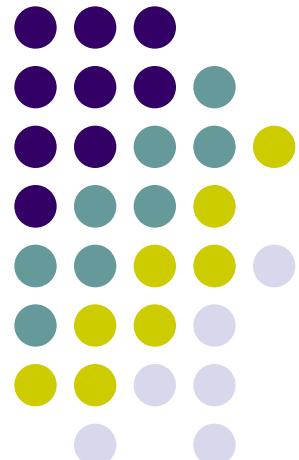


Nguyên lý hệ điều hành

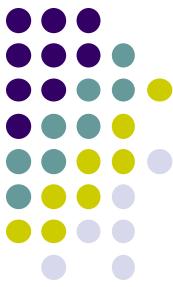
Lê Đức Trọng
Khoa CNTT
Trường Đại học Công nghệ
(Slide credit: PGS. TS. Nguyễn Hải Châu)





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 như Linux, Unix
- 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

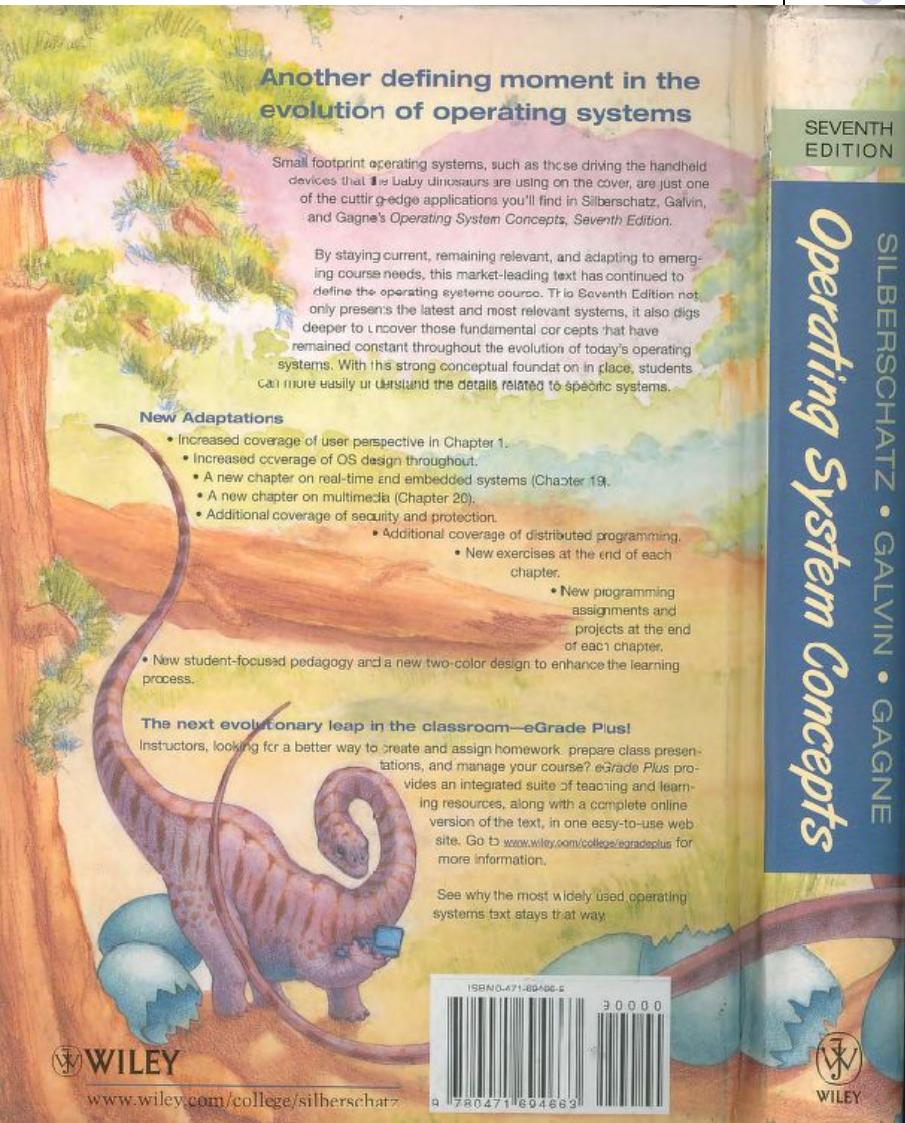
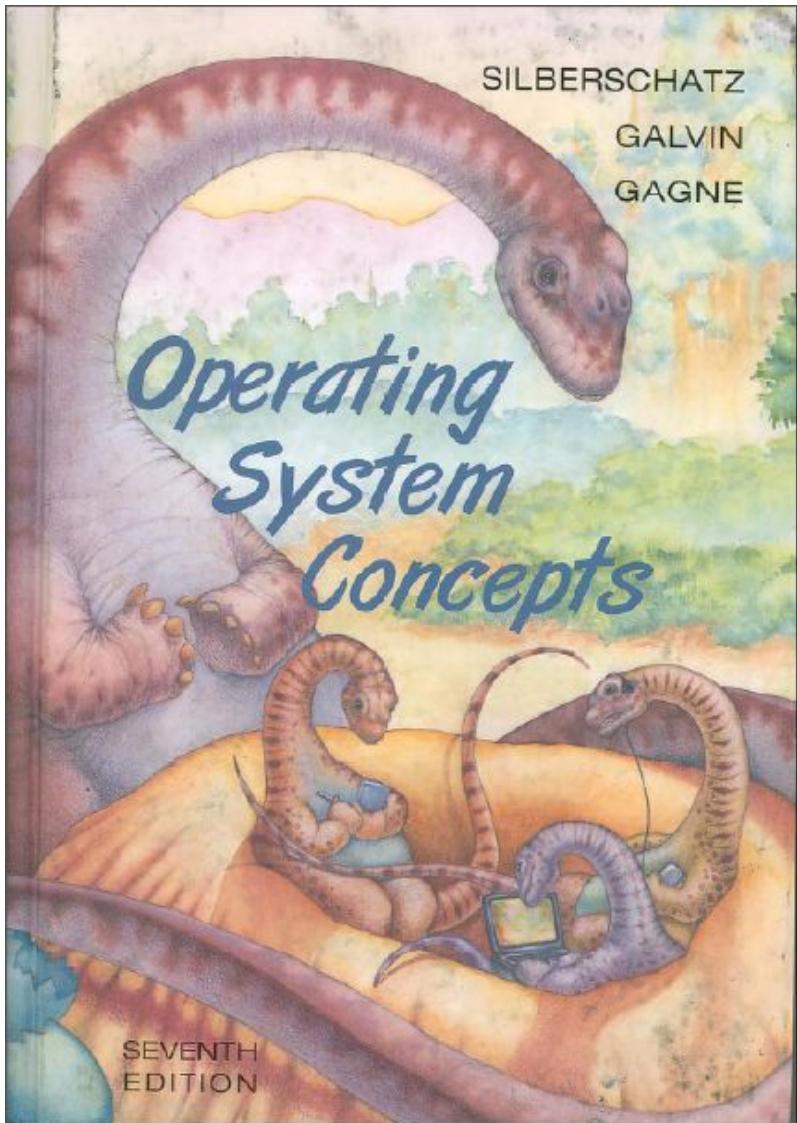


Nội dung

- Gồm có 5 phần chính:
 - Tổng quan (4 tiết)
 - Quản lý tiến trình (16 tiết)
 - Quản lý lưu trữ (16 tiết)
 - Hệ vào/ra (12 tiết)
 - Bảo vệ và an ninh (8 tiết)



Giáo trình





Đánh giá, thi và kiểm tra

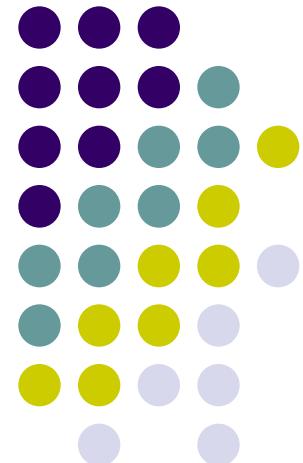
- Chuyên cần, kiểm tra hàng tuần (10%)
- Kiểm tra giữa kỳ (30%)
 - Thi trắc nghiệm MCQ (30p – 30 câu)
 - Là điều kiện bắt buộc để được thi cuối kỳ
 - Tuần 8, sau phần quản lý tiến trình
 - Được sử dụng tài liệu
- Đánh giá cuối kỳ (60%)
 - Bài tập lớn
 - Trình bày tuần 13-15



Bài tập lớn

- Sinh viên thực hiện theo nhóm 4 bạn
- Đề bài: Tìm hiểu cấu trúc, phương thức hoạt động của 01 hệ điều hành
- Thời gian:
 - Đăng ký nhóm + đề tài: Tuần 4
 - Trình bày báo cáo: Tuần 13-15
- Mỗi bạn phải chịu trách nhiệm ít nhất 01 nội dung

Giới thiệu

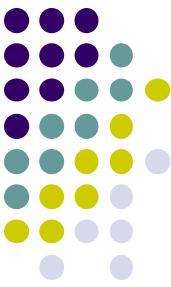




Máy tính - tài nguyên máy tính



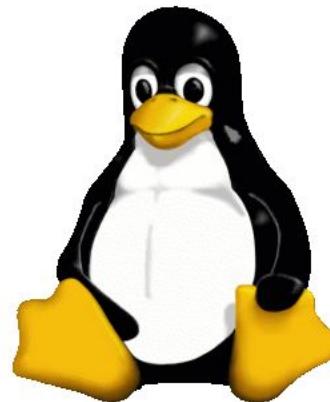
- Tài nguyên:
 - CPU
 - Bộ nhớ trong
 - Đĩa cứng
 - Thiết bị ngoại vi (máy in, màn hình, bàn phím, card giao tiếp mạng, USB...)



Hệ điều hành là gì?



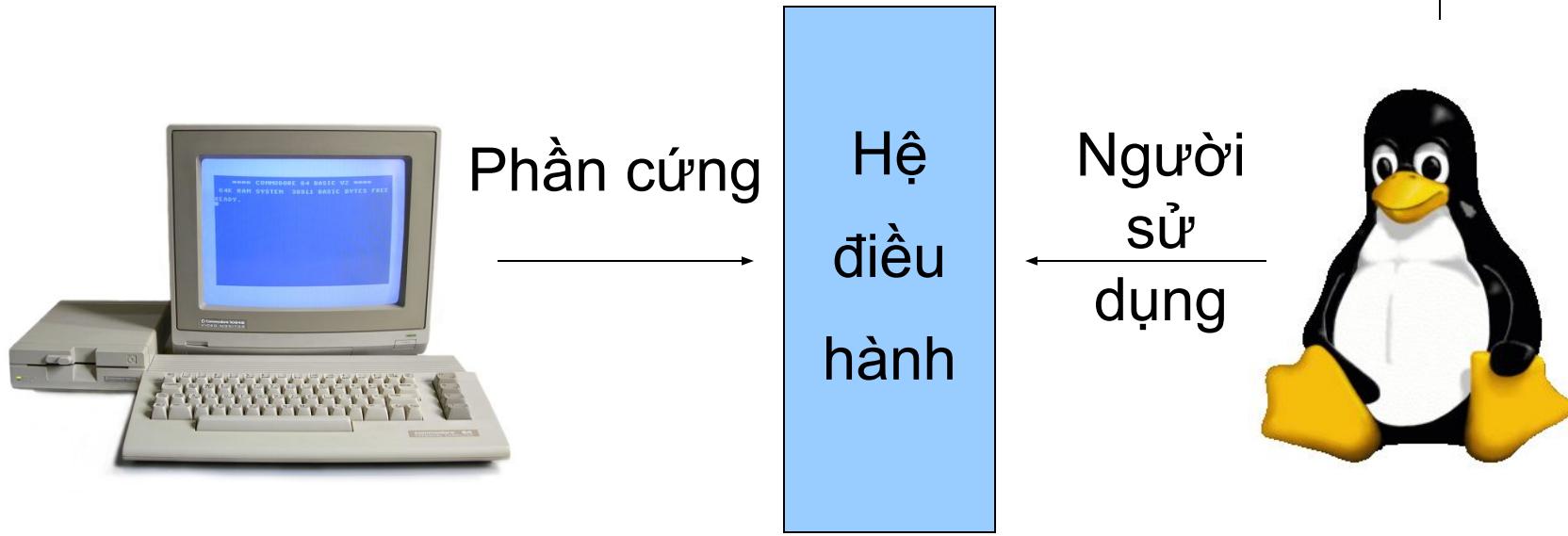
Hệ
điều
hành



- 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*

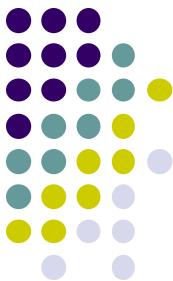


Hai cách nhìn hệ điều hành



- Phàn 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

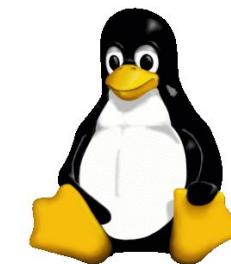
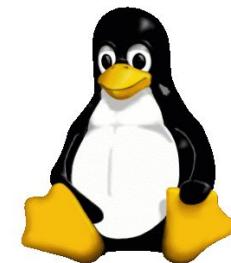
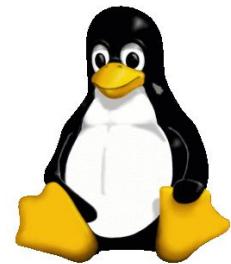
Hệ thống máy tính



Phần cứng

Hệ
điều
hành

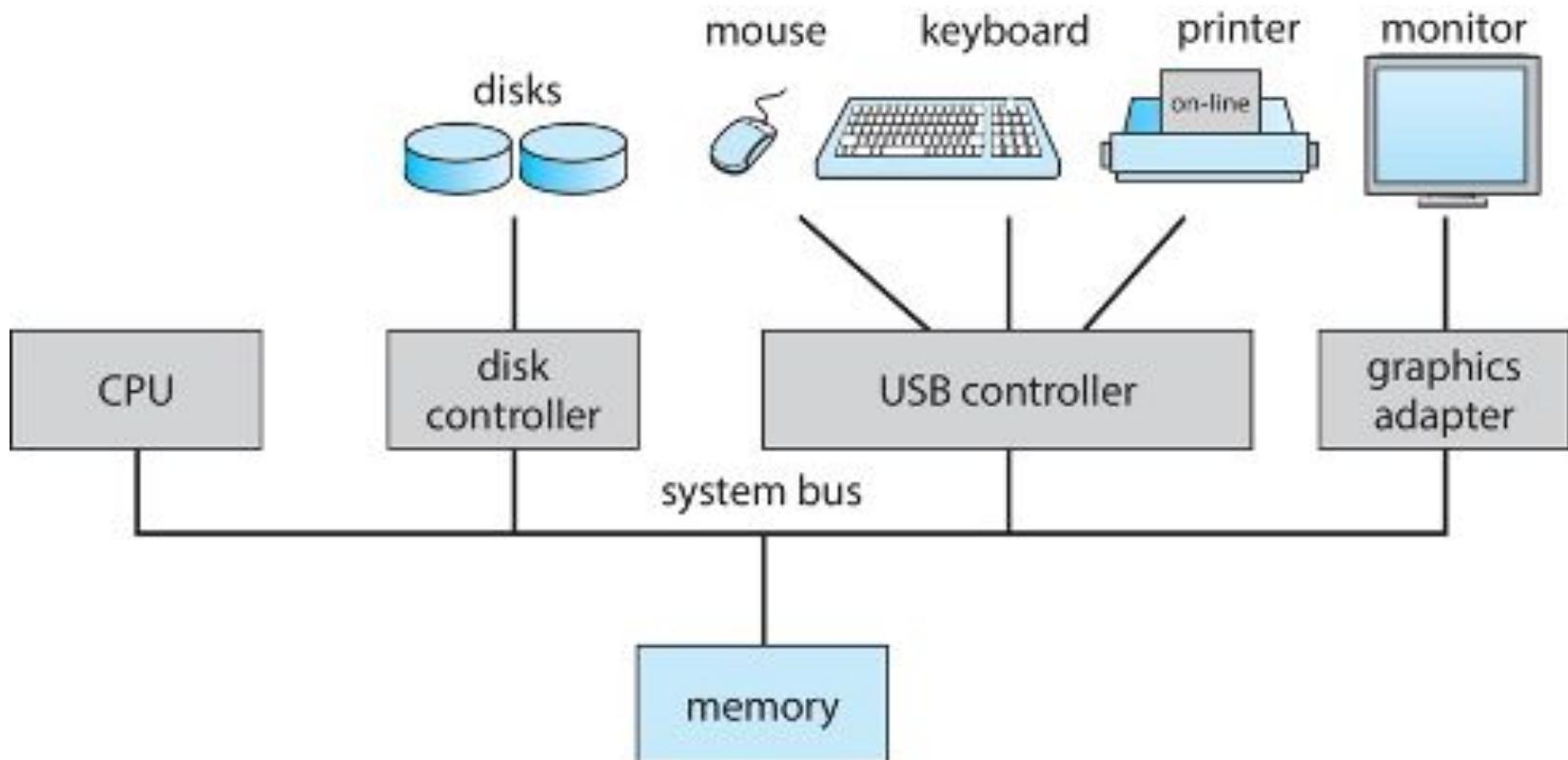
Các
chương
trình
hệ thống
và
ứng dụng



Người sử dụng



Hệ thống máy tính





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)



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



Đa chương trình

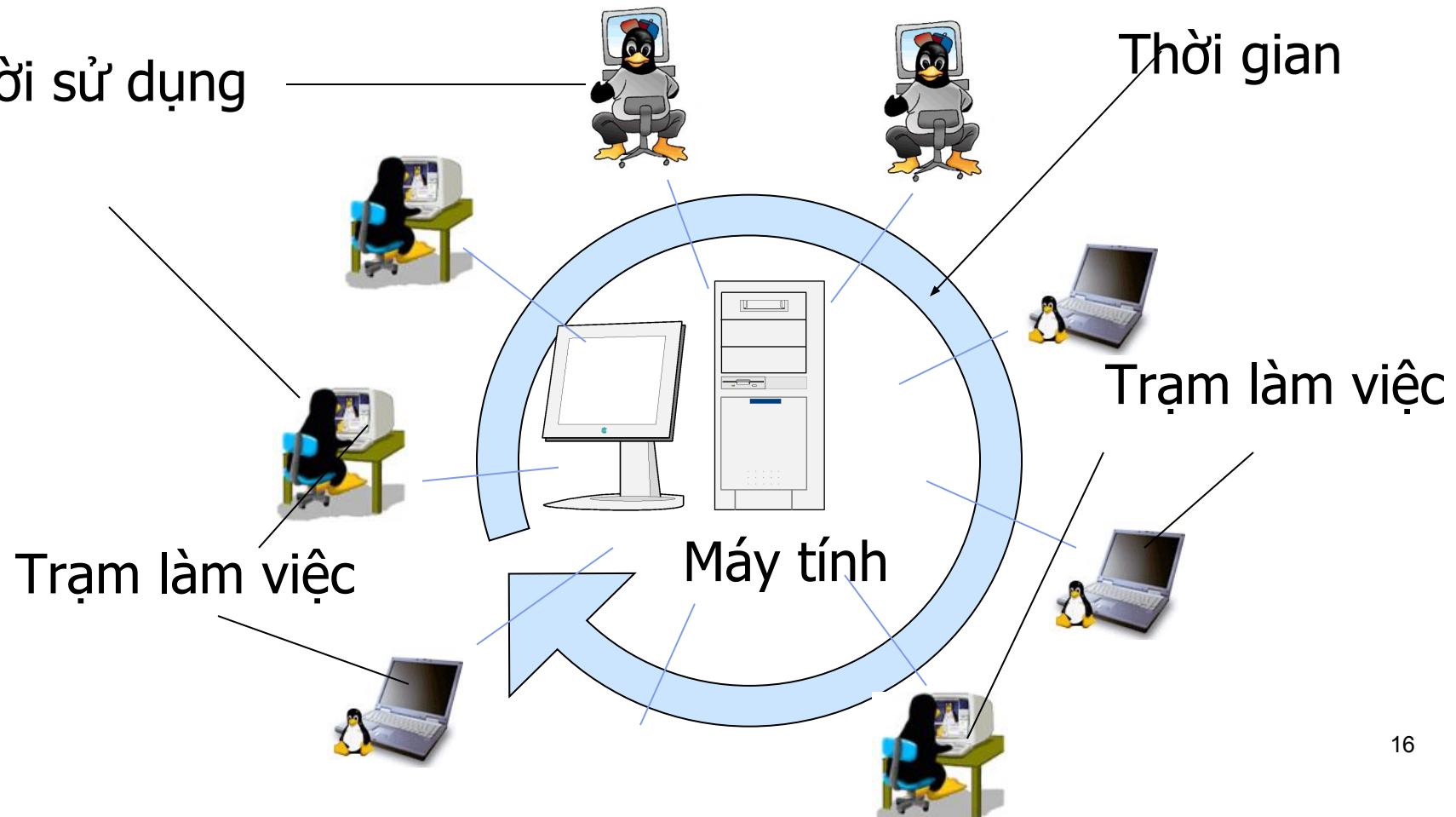
- Thuật ngữ: *Multiprogramming*
- Các chương trình được xếp vào hàng đợi
- Một chương trình được thực hiện và chiếm giữ CPU cho đến khi (1) có yêu cầu vào/ra, hoặc (2) kết thúc
- Khi (1) hoặc (2) xảy ra, chương trình khác sẽ được thực hiện
- Tận dụng CPU tốt hơn xử lý theo lô đơn giản



