Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Mục tiêu của môn học

- Cung cấp những khái niệm cơ bản về hệ điều hành máy tính: phân loại, nguyên lý, cách làm việc, phân tích thiết kế và chi tiết về một số hệ điều hành cụ thể
- Yêu cầu sinh viên: Nắm vững các nguyên lý cơ bản, làm tốt các bài tập để lấy đó làm cơ sở - nguyên lý cho các vấn đề khác trong thiết kế và cài đặt các hệ thống thông tin
- Chú ý liên hệ nội dung môn học với các tình huống thực tế về khía cạnh quản lý, tổ chức

2

Nội dung

- Gồm có 6 phần chính:
  - Tổng quan (3 tiết)
  - Quản lý tiến trình (12 tiết)
  - Quản lý lưu trữ (12 tiết)
  - Hệ vào/ra (9 tiết)
  - Bảo vệ và an ninh (6 tiết)
  - Hệ điều hành Linux (*optional*) + Ôn tập (3 tiết)

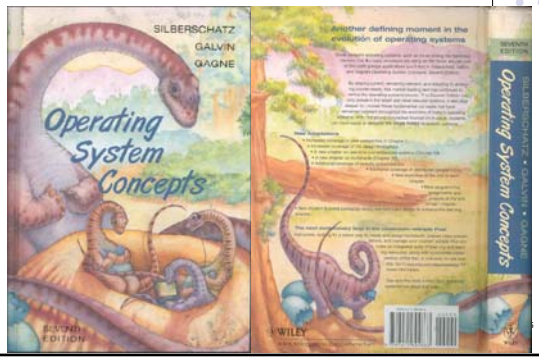
3

Tài liệu tham khảo

- **Abraham Silberschatz, Peter Baer Galvin, Greg Gagne, Operating System Concepts, 7th edition, John Wiley & Sons, Inc., 2005.**
- William Stallings, Operating Systems: Internals and Design Principles 5th edition, Prentice-Hall, 2005.
- Andrew S. Tanenbaum, Modern Operating Systems, 2nd edition, Prentice-Hall, 2001.
- Andrew S. Tanenbaum, Albert S Woodhull, Operating Systems: Design and Implementation, 3rd edition, Prentice-Hall. 2006. (Có mã nguồn kèm theo).
- Hà Quang Thụy, Nguyên lý hệ điều hành, NXB KHKT, 2002.
- Robert Love, Linux Kernel Development, Sams Publishing, 2003.
- Daniel P. Bovet, Marco Cesati, Understanding Linux Kernel, 2nd edition, O'Reilly & Associates, 2002.
- W. Richard Stevens, Advanced Programming in the UNIX Environment, Addison-Wesley, 1992.

4

Giáo trình



5

Bản điện tử của giáo trình

- Website của Bộ môn Các hệ thống thông tin: http://coltech.vnu.edu.vn/httt
- Chọn "Góc học tập" ở menu bên trái
- Chọn "Nguyên lý hệ điều hành" ở phần nội dung chính của trang web
- Download sách theo chỉ dẫn

6

Thi và kiểm tra

- Kiểm tra giữa kỳ: viết, 45-60 phút
  - Là điều kiện bắt buộc để được thi cuối kỳ
  - Sau phần quản lý bộ nhớ/lưu trữ
  - Được sử dụng tài liệu
- Thi cuối kỳ:
  - Thi viết 60-90 phút
  - Được sử dụng tài liệu

7

Giới thiệu

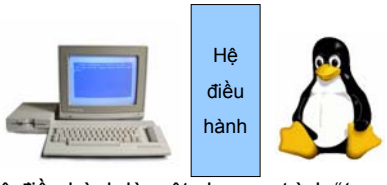
8

Máy tính - tài nguyên máy tính

- Tài nguyên:
  - CPU
  - Bộ nhớ trong
  - Đĩa cứng
  - Thiết bị ngoại vi (máy in, màn hình, bàn phím, card giao tiếp mạng, USB...)

9

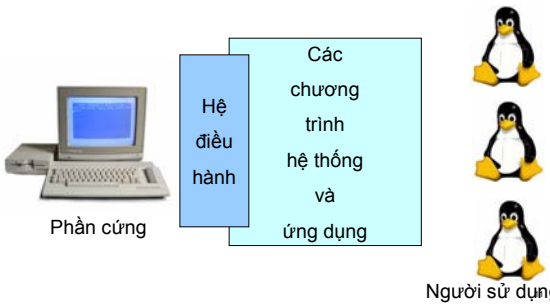
Hệ điều hành là gì?



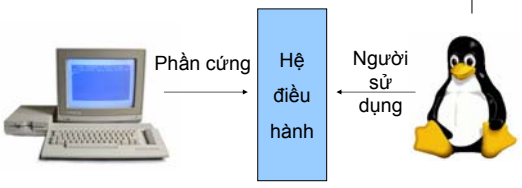
- Hệ điều hành là một chương trình "*trung gian*" (nhân – kernel) giữa NSD và máy tính :
  - *Quản lý* phần cứng máy tính (các tài nguyên)
  - Cung cấp cho NSD môi trường làm việc *tiện lợi* và *hiệu quả*

10

Hệ thống máy tính



Hai cách nhìn hệ điều hành



- Phần cứng: Quản lý & cấp phát tài nguyên để sử dụng tối đa năng lực phần cứng
- Người sử dụng: Dễ sử dụng, hiệu quả, ứng dụng phong phú

12

Một số loại hệ điều hành

- Xử lý theo lô (batch processing)
- Đa chương trình (multiprogramming)
- Phân chia thời gian (time-sharing/multitasking)
- Hệ điều hành cho máy cá nhân
- Xử lý song song (parallel)
- Thời gian thực (real-time)
- Nhúng (embedded)
- Cầm tay (portable)
- Đa phương tiện (multimedia)
- Chuyên dụng (special-purpose)

13

Các hệ xử lý theo lô đơn giản

- Thuật ngữ: *Batch processing*
- Các chương trình được đưa vào hàng chờ
- Máy tính thực hiện tuần tự các chương trình của người sử dụng
- Chương trình không có giao tiếp với người sử dụng

14

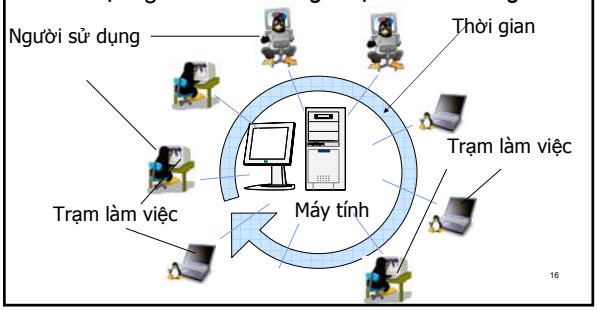
Đa chương trình

- Thuật ngữ: *Multiprogramming*
- Các chương trình được xếp hàng
- Một chương trình được thực hiện và chiếm giữ CPU cho đến khi (1) có yêu cầu vào/ra, hoặc (2) kết thúc
- Khi (1) hoặc (2) xảy ra, chương trình khác sẽ được thực hiện
- Tận dụng CPU tốt hơn xử lý theo lô đơn giản

15

Phân chia thời gian/đa nhiệm

- Thuật ngữ: *time-sharing* hoặc *multitasking*



16

Một số hệ điều hành

- UNIX (UNiplexed Information and Computing Service): (1) AT&T System V (2) Berkeley (BSD)
  - AIX dựa trên System V (IBM)
  - HP-UX dựa trên BSD (Hewlett-Packard)
  - IRIX dựa trên System V (Silicon Graphics Inc.)
  - Linux
  - Solaris, SunOS (Sun Microsystems)
  - Minix

17

Một số hệ điều hành

- Windows (Microsoft): Windows 3.x, Windows 95, Windows 98, Windows 2000, Windows NT, Windows XP, Windows Vista
- Mac OS, Mac OS X (Apple Inc.)
- BeOS
- OS 9
- OS/2
- DOS
- PalmOS, Symbian

18

Cấu trúc hệ điều hành

19

Các thành phần của hệ thống

- Quản lý tiến trình
- Quản lý bộ nhớ trong
- Quản lý tệp
- Quản lý vào/ra
- Quản lý lưu trữ trên bộ nhớ ngoài
- Liên kết mạng
- Bảo vệ và an ninh
- Thông dịch lệnh

20

Các dịch vụ của hệ điều hành

- Giao diện với người sử dụng
- Thực hiện các chương trình
- Thực hiện các thao tác vào/ra
- Quản lý hệ thống tệp
- Truyền thông
- Phát hiện lỗi
- Cấp phát tài nguyên
- "Kế toán"
- Đưa ra các cơ chế bảo vệ và an ninh

21

Các hàm hệ thống

- Các *hàm hệ thống* (system calls) cung cấp giao diện lập trình tới các dịch vụ do hệ điều hành cung cấp
- Ví dụ trong hệ điều hành Unix:
  - Tạo một tiến trình mới: **fork();**
  - Thoát khỏi tiến trình đang thực hiện: **exit(1);**
  - **fork** và **exit** là các hàm hệ thống (Hàm HT)

22

Hàm HT điều khiển tiến trình

- Kết thúc tiến trình bình thường/bất thường
- Nạp, thực hiện tiến trình
- Tạo, kết thúc tiến trình
- Đọc hoặc thiết lập các thuộc tính cho tiến trình
- Yêu cầu tiến trình vào trạng thái chờ
- Cấp phát và giải phóng bộ nhớ
- Xử lý các sự kiện không đồng bộ

23

Hàm HT quản trị tệp

- Tạo, xóa tệp
- Đóng, mở tệp
- Đọc, ghi, định vị con trỏ tệp
- Đọc, thiết lập thuộc tính của tệp

24

Hàm HT quản trị thiết bị

- Yêu cầu sử dụng hoặc thôi sử dụng thiết bị
- Đọc, ghi, định vị con trỏ
- Đọc, thiết lập thuộc tính cho thiết bị
- Attach/detach thiết bị về mặt logic

25

Hàm HT bảo trì thông tin

- Đọc, thiết lập thời gian hệ thống
- Đọc, ghi dữ liệu về hệ thống
- Đọc thuộc tính tệp, thiết bị, tiến trình
- Thiết lập thuộc tính tệp, thiết bị, tiến trình

26

Hàm HT về truyền thông

- Tạo, hủy các kết nối mạng
- Truyền nhận các thông điệp
- Lấy thông tin trạng thái truyền thông
- Attach/detach các thiết bị ở xa

27

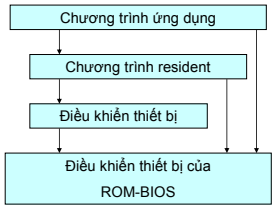
Các chương trình hệ thống

- Các chương trình hệ thống cung cấp môi trường thuận tiện cho việc thực hiện và phát triển chương trình. Chúng được phân loại như sau:
  - Thao tác với tệp
  - Thông tin về trạng thái của hệ thống
  - Sửa đổi tệp
  - Hỗ trợ ngôn ngữ lập trình
  - Nạp và thực hiện chương trình
  - Truyền thông
  - Cách nhìn HĐH của NSD được xác định qua các chương trình hệ thống, không thực sự qua các hàm hệ thống (system calls).

28

Cấu trúc HĐH: Đơn giản

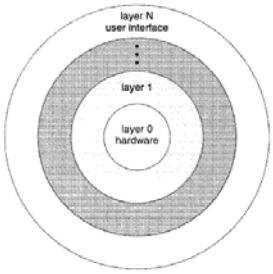
- Thuật ngữ: Simple approach
- Ví dụ MS-DOS. (tương tự: UNIX thời gian đầu)



29

Cấu trúc HĐH: Phân tầng

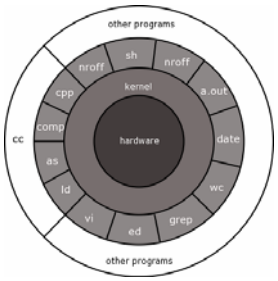
- Thuật ngữ: Layered apparoach



30

Cấu trúc HĐH: Phân tầng

- Ví dụ UNIX



31

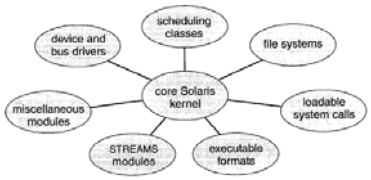
Cấu trúc HĐH: Vi nhân

- Thuật ngữ: Microkernel
- Giữ cho nhân có các đủ các chức năng thiết yếu nhất để giảm cỡ
- Các chức năng khác được đưa ra ngoài nhân
- Ví dụ: Mach, Tru64 UNIX, QNX

32

Cấu trúc HĐH: Module

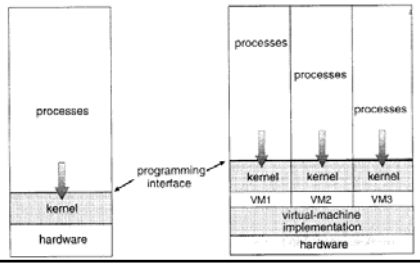
- Thuật ngữ: Module approach
- Hiện tại đây là cách tiếp cận tốt nhất (sử dụng được các kỹ thuật lập trình hướng đối tượng). Ví dụ: Solaris của Sun Microsystem:



33

Máy ảo

- Thuật ngữ (Virtual Machine)
- Ví dụ: VMware (sản phẩm thương mại)



Tóm tắt

- Khái niệm HĐH, nhân
- Hai cách nhìn HĐH từ NSD và hệ thống
- Các khái niệm xử lý theo lô, đa chương trình và phân chia thời gian
- Các thành phần và dịch vụ của HĐH
- Các hàm hệ thống
- Một số cấu trúc phổ biến của HĐH
- Máy ảo

35

Tìm hiểu thêm

- **Không bắt buộc**
- Bổ sung một hàm hệ thống mới vào nhân Linux và sử dụng hàm đó:
  - Đọc hướng dẫn trong giáo trình từ trang 74-78
  - Thử nghiệm trên RedHat Fedora hoặc Ubuntu/Debian

36

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

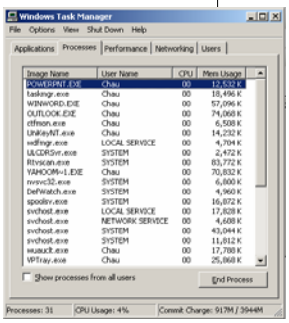
1

Khái niệm tiến trình

2

Tiến trình là gì?

- Thuật ngữ: Process (tiến trình/quá trình)
- Là một *chương trình đang được thực hiện*
- Được xem là đơn vị làm việc trong các HĐH
- Có hai loại tiến trình:
  - Tiến trình của HĐH
  - Tiến trình của NSD

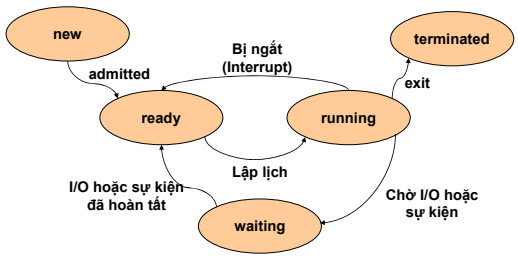


Tiến trình gồm có…

- Đoạn mã lệnh (code, có sách gọi là text)
- Đoạn dữ liệu
- Đoạn ngăn xếp và heap (stack/heap)
- Các *hoạt động hiện tại* được thể hiện qua con đếm lệnh (IP) và nội dung các thanh ghi (registers) của bộ xử lý
- Chú ý:
  - Tiến trình là *thực thể chủ động*
  - Chương trình là *thực thể bị động*

4

Trạng thái tiến trình



5

Khối điều khiển tiến trình

- **Thuật ngữ: Process Control Block (PCB)**
- **Các thông tin:**
  - **Trạng thái tiến trình**
  - **Con đếm**
  - **Các thanh ghi**
  - **Thông tin về lập lịch**
  - **Thông tin về bộ nhớ**
  - **Thông tin accounting**
  - **Thông tin vào/ra**

Con trỏ	Trạng thái tiến trình
Số hiệu tiến trình (Process number)	
Con đếm (program counter)	
Các thanh ghi (registers)	
Giới hạn bộ nhớ	
Danh sách các tệp đang mở	
.....	

6

Lập lịch tiến trình

7

Tại sao phải lập lịch?

- Số lượng NSD, số lượng tiến trình luôn lớn hơn số lượng CPU của máy tính rất nhiều
- Tại một thời điểm, chỉ có duy nhất một tiến trình được thực hiện trên một CPU
- Vấn đề:
  - Số lượng yêu cầu sử dụng nhiều hơn số lượng tài nguyên đang có (CPU)
  - Do đó cần lập lịch để phân phối thời gian sử dụng CPU cho các tiến trình của NSD và hệ thống

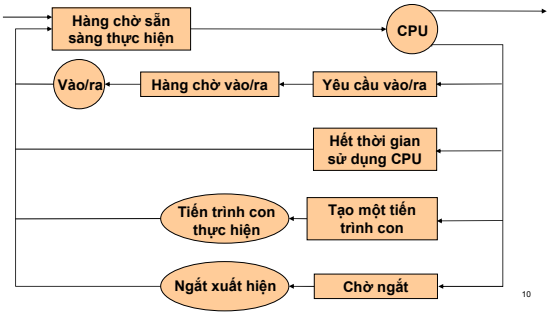
8

Hàng chờ lập lịch

- Thuật ngữ: Queue
- Các tiến trình chưa được phân phối sử dụng CPU sẽ được đưa vào *hàng chờ* (*queue*)
- Có thể có nhiều hàng chờ trong hệ thống: Hàng chờ sử dụng CPU, hàng chờ sử dụng máy in, hàng chờ sử dụng ổ đĩa CD…
- Trong suốt thời gian tồn tại, tiến trình phải di chuyển giữa các hàng chờ

9

Hàng chờ lập lịch tiến trình



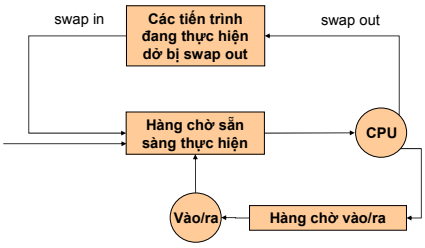
10

Phân loại các bộ lập lịch

- *Bộ lập lịch dài hạn* (long-term scheduler)
  - Thường dùng trong các hệ xử lý theo lô
  - Đưa tiến trình từ spool vào bộ nhớ trong
- *Bộ lập lịch ngắn hạn* (short-term scheduler)
  - Còn gọi là bộ lập lịch CPU
  - Lựa chọn tiến trình tiếp theo được sử dụng CPU
- *Bộ lập lịch trung hạn* (medium-term scheduler)
  - Hay còn gọi là swapping (tráo đổi)
  - Di chuyển tiến trình đang trong trạng thái chờ giữa bộ nhớ trong và bộ nhớ ngoài

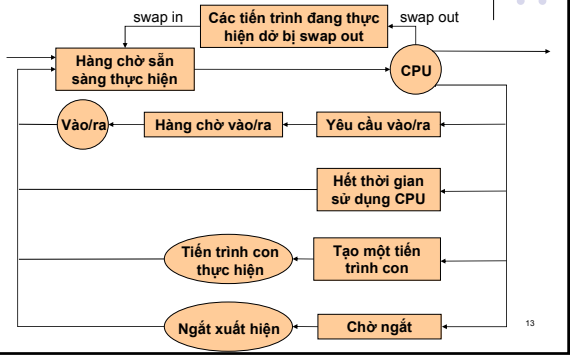
11

Minh họa bộ lập lịch trung hạn



12

Hàng chờ lập lịch tiến trình



13

Chuyển trạng thái

- Thuật ngữ: Context switch
- Xảy ra khi một tiến trình A bị ngắt ra khỏi CPU, tiến trình B bắt đầu được sử dụng CPU
- Cách thực hiện:
  - Nhân HĐH ghi lại toàn bộ trạng thái của A, lấy từ PCB (khối điều khiển tiến trình) của A
  - Đưa A vào hàng chờ
  - Nhân HĐH nạp trạng thái của B lấy từ PCB của B
  - Thực hiện B

14

Chuyển trạng thái

- Việc chuyển trạng thái, nói chung, là lãng phí thời gian của CPU
- Do đó việc chuyển trạng thái cần được thực hiện càng nhanh càng tốt
- Thông thường thời gian chuyển trạng thái mất khoảng 1-1000 micro giây

15

Các thao tác với tiến trình

16

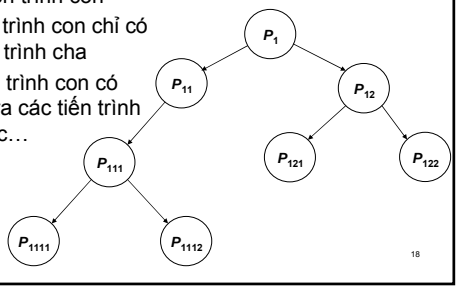
Tạo tiến trình

- HĐH cung cấp hàm **create-process** để tạo một tiến trình mới
  - Tiến trình gọi đến hàm **create-process** là tiến trình cha (parent process)
  - Tiến trình được tạo ra sau khi thực hiện hàm **create-process** là tiến trình con (child process)
- Sau khi tiến trình con được tạo, tiến trình cha có thể:
  - Chờ tiến trình con kết thúc rồi tiếp tục thực hiện
  - Thực hiện "song song" với tiến trình con

17

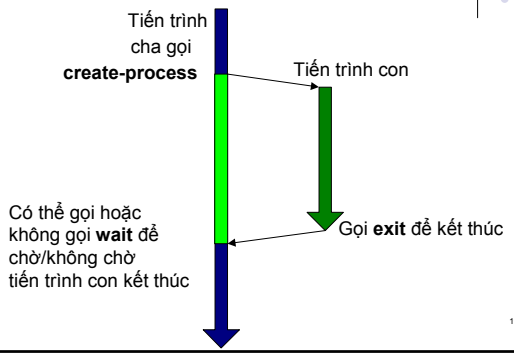
Cây tiến trình

- Tiến trình cha có thể có nhiều tiến trình con
- Mỗi tiến trình con chỉ có một tiến trình cha
- Các tiến trình con có thể tạo ra các tiến trình con khác…



18

Minh họa tiến trình cha và con



19

Kết thúc tiến trình

- Một tiến trình kết thúc khi:
  - Thực hiện xong và gọi hàm hệ thống **exit** (kết thúc bình thường)
  - Gọi đến hàm **abort** hoặc **kill** (kết thúc bất thường khi có lỗi hoặc có sự kiện)
  - Bị hệ thống hoặc tiến trình cha áp dụng hàm **abort** hoặc **kill** do:
    - Sử dụng quá quota tài nguyên
    - Tiến trình con không còn cần thiết
    - Khi tiến trình cha đã kết thúc (trong một số HĐH)

20

Minh họa tiến trình trong UNIX

```
#include <stdio.h>
main()
{
    int pid=fork(); /* Tạo tiến trình mới bằng hàm fork() */
    if (pid<0) { perror("Cannot create process"); return(-1); }
    else if (pid==0) { /* Tiến trình con */
        execlp();
    }
    else { /* Tiến trình cha */
        wait(NULL);  /* Nếu không có lệnh này tiến trình cha thực hiện
                        "song song" với tiến trình con */
        printf("Child completed\n");
        return(0);
    }
}
```

21

Hợp tác giữa các tiến trình

- Các tiến trình có thể hoạt động *độc lập* hoặc *hợp tác* với nhau
- Các tiến trình cần hợp tác khi:
  - Sử dụng chung thông tin
  - Thực hiện một số nhiệm vụ chung
  - Tăng tốc độ tính toán
  - …
- Để hợp tác các tiến trình, cần có các cơ chế truyền thông/liên lạc giữa các tiến trình (Interprocess communication – IPC)

22

Truyền thông giữa các tiến trình (IPC)

23

Các hệ thống truyền thông điệp

- Cho phép các tiến trình truyền thông với nhau qua các toán tử **send** và **receive**
- Các tiến trình cần có *tên* để tham chiếu
- Cần có một kết nối (*logic)* giữa tiến trình *P* và *Q* để truyền thông điệp
- Một số loại truyền thông:
  - Trực tiếp hoặc gián tiếp
  - Đối xứng hoặc không đối xứng
  - Sử dụng vùng đệm tự động hoặc không tự động

24

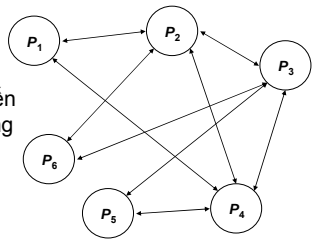
Truyền thông trực tiếp

- Hai toán tử
  - **send**(*P*, *msg*): Gửi *msg* đến tiến trình *P*
  - **receive**(*Q*, *msg*): Nhận *msg* từ tiến trình *Q*
- Cải tiến:
  - **send**(*P*, *msg*): Gửi *msg* đến tiến trình *P*
  - **receive**(*id*, *msg*): Nhận *msg* từ bất kỳ tiến trình nào

25

Minh họa truyền thông trực tiếp

- Mỗi kết nối được thiết lập cho một cặp tiến trình duy nhất
- Mỗi tiến trình chỉ cần biết tên/số hiệu của tiến trình kia là truyền thông được
- Tồn tại duy nhất một kết nối giữa một cặp tiến trình



26

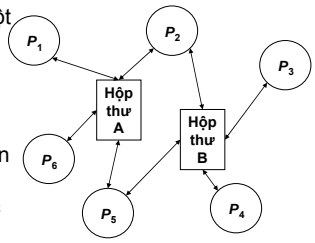
Truyền thông gián tiếp

- Các thông điệp được gửi và nhận qua các *hộp thư (*mailbox) hoặc qua các *cổng* (port)
- Hai toán tử:
  - **send**(*A*, *msg*): Gửi *msg* đến hộp thư *A*
  - **receive**(*B*, *msg*): Nhận *msg* từ hộp thư *B*
- Minh họa: Topo mạng hình sao

27

Minh họa truyền thông gián tiếp

- Hai tiến trình có kết nối nếu sử dụng chung một hộp thư
- Một kết nối có thể sử dụng cho nhiều tiến trình (>=2)
- Nhiều kết nối có thể tồn tại giữa một cặp tiến trình (nếu sử dụng các hộp thư khác nhau)



28

Vấn đề đồng bộ hóa

- Thuật ngữ: Synchronization
- Liên quan tới phương thức cài đặt các toán tử **send** và **receive**:
  - Phương thức có chờ (blocking)
  - Phương thức không chờ (non-blocking)

29

Các phương thức send/receive

	send(*P*, *msg*)	**receive**(*Q*, *msg*)
Blocking	Tiến trình truyền thông điệp chờ đến khi *msg* được nhận hoặc *msg* được phân phát đến hộp thư	Tiến trình nhận tạm dừng thực hiện cho đến khi *msg* được chuyển tới
Non-blocking	Tiến trình truyền không phải chờ *msg* đến đích để tiếp tục thực hiện	Tiến trình nhận trả lại kết quả là *msg* (nếu nhận được) hoặc báo lỗi (nếu chưa nhận được)

30

Vấn đề sử dụng vùng đệm

- Các thông điệp nằm trong hàng chờ tạm thời
- Cỡ của hàng chờ:
  - Chứa được 0 thông điệp: **send** blocking
  - Chứa được *n* thông điệp: **send** non-blocking cho đến khi hàng chờ có *n* thông điệp, sau đó **send** blocking
  - Vô hạn: **send** non-blocking

31

Luồng (thread)

- Sinh viên tự tìm hiểu trong giáo trình trang

32

Tóm tắt

- Khái niệm tiến trình
- Các trạng thái, chuyển trạng thái tiến trình
- Khối điều khiển tiến trình
- Lập lịch tiến trình, các loại bộ lập lịch
- Truyền thông giữa các tiến trình
  - Gián tiếp, trực tiếp
  - Blocking và non-blocking (đồng bộ hóa)
  - Vấn đề sử dụng vùng đệm (buffer)

33

Bài tập

- Viết chương trình C trong Linux/Unix tạo ra 16 tiến trình con. Tiến trình cha chờ cho 16 tiến trình con này kết thúc rồi mới kết thúc bằng hàm **exit.** Sử dụng các hàm **fork** và **wait** để thực hiện yêu cầu.
- Hãy tìm một số ví dụ thực tế minh họa cho các khái niệm lập lịch/hàng chờ trong tình huống có nhiều người sử dụng và ít tài nguyên.

