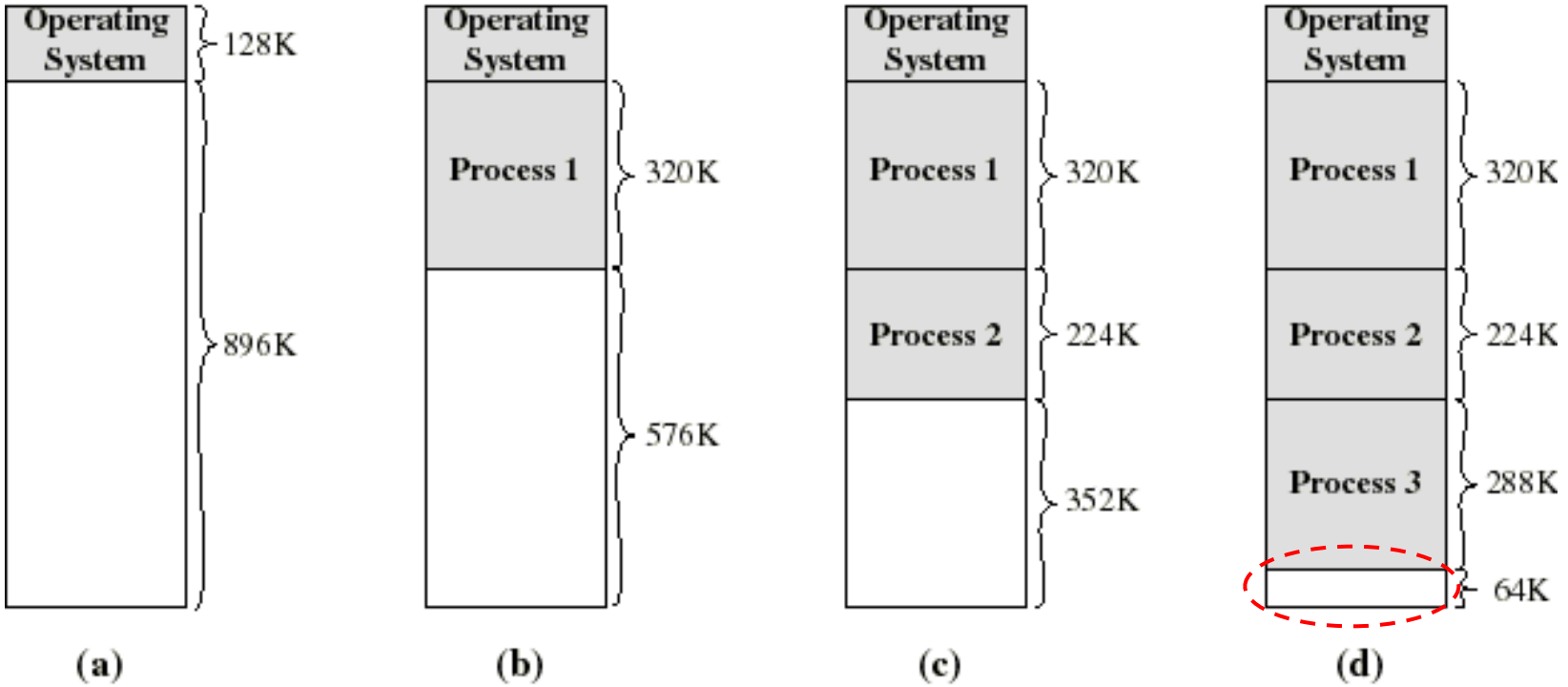
Chiến lược placement (tt)

- 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



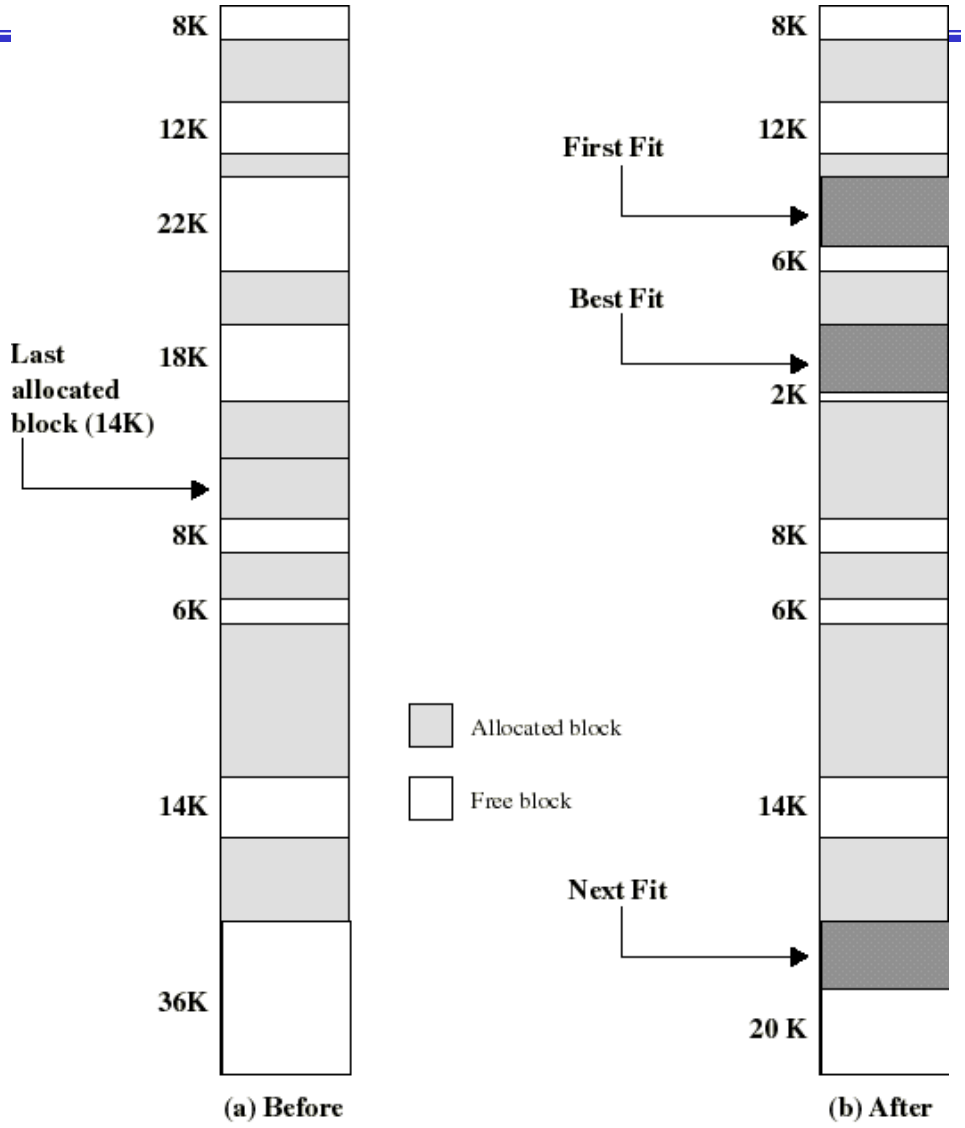
Dynamic partitioning

- Số lượng 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

- Dùng để 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

Cấp phát không liên tục

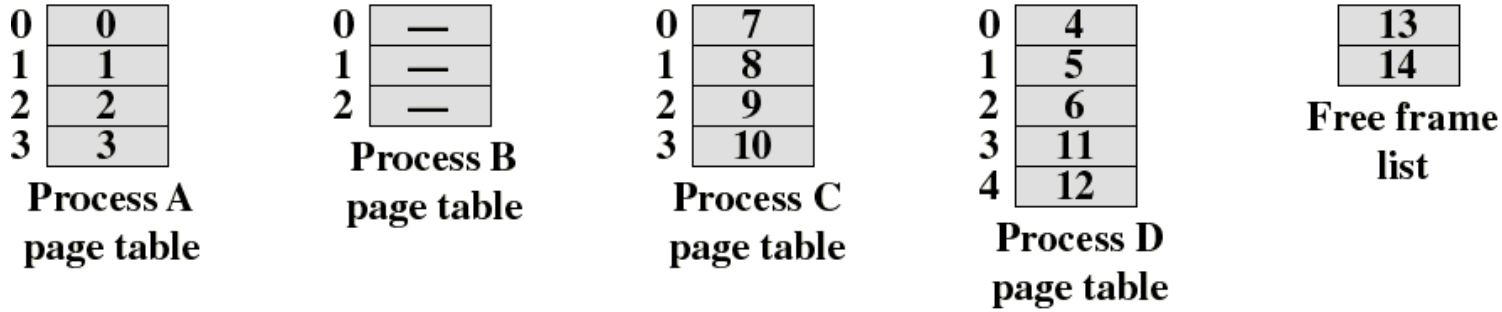
1.Cơ chế *phân trang* (paging)

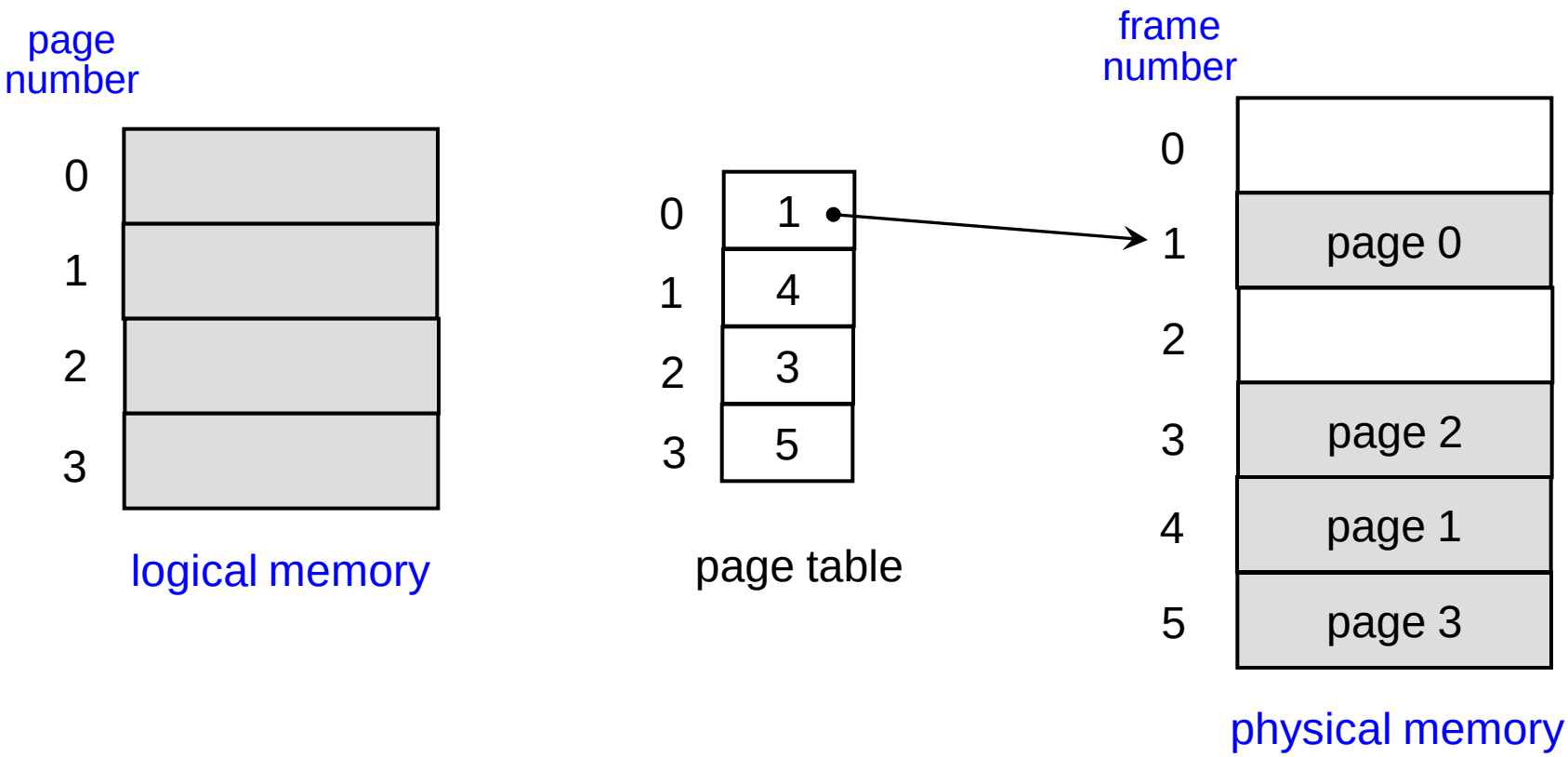
Bộ nhớ vật lý →khung trang (*frame).*

– Kích thước của frame là lũy thừa của 2, từ khoảng 512 byte đến 16MB.