Phân chia thời gian/đa nhiệm

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

Người sử dụng



Trạm làm việc

Thời gian

Trạm làm việc



Một số hệ điều hành

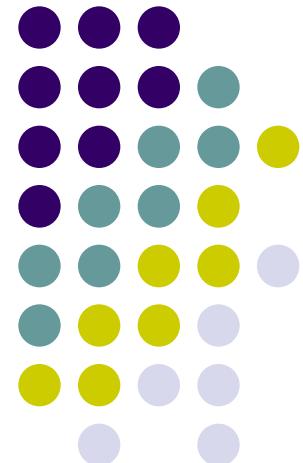
- UNIX (UNiplexed Information and Computing Service): (1) AT&T System V (2) Berkeley (BSD)
 - AIX dựa trên System V (IBM)
 - HP-UX dựa trên BSD (Hewlett-Packard)
 - IRIX dựa trên System V (Silicon Graphics Inc.)
 - Linux
 - Solaris, SunOS (Sun Microsystems)
 - Xenix (Microsoft)



Một số hệ điều hành

- Windows (Microsoft): Windows 3.x, Windows 95, Windows 98, Windows 2000, Windows NT, Windows XP, Vista, 7, 10
- Apple: Mac OS, Mac OS X, iOS, iPadOS
- BeOS (Be Inc. 1991) => Haiku
- DOS
- PalmOS, Symbian
- Android

Cấu trúc hệ điều hành





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
- Quản lý kết nối mạng
- Bảo vệ và an ninh
- Thông dịch lệnh



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 tập
- Kết nối
- Phát hiện lỗi
- Cấp phát tài nguyên
- Đưa ra các cơ chế bảo vệ và an ninh



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)



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



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



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



Hàm HT bảo trì thông tin

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



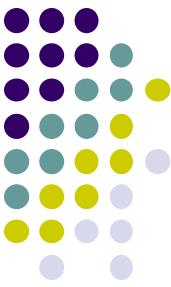
Hàm HT về kết nối

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



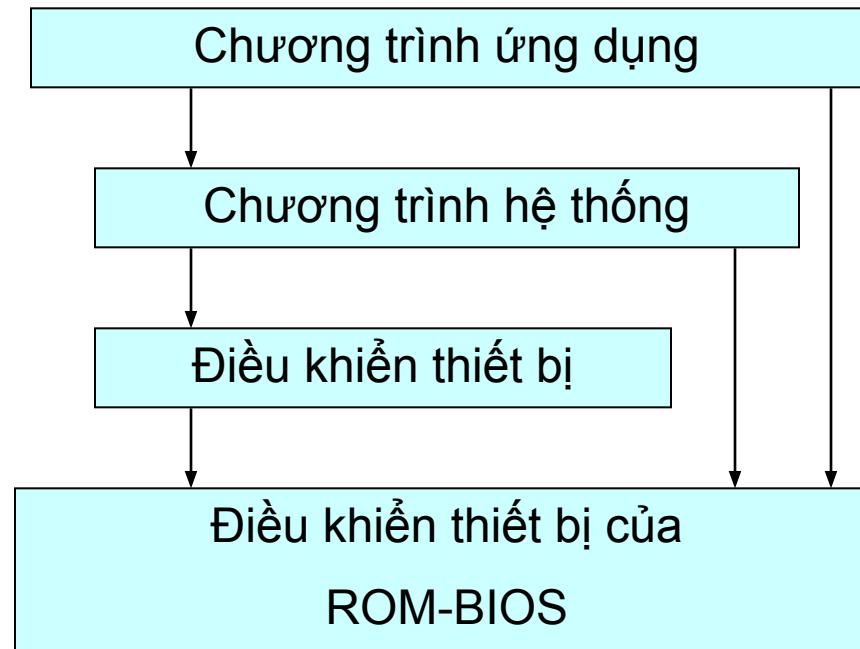
Các chương trình hệ thống

- Các chương trình hệ thống cung cấp môi trường thuận tiện cho việc thực hiện và phát triển chương trình, 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
 - Kết nối
 - Cách nhìn HĐH của NSD được xác định qua các chương trình hệ thống, không thực sự qua các hàm hệ thống (system calls).



Cấu trúc HĐH: Đơn giản

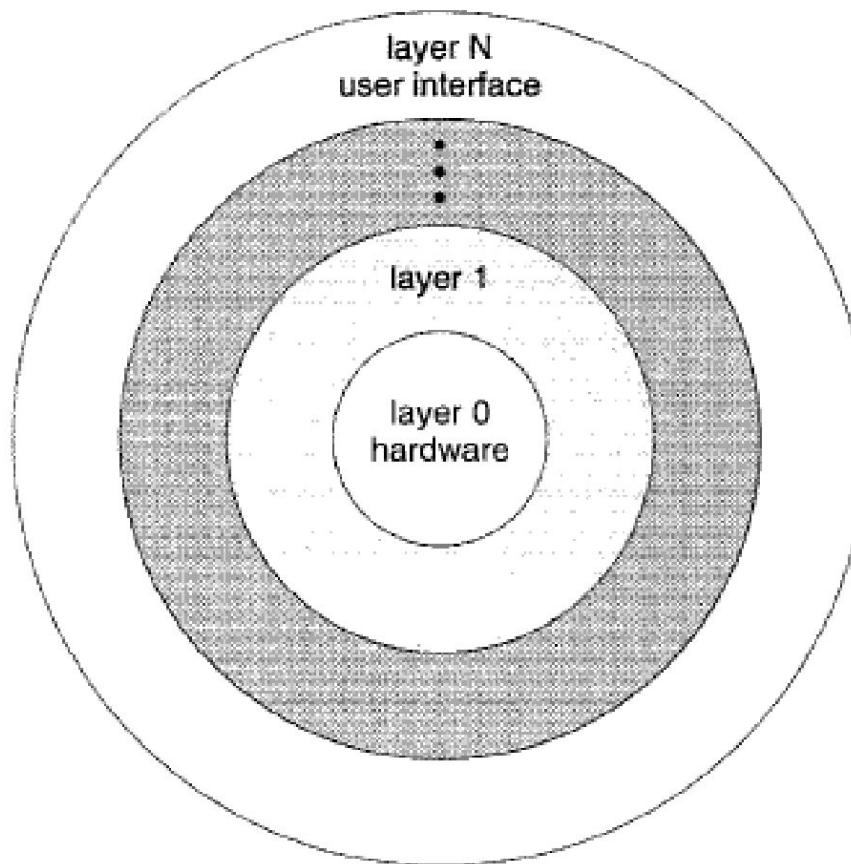
- Thuật ngữ: Simple approach
- Ví dụ MS-DOS. (tương tự: UNIX thời gian đầu)





Cấu trúc HĐH: Phân tầng

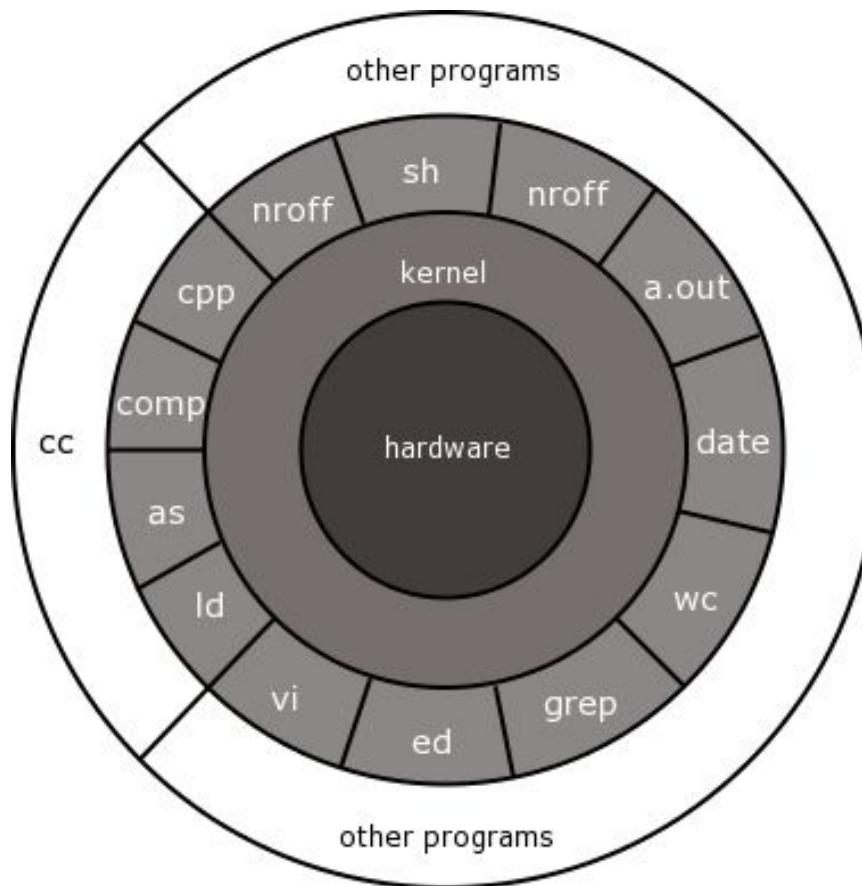
- Thuật ngữ: Layered approach





Cấu trúc HĐH: Phân tầng

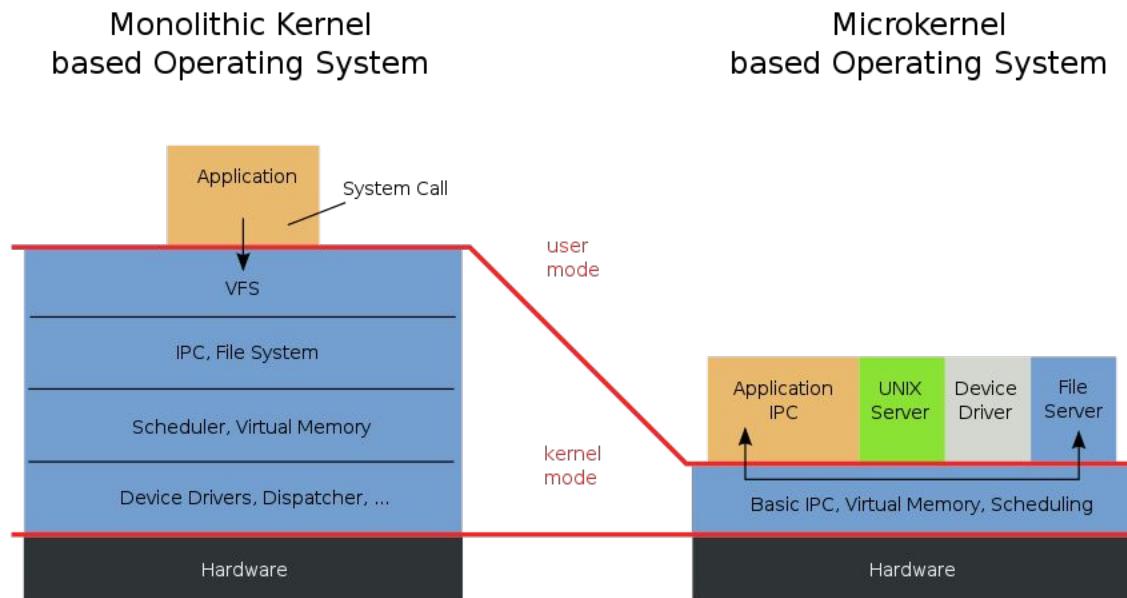
- Ví dụ UNIX





Cấu trúc HĐH: Vi nhân

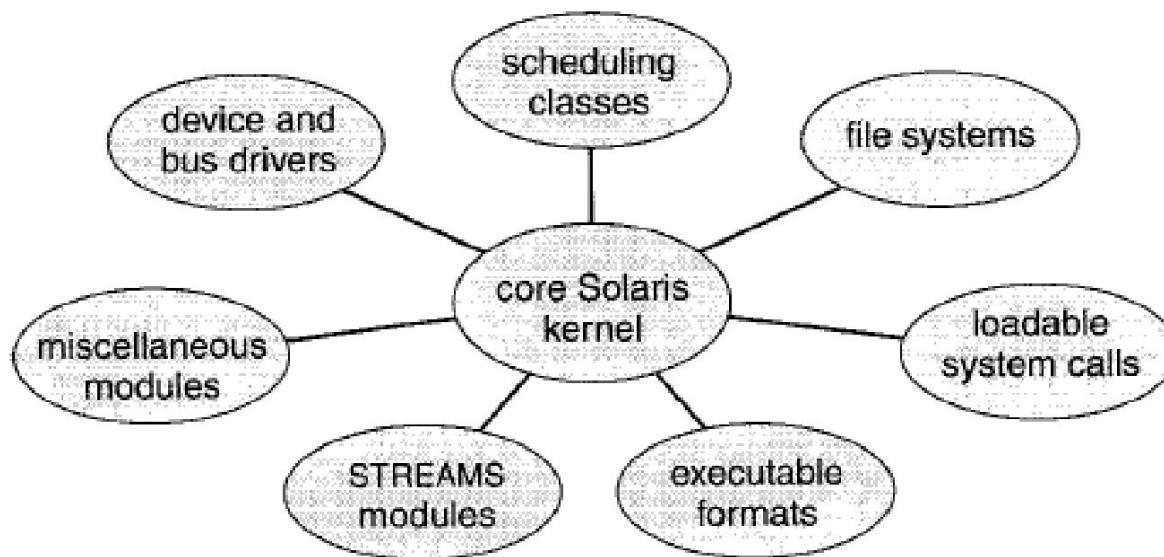
- Thuật ngữ: Microkernel
- Giữ cho nhân có các đủ các chức năng thiết yếu nhất để giảm cỡ
- Các chức năng khác được đưa ra ngoài nhân





Cấu trúc HĐH: Module

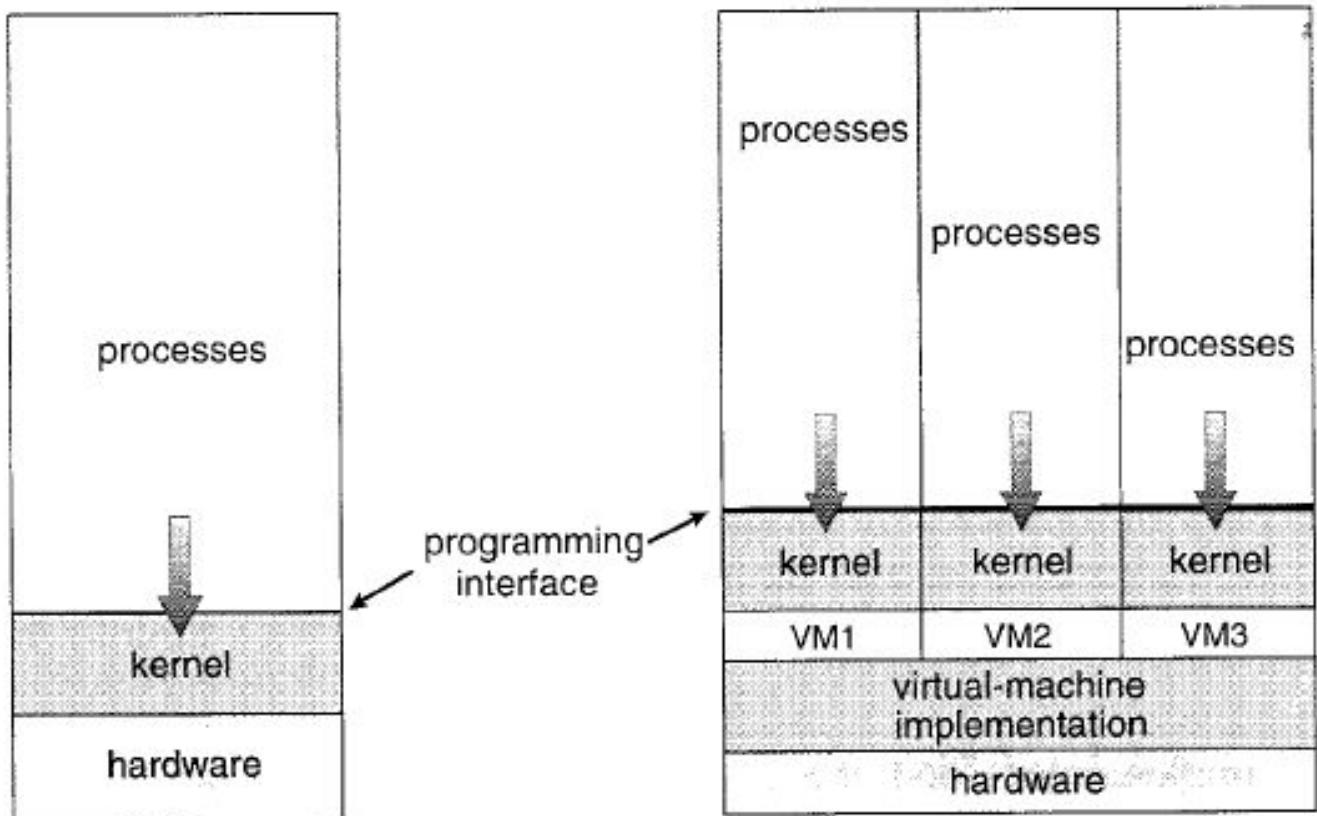
- Thuật ngữ: Module approach
- Hiện tại đây là cách tiếp cận tốt nhất (sử dụng được các kỹ thuật lập trình hướng đối tượng). Ví dụ: Solaris của Sun Microsystem:





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

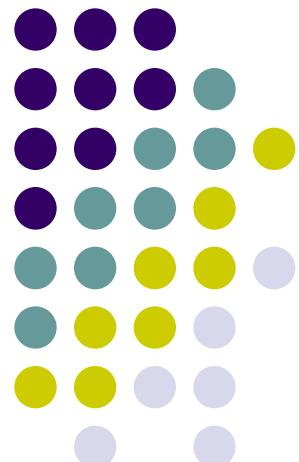


Tìm hiểu thêm

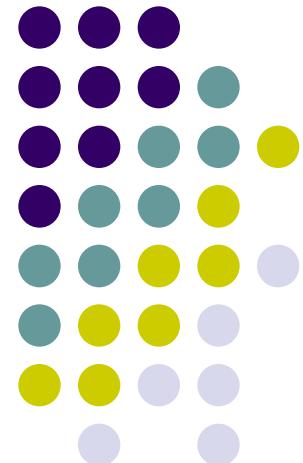
- **Không bắt buộc**
- Chương 1, 2:
<https://codex.cs.yale.edu/avi/os-book/OS10/slides-dir/index.html>
- Bổ sung một hàm hệ thống mới vào nhân Linux và sử dụng hàm đó:
 - Đọc hướng dẫn trong giáo trình từ trang 74-78
 - Thủ nghiệm trên RedHat Fedora hoặc Ubuntu/Debian

Nguyên lý hệ điều hành

Lê Đức Trọng
DS&KTLab, Khoa CNTT
Trường Đại học Công nghệ
(Slide credit: PGS. TS. Nguyễn Hải Châu)



Khái niệm tiến trình





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

The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. The window lists various processes with their names, users, CPU usage, and memory usage. The table has columns for Image Name, User Name, CPU, and Mem Usage.

| Image Name | User Name | CPU | Mem Usage |
|---------------|-----------------|-----|-----------|
| POWERPNT.EXE | Chau | 00 | 12,532 K |
| taskmgr.exe | Chau | 00 | 18,496 K |
| WINWORD.EXE | Chau | 00 | 57,096 K |
| OUTLOOK.EXE | Chau | 00 | 74,068 K |
| ctfmon.exe | Chau | 00 | 6,508 K |
| UniKeyNT.exe | Chau | 00 | 14,232 K |
| wdfmgr.exe | LOCAL SERVICE | 00 | 4,704 K |
| ULCDRSvr.exe | SYSTEM | 00 | 2,472 K |
| Rtvscan.exe | SYSTEM | 00 | 83,772 K |
| YAHOOOM~1.EXE | Chau | 00 | 70,832 K |
| nvsvc32.exe | SYSTEM | 00 | 6,800 K |
| DefWatch.exe | SYSTEM | 00 | 4,960 K |
| spoolsv.exe | SYSTEM | 00 | 16,872 K |
| svchost.exe | LOCAL SERVICE | 00 | 17,828 K |
| svchost.exe | NETWORK SERVICE | 00 | 4,608 K |
| svchost.exe | SYSTEM | 00 | 43,044 K |
| svchost.exe | SYSTEM | 00 | 11,812 K |
| wuauctl.exe | Chau | 00 | 17,788 K |
| VPTray.exe | Chau | 00 | 25,868 K |

Show processes from all users End Process

Processes: 31 CPU Usage: 4% Commit Charge: 917M / 3944M

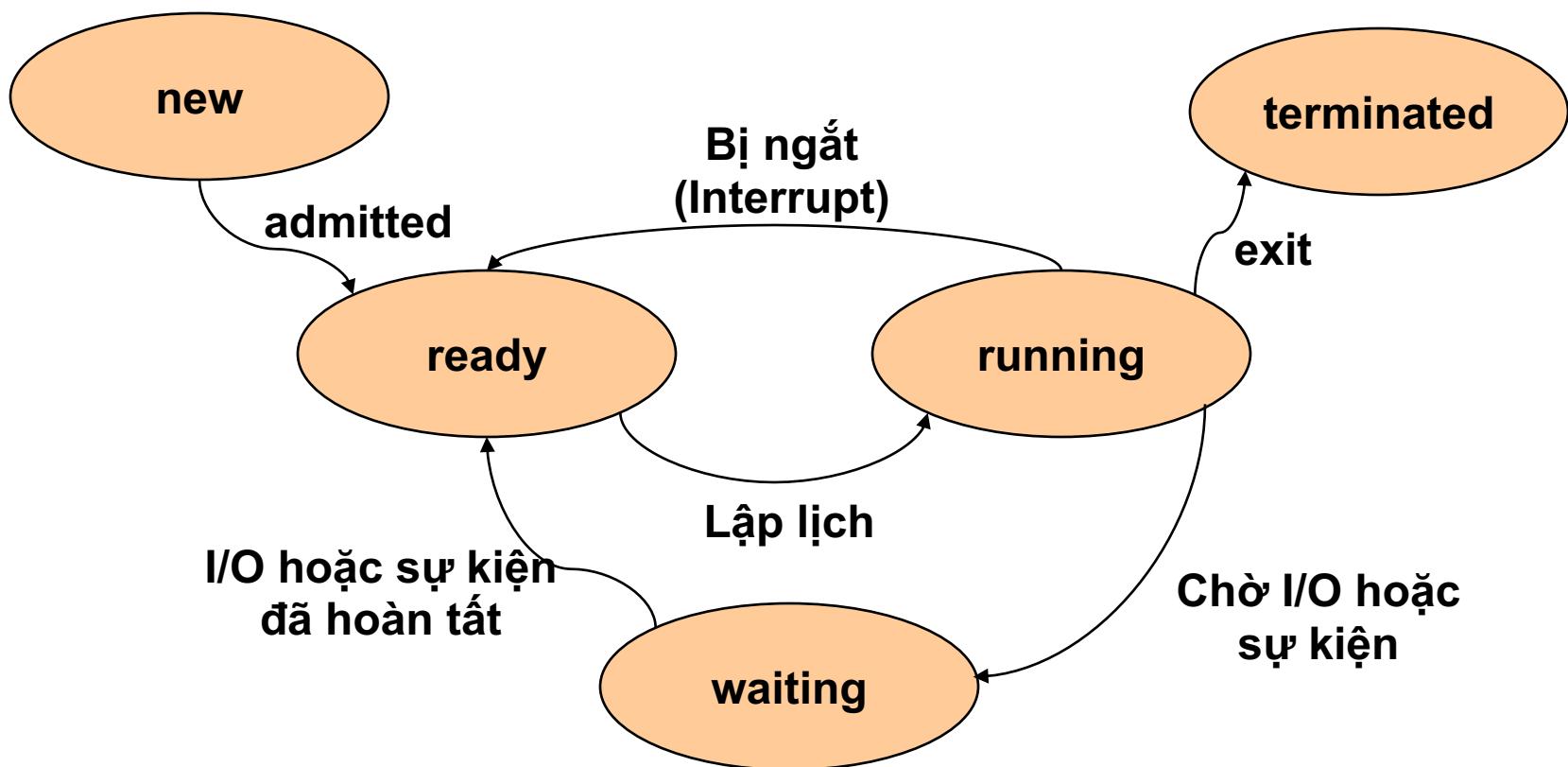


Tiến trình gồm có...