34

Bài tập

- Hãy viết chương trình minh họa cho các cơ chế truyền thông non-blocking, blocking
- Hãy viết chương trình minh họa các cơ chế truyền thông điệp sử dụng buffer có độ dài *n* trong hai trường hợp: *n*>0 và *n*=0
- Chú ý: Để làm hai bài tập trên cần sử dụng hai tiến trình; có thể thực hiện bài tập với UNIX/Linux hoặc Windows

35

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Lập lịch CPU

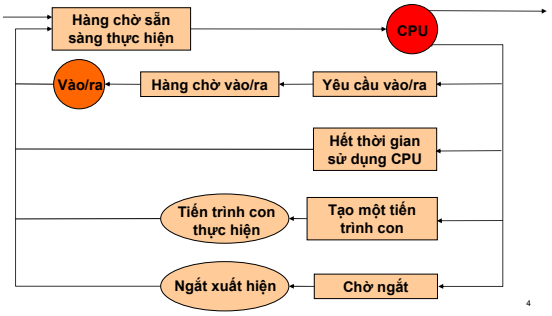
2

Tại sao phải lập lịch CPU?

- Số lượng NSD, số lượng tiến trình luôn lớn hơn số lượng CPU của máy tính rất nhiều
- Tại một thời điểm, chỉ có duy nhất một tiến trình được thực hiện trên một CPU
- Vấn đề:
  - Nhu cầu sử dụng nhiều hơn tài nguyên (CPU) đang có
  - Do đó cần lập lịch để phân phối thời gian sử dụng CPU cho các tiến trình của NSD và hệ thống

3

Hàng chờ lập lịch tiến trình

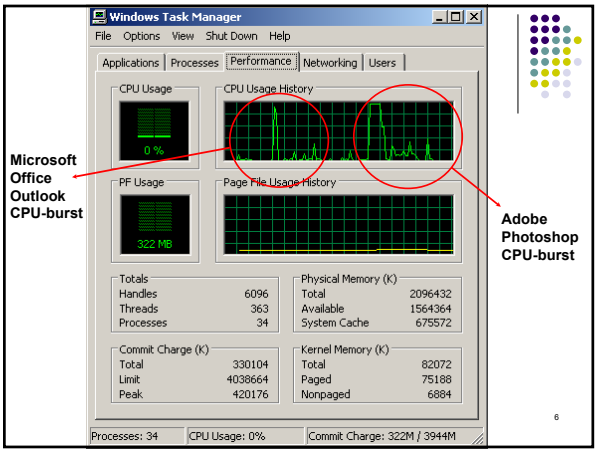


4

CPU-burst và IO-burst

- Trong suốt thời gian tồn tại trong hệ thống, tiến trình được xem như thực hiện hai loại công việc chính:
  - Khi tiến trình ở trạng thái running: Sử dụng CPU (thuật ngữ: *CPU-burst*)
  - Khi tiến trình thực hiện các thao tác vào ra: Sử dụng thiết bị vào/ra (thuật ngữ: *I/O burst*)

5



6

Hai loại tiến trình chính

- Căn cứ theo cách sử dụng CPU của tiến trình, có hai loại tiến trình:
  - Tiến trình loại CPU-bound: Tiến trình có một hoặc nhiều phiên sử dụng CPU dài
  - Tiến trình loại I/O-bound: Tiến trình có nhiều phiên sử dụng CPU ngắn (tức là thời gian vào ra nhiều)

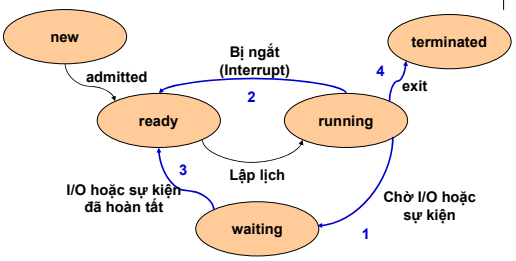
7

Bộ lập lịch ra hoạt động khi…

1. Một tiến trình chuyển từ trạng thái running sang waiting
2. Một tiến trình chuyển từ trạng thái running sang ready
3. Một tiến trình chuyển từ trạng thái waiting sang ready
4. Một tiến trình kết thúc

8

Các phương pháp lập lịch



- 1 và 4: Lập lịch non-preemptive
- Ngược lại: Lập lịch preemptive

9

Lập lịch non-preemptive

- Một tiến trình giữ CPU đến khi nó kết thúc hoặc chuyển sang trạng thái waiting.
- Ví dụ: Microsoft Windows 3.1, Apple Macintosh sử dụng lập lịch non-preemptive
- Có thể sử dụng trên nhiều loại phần cứng vì không đòi hỏi timer

10

Lập lịch preemptive

- Hiệu quả hơn lập lịch non-preemptive
- Thuật toán phức tạp hơn non-preemptive và sử dụng nhiều tài nguyên CPU hơn
- Ví dụ: Microsoft Windows XP, Linux, UNIX sử dụng lập lịch preemptive

11

Bộ điều phối (dispatcher)

- Nhiệm vụ:
  - Chuyển trạng thái (context switch)
  - Chuyển về user-mode
  - Thực hiện tiến trình theo trạng thái đã lưu
- Cần hoạt động hiệu quả (tốc độ nhanh)
- Thời gian cần để bộ điều phối dừng một tiến trình và thực hiện tiến trình khác gọi là độ trễ (latency) của bộ điều phối

12

Các tiêu chí đánh giá lập lịch

- Khả năng tận dụng CPU (*CPU utilization*): Thể hiện qua tải CPU – là một số từ 0% đến 100%.
  - Trong thực tế các hệ thống thường có tải từ 40% (tải thấp) đến 90% (tải cao)
- Thông lượng (*throughput*): Là số lượng các tiến trình hoàn thành trong một đơn vị thời gian

13

Các tiêu chí đánh giá lập lịch

- Thời gian hoàn thành (*turnaround time*):
  - $t_{turnaround} = t_o - t_i$ với t_i là thời điểm tiến trình vào hệ thống, t_o là thời điểm tiến trình ra khỏi hệ thống (kết thúc thực hiện)
  - Như vậy $t_{turnaround}$ là tổng: thời gian tải vào bộ nhớ, thời gian thực hiện, thời gian vào ra, thời gian nằm trong hàng chờ…

14

Các tiêu chí đánh giá lập lịch

- Thời gian chờ (*waiting time*): Là tổng thời gian tiến trình phải nằm trong hàng chờ ready ($t_{waiting}$)
  - Các thuật toán lập lịch CPU không có ảnh hưởng đến tổng thời gian thực hiện một tiến trình mà chỉ có ảnh hưởng đến thời gian chờ của một tiến trình trong hàng chờ ready
  - Thời gian chờ trung bình (*average waiting time*): $t_{averagewaiting}=t_{waiting} / n$, n là số lượng tiến trình trong hàng chờ

15

Các tiêu chí đánh giá lập lịch

- Thời gian đáp ứng (*response time*): Là khoảng thời gian từ khi tiến trình nhận được một yêu cầu cho đến khi *bắt đầu đáp ứng* yêu cầu đó
- $t_{res} = t_r - t_s$, trong đó t_r là thời điểm nhận yêu cầu, t_s là thời điểm bắt đầu đáp ứng yêu cầu

16

Đánh giá các thuật toán lập lịch

Tiêu chí	Giá trị thấp	Giá trị cao
Khả năng tận dụng CPU (CPU utilization)	Xấu	Tốt
Thông lượng (throughput)	Xấu	Tốt
Thời gian hoàn thành (turnaround time)	Tốt	Xấu
Thời gian chờ (waiting time) -> Thời gian chờ trung bình	Tốt	Xấu
Thời gian đáp ứng (response time)	Tốt	Xấu

17

Các thuật toán lập lịch

18

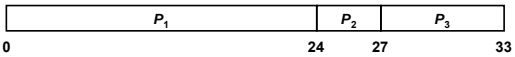
FCFS (First Come First Served)

- Tiến trình nào có yêu cầu sử dụng CPU trước sẽ được thực hiện trước
- Ưu điểm: Thuật toán đơn giản nhất
- Nhược điểm: Hiệu quả của thuật toán phụ thuộc vào *thứ tự* của các tiến trình trong hàng chờ, vì thứ tự này ảnh hưởng rất lớn đến *thời gian chờ trung bình* (*average waiting time*)

19

Ví dụ FCFS 1a

- Giả sử có 3 tiến trình P_1, P_2, P_3 với thời gian thực hiện tương ứng là 24ms, 3ms, 6ms
- Giả sử 3 tiến trình xếp hàng theo thứ tự P_1, P_2, P_3. Khi đó ta có biểu đồ Gantt như sau:

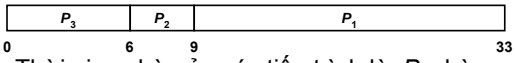


- Thời gian chờ của các tiến trình là: P_1 chờ 0ms, P_2 chờ 24ms, P_3 chờ 27ms
- Thời gian chờ trung bình: (0+24+27)/3=17ms

20

Ví dụ FCFS 1b

- Xét ba tiến trình trong ví dụ 1a với thứ tự xếp hàng P_3, P_2, P_1
- Biểu đồ Gantt:



- Thời gian chờ của các tiến trình là: P_3 chờ 0ms, P_2 chờ 6ms, P_3 chờ 9ms
- Thời gian chờ trung bình: (0+6+9)/3=5ms
- Thời gian chờ trung bình thấp hơn thời gian chờ trung bình trong ví dụ 1a

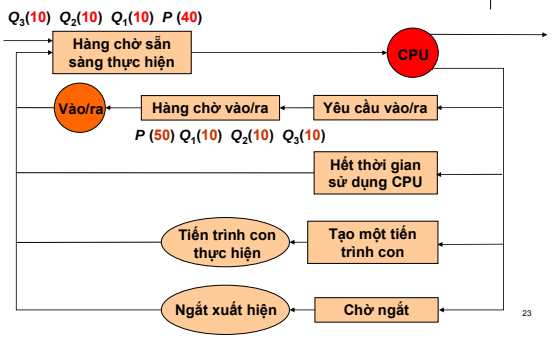
21

Hiện tượng "đoàn hộ tống"

- Thuật ngữ: *convoy effect*
- Xảy ra khi có một tiến trình "lớn" *P* nằm ở đầu hàng chờ và nhiều tiến trình "nhỏ" Q_i xếp hàng sau *P*.
- "Lớn": Sử dụng nhiều thời gian CPU và vào ra
- "Nhỏ": Sử dụng ít thời gian CPU và vào ra
- Thuật toán lập lịch được sử dụng là FCFS.:
- Hiện tượng xảy ra: CPU, thiết bị vào ra có nhiều thời gian rỗi, thời gian chờ trung bình của các tiến trình cao

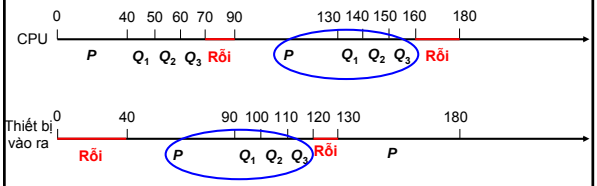
22

Ví dụ convoy effect



23

Convoy effect: Thời gian sử dụng CPU và thiết bị vào ra



- Giả sử *P* là tiến trình "lớn" có chu kỳ sử dụng CPU trong 40ms và vào ra trong 50ms
- Q_1, Q_2, Q_3 là 3 tiến trình "nhỏ" có chu kỳ sử dụng CPU trong 10ms và vào ra trong 10ms
- Thứ tự xếp hàng: *P*, Q_1, Q_2, Q_3

24

SJF (Shortest Job First)

- Với SJF, tham số lập lịch có thêm *độ dài của phiên sử dụng CPU tiếp theo* $t_{nextburst}$
- Tiến trình có $t_{nextburst}$ nhỏ nhất sẽ được lập lịch sử dụng CPU trước
- Nếu hai tiến trình có $t_{nextburst}$ bằng nhau thì FCFS được áp dụng

25

SJF (Shortest Job First)

- Với lập lịch dài hạn: Có thể biết $t_{nextburst}$ vì có tham số chỉ ra thời gian chạy của tiến trình khi đưa tiến trình vào hệ thống (job submit)
- Khó khăn: Chỉ có thể phỏng đoán được $t_{nextburrt}$ với lập lịch ngắn hạn
- Đọc ở nhà: Dự đoán $t_{nextburst}$ bằng công thức trung bình mũ (exponential average) trang 160-162 trong giáo trình
- SJF không tối ưu được với lập lịch ngắn hạn
- SJF thường được áp dụng cho lập lịch dài hạn

26

Lập lịch có độ ưu tiên

- Thuật ngữ: Priority scheduling
- Mỗi tiến trình được gắn một tham số lập lịch gọi là độ ưu tiên *p*, với *p* là một số thực
- Tiến trình có độ ưu tiên cao nhất được sử dụng CPU
- Qui ước trong môn học này:
  - Tiến trình P_1 và P_2 có độ ưu tiên p_1, p_2 tương ứng
  - $p_1 > p_2$ có nghĩa là tiến trình P_1 có độ ưu tiên thấp hơn P_2
- SJF là trường hợp đặc biệt của lập lịch có ưu tiên với ưu tiên của các tiến trình là nghịch đảo độ dài phiên sử dụng

27

Lập lịch có độ ưu tiên

- Hai cách xác định độ ưu tiên:
  - Độ ưu tiên trong: Được tính toán dựa trên các tham số định lượng của tiến trình như giới hạn về thời gian, bộ nhớ, số file đang mở, thời gian sử dụng CPU
  - Độ ưu tiên ngoài: Dựa vào các yếu tố như mức độ quan trọng, chi phí thuê máy tính…
- Chờ không xác định: Một tiến trình có độ ưu tiên thấp có thể nằm trong hàng chờ trong một khoảng thời gian dài nếu trong hàng chờ luôn có các tiến trình có độ ưu tiên cao hơn

28

Lập lịch có độ ưu tiên

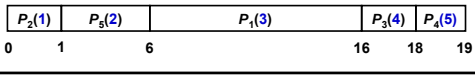
- Để tránh hiện tượng chờ không xác định, có thêm tham số *tuổi* để xác định thời gian tiến trình thời gian nằm trong hàng chờ
- Tham số *tuổi* được sử dụng để làm tăng độ ưu tiên của tiến trình
- Ví dụ thực tế: Khi bảo dưỡng máy tính IBM 7094 của MIT năm 1973, người ta thấy một tiến trình nằm trong hàng chờ từ năm 1967 nhưng vẫn chưa được thực hiện

29

Ví dụ lập lịch có độ ưu tiên

- Xét 5 tiến trình như trong bảng với thứ tự trong hàng chờ là P_1, P_2, P_3, P_4, P_5
- Sử dụng thuật toán lập lịch có ưu tiên, ta có thứ tự thực hiện của các tiến trình như sau biểu đồ dưới
- Thời gian chờ trung bình là (1+6+16+18)/5=8.2ms

Tiến trình	Thời gian thực hiện (ms)	Độ ưu tiên
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2



30

Round-robin (RR)

- Còn gọi là lập lịch quay vòng
- Được thiết kế để áp dụng cho các hệ phân chia thời gian (time-sharing)
- RR hoạt động theo chế độ preemptive
- Tham số lượng tử thời gian (time quantum) $t_{quantum}$: Mỗi tiến trình được sử dụng CPU trong nhiều nhất bằng $t_{quantum}$, sau đó đến lượt tiến trình khác

31

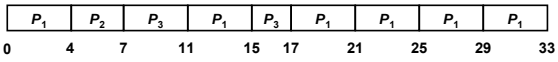
Round-robin (RR)

- Hiệu quả của RR phụ thuộc độ lớn của $t_{quantum}$
  - Nếu $t_{quaptum}$ nhỏ thì hiệu quả của RR giảm vì bộ điều phối phải thực hiện nhiều thao tác chuyển trạng thái, lãng phí thời gian CPU
  - Nếu $t_{quantum}$ lớn thì số thao tác chuyển trạng thái giảm đi
- Nếu $t_{quantum}$ rất nhỏ (ví dụ 1ms) thì RR được gọi là processor sharing
- Nếu $t_{quantum}$ = ∞ thì RR trở thành FCFS

32

Ví dụ RR

- Giả sử có 3 tiến trình P_1, P_2, P_3 với thời gian thực hiện tương ứng là 24ms, 3ms, 6ms, thứ tự trong hàng chờ P_1, P_2, P_3 ,vào hàng chờ cùng thời điểm 0
- Giả sử $t_{quantum}$=4ms
- RR lập lịch các tiến trình như sau:



- Thời gian chờ
  - P_1: 0+(11-4)+(17-15)=9ms
  - P_2: 4ms
  - P_3: 7+(15-11)=11ms
- Thời gian chờ trung bình: (9+4+11)/3=8ms

33

Lập lịch với hàng chờ đa mức

- Thuật ngữ: Multilevel queue scheduling
- Được sử dụng khi ta có thể chia các tiến trình thành nhiều lớp khác nhau để lập lịch theo các tiêu chí khác nhau, ví dụ:
  - Lớp các tiến trình có tương tác (interactive hoặc foreground process) cần có độ ưu tiên cao hơn
  - Lớp các tiến trình chạy nền (background) thường không có tương tác với NSD: Độ ưu tiên thấp hơn

34

Lập lịch với hàng chờ đa mức

- Thuật toán lập lịch với hàng chờ đa mức chia hàng chờ ready thành nhiều hàng chờ con khác nhau, mỗi hàng chờ con được áp dụng một loại thuật toán khác nhau, ví dụ:
  - Hàng chờ các tiến trình background: FCFS
  - Hàng chờ các tiến trình có tương tác:RR
- Các tiến trình được phân lớp dựa vào đặc tính như bộ nhớ, độ ưu tiên, …
- Cần có thuật toán lập lịch cho các hàng chờ con, ví dụ: preemptive có độ ưu tiên cố định

35

Ví dụ hàng chờ đa mức

- Ví dụ các hàng chờ đa mức có độ ưu tiên giảm dần:
  - Hàng chờ các tiến trình hệ thống
  - Hàng chờ các tiến trình có tương tác
  - Hàng chờ các tiến trình là editor
  - Hàng chờ các tiến trình hoạt động theo lô
  - Hàng chờ các tiến trình thực tập của sinh viên

36

Lập lịch với hàng chờ đa mức có phản hồi

- Thuật ngữ: Multilevel feedback-queue scheduling
- Thuật toán lập lịch kiểu này nhằm khắc phục nhược điểm *không mềm dẻo* của lập lịch với hàng chờ đa mức
- Ý tưởng chính: Cho phép tiến trình chuyển từ hàng chờ này sang hàng chờ khác, trong khi lập lịch với hàng chờ đa mức không cho phép điều này

37

Lập lịch với hàng chờ đa mức có phản hồi

- Cách thực hiện:
  - Độ dài phiên sử dụng CPU và thời gian đã nằm trong hàng chờ là tiêu chuẩn chuyển tiến trình giữa các hàng chờ
  - Tiến trình chiếm nhiều thời gian CPU sẽ bị chuyển xuống hàng chờ có độ ưu tiên thấp
  - Tiến trình nằm lâu trong hàng chờ sẽ được chuyển lên hàng chờ có độ ưu tiên cao hơn

38

Các tham số của bộ lập lịch hàng chờ đa mức có phản hồi

- Số lượng các hàng chờ
- Thuật toán lập lịch cho mỗi hàng chờ
- Phương pháp tăng độ ưu tiên cho một tiến trình
- Phương pháp giảm độ ưu tiên cho một tiến trình
- Phương pháp xác định hàng đợi nào để đưa một tiến trình vào

39

Các thuật toán lập lịch khác

- Sinh viên tìm hiểu trong giáo trình:
  - Lập lịch đa xử lý
  - Lập lịch thời gian thực

40

Các phương pháp đánh giá thuật toán lập lịch

41

Mô hình xác định

- Thuật ngữ: Deterministic modeling
- Dựa vào các trường hợp cụ thể (chẳng hạn các ví dụ đã nói trong bài giảng này) để rút ra các kết luận đánh giá
- Ưu điểm: Nhanh và đơn giản
- Nhược điểm: Không rút ra được kết luận đánh giá cho trường hợp tổng quát

42

Mô hình hàng chờ

- Thuật ngữ: Queueing model
- Dựa trên lý thuyết xác suất, quá trình ngẫu nhiên. Tài liệu tham khảo:
  - Leonard Kleinrock, *Queueing Systems: Volume I – Theory* (Wiley Interscience, New York, 1975)
  - Leonard Kleinrock, *Queueing Systems: Volume II – Computer Applications* (Wiley Interscience, New York, 1976)
- Ưu điểm: So sánh được các thuật toán lập lịch trong một số trường hợp
- Hạn chế: Phức tạp về mặt toán học, lớp các thuật toán phân tích so sánh được còn hạn chế

43

Mô phỏng

- Thuật ngữ: Simulation
- Thực hiện các tình huống giả định trên máy tính để đánh giá hiệu quả của các thuật toán lập lịch, kiểm nghiệm các kết quả lý thuyết
- Mô phỏng thường tốn thời gian CPU và không gian lưu trữ
- Được sử dụng nhiều trong công nghiệp
- Ví dụ các hệ mô phỏng mạng (có module hàng chờ): OPNET, NS-2, Qualnet

44

Cài đặt thử trong thực tế

- Mô phỏng có thể xem như "qui nạp không hoàn toàn"
- Có thể xây dựng hệ thống thử trong thực tế
- Ưu điểm: Đánh giá được hiệu quả thực sự khi sử dụng
- Nhược điểm:
  - Chi phí cao
  - Hành vi của người sử dụng có thể thay đổi theo môi trường hệ thống

45

Tóm tắt

- Khái niệm lập lịch, các tiêu chí đánh giá thuật toán lập lịch
- Các phương thức hoạt động preemptive và non-preemptive
- Các thuật toán lập lịch FCFS, SJF, ưu tiên, RR
- Lập lịch với hàng chờ đa mức, có và không có phản hồi
- Các phương pháp đánh giá thuật toán lập lịch

46

Bài tập

- Thực hiện ví dụ RR với lượng tử thời gian 2ms, 6ms và 6ms.
  - Tính thời gian chờ trung bình của các tiến trình trong các trường hợp này.
  - Khi lượng tử thời gian thay đổi, thời gian chờ trung bình thay đổi thế nào?
  - Tính thời gian hoàn thành (turnaround time) của tất cả các tiến trình trong các trường hợp trên
  - Nhận xét về sự thay đổi thời gian hoàn thành của các tiến trình khi lượng tử thời gian thay đổi

47

Bài tập

- Hãy xây dựng một ví dụ về hiện tượng convoy effect sao cho thời gian rỗi của thiết bị vào ra ít hơn thời gian rỗi của CPU. Giả sử có 4 tiến trình, trong đó *P* là tiến trình "lớn", Q_1, Q_2, Q_3 là các tiến trình "nhỏ" được nằm trong hàng chờ theo thứ tự: *P,* Q_1, Q_2, Q_3
- Tìm hai ví dụ trong thực tế về hàng chờ đa mức và hàng chờ đa mức có phản hồi và giải thích các ví dụ này.

48

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

Đồng bộ hóa tiến trình

Ví dụ đồng bộ hóa (1)

Tiến trình ghi *P*:

```
while (true) {
   while (counter==SIZE) ;
   buf[in] = nextItem;
   in = (in+1) % SIZE;
   counter++;
}
```

buf: Buffer
SIZE: cỡ của buffer
counter: Biến chung

Tiến trình đọc *Q*:

```
while (true) {
   while (counter==0) ;
   nextItem = buf[out];
   out = (out+1) % SIZE;
   counter--;
}
```

- Đây là *bài toán vùng đệm có giới hạn*

Ví dụ đồng bộ hóa (2)

- Các toán tử ++ và -- có thể được cài đặt như sau:

- counter++

```
register1 = counter;
register1 = register1 + 1;
counter = register1;
```

- counter--

```
register2 = counter;
register2 = register2 - 1;
counter = register2;
```

P và Q có thể nhận được các giá trị khác nhau của counter tại cùng 1 thời điểm nếu như đoạn mã xanh và đỏ thực hiện xen kẽ nhau.