- *Bộ nhớ luận lý* (logical memory) hay *không gian địa chỉ luận lý* là tập mọi địa chỉ luận lý mà một chương trình bất kỳ có thể sinh ra → page.
  – Ví dụ
  • MOV REG,1000 //1000 là một địa chỉ luận lý
- *Bảng phân trang* (page table) để ánh xạ địa chỉ luận lý thành địa chỉ thực

1.Cơ chế phân trang (tt)

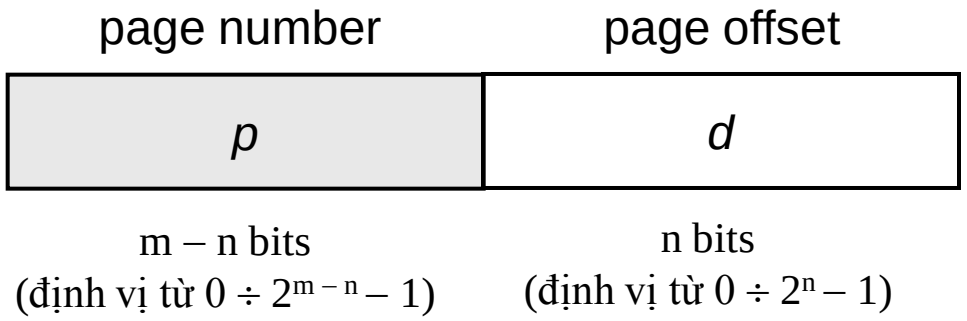




1.Cơ chế phân trang (tt)

A) Chuyển đổi địa chỉ trong paging

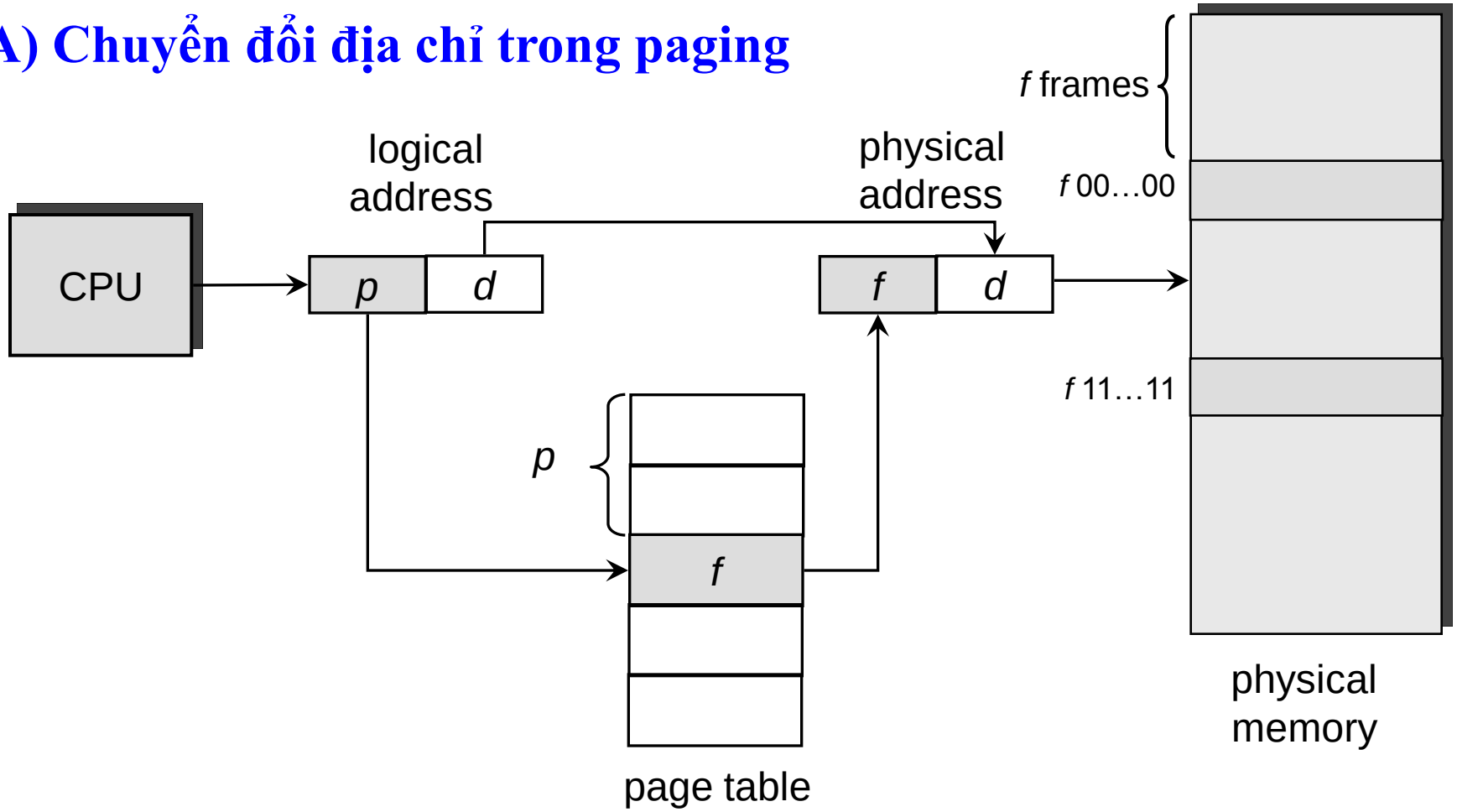
- Địa chỉ luận lý gồm có:
  - *Số hiệu trang (Page number)* p
  - *Địa chỉ tương đối trong trang (Page offset)* d
- Nếu kích thước của không gian địa chỉ ảo là 2^m, và kích thước của trang là 2^n (đơn vị là byte hay word tùy theo kiến trúc máy) thì



Bảng phân trang sẽ có tổng cộng $2^m/2^n = 2^{m-n}$ *mục* (entry)

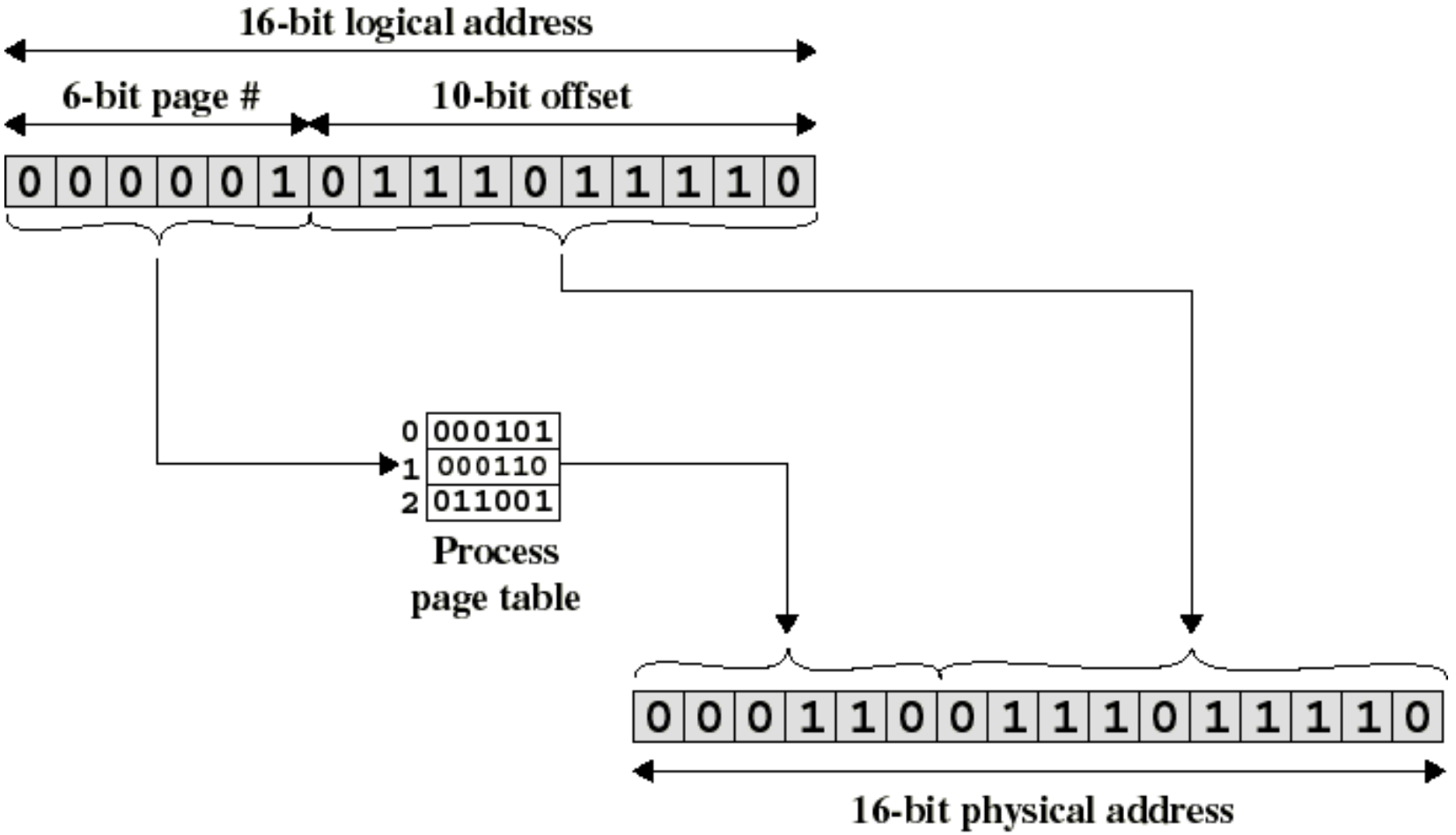
1.Cơ chế phân trang (tt)

A) Chuyển đổi địa chỉ trong paging

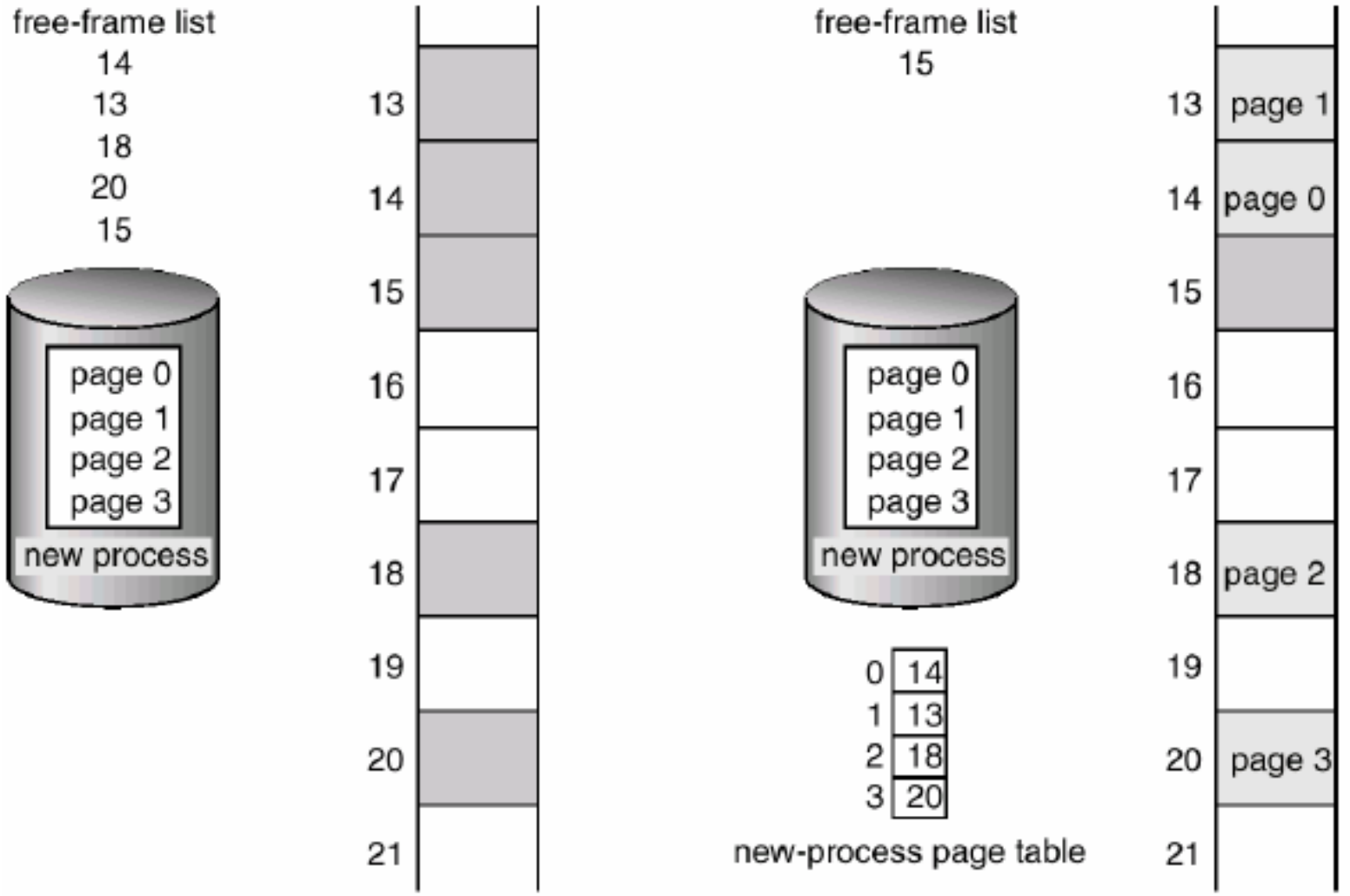


1.Cơ chế phân trang (tt)

Ví dụ: Chuyển đổi địa chỉ nhớ trong paging



1.Cơ chế phân trang (tt)



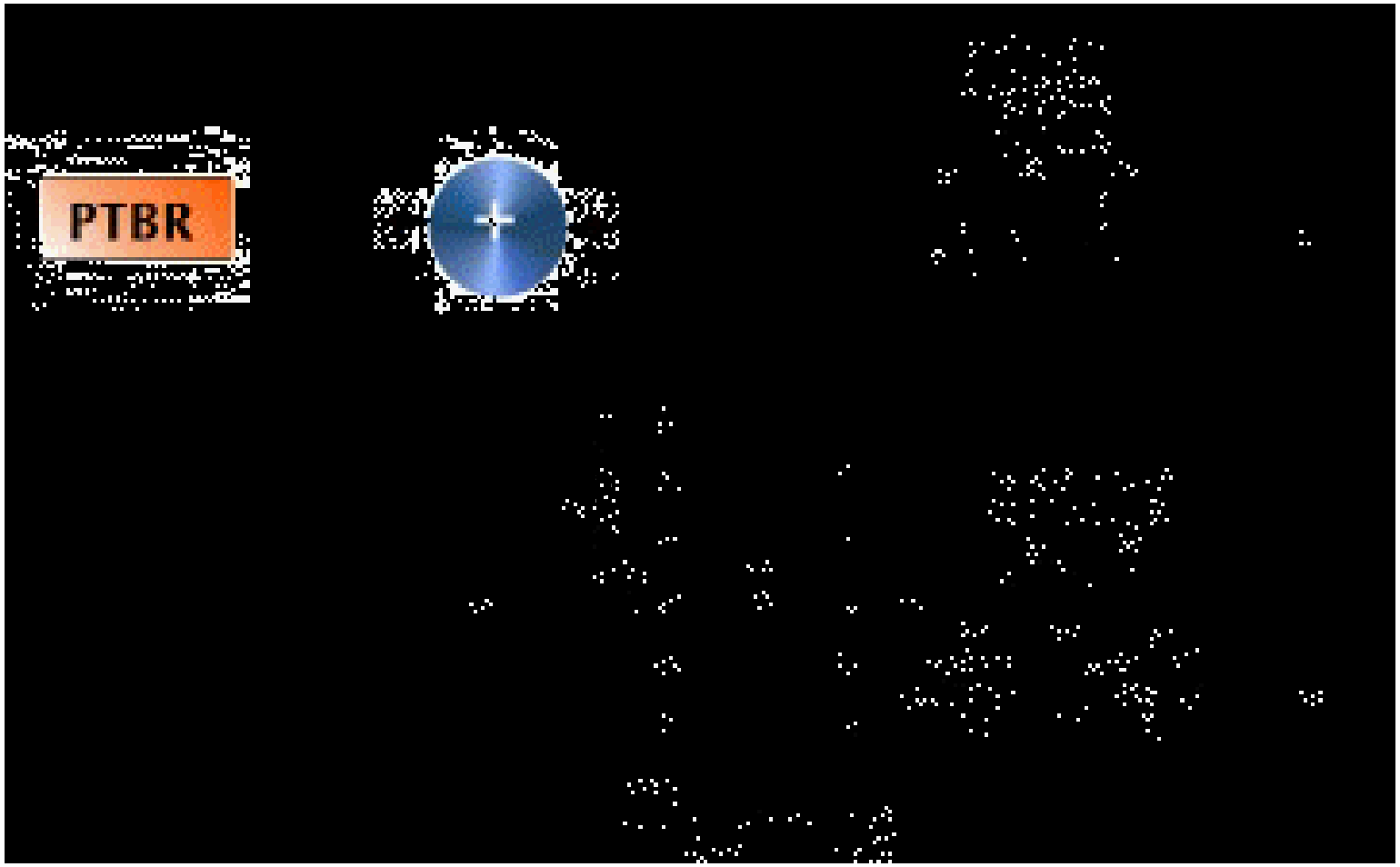
Trước khi và sau khi cấp phát cho Process mới