- 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*



Trạng thái tiến trình



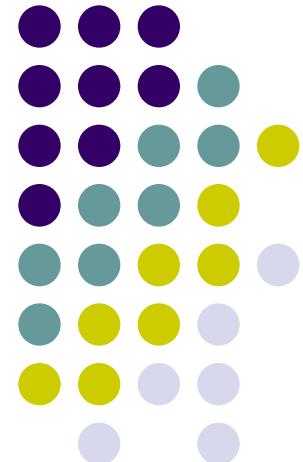


Khối điều khiển tiến trình

- **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 | |
| | |

Lập lịch tiến trình





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

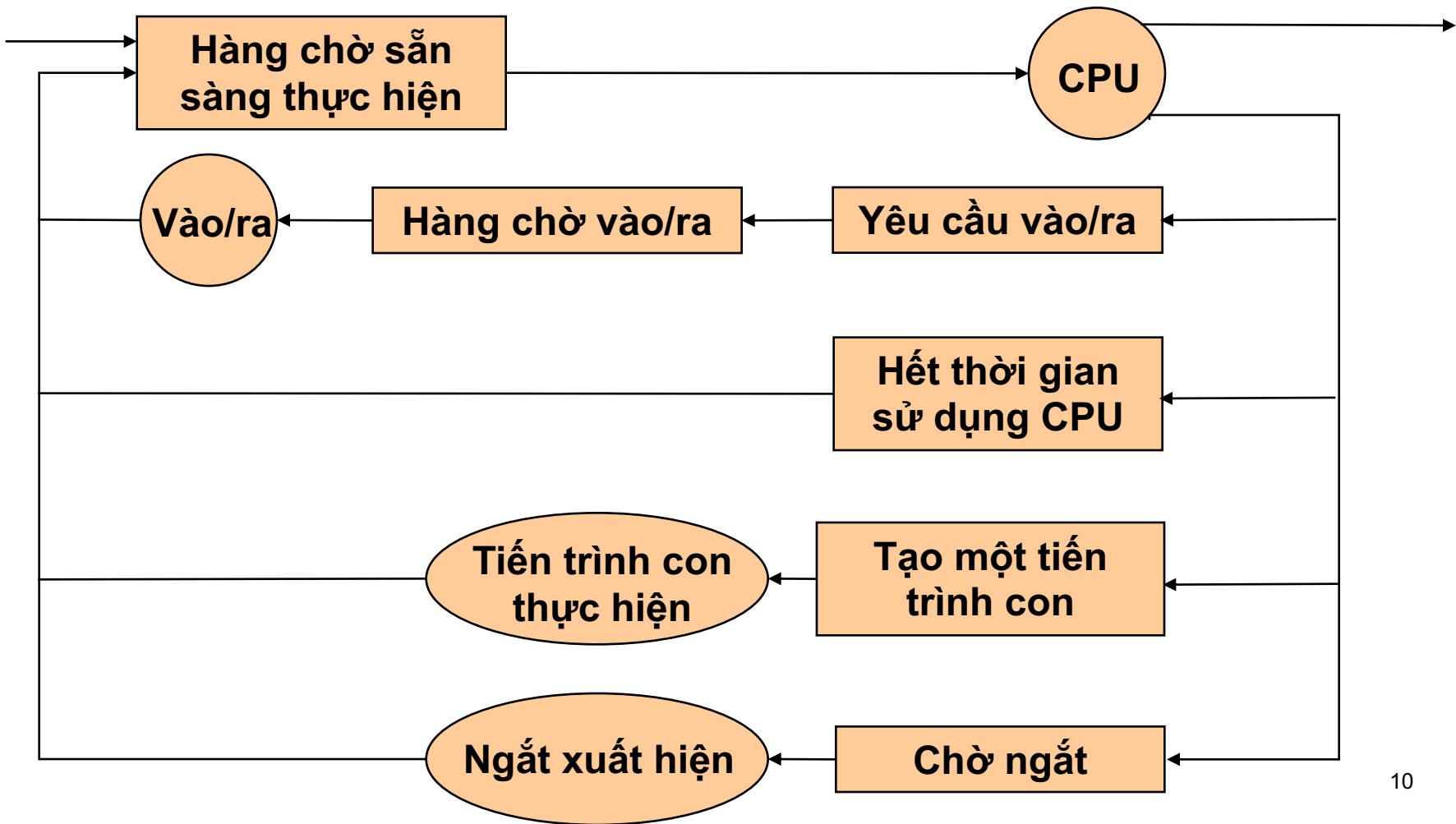


Hàng chờ lập lịch

- 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



Hàng chờ lập lịch tiến trình



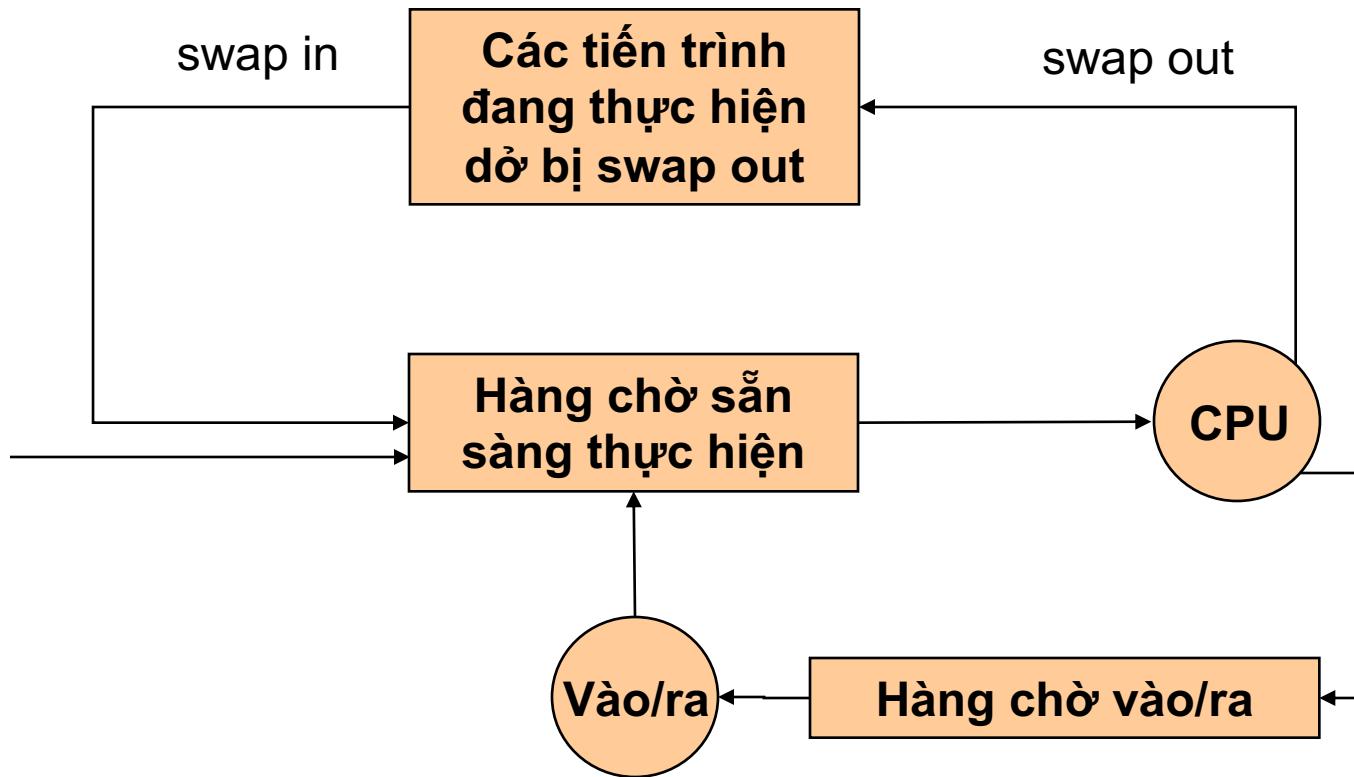


Phân loại các bộ lập lịch

- *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

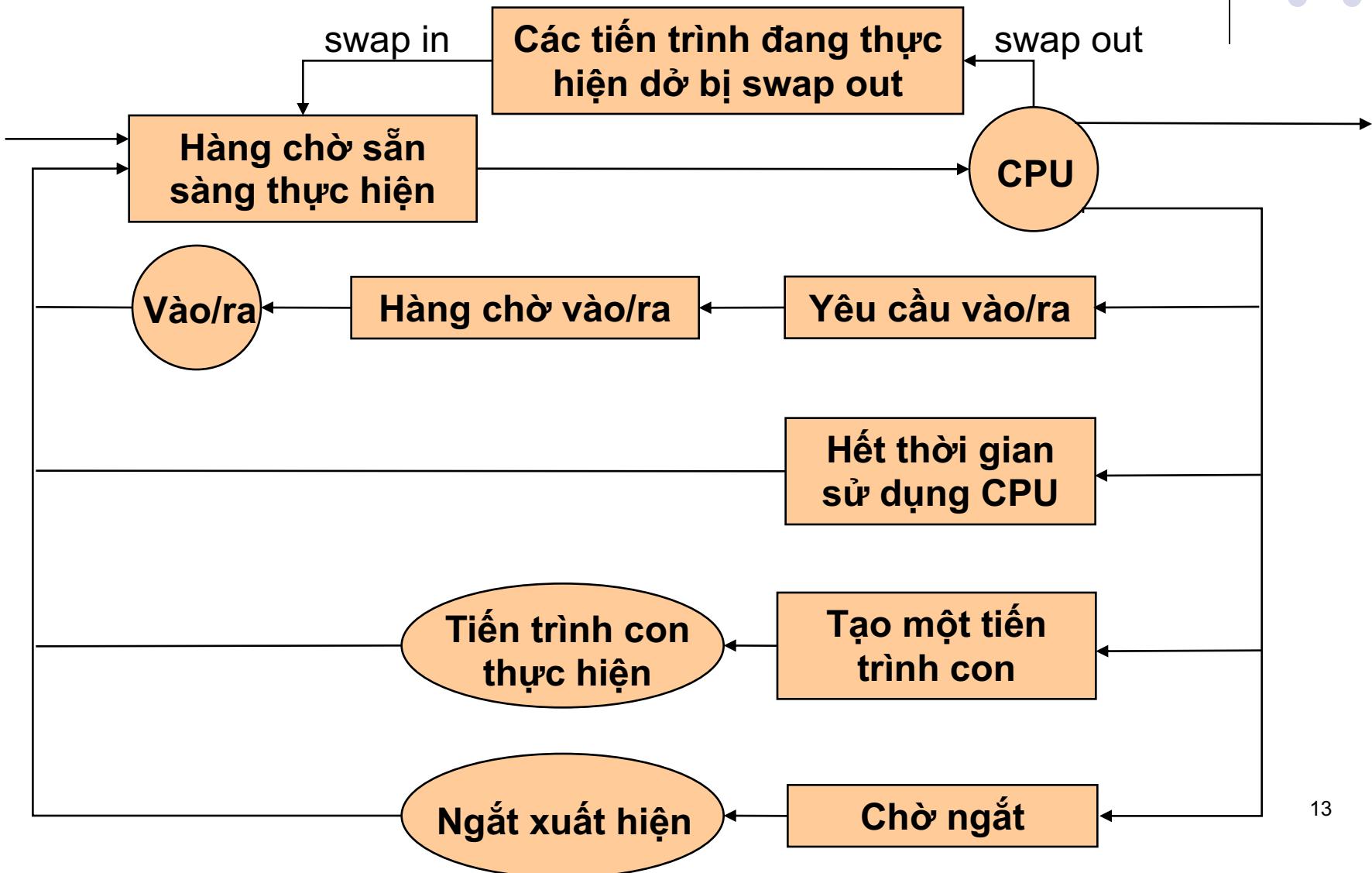


Minh họa bộ lập lịch trung hạn





Hàng chờ lập lịch tiến trình





Chuyển trạng thái

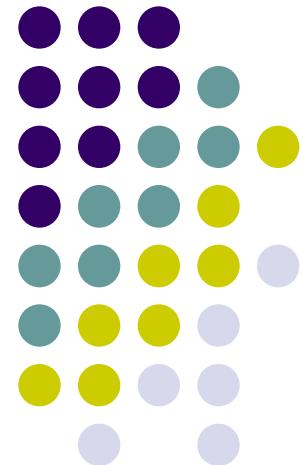
- Thuật ngữ: Context switch
- Xảy ra khi một tiến trình A bị ngắt ra khỏi CPU, tiến trình B bắt đầu được sử dụng CPU
- Cách thực hiện:
 - Nhân HĐH ghi lại toàn bộ trạng thái của A, lấy từ PCB (khối điều khiển tiến trình) của A
 - Đưa A vào hàng chờ
 - Nhân HĐH nạp trạng thái của B lấy từ PCB của B
 -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



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

Các thao tác với tiến trình





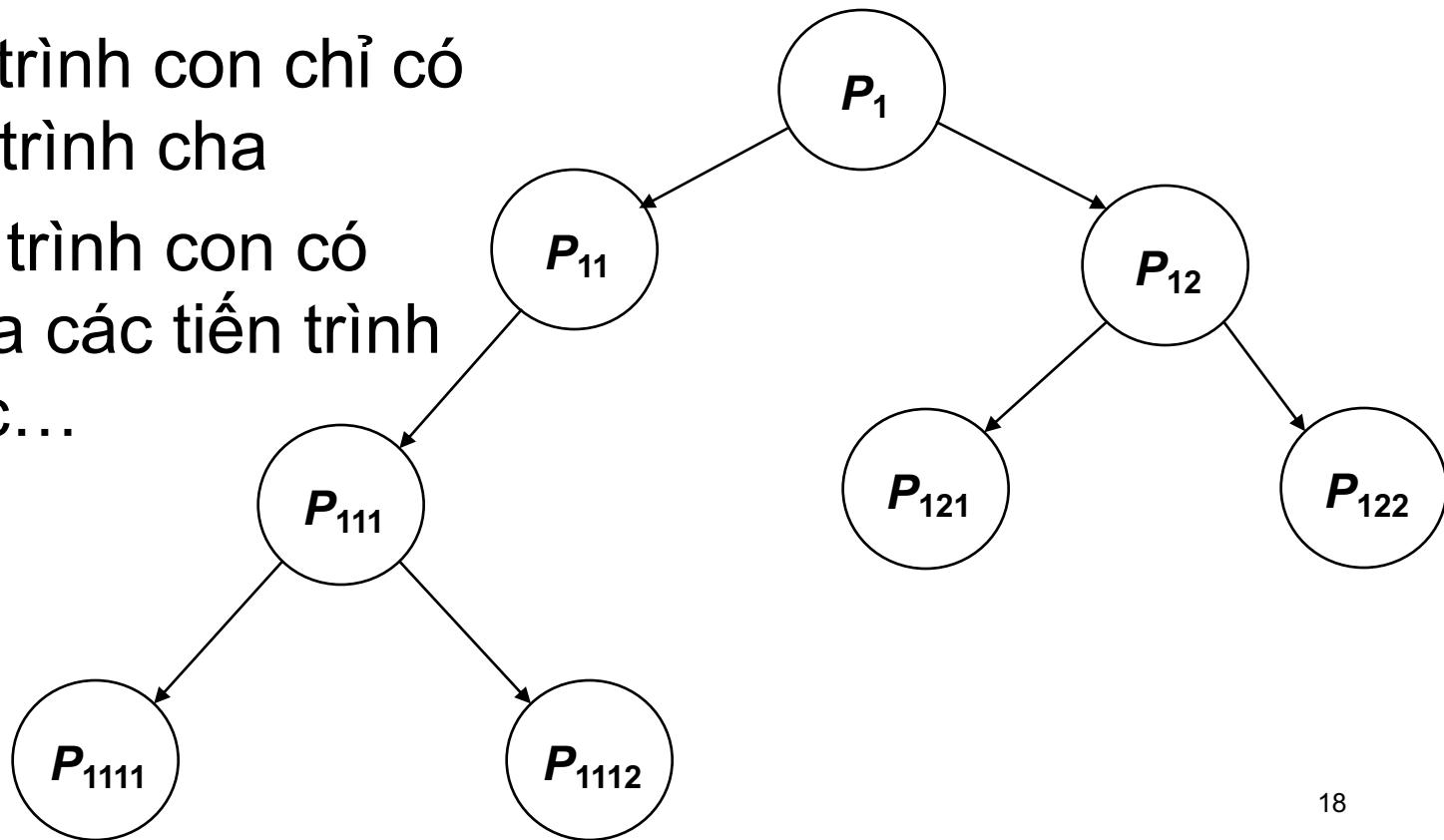
Tạo tiến trình

- 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



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...