Ví dụ đồng bộ hóa (3)

- Giả sử P và Q thực hiện song song với nhau và giá trị của counter là 5:

```
register1 = counter;          // register1=5
register1 = register1 + 1;    // register1=6
register2 = counter;          // register2=5
register2 = register2 - 1;    // register2=4
counter = register1;          // counter=6 !!
counter = register2;          // counter=4 !!
```


Ví dụ đồng bộ hóa (4)

- Lỗi: Cho phép *P* và *Q* **đồng thời thao tác trên biến chung counter**. Sửa lỗi:

```
register1 = counter;          // register1=5
register1 = register1 + 1;    // register1=6
counter = register1;          // counter=6
register2 = counter;          // register2=6
register2 = register2 - 1;    // register2=5
counter = register2;          // counter=5
```

Tương tranh và đồng bộ

- Tình huống xuất hiện khi nhiều tiến trình cùng thao tác trên dữ liệu chung và kết quả các thao tác đó phụ thuộc vào thứ tự thực hiện của các tiến trình trên dữ liệu chung gọi là *tình huống tương tranh* (*race condition*)
- Để tránh các tình huống tương tranh, các tiến trình cần được *đồng bộ* theo một phương thức nào đó ⇒ Vấn đề nghiên cứu: Đồng bộ hóa các tiến trình

7

Khái niệm về đoạn mã găng (1)

- Thuật ngữ: Critical section
- Thuật ngữ tiếng Việt: Đoạn mã găng, đoạn mã tới hạn.
- Xét một hệ có n tiến trình P_0, P_1, ..., P_n, mỗi tiến trình có một đoạn mã lệnh gọi là đoạn mã găng, ký hiệu là CS_i, nếu như trong đoạn mã này, các tiến trình thao tác trên các biến chung, đọc ghi file... (tổng quát: thao tác trên dữ liệu chung)

8

Khái niệm về đoạn mã găng (2)

- Đặc điểm quan trọng mà hệ n tiến trình này cần có là: Khi một tiến trình P_i thực hiện đoạn mã CS_i thì không có tiến trình P_j nào khác được phép thực hiện CS_j
- Mỗi tiến trình P_i phải "xin phép" (entry section) trước khi thực hiện CS_i và thông báo (exit section) cho các tiến trình khác sau khi thực hiện xong CS_i.

9

Khái niệm về đoạn mã găng (3)

- Cấu trúc chung của P_i để thực hiện đoạn mã găng CS_i.

```
do {
   Xin phép (ENTRY_i) thực hiện CS_i; // Entry section
   Thực hiện CS_i;
   Thông báo (EXIT_i) đã thực hiện xong CS_i; // Exit section
   Phần mã lệnh khác (REMAIN_i); // Remainder section
} while (TRUE);
```

10

Khái niệm về đoạn mã găng (4)

- Viết lại cấu trúc chung của đoạn mã găng:

```
do {
   ENTRY_i;        // Entry section
   Thực hiện CS_i; // Critical section
   EXIT_i;         // Exit section
   REMAIN_i;       // Remainder section
} while (TRUE);
```

11

Giải pháp cho đoạn mã găng

- Giải pháp cho đoạn mã găng cần thỏa mãn 3 điều kiện:
  - Loại trừ lẫn nhau (mutual exclusion): Nếu P_i đang thực hiện CS_i thì P_j không thể thực hiện CS_j $\forall j \neq i$.
  - Tiến triển (progress): Nếu không có tiến trình P_i nào thực hiện CS_i và có m tiến trình P_{j1}, P_{j2}, ..., P_{jm} muốn thực hiện CS_{j1}, CS_{j2}, ..., CS_{jm} thì chỉ có các tiến trình không thực hiện $REMAIN_{jk}$ (k=1,...,m) mới được xem xét thực hiện CS_{jk}.
  - Chờ có giới hạn (bounded waiting): *sau khi* một tiến trình P_i có yêu cầu vào CS_i và *trước khi* yêu cầu đó được chấp nhận, số lần các tiến trình P_j (với $j \neq i$) được phép thực hiện CS_j phải bị giới hạn.

12

Ví dụ: giải pháp của Peterson

- Giả sử có 2 tiến trình P_0 và P_1 với hai đoạn mã găng tương ứng CS_0 và CS_1
- Sử dụng một biến nguyên *turn* với giá trị khởi tạo 0 hoặc 1 và mảng boolean *flag*[2]
- *turn* có giá trị *i* có nghĩa là P_i được phép thực hiện CS_i (*i*=0,1)
- nếu flag[i] là TRUE thì tiến trình P_i đã sẵn sàng để thực hiện CS_i

13

Ví dụ: giải pháp của Peterson

- Mã lệnh của P_i:

```
do {
   flag[i] = TRUE;
   turn = j;
   while (flag[j] && turn == j) ;
   CSi;
   flag[j] = FALSE;
   REMAINi;
} while (1);
```

14

Chứng minh giải pháp Peterson

- Xem chứng minh giải pháp của Peterson thỏa mãn 3 điều kiện của đoạn mã găng trong giáo trình (trang 196)
- Giải pháp "kiểu Peterson":
  - Phức tạp khi số lượng tiến trình tăng lên
  - Khó kiểm soát

15

Semaphore


16

Thông tin tham khảo

- Edsger Wybe Dijkstra (người Hà Lan) phát minh ra khái niệm semaphore trong khoa học máy tính vào năm 1972
- Semaphore được sử dụng lần đầu tiên trong cuốn sách "The operating system" của ông

Edsger Wybe Dijkstra (1930-2002)

17

Định nghĩa

- Semaphore là một biến nguyên, nếu không tính đến toán tử khởi tạo, chỉ có thể truy cập thông qua hai toán tử *nguyên tố* là wait (hoặc P) và signal (hoặc V).
  - P: proberen – kiểm tra (tiếng Hà Lan)
  - V: verhogen – tăng lên (tiếng Hà Lan)
- Các tiến trình có thể *sử dụng chung* semaphore
- Các toán tử là nguyên tố để đảm bảo không xảy ra trường hợp như ví dụ đồng bộ hóa đã nêu

18

Toán tử wait và signal

```
wait(S)     // hoặc P(S)
{
   while (S<=0);
   S--;
}
```

- Toán tử wait: Chờ khi semaphore S âm và giảm S đi 1 nếu S>0

```
signal(S)     // hoặc V(S)
{
   S++;
}
```

- Toán tử signal: Tăng S lên 1

19

Sử dụng semaphore (1)

- Với bài toán đoạn mã găng:

```
do {
   wait(mutex); // mutex là semaphore khởi tạo 1
   CS_i;
   signal(mutex);
   REMAIN_i;
} while (1);
```

20

Sử dụng semaphore (2)

- Xét hai tiến trình P_1 và P_2, P_1 cần thực hiện toán tử O_1, P_2 cần thực hiện O_2 và O_2 chỉ được thực hiện sau khi O_1 đã hoàn thành
- Giải pháp: Sử dụng semaphore *synch* = 0

- P_1:

```
...
O1;
signal(synch);
...
```

- P_2:

```
...
wait(synch);
O2;
...
```

21

Cài đặt semaphore cổ điển

- Định nghĩa cổ điển của wait cho ta thấy toán tử này có *chờ bận* (busy waiting), tức là tiến trình phải chờ toán tử wait kết thúc nhưng CPU vẫn phải làm việc: Lãng phí tài nguyên
- Liên hệ cơ chế polling trong kiến trúc máy tính
- Cài đặt semaphore theo định nghĩa cổ điển:
  - Lãng phí tài nguyên CPU với các máy tính 1 CPU
  - Có lợi nếu thời gian chờ wait ít hơn thời gian thực hiện context switch
  - Các semaphore loại này gọi là *spinlock*

22

Cài đặt semaphore theo cấu trúc

- Khắc phục chờ bận: Chuyển vòng lặp chờ thành việc sử dụng toán tử block (tạm dừng)
- Để khôi phục thực hiện từ block, ta có toán tử wakeup
- Khi đó để cài đặt, ta có cấu trúc dữ liệu mới cho semaphore:

```
typedef struct {
   int value; // Giá trị của semaphore
   struct process *L; // Danh sách tiến trình chờ...
} semaphore;
```

23

Cài đặt semaphore theo cấu trúc

```
void wait(semaphore *S)
{
   S->value--;
   if (S->value<0) {
      Thêm tiến trình gọi
      toán tử vào s->L;
      block();
   }
}
```

```
void signal(semaphore *S)
{
   S->value++;
   if (S->value<=0) {
      Xóa một tiến trình P
      ra khỏi s->L;
      wakeup(P);
   }
}
```

24

Semaphore nhị phân

- Là semaphore chỉ nhận giá trị 0 hoặc 1
- Cài đặt semaphore nhị phân đơn giản hơn semaphore không nhị phân (thuật ngữ: counting semaphore)

25

Một số bài toán đồng bộ hóa cơ bản

26

Bài toán vùng đệm có giới hạn

- Đã xét ở ví dụ đầu tiên (the bounded-buffer problem)
- Ta sử dụng 3 semaphore tên là *full*, *empty* và *mutex* để giải quyết bài toán này
- Khởi tạo:
  - *full*: Số lượng phần tử buffer đã có dữ liệu (0)
  - *empty*: Số lượng phần tử buffer chưa có dữ liệu (*n*)
  - *mutex*: 1 (Chưa có tiến trình nào thực hiện đoạn mã găng)

27

Bài toán vùng đệm có giới hạn

Tiến trình ghi *P*:

```
do {
   wait(empty);
   wait(mutex);
   // Ghi một phần tử mới
   // vào buffer
   signal(mutex);
   signal(full);
} while (TRUE);
```

Tiến trình đọc *Q*:

```
do {
   wait(full);
   wait(mutex);
   // Đọc một phần tử ra
   // khỏi buffer
   signal(mutex);
   signal(empty);
} while (TRUE);
```

28

Bài toán tiến trình đọc - ghi

- Thuật ngữ: the reader-writer problem
- Tình huống: Nhiều tiến trình cùng thao tác trên một cơ sở dữ liệu trong đó
  - Một vài tiến trình chỉ đọc dữ liệu (ký hiệu: reader)
  - Một số tiến trình vừa đọc vừa ghi (ký hiệu: writer)
- Khi có đọc/ghi đồng thời của nhiều tiến trình trên cùng một cơ sở dữ liệu, có 2 bài toán:
  - Bài toán 1: reader không phải chờ, trừ khi writer đã được phép ghi vào CSDL (hay các reader không loại trừ lẫn nhau khi đọc)
  - Bài toán 2: Khi writer đã sẵn sàng ghi, nó sẽ được ghi trong thời gian sớm nhất (nói cách khác khi writer đã sẵn sàng, không cho phép các reader đọc dữ liệu)

29

Bài toán tiến trình đọc-ghi số 1

- Sử dụng các semaphore với giá trị khởi tạo: *wrt* (1), *mutex* (1)
- Sử dụng biến *rcount* (khởi tạo 0) để đếm số lượng reader đang đọc dữ liệu
- *wrt*: Đảm bảo loại trừ lẫn nhau khi writer ghi
- *mutex*: Đảm bảo loại trữ lẫn nhau khi cập nhật biến *rcount*

30

Bài toán tiến trình đọc-ghi số 1

- **Tiến trình writer P_w:**

```
do {
    wait(wrt);
    // Thao tác ghi đang được
    // thực hiện
    signal(wrt);
while (TRUE);
```

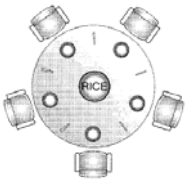
- **Tiến trình reader P_r:**

```
do {
    wait(mutex);
    rcount++;
    if (rcount==1) wait(wrt);
    signal(mutex);
    // Thực hiện phép đọc
    wait(mutex);
    rcount--;
    if (rcount==0) signal(wrt);
    signal(mutex);
} while (TRUE);
```

31

Bữa ăn tối của các triết gia

- Thuật ngữ: the dining-philosophers problem
- Có 5 triết gia, 5 chiếc đũa, 5 bát cơm và một âu cơm bố trí như hình vẽ
- Đây là bài toán cổ điển và là ví dụ minh họa cho một lớp nhiều bài toán tương tranh (concurrency): *Nhiều tiến trình khai thác nhiều tài nguyên chung*



32

Bữa ăn tối của các triết gia

- Các triết gia chỉ làm 2 việc: Ăn và suy nghĩ
  - Suy nghĩ: Không ảnh hưởng đến các triết gia khác, đũa, bát và âu cơm
  - Để ăn: Mỗi triết gia phải có đủ 2 chiếc đũa gần nhất ở bên phải và bên trái mình; chỉ được lấy 1 chiếc đũa một lần và không được phép lấy đũa từ tay triết gia khác
  - Khi ăn xong: Triết gia bỏ cả hai chiếc đũa xuống bàn và tiếp tục suy nghĩ

33

Giải pháp cho bài toán Bữa ăn...

- Biểu diễn 5 chiếc đũa qua mảng semaphore:

  semaphore chopstick[5];

  các semaphore được khởi tạo giá trị 1
- Mã lệnh của triết gia như hình bên
- Mã lệnh này có thể gây bế tắc (deadlock) nếu cả 5 triết gia đều lấy được 1 chiếc đũa và chờ để lấy chiếc còn lại nhưng không bao giờ lấy được!!

- Mã lệnh của triết gia *i*:

```
do {
    wait(chopstick[i]);
    wait(chopstick[(i+1)%5];
    // Ăn...
    signal(chopstick[i]);
    signal(chopstick[(i+1)%5];
    // Suy nghĩ...
} while (TRUE);
```

34

Một số giải pháp tránh bế tắc

- Chỉ cho phép nhiều nhất 4 triết gia đồng thời lấy đũa, dẫn đến có ít nhất 1 triết gia lấy được 2 chiếc đũa
- Chỉ cho phép lấy đũa khi cả hai chiếc đũa bên phải và bên trái đều nằm trên bàn
- Sử dụng giải pháp bất đối xứng: Triết gia mang số lẻ lấy chiếc đũa đầu tiên ở bên trái, sau đó chiếc đũa ở bên phải; triết gia mang số chẵn lấy chiếc đũa đầu tiên ở bên phải, sau đó lấy chiếc đũa bên trái

35

Hạn chế của semaphore

- Mặc dù semaphore cho ta cơ chế đồng bộ hóa tiện lợi song sử dụng semaphore không đúng cách có thể dẫn đến bế tắc hoặc lỗi do trình tự thực hiện của các tiến trình
- Trong một số trường hợp: khó phát hiện bế tắc hoặc lỗi do trình tự thực hiện khi sử dụng semaphore không đúng cách
- Sử dụng không đúng cách gây ra bởi *lỗi lập trình* hoặc do *người lập trình không cộng tác*

36

Ví dụ hạn chế của semaphore (1)

- Xét ví dụ về đoạn mã găng:

Mã đúng:	Mã sai:

...
wait(mutex);
// Đoạn mã găng
signal(mutex);
...
	...
	signal(mutex);
	// Đoạn mã găng
	wait(mutex);
	...
	• Đoạn mã sai này gây ra vi phạm điều kiện loại trừ lẫn nhau

37

Ví dụ hạn chế của semaphore (2)

- Xét ví dụ về đoạn mã găng:

Mã đúng:	Mã sai:

...
wait(mutex);
// Đoạn mã găng
signal(mutex);
...
	...
	wait(mutex);
	// Đoạn mã găng
	wait(mutex);
	...
	• Đoạn mã sai này gây ra bế tắc

38

Ví dụ hạn chế của semaphore (3)

- Nếu người lập trình quên các toán tử wait() hoặc signal() trong trong các đoạn mã găng, hoặc cả hai thì có thể gây ra:
  - Bế tắc
  - Vi phạm điều kiện loại trừ lẫn nhau

39

Ví dụ hạn chế của semaphore (4)

Tiến trình P_1	Tiến trình P_2

...
wait(S);
wait(Q);
...
signal(S);
signal(Q);
	...
	wait(Q);
	wait(S);
	...
	signal(Q);
	signal(S);

- Hai tiến trình P_1 và P_2 đồng thời thực hiện sẽ dẫn tới bế tắc

40

Cơ chế monitor

41

Thông tin tham khảo

- Per Brinch Hansen (người Đan Mạch) là người đầu tiên đưa ra khái niệm và cài đặt monitor năm 1972
- Monitor được sử dụng lần đầu tiên trong ngôn ngữ lập trình Concurrent Pascal

Per Brinch Hansen
(1938-2007)

42

Monitor là gì?

- Thuật ngữ monitor: *giám sát*
- Định nghĩa không hình thức: Là một loại construct trong ngôn ngữ bậc cao dùng để phục vụ các thao tác đồng bộ hóa
- Monitor được nghiên cứu, phát triển để khắc phục các hạn chế của semaphore như đã nêu trên

43

Định nghĩa tổng quát

- Monitor là một cách tiếp cận để đồng bộ hóa các tác vụ trên máy tính khi phải sử dụng các tài nguyên chung. Monitor thường gồm có:
  - Tập các procedure thao tác trên tài nguyên chung
  - Khóa loại trừ lẫn nhau
  - Các biến tương ứng với các tài nguyên chung
  - Một số các giả định bất biến nhằm tránh các tình huống tương tranh
- Trong bài này: Nghiên cứu một loại cấu trúc monitor: Kiểu monitor (monitor type)

44

Monitor type

- Một kiểu (type) hoặc kiểu trừu tượng (abstract type) gồm có các dữ liệu *private* và các phương thức *public*
- Monitor type được đặc trưng bởi tập các toán tử của người sử dụng định nghĩa
- Monitor type có các biến xác định các trạng thái; mã lệnh của các procedure thao tác trên các biến này

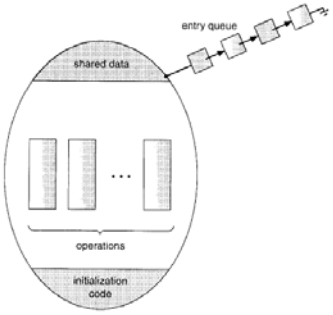
45

Cấu trúc một monitor type

```
monitor tên_monitor {
   // Khai báo các biến chung
   procedure P1(...) { ...
   }
   procedure P2(...) { ...
   }
   ...
   procedure Pn(...) { ...
   }
   initialization_code (..) { ...
   }
}
```

46

Minh họa cấu trúc monitor



47

Cách sử dụng monitor

- Monitor được cài đặt sao cho *chỉ có một tiến trình được hoạt động trong monitor* (*loại trừ lẫn nhau*). Người lập trình không cần viết mã lệnh để đảm bảo điều này
- Monitor như định nghĩa trên chưa đủ mạnh để xử lý mọi trường hợp đồng bộ hóa. Cần thêm một số cơ chế "tailor-made" về đồng bộ hóa
- Các trường hợp đồng bộ hóa "tailor-made": sử dụng kiểu *condition*.

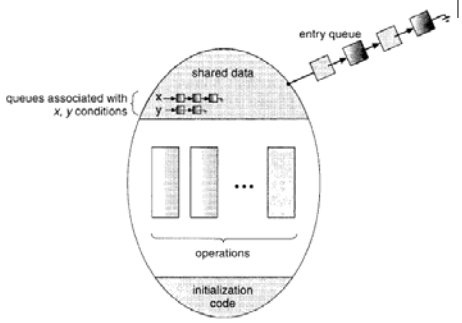
48

Kiểu condition

- Khai báo:
  condition x, y; // x, y là các biến kiểu condition
- Sử dụng kiểu condition: Chỉ có 2 toán tử là wait và signal
  - x.wait(): tiến trình gọi đến x.wait() sẽ được chuyển sang trạng thái chờ (wait hoặc suspend)
  - x.signal(): tiến trình gọi đến x.signal() sẽ khôi phục việc thực hiện (wakeup) một tiến trình đã gọi đến x.wait()

49

Monitor có kiểu condition



50

Đặc điểm của x.signal()

- x.signal() chỉ đánh thức duy nhất một tiến trình đang chờ
- Nếu không có tiến trình chờ, x.signal() không có tác dụng gì
- x.signal() khác với signal trong semaphore cổ điển: signal cổ điển luôn làm thay đổi trạng thái (giá trị) của semaphore

51

Signal wait/continue

- Giả sử có hai tiến trình P và Q:
  - *Q* gọi đến x.wait(), sau đó *P* gọi đến x.signal()
  - *Q* được phép tiếp tục thực hiện (wakeup)
- Khi đó *P* phải vào trạng thái wait vì nếu ngược lại thì *P* và *Q* cùng thực hiện trong monitor
- Khả năng xảy ra:
  - Signal-and-wait: *P* chờ đến khi *Q* rời monitor hoặc chờ một điều kiện khác (*)
  - Signal-and-continue: *Q* chờ đến khi *P* rời monitor hoặc chờ một điều kiện khác

52

Bài toán Ăn tối.. với monitor

- Giải quyết bài toán Ăn tối của các triết gia với monitor để không xảy ra bế tắc khi hai triết gia ngồi cạnh nhau cùng lấy đũa để ăn
- Trạng thái của các triết gia:
  enum {thinking, hungry, eating} state[5];
- Triết gia *i* chỉ có thể ăn nếu cả hai người ngồi cạnh ông ta không ăn:
  (state[(i+4)%5]!=eating) and (state[(i+1)%5]!=eating)
- Khi triết gia *i* không đủ điều kiện để ăn: cần có biến condition: condition self[5];

53

Monitor của bài toán Ăn tối...

```
monitor dp {
   enum {thinking, hungry, eating} state[5];
   condition self[5];
   void pickup(int i) {
        state[i] = hungry;
        test(i);
        if (state[i] != eating) self[i].wait();
   }
}
```

54

Monitor của bài toán Ăn tối...

```
void putdown(int i) {
    state[i] = thinking;
    test((i+4)%5);
    test((i+1)%5);
}
initialization_code() {
    for (int i=0;i<5;i++) state[i] = thinking;
}
```

55

Monitor của bài toán Ăn tối...

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

56

Đọc thêm ở nhà

- Khái niệm về miền găng (critical region)
- Cơ chế monitor của Java:

```
public class XYZ {
    ...
    public synchronized void safeMethod() {
      ...
    }
}
```

- Toán tử wait() và notify() trong java.util.package (tương tự toán tử wait() và signal())
- Cách cài đặt monitor bằng semaphore

57

Tóm tắt

- Khái niệm đồng bộ hóa
- Khái niệm đoạn mã găng, ba điều kiện của đoạn mã găng
- Khái niệm semaphore, semaphore nhị phân
- Hiện tượng bế tắc do sử dụng sai semaphore
- Một số bài toán cổ điển trong đồng bộ hóa
- Miền găng
- Cơ chế monitor

58

Bài tập

- Chỉ ra điều kiện nào của đoạn mã găng bị vi phạm trong đoạn mã găng sau của P_i:

```
do {
   while (turn != i) ;
   CS_i;
   turn = j;
   REMAIN_i;
} while (1);
```

59

Bài tập

- Cài đặt giải pháp cho bài toán Bữa ăn tối của các triết gia trong Java bằng cách sử dụng synchronized, wait() và notify()
- Giải pháp monitor cho bài toán Bữa ăn tối... tránh được bế tắc, nhưng có thể xảy ra trường hợp tất cả các triết gia đều không được ăn. Hãy chỉ ra trường hợp này và tìm cách giải quyết bằng cơ chế monitor
- Chú ý: Sinh viên cần làm bài tập để hiểu tốt hơn về đồng bộ hóa

60

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Bế tắc (Deadlock)

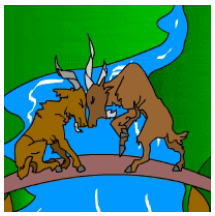
2

Định nghĩa

- Bế tắc là tình huống xuất hiện khi hai hay nhiều "hành động" phải chờ một hoặc nhiều hành động khác để kết thúc, nhưng không bao giờ thực hiện được
- Máy tính: Bế tắc là tình huống xuất hiện khi hai tiến trình phải chờ đợi nhau giải phóng tài nguyên hoặc nhiều tiến trình chờ sử dụng các tài nguyên theo một "vòng tròn" (circular chain)

3

Hai con dê qua cầu: Bế tắc



4

Bế tắc giao thông tại ngã tư

5

Bế tắc trong máy tính

- Tiến trình A:

```
{
   …
   Khóa file F1;
   ...
   Mở file F2;
   …
   Đóng F1 (mở khóa F1);
}
```

- Tiến trình B

```
{
   …
   Khóa file F2;
   ...
   Mở file F1;
   …
   Đóng F1 (mở khóa F1);
}
```

6

Qui trình sử dụng tài nguyên

- Một tiến trình thường sử dụng tài nguyên theo các bước tuần tự sau:
  - Xin phép sử dụng (request)
  - Sử dụng tài nguyên (use)
  - Giải phóng tài nguyên sau khi sử dụng (release)

7

Điều kiện cần để có bế tắc

- Bế tắc xuất hiện nếu 4 điều kiện sau xuất hiện đồng thời (điều kiện cần):
  - C1: Loại trừ lẫn nhau (mutual exclusion)
  - C2: Giữ và chờ (hold and wait)
  - C3: Không có đặc quyền (preemption)
  - C4: Chờ vòng (circular wait)

8

C1: Loại trừ lẫn nhau

- Một tài nguyên bị chiếm bởi một tiến trình, và không tiến trình nào khác có thể sử dụng tài nguyên này

9

C2: Giữ và chờ

- Một tiến trình giữ ít nhất một tài nguyên và chờ một số tài nguyên khác rỗi để sử dụng. Các tài nguyên này đang bị một tiến trình khác chiếm giữ

10

C3: Không có đặc quyền

- Tài nguyên bị chiếm giữ chỉ có thể rỗi khi tiến trình "tự nguyện" giải phóng tài nguyên sau khi đã sử dụng xong.