B) Cài đặt bảng trang (Paging hardware)

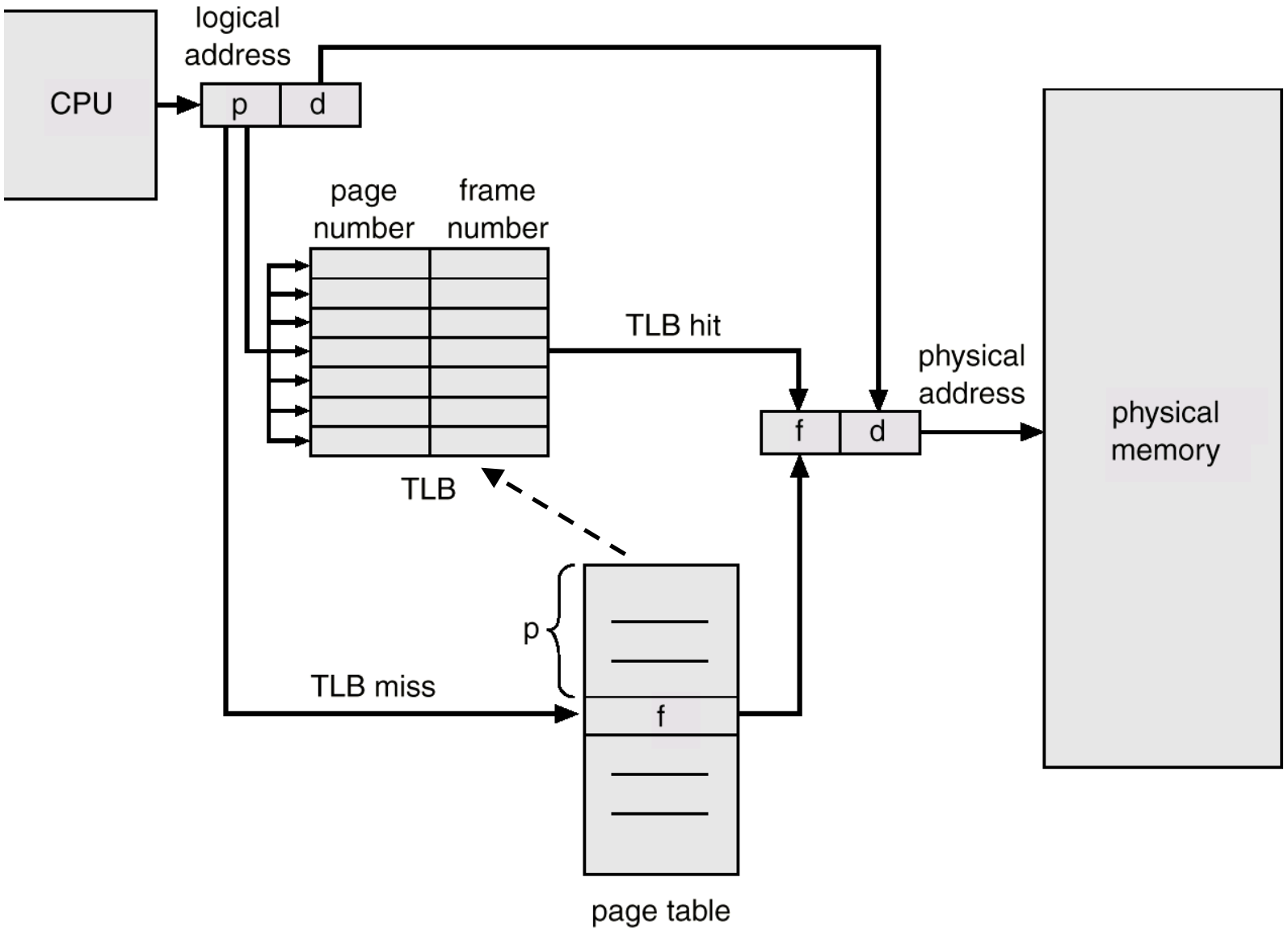
- Bảng phân trang thường được lưu giữ trong bộ nhớ chính
  - Mỗi process được hệ điều hành cấp một bảng phân trang
  - Thanh ghi *page-table base* (PTBR) trỏ đến bảng phân trang
  - Thanh ghi *page-table length* (PTLR) biểu thị kích thước của bảng phân trang (có thể được dùng trong cơ chế bảo vệ bộ nhớ)
- Thường dùng một bộ phận cache phần cứng có tốc độ truy xuất và tìm kiếm cao, gọi là *thanh ghi kết hợp* (associative register) hoặc *translation look-aside buffers* (TLBs)

B) Cài đặt bảng trang (Paging hardware)

- Dùng thanh ghi Page-Table Base Register (PTBR)



Paging hardware với TLB



C) Effective access time (EAT)

- Tính thời gian truy xuất hiệu dụng (effective access time, EAT)
- Thời gian tìm kiếm trong TLB (associative lookup): ε
- Thời gian một chu kỳ truy xuất bộ nhớ: *x*
- *Hit ratio*: tỉ số giữa số lần chỉ số trang được tìm thấy (hit) trong TLB và số lần truy xuất khởi nguồn từ CPU
  - Kí hiệu hit ratio: α
- Thời gian cần thiết để có được chỉ số frame
  - Khi chỉ số trang có trong TLB (hit) ε + x
  - Khi chỉ số trang không có trong TLB (miss) ε + x + x
- *Thời gian truy xuất hiệu dụng*

$$EAT = (\varepsilon + x)\alpha + (\varepsilon + 2x)(1 - \alpha)$$

$$= (2 - \alpha)x + \varepsilon$$

C) Effective access time (EAT)

- Ví dụ 1: đơn vị thời gian nano giây
  - Associative lookup = 20
  - Memory access = 100
  - Hit ratio = 0.8
  - EAT = (100 + 20) × 0.8 + (200 + 20) × 0.2
    = 1.2 × 100 + 20
    = 140

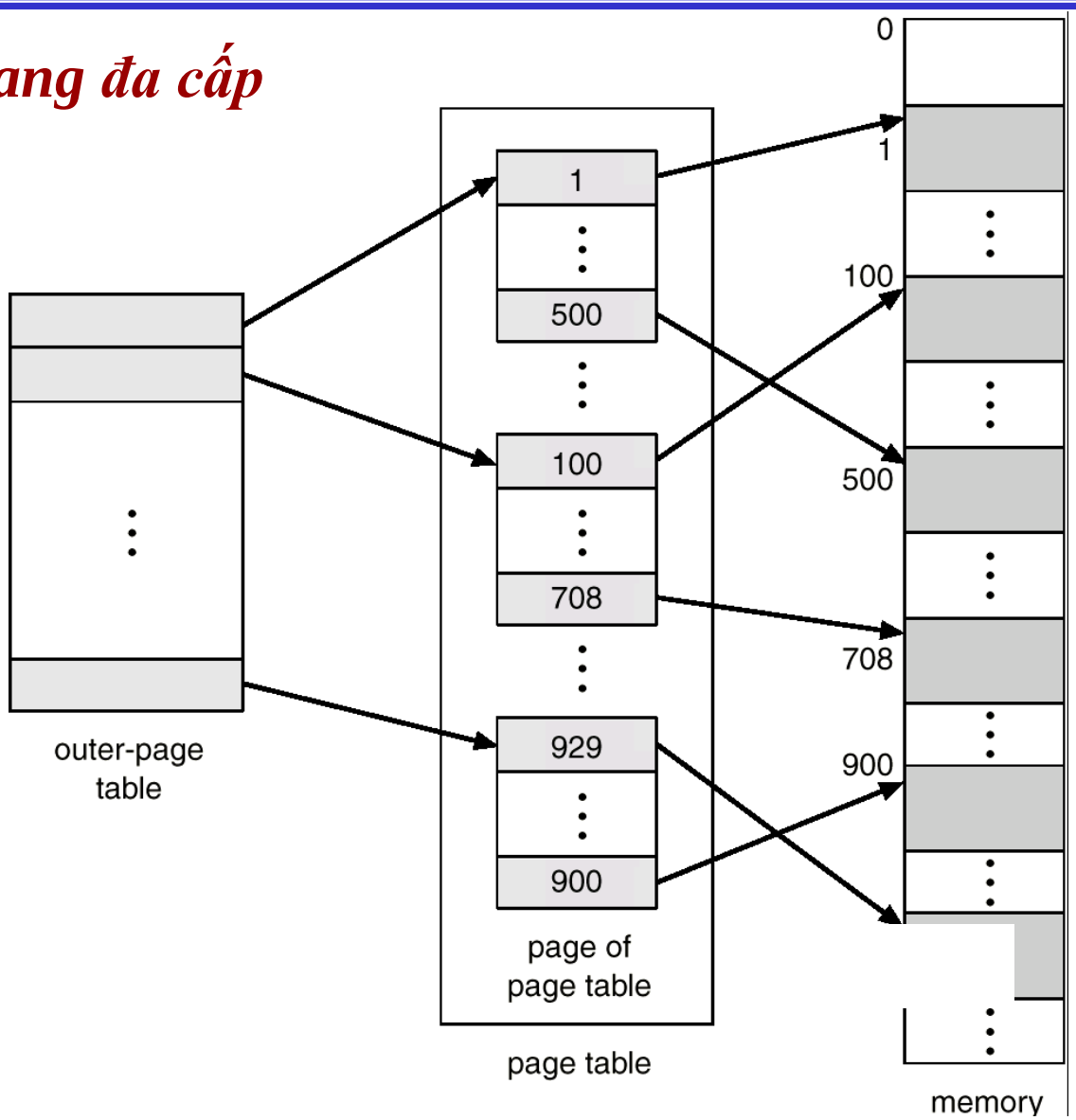
- Ví dụ 2
  - Associative lookup = 20
  - Memory access = 100
  - Hit ratio = 0.98
  - EAT = (100 + 20) × 0.98 + (200 + 20) × 0.02
    = 1.02 × 100 + 20
    = 122

D) Tổ chức bảng trang - *Phân trang đa cấp*

- Các hệ thống hiện đại đều hỗ trợ không gian địa chỉ ảo rất lớn (2^{32} đến 2^{64}), ở đây giả sử là 2^{32}
  - Giả sử kích thước trang nhớ là 4KB ($= 2^{12}$) $\Rightarrow$ bảng phân trang sẽ có $2^{32}/2^{12} = 2^{20} =$ 1M mục.
  - Giả sử mỗi mục gồm 4 byte thì mỗi process cần 4MB cho bảng phân trang

D) Tổ chức bảng trang

Phân trang đa cấp



D) Tổ chức bảng trang

- Bảng trang nghịch đảo: sử dụng cho tất cả các Process

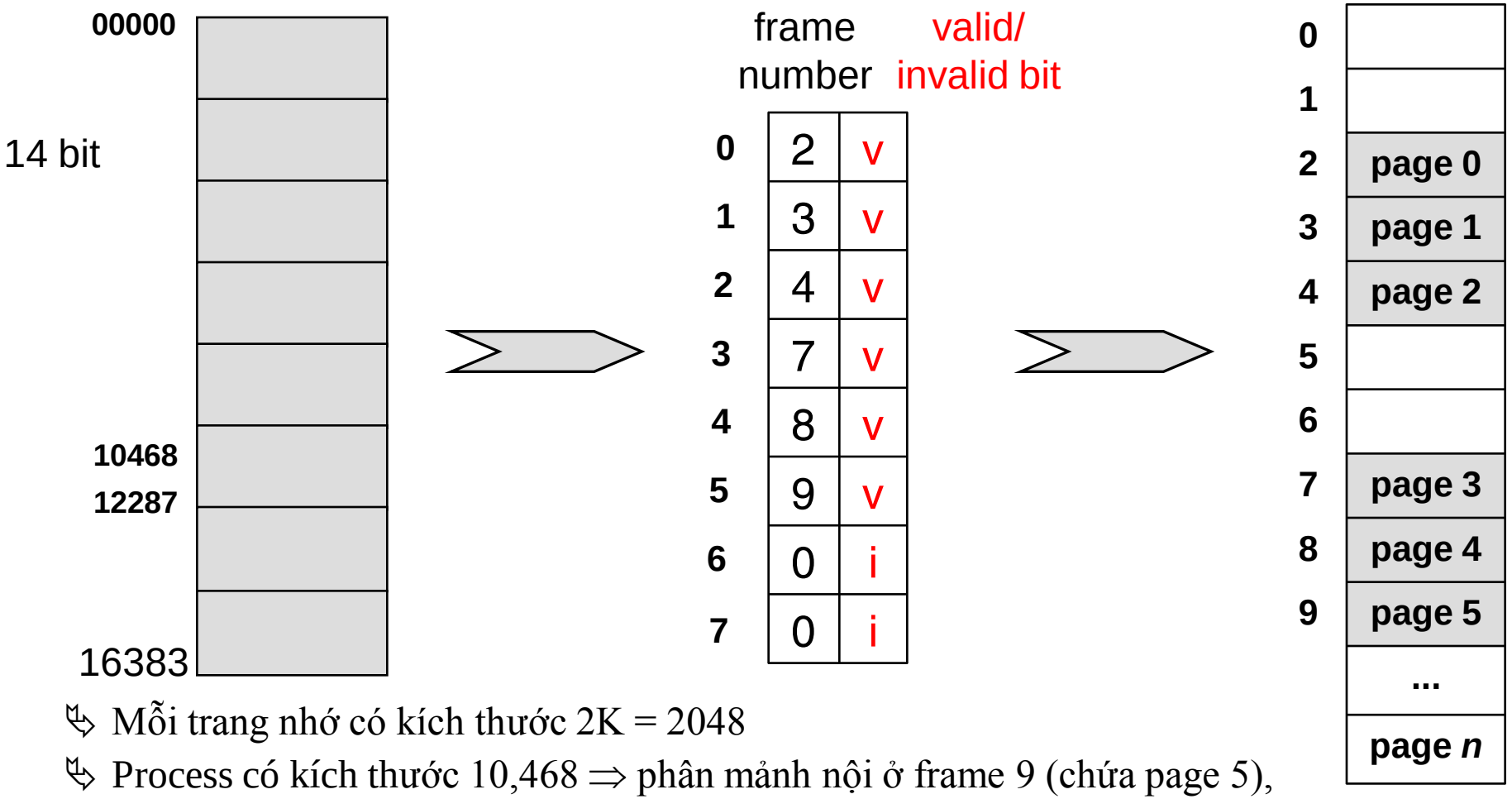
<IDP,p,d>

E) Bảo vệ bộ nhớ

- Việc bảo vệ bộ nhớ được hiện thực bằng cách gắn với frame các *bit bảo vệ* (protection bits) được giữ trong bảng phân trang. Các bit này biểu thị các thuộc tính sau
  - read-only, read-write, execute-only