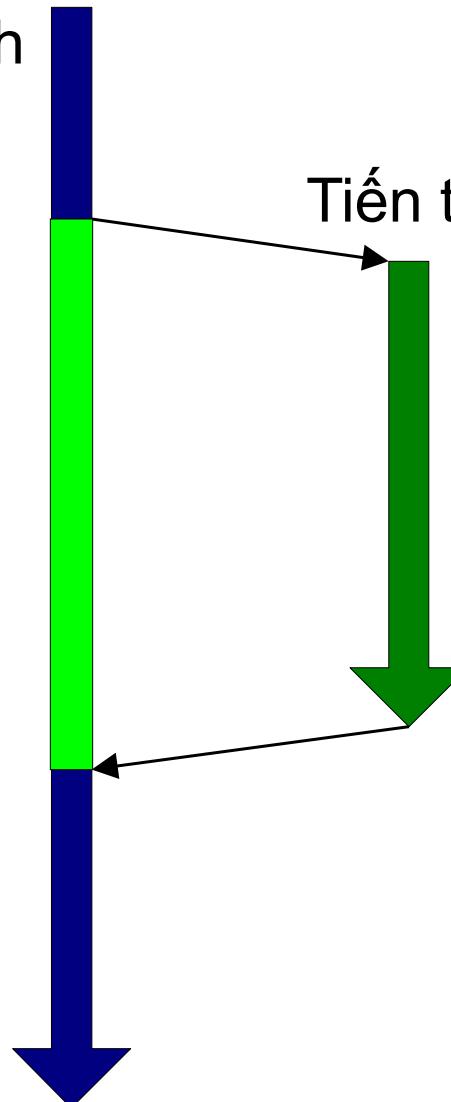
Minh họa tiến trình cha và con

Tiến trình
cha gọi
create-process

Tiến trình con

Có thể gọi hoặc
không gọi **wait** để
chờ/không chờ
tiến trình con kết thúc

Gọi **exit** để kết thúc





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)



Minh họa tiến trình trong UNIX

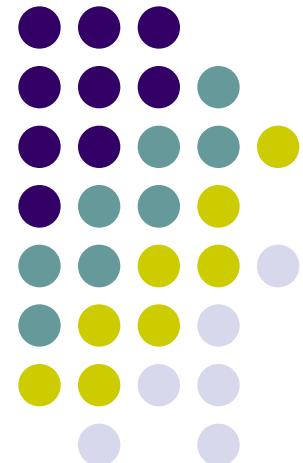
```
#include <stdio.h>
main()
{
    int pid=fork(); /* Tạo tiến trình mới bằng hàm fork() */
    if (pid<0) { perror("Cannot create process"); return(-1); }
    else if (pid==0) { /* Tiến trình con */
        execlp();
    }
    else { /* Tiến trình cha */
        wait(NULL); /* Nếu không có lệnh này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



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)

Truyền thông giữa các tiến trình (IPC)





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



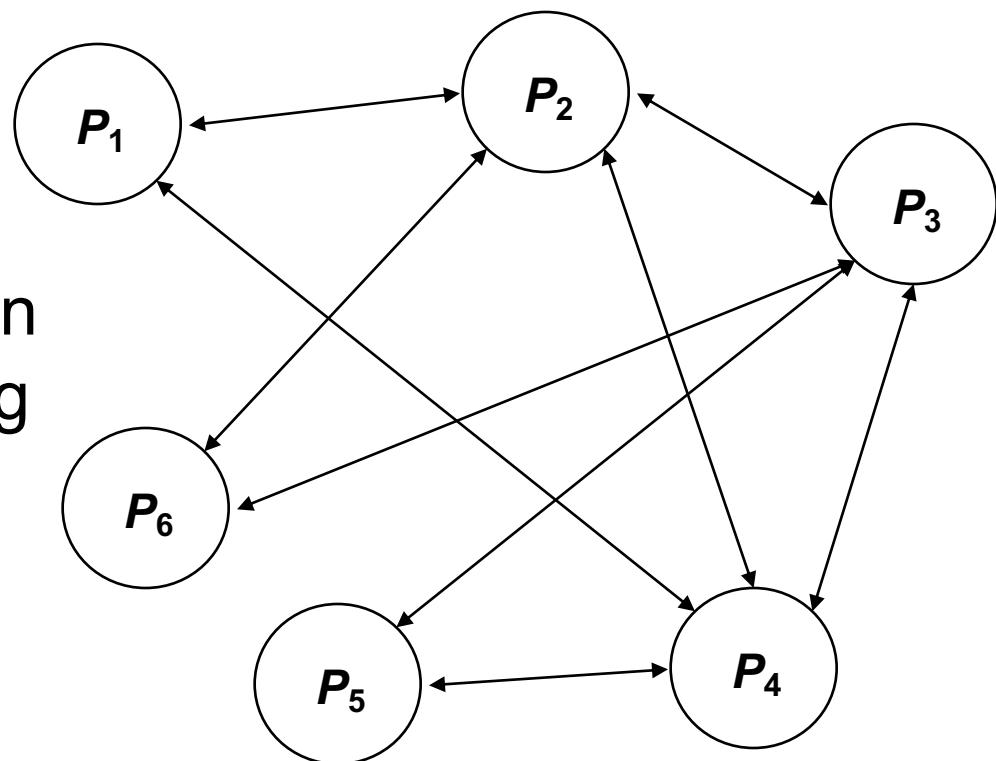
Truyền thông trực tiếp

- 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



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





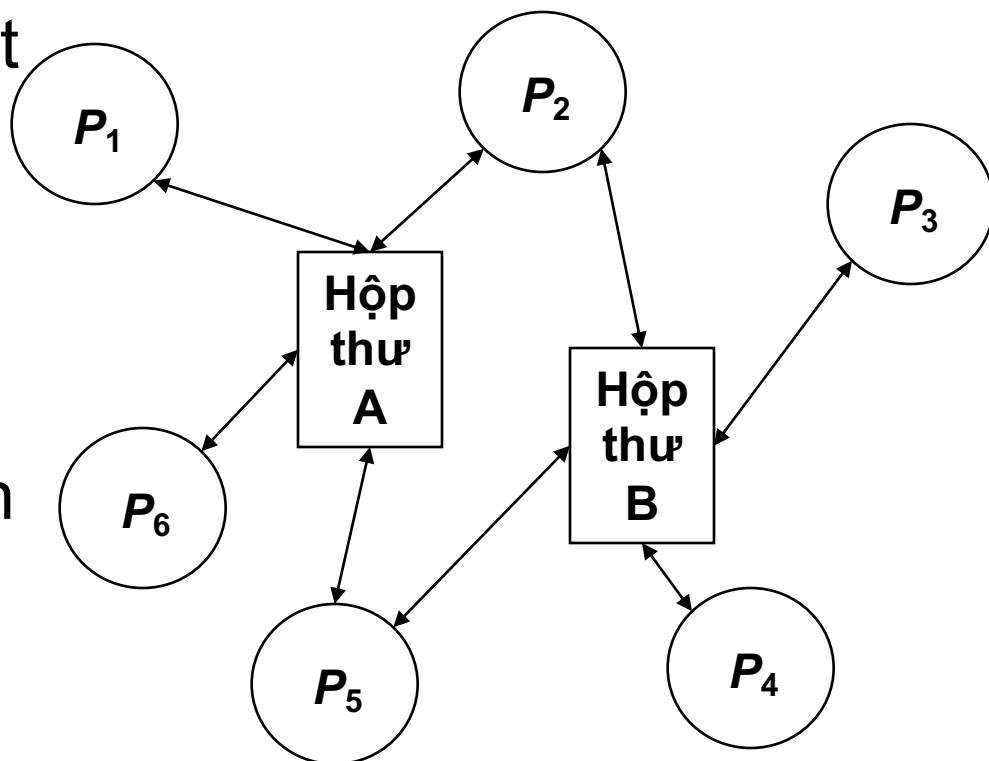
Truyền thông gián tiếp

- 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



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)





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)



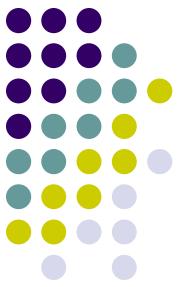
Các phương thức send/receive

| | 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) |



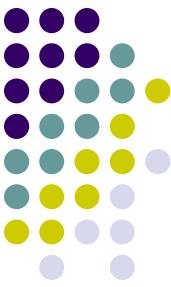
Vấn đề sử dụng vùng đệm

- 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



Luồng (thread)

- Sinh viên tự tìm hiểu trong giáo trình Operating System Concepts trang 127-152.



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)



Bài tập

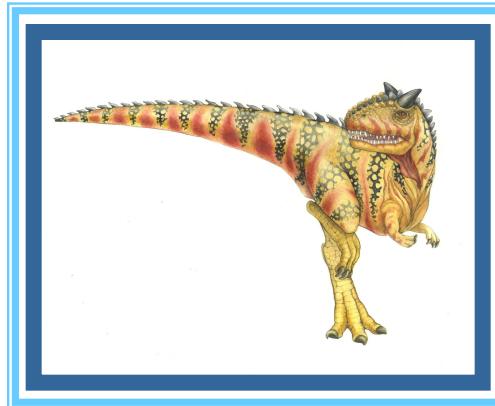
- Viết chương trình C trong Linux/Unix tạo ra 16 tiến trình con. Tiến trình cha chờ cho 16 tiến trình con này kết thúc rồi mới kết thúc bằng hàm **exit**. Sử dụng các hàm **fork** và **wait** để thực hiện yêu cầu.
- Hãy tìm một số ví dụ thực tế minh họa cho các khái niệm lập lịch/hàng chờ trong tình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

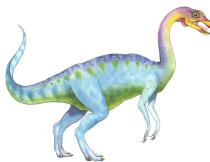


Bài tập

- 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

Chapter 4: Threads





Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples





Objectives

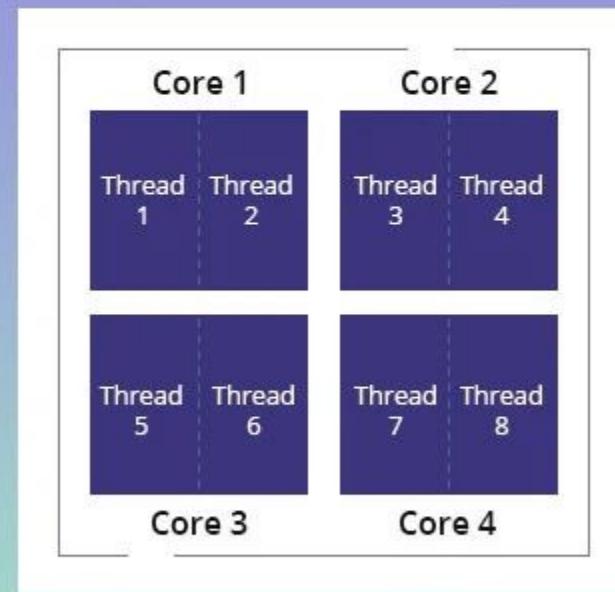
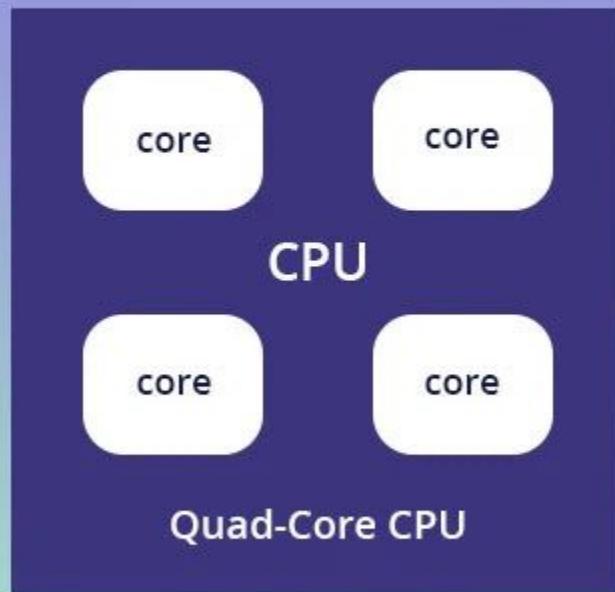
- To introduce the notion of a thread—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Windows, and Java thread libraries
- To explore several strategies that provide implicit threading
- To examine issues related to multithreaded programming
- To cover operating system support for threads in Windows and Linux

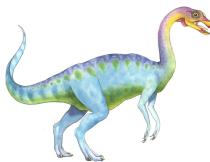




Core vs Thread

CORES vs THREADS





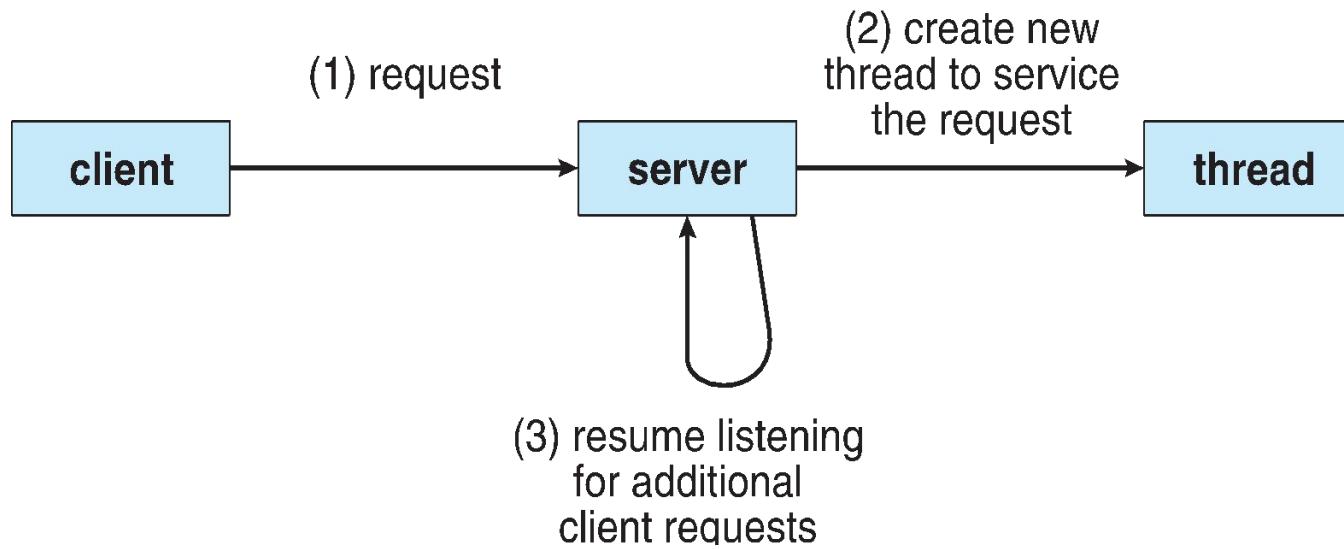
Motivation of Thread

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks within the application can be implemented by separate threads
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded





Multithreaded Server Architecture

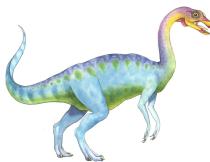




Benefits

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures





Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
 - Single processor / core, scheduler providing concurrency





Parallelism

- Types of parallelism
 - **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
 - **Task parallelism** – distributing threads across cores, each thread performing unique operation
- As the number of threads grows, so does architectural support for threading
 - CPUs have cores as well as **hardware threads**
 - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core



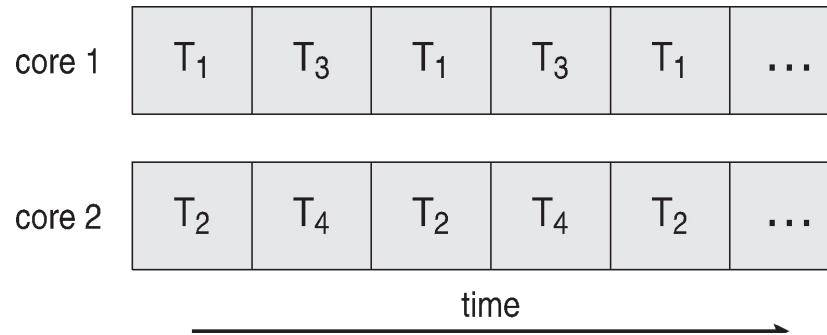


Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**

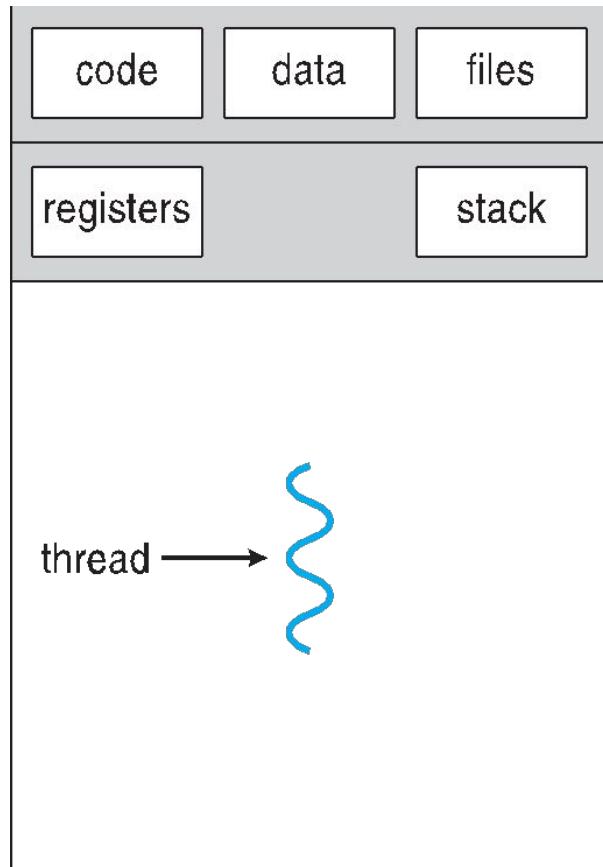


- **Parallelism on a multi-core system:**

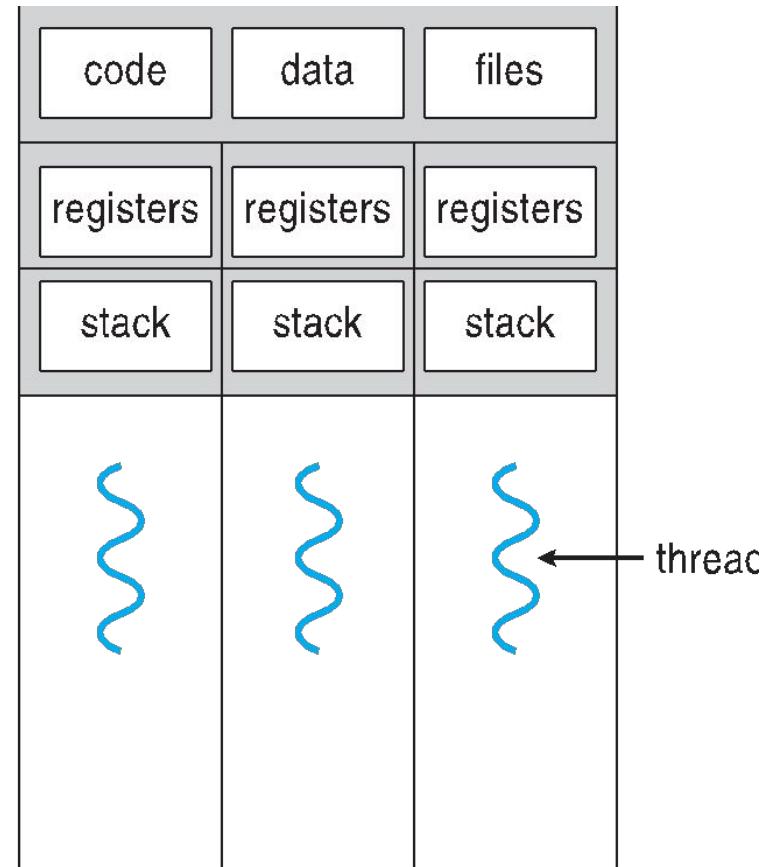




Single and Multithreaded Processes

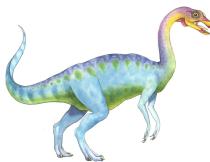


single-threaded process



multithreaded process

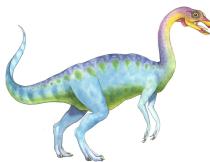




User Threads and Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
 - POSIX **Pthreads**
 - Windows threads
 - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
 - Windows
 - Solaris
 - Linux
 - Tru64 UNIX
 - Mac OS X





Multithreading Models

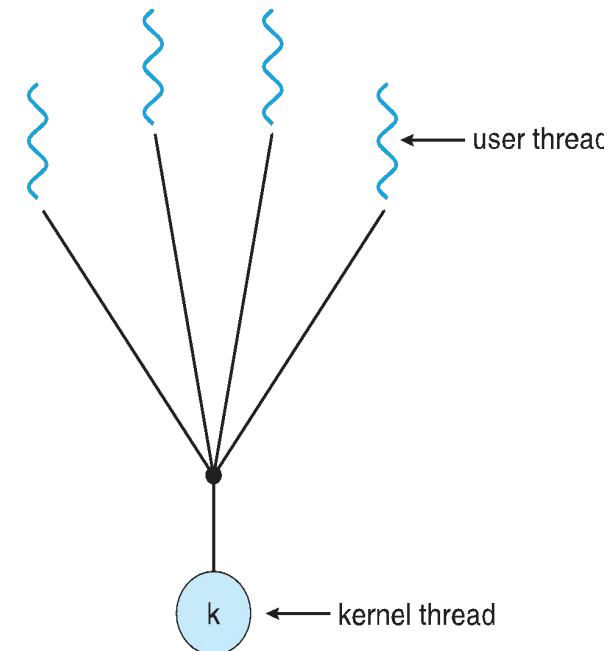
- Many-to-One
- One-to-One
- Many-to-Many

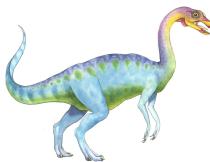




Many-to-One

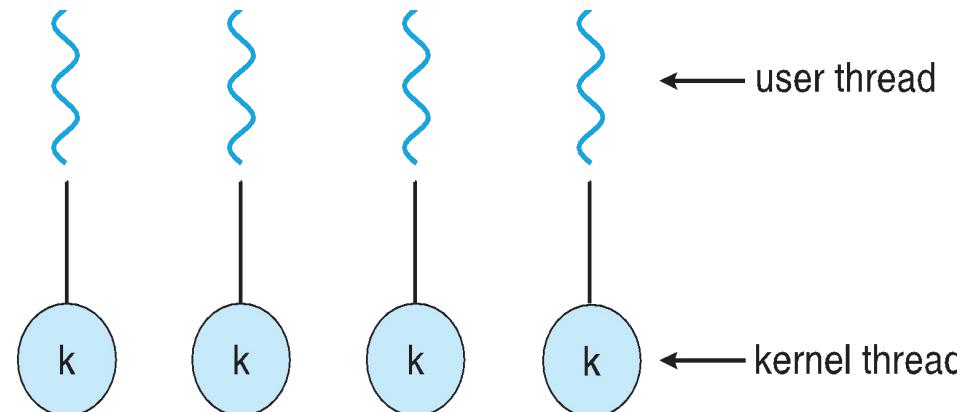
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
 - Solaris Green Threads
 - GNU Portable Threads

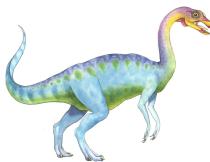




One-to-One

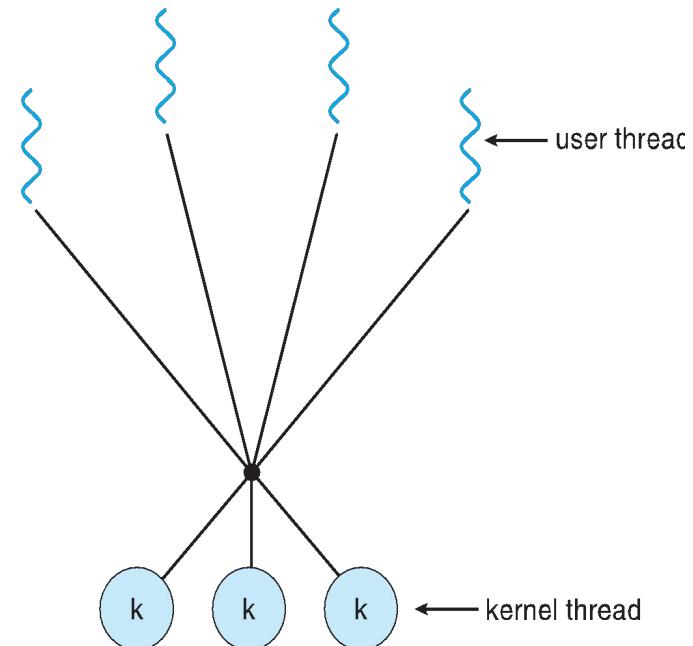
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
 - Windows
 - Linux
 - Solaris 9 and later





Many-to-Many Model

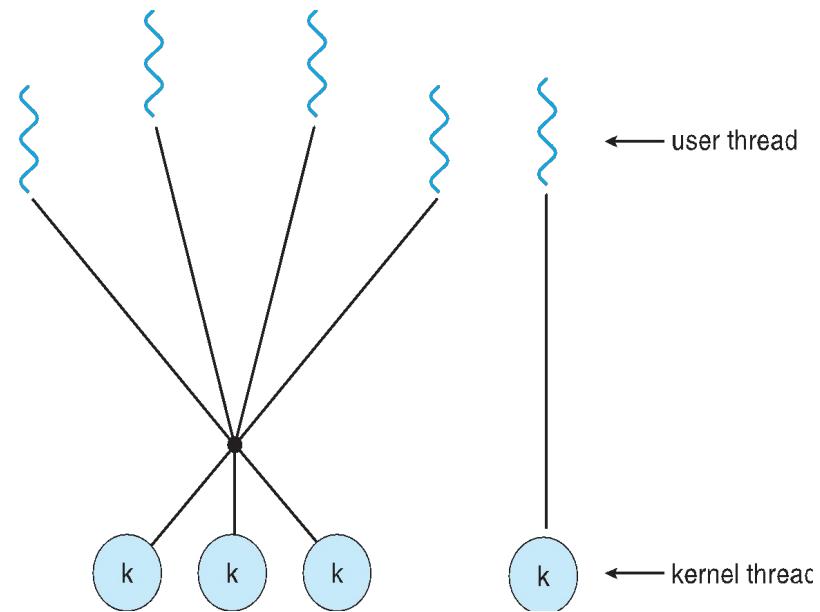
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package





Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

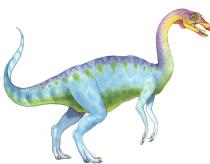




Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
 - Library entirely in user space
 - Kernel-level library supported by the OS





PThreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification, not implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)





PThreads Example (1)

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
}
```