11

C3: Chờ vòng

- Một tập tiến trình $\{P_0, P_1, ..., P_n\}$ có xuất hiện điều kiện "chờ vòng" nếu P_0 chờ một tài nguyên do P_1 chiếm giữ, P_1 chờ một tài nguyên khác do P_2 chiếm giữ, ..., P_{n-1} chờ tài nguyên do P_n chiếm giữ và P_n chờ tài nguyên do P_0 chiếm giữ

12

Đồ thị cấp phát tài nguyên

- Thuật ngữ: Resource allocation graph
- Để mô tả một cách chính xác bế tắc, chúng ta sử dụng đồ thị có hướng gọi là "đồ thị cấp phát tài nguyên" $G=(V, E)$ với V là tập đỉnh, E là tập cung
- E được chia thành hai tập con $P=\{P_0, P_1, ..., P_n\}$ là tập các tiến trình trong hệ thống và $R=\{R_0, R_1, ..., R_m\}$ là tập các loại tài nguyên trong hệ thống thỏa mãn $P \cup R=E$ và $P \cap R= \varnothing$

13

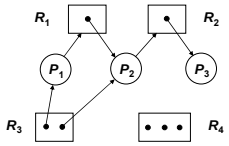
Đồ thị cấp phát tài nguyên

- Cung có hướng từ tiến trình P_i đến tài nguyên R_j, ký hiệu là $P_i \rightarrow R_j$ có ý nghĩa: Tiến trình P_i yêu cầu một *thể hiện* của R_j. Ta gọi $P_i \rightarrow R_j$ là *cung yêu cầu* (*request edge*)
- Cung có hướng từ tài nguyên R_j đến tiến trình P_i ký hiệu là $R_j \rightarrow P_i$ có ý nghĩa: Một thể hiện của tài nguyên Rj đã được cấp phát cho tiến trình Pi. Ta gọi $R_j \rightarrow P_i$ là *cung cấp phát* (*asignment edge*)

14

Đồ thị cấp phát tài nguyên

- Ký hiệu hình vẽ:
  - P_i là hình tròn
  - R_j là các hình chữ nhật với mỗi chấm bên trong là số lượng các thể hiện của tài nguyên
- Minh họa đồ thị cấp phát tài nguyên:



15

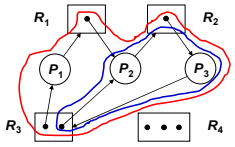
Đồ thị cấp phát tài nguyên

- Nếu không có chu trình trong đồ thị cấp phát tài nguyên: Không có bế tắc. Nếu có chu trình: *Có thể* xảy ra bế tắc.
- Nếu trong một chu trình trong đồ thị cấp phát tài nguyên, mỗi loại tài nguyên chỉ có đúng một thể hiện: Bế tắc đã xảy ra (Điều kiện cần và đủ)
- Nếu trong một chu trình trong đồ thị cấp phát tài nguyên một số tài nguyên có nhiều hơn một thể hiện: Có thể xảy ra bế tắc (Điều kiện cần nhưng không đủ)

16

Ví dụ chu trình dẫn đến bế tắc

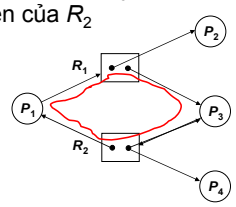
- Giả sử P_3 yêu cầu một thể hiện của R_3
- Khi đó có 2 chu trình xuất hiện:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$, và
  - $P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$
- Khi đó các tiến trình P_1, P_2, P_3 bị bế tắc



17

Ví dụ chu trình không dẫn đến bế tắc

- Chu trình: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- Bế tắc không xảy ra vì P_4 có thể giải phóng một thể hiện tài nguyên R_2 và P_3 sẽ được cấp phát một thể hiện của R_2



18

Các phương pháp xử lý bế tắc

Các phương pháp xử lý bế tắc

- Một cách tổng quát, có 3 phương pháp:
  - Sử dụng một giao thức để hệ thống không bao giờ rơi vào trạng thái bế tắc: *Deadlock prevention* (*ngăn chặn bế tắc*) hoặc *Deadlock avoidance* (*tránh bế tắc*)
  - Có thể cho phép hệ thống bị bế tắc, phát hiện bế tắc và khắc phục nó
  - Bỏ qua bế tắc, xem như bế tắc không bao giờ xuất hiện trong hệ thống (*Giải pháp này dùng trong nhiều hệ thống, ví dụ Unix, Windows*!!)

Ngăn chặn bế tắc (Deadlock prevention)

Giới thiệu

- Ngăn chặn bế tắc (deadlock prevention) là phương pháp xử lý bế tắc, không cho nó xảy ra bằng cách làm cho ít nhất một điều kiện cần của bế tắc là C1, C2, C3 hoặc C4 không được thỏa mãn (không xảy ra)
- Ngăn chặn bế tắc theo phương pháp này có tính chất tĩnh (statically)

Ngăn chặn "loại trừ lẫn nhau"

- C1 (Loại trừ lẫn nhau): là điều kiện bắt buộc cho các tài nguyên không sử dụng chung được → *Khó làm cho C1 không xảy ra* vì các hệ thống luôn có các tài nguyên không thể sử dụng chung được

Ngăn chặn "giữ và chờ"

- C2 (Giữ và chờ): Có thể làm cho C2 không xảy ra bằng cách đảm bảo:
  - Một tiến trình luôn yêu cầu cấp phát tài nguyên chỉ khi nó không chiếm giữ bất kỳ một tài nguyên nào, hoặc
  - Một tiến trình chỉ thực hiện khi nó được cấp phát toàn bộ các tài nguyên cần thiết

Ngăn chặn "không có đặc quyền"

- Để ngăn chặn không cho điều kiện này xảy ra, có thể sử dụng giao thức sau:
  - Nếu tiến trình P (đang chiếm tài nguyên R_1, ..., R_{n-1}) yêu cầu cấp phát tài nguyên R_n nhưng không được cấp phát ngay (có nghĩa là P phải chờ) thì tất cả các tài nguyên R_1, ..., R_{n-1} phải được "thu hồi"
  - Nói cách khác, R_1, ..., R_{n-1} phải được "giải phóng" một cách áp đặt, tức là các tài nguyên này phải được đưa vào danh sách các tài nguyên mà P đang chờ cấp phát.

25

Ngăn chặn "không có đặc quyền": mã lệnh

Tiến trình P yêu cầu cấp phát tài nguyên R_1, ..., R_{n-1}

if (R_1, ..., R_{n-1} rỗi)

then cấp phát tài nguyên cho P

else if ({R_i...R_j} được cấp phát cho Q và Q đang trong trạng thái chờ một số tài nguyên S khác)

then thu hồi {R_i...R_j} và cấp phát cho P

else đưa P vào trạng thái chờ tài nguyên R_1, ..., R_{n-1}

26

Ngăn chặn "chờ vòng"

- Một giải pháp ngăn chặn chờ vòng là đánh số thứ tự các tài nguyên và bắt buộc các tiến trình yêu cầu cấp phát tài nguyên theo số thứ tự tăng dần
- Giả sử có các tài nguyên {R_1, ..., R_n}. Ta gán cho mỗi tài nguyên một số nguyên dương duy nhất qua một ánh xạ 1-1

  $f : R \rightarrow N$, với N là tập các số tự nhiên

  Ví dụ: f(ổ cứng) = 1, f(băng từ) = 5, f(máy in) = 11

27

Ngăn chặn "chờ vòng"

- Giao thức ngăn chặn chờ vòng:
  - Khi tiến trình P không chiếm giữ tài nguyên nào, nó có thể yêu cầu cấp phát nhiều *thể hiện* của *một* tài nguyên R_i bất kỳ
  - Sau đó P chỉ có thể yêu cầu các thể hiện của tài nguyên R_j nếu và chỉ nếu $f(R_j) > f(R_i)$. Một cách khác, nếu P muốn yêu cầu cấp phát tài nguyên R_j, nó đã giải phóng tất cả các tài nguyên R_i thỏa mãn $f(R_i) \geq f(R_j)$
  - Nếu P cần được cấp phát nhiều loại tài nguyên, P phải *lần lượt* yêu cầu các thể hiện của *từng* tài nguyên đó

28

Chứng minh giải pháp ngăn chặn chờ vòng

- Sử dụng chứng minh phản chứng
- Giả sử giải pháp ngăn chặn gây ra chờ vòng {P_0, P_1, ..., P_n} trong đó P_i chờ tài nguyên R_i bị chiếm giữ bởi $P_{(i+1) \bmod n}$
- Vì P_{i+1} đang chiếm giữ R_i và yêu cầu R_{i+1}, do đó $f(R_i) < f(R_{(i+1) \bmod n})$ $\forall i$, có nghĩa là ta có:
  - $f(R_0) < f(R_1) < f(R_2) < .. < f(R_n) < f(R_0)$
  - Mâu thuẫn! $\rightarrow$ Giải pháp được chứng minh

29

Ưu nhược điểm của ngăn chặn giải pháp bế tắc

- Ưu điểm: ngăn chặn bế tắc (deadlock prevention) là phương pháp tránh được bế tắc bằng cách làm cho điều kiện cần không được thỏa mãn
- Nhược điểm:
  - Giảm khả năng tận dụng tài nguyên và giảm thông lượng của hệ thống
  - Không mềm dẻo

30

Tránh bế tắc (Deadlock avoidance)

31

Giới thiệu

- Tránh bế tắc là phương pháp sử dụng thêm các thông tin về phương thức yêu cầu cấp phát tài nguyên để ra quyết định cấp phát tài nguyên sao cho bế tắc không xảy ra.
- Có nhiều thuật toán theo hướng này
- Thuật toán đơn giản nhất và hiệu quả nhất là: Mỗi tiến trình *P* đăng ký số thể hiện của mỗi loại tài nguyên mà *P* sẽ sử dụng. Khi đó hệ thống sẽ có đủ thông tin để xây dựng thuật toán cấp phát không gây ra bế tắc

32

Giới thiệu

- Các thuật toán như vậy kiểm tra *trạng thái cấp phát tài nguyên* một cách "động" để đảm bảo điều kiện chờ vòng không xảy ra
- Trạng thái cấp phát tài nguyên được xác định bởi số lượng tài nguyên rỗi, số lượng tài nguyên đã cấp phát và số lượng lớn nhất các yêu cầu cấp phát tài nguyên của các tiến trình
- Hai thuật toán sẽ nghiên cứu: *Thuật toán đồ thị cấp phát tài nguyên* và thuật toán *banker*

33

Trạng thái an toàn (safe-state)

- Một trạng thái (cấp phát tài nguyên) được gọi là an toàn nếu hệ thống có thể cấp phát tài nguyên cho các tiến trình theo một thứ tự nào đó mà vẫn tránh được bế tắc, hay
- Hệ thống ở trong trạng thái an toàn nếu và chỉ nếu tồn tại một *thứ tự an toàn* (safe-sequence)

34

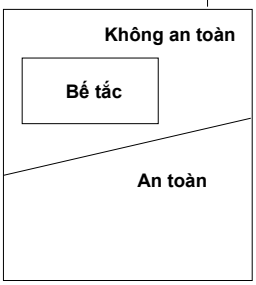
Thứ tự an toàn

- Thứ tự các tiến trình $<P_1,...,P_n>$ gọi là một thứ tự an toàn (safe-sequence) cho trạng thái cấp-phát hiện-tại nếu với mỗi P_i, yêu cầu cấp phát tài nguyên của P_i vẫn có thể được thỏa mãn căn cứ vào trạng thái của:
  - Tất cả các tài nguyên rỗi hiện có, và
  - Tất cả các tài nguyên đang bị chiếm giữ bởi tất cả các P_j $\forall j<i$.

35

Các trạng thái an toàn, không an toàn và bế tắc

- Trạng thái an toàn không là trạng thái bế tắc
- Trạng thái bế tắc là trạng thái không an toàn
- Trạng thái không an toàn có thể là trạng thái bế tắc hoặc không



36

Ví dụ trạng thái an toàn, bế tắc

- Xét một hệ thống có 12 tài nguyên là 12 băng từ và 3 tiến trình P_0, P_1, P_2 với các yêu cầu cấp phát:
  - P_0 yêu cầu nhiều nhất 10 băng từ
  - P_1 yêu cầu nhiều nhất 4 băng từ
  - P_2 yêu cầu nhiều nhất 9 băng từ
- Giả sử tại một thời điểm t_0, P_0 đang chiếm 5 băng từ, P_1 và P_2 mỗi tiến trình chiếm 2 băng từ. Như vậy có 3 băng từ rỗi

37

Ví dụ trạng thái an toàn, bế tắc

	Yêu cầu nhiều nhất	Yêu cầu hiện tại
P_0	10	5
P_1	4	2
P_2	9	2

- Tại thời điểm t_0, hệ thống ở trạng thái an toàn
- Thứ tự <P_1, P_0, P_2> thỏa mãn điều kiện an toàn
- Giả sử ở thời điểm t_1, P_2 có yêu cầu và được cấp phát 1 băng từ: Hệ thống không ở trạng thái an toàn nữa... -> quyết đinh cấp tài nguyên cho P_2 là sai.

38

Thuật toán đồ thị cấp phát tài nguyên

- Giả sử các tài nguyên chỉ có 1 thể hiện
- Sử dụng đồ thị cấp phát tài nguyên như ở slide 16 và thêm một loại cung nữa là *cung báo trước* (claim)
- Cung báo trước $P_i \rightarrow R_j$ chỉ ra rằng P_i có thể yêu cầu cấp phát tài nguyên R_j, được biểu diễn trên đồ thị bằng các đường nét đứt
- Khi tiến trình P_i yêu cầu cấp phát tài nguyên R_j, đường nét đứt trở thành đường nét liền

39

Thuật toán đồ thị cấp phát tài nguyên

- Chú ý rằng các tài nguyên phải được thông báo trước khi tiến trình thực hiện
- Các cung báo trước sẽ phải có trên đồ thị cấp phát tài nguyên
- Tuy nhiên có thể giảm nhẹ điều kiện: cung thông báo $P_i \rightarrow R_j$ được thêm vào đồ thị nếu tất cả các cung gắn với P_i đều là cung thông báo

40

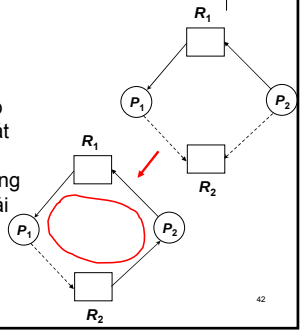
Thuật toán đồ thị cấp phát tài nguyên

- Giả sử P_i yêu cầu cấp phát R_j. Yêu cầu này chỉ có thể được chấp nhận nếu ta chuyển cung báo trước $P_i \rightarrow R_j$ thành cung cấp phát $R_j \rightarrow P_i$ và không tạo ra một chu trình
- Chúng ta kiểm tra bằng cách sử dụng thuật toán phát hiện chu trình trong đồ thị: Nếu có n tiến trình trong hệ thống, thuật toán phát hiện chu trình có độ phức tạp tính toán $O(n^2)$
- Nếu không có chu trình: Cấp phát->trạng thái an toàn, ngược lại: Trạng thái không an toàn

41

Ví dụ

- Giả sử P_1 yêu cầu cấp phát R_2
- Mặc dù R_2 rỗi nhưng chúng ta không thể cấp phát R_2, vì nếu cấp phát ta sẽ có chu trình trong đồ thị và gây ra chờ vòng → Hệ thống ở trạng thái không an toàn



42

Thuật toán banker

- Thuật toán đồ thị phân phối tài nguyên không áp dụng được cho các hệ thống có những tài nguyên có nhiều thể hiện
- Thuật toán *banker* được dùng cho các hệ có tài nguyên nhiều thể hiện, nó kém hiệu quả hơn thuật toán đồ thị phân phối tài nguyên
- Thuật toán banker có thể dùng trong ngân hàng: Không bao giờ cấp phát tài nguyên (tiền) gây nên tình huống sau này không đáp ứng được nhu cầu của tất cả các khách hàng

43

Ký hiệu dùng trong banker

- Tài nguyên rỗi: Vector *m* thành phần *Available*, *Available*[*j*]=*k* nghĩa là có *k* thể hiện của R_j rỗi
- Max: Ma trận *nxm* xác định yêu cầu tài nguyên max của mỗi tiến trình. *Max*[*i*][*j*]=*k* có nghĩa là tiến trình P_i yêu cầu nhiều nhất *k* thể hiện của tài nguyên R_j.

44

Ký hiệu dùng trong banker

- Cấp phát: Ma trận *nxm* xác định số thể hiện của các loại tài nguyên đã cấp phát cho mỗi tiến trình. *Allocation*[*i*][*j*]=*k* có nghĩa là tiến trình P_i được cấp phát *k* thể hiện của R_j.
- Cần thiết: Ma trận *nxm* chỉ ra số lượng thể hiện của các tài nguyên mỗi tiến trình cần cấp phát tiếp. *Need*[*i*][*j*]=*k* có nghĩa là tiến trình P_i còn có thể cần thêm *k* thể hiện nữa của tài nguyên R_j.

45

Ký hiệu dùng trong banker

- Số lượng và giá trị các biến trên biến đổi theo trạng thái của hệ thống
- Qui ước: Nếu hai vector *X*, *Y* thỏa mãn *X*[*i*]≤*Y*[*i*] ∀*i* thì ta ký hiệu *X*≤*Y*.
- Giả sử *Work* và *Finish* là các vector *m* và *n* thành phần.
- *Request*[*i*] là vector yêu cầu tài nguyên của tiến trình P_i. *Request*[*i*][*j*]=*k* có nghĩa là tiến trình P_i yêu cầu *k* thể hiện của tài nguyên R_j

46

Thuật toán trạng thái an toàn

1. Khởi tạo *Work*=*Available* và *Finish*[*i*]=false ∀*i*=1..*n*
2. Tìm *i* sao cho *Finish*[*i*]==false và *Need*[*i*]≤*Work*
   Nếu không tìm được *i*, chuyển đến bước 4
3. *Work*=*Work*+*Allocation*[i], *Finish*[*i*]=true
   Chuyển đến bước 2
4. Nếu *Finish*[*i*]==true ∀*i* thì hệ thống ở trạng thái an toàn

- Độ phức tạp tính toán của thuật toán trạng thái an toàn: $O(m.n^2)$

47

Thuật toán yêu cầu tài nguyên

1. Nếu *Request*[*i*]≤*Need*[*i*], chuyển đến bước 2
   Ngược lại thông báo lỗi (không có tài nguyên rỗi)
2. Nếu *Request*[*i*]≤*Available*, chuyển đến bước 3.
   Ngược lại P_i phải chờ vì không có tài nguyên
3. Nếu việc thay đổi trạng thái giả định sau đây:
   Available=*Availalble*-*Request*[i]
   Allocation=*Allocation*+*Request*[i]
   Need[i]=*Need*[i]-*Request*[i]
   đưa hệ thống vào trạng thái an toàn thì cấp phát tài nguyên cho P_i, ngược lại P_i phải chờ *Request*[*i*] và trạng thái của hệ thống được khôi phục như cũ

48

Ví dụ banker

- Xét một hệ thống các tiến trình và tài nguyên như sau:

	Allocation			Max			Available			Need		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	3	3	2	7	4	3
P1	2	0	0	3	2	2				1	2	2
P2	3	0	2	9	0	2				6	0	0
P3	2	1	1	2	2	2				0	1	1
P4	0	0	2	4	3	3				4	3	1

49

Ví dụ banker

- Hệ thống hiện đang ở trạng thái an toàn
- Thứ tự $<P_1,P_3,P_4,P_2,P_0>$ thỏa mãn tiêu chuẩn an toàn
- Giả sử P_1 có yêu cầu: *Request*[1]=(1,0,2)
- Để quyết định xem có cấp phát tài nguyên theo yêu cầu này không, trước hết ta kiểm tra *Request*[1]≤*Available*: (1,0,2)<(3,3,2): Đúng
- Giả sử yêu cầu này được cấp phát, khi đó trạng thái giả định của hệ thống là:

50

Ví dụ banker

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	3	3	2
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

- Thực hiện thuật toán trạng thái an toàn và thấy rằng thứ tự $<P_1,P_3,P_4,P_0,P_2>$. Do đó có thể cấp phát tài nguyên cho P_1 ngay.

51

Ví dụ banker

- Tuy nhiên, nếu hệ thống ở trạng thái sau thì
  - Yêu cầu (3,3,0) của P4 không thể cấp phát ngay vì các tài nguyên không rỗi
  - Yêu cầu (0,2,0) của P0 cũng không thể cấp phát ngay vì mặc dù các tài nguyên rỗi nhưng việc cấp phát sẽ làm cho hệ thống rơi vào trạng thái không an toàn
- **Bài tập**: Thực hiện kiểm tra và quyết định cấp phát hai yêu cầu trên

52

Phát hiện bế tắc

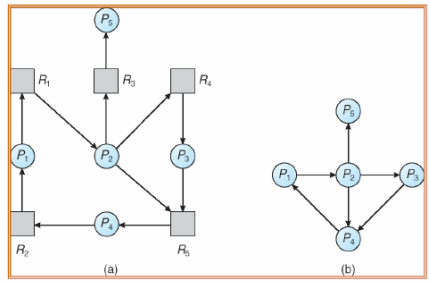
- Nếu không áp dụng phòng tránh hoặc ngăn chặn bế tắc thì hệ thống có thể bị bế tắc
- Khi đó:
  - Cần có thuật toán kiểm tra trạng thái để xem có bế tắc xuất hiện hay không
  - Thuật toán khôi phục nếu bế tắc xảy ra

53

Tài nguyên chỉ có một thể hiện

- Sử dụng thuật toán đồ thị chờ: Đồ thị chờ có được từ đồ thị cấp phát tài nguyên bằng cách xóa các đỉnh tài nguyên và nối các cung liên quan
  - Cung $P_i \rightarrow P_j$ có nghĩa là P_i đang chờ P_j giải phóng tài nguyên mà P_i cần
  - Cung $P_i \rightarrow P_j$ tồn tại trong đồ thị chờ nếu và chỉ nếu đồ thị cấp phát tài nguyên tương ứng có hai cung $P_i \rightarrow R_q$ và $R_q \rightarrow P_j$ với R_q là tài nguyên
  - Hệ thống có bế tắc nếu đồ thị chờ có chu trình
  - Để phát hiện bế tắc: Cần cập nhật đồ thị chờ và thực hiện định kỳ thuật toán phát hiện chu trình

54



55

Tài nguyên có nhiều thể hiện

- *Available*: Vector *m* thành phần chỉ ra số lượng thể hiện của mỗi loại tài nguyên
- *Allocation*: Ma trận *nxm* xác định số thể hiện của mỗi loại tài nguyên đang được cấp phát cho các tiến trình
- *Request*: Ma trận *nxm* xác định yêu cầu hiện tại của mỗi tiến trình. Nếu *Request*[*i*][*j*]=*k* thì tiến trình P_i yêu cầu cấp phát *k* thể hiện của tài nguyên R_j.

56

Tài nguyên có nhiều thể hiện

1. Giả sử *Work* và *Finish* là các vector *m* và *n* thành phần. Khởi tạo *Work*=*Available*. Với mỗi *i*=0..*n*-1 gán *Finish*[*i*]=false nếu *Allocation*[*i*]≠0, ngược lại gán *Finish*[*i*]=true
2. Tìm *i* sao cho *Finish*[*i*]==false và *Request*[*i*]≤*Work*. Nếu không tìm thấy *i*, chuyển đến bước 4
3. *Work*=*Work*+*Allocation*, *Finish*[*i*]=true; chuyển đến bước 2
4. Nếu *Finish*[*i*]==false với 0≤*i*≤*n*-1 thì hệ thống đang bị bế tắc (và tiến trình P_i đang bế tắc).

- Độ phức tạp tính toán của thuật toán: $O(m.n^2)$

57

Sử dụng thuật toán phát hiện

- Tần suất sử dụng phụ thuộc:
  - Tần suất xảy ra bế tắc
  - Bao nhiêu tiến trình bị ảnh hưởng bởi bế tắc?
- Sử dụng thuật toán phát hiện:
  - Định kỳ: Có thể có nhiều chu trình trong đồ thị, không biết được tiến trình/request nào gây ra bế tắc
  - Khi có yêu cầu cấp phát tài nguyên: Tốn tài nguyên CPU

58

Khôi phục khi có bế tắc

- Kết thúc tiến trình:
  - Kết thúc toàn bộ các tiến trình bị bế tắc (1)
  - Kết thúc từng tiến trình và dừng quá trình này khi bế tắc chấm dứt (2)
- Tiến trình bị kết thúc ở (2) căn cứ vào:
  - Độ ưu tiên
  - Thời gian đã thực hiện và thời gian còn lại
  - Số lượng và các loại tài nguyên đã sử dụng
  - Các tài nguyên cần cấp phát thêm
  - Số lượng các tiến trình phải kết thúc
  - Tiến trình là tương tác hay xử lý theo lô (batch)

59

Khôi phục khi có bế tắc

- Giải phóng tài nguyên một cách bắt buộc (preemption):
  - Chọn tài nguyên nào và tiến trình nào để thực hiện?
  - Khôi phục trạng thái của tiến trình đã chọn ở (1) như thế nào?
  - Làm thế nào để tránh tình trạng một tiến trình luôn bị bắt buộc giải phóng tài nguyên?

60

Tóm tắt

- Khái niệm bế tắc
- Các điều kiện cần để có bế tắc
- Đồ thị phân phối tài nguyên
- Các phương pháp xử lý bế tắc: Ngăn chặn và tránh bế tắc (thuật toán đồ thị cấp phát tài nguyên và thuật toán banker)
- Khôi phục khi bế tắc đã xảy ra: Kết thúc tiến trình và preemption

61

Bài tập

- Thực hiện lại ví dụ phát hiện bế tắc ở trang 264 trong giáo trình
- Làm bài tập số 7.1 trong giáo trình (trang 268)
- Làm bài tập số 7.11 trong giáo trình (trang 270)

62

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Quản lý bộ nhớ

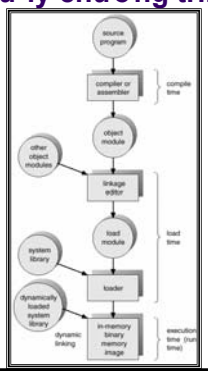
2

Giới thiệu

- Chương trình được HĐH đưa vào bộ nhớ, sau đó tạo tiến trình để thực hiện
- *Input queue* – Là hàng chờ các tiến trình trên đĩa đang chờ được đưa vào bộ nhớ để thực hiện
- Các chương trình của NSD phải qua một số bước chuẩn bị trước khi được thực hiện

3

Các bước xử lý chương trình NSD



4

Chuyển đổi địa chỉ

Có 3 cách chuyển đổi địa chỉ lệnh và dữ liệu của chương trình vào bộ nhớ:

- **Khi dịch chương trình (compile-time)**: Sinh mã có địa chỉ cố định; phải dịch lại nếu cần thay đổi địa chỉ.
- **Khi nạp chương trình (load-time)**: Phải sinh mã có thể định vị lại nếu như địa chỉ bộ nhớ không được biết ở thời điểm dịch chương trình
- **Khi thực hiện chương trình (execution-time)**: Ánh xạ địa chỉ khi chương trình được thực hiện nếu như tiến trình có thể chuyển giữa các segment bộ nhớ. Cần có hỗ trợ từ phần cứng (ví dụ thanh ghi *base* và *limit*)

5

Không gian địa chỉ logic (ảo) và địa chỉ vật lý (địa chỉ thật)

- Để quản lý bộ nhớ một cách hoàn chỉnh, cần có hai cách nhìn địa chỉ khác nhau:
  - *Địa chỉ logic* (*Logical address*) – sinh bởi CPU; còn gọi là địa chỉ ảo (*virtual address*).
  - *Địa chỉ vật lý* (*Physical address*); còn gọi là địa chỉ thật – sinh bởi đơn vị quản lý bộ nhớ
- Địa chỉ thật và ảo giống nhau trong lược đồ ánh xạ địa chỉ "compile-time" và "load-time" và khác nhau trong "execution-time".