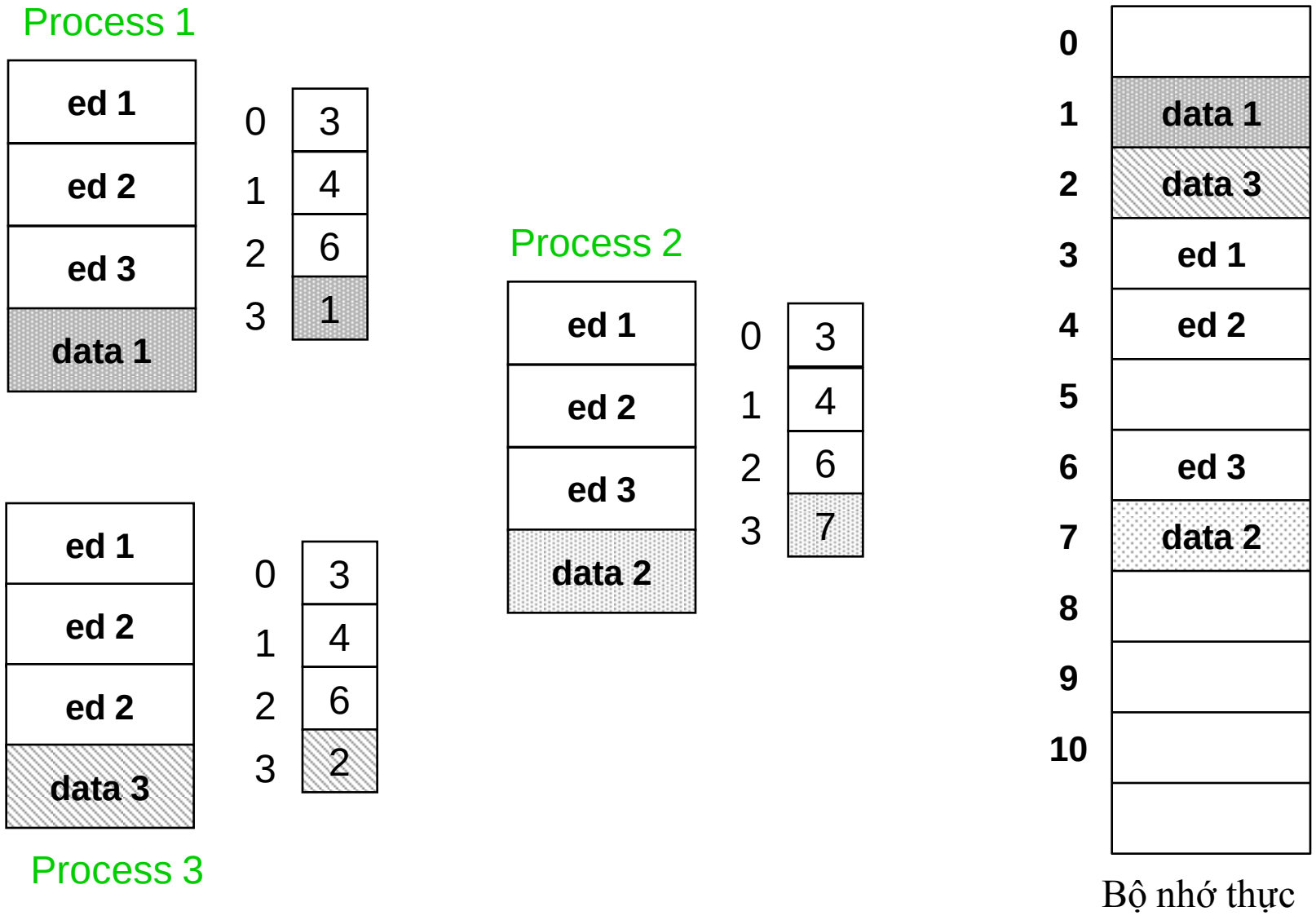
- Ngoài ra, còn có một *valid/invalid bit* gắn với mỗi mục trong bảng phân trang
  - "valid": cho biết là trang của process, do đó là một trang hợp lệ.
  - "invalid": cho biết là trang không của process, do đó là một trang bất hợp lệ.

Bảo vệ bằng valid/invalid bit



- Mỗi trang nhớ có kích thước 2K = 2048
- Process có kích thước 10,468 ⇒ phân mảnh nội ở frame 9 (chứa page 5), các địa chỉ ảo > 12287 là các địa chỉ invalid.
- Dùng PTLR để kiểm tra truy xuất đến bảng phân trang có nằm trong bảng hay không.

F) Chia sẻ các trang nhớ



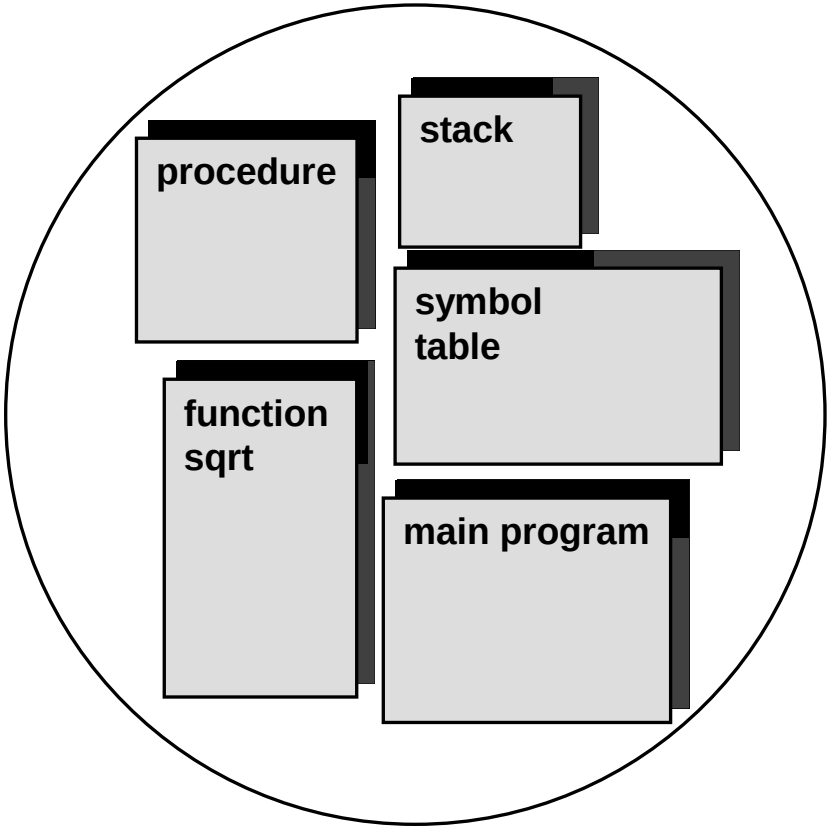
2.Phân đoạn (segmentation)

- Nhìn lại cơ chế phân trang
  - user view (không gian địa chỉ ảo) tách biệt với không gian bộ nhớ thực. Cơ chế phân trang thực hiện phép ánh xạ user-view vào bộ nhớ thực.

- Trong thực tế, dưới góc nhìn của user, một chương trình cấu thành từ nhiều *đoạn* (segment). Mỗi đoạn là một đơn vị luận lý của chương trình, như
  - main program, procedure, function
  - local variables, global variables, common block, stack, symbol table, arrays,…

User view của một chương trình

- Thông thường, một chương trình được biên dịch. Trình biên dịch sẽ tự động xây dựng các segment.
- Ví dụ, trình biên dịch Pascal sẽ tạo ra các segment sau:
  - Global variables
  - Procedure call stack
  - Procedure/function code
  - Local variable
- Trình loader sẽ gán mỗi segment một số định danh riêng.

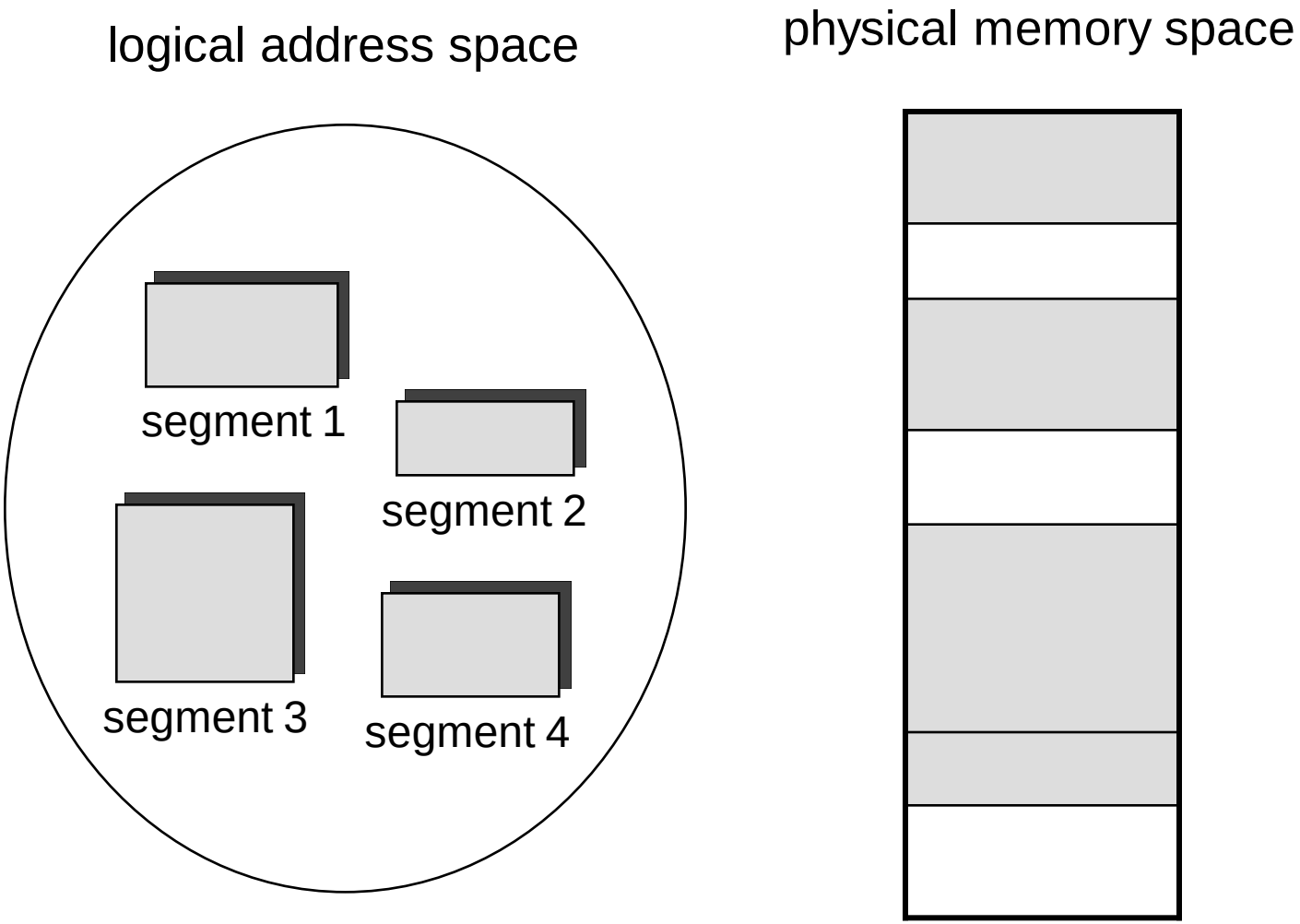


Logical address space

Phân đoạn

- Dùng cơ chế *phân đoạn* để quản lý bộ nhớ có hỗ trợ user view
  - *Không gian địa chỉ ảo* là một tập các đoạn, mỗi đoạn có tên và kích thước riêng.
  - Một địa chỉ luận lý được định vị bằng tên đoạn và độ dời (offset) bên trong đoạn đó (so sánh với phân trang!)