Pthreads Example (2)

```
/* get the default attributes */
pthread_attr_init(&attr);
/* create the thread */
pthread_create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread_join(tid,NULL);

printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```





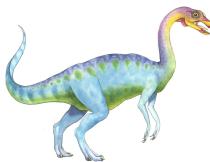
Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```





Windows Multithreaded C Program (1)

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }
}
```





Windows Multithreaded C Program (2)

```
/* create the thread */
ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

if (ThreadHandle != NULL) {
    /* now wait for the thread to finish */
    WaitForSingleObject(ThreadHandle, INFINITE);

    /* close the thread handle */
    CloseHandle(ThreadHandle);

    printf("sum = %d\n", Sum);
}

}
```





Java Threads

- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

- Extending Thread class
- Implementing the Runnable interface





Java Multithreaded Program (1)

```
class Sum
{
    private int sum;

    public int getSum() {
        return sum;
    }

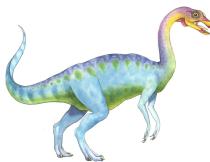
    public void setSum(int sum) {
        this.sum = sum;
    }
}

class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;
        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```





Java Multithreaded Program (2)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                System.err.println(args[0] + " must be >= 0.");
            else {
                Sum sumObject = new Sum();
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Summation(upper, sumObject));
                thrd.start();
                try {
                    thrd.join();
                    System.out.println
                        ("The sum of "+upper+" is "+sumObject.getSum());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Usage: Summation <integer value>"); }
    }
```

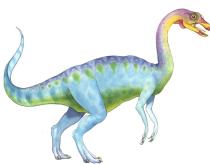




Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
 - **Thread Pools**
 - OpenMP
 - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package



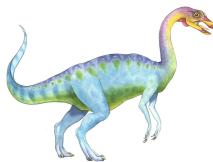


Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
 - Usually slightly faster to service a request with an existing thread than create a new thread
 - Allows the number of threads in the application(s) to be bound to the size of the pool
 - Separating task to be performed from mechanics of creating task allows different strategies for running task
 - 4 i.e. Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(VOID Param) {  
    /*  
     * this function runs as a separate thread.  
     */  
}
```

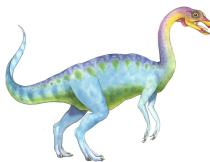




Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Signal handling
 - Synchronous and asynchronous
- Thread cancellation of target thread
 - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations





Semantics of fork() and exec()

- Does `fork()` duplicate only the calling thread or all threads?
 - Some UNIXes have two versions of fork
- `exec()` usually works as normal – replace the running process including all threads

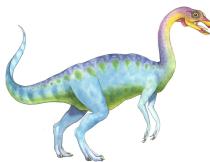




Signal Handling (1)

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
 1. Signal is generated by particular event
 2. Signal is delivered to a process
 3. Signal is handled by one of two signal handlers:
 1. default
 2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
 - **User-defined signal handler** can override default
 - For single-threaded, signal delivered to process

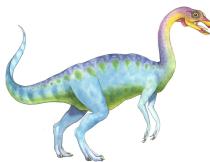




Signal Handling (2)

- Where should a signal be delivered for multi-threaded?
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process



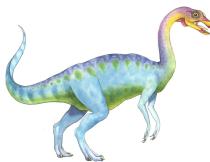


Thread Cancellation (1)

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
 - **Asynchronous cancellation** terminates the target thread immediately
 - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;  
  
/* create the thread */  
pthread_create(&tid, 0, worker, NULL);  
  
.  
.  
.  
  
/* cancel the thread */  
pthread_cancel(tid);
```





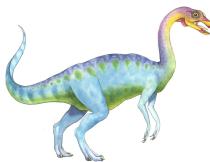
Thread Cancellation (2)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

| Mode | State | Type |
|--------------|----------|--------------|
| Off | Disabled | - |
| Deferred | Enabled | Deferred |
| Asynchronous | Enabled | Asynchronous |

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
 - Cancellation only occurs when thread reaches **cancellation point**
 - 4 I.e. `pthread_testcancel()`
 - 4 Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals





Thread-Local Storage

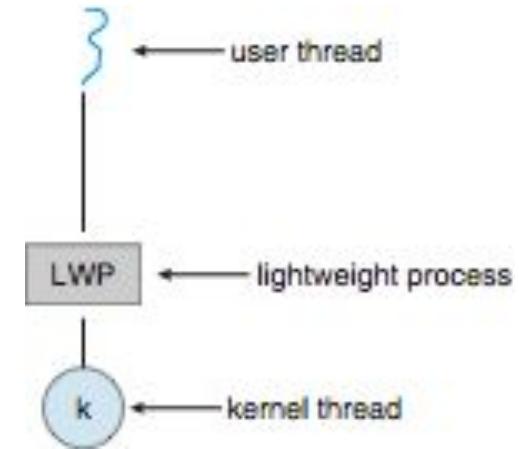
- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
 - Local variables visible only during single function invocation
 - TLS visible across function invocations
- Similar to `static` data
 - TLS is unique to each thread





Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process (LWP)**
 - Appears to be a virtual processor on which process can schedule user thread to run
 - Each LWP attached to a kernel thread
 - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number kernel threads





Operating System Examples

- Windows Threads
- Linux Threads

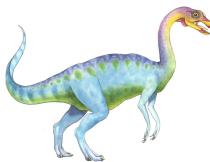




Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7
- Implements the one-to-one mapping, kernel-level
- Each thread contains
 - A thread id
 - Register set representing state of processor
 - Separate user and kernel stacks for when thread runs in user mode or kernel mode
 - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- The register set, stacks, and private storage area are known as the **context** of the thread

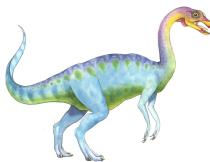




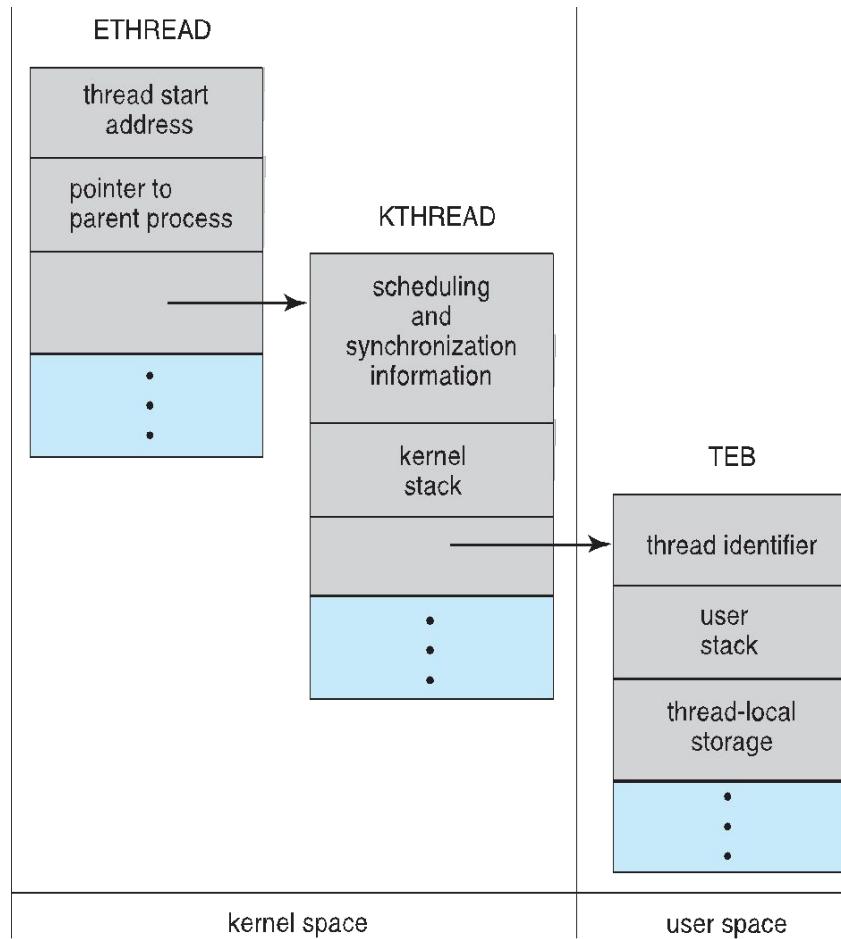
Windows Threads (Cont.)

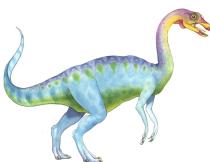
- The primary data structures of a thread include:
 - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
 - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
 - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space





Windows Threads Data Structures





Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
 - Flags control behavior

| flag | meaning |
|----------------------------|------------------------------------|
| <code>CLONE_FS</code> | File-system information is shared. |
| <code>CLONE_VM</code> | The same memory space is shared. |
| <code>CLONE_SIGHAND</code> | Signal handlers are shared. |
| <code>CLONE_FILES</code> | The set of open files is shared. |

- **`struct task_struct`** points to process data structures (shared or unique)



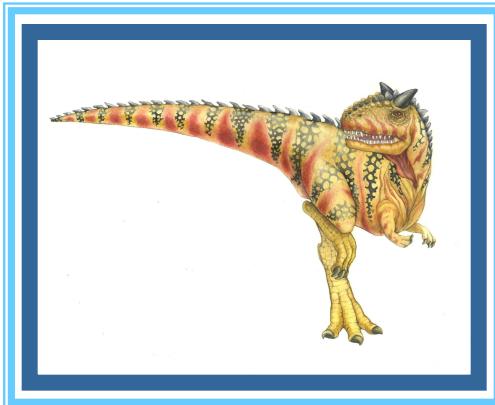


Homework

- Ex1: Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.
- Ex2: Write a multithreaded Java, or Pthreads, Win32 program that outputs prime numbers. This program should work as follows: The user will run the program and will enter a number on the command line. The program will then create a separate thread that outputs all the prime numbers less than or equal to the number entered by the user.

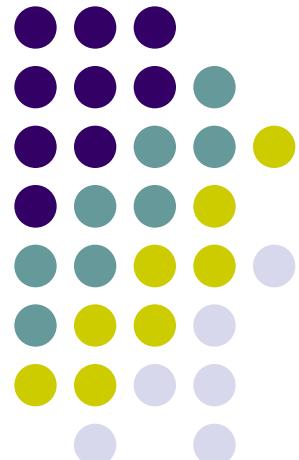


End of Chapter 4

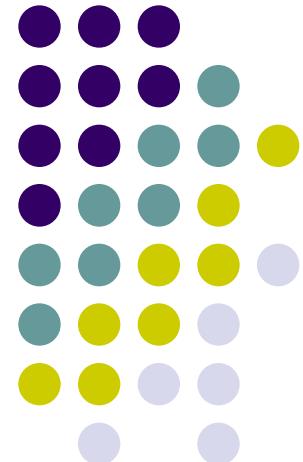


Nguyên lý hệ điều hành

Lê Đức Trọng
DS&KTLab, Khoa CNTT
Trường Đại học Công nghệ
(Slide credit: PGS. TS. Nguyễn Hải Châu)



Lập lịch CPU



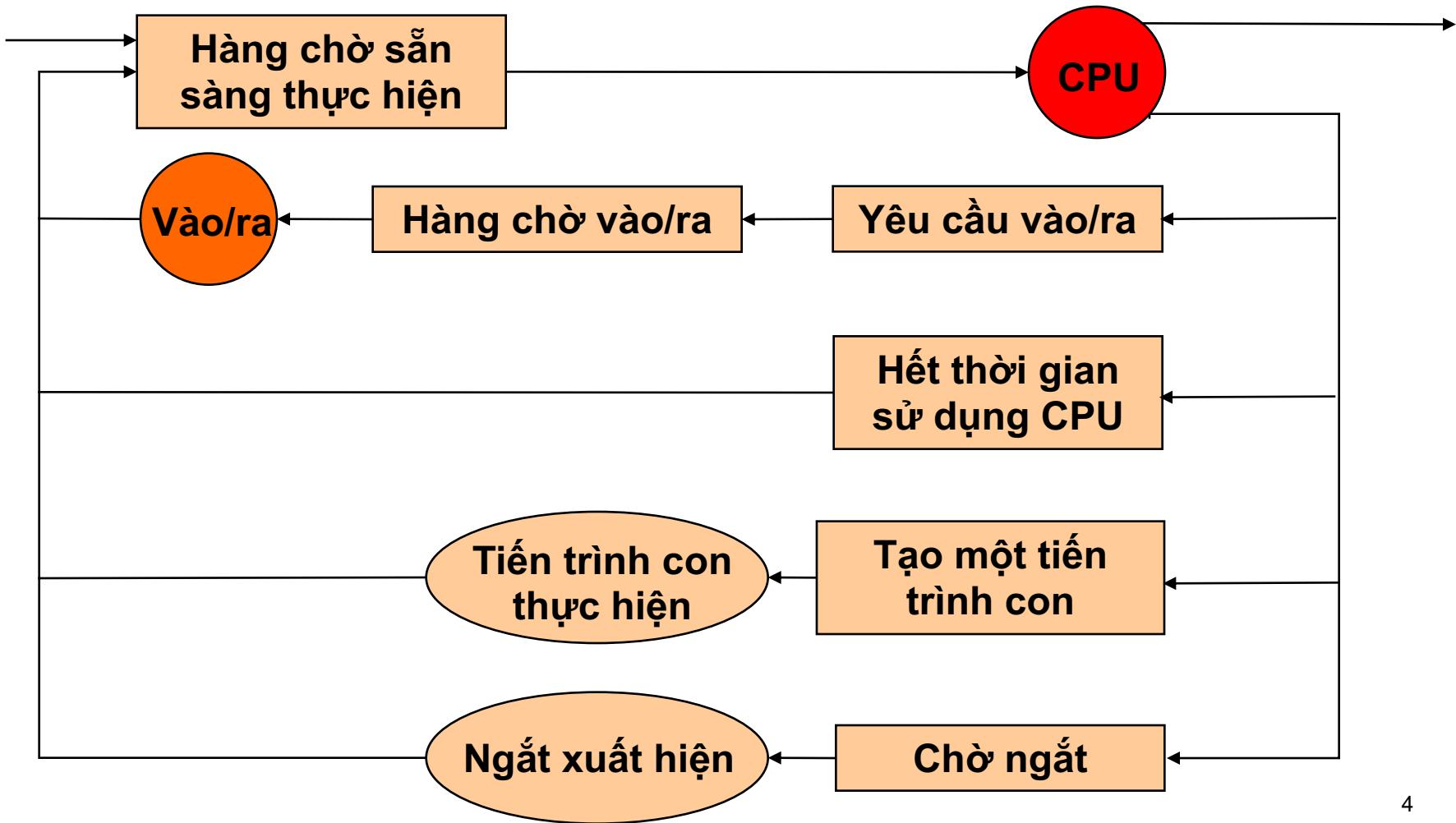


Tại sao phải lập lịch CPU?

- 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



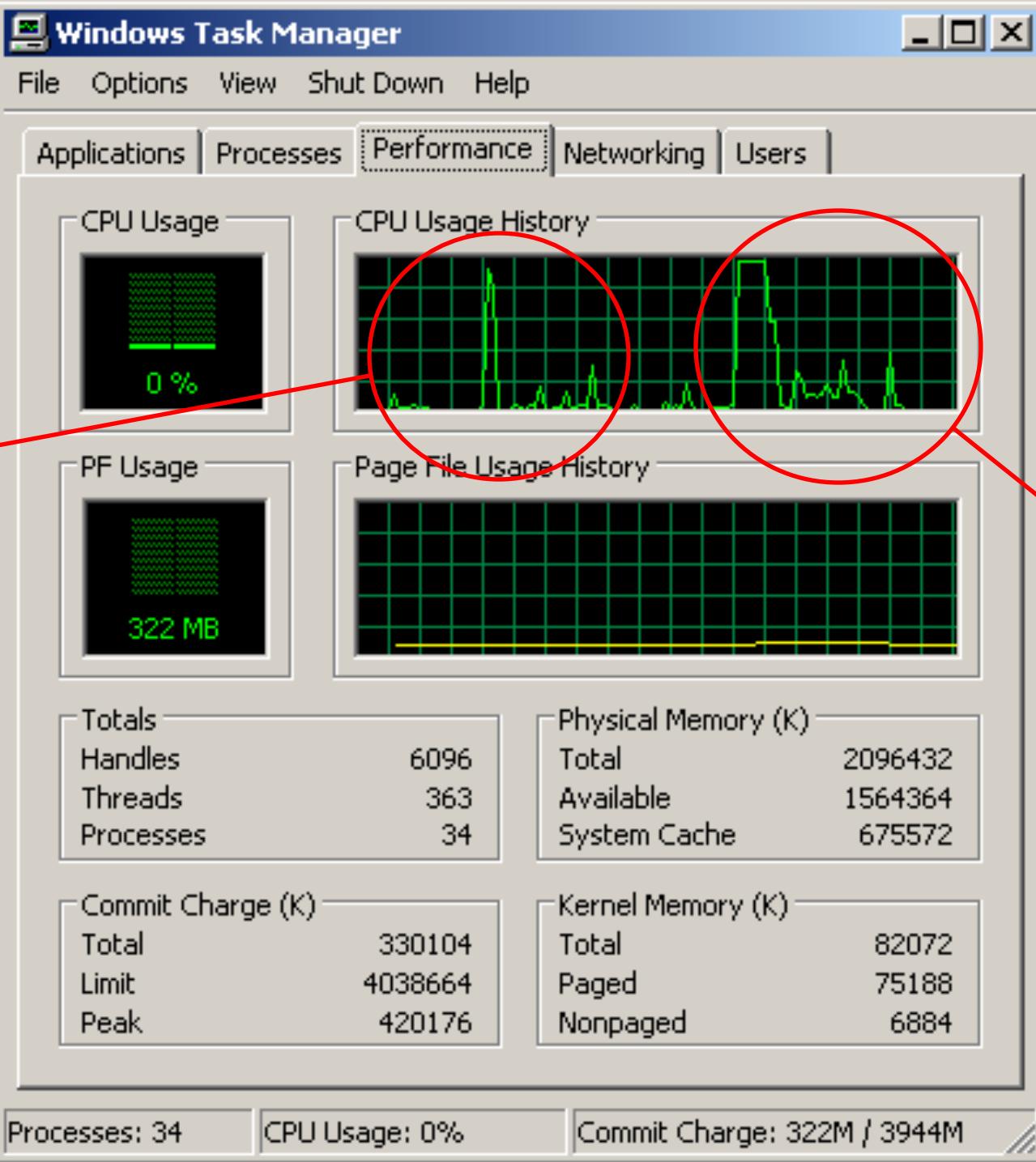
Hàng chờ lập lịch tiến trình





CPU-burst và I/O-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*)





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)

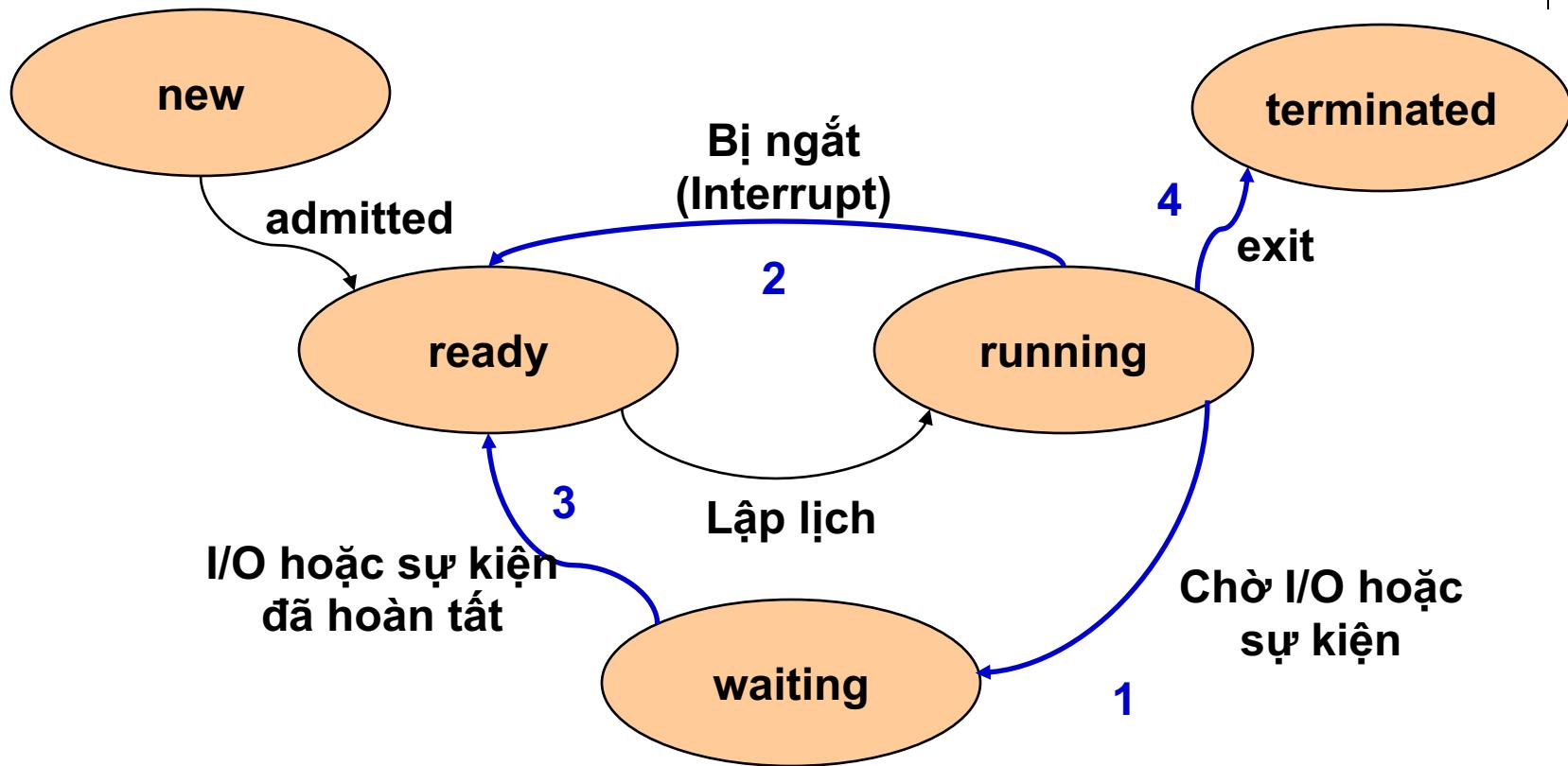


Bộ lập lịch ra hoạt động khi...