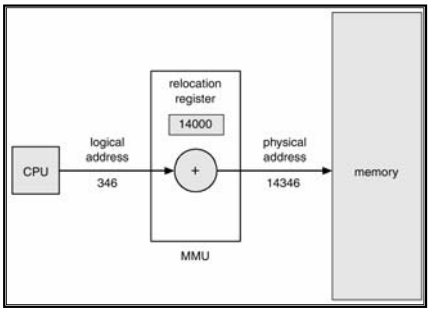
6

Đơn vị quản lý bộ nhớ (MMU)

- Là thiết bị phần cứng dùng để ánh xạ địa chỉ ảo sang địa chỉ vật lý
- Trong MMU, có thanh ghi relocation (định vị lại) dùng để tính toán địa chỉ thực (vật lý) từ địa ảo của một tiến trình của NSD
- Chương trình của NSD làm việc trên địa chỉ ảo và không bao giờ biết địa chỉ vật lý

7

Sử dụng thanh ghi relocation



8

Nạp chương trình động (Dynamic loading)

- Các hàm, thủ tục không được nạp cho đến khi được sử dụng (được gọi đến)
- Cách nạp động này sử dụng bộ nhớ hiệu quả hơn: Các hàm, thủ tục không dùng đến không bao giờ được nạp vào bộ nhớ
- Hữu ích khi có một đoạn mã lớn được sử dụng với tần suất thấp
- Không cần có các đặc điểm đặc biệt từ hệ điều hành về phần cứng/phần mềm

9

Liên kết động (dynamic linking) và thư viện chung (shared library)

- Liên kết chương trình được thực hiện khi chương trình được thực hiện.
- Một đoạn mã ngắn (stub) được dùng để định vị các hàm tương ứng đã được nạp sẵn trong bộ nhớ
- Stub được thay thế bằng địa chỉ của hàm/thủ tục cần thiết, sau đó thực hiện hàm/thủ tục đó
- HĐH cần kiểm tra các hàm/thủ tục đã được nạp chưa
- Liên kết động rất có lợi khi xây dựng các thư viện chung, khi sửa lỗi (các miếng vá – patch)

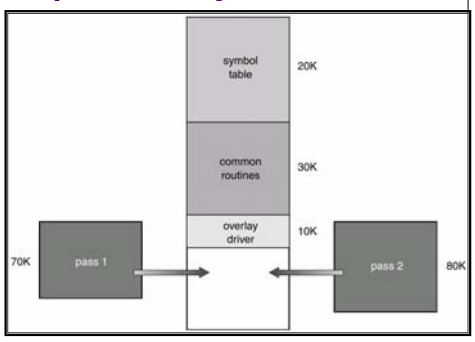
10

Overlays

- Chỉ lưu trong bộ nhớ các phần lệnh và dữ liệu phải sử dụng trong suốt quá trình thực hiện
- Sử dụng khi tiến trình có yêu cầu bộ nhớ lớn hơn dung lượng được cấp phát.
- Cài đặt bởi người sử dụng, lập trình overlays rất phức tạp

11

Ví dụ về overlays



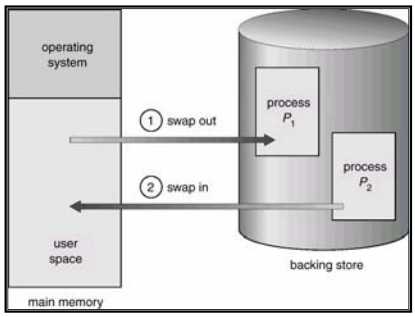
12

Swapping

- *Swapping*: Đưa một tiến trình ra *backing store* để lưu trữ tạm thời, sau đó đưa trở lại bộ nhớ trong để thực hiện.
  - *Backing store* – Vùng đĩa có tốc độ truy cập cao, đủ lớn để chứa được nhiều tiến trình của NSD, có thể truy cập trực tiếp
- *Roll out, roll in* – Phương án swap dành cho lập lịch có ưu tiên: Tiến trình ưu tiến thấp: *roll out*, ưu tiên cao: *roll in* để tiếp tục thực hiện
- Thời gian swap tỷ lệ thuận với dung lượng bộ nhớ được swap vào/ra
- UNIX, Linux, and Windows sử dụng swapping

13

Minh họa swapping



14

Cấp phát liên tục (Contiguous allocation)

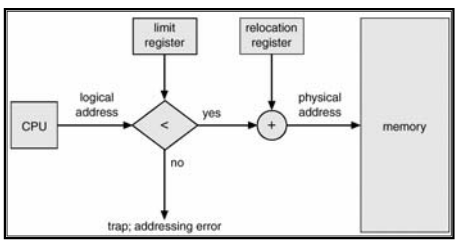
15

Cấp phát bộ nhớ liên tục

- Bộ nhớ trong thường được chia thành 2 phần:
  - Phần dành cho hệ điều hành (resident) thường dùng phần thấp của bộ nhớ với các ngắt
  - NSD dùng phần cao của bộ nhớ. Mỗi tiến trình được cấp phát một *vùng liên tục của bộ nhớ*
- Thanh ghi *relocation* dùng để bảo vệ các tiến trình của NSD và để tránh thay đổi mã và dữ liệu của HĐH
- Thanh ghi *relocation* chứa giá trị nhỏ nhất của địa chỉ vật lý, thanh ghi *limit* chứa độ lớn của miền địa chỉ ảo (*địa chỉ ảo* < *limit*)

16

Minh họa thanh ghi *relocation*, *limit*



17

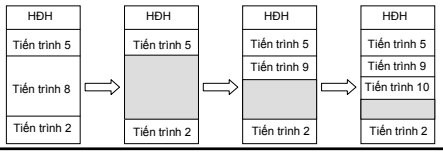
Cấp phát liên tục (tiếp): MFT

- Bộ nhớ được chia thành các khối với cỡ cố định, mỗi tiến trình được cấp phát một khối
- Khi tiến trình kết thúc, khối bộ nhớ đã cấp phát cho tiến trình được giải phóng để cấp phát cho tiến trình khác
- Mức độ đa chương trình bị hạn chế bởi các khối
- Cỡ của tiến trình bị hạn chế bởi cỡ của khối
- Các HĐH/máy tính sử dụng MFT: IBM/360

18

Cấp phát liên tục (tiếp): MVT

- Cấp phát MVT
  - *Hole* – khối bộ nhớ rỗi; các khối rỗi với kích cỡ khác nhau rải rác trong bộ nhớ
  - Một tiến trình sẽ được cấp phát một khối bộ nhớ đủ lớn để thực hiện
  - HĐH có thông tin về các khối đã cấp phát và khối rỗi



19

Các chiến lược cấp phát

- **First-fit**: Cấp phát khối nhớ đầu tiên thỏa mãn điều kiện.
- **Best-fit**: Cấp phát khối nhớ bé nhất thỏa mãn điều kiện: Phải duyệt toàn bộ danh sách khối nhớ
- **Worst-fit**: Cấp phát khối nhớ lớn nhất thỏa mãn điều kiện: Phải duyệt toàn bộ danh sách khối nhớ
- First-fit và best-fit tốt hơn worst-fit theo nghĩa tốc độ và tận dụng bộ nhớ

20

Vấn đề phân mảnh

- **External Fragmentation (*Phân mảnh ngoài*)**: Tổng dung lượng đáp ứng được nhu cầu cấp phát nhưng các khối không liên tục
- **Internal Fragmentation (*Phân mảnh trong*)** – Dung lượng bộ nhớ đã cấp phát cho tiến trình không được sử dụng hết
- Giảm phân mảnh ngoài: Compaction
  - Xáo trộn các khối để các khối nhớ rỗi nằm liên tục
  - Compaction chỉ thực hiện được khi relocation là động, và được thực hiện ở execution-time
- Ví dụ: Tiện ích Defragmentation của Windows

21

Phân trang (Paging)

22

Phân trang (paging)

- Phân trang là chiến lược cấp phát bộ nhớ cho phép không gian địa chỉ logic của một tiến trình có thể không liên tục; tiến trình được cấp phát bộ nhớ vật lý khi có bộ nhớ rỗi
- Bộ nhớ vật lý được chia thành các frame cỡ cố định, nhỏ (là lũy thừa của 2, ví dụ 512, 1024, 8192)
- Chia bộ nhớ ảo thành các khối *cùng cỡ* gọi là trang (page)
- HĐH có danh sách các frame rỗi
- Để thực hiện một chương trình cỡ *n* trang, cần tìm *n* frame rỗi để nạp chương trình
- Có một bảng trang để ánh xạ trang→frame
- Bảng trang: chung trong HĐH, mỗi tiến trình có một copy

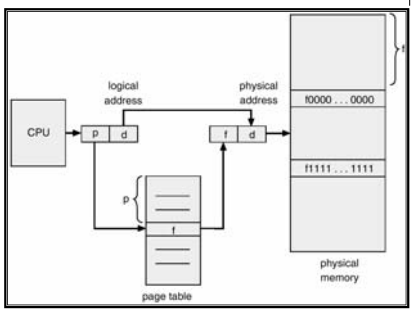
23

Cách đánh địa chỉ theo trang

- Địa chỉ được đánh một cách phân cấp:
  - *Số hiệu trang* (*Page number* - *p*) – Được sử dụng làm chỉ số đến phần tử trong bảng trang chứa địa chỉ cơ sở của các frame trong bộ nhớ vật lý
  - *Offset trang* (*Page offset* - *d*) – Địa chỉ tương đối trong trang
- Địa chỉ ảo có *m* bit, sử dụng *m-n* bit cao làm số hiệu trang và *n* bit thấp làm offset
- Không có *phân mảnh ngoài*, có *phân mảnh trong*:
  - Giảm cỡ trang→Giảm phân mảnh trong→Giảm hiệu năng
  - Tăng cỡ trang→Tăng hiệu suất→Tăng phân mảnh trong

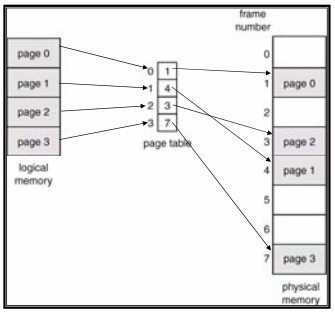
24

Chuyển đổi địa chỉ



25

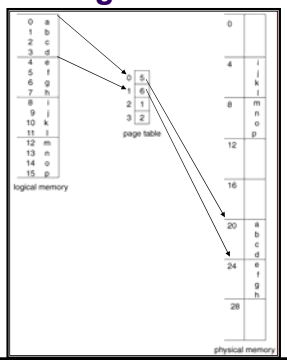
Ví dụ phân trang 1



26

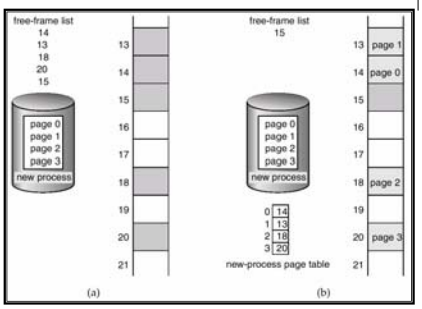
Ví dụ phân trang 2

Cỡ của một trang là 4 bytes



27

Bảng frame rỗi



Trước cấp phát Sau cấp phát

28

Cài đặt bảng trang

- Bảng trang được lưu ở bộ nhớ trong
- *Thanh ghi cơ sở bảng trang* (*page-table base register*) (PTBR) trỏ đến bảng trang
- *Thanh ghi độ dài bảng trang* (*page-table length register*) (PTLR) lưu cỡ bảng trang
- Sử dụng bảng trang, mọi thao tác truy cập dữ liệu/lệnh cần tới 2 lần truy cập bộ nhớ (1 cho bảng trang, 1 cho dữ liệu/lệnh)

29

Cài đặt bảng trang (tiếp)

- Truy cập bộ nhớ hai lần: Giảm tốc độ
- Giải quyết vấn đề 2 lần truy cập bộ nhớ: Sử dụng phần cứng cache có tốc độ truy cập cao gọi là bộ nhớ kết hợp (*associative memory*) hoặc vùng đệm hỗ trợ chuyển đổi (*translation look-aside buffers -TLB*)
- Mỗi phần tử trong TLB có hai phần: khóa và giá trị
- Số lượng các phần tử của TLB thường từ 64 đến 1024

30

Bộ nhớ kết hợp

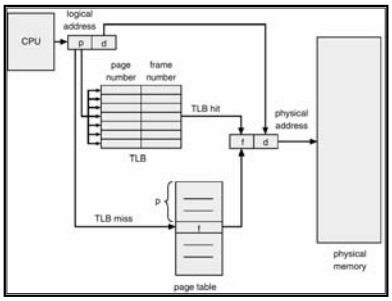
- Bộ nhớ kết hợp

Page #	Frame #

- Chuyển đổi địa chỉ (A´, A´´)
  if A´ nằm trong thanh ghi kết hợp, lấy *frame#*.
  else lấy *frame#* từ bảng trang trong bộ nhớ

31

Phân trang phần cứng với TLB



32

Thời gian truy cập hiệu quả

- Thời gian tìm kiếm ở thanh ghi kết hợp = ε (đơn vị thời gian)
- Thời gian truy cập bộ nhớ là *n* đơn vị thời gian
- Hit ratio: Số phần trăm (%) địa chỉ trang được tìm thấy ở các thanh ghi kết hợp/TLB
- Hit ratio = α
- Thời gian truy cập hiệu quả (EAT):
  EAT = $(n + \varepsilon)\,\alpha + (2n + \varepsilon)(1 - \alpha) = 2n + \varepsilon - \alpha n$

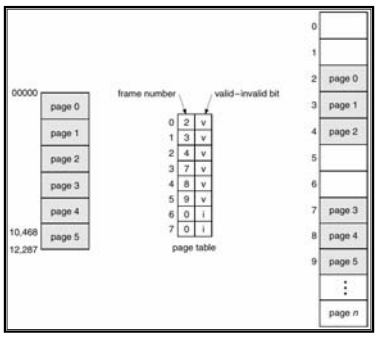
33

Bảo vệ bộ nhớ

- Bộ nhớ được bảo vệ nhờ kết hợp bit bảo vệ trong mỗi phần tử ở bảng trang
- Bit hợp lệ-không hợp lệ (*valid-invalid*) kết nối với mỗi phần tử trong bảng trang:
  - "valid" chỉ ra rằng trang thuộc không gian địa chỉ logic của tiến trình → trang hợp lệ
  - "invalid" chỉ ra rằng trang không thuộc không gian địa chỉ logic của tiến trình

34

Ví dụ bit valid (v)/invalid (i) trong bảng trang



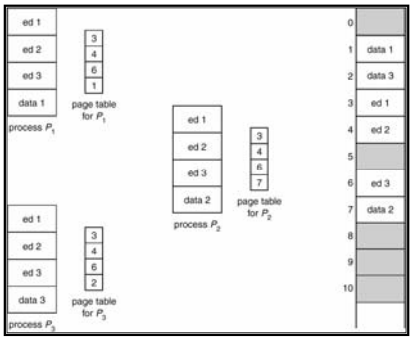
35

Các trang chung

- Mã dùng chung
  - Nhiều tiến trình (soạn thảo, compiler...) có thể dùng chung các đoạn mã reentrant (đoạn mã không tự thay đổi chính nó)
  - Đoạn mã chung phải xuất hiện ở cùng một vị trí địa chỉ trong không gian địa chỉ logic/ảo của tất cả các tiến trình
- Mã lệnh và dữ liệu riêng
  - Mỗi tiến trình có một bản riêng chứa lệnh và dữ liệu
  - Các trang chứa lệnh và dữ liệu riêng có thể ở bất kỳ vị trí nào trong không gian địa chỉ của tiến trình

36

Ví dụ các trang chung



37

Cấu trúc bảng trang

Bảng trang phân cấp

Bảng trang băm

Bảng trang ngược

38

Bảng trang phân cấp

- Bộ nhớ máy tính lớn (2^{32}-2^{64} bytes): Nếu dùng bảng trang một cấp thì bảng trang có cỡ rất lớn: Tốn bộ nhớ, tìm kiếm chậm
- Không gian địa chỉ logic được quản lý bởi nhiều bảng trang ở nhiều cấp
- Một kỹ thuật đơn giản nhất là bảng trang hai cấp. Có thể có bảng trang hai, ba, bốn cấp

39

Ví dụ bảng trang hai cấp

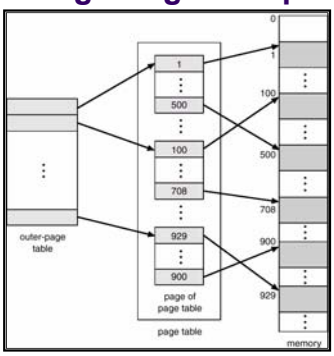
- Địa chỉ logic (trên máy 32-bit, trang cỡ 4K=2^{12}) được chia thành:
  - Địa chỉ trang: 20 bits.
  - Địa chỉ offset: 12 bits.
- Bảng trang 2 cấp (địa chỉ 20 bit) được chia thành:
  - 10-bit địa chỉ trang cấp 1
  - 10-bit địa chỉ trang cấp 2
- Khi đó địa chỉ logic có dạng:

Địa chỉ trang		Offset
p_1	p_2	d
10	10	12

trong đó p_1 là chỉ số đến bảng trang ngoài, p_2 là chỉ số đến trang (thực sự) ở bảng trang ngoài

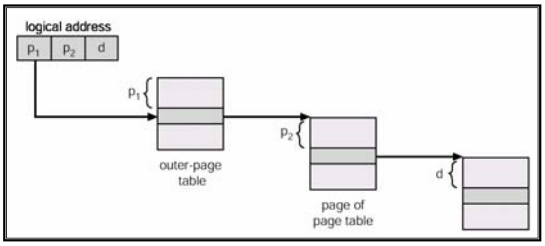
40

Sơ đồ bảng trang hai cấp



41

Tính địa chỉ với bảng trang hai cấp



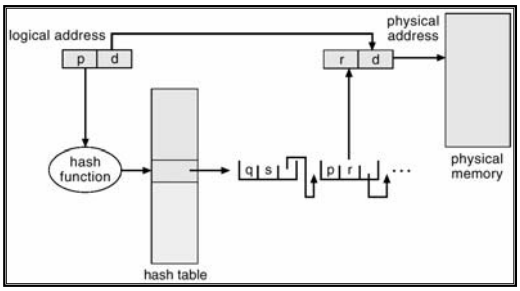
42

Bảng trang băm

- Thường sử dụng khi địa chỉ > 32 bit
- Số hiệu/địa chỉ trang được băm trong bảng trang. Bảng trang này chứa dãy các phần tử (các trang) được băm ở cùng một vị trí
- Số hiệu trang được so sánh trong dãy các trang được băm ở cùng một vị trí để từ đó tìm ra frame vật lý

43

Bảng trang băm



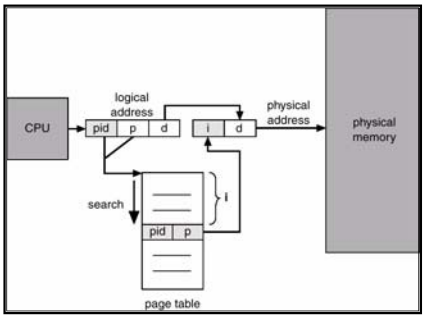
44

Bảng trang ngược

- Giải pháp giảm bộ nhớ lưu các bảng trang
- Mỗi phần tử trong bảng ứng với một frame
- Mỗi phần tử chứa địa chỉ ảo của trang và thông tin về tiến trình đang sử dụng trang đó
- Giảm dung lượng bộ nhớ cần để lưu các bảng trang, nhưng tăng thời gian cần để tìm trong bảng khi cần tham chiếu đến một trang
- Sử dụng bảng băm để hạn chế số lần tìm kiếm trong các phần tử bảng trang

45

Kiến trúc bảng trang ngược



46

Phân đoạn (Segmentation)

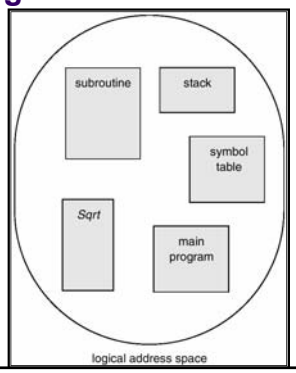
47

Phân đoạn

- Phương thức quản lý bộ nhớ cho phép NSD "nhìn" bộ nhớ một cách dễ dàng dưới góc độ lập trình
- Một chương trình gồm nhiều phân đoạn, mỗi phân đoạn thể hiện dưới góc độ lập trình ở dạng:
  - main program, // Chương trình chính
  - function, // Các hàm
  - method, // Các phương thức
  - object, // Các đối tượng, lớp
  - local/global variables, // Các biến
  - common block, // Các khối chung
  - stack, // Ngăn xếp
  - symbol table, arrays // Bảng ký hiệu, mảng

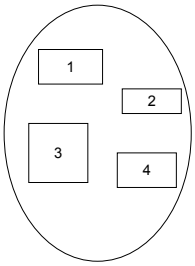
48

Chương trình nhìn từ NSD

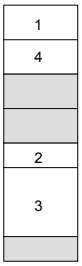


49

Phân đoạn: Cách nhìn logic



Không gian địa chỉ của NSD



Không gian bộ nhớ vật lý

50

Kiến trúc phân đoạn

- Địa chỉ ảo/logic là một bộ đôi:<segment, offset>
- Bảng phân đoạn (*segment table*) – ánh xạ địa chỉ vật lý 2 cấp; mỗi phần tử bảng có:
  - *base*: Địa chỉ vật lý bắt đầu của phân đoạn (segment)
  - *limit*: Độ dài của phân đoạn (segment).

51

Kiến trúc phân đoạn (tiếp)

- Thanh ghi cơ sở bảng phân đoạn (*Segment-table base register STBR*) trỏ đến base
- Thanh ghi độ dài bảng phân đoạn (*Segment-table length register - STLR*) chỉ ra số lượng phân đoạn được sử dụng trong tiến trình;
- Số hiệu phân đoạn *s* là hợp lệ nếu thỏa mãn điều kiện: *s* < STLR.

52

Kiến trúc phân đoạn (tiếp)

- Định vị lại (relocation)
  - Động
  - Sử dụng bảng phân đoạn
- Dùng chung (sharing)
  - Có các phân đoạn dùng chung
  - Sử dụng cùng một số hiệu phân đoạn (segment number)
- Cấp phát (allocation)
  - first fit/best fit
  - Phân mảnh ngoài

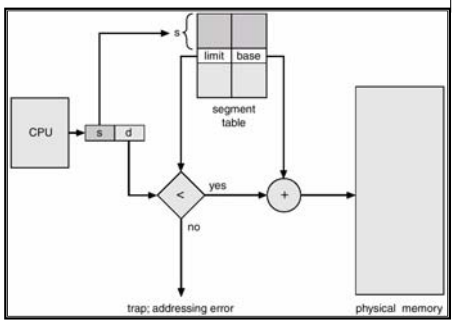
53

Kiến trúc phân đoạn (tiếp)

- Bảo vệ bộ nhớ:Mỗi phân đoạn có:
  - Bit kiểm tra = 0 ⇒ phân đoạn không hợp lệ
  - read/write/execute privileges
- Protection bits associated with segments; code sharing occurs at segment level.
- Do phân đoạn có cỡ biến đổi → Gặp vấn đề tương tự trong cấp phát bộ nhớ liên tục
- Kết hợp phân đoạn với phân trang để tăng hiệu quả sử dụng bộ nhớ, dễ cấp phát hơn (ví dụ: MULTICS, Intel 386)

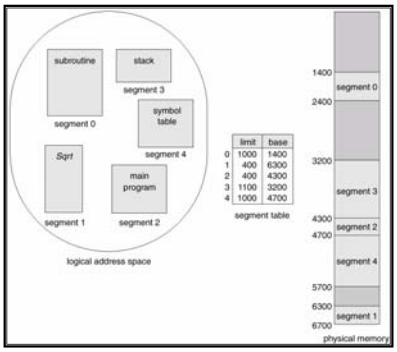
54

Phần cứng phân đoạn



55

Ví dụ phân đoạn



56

Tóm tắt

- Địa chỉ logic (ảo)/Địa chỉ vật lý (thật)
- Các phương án ánh xạ địa chỉ của chương trình vào bộ nhớ
- Cấp phát bộ nhớ liên tục, phân mảnh, các chiến lược cấp phát first-fit, best-fit, worst-fit
- Phân trang
  - Trang, frame
  - Bảng trang, bảng trang phân cấp, bảng trang ngược
- Phân đoạn, bảng phân đoạn

57

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Bộ nhớ ảo (Virtual Memory)

Yêu cầu phân trang
Tạo tiến trình
Thay thế trang
Cấp phát frame
Thrashing

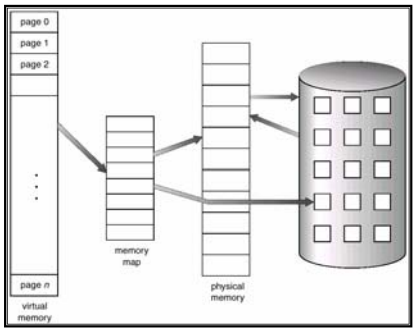
2

Virtual memory (Bộ nhớ ảo)

- **Bộ nhớ ảo** – tách biệt bộ nhớ logic và vật lý.
  - Cho phép tiến trình có cỡ lớn hơn bộ nhớ trong có thể thực hiện được
  - Không gian địa chỉ ảo có thể lớn hơn nhiều so với không gian địa chỉ vật lý (về dung lượng)
  - Cho phép các tiến trình sử dụng chung không gian địa chỉ
  - Cho phép tạo tiến trình hiệu quả hơn
- Bộ nhớ ảo có thể được cài đặt thông qua:
  - Yêu cầu phân trang (demand paging)
  - Yêu cầu phân đoạn (demand segmentation)

3

Minh họa bộ nhớ ảo có dung lượng lớn hơn bộ nhớ vật lý



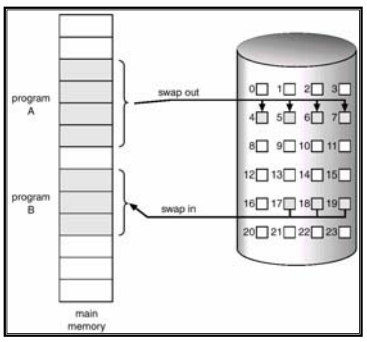
4

Yêu cầu trang (demand paging)

- Chỉ đưa một trang vào bộ nhớ khi cần thiết
  - Giảm thao tác các vào ra
  - Tiết kiệm bộ nhớ
  - Đáp ứng nhanh
  - Tăng được số người sử dụng (tiến trình)
- Khi cần một trang ⇒ tham chiếu đến nó
  - Tham chiếu lỗi ⇒ Hủy bỏ
  - Không nằm trong bộ nhớ ⇒ Đưa trang vào bộ nhớ

5

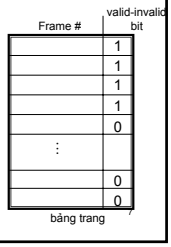
Chuyển một trang (trong bộ nhớ) ra vùng đĩa liên tục



6

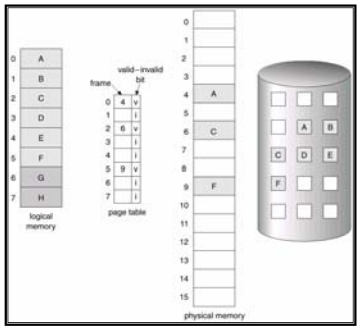
Valid-Invalid Bit

- Mỗi phần tử bảng trang có một bit hợp lệ/không hợp lệ (1: trong bộ nhớ, 0: không trong bộ nhớ)
- Khởi đầu: valid–invalid bằng 0.
- Ví dụ bảng trang+bit invalid/valid:
- Khi tính địa chỉ, nếu valid–invalid ở bảng trang là 0: ⇒ lỗi trang (page-fault trap)



7

Bảng trang với một số trang không nằm trong bộ nhớ



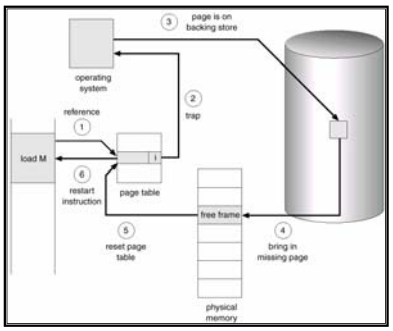
8

Xử lý page-fault (lỗi trang)

- HĐH sẽ kiểm tra nguyên nhân lỗi:
  - Lỗi từ tiến trình: kết thúc tiến trình, hoặc
  - Trang không nằm trong bộ nhớ: Thực hiện tiếp:
- Tìm một frame rỗi và đưa trang vào bộ nhớ
- Sửa lại bảng trang (bit = valid)
- Thực hiện lại lệnh tham chiếu trang
- Vấn đề hiệu năng: Nếu tại một thời điểm có yêu cầu nhiều trang (ví dụ: Một trang cho lệnh và vài trang cho dữ liệu)?