Phân đoạn (tt)


logical address space
physical memory space
segment 1
segment 2
segment 3
segment 4

Cài đặt phân đoạn

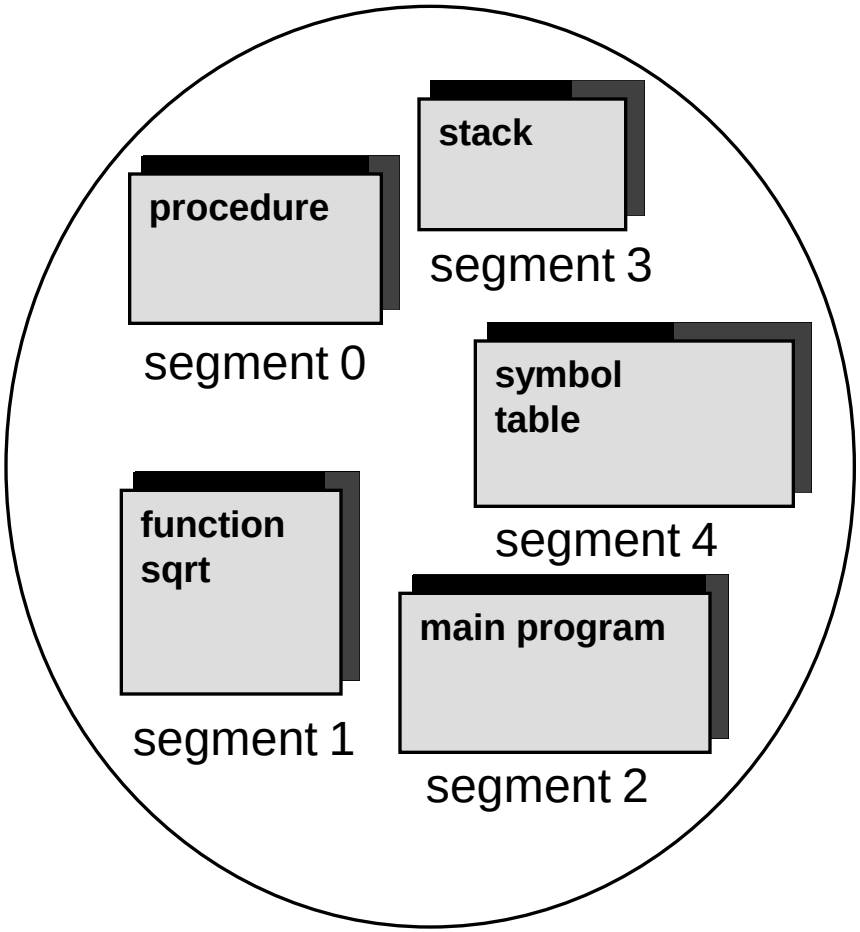
- *Địa chỉ luận lý* là một cặp giá trị

  (segment number, offset)

- *Bảng phân đoạn* (segment table): gồm nhiều mục, mỗi mục chứa
  - base, chứa địa chỉ khởi đầu của segment trong bộ nhớ
  - limit, xác định kích thước của segment
- *Segment-table base register* (STBR): trỏ đến vị trí bảng phân đoạn trong bộ nhớ
- *Segment-table length register* (STLR): số lượng segment của chương trình

  ⇒ Một chỉ số segment s là hợp lệ nếu s < STLR

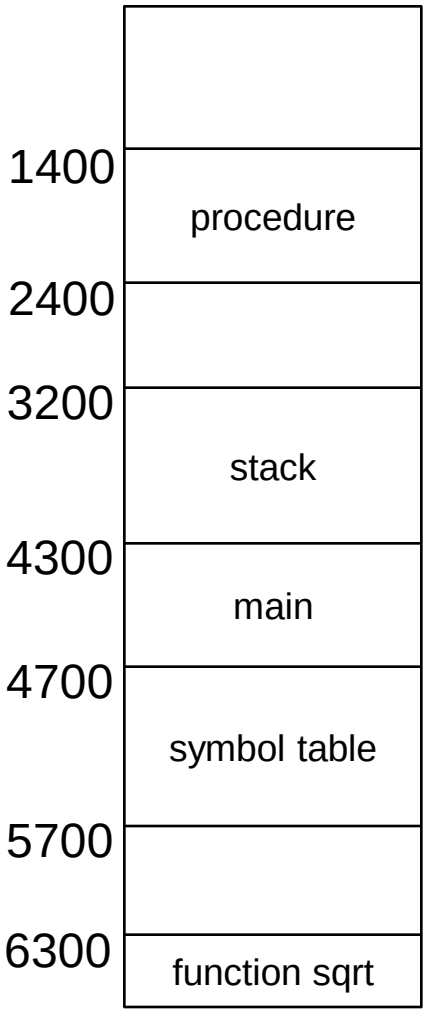
Một ví dụ về phân đoạn



logical address space

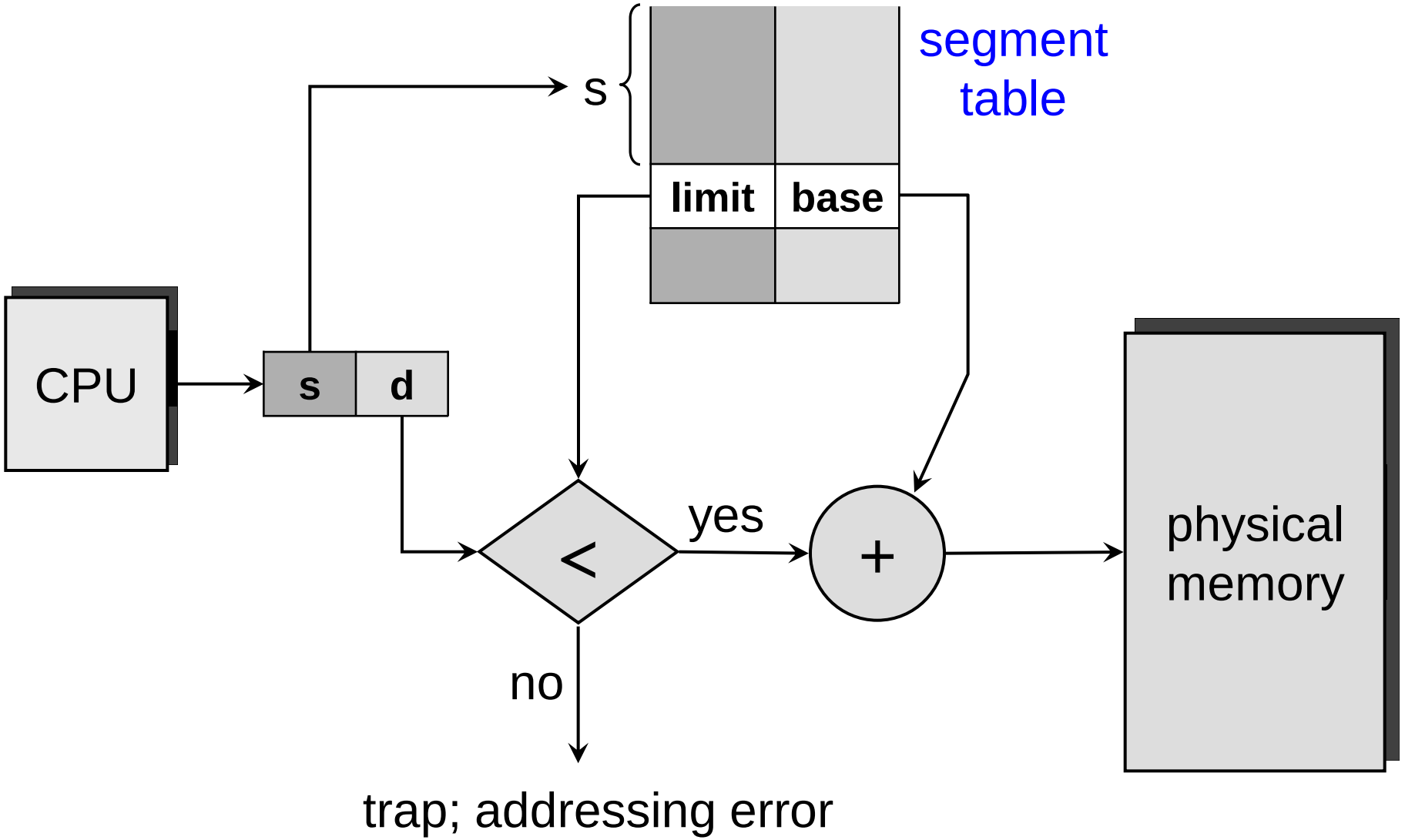
	limit	base
0	1000	1400
1	400	6300
2	400	4300
3	1100	3200
4	1000	4700