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



Các phương pháp lập lịch

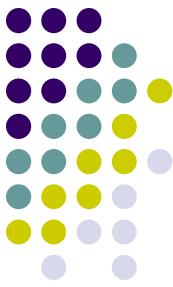


- 1 và 4: Lập lịch non-preemptive
- Ngược lại: Lập lịch preemptive



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



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



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



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



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...



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



Các tiêu chí đánh giá lập lịch

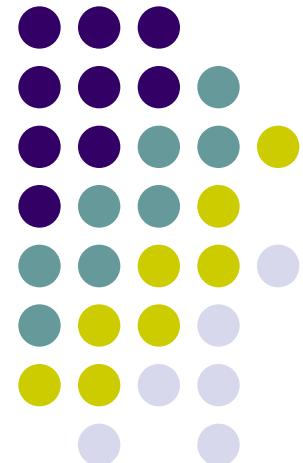
- Thời gian đáp ứng (*response time*): Là khoảng thời gian từ khi tiến trình nhận được một yêu cầu cho đến khi *bắt đầu đáp ứng* yêu cầu đó
- $t_{res} = t_r - t_s$, trong đó t_r là thời điểm nhận yêu cầu, t_s là thời điểm bắt đầu đáp ứng yêu cầu



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 |

Các thuật toán lập lịch





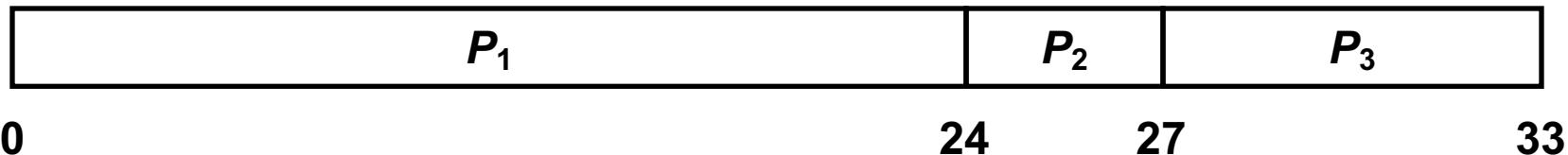
FCFS (First Come First Served)

- 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)*



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: $(0+24+27)/3=17\text{ms}$



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_1 chờ 9ms
- Thời gian chờ trung bình: $(0+6+9)/3=5\text{ms}$
- Thời gian chờ trung bình thấp hơn thời gian chờ trung bình trong ví dụ 1a

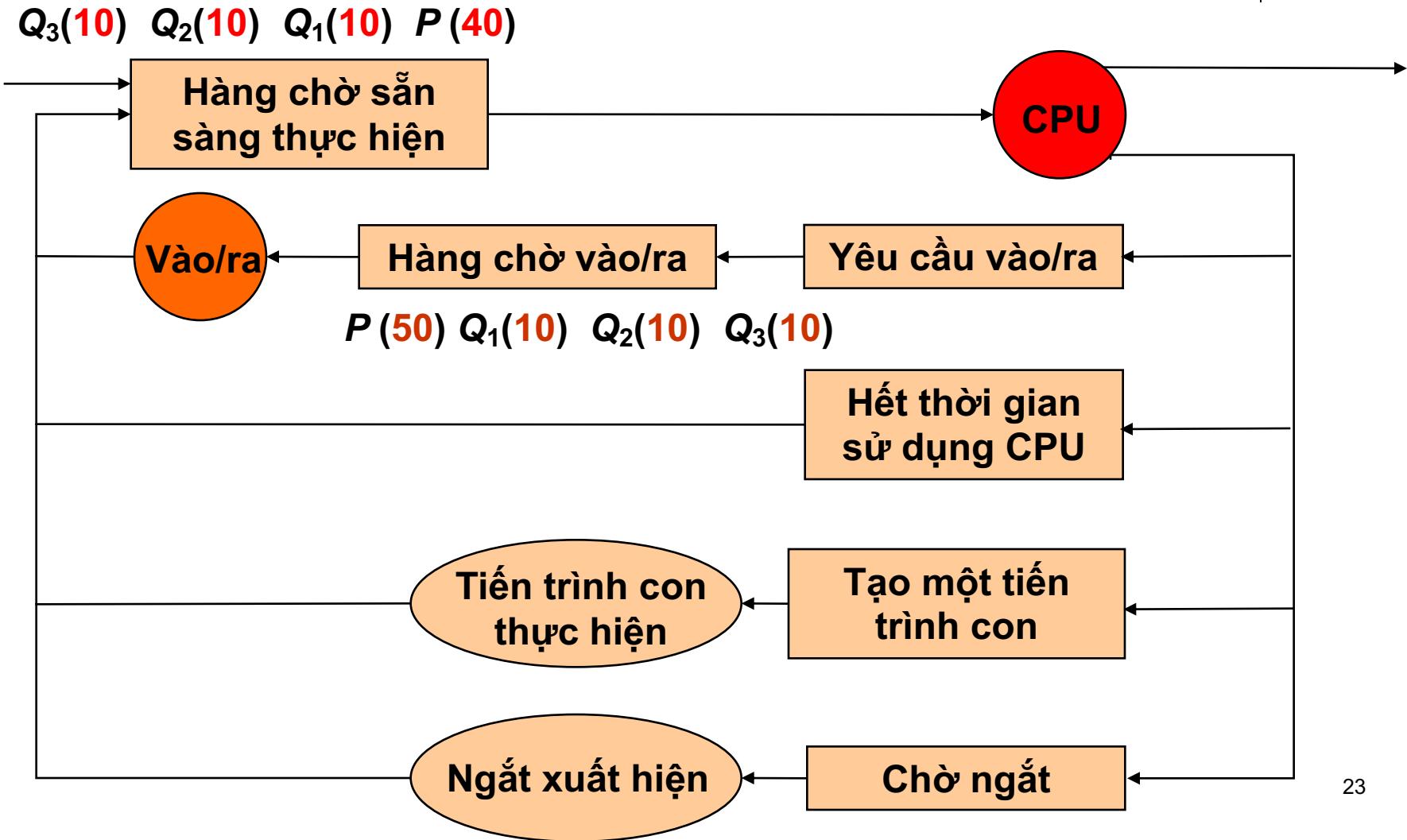


Hiện tượng “đoàn hộ tống”

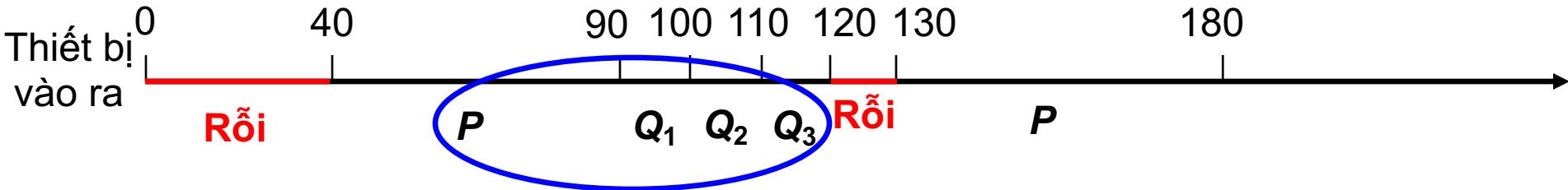
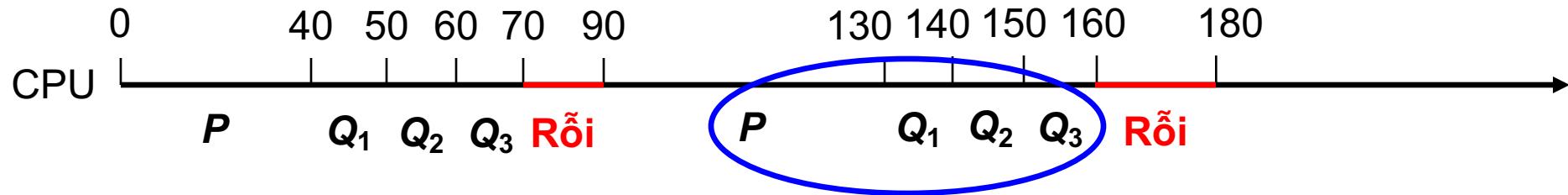
- Thuật ngữ: *convoy effect*
- Xảy ra khi có một tiến trình “lớn” P nằm ở đầu hàng chờ và nhiều tiến trình “nhỏ” Q_i xếp hàng sau P .
- “Lớn”: Sử dụng nhiều thời gian CPU và vào ra
- “Nhỏ”: Sử dụng ít thời gian CPU và vào ra
- Thuật toán lập lịch được sử dụng là FCFS..
- Hiện tượng xảy ra: CPU, thiết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



Ví dụ convoy effect



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



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



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 $t_{nextburst}$ 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



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



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



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



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 $(1+6+16+18)/5=8.2\text{ms}$

| 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 |





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



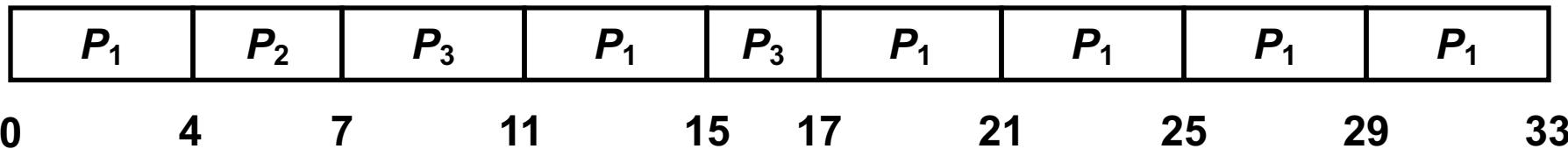
Round-robin (RR)

- Hiệu quả của RR phụ thuộc độ lớn của $t_{quantum}$
 - Nếu $t_{quantum}$ 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 $t_{quantum} = \infty$ thì RR trở thành FCFS



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 $t_{quantum} = 4\text{ms}$
- RR lập lịch các tiến trình như sau:



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



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



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



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

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

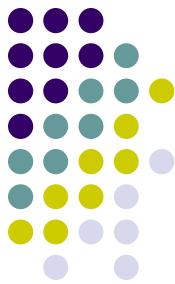
Lập lịch với hàng chờ đa mức có phản hồi



- 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

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



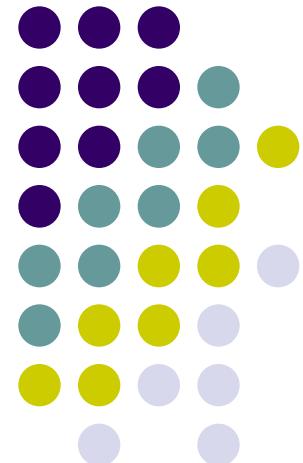
- Số lượng các hàng chờ
- Thuật toán lập lịch cho mỗi hàng chờ
- Phương pháp tăng độ ưu tiên cho một tiến trình
- Phương pháp giảm độ ưu tiên cho một tiến trình
- Phương pháp xác định hàng đợi nào để đưa một tiến trình vào



Các thuật toán lập lịch khác

- Sinh viên tìm hiểu trong giáo trình:
 - Lập lịch đa xử lý (Multi-Processor Scheduling)
 - Lập lịch theo luồng (Thread Scheduling)

Các phương pháp đánh giá thuật toán lập lịch





Mô hình tất 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



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



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



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



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



Bài tập

- Thực hiện ví dụ RR với lượng tử thời gian 2ms, 6ms và 9ms.
 - 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

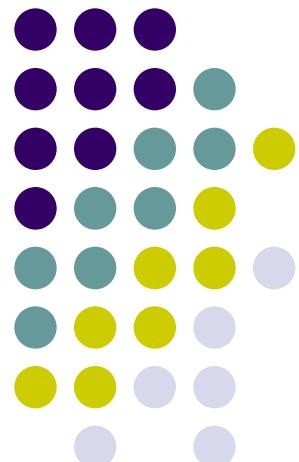


Bài tập

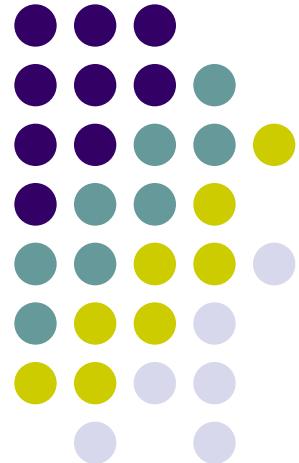
- Hãy xây dựng một ví dụ về hiện tượng convoy effect sao cho thời gian rỗi của thiết bị vào ra ít hơn thời gian rỗi của CPU. Giả sử có 4 tiến trình, trong đó P là tiến trình “lớn”, Q_1 , Q_2 , Q_3 là các tiến trình “nhỏ” được nằm trong hàng chờ theo thứ tự: P , Q_1 , Q_2 , Q_3
- Tìm hai ví dụ trong thực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

Nguyên lý hệ điều hành

Lê Đức Trọng
DS&KTLAB, Khoa CNTT
Trường Đại học Công nghệ
(Slide credit: PGS. TS. Nguyễn Hải Châu)



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



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, \dots, 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)



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 .



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);



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);



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 $P_{j1}, P_{j2}, \dots, P_{jm}$ muốn thực hiện $CS_{j1}, CS_{j2}, \dots, CS_{jm}$ thì chỉ có các tiến trình đang không thực hiện $REMAIN_{jk}$ ($k=1, \dots, m$) 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.



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 $\text{flag}[i]$ là TRUE thì tiến trình P_i đã sẵn sàng để thực hiện CS_i



Ví dụ: giải pháp của Peterson

- Mã lệnh của P_i :

```
do {
```

```
    flag[i] = TRUE;
```

```
    turn = j;
```

```
    while (flag[j] && turn == j) ;
```

```
    CSi;
```

```
    flag[i] = FALSE;
```

```
    REMAINi;
```

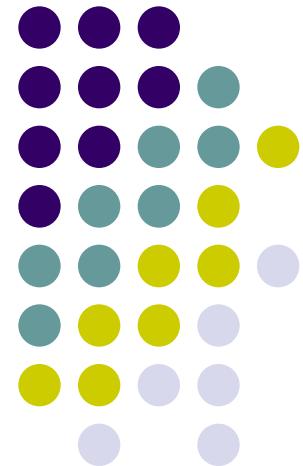
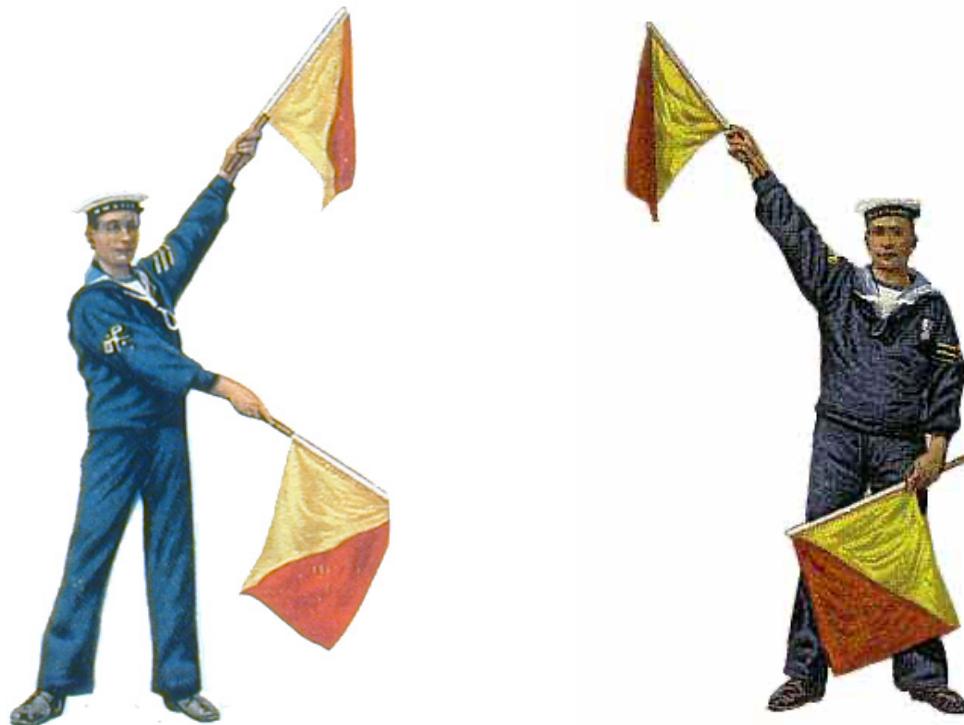
```
} while (1);
```



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

Semaphore





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)



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



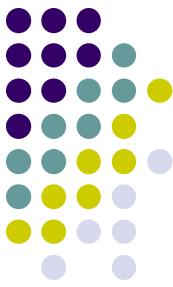
Toán tử wait và signal

```
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



Sử dụng semaphore (1)

- Với bài toán đoạn mã gắp:

```
do {
```

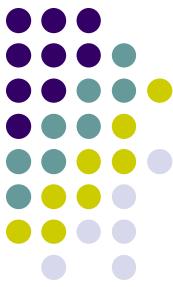
```
    wait(mutex); // mutex là semaphore khởi tạo 1
```

```
    CSi;
```

```
    signal(mutex);
```

```
    REMAINi;
```

```
} while (1);
```



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 :

...

$O_1;$

signal(synch);

...

• P_2 :

...

wait(synch);

$O_2;$

...



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*



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;
```



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();
    }
}
```

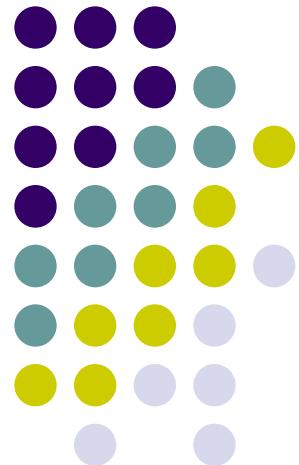
```
void signal(semaphore *S)
{
    S->value++;
    if (S->value<=0) {
        Xóa một tiến trình  $P$ 
        ra khỏi s->L;
        wakeup( $P$ );
    }
}
```



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)

Một số bài toán đồng bộ hóa cơ bản





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)



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);
```



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)



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 trừ lẫn nhau khi cập nhật biến *rcount*



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);
```

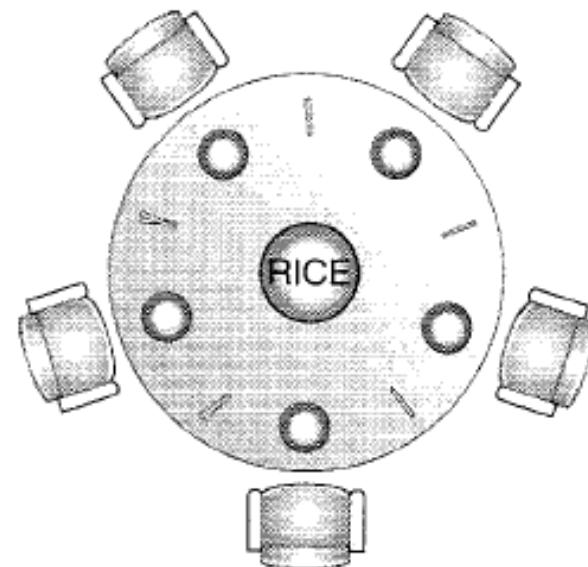
- **Tiến trình reader P_r :**

```
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);
```



Bữa ăn tối của các triết gia

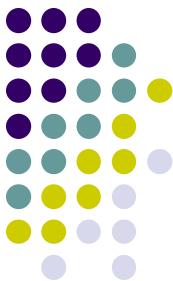
- Thuật ngữ: the dining-philosophers problem
- Có 5 triết gia, 5 chiếc đũa, 5 bát cơm và một âu cơm bố trí như hình vẽ
- Đây là bài toán cổ điển và là ví dụ minh họa cho một lớp nhiều bài toán tương tranh (concurrency): *Nhiều tiến trình khai thác nhiều tài nguyên chung*





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
 - Đè 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



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);



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



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*



Ví dụ hạn chế của semaphore (1)

- Xét ví dụ về đoạn mã găng:

- Mã đúng:

...