9

Các bước xử lý page-fault



10

Nếu không có frame rỗi?

- Thực hiện thay thế trang – swap out một số trang đang ở trong bộ nhớ nhưng hiện tại không được sử dụng
  - Thuật toán nào tốt?
  - Hiệu năng: Cần một thuật toán có ít page-fault nhất để hạn chế vào/ra
- Nhiều trang có thể được đưa vào bộ nhớ tại cùng một thời điểm.
- Thuật toán thay thế trang: FIFO, Optimal, LRU, LRU-approximation

11

Hiệu năng của yêu cầu trang

- Tỷ lệ page-fault là p: $0 \le p \le 1.0$
  - nếu $p = 0$: Không có page-fault
  - nếu $p = 1$, mọi yêu cầu truy cập đến trang đều gây ra page-fault
- Effective Access Time (EAT)

  EAT = (1 – p) x memory access
  + p (page fault overhead
  + [swap page out]
  + swap page in
  + restart overhead)

12

Tạo tiến trình

- Bộ nhớ ảo có ưu điểm khi khởi tạo một tiến trình mới:

  - Copy-on-Write (Chỉ tạo copy của trang khi có thay đổi)

  - Memory-Mapped Files (Các file ánh xạ bộ nhớ)

13

Copy-on-Write

- Copy-on-Write (COW) cho phép tiến trình cha và con dùng chung trang trong bộ nhớ khi mới khởi tạo tiến trình con
- Chỉ khi nào một trong hai tiến trình sửa đổi trang dùng chung, thì trang đó mới được copy một bản mới
- COW làm cho việc tạo tiến trình hiệu quả hơn: Chỉ các trang bị sửa đổi mới được copy
- Các trang rỗi được cấp phát từ một tập hợp (pool) các trang được xóa trắng với số 0

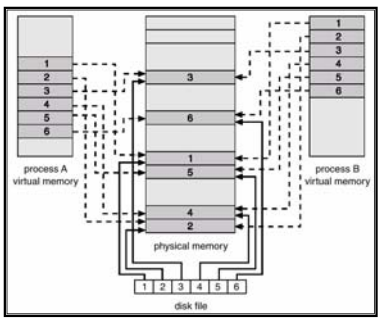
14

Các file ánh xạ bộ nhớ

- Các file được xem như một phần bộ nhớ trong bằng các ánh xạ một khối đĩa vào một trang trong bộ nhớ
- Khởi đầu các file được đọc khi có yêu cầu trang: Một phần của file (cỡ=cỡ trang) được đọc vào bộ nhớ
- Các thao tác đọc/ghi trên file sau đó được xem như đọc/ghi trong bộ nhớ
- Đơn giản hóa việc truy cập file thông qua bộ nhớ hơn là sử dụng các hàm hệ thống **read()** và **write()**.
- Cho phép các tiến trình có thể ánh xạ chung một file, do đó cho phép các trang dùng chung trong bộ nhớ

15

Ví dụ file ánh xạ bộ nhớ



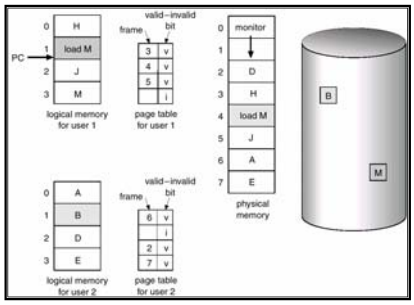
16

Thay thế trang

- Thay thế trang dùng để tránh thực hiện nhiều lần cấp phát mỗi khi có page-fault
- Sử dụng *modify bit* để giảm chi phí (overhead) vào/ra với các trang: Chỉ các trang có thay đổi mới được ghi ra đĩa
- Thay thế trang là một trong các yếu tố xóa đi sự khác biệt của bộ nhớ ảo và thật: Tiến trình lớn hơn dung lượng bộ nhớ trong có thể thực hiện được

17

Ví dụ: Yêu cầu thay thế trang



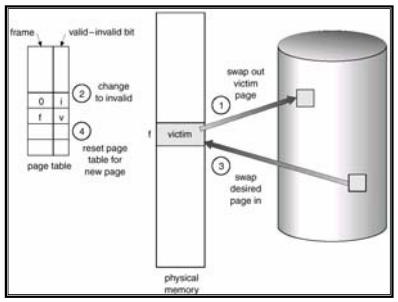
18

Cơ sở thay thế trang

1. Tìm vị trí của trang *p* cần thay trên đĩa
2. Tìm một frame rỗi *f*:
   1. Nếu có frame rỗi: Sử dụng frame đó
   2. Nếu không có frame rỗi: Sử dụng thuật toán thay thế trang để đưa một trang trong bộ nhớ ra để sử dụng frame ứng với trang đó
3. Đọc trang *p* vào frame *f* vừa tìm được và cập nhật bảng trang, bảng frame
4. Lặp lại quá trình này

19

Thay thế trang



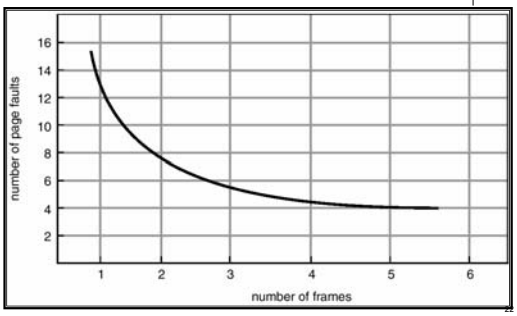
20

Thuật toán thay thế trang

- Cần tỷ lệ page-fault thấp nhất
- Đánh giá thuật toán: Thực hiện trên một danh sách các yêu cầu truy cập bộ nhớ và tính số lượng các page-fault
- Trong tất cả các ví dụ, ta sử dụng danh sách yêu cầu truy cập bộ nhớ:
  1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.

21

Đồ thị page-fault



22

Thuật toán FIFO

- Yêu cầu truy cập: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- Bộ nhớ VL có 3 frame: 9 page-faults
- Bộ nhớ VL có 4 frame: 10 page-faults

9 page faults

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

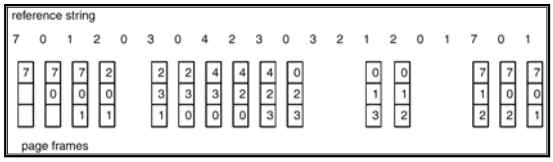
- Thay thế FIFO – Belady's anomaly
  - Có nhiều frame ⇒ ít page-fault

10 page faults

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

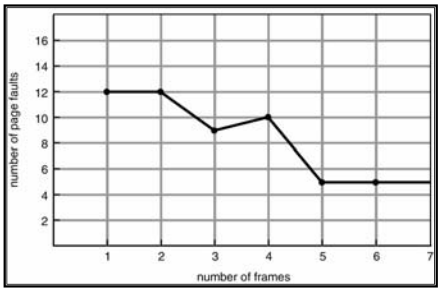
23

Thay thế trang FIFO



24

FIFO Illustrating Belady's Anamoly



25

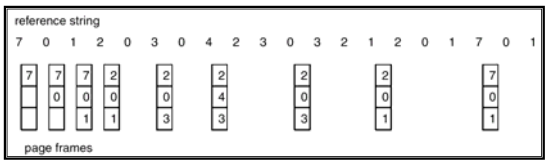
Thuật toán tối ưu

- Thay thế các trang *sẽ không được sử dụng* trong khoảng thời gian dài nhất
- Danh sách yêu cầu truy cập: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5; có 4 frame

1	4
2	6 page-fault
3	
4	5

26

Thay thế trang tối ưu



27

Thuật toán LRU (Least Recently Used)

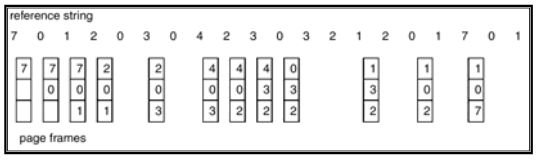
- Thay thế trang *không được sử dụng lâu nhất*
- Danh sách yêu cầu truy cập: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

1	5	
2		
3	5	4
4	3	

- Cài đặt sử dụng biến đếm
  - Mỗi trang có một biến đếm; mỗi khi trang được truy cập, gán giá trị đồng hồ thời gian cho biến đếm.
  - Khi một cần phải thay thế trang, căn cứ vào giá trị các biến đếm của trang: Cần tìm kiếm trong danh sách các trang

28

Thay thế trang LRU



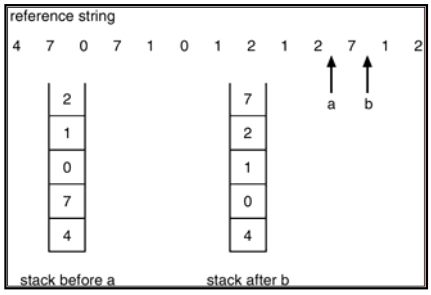
29

Thuật toán LRU (tiếp)

- Cài đặt sử dụng ngăn xếp: Sử dụng một ngăn xếp (stack) lưu các số hiệu trang ở dạng danh sách móc nối kép:
  - Khi trang được tham chiếu đến:
    - Chuyển trang lên đỉnh ngăn xếp
    - Cần phải thay đổi 6 con trỏ
  - Khi thay thế trang không cần tìm kiếm

30

Sử dụng ngăn xếp để ghi lại trang vừa mới được sử dụng



31

Thuật toán LRU xấp xỉ

- Thuật toán bit tham chiếu
- Sử dụng bit tham chiếu đánh dấu các trang đã được sử dụng/chưa được sử dụng
  - Mỗi trang có 1 bit được khởi tạo bằng 0
  - Khi trang được tham chiếu đến, đặt bit bằng 1
  - Không biết thứ tự sử dụng các trang

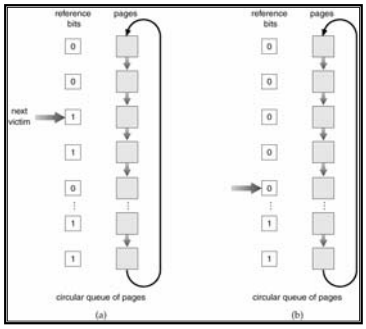
32

Thuật toán LRU xấp xỉ (tiếp)

- Thuật toán "Second-chance"
  - Là một thuật toán kiểu FIFO
  - Cần sử dụng bit tham chiếu
  - Thay thế theo thứ tự thời gian
  - Nếu trang cần được thay thế (theo thứ tự thời gian) có bit tham chiếu là 1 thì:
    - Đặt bit tham chiếu bằng 0, để trang đó trong bộ nhớ, chưa thay thế ngay (*second chance*)
    - Thay thế trang tiếp theo (theo thứ tự thời gian) tuân theo qui tắc tương tự
    - Có thể cài đặt bằng buffer vòng

33

Thuật toán second-chance



34

Các thuật toán đếm

- Sử dụng biến đếm để đếm số lần tham chiếu đến trang
- Thuật toán LFU (Least Frequently Used): Thay thế các trang có giá trị biến đếm nhỏ nhất
- Thuật toán MFU (Most Frequently Used): Ngược lại với LFU, dựa trên cơ sở: Các trang ít được sử dụng nhất (giá trị biến đếm nhỏ nhất) là các trang vừa được đưa vào bộ nhớ trong

35

Cấp phát các frame

- Mỗi tiến trình cần một số lượng tối thiểu các trang để thực hiện được
- Ví dụ: IBM 370 cần 6 để thực hiện lệnh SS MOVE:
  - Lệnh dài 6 bytes, có thể chiếm 2 trang.
  - 2 trang để thao tác **from**.
  - 2 trang để thao tác **to**.
- Hai cách cấp phát:
  - Cấp phát cố định
  - Cấp phát ưu tiên

36

Cấp phát cố định

- Cấp phát bình đẳng: nếu cấp phát 100 frame cho 5 tiến trình, mỗi tiến trình có 20 frame.
- Cấp phát tỷ lệ: Dựa theo cỡ tiến trình.
- Cấp phát tỷ lệ:

s_i = size of process p_i
$S = \sum s_i$
m = total number of frames
a_i = allocation for $p_i = \frac{s_i}{S} \times m$

$m = 64$
$s_i = 10$
$s_2 = 127$
$a_1 = \frac{10}{137} \times 64 \approx 5$
$a_2 = \frac{127}{137} \times 64 \approx 59$

37

Cấp phát ưu tiên

- Sử dụng cấp phát tỷ lệ căn cứ vào độ ưu tiên của tiến trình, không căn cứ vào cỡ tiến trình
- Nếu tiến trình P_i sinh ra page-fault:
  - Thay thế một trong các frame của tiến trình đó
  - Thay thế một trong các frame của tiến trình khác có độ ưu tiên thấp hơn

38

Cấp phát tổng thể và cục bộ

- Thay thế tổng thể – Các trang/frame có thể được chọn để thay thế từ tập hợp tất cả các frame/trang. Tiến trình này có thể dùng lại frame/trang của tiến trình khác
- Thay thế cục bộ – Mỗi tiến trình chỉ thay thế trong các trang/frame của chính nó đã được cấp phát

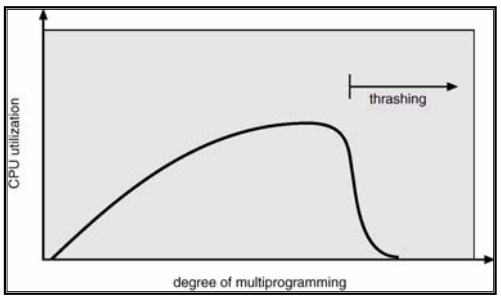
39

Thrashing

- Nếu tiến trình không có đủ trang, tỷ lệ page-fault rất cao, điều đó dẫn tới:
  - Khả năng tận dụng CPU thấp, do đó
  - HĐH có thể tăng mức độ đa chương trình →
  - Các tiến trình được tiếp tục đưa vào hệ thống
- → Hiện tượng thrashing
- Tiến trình thrashing: Một tiến trình luôn bận để swap-in và swap-out

40

Minh họa thrashing



41

Cách hạn chế/ngăn chặn thrashing

- Sử dụng thuật toán thay thế trang cục bộ hoặc thay thế trang ưu tiên để hạn chế thrashing: Chưa giải quyết tốt
- Sử dụng mô hình cục bộ: mô hình working-set
  - Khi tiến trình thực hiện, nó chuyển từ điều kiện cục bộ này sang điều kiện cục bộ khác
  - Điều kiện cục bộ được xác định dựa trên cấu trúc của chương trình và cấu trúc dữ liệu

42

Mô hình working-set

- Sử dụng tham số Δ là tổng số lần tham chiếu trang gần nhất
- Tập các trang sử dụng trong Δ lần gần nhất là working-set
- WSS_i là cỡ working-set của tiến trình P_i: Rõ ràng là WSS_i phụ thuộc vào độ lớn của Δ
  - Nếu Δ quá nhỏ: Không phản ánh đúng tính cục bộ
  - Nếu Δ quá lớn: vượt qua tính cục bộ
  - Nếu $\Delta = \infty$: WSS_i tập toàn bộ các trang trong quá trình thực hiện tiến trình

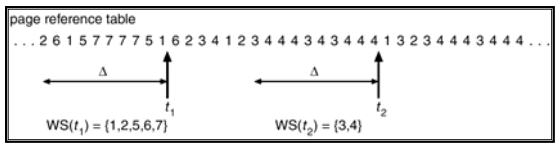
43

Mô hình working-set

- $D = \Sigma\ WSS_i \equiv$ Tổng số các frame được yêu cầu
- Gọi m là tổng số fram rỗi: nếu $D > m$ thì trong hệ thống sẽ xuất hiện thrashing
- Chính sách cấp phát: nếu $D > m$ thì tạm dừng thực hiện một số tiến trình

44

Mô hình working-set



45

Các tệp ánh xạ bộ nhớ

- Sinh viên tự tìm hiểu trong giáo trình từ trang 348 đến trang 353

46

Các vấn đề cần nhớ

- Bộ nhớ ảo
- Yêu cầu trang
- Thay thế trang
- Các thuật toán thay thế trang: FIFO, tối ưu, LRU, LRU xấp xỉ, LFU, MFU, thuật toán đếm
- Thrashing: Định nghĩa, nguyên nhân, cách khắc phục và phòng tránh
- Mô hình working-set

47

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ thông tin
Trường Đại học Công nghệ

1

Giao diện hệ thống tệp

Khái niệm tệp
Các phương pháp truy cập
Cấu trúc thư mục
Nối hệ thống tệp
Dùng chung tệp
Bảo vệ

2

Khái niệm tệp

- Không gian địa chỉ logic liên tục
- Các kiểu:
  - Dữ liệu
    - Số
    - Ký tự
    - Nhị phân
  - Chương trình

3

Cấu trúc tệp

- Chuỗi các từ, byte
- Cấu trúc bản ghi đơn giản: gồm các dòng, độ dài cố định, độ dài thay đổi
- Cấu trúc phức tạp: tài liệu có khuôn dạng, các tệp nạp có định vị lại
- Yếu tố quyết định cấu trúc:
  - Hệ điều hành
  - Chương trình

4

Thuộc tính tệp

- **Name** – Thông tin người đọc được về tệp
- **Type** – cần cho chương trình, hệ điều hành
- **Location** – Vị trí tệp trên các thiết bị lưu trữ
- **Size** – Cỡ hiện tại của tệp
- **Protection** – Điều khiển các quyền truy cập
- **Time**, **date**, **and user ID** – Dữ liệu về thời gian và định danh người sử dụng
- Thông tin về cách lưu trữ tệp trên thiết bị, được lưu trong cấu trúc của thư mục

5

Các toán tử trên tệp

- **Create**: Tạo tệp mới
- **Write**: Ghi vào tệp
- **Read**: Đọc từ tệp
- **Seek** – Định vị lại con trỏ tệp
- **Delete**: Xóa tệp
- **Truncate**: Xóa dữ liệu hiện có trong tệp
- **Open**(F_i) – mở tệp F_i (tìm phần tử F_i trong thư mục và đưa nội dung của F_i vào bộ nhớ)
- **Close**(F_i) – đóng tệp F_i (đưa nội dung của F_i trong bộ nhớ ra đĩa)

6

Mở tệp

- Một số thông tin cần quản lý khi mở tệp:
  - Con trỏ tệp (file pointer): Con trỏ đến vị trí đọc/ghi cuối cùng của mỗi tiến trình
  - Đếm số lượng mở tệp (file-open count): Biến đếm số lần tệp được mở, để cho phép xóa dữ liệu từ bảng mở tệp khi tiến trình cuối cùng đóng tệp
  - Vị trí trên đĩa của tệp: thông tin truy cập dữ liệu của tệp lưu trên đĩa
  - Quyền truy cập (access rights): Thông tin về các quyền truy cập tệp của mỗi tiến trình

7

Mở tệp có khóa

- Một số HĐH có toán tử này
- Dùng để điều khiển truy cập đồng thời đến tệp
- Có hai cách khóa:
  - **Mandatory** – Khóa mang tính chất toàn cục
  - **Advisory** – Khóa mang tính chất hợp tác giữa các tiến trình

8

Kiểu tệp, tên và phần mở rộng

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

9

Các phương pháp truy cập

- **Truy cập tuần tự**

  read next
  write next
  reset
  no read after last write
  (rewrite)

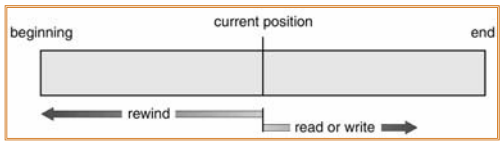
- **Truy cập trực tiếp**

  read *n*
  write *n*
  position to *n*
  read next
  write next
  rewrite *n*

n = relative block number

10

Tệp truy cập tuần tự



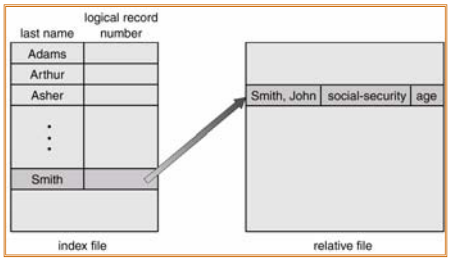
11

Mô phỏng truy cập tuần tự và trực tiếp

sequential access	implementation for direct access
reset	*cp = 0;*
read next	*read cp;* *cp = cp+1;*
write next	*write cp;* *cp = cp+1;*

12

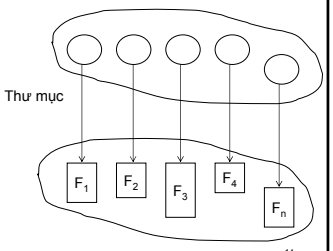
Ví dụ về tệp chỉ số và Relative Files



13

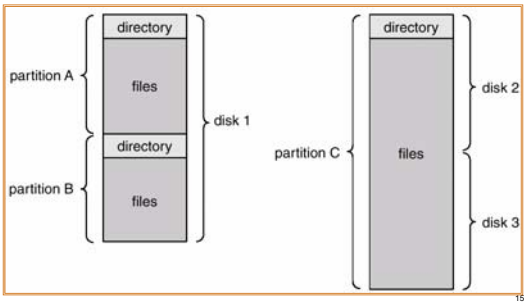
Cấu trúc thư mục

- Thư mục là một tập các node chứa thông tin về tất cả các tệp
- Thư mục và tệp đều nằm trên đĩa
- Backup của thư mục và đĩa có thể nằm trên băng từ



14

Ví dụ một hệ thống tệp



15

Thông tin trên thư mục thiết bị

- Tên
- Kiểu
- Địa chỉ
- Độ dài hiện tại
- Độ dài lớn nhất
- Thời gian của lần truy cập cuối (để lưu trữ)
- Thời gian của lần cập nhật cuối (for dump)
- ID của người chủ tệp
- Các thông tin bảo vệ

16

Các toán tử trên thư mục

- Tìm một tệp
- Tạo một tệp
- Xóa một tệp
- Liệt kê nội dung thư mục
- Đổi tên một tệp
- Duyệt toàn bộ hệ thống tệp

17

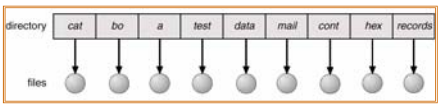
Cần tổ chức thư mục để đạt được:

- Tính hiệu quả: tìm thấy một tệp nhanh chóng
- Tên tệp mang lại sự tiện lợi cho người dùng
  - Hai NSD có thể đặt cùng tên cho hai tệp khác nhau
  - Một tệp có thể có nhiều tên khác nhau
- Nhóm tệp: Các tệp có thể được nhóm lại dựa trên thuộc tính (ví dụ nhóm các tệp chương trình nguồn Java, nhóm các tệp thực hiện được...)

