INDUSTRIAL UNIVERSITY OF HO CHI MINH CITY

Computer System

Process

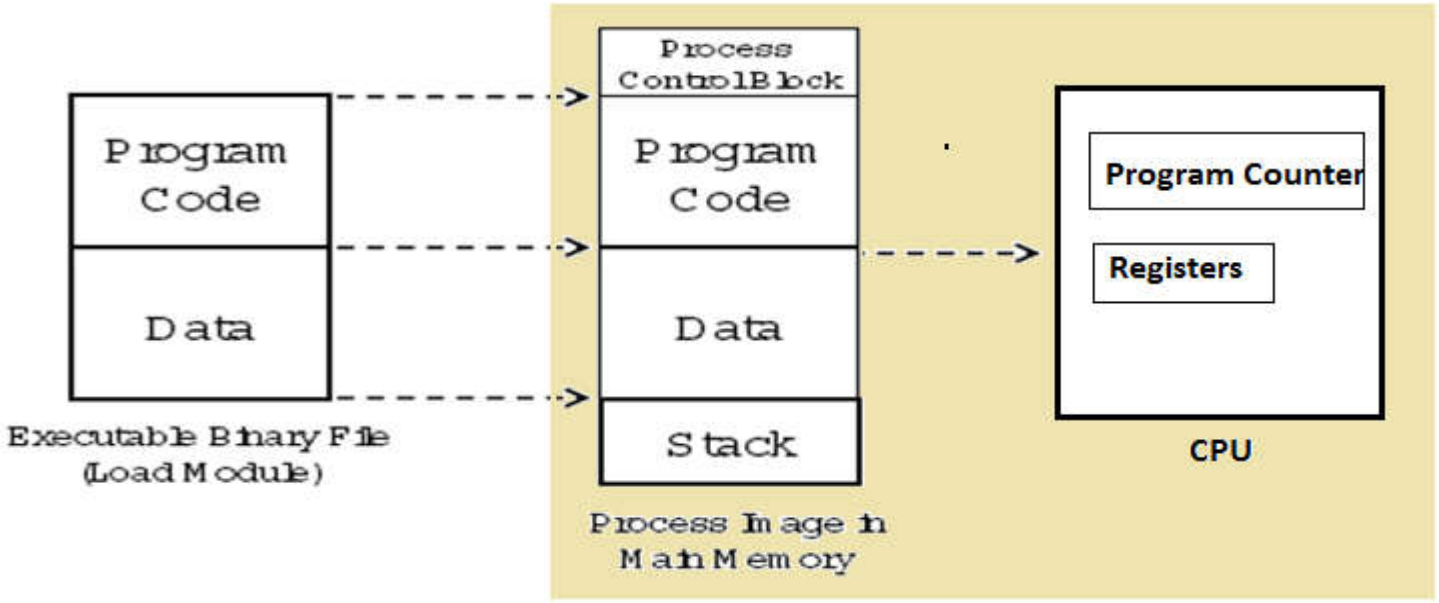
Lectured by Dr. Ngo Huu Dung

Nội dung

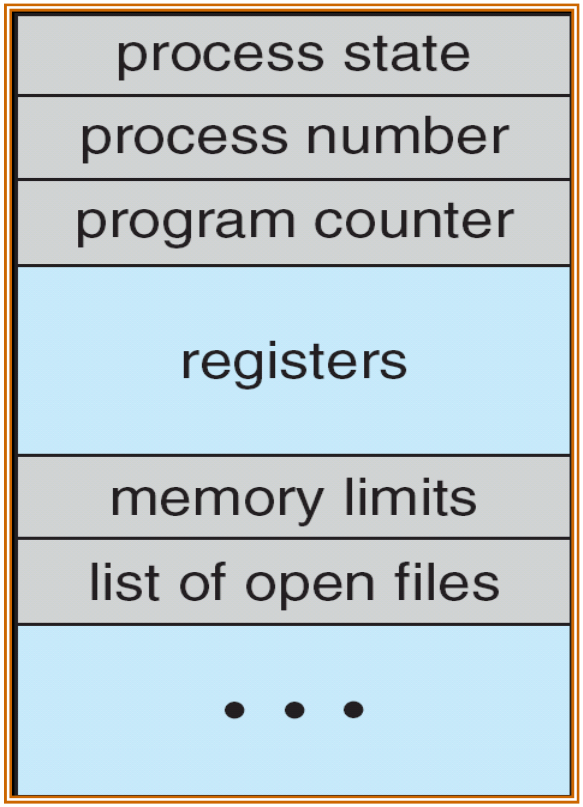
- Các khái niệm
  - chương trình và tiến trình
  - các thao tác & trạng thái của tiến trình
  - khối điều khiển tiến trình PCB
- Điều phối tiến trình
- Liên lạc giữa các tiến trình
- Đồng bộ tiến trình
- Deadlock

Tiến trình là gì?

- Process – tiến trình / quá trình = một chương trình đang thực thi
  - một tập các tài nguyên dùng để chạy một chương trình
  - nội dung bộ nhớ + nội dung các thanh ghi (+ trạng thái I/O)
- Chương trình (program) – tĩnh
- Tiến trình (process) – động



Process Control Block



Trạng thái của mỗi tiến trình được lưu trong ***process control block*** (PCB)

- Được xem là dữ liệu trong phần dữ liệu của HĐH
- Tương tự như là đối tượng của một lớp

PCB

- Các thông tin của một tiến trình:
  - Identification: tiến trình, tiến trình cha, người dùng, nhóm,…
  - Trạng thái tiến trình – Process state
  - Bộ đếm chương trình - Program counter
  - Thanh ghi CPU - CPU registers
  - Thông tin lập lịch của CPU
  - Thông tin liên quan bộ nhớ
  - Thông tin sổ sách: *mã số tiến trình, số lượng CPU, thời gian sử dụng CPU,…*
  - Trạng thái I/O: *danh sách file đang mở, danh sách các thiết bị đã cấp cho tiến trình*
  - Lưu vết thông tin

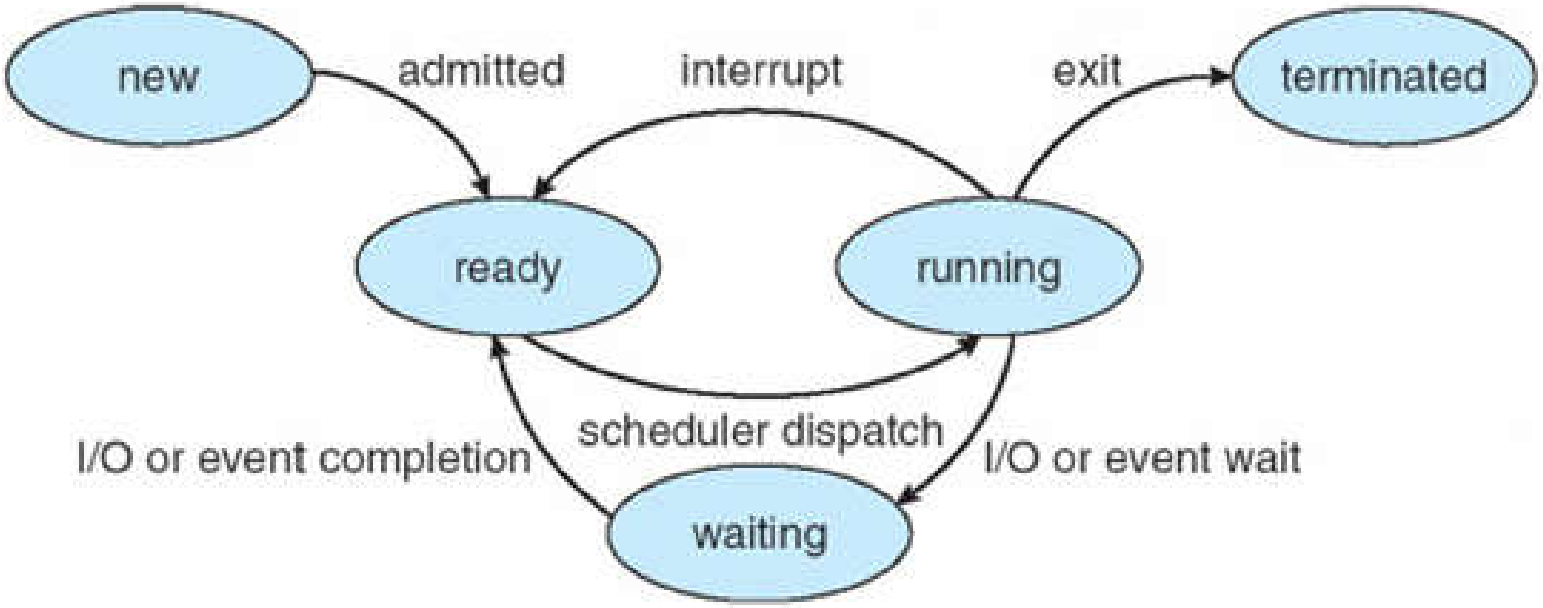
Các trạng thái của Tiến trình

- States:
  - ***New***: process được khởi tạo
  - ***Running***: process ở trong CPU các lệnh đang được thực hiện
  - ***Waiting (Blocked)***: process đang chờ một sự kiện nào đó xuất hiện
  - ***Ready***: process đang chờ đến lượt để được thực thi.
  - ***Exit (Terminated)***: completed/error exit