```
wait(mutex);  
// Đoạn mã găng  
signal(mutex);
```

...

- Mã sai:

...

```
signal(mutex);  
// Đoạn mã găng  
wait(mutex);
```

...

- Đoạn mã sai này gây ra vi phạm điều kiện loại trừ lẫn nhau



Ví dụ hạn chế của semaphore (2)

- Xét ví dụ về đoạn mã găng:

- Mã đúng:

...

```
wait(mutex);  
// Đoạn mã găng  
signal(mutex);
```

...

- Mã sai:

...

```
wait(mutex);  
// Đoạn mã găng  
wait(mutex);
```

...

- Đoạn mã sai này gây ra bế tắc



Ví dụ hạn chế của semaphore (3)

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



Ví dụ hạn chế của semaphore (4)

- Tiến trình P_1

...

wait(S);

wait(Q);

...

signal(S);

signal(Q);

- Tiến trình P_2

...

wait(Q);

wait(S);

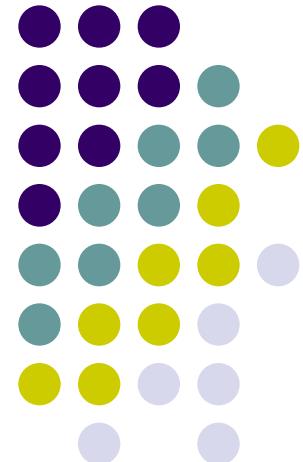
...

signal(Q);

signal(S);

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

Cơ chế monitor





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)



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



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 tranh
- Trong bài này: Nghiên cứu một loại cấu trúc monitor: Kiểu monitor (monitor type)



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

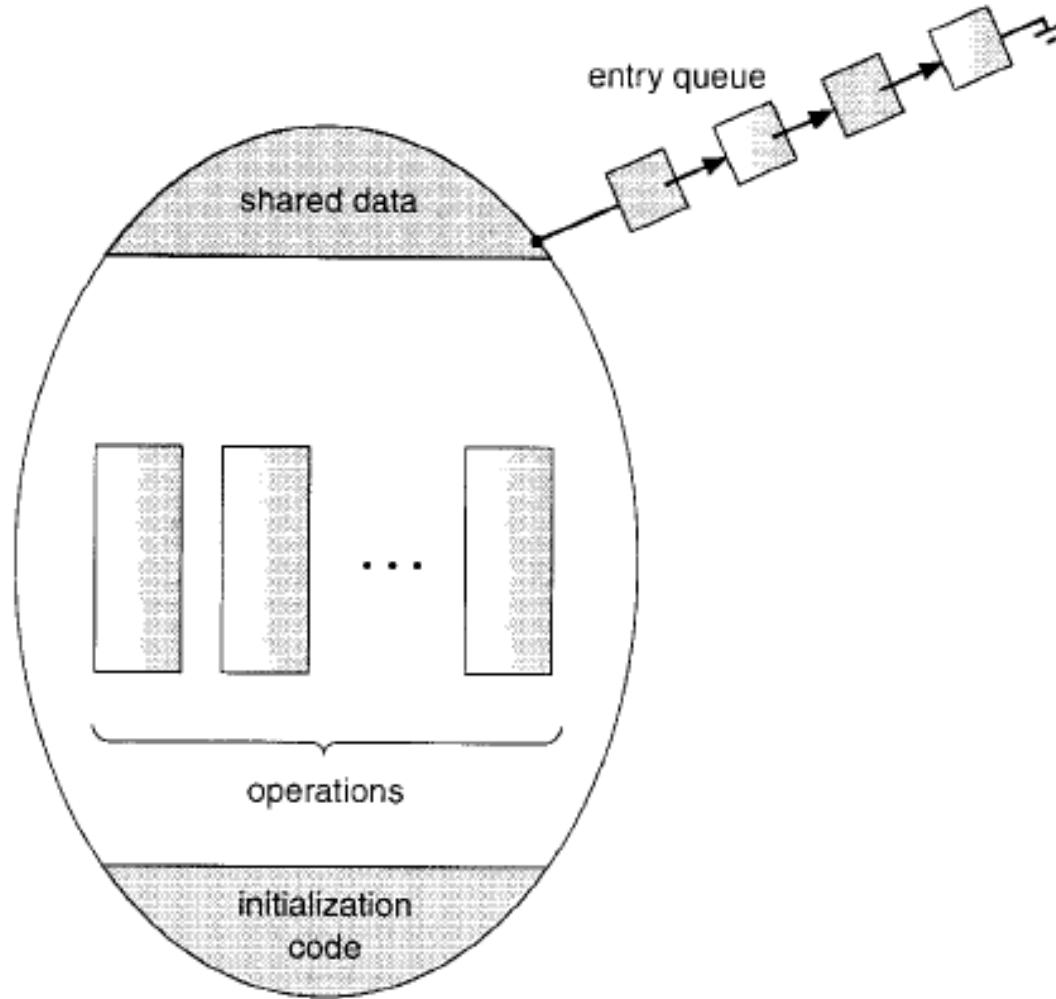


Cấu trúc một monitor type

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



Minh họa cấu trúc monitor





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*.

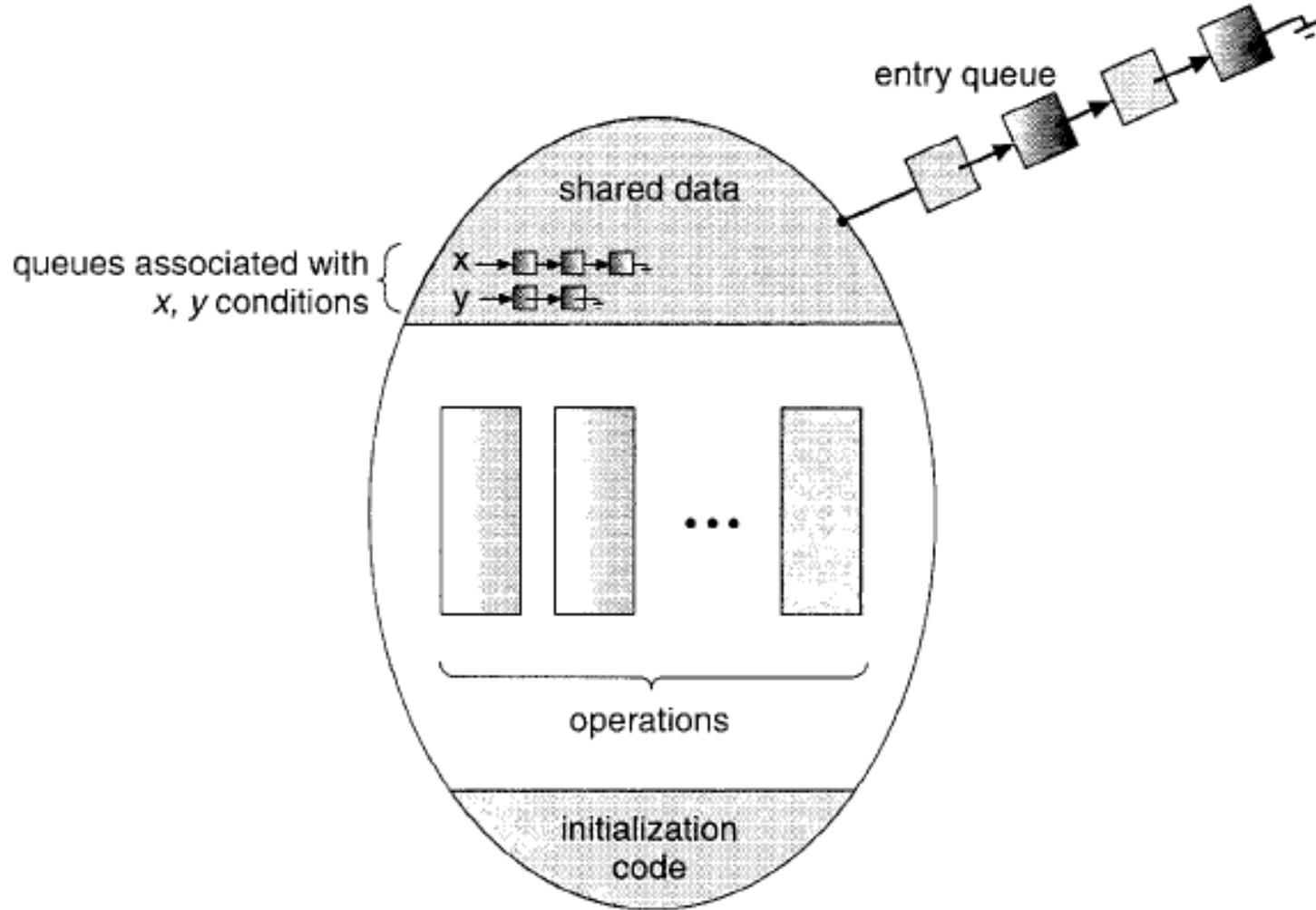


Kiểu condition

- 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()



Monitor có kiểu condition





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



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



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:
 $(state[(i+4)\%5] \neq \text{eating}) \text{ and } (state[(i+1)\%5] \neq \text{eating})$
- Khi triết gia i không đủ điều kiện để ăn: cần có biến condition: condition self[5];



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();  
    }  
}
```



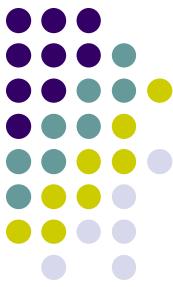
Monitor của bài toán Ăn tối...

```
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;  
}
```



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();  
    }  
}
```



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



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



Bài tập

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

do {

 while (turn != i) ;

 CS_i;

 turn = j;

 REMAIN_i;

} while (1);

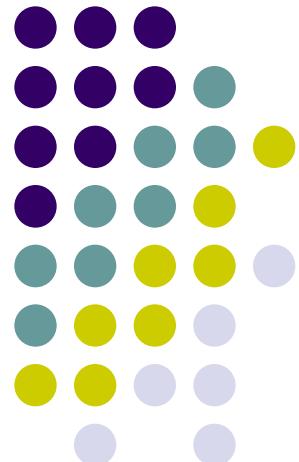


Bài tập

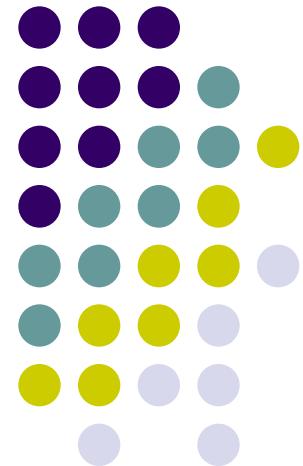
- Cài đặt giải pháp cho bài toán Bữa ăn tối của các triết gia trong Java bằng cách sử dụng synchronized, wait() và notify()
- Giải pháp monitor cho bài toán Bữa ăn tối... tránh được bế tắc, nhưng có thể xảy ra trường hợp tất cả các triết gia đều không được ăn. Hãy chỉ ra trường hợp này và tìm cách giải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

Nguyên lý hệ điều hành

Lê Đức Trọng
DS&KTLab, Khoa CNTT
Trường Đại học Công nghệ
(Slide credit: PGS.TS. Nguyễn Hải Châu)



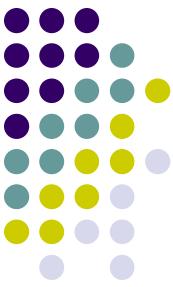
Bế tắc (Deadlock)



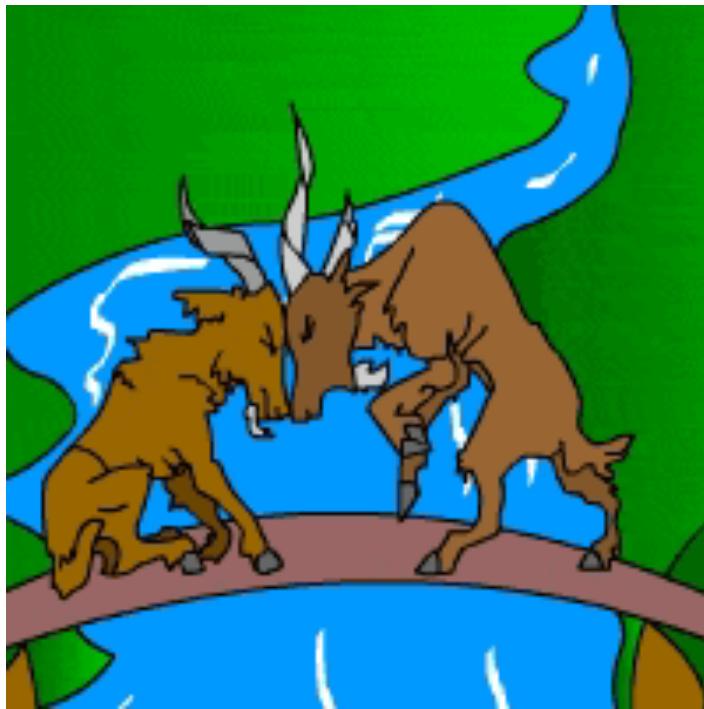


Định nghĩa

- 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)

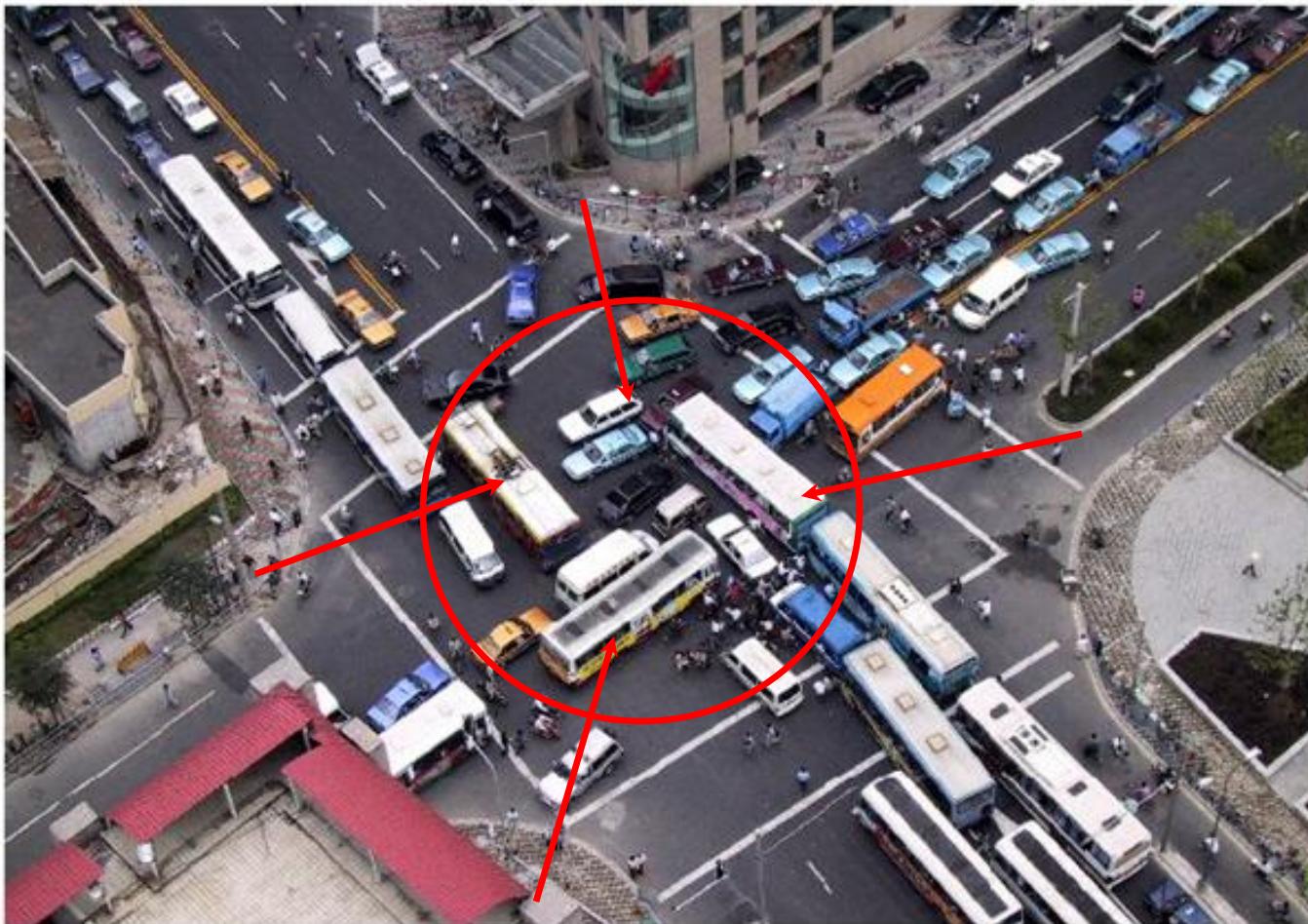


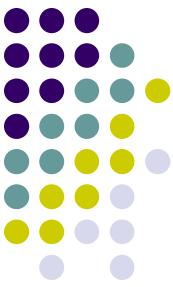
Hai con dê qua cầu: Bế tắc





Bế tắc giao thông tại ngã tư





Bế tắc trong máy tính

- Tiến trình A:

{

...

Khóa file F_1 ;

...

Mở file F_2 ;

...

Đóng F_1 (mở khóa F_1);

}

- Tiến trình B

{

...

Khóa file F_2 ;

...

Mở file F_1 ;

...

Đóng F_1 (mở khóa F_1);

}



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)



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)



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



C2: Giữ và chờ

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



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.



C4: Chờ vòng

- Một tập tiến trình $\{P_0, P_1, \dots, P_n\}$ 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



Đồ thị cấp phát tài nguyên

- Thuật ngữ: Resource allocation graph
- Để mô tả một cách chính xác bối cảnh, 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
- V được chia thành hai tập con $P=\{P_0, P_1, \dots, P_n\}$ là tập các tiến trình trong hệ thống và $R=\{R_0, R_1, \dots, R_m\}$ là tập các loại tài nguyên trong hệ thống thỏa mãn $P \cup R = V$ và $P \cap R = \emptyset$



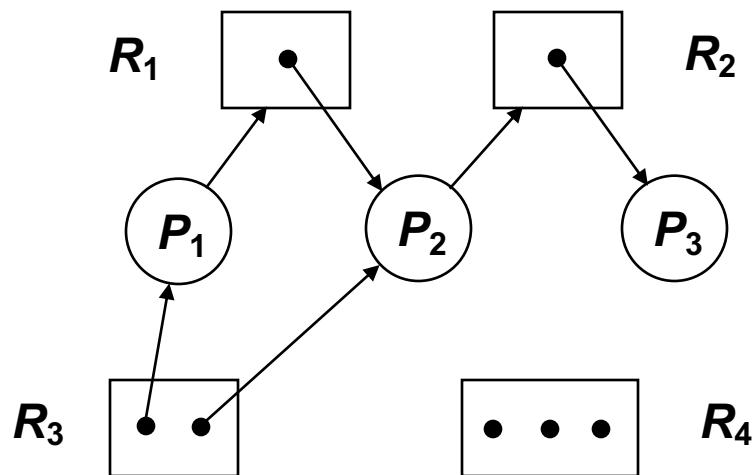
Đồ thị cấp phát tài nguyên

- 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 R_j đã được cấp phát cho tiến trình P_i . Ta gọi $R_j \rightarrow P_i$ là *cung cấp phát* (*asignment edge*)



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:





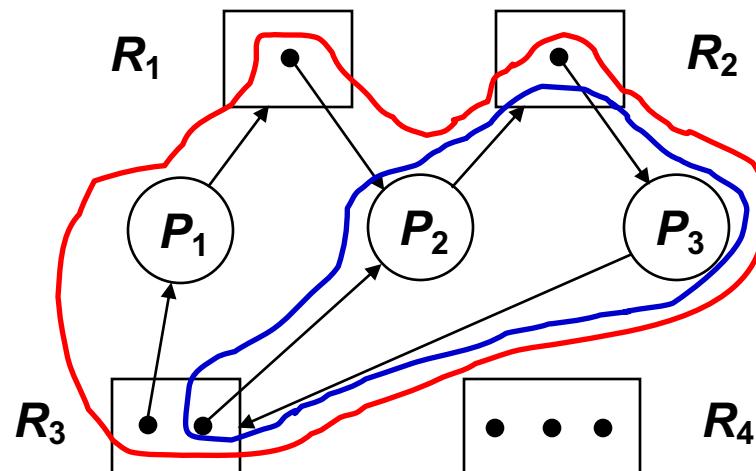
Đồ thị cấp phát tài nguyên

- 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)



Ví dụ chu trình dẫn đến bế tắc

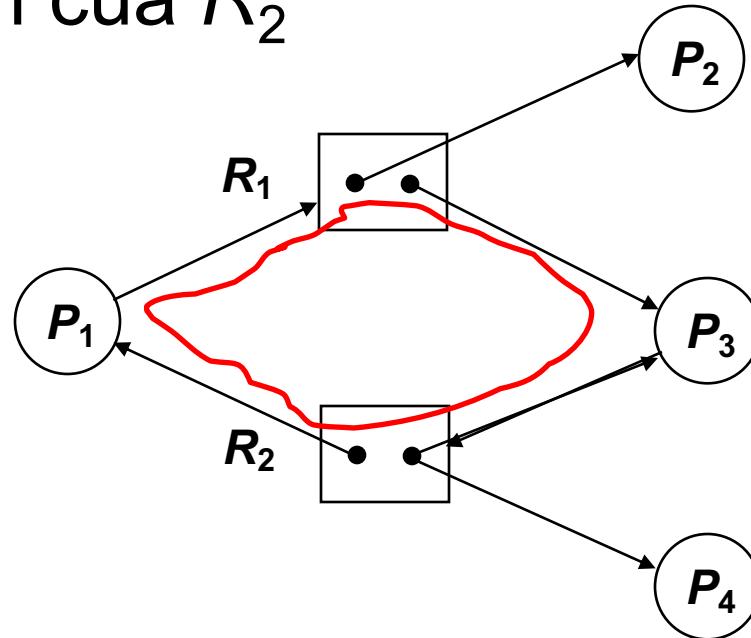
- Giả sử P_3 yêu cầu một thẻ hiện của R_3
- Khi đó có 2 chu trình xuất hiện:
 - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_1$, và
 - $P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow R_3 \rightarrow P_2$
- Khi đó các tiến trình P_1, P_2, P_3 bị bế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



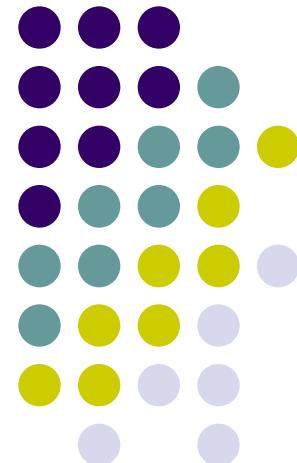
Ví dụ chu trình không dẫn đến bế tắc



- 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 thẻ hiện tài nguyên R_2 và P_3 sẽ được cấp phát một thẻ hiện của R_2



Các phương pháp xử lý bế tắc

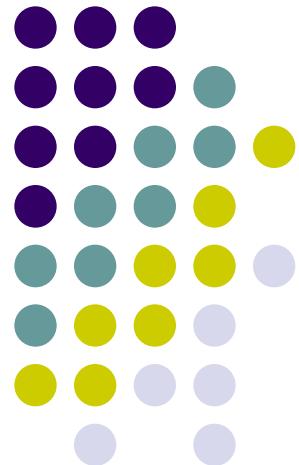




Các phương pháp xử lý bế tắc

- 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”

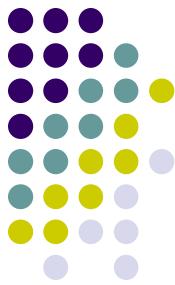
- C2 (Giữ và chờ): Có thể làm cho C2 không xảy ra bằng cách đảm bảo:
 - Một tiến trình luôn yêu cầu cấp phát tài nguyên chỉ khi nó không chiếm giữ bất kỳ một tài nguyên nào, hoặc
 - Một tiến trình chỉ thực hiện khi nó được cấp phát toàn bộ các tài nguyên cần thiết



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, \dots, 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, \dots, R_{n-1} phải được “thu hồi”
 - Nói cách khác, R_1, \dots, 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.

Ngăn chặn “không có đặc quyền”: mã lệnh



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

if (R_1, \dots, R_{n-1} rõ)

then cấp phát tài nguyên cho P

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

then thu hồi $\{R_i \dots R_j\}$ và cấp phát cho P

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



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 $\{R_1, \dots, R_n\}$. 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: $f(\text{ổ cứng}) = 1$, $f(\text{băng từ}) = 5$, $f(\text{máy in}) = 11$



Ngăn chặn “chờ vòng”

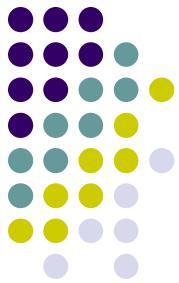
- Giao thức ngăn chặn chờ vòng:
 - Khi tiến trình P không chiếm giữ tài nguyên nào, nó có thể yêu cầu cấp phát nhiều *thể hiện* của *một* tài nguyên R_i bất kỳ
 - Sau đó P chỉ có thể yêu cầu các thể hiện của tài nguyên R_j nếu và chỉ nếu $f(R_j) > f(R_i)$. Một cách khác, nếu P muốn yêu cầu cấp phát tài 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

Chứng minh giải pháp ngăn chặn chờ vòng



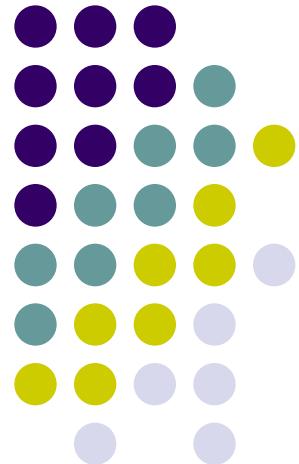
- Sử dụng chứng minh phản chứng
- Giả sử giải pháp ngăn chặn gây ra chờ vòng $\{P_0, P_1, \dots, P_n\}$ trong đó P_i chờ tài nguyên R_i bị chiếm giữ bởi $P_{(i+1) \bmod n}$
- Vì P_{i+1} đang chiếm giữ R_i và yêu cầu R_{i+1} , do đó $f(R_i) < f(R_{(i+1) \bmod n}) \quad \forall i$, có nghĩa là ta có:
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

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

Tránh bế tắc (Deadlock avoidance)





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



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*



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)



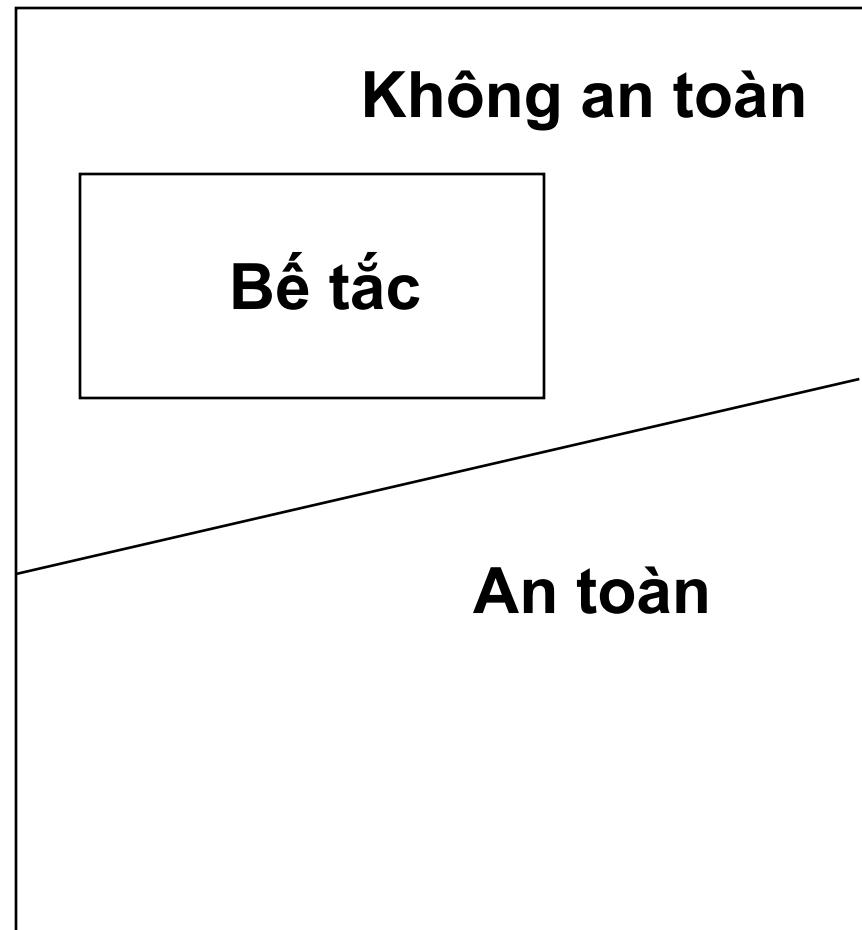
Thứ tự an toàn

- Thứ tự các tiến trình $\langle P_1, \dots, P_n \rangle$ 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 $P_j \forall j < i$.

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





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

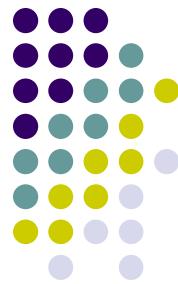


Ví dụ trạng thái an toàn, bế tắc

| | <u>Yêu cầu nhiều nhất</u> | <u>Yêu cầu hiện tại</u> |
|-------|---------------------------|-------------------------|
| 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 $\langle P_1, P_0, P_2 \rangle$ 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 định cấp tài nguyên cho P_2 là sai.

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

Thuật toán đồ thị cấp phát tài nguyên

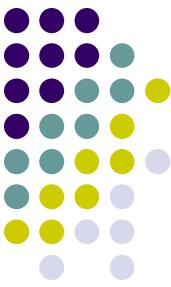


- Chú ý rằng các tài nguyên phải được thông báo trước khi tiến trình thực hiện
- Các cung báo trước sẽ phải có trên đồ thị cấp phát tài nguyên
- Tuy nhiên có thể giảm nhẹ điều kiện: cung thông báo $P_i \rightarrow R_j$ được thêm vào đồ thị nếu tất cả các cung gắn với P_i đều là cung thông báo

Thuật toán đồ thị cấp phát tài nguyên

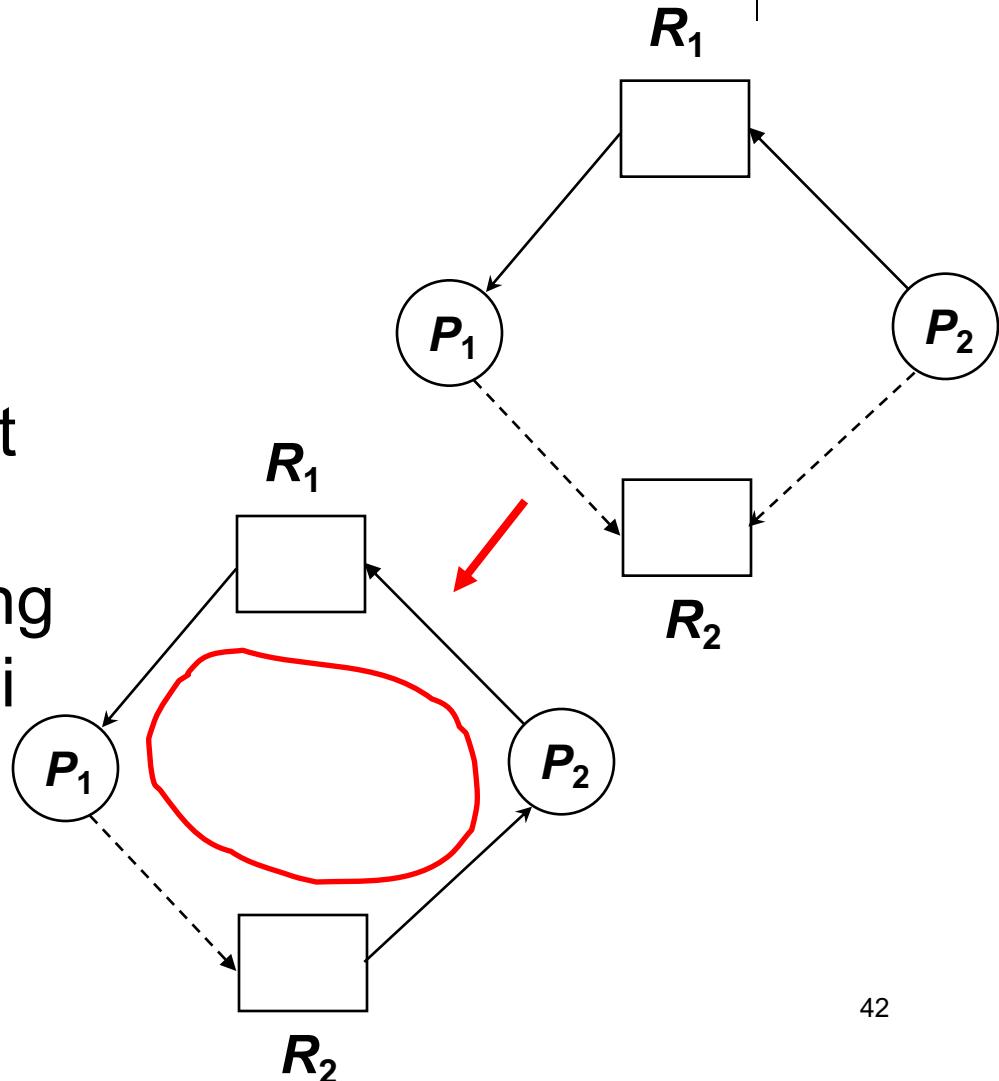


- Giả sử P_i yêu cầu cầu cấp phát R_i . Yêu cầu này chỉ có thể được chấp nhận nếu ta chuyển cung bão trước $P_i \rightarrow R_j$ thành cung cấp phát $R_j \rightarrow P_i$ và không tạo ra một chu trình
- Chúng ta kiểm tra bằng cách sử dụng thuật toán phát hiện chu trình trong đồ thị: Nếu có n tiến trình 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



Ví dụ

- Giả sử P_2 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





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



Ký hiệu dùng trong banker

- Tài nguyên rõi: Vector m thành phần $Available$, $Available[j]=k$ nghĩa là có k thể hiện của R_j rõi
- Max: Ma trận $n \times m$ 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 .



Ký hiệu dùng trong banker

- Cấp phát: Ma trận $n \times m$ 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 $n \times m$ 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 .



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 $X[i] \leq Y[i] \quad \forall i$ thì ta ký hiệu $X \leq Y$.
- 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



Thuật toán trạng thái an toàn

1. Khởi tạo $Work=Available$ và $Finish[i]=\text{false}$ $\forall i=1..n$
 2. Tìm i sao cho $Finish[i]==\text{false}$ và $Need[i] \leq Work$
Nếu không tìm được i , chuyển đến bước 4
 3. $Work=Work+Allocation[i]$, $Finish[i]=\text{true}$
Chuyển đến bước 2
 4. Nếu $Finish[i]==\text{true}$ $\forall i$ thì hệ thống ở trạng thái an toàn
- Độ phức tạp tính toán của thuật toán trạng thái an toàn: $O(m.n^2)$



Thuật toán yêu cầu tài nguyên

1. Nếu $\text{Request}[i] \leq \text{Need}[i]$, chuyển đến bước 2
Ngược lại thông báo lỗi (không có tài nguyên rõ)
2. Nếu $\text{Request}[i] \leq \text{Available}$, 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:
 $\text{Available} = \text{Available} - \text{Request}[i]$
 $\text{Allocation} = \text{Allocation} + \text{Request}[i]$
 $\text{Need}[i] = \text{Need}[i] - \text{Request}[i]$
đưa hệ thống vào trạng thái an toàn thì cấp phát tài nguyên cho P_i , ngược lại P_i phải chờ $\text{Request}[i]$ và trạng thái của hệ thống được khôi phục như cũ



Ví dụ banker

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

| | <u>Allocation</u> | | | <u>Max</u> | | | <u>Available</u> | | | <u>Need</u> | | |
|----|-------------------|---|---|------------|---|---|------------------|---|---|-------------|---|---|
| | 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 |



Ví dụ banker

- Hệ thống hiện đang ở trạng thái an toàn
- Thứ tự $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ 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 $Request[1] \leq Available: (1,0,2) < (3,3,2)$: Đúng
- Giả sử yêu cầu này được cấp phát, khi đó trạng thái giả định của hệ thống là:



Ví dụ banker

| | <u>Allocation</u> | | | <u>Need</u> | | | <u>Available</u> | | |
|----|-------------------|---|---|-------------|---|---|------------------|---|---|
| | A | B | C | A | B | C | A | B | C |
| P0 | 0 | 1 | 0 | 7 | 4 | 3 | 2 | 3 | 0 |
| 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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$. Do đó có thể cấp phát tài nguyên cho P_1 ngay.



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 rõi
 - Yêu cầu (0,2,0) của P0 cũng không thể cấp phát ngay vì mặc dù các tài nguyên rõi 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



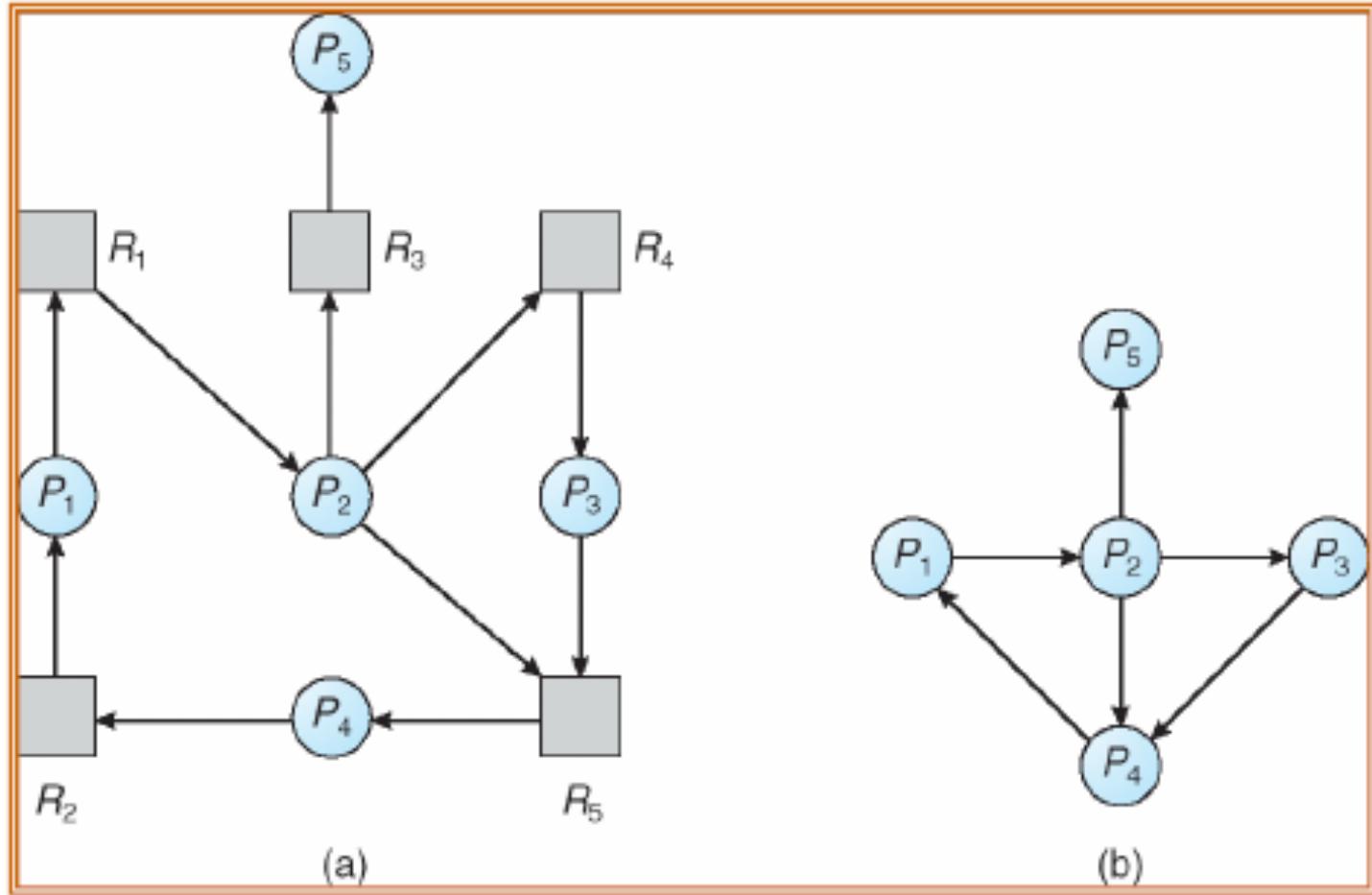
Phát hiện bế tắc

- 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



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 tiến trình 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





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 $n \times m$ xác định số thể hiện của mỗi loại tài nguyên đang được cấp phát cho các tiến trình
- *Request*: Ma trận $n \times m$ xác định yêu cầu hiện tại của mỗi tiến trình. Nếu $\text{Request}[i][j]=k$ thì tiến trình P_i yêu cầu cấp phát k thể hiện của tài nguyên R_j .



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 $Finish[i]=\text{false}$ nếu $Allocation[i]\neq 0$, ngược lại gán $Finish[i]=\text{true}$
 2. Tìm i sao cho $Finish[i]==\text{false}$ và $Request[i]\leq Work$. Nếu không tìm thấy i , chuyển đến bước 4
 3. $Work=Work+Allocation$, $Finish[i]=\text{true}$; chuyển đến bước 2
 4. Nếu $Finish[i]==\text{false}$ với $0\leq i\leq n-1$ thì hệ thống đang bị bế tắc (và tiến trình P_i đang bế tắc).
- Độ phức tạp tính toán của thuật toán: $O(m.n^2)$



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



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)



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?



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



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)