18

Thư mục một mức

- Một thư mục cho tất cả NSD

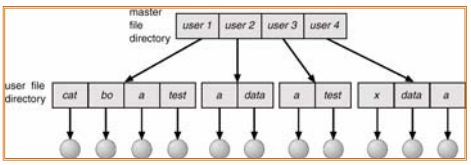


- Vấn đề đặt tên
- Vấn đề nhóm các tệp với nhau

19

Thư mục hai mức

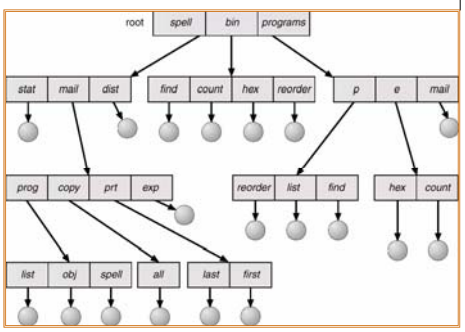
- Mỗi NSD có các thư mục riêng



- Đường dẫn
- Hai NSD có thể đặt cùng tên cho hai tệp khác nhau
- Tìm kiếm tệp hiệu quả
- Không có khả năng nhóm các tệp

20

Thư mục cấu trúc cây



21

Thư mục cấu trúc cây (tiếp)

- Tìm kiếm hiệu quả
- Có khả năng nhóm các tệp
- Thư mục làm việc hiện hành
  - Đổi thư mục làm việc hiện hành

22

Thư mục cấu trúc cây (tiếp)

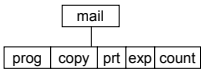
- Đường dẫn tuyệt đối và tương đối
- Tạo một tệp trong thư mục hiện hành
- Xóa tệp

  rm <tên tệp>
- Tạo thư mục con trong thư mục hiện hành

  mkdir <tên thư mục>

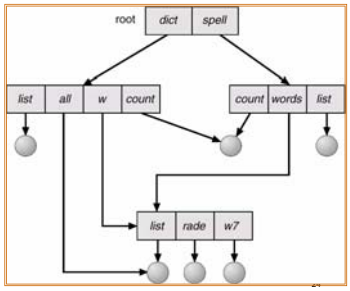
Ví dụ: Nếu thư mục hiện hành là /mail

mkdir count



Thư mục với cấu trúc đồ thị phi chu trình

- Các thư mục có thể có chung thư mục con và tệp

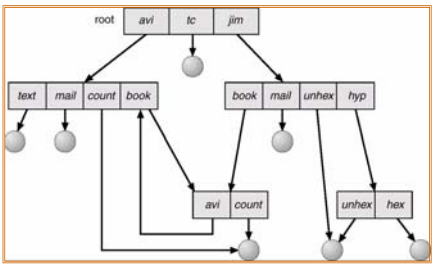


Thư mục với cấu trúc đồ thị phi chu trình (tiếp)

- Tệp hoặc thư mục có thể có các tên khác nhau
  - Shortcut trong Windows
  - Link trong Unix/Linux
- Mềm dẻo hơn cấu trúc cây nhưng phức tạp hơn:
  - Khi xóa một tệp hoặc thư mục có nhiều tên
    - Cần sử dụng con trỏ ngược
    - Sử dụng biến đếm số tên

25

Thư mục đồ thị tổng quát



26

Thư mục đồ thị tổng quát (tiếp)

- Làm cách nào để đảm bảo không có chu trình?
  - Chỉ có phép link tới tệp, không cho link đến thư mục
  - "Dọn dẹp" hệ thống tệp (garbage collection)
  - Mỗi khi có link mới, thực hiện thuật toán phát hiện chu trình

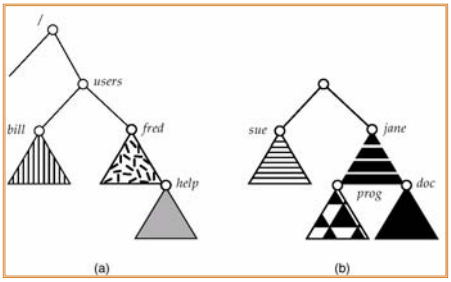
27

Kết nối (mount) hệ thống tệp

- Một hệ thống tệp phải được mount trước khi có thể truy cập tới (sử dụng)
- Một tệp được mount tại điểm kết nối (mount point)

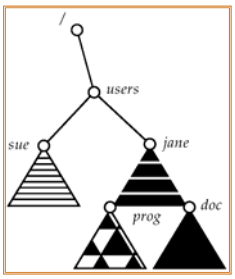
28

Minh họa mount/unmount



29

Điểm kết nối (mount point)



30

Tệp dùng chung

- Cần có tệp dùng chung trên các hệ đa người dung (multi-user)
- Dùng chung tệp có thể thực hiện thông qua một phương pháp bảo vệ
- Với các hệ phân tán, NSD có thể dùng chung tệp trên mạng
- Network File System (NFS) là một phương pháp dùng chung tệp phổ biến

31

Đa người dùng

- **User IDs** định danh NSD để từ đó xác định các quyền và phương pháp bảo vệ
- **Group IDs** xác định nhóm NSD để từ đó xác định các quyền truy cập nhóm

32

Hệ thống tệp từ xa

- Sử dụng tệp để truy cập các hệ thống file ở các vị trí khác nhau
  - Thủ công: ví dụ FTP
  - Tự động: hệ thống tệp phân tán **distributed file systems**
  - Bán tự động: **world wide web**
- Mô hình khách-chủ **(Client-server model)** cho phép máy khách mount hệ thống tệp của máy chủ từ xa
  - Máy chủ có thể phục vụ nhiều máy khách
  - Định danh máy khách và NSD trên máy khách có thể đơn giản (không an toàn - insecure) hoặc rất phức tạp

33

Hệ thống tệp từ xa (tiếp)

- **NFS** là giao thức sử dụng chung tệp chuẩn trên UNIX cho mô hình client-server
- **CIFS** là chuẩn trên Windows
- Các hàm hệ thống chuẩn được chuyển đổi thành lời gọi từ xa (remote call)
- Các dịch vụ đặt tên phân tán (**distributed naming services**) như LDAP, DNS, NIS cho ta cách truy cập thống nhất đến các thông tin cần thiết cho tính toán từ xa

34

Lỗi trong hệ thống tệp từ xa

- Có nhiều nguyên nhân gây lỗi trong hệ thống tệp từ xa: Do lỗi mạng, lỗi server...
- Khôi phục lỗi cần có thông tin trạng thái đối với mỗi yêu cầu phục vụ từ xa
- Các giao thức như NFS lưu đưa toàn bộ các thông tin trạng thái vào mỗi yêu cầu do đó dễ khôi phục, nhưng kém an ninh

35

Nhất quán về ngữ nghĩa

- **Nhất quán ngữ nghĩa** chỉ định cách truy cập đồng thời của nhiều NSD đến tệp dùng chung
  - Andrew File System (AFS) cài đặt hệ thống ngữ nghĩa phức tạp cho hệ thống tệp truy cập từ xa
    - AFS có ngữ nghĩa theo phiên: các toán tử **write** chỉ có tác dụng sau khi tệp được **close**
  - Unix file system (UFS) cài đặt:
    - Các toán tử **write** ngay lập tức có tác dụng trên các tệp chung (người đọc nhìn thấy kết quả của **write**)
    - Dùng chung tệp cho phép nhiều NSD đọc và ghi đồng thời

36

Bảo vệ

- Người tạo tệp (chủ tệp) được phép qui định
  - Các toán tử nào trên tệp có thể được thực hiện...
  - ... và do ai thực hiện
- Các toán tử:
  - Read
  - Write
  - Execute
  - Append
  - Delete
  - List

37

Danh sách và nhóm truy cập

- Các toán tử: read, write, execute
- Ba lớp NSD là **owner**, **group** và **public**

		RWX
a) **owner access** 7	⇒	1 1 1
		RWX
b) **group access** 6	⇒	1 1 0
		RWX
c) **public access** 1	⇒	0 0 1

- Xem thêm quyền truy cập tệp của HĐH Unix/Linux

38

Các vấn đề cần nhớ

- Khái niệm tệp
- Các phương pháp truy cập tệp
- Cấu trúc thư mục một cấp, nhiều cấp, cấu trúc thư mục cây, đồ thị phi chu trình, đồ thị tổng quát
- Nối hệ thống tệp
- Dùng chung tệp
- Hệ thống tệp từ xa
- Quyền truy cập tệp

39

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ Thông tin
Trường Đại học Công nghệ

1

Hệ vào/ra

Phần cứng
Giao diện vào/ra với ứng dụng
Hệ vào/ra của nhân
Chuyển yêu cầu vào/ra thành thao tác phần cứng
Streams
Các vấn đề về hiệu năng

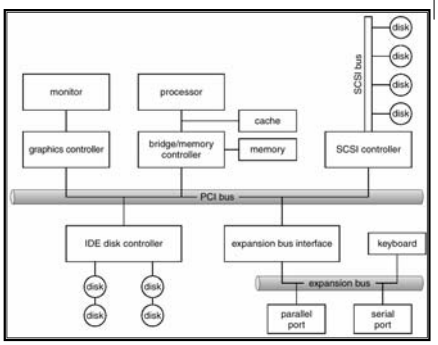
2

Phần cứng vào/ra

- Có rất nhiều loại thiết bị vào/ra
- Các khái niệm chung
  - Port (cổng vào/ra)
  - Bus
  - Controller
- Các vi lệnh điều khiển thiết bị vào/ra
- Thiết bị vào/ra có địa chỉ được sử dụng bởi:
  - Các lệnh vào/ra trực tiếp
  - Vào/ra thông qua ánh xạ bộ nhớ

3

Cấu trúc bus của máy PC



4

Một số địa chỉ vào/ra của PC

I/O address range (hexadecimal)	device
000-00F	DMA controller
020-021	interrupt controller
040-043	timer
200-20F	game controller
2F8-2FF	serial port (secondary)
320-32F	hard-disk controller
378-37F	parallel port
3D0-3DF	graphics controller
3F0-3F7	diskette-drive controller
3F8-3FF	serial port (primary)

5

Polling

- Xác định trạng thái của thiết bị:
  - command-ready (lệnh sẵn sàng?)
  - busy (bận?)
  - Error (lỗi?)
- Thực hiện vòng lặp *chờ bận* để chờ vào/ra với thiết bị

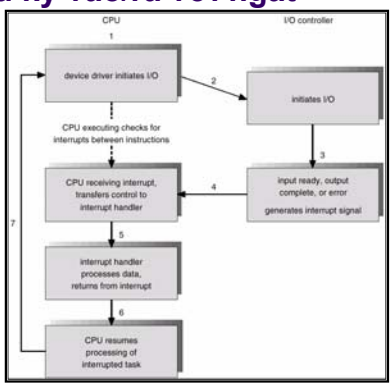
6

Interrupts (ngắt)

- Thiết bị vào/ra kích hoạt đường yêu cầu ngắt CPU
- Bộ thao tác ngắt nhận ngắt
- CPU có thể bỏ qua hoặc làm trễ việc xử lý một số ngắt
- Vector ngắt giúp CPU tìm được hàm xử lý ngắt
  - Dựa trên độ ưu tiến
  - Một số ngắt là không che được (unmaskable)
- Cơ chế ngắt có thể dùng cho exceptions

7

Chu kỳ vào/ra với ngắt



8

Bảng vector ngắt của BXL Intel

vector number	description
0	divide error
1	debug exception
2	null interrupt
3	breakpoint
4	INTO-detected overflow
5	bound range exception
6	invalid opcode
7	device not available
8	double fault
9	coprocessor segment overrun (reserved)
10	invalid task state segment
11	segment not present
12	stack fault
13	general protection
14	page fault
15	(Intel reserved, do not use)
16	floating-point error
17	alignment check
18	machine check
19Đ31	(Intel reserved, do not use)
32Đ255	maskable interrupts

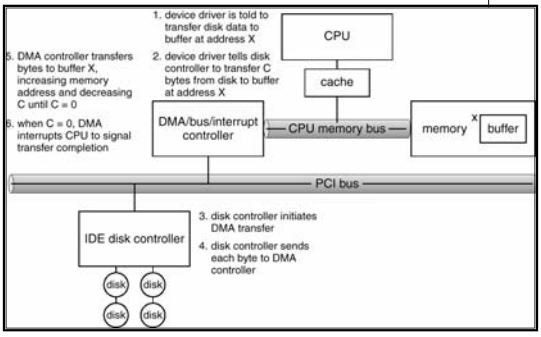
9

Truy cập bộ nhớ trực tiếp

- Thuật ngữ: Direct memory access (DMA)
- Được sử dụng để tránh lập trình vào/ra với dung lượng dữ liệu lớn
- Phần cứng cần có: Bộ điều khiển DMA
- CPU truyền dữ liệu trực tiếp giữa bộ nhớ và thiết bị vào/ra

10

Quá trình 6 bước thực hiện vào/ra theo DMA

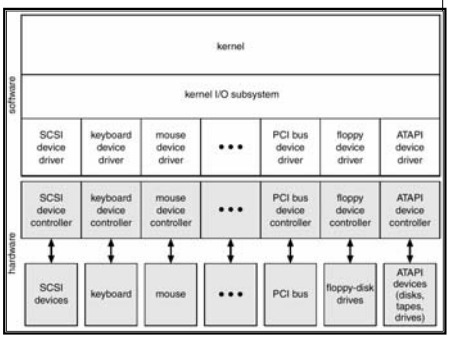


Giao diện vào/ra với ứng dụng

- Các hàm hệ thống vào/ra của một thiết bị được đóng gói trong các class chung
- Tầng điều khiển thiết bị (device-driver layer) che đi sự khác biệt giữa các bộ điều khiển vào/ra
- Có nhiều loại thiết bị căn cứ theo các tiêu chí:
  - Character-stream / block
  - Sequential / random-access
  - Sharable / dedicated
  - Speed of operation
  - read-write / read only / write only

12

Cấu trúc vào ra của nhân



13

Đặc tính các thiết bị vào/ra

aspect	variation	example
data-transfer mode	character block	terminal disk
access method	sequential random	modem CD-ROM
transfer schedule	synchronous asynchronous	tape keyboard
sharing	dedicated sharable	tape keyboard
device speed	latency seek time transfer rate delay between operations	
I/O direction	read only write only readÐwrite	CD-ROM graphics controller disk

14

Các thiết bị *block* và *character*

- Các thiết bị *block* (ví dụ đĩa cứng):
  - Các lệnh làm việc: read, write, seek
  - Có thể thực hiện vào/ra theo chế độ *raw* I/O hoặc thông qua truy cập hệ thống tệp
  - Có thể truy cập qua tệp memory-mapped (ánh xạ bộ nhớ)
- Các thiết bị *character* (ví dụ bàn phím, chuột, cổng COM):
  - Các lệnh làm việc: get, put
  - Bổ sung thư viện cho phép làm việc theo dòng (line)

15

Các thiết bị mạng

- Có thể là thiết bị block hoặc character
- Unix và Windows NT/9*i*/2000 có giao diện lập trình socket
  - Tách biệt giao thức mạng với các thao tác mạng
  - Có tính năng `select`
- Nhiều cách tiếp cận vào/ra (pipes, FIFOs, streams, queues, mailboxes)

16

Đồng hồ (clock) và timer

- Cung cấp thông tin về giờ hiện tại, giờ đã trôi qua, timer
- Nếu phần cứng clock/timer lập trình được:Có thể tạo ngắt định kỳ (Cần cho các hệ time-sharing)

17

Vào/ra blocking và nonblocking

- Blocking – Tiến trình treo đến khi vào/ra hoàn thành
  - Dễ hiểu, dễ sử dụng
  - Không đủ đối với một số loại yêu cầu vào/ra
- Nonblocking – Hàm vào/ra trả lại kết quả ngay không cần vào/ra hoàn thành
  - Giao diện NSD, copy dữ liệu có buffered vào/ra
  - Được cài đặt qua kỹ thuật đa luồng
  - Trả lại ngay số byte được đọc/ghi

18

Vào/ra không đồng bộ

- Asynchronous (không đồng bộ): Tiến trình chạy trong khi vào/ra đang được thực hiện
  - Khó sử dụng
  - Hệ vào/ra gửi tín hiệu cho tiến trình khi vào/ra hoàn thành

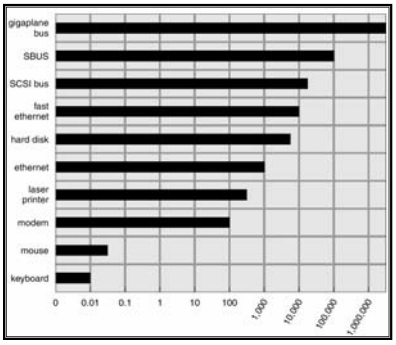
19

Hệ vào/ra của nhân

- Lập lịch
  - Một số yêu cầu vào/ra được phục vụ thông qua hàng chờ vào/ra của từng thiết bị
  - Một số HĐH cố gắng đảm bảo tính công bằng
- Tạo vùng đệm lưu dữ liệu trong bộ nhớ khi truyền dữ liệu giữa các thiết bị:
  - Khắc phục sự khác nhau về tốc độ của các thiết bị
  - Khắc phục sự khác nhau về độ dài gói dữ liệu
  - Để duy trì ngữ nghĩa copy

20

Tốc độ truyền dữ liệu của các thiết bị trên Sun Enterprise 6000



21

Hệ vào/ra của nhân

- Caching – Bộ nhớ tốc độ cao chứa các bản copy của dữ liệu
  - Dữ liệu luôn là bản copy
  - Cải thiện đáng kể hiệu năng hệ thống
- Spooling –Lưu dữ liệu ra (output) cho một thiết bị
  - Sử dụng khi thiết bị chỉ phục vụ được một yêu cầu tại một thời điểm
  - Ví dụ: Máy in

22

Hệ vào/ra của nhân

- Cung cấp khả năng sử dụng "độc quyền" một thiết bị
  - Hàm hệ thống: cấp phát và giải phóng thiết bị
  - Cơ chế chống bế tắc

23

Xử lý lỗi

- HĐH có thể khôi phục lỗi gây ra do đọc đĩa, thiết bị chưa sẵn sàng, ghi lỗi...
- Khi có lỗi vào/ra: Hàm điều khiển trả lại mã lỗi
- Hệ thống có log ghi lại các lỗi vào/ra

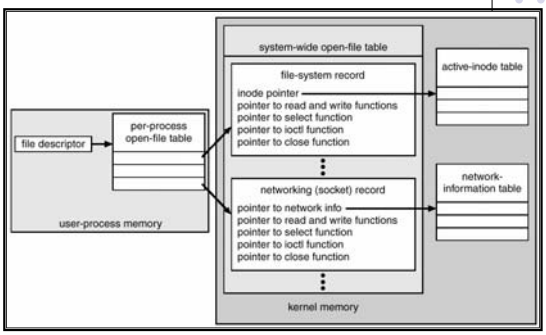
24

Cấu trúc dữ liệu của nhân

- Nhân giữ các thông tin trạng thái cho các thành phần của hệ vào/ra, bao gồm bảng các mở tệp, kết nối mạng, trạng thái các thiết bị *character*
- Nhiều cấu trúc dữ liệu phức tạp để lưu vết các vùng đệm, cấp phát bộ nhớ, các khối nhớ rỗi...
- Một số HĐH sử dụng phương pháp hướng đối tượng và message-passing để cài đặt hệ vào/ra

25

Cấu trúc vào/ra trong nhân UNIX

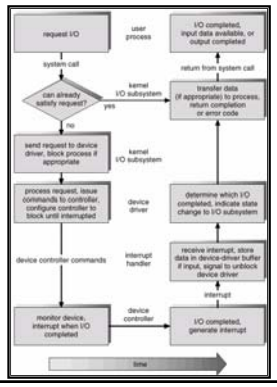


Chuyển đổi yêu cầu vào/ra thành thao tác phần cứng

- Giả sử một tiến trình đọc tệp từ đĩa cứng. Các bước thực hiện như sau:
  - Xác định thiết bị chứa tệp
  - Biến đổi tên tệp thành dạng biểu diễn của tệp trên thiết bị
  - Đọc dữ liệu (vật lý) từ đĩa vào vùng đệm
  - Cho phép tiến trình được đọc dữ liệu từ vùng đệm
  - Trả lại điều khiển cho tiến trình

27

Thực hiện một yêu cầu vào/ra



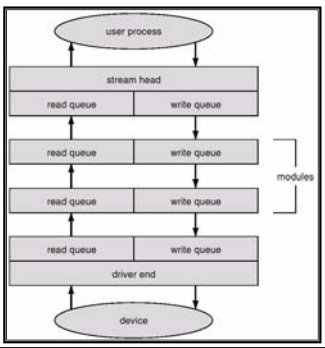
28

STREAMS

- **STREAM** – kênh liên lạc full-duplex giữa một tiến trình của NSD và một thiết bị
- Một STREAM gồm có:
  - **STREAM head** dùng để giao tiếp với tiến trình của NSD
  - **driver end** giao tiếp với thiết bị
  - *n* STREAM module giữa head và end (n≥0).
- Mỗi module có một **read queue** và một **write queue**
- Message passing được sử dụng để truyền thông giữa các queue

29

Cấu trúc STREAMS



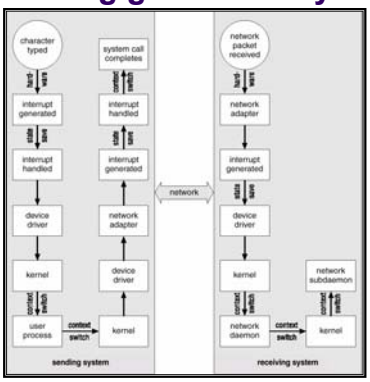
30

Hiệu năng

- Vào/ra là yếu tố chính ảnh hưởng đến hiệu năng của hệ thống:
  - Yêu cầu CPU thực hiện mã vào/ra của driver, mã vào/ra của kernel
  - Thực hiện Context switch khi có ngắt
  - Copy dữ liệu
  - Truyền dữ liệu mạng

31

Truyền thông giữa các máy tính



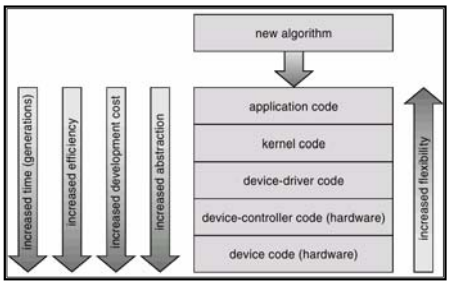
32

Cải tiến hiệu năng

- Giảm số lượng các context switches
- Giảm copy dữ liệu
- Giảm số lần ngắt bằng cách truyền lượng lớn dữ liệu trong mỗi lần vào/ra, dùng controller thông minh...
- Sử dụng DMA
- Cân bằng tải giữa hiệu năng CPU, bộ nhớ, bus, và vào/ra để đạt thông lượng tốt nhất

33

Chức năng các thiết bị



34

Các vấn đề cần nhớ

- Giao diện vào/ra với ứng dụng
- Hệ vào/ra của nhân HĐH
- Streams
- Các vấn đề về hiệu năng hệ thống chịu ảnh hưởng của vào/ra

35

Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ Thông tin
Trường Đại học Công nghệ

Các hệ thống lưu trữ

Cấu trúc đĩa
Lập lịch đĩa
Quản lý đĩa
Quản lý không gian swap
Cấu trúc RAID
...

Cấu trúc đĩa

- Các ổ đĩa được đánh địa chỉ như một mảng lớn, 1 chiều với mỗi phần tử là một khối logic – đơn vị truyền nhận nhỏ nhất
- Mảng một chiều nói trên được ánh xạ vào các sector đĩa một cách tuần tự
  - Sector 0 là sector đầu tiên trên rãnh đầu tiên của track nằm ngoài cùng trên đĩa
  - Quá trình ánh xạ theo thứ tự chỉ số: sector, track, cylinder (từ ngoài vào trong)

Lập lịch đĩa (1)

- HĐH cần sử dụng phần cứng một cách hiệu quả - với đĩa: thời gian truy cập nhanh và băng thông lớn
- Thời gian truy cập bị ảnh hưởng bởi:
  - *Seek time* (thời gian dịch đầu đọc): Thời gian chuyển đầu đọc đến cylinder chứa sector cần truy cập
  - *Rotational latency* (Độ trễ quay): Thời gian chờ đĩa quay để đầu đọc gặp sector cần truy cập

Lập lịch đĩa (2)

- Seek time: Càng nhỏ càng tốt
  - Tương đương: Khoảng cách dịch đầu đọc càng nhỏ càng tốt
- Băng thông đĩa là tổng số byte đã được truyền chia cho tổng thời gian giữa yêu cầu đầu tiên và thời gian hoàn thành lần truyền dữ liệu cuối cùng

Lập lịch đĩa (3)

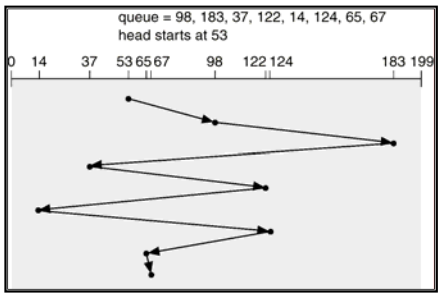
- Có nhiều thuật toán lập lịch đĩa
- Chúng ta minh họa với dãy các yêu cầu (Giả sử đĩa có 200 track từ 0-199):

98, 183, 37, 122, 14, 124, 65, 67

Đầu đọc đang nằm ở cylinder 53

FCFS (First come first serve)

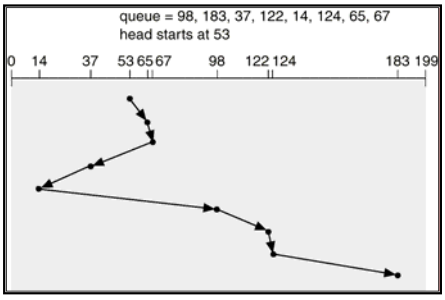
Tổng số bước di chuyển của đầu đọc là: 640 cylinder.



SSTF: Shortest seek time first

- Yêu cầu có seek time nhỏ nhất tính từ vị trí hiện tại của đầu đọc
- Lập lịch SSTF là một dạng của lập lịch SJF có thể gây ra một số yêu cầu không bao giờ được phục vụ (starvation)
- Ví dụ minh họa: Tổng số bước di chuyển của đầu đọc là 236 cylinder.

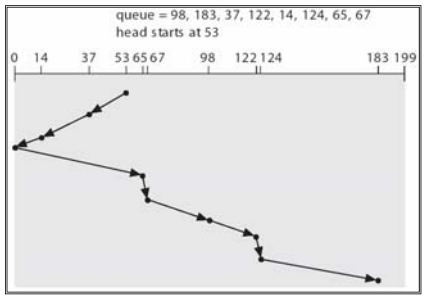
Ví dụ SSTF



SCAN

- Đầu đọc của đĩa di chuyển từ một phía (ví dụ bên ngoài hoặc bên trong đĩa) sang phía kia để phục vụ các yêu cầu đọc, sau đó di chuyển ngược lại... quá trình này lặp đi lặp lại
- Phương thức hoạt động tương tự thang máy nên thuật toán này còn được gọi là thuật toán thang máy (*elevator algorithm*)
- Ví dụ minh họa: Đầu đọc phải dịch chuyển 208 cylinder.

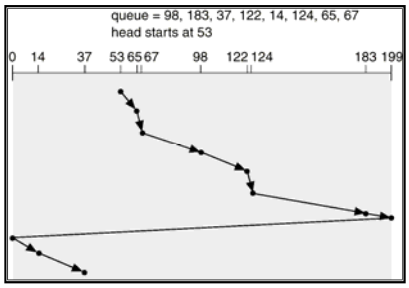
Ví dụ SCAN



C-SCAN

- Đầu đọc chuyển từ một phía (trong/ngoài) sang phía kia và phục vụ các yêu cầu. Khi sang đến phía kia, đầu đọc quay trở lại nhưng trong khi quay trở lại không phục vụ yêu cầu nào.
- C-SCAN xem các cylinders như một danh sách vòng

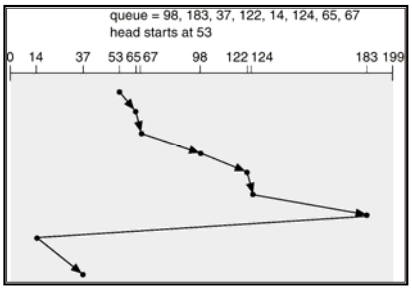
Ví dụ C-SCAN



C-LOOK

- Là một trường hợp của C-SCAN
- C-SCAN: Đầu đọc chuyển giữa cylinder 0 và *n* (cynlinder cuối)
- C-LOOK: Đầu đọc chuyển giữa c_{min} và c_{max} trong đó c_{min} là cynlinder có số thứ tự nhỏ nhất trong số các yêu cầu; c_{max} là cynlinder có số thứ tự nhỏ nhất trong số các yêu cầu
- C-LOOK giảm quãng đường di chuyển của đầu đọc so với C-SCAN

Ví dụ C-LOOK



Chọn thuật toán lập lịch đĩa

- SSTF là phổ biến và "tự nhiên"
- SCAN và C-SCAN thực hiện tốt với các hệ thống đọc ghi đĩa nhiều
- Hiệu năng nói chung phụ thuộc vào số lượng và tính chất của các yêu cầu truy cập đĩa
- Yêu cầu đọc ghi đia có thể bị ảnh hưởng bởi phương pháp cấp phát tệp

Chọn thuật toán lập lịch đĩa

- Các thuật toán lập lịch đĩa nên được cài đặt như một module độc lập của HĐH để dễ thay thế khi cần thiết
- SSTF hoặc LOOK có thể chọn là thuật toán ngầm định

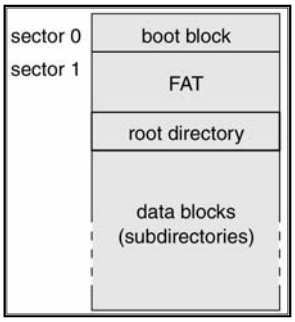
Quản lý đĩa

- *Low-level format* hoặc *physical format* — Chia đĩa thành các sector để bộ điều khiển đĩa (disk controller) có thể đọc/ghi
- Để lưu tệp lên đĩa, HĐH cần ghi cấu trúc dữ liệu lên đĩa:
  - HĐH chia đĩa thành các *partition* (phân vùng) – mỗi partition là một nhóm các cylinder
  - HĐH thực hiện *logical formatting* hay tạo hệ thống tệp.

Tổ chức đĩa

- Đĩa cứng có boot block để khởi tạo hệ thống:
  - bootstrap được lưu trong ROM
  - *Bootstrap loader* là một chương trình nhỏ nằm trên boot block của đĩa cứng.
- Ví dụ bootstrap loader:
  - Linux: GRUB, LILO
  - Windows: NTLDR
- Với các sector hỏng: Phương pháp *sector sparing*

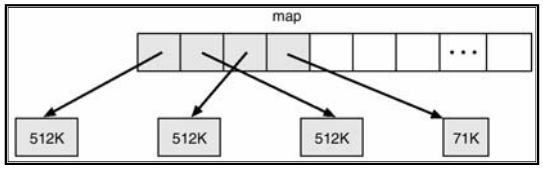
Tổ chức đĩa của MS-DOS



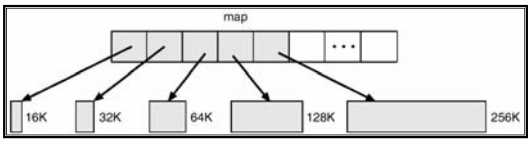
Quản lý không gian swap

- Không gian swap được xem như một phần mở rộng của bộ nhớ trong và nằm trên đĩa
- Không gian swap có thể nằm trên hệ thống tệp hoặc trên một partition riêng
- Quản lý không gian swap
  - UNIX BSD 4.3 cấp phát swap khi tiến trình bắt đầu thực hiện và lưu các segment: text và data
  - Nhân dùng *swap maps* để quản lý việc sử dụng không gian swap.

4.3 BSD Text-Segment Swap Map



4.3 BSD Data-Segment Swap Map



Cấu trúc RAID

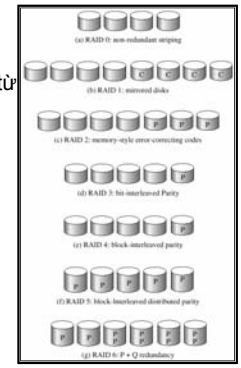
- **RAID** – Hệ thống lưu trữ sử dụng nhiều ổ đĩa để tăng độ tin cậy (**reliability)** thông qua sự dư thừa (**redundancy**).
- Có 6 mức RAID.

RAID (tiếp)

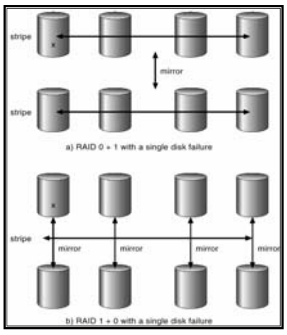
- Disk striping: Sử dụng một tập đĩa như một đĩa
- RAID cải thiện hiệu năng và độ tin cậy của hệ lưu trữ bằng cách lưu trữ có dư thừa
  - *Mirroring/shadowing*: Mỗi đĩa có một bản copy (duplicate).
  - *Block interleaved parity*: Mức độ dư thừa ít hơn mirroring.

Các mức RAID

- Xem chi tiết các mức RAID trong giáo trình từ trang 471 đến 475



RAID (0 + 1) và (1 + 0)

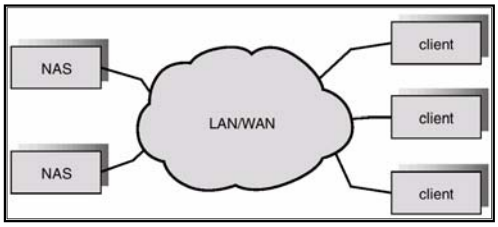


Kết nối đĩa

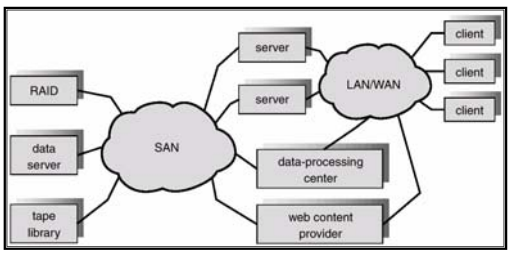
- Có thể kết nối đĩa theo 2 cách:

1. **Host attached** thông qua cổng vào/ra
2. **Network attached** thông qua kết nối mạng

Network-Attached Storage



Storage-Area Network



Cài đặt hệ lưu trữ ổn định

- Write-ahead log scheme requires stable storage.
- Để cài đặt hệ lưu trữ ổn định:
  - Replicate information on more than one nonvolatile storage media with independent failure modes.
  - Update information in a controlled manner to ensure that we can recover the stable data after any failure during data transfer or recovery.

Các thiết bị tertiary

- Thuật ngữ: Tertiary storage
- Đặc tính của tertiary storage là giá rẻ
- Nói chung, tertiary storage được làm để *tháo được*
- Ví dụ: Đĩa mềm, CD-ROM, USB

Đĩa tháo được

- Đĩa mềm: Đĩa phủ từ nằm trong vỏ bảo vệ
  - Hầu hết các đĩa mềm chứa được khoảng 1 MB; một số đĩa sử dụng công nghệ tương tự có thể chứa 1GB.
  - Tốc độ đĩa mềm nhanh nhưng dễ mất dữ liệu do hỏng bề mặt (tiếp xúc)

Đĩa tháo được

- Đĩa quang-từ ghi dữ liệu trên đĩa nhựa cứng bề mặt phủ vật liệu từ.
  - Nhiệt laser được sử dụng để khuếch đại từ trường yếu để ghi 1 bit
  - Ánh sáng laser light is được sử dụng để đọc dữ liệu (hiệu ứng Kerr).
  - Đầu đọc đĩa loại này xa bề mặt đĩa hơn là đầu đọc đĩa từ → giảm hỏng do xước, va chạm.
- Đĩa quang không sử dụng vật liệu từ mà dùng vật liệu đặc biệt có thể bị biến đổi do tia laser

Đĩa WORM

- WORM (“Write Once, Read Many Times”) Ghi một lần, đọc nhiều lần
- Đĩa có 3 lớp: Hai lớp nhựa ở 2 mặt, ở giữa là một lớp phim mỏng chế tạo từ nhôm
- Để ghi 1 bit: Ổ đĩa dùng tia laser đốt một lỗ nhỏ trên lớp phim nhôm.
- Rất bền và tin cậy
- Ví dụ: CD-ROM, DVD-ROM

Băng

- Băng rẻ hơn đĩa, nhưng truy cập ngẫu nhiên chậm hơn
- Băng thường được dùng cho các ứng dụng không yêu cầu truy cập nhanh. Ví dụ: Lưu trữ dữ liệu, backup
- Các hệ thống băng từ lớn sử dụng robot để thay băng: Chuyển băng giữa các ổ băng và thư viện băng
  - stacker – Thư viện băng nhỏ (một vài băng)
  - silo – Thư viện băng lớn (vài nghìn băng)

Các vấn đề của HĐH

- Nhiệm vụ chính của HĐH là quản lý các thiết bị vật lý và cung cấp một máy ảo cho ứng dụng (thông qua trừu tượng hóa)
- Với đĩa cứng có hai mức trừu tượng hóa:
  - Thiết bị – Một mảng các khối dữ liệu
  - Hệ thống tệp – HĐH phục vụ các yêu cầu truy cập đĩa (qua cơ chế hàng chờ/lập lịch) từ nhiều ứng dụng

Tốc độ truy cập đĩa

- Hai yếu tố ảnh hưởng đến tốc độ truy cập đĩa: băng thông bandwidth và độ trễ (latency).
- Băng thông được đo bằng byte/second.
  - Sustained bandwidth: Tốc độ trao đổi dữ liệu trung bình trong một lần đọc/ghi lớn (tổng số byte/thời gian)
  - Effective bandwidth – Băng thông trung bình của toàn bộ các lần vào/ra bao gồm **seek** / **locate**...

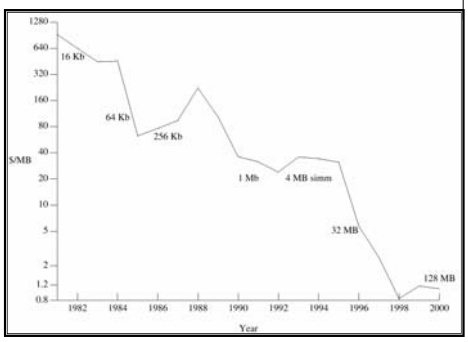
Tốc độ truy cập đĩa

- Độ trễ truy cập: Thời gian cần để định vị dữ liệu trên đĩa.
  - Độ trễ cho đĩa: Chuyển đầu đọc đến cylinder cần thiết + độ trễ quay (thường < 35ms)
  - Độ trễ băng: Cần tua băng → Độ trễ từ vài chục đến vài trăm giây

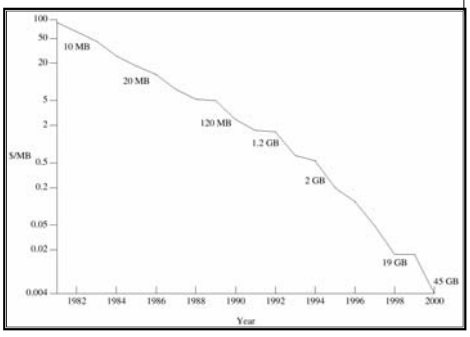
Độ tin cậy

- Đĩa cứng có độ tin cậy cao hơn băng hoặc đĩa tháo được
- Lưu trữ trên đĩa quang tin cậy hơn đĩa từ hoặc băng
- Đầu đọc đĩa cứng hỏng→ mất dữ liệu
- Đầu đọc băng, CD... hỏng không gây mất dữ liệu

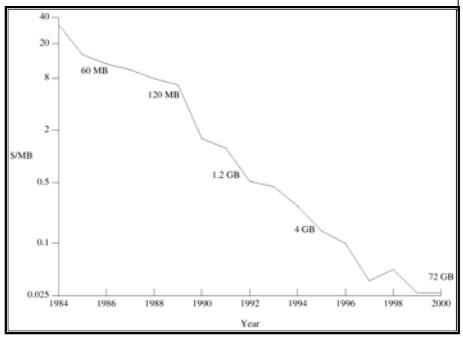
Giá 1MB DRAM từ 1981 đến 2000



Giá 1MB đĩa cứng từ 1981 đến 2000



Giá 1MB băng, từ 1984 đến 2000



Nguyên lý hệ điều hành

Nguyễn Hải Châu
Khoa Công nghệ Thông tin
Trường Đại học Công nghệ

Bảo vệ và an ninh

Bảo vệ

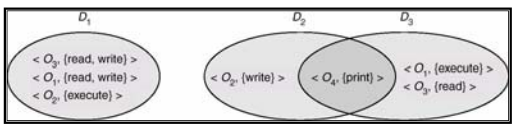
Mục đích bảo vệ
Các miền bảo vệ
Ma trận truy cập
Cài đặt ma trận truy cập
Hủy bỏ quyền truy cập

Bảo vệ

- HĐH gồm một tập các đối tượng, đối tượng: phần cứng hoặc phần mềm
- Mỗi đối tượng có một tên duy nhất và có thể truy cập đến thông qua một số toán tử (hàm hệ thống)
- Bảo vệ: Đảm bảo mỗi đối tượng được truy cập đúng cách và chỉ bởi các tiến trình được phép

Cấu trúc miền bảo vệ

- Quyền truy cập = <*tên đối tượng*, *tập các toán tử*> trong đó *tập các toán tử* là một tập con của tập tất cả các toán tử hợp lệ có thể thực hiện trên *đối tượng*
- Miền = Tập các quyền truy cập

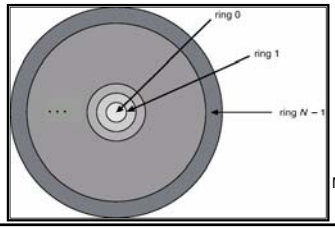


Cài đặt miền trên UNIX

- Hệ Unix có 2 miền:
  - Người sử dụng (user)
  - Người quản trị hệ thống (supervisor/root)
- UNIX
  - Miền = user-id
  - "Chuyển" miền bảo vệ thông qua hệ thống tệp:
    - Mỗi tệp có 1 bit gắn với miền (setuid bit).
    - Khi tệp *f* được thực hiện và bit setuid=1 thì user-id được đặt là owner của tệp *f*. Khi thực hiện xong, user-id được trả lại giá trị cũ

Cài đặt tên miền trên Multics

- Các miền bảo vệ bao nhau (ring)
- Gọi D_i và D_j là hai miền bảo vệ bất kỳ.
- Nếu $j < i \Rightarrow D_i \subseteq D_j$



Multics Rings

Ma trận truy cập

- Biểu diễn các miền bảo vệ dưới dạng ma trận (Ma trận truy cập - *access matrix*). Giả sử ma trận là *access*
- Các hàng biểu diễn các miền
- Các tên cột biểu diễn các đối tượng
- Phần tử *access(i, j)* là tập các toán tử một tiến trình thực hiện trong miền D_i được thao tác trên đối tượng O_j

Ma trận truy cập

domain \ object	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Sử dụng ma trận truy cập

- Nếu một tiến trình trong miền D_i muốn thực hiện toán tử "op" trên đối tượng O_j, thì "op" phải nằm trong ma trận truy cập
- Mở rộng: Bảo vệ "động"
  - Các toán tử để thêm, xóa các quyền truy cập
  - Các quyền truy cập đặc biệt:
    - *Chủ của đối tượng O_i*
    - *Sao chép toán tử "op" từ O_i sang O_j*
    - *Quyền điều khiển – D_i có thể sửa đổi quyền truy cập của D_j*
    - *transfer – switch từ miền D_i sang D_j*

Ma trận truy cập với các miền được xem như các đối tượng

domain \ object	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Ma trận truy cập với quyền truy cập *Copy*

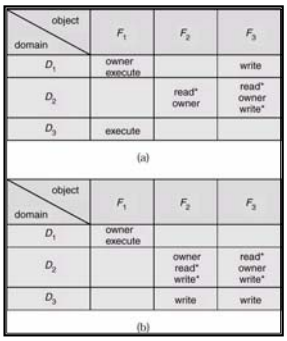
domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domain \ object	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Ma trận truy cập với quyền truy cập *Owner*

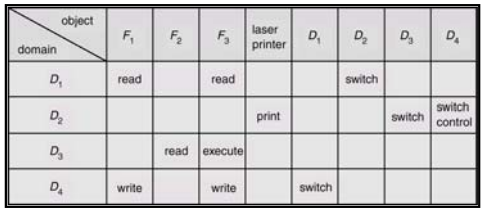


object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write*
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		
D_2		owner read* write*	read* owner write*
D_3		write	write

(b)



object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Cài đặt ma trận truy cập

- Có 4 cách cài đặt:
  - Bảng toàn cục (global table): Phương pháp đơn giản nhất
  - Danh sách truy cập (access list) cho các đối tượng
  - Danh sách khả năng (capability list) cho các miền
  - Cơ chế khóa – chìa (Lock-Key)

Bảng toàn cục

- Là một bảng với các phần tử là bộ ba:
  - <miền, đối tượng, tập các quyền>
- Bảng toàn cục thường lớn nên không nằm toàn bộ trong bộ nhớ trong → Cần nhiều thao tác vào/ra
- Tốn thời gian tìm kiếm trên bảng toàn cục

Danh sách truy cập

- Mỗi cột trong ma trận truy cập có thể được cài đặt thành một danh sách truy cập cho một đối tượng
- Danh sách gồm các phần tử là bộ đôi <miền, tập các quyền>
- Các hệ điều hành: UNIX, Windows sử dụng danh sách truy cập

Danh sách khả năng

- Mỗi hàng trong ma trận truy cập được cài đặt thành một danh sách khả năng cho một miền
- Một danh sách khả năng là một danh sách đối tượng kèm theo các quyền

Cơ chế khóa – chìa

- Là sự kết hợp giữa danh sách truy cập và danh sách khả năng
- Đối tượng có danh sách các mẫu bit gọi là *khóa*
- Mỗi miền có danh sách các mẫu bit gọi là *chìa*
- Một tiến trình thực hiện trong một miền xem như có *chìa* và được thao tác trên đối tượng nếu *chìa* khớp với *khóa*

So sánh các phương pháp cài đặt ma trận truy cập

- Bảng toàn cục: Cài đặt đơn giản, tốn bộ nhớ
- Danh sách truy cập: Liên quan trực tiếp đến nhu cầu NSD, khó xác định quyền truy cập cho các miền
- Danh sách khả năng: Dễ dàng xác định quyền truy cập cho các miền, không liên quan trực tiếp đến nhu cầu NSD
- Khóa-chìa: Kết hợp được ưu điểm của danh sách truy cập và danh sách khả năng

Hủy bỏ quyền truy cập

- Các vấn đề cần xem xét:
  - Hủy ngay hay có trễ? Nếu có trễ→Khi nào?
  - Phạm vi ảnh hưởng: Toàn bộ NSD hay chỉ một nhóm NSD nhất định?
  - Hủy bỏ một số quyền nhất định hay tất cả các quyền?
  - Hủy tạm thời hay vĩnh viễn?

An ninh

Vấn đề an ninh

Xác thực

Các mối đe dọa chương trình và hệ thống

Mã hóa

Vấn đề an ninh

- An ninh: Xem xét môi trường bên ngoài hệ thống để bảo vệ hệ thống khỏi:
  - Truy cập trái phép
  - Sửa đổi hoặc phá hoại hệ thống
  - Vô tình làm hỏng tính nhất quán của hệ thống
- Dễ đảm bảo an ninh tránh các hành động vô ý hơn là đảm bảo anh ninh cho sự phá hoại/truy cập trái phép có mục đích

Xác thực

- Định danh người sử dụng thường được thực hiện qua mật khẩu
- Mật khẩu phải được giữ bí mật
  - Thường xuyên đổi mật khẩu
  - Sử dụng mật khẩu là các chuỗi ký tự khó đoán
  - Ghi lại tất cả những lần login không thành công
- Mật khẩu có thể được mã hóa hoặc sử dụng một lần (ví dụ: SecurID)
- Có thể sử dụng công nghệ mới, ví dụ xác thực sinh trắc học

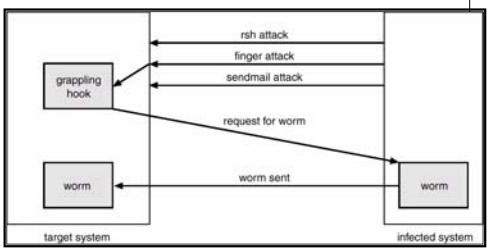
Đe dọa chương trình

- Trojan Horse (Con ngựa thành T'roa)
  - Đoạn mã được sử dụng sai mục đích
  - Khai thác cơ chế setuid
- Trap Door (Cửa sập)
  - Người lập trình để ngỏ một "cửa" chỉ mình anh ta biết để sử dụng sai mục đích, vi phạm an ninh
  - Có thể xuất hiện trong chương trình dịch
- Stack/buffer overflow (tràn bộ đệm/ngăn xếp)

Đe dọa hệ thống

- Worms (Sâu): Chương trình độc lập, có cơ chế tự sinh
- Internet worm (sâu Internet)
  - Khai thác đặc điểm mạng của UNIX (truy cập từ xa) lỗi trong các chương trình *finger* và *sendmail*
  - Grappling hook program uploaded main worm program.
- Viruses (Vi rút) – Đoạn mã ký sinh vào các chương trình khác
  - Chủ yếu ảnh hưởng đến các *máy vi tính*
  - Lây nhiễm qua các phương tiện lưu trữ, qua chương trình
  - *Safe computing*. (Tính toán an toàn)
- Từ chối dịch vụ: Làm cho máy bị tấn công hoạt động quá tải dẫn đến không phục vụ được các yêu cầu

Sâu Internet của Albert Morris (1998)



Kiểm soát các đe dọa

- Kiểm tra các hành động có thể gây mất an ninh (ví dụ liên tục gõ sai mật khẩu)
- Ghi nhật ký hệ thống: Thời gian, NSD các loại truy cập đến các đối tượng – hữu ích cho việc tìm ra cơ chế an ninh tốt hơn cũng như khôi phục việc mất an ninh
- Quét hệ thống định kỳ để tìm ra các lỗ hổng an ninh

Kiểm soát

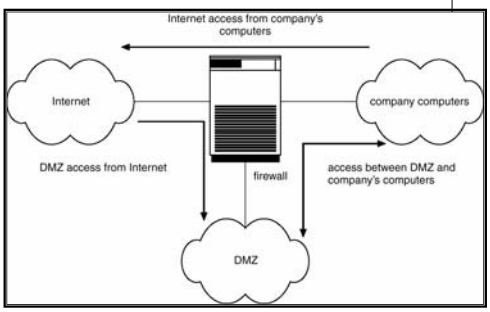
- Cần kiểm soát:
  - Mật khẩu ngắn và dễ đoán (ví dụ abc123)
  - Các chương trình có setuid và chưa được xác thực
  - Các chương trình chưa được xác thực trong các thư mục hệ thống
  - Các tiến trình thực hiện quá lâu
  - Các thư mục được bảo vệ không đúng cách

- Các tệp dữ liệu hệ thống được bảo vệ không đúng cách
- Các phần tử "nguy hiểm" trong PATH (ví dụ Trojan horse)
- Kiểm tra các chương trình hệ thống có bị thay đổi hay không thông qua checksum, MD5...

Tường lửa (firewall)

- Tường lửa được đặt (hoạt động) giữa các máy chủ tin cậy và các máy không tin cậy
- Tường lửa hạn chế truy cập qua mạng giữa hai miền an ninh khác nhau

Tường lửa



Phát hiện đột nhập

- Phát hiện các cố gắng đột nhập vào hệ thống máy tính
- Phương pháp phát hiện:
  - "Kiểm toán" và ghi nhật ký
  - Tripwire (Phần mềm của UNIX kiểm tra xem một số tệp và thư mục có bị thay đổi không)
- Kiểm soát các hàm hệ thống

Mã hóa

- Mã hóa: Bản rõ → Bản mã
- Đặc điểm của kỹ thuật mã hóa tốt:
  - Tương đối đơn giản để NSD đã được xác thực có thể sử dụng để mã và giải mã
  - Sơ đồ mã hóa không chỉ phụ thuộc thuật toán mà còn phụ thuộc tham số (ví dụ khóa)
  - Rất khó phát hiện khóa
- Hệ mã hóa công khai sử dụng hai khóa
  - public key – khóa công khai, dùng để mã hóa.
  - private key – khóa bí mật, dùng để giải mã

Ví dụ mã hóa: SSL

- SSL – Secure Socket Layer
- Là giao thức mã hóa cho phép hai máy tính trao đổi dữ liệu an toàn với nhau
- Thường sử dụng giữa web server và browser để trao đổi thông tin một cách an toàn (ví dụ nạp số thẻ tín dụng)
- Web server được kiểm tra qua chứng chỉ
- Sau khi đã thiết lập kết nối an toàn SSL, hai máy tính truyền thông với khóa đối xứng

Phân loại an ninh máy tính

- Bộ Quốc phòng Mỹ chia an ninh máy tính thành 4 mức từ cao đến thấp: A, B, C, D.
- Xem thêm trong giáo trình về 4 mức an ninh này

Các vấn đề cần nhớ

- Phân biệt bảo vệ và an ninh
- Quyền truy cập, miền bảo vệ
- Ma trận truy cập, các phương pháp cài đặt ma trận truy cập, so sánh các phương pháp đó
- An ninh máy tính: Xác thực, các mối đe dọa, kiểm soát, mã hóa