segment table



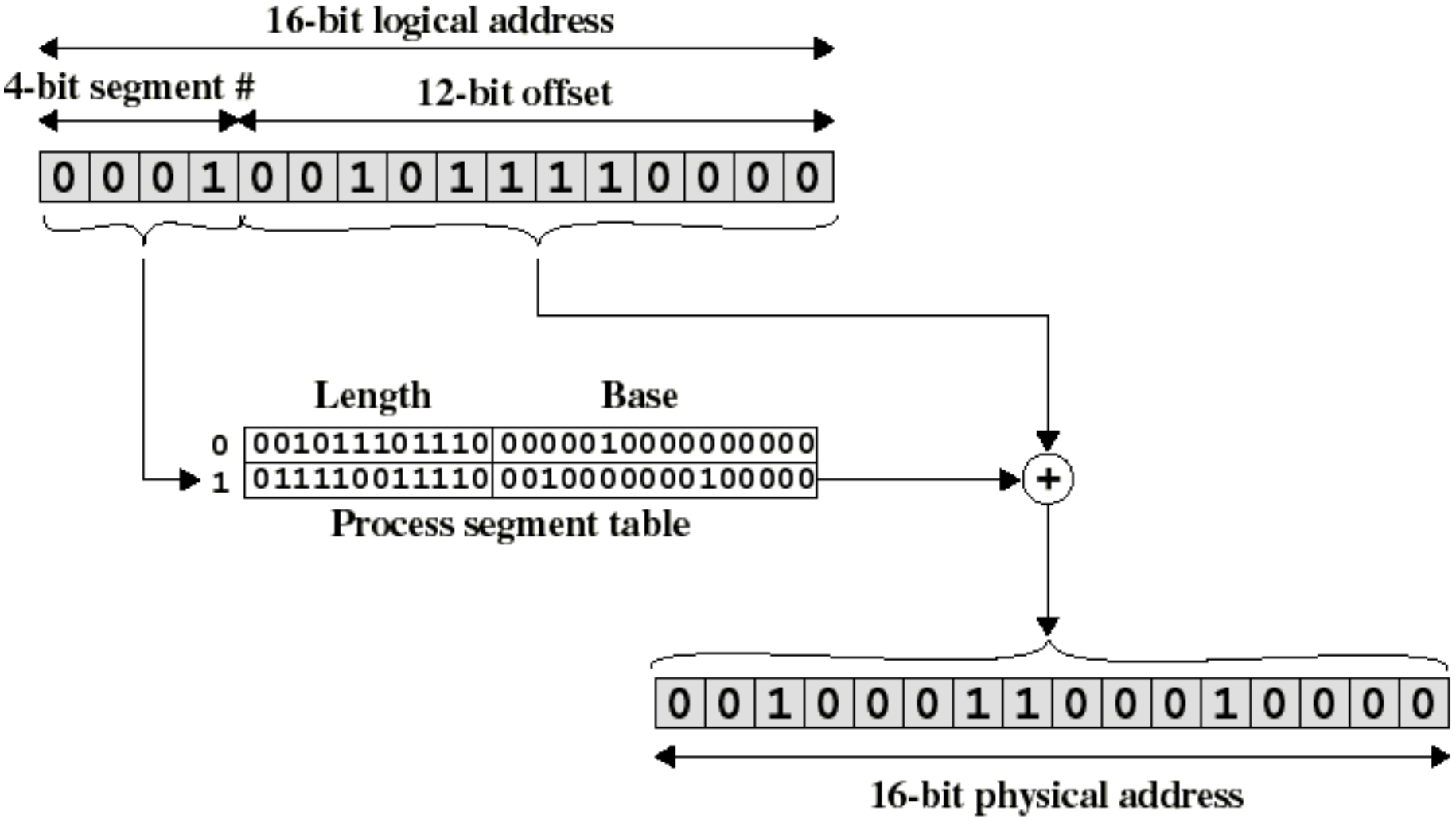
physical memory space

Phần cứng hỗ trợ phân đoạn

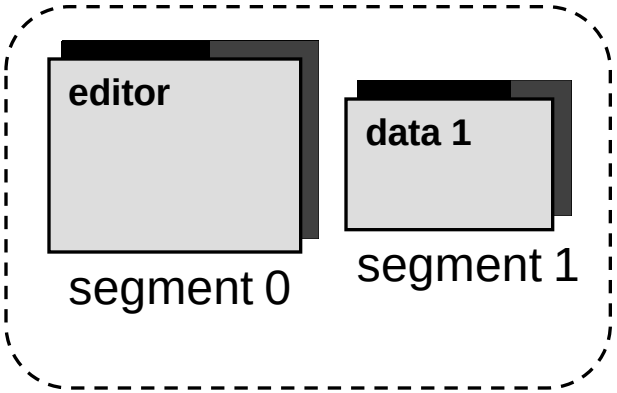


Chuyển đổi địa chỉ trong cơ chế phân đoạn

Ví dụ



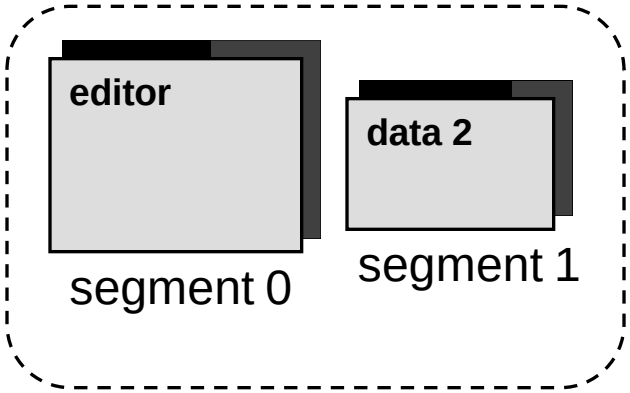
Chia sẻ các đoạn



logical address space
process P_1

	limit	base
0	25286	43062
1	4425	68348

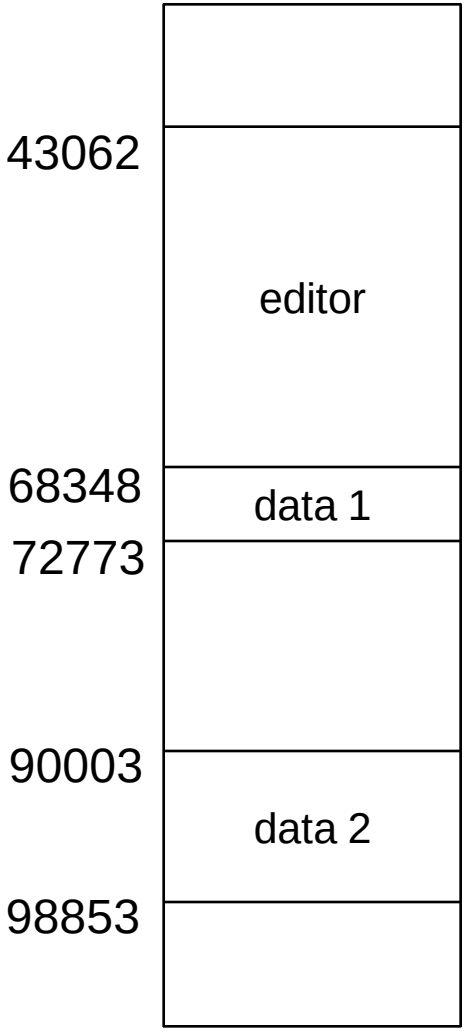
segment table
process P_1



logical address space
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2



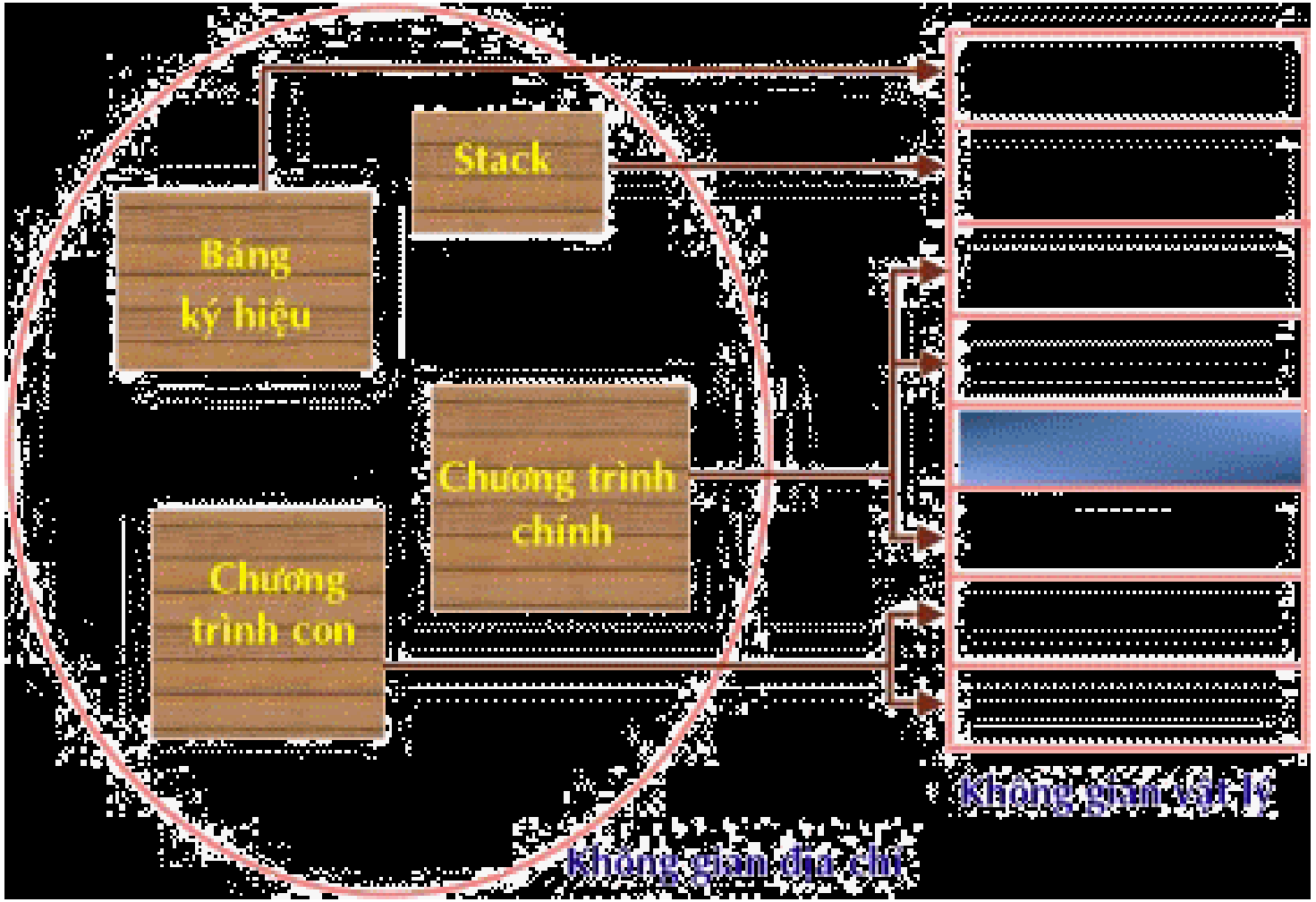
physical memory

3.Kết hợp phân trang và phân đoạn

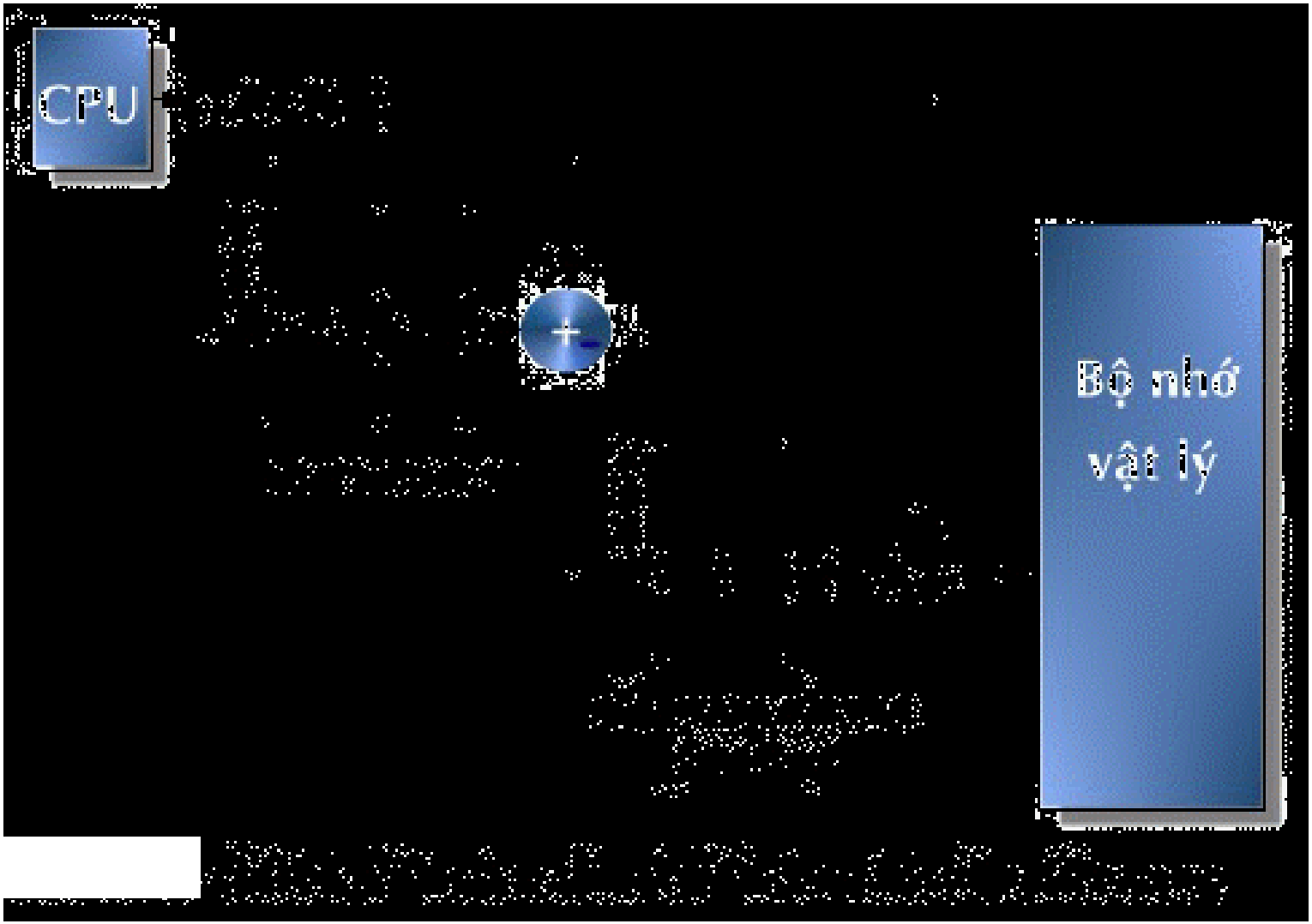
- Kết hợp phân trang và phân đoạn nhằm kết hợp các ưu điểm đồng thời hạn chế các khuyết điểm của phân trang và phân đoạn:
  - Vấn đề của phân đoạn: Nếu một đoạn quá lớn thì có thể không nạp nó được vào bộ nhớ.
  - Ý tưởng giải quyết: paging đoạn, khi đó chỉ cần giữ trong bộ nhớ các page của đoạn hiện đang cần.

Logic Addr = <s,p,d>

3.Kết hợp phân trang và phân đoạn



3.Kết hợp phân trang và phân đoạn



Chương 8

Bộ Nhớ Ảo

Nội dung trình bày

- Tổng quan về bộ nhớ ảo
- Cài đặt bộ nhớ ảo : demand paging
- Cài đặt bộ nhớ ảo : Page Replacement
  - Các giải thuật thay trang (Page Replacement Algorithms)
- Vấn đề cấp phát Frames
- Vấn đề Thrashing
- Cài đặt bộ bộ nhớ ảo : Demand Segmentation

1. Tổng quan bộ nhớ ảo

- Nhận xét: không phải tất cả các phần của một process cần thiết phải được nạp vào bộ nhớ chính tại cùng một thời điểm
- Ví dụ
  - Đoạn mã điều khiển các lỗi hiếm khi xảy ra
  - Các arrays, list, tables được cấp phát bộ nhớ (cấp phát tĩnh) nhiều hơn yêu cầu thực sự
  - Một số tính năng ít khi được dùng của một chương trình
  - Cả chương trình thì cũng có đoạn code chưa cần dùng
- *Bộ nhớ ảo* (virtual memory): Bộ nhớ ảo là một kỹ thuật cho phép xử lý một tiến trình không được nạp toàn bộ vào bộ nhớ vật lý

1. Bộ nhớ ảo (tt)

Ưu điểm của bộ nhớ ảo

- Số lượng process trong bộ nhớ nhiều hơn
- Một process có thể thực thi ngay cả khi kích thước của nó lớn hơn bộ nhớ thực
- Giảm nhẹ công việc của lập trình viên

- *Không gian tráo đổi* giữa bộ nhớ chính và bộ nhớ phụ(swap space).
- Ví dụ:
  - swap partition trong Linux
  - file pagefile.sys trong Windows

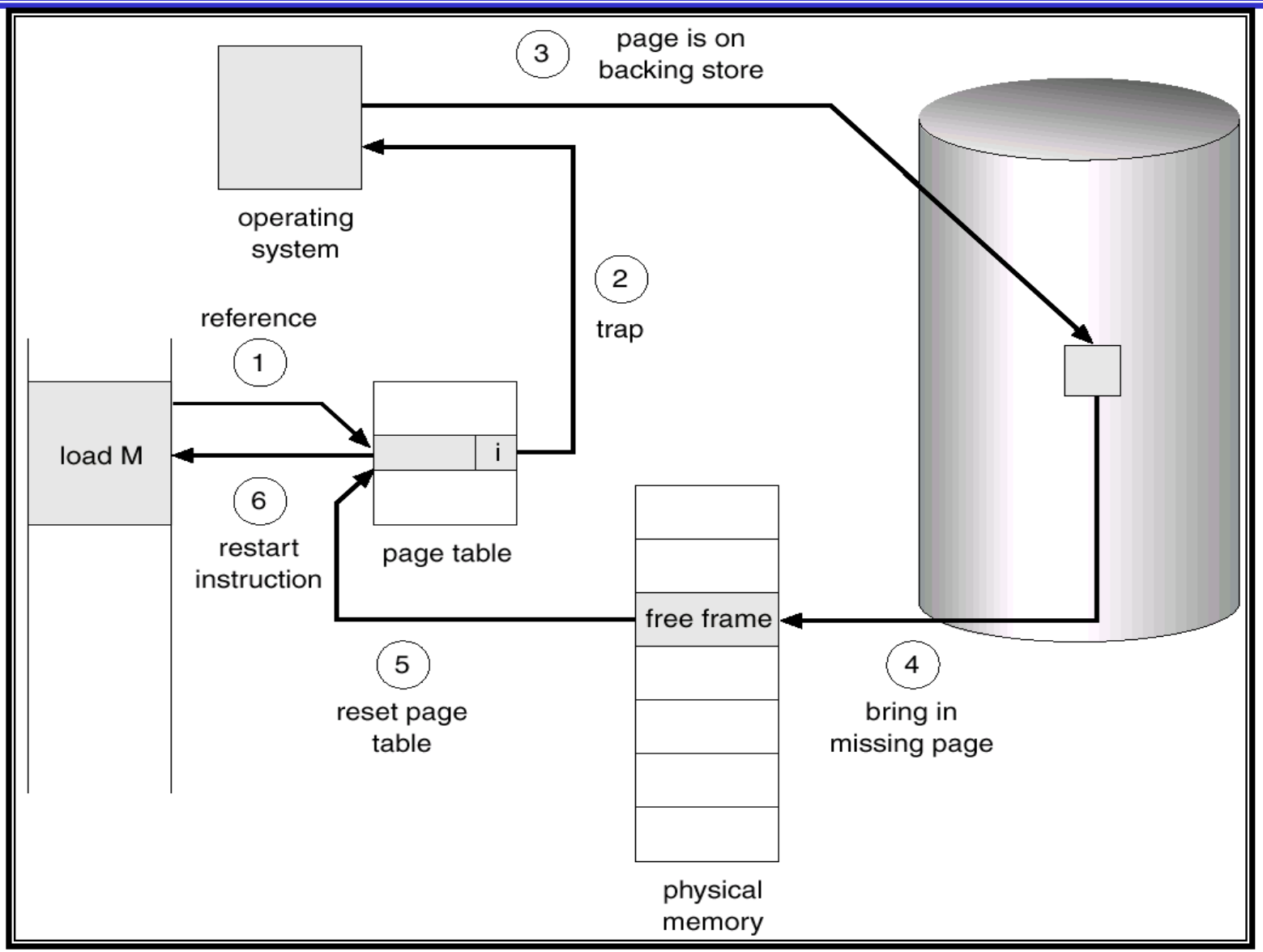
2. Cài đặt bộ nhớ ảo

- Có hai kỹ thuật:
  - Phân trang theo yêu cầu (Demand Paging)
  - Phân đoạn theo yêu cầu (Segmentation Paging)
- Phần cứng memory management phải hỗ trợ paging và/hoặc segmentation
- OS phải quản lý sự di chuyển của trang/đoạn giữa bộ nhớ chính và bộ nhớ thứ cấp
- Trong chương này,
  - Chỉ quan tâm đến paging
  - Phần cứng hỗ trợ hiện thực bộ nhớ ảo
  - Các giải thuật của hệ điều hành

2.1.Phân trang theo yêu cầu demand paging

- *Demand paging*: các trang của quá trình chỉ được nạp vào bộ nhớ chính khi được yêu cầu.
- Khi có một tham chiếu đến một trang mà không có trong bộ nhớ chính (valid bit) thì phần cứng sẽ gây ra một ngắt (gọi là *page-fault trap*) kích khởi *page-fault service routine* (PFSR) của hệ điều hành.
- PFSR:
  1. Chuyển process về trạng thái blocked
  2. Phát ra một yêu cầu đọc đĩa để nạp trang được tham chiếu vào một frame trống; trong khi đợi I/O, một process khác được cấp CPU để thực thi
  3. Sau khi I/O hoàn tất, đĩa gây ra một ngắt đến hệ điều hành; PFSR cập nhật page table và chuyển process về trạng thái ready.

2.2. Lỗi trang và các bước xử lý

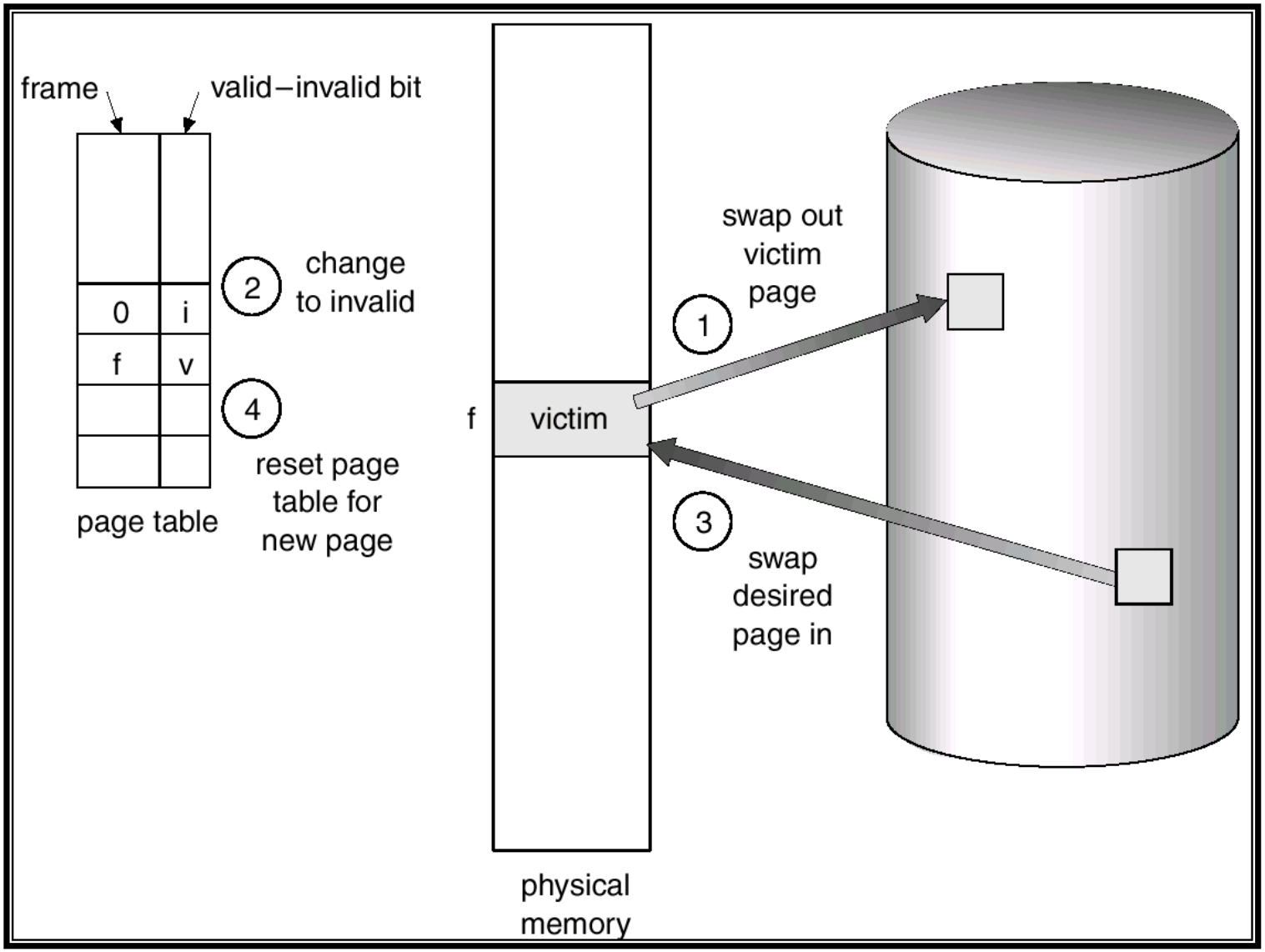


2.3. Thay thế trang nhớ

- Bước 2 của PFSR giả sử phải *thay trang* vì không tìm được frame trống, PFSR được bổ sung như sau

1. Xác định vị trí trên đĩa của trang đang cần
2. Tìm một frame trống:
   a. Nếu có frame trống thì dùng nó
   b. Nếu không có frame trống thì dùng một giải thuật thay trang để chọn một *trang hy sinh* (victim page)
   c. Ghi victim page lên đĩa; cập nhật page table và frame table tương ứng
3. Đọc trang đang cần vào frame trống (đã có được từ bước 2); cập nhật page table và frame table tương ứng.

2.3. Thay thế trang nhớ (tt)ù



2.4. Các thuật toán thay thế trang

- Hai vấn đề chủ yếu:
- Frame-allocation algorithm
  - Cấp phát cho process bao nhiêu frame của bộ nhớ thực?
- Page-replacement algorithm
  - Chọn frame của process sẽ được thay thế trang nhớ
  - Mục tiêu: số lượng page-fault nhỏ nhất
  - Được đánh giá bằng cách thực thi giải thuật đối với một *chuỗi tham chiếu bộ nhớ* (memory reference string) và xác định số lần xảy ra page fault

- Ví dụ
- Thứ tự tham chiếu các địa chỉ nhớ, với page size = 100:
  - **01**00, **04**32, **01**01, **06**12, **01**02, **01**03, **01**04, **01**01, **06**11, **01**02, **01**03, **01**04, **01**01, **06**10, **01**02, **01**03, **01**04, **01**01, **06**09, **01**02, **01**05
- [redacted]ác trang nhớ sau được tham chiếu lần lượt = chuỗi tham chiếu bộ nhớ (trang nhớ)
  - **1, 4, 1, 6, 1,**
  - **1, 1, 1, 6, 1,**
  - **1, 1, 1, 6, 1,**
  - **1, 1, 1, 6, 1,**
  - **1**

a) Giải thuật thay trang *FIFO*

- Các dữ liệu cần biết ban đầu:
  - Số khung trang
  - Tình trạng ban đầu
  - Chuỗi tham chiếu

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

Nghịch lý *Belady*

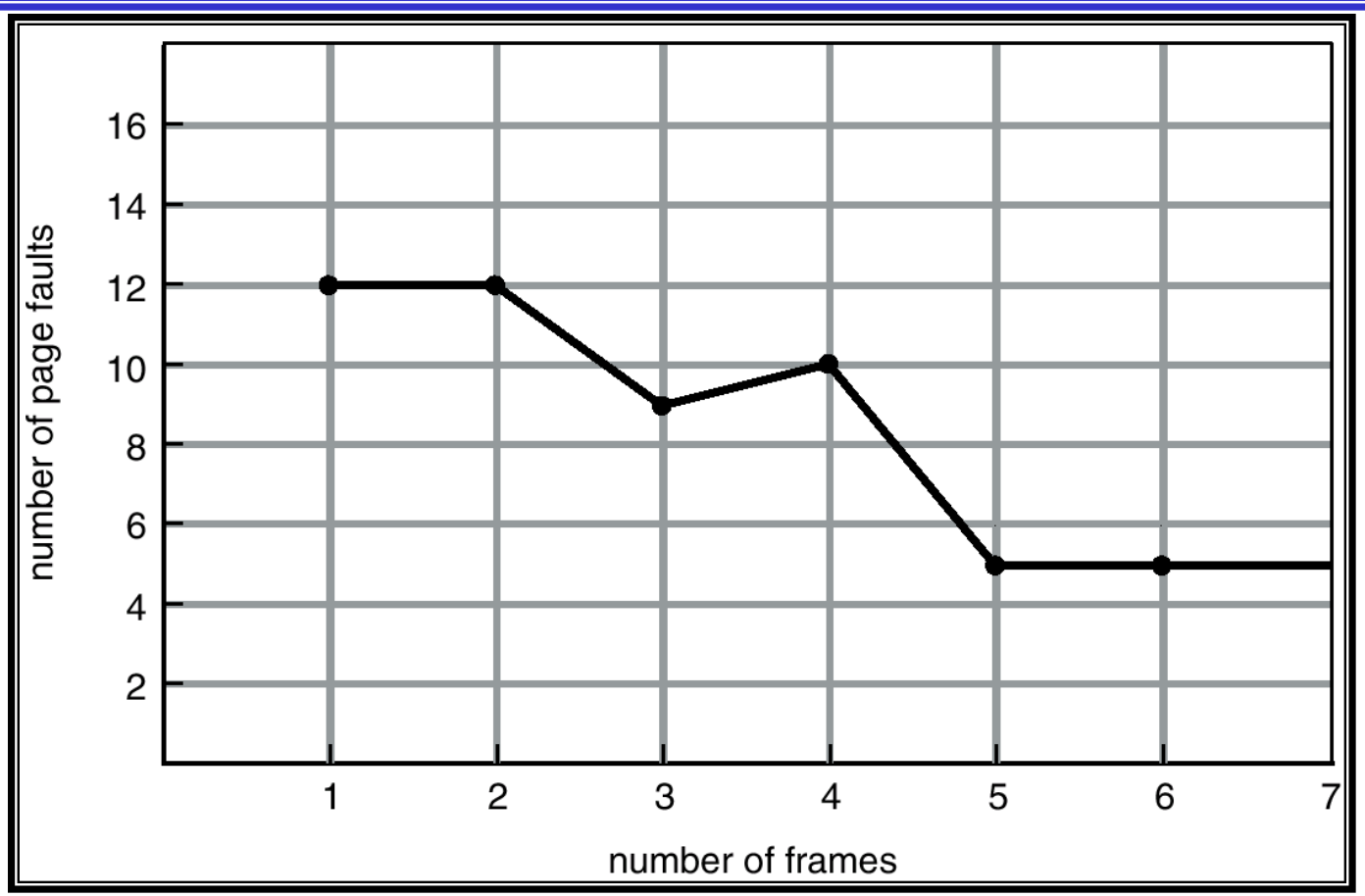
Sử dụng 3 khung trang , sẽ có 9 lỗi trang phát sinh

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	4	4	4	5	5	5	5	5	5
	2	2	2	1	1	1	1	1	3	3	3
		3	3	3	2	2	2	2	2	4	4
*	*	*	*	*	*	*			*	*	

Sử dụng 4 khung trang , sẽ có 10 lỗi trang phát sinh

1	2	3	4	1	2	5	1	2	3	4	5
1	1	1	1	1	1	5	5	5	5	4	4
	2	2	2	2	2	2	1	1	1	1	5
		3	3	3	3	3	3	2	2	2	2
			4	4	4	4	4	4	3	3	3
*	*	*	*			*	*	*	*	*	*

Nghịch lý *Belady*



Bất thường (anomaly) Belady: số page fault tăng mặc dầu quá trình đã được cấp nhiều frame hơn.

2.4 b)Giải thuật thay trang *OPT(optimal)*

- Giải thuật thay trang OPT
  - Thay thế trang nhớ sẽ được tham chiếu trễ nhất trong tương lai
- Ví dụ: một process có 7 trang, và được cấp 3 frame

sử dụng 3 khung trang, khởi đầu đều trống:

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

c) Giải thuật lâu nhất chưa sử dụng
Least Recently Used (LRU)

- Ví dụ:

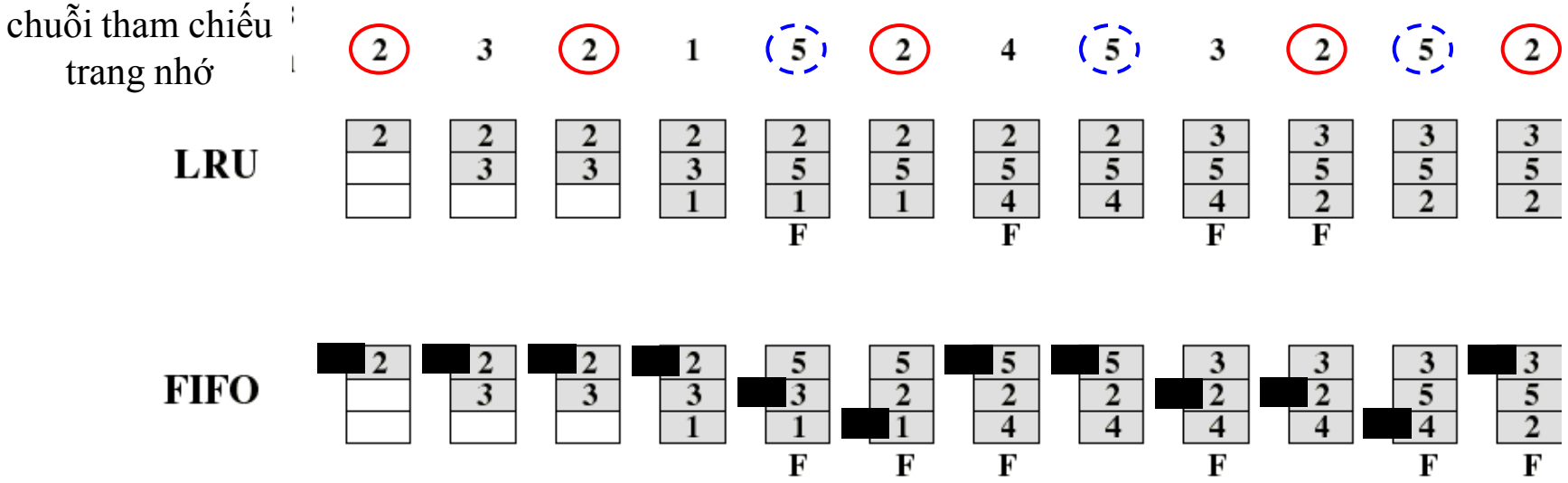
sử dụng 3 khung trang, khởi đầu đều trống:

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

- Mỗi trang được ghi nhận (trong bảng phân trang) thời điểm được tham chiếu trang LRU là trang nhớ có thời điểm tham chiếu nhỏ nhất (OS tốn chi phí tìm kiếm trang nhớ LRU này mỗi khi có page fault)
- Do vậy, LRU cần sự hỗ trợ của phần cứng và chi phí cho việc tìm kiếm. Ít CPU cung cấp đủ sự hỗ trợ phần cứng cho giải thuật LRU.

LRU và FIFO

- So sánh các giải thuật thay trang LRU và FIFO



2.5.Số lượng frame cấp cho process

- OS phải quyết định cấp cho mỗi process bao nhiêu frame.
  - Cấp ít frame [redacted]hiều page fault
  - Cấp nhiều frame [redacted]ảm mức độ multiprogramming

- Chiến lược cấp phát tĩnh (fixed-allocation)
  - Số frame cấp cho mỗi process không đổi, được xác định vào thời điểm loading và có thể tùy thuộc vào từng ứng dụng (kích thước của nó,…)
- Chiến lược cấp phát động (variable-allocation)
  - Số frame cấp cho mỗi process có thể thay đổi trong khi nó chạy
    - Nếu tỷ lệ page-fault cao [redacted]p thêm frame
    - Nếu tỷ lệ page-fault thấp [redacted]ảm bớt frame
  - OS phải mất chi phí để ước định các process

a) Chiến lược cấp phát tĩnh

- *Cấp phát bằng nhau*: Ví dụ, có 100 frame và 5 process [redacted] mỗi process được 20 frame
- *Cấp phát theo tỉ lệ*: dựa vào kích thước process

s_i [redacted] ize of process p_i

S [redacted]

m [redacted] otal number of frames

a_i [redacted] llocation for p_i [redacted] $\frac{s_i}{S}$ [redacted] m

Ví dụ:

m [redacted] 64

s_i [redacted] 10

s_2 [redacted] 127

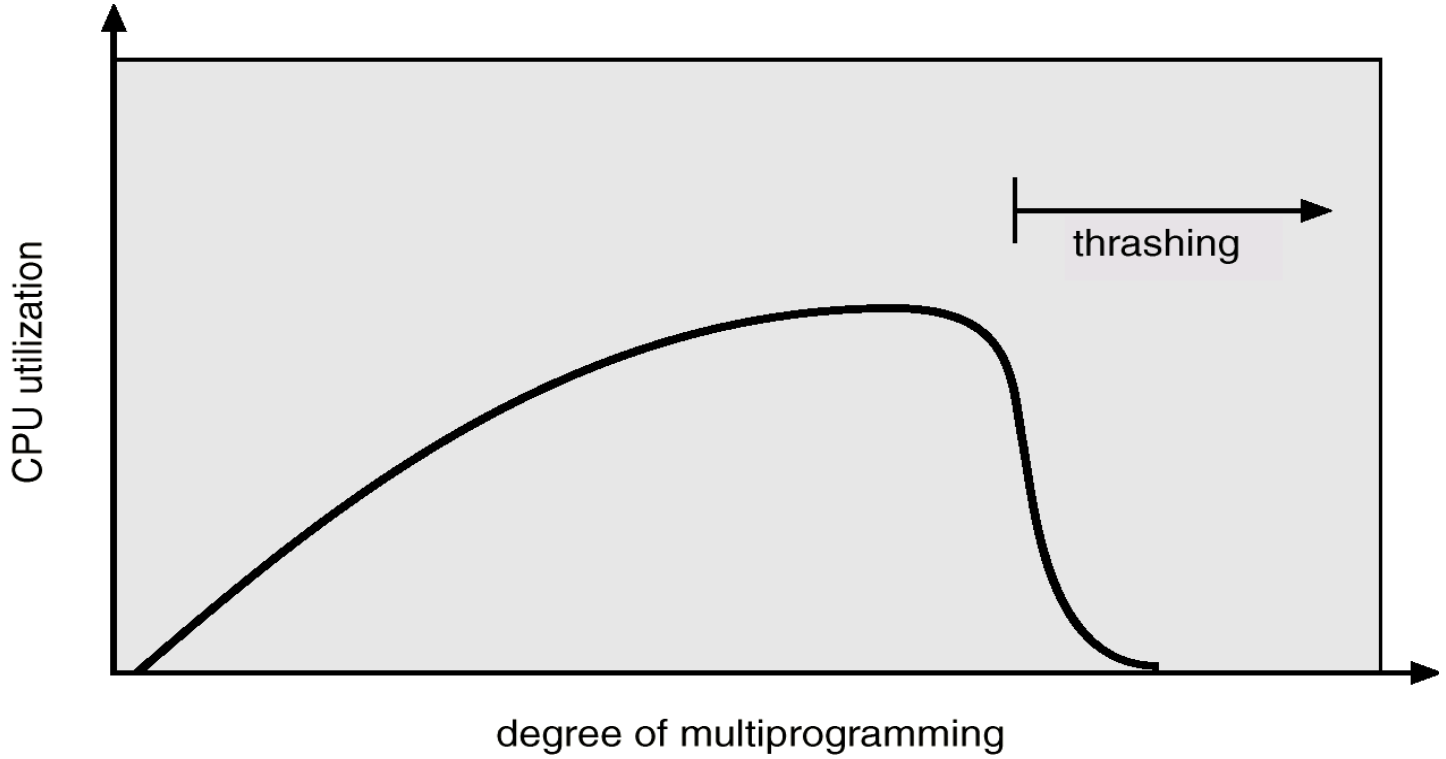
a_1 [redacted] $\frac{10}{137}$ [redacted] 64 [redacted]

a_2 [redacted] $\frac{127}{137}$ [redacted] 64 [redacted] 59

- Cấp phát theo độ ưu tiên

3. Trì trên toàn bộ hệ thống
Thrashing

- Nếu một process không có đủ số frame cần thiết thì tỉ số page faults/sec rất cao.
- *Thrashing*: hiện tượng các trang nhớ của một process bị hoán chuyển vào/ra liên tục.



a)Mô hình cục bộ (Locality)

- Để hạn chế thrashing, hệ điều hành phải cung cấp cho process càng "đủ" frame càng tốt. Bao nhiêu frame thì đủ cho một process thực thi hiệu quả?

  Nguyên lý locality (locality principle)

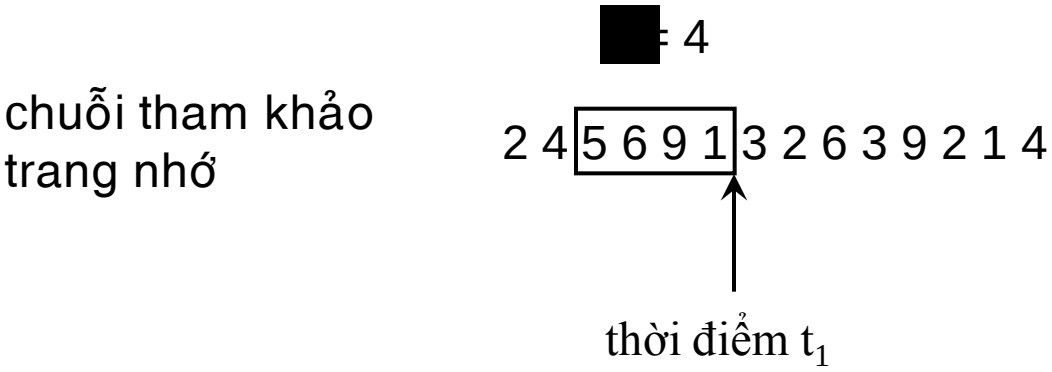
  - *Locality* là tập các trang được tham chiếu gần nhau
  - Một process gồm nhiều locality, và trong quá trình thực thi, process sẽ chuyển từ locality này sang locality khác

- Vì sao hiện tượng thrashing xuất hiện?

  Khi [redacted]ze of locality > memory size

b) Giải pháp tập làm việc (working set)

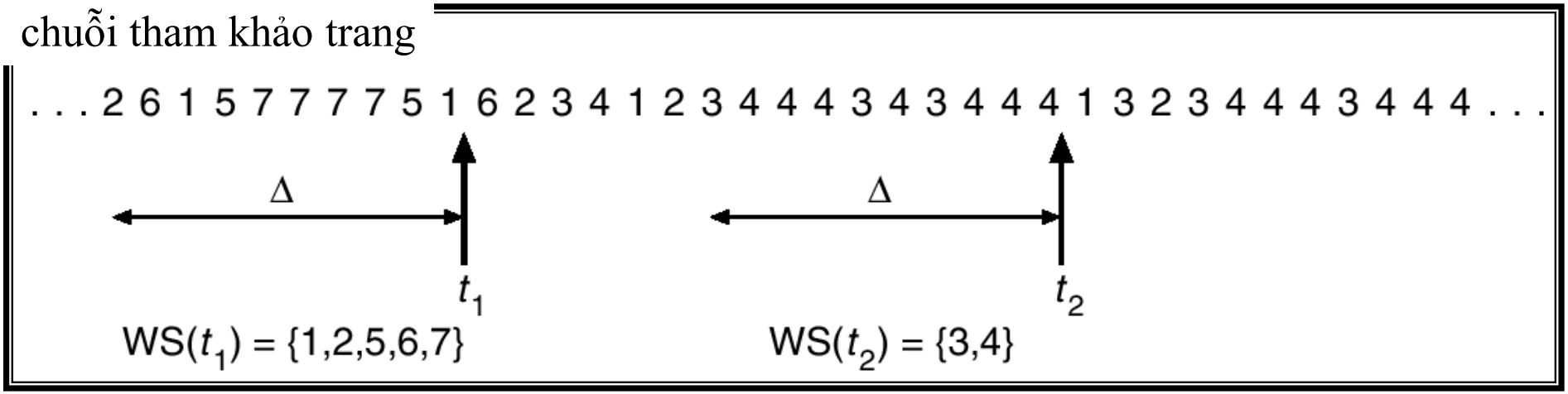
- Được thiết kế dựa trên nguyên lý locality.
- Xác định xem process thực sự sử dụng bao nhiêu frame.
- Định nghĩa:
  - WS(t) - số lượng các tham chiếu trang nhớ của process gần đây nhất cần được quan sát.
  - △ - khoảng thời gian tham chiếu
- Ví dụ:



b) Giải pháp tập làm việc (working set)

- Định nghĩa: *working set của process P_i* , ký hiệu WS_i , là tập gồm [redacted]ác trang được sử dụng gần đây nhất.

Ví dụ: [redacted]= 10 và chuỗi tham khảo trang



- Nhận xét:
  - [redacted]uá nhỏ [redacted]hông đủ bao phủ toàn bộ locality.
  - [redacted]uá lớn [redacted]ao phủ nhiều locality khác nhau.
  - [redacted]= [redacted] [redacted]ao gồm tất cả các trang được sử dụng.

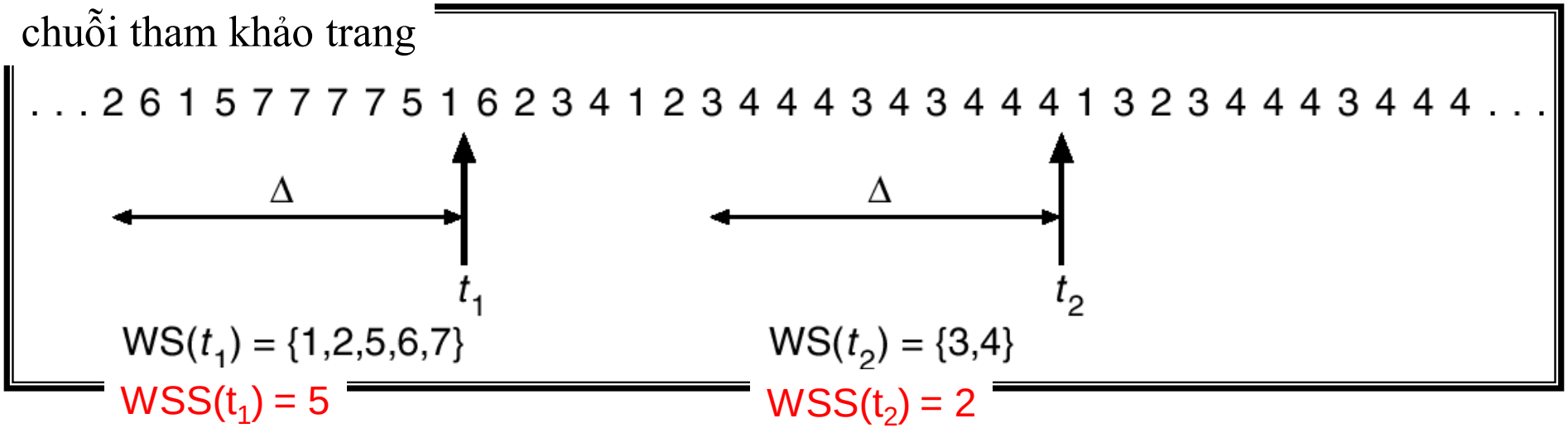
  Dùng working set của một process để xấp xỉ locality của nó.

b) Giải pháp tập làm việc (working set)

Định nghĩa WSS_i là kích thước của working set của P_i :

WSS_i = số lượng các trang trong WS_i

Ví dụ (tiếp): [redacted] = 10 và chuỗi tham khảo trang



b) Giải pháp tập làm việc (working set)

- Đặt D = [redacted]WSS_i = tổng các working-set size của mọi process trong hệ thống.
  - Nhận xét: Nếu D > m (số frame của hệ thống) [redacted] xảy ra thrashing.

- *Giải pháp working set*:
  - Khi khởi tạo một quá trình: cung cấp cho quá trình số lượng frame thỏa mản working-set size của nó.
  - Nếu D > m [redacted]m dừng một trong các process.
    - Các trang của quá trình được chuyển ra đĩa cứng và các frame của nó được thu hồi.

b) Giải pháp tập làm việc (working set)

- WS loại trừ được tình trạng trì trệ mà vẫn đảm bảo mức độ đa chương
- Theo vết các WS? => WS xấp xỉ (đọc thêm trong sách)

Đọc thêm:

- Hệ thống tập tin
- Hệ thống nhập xuất
- Hệ thống phân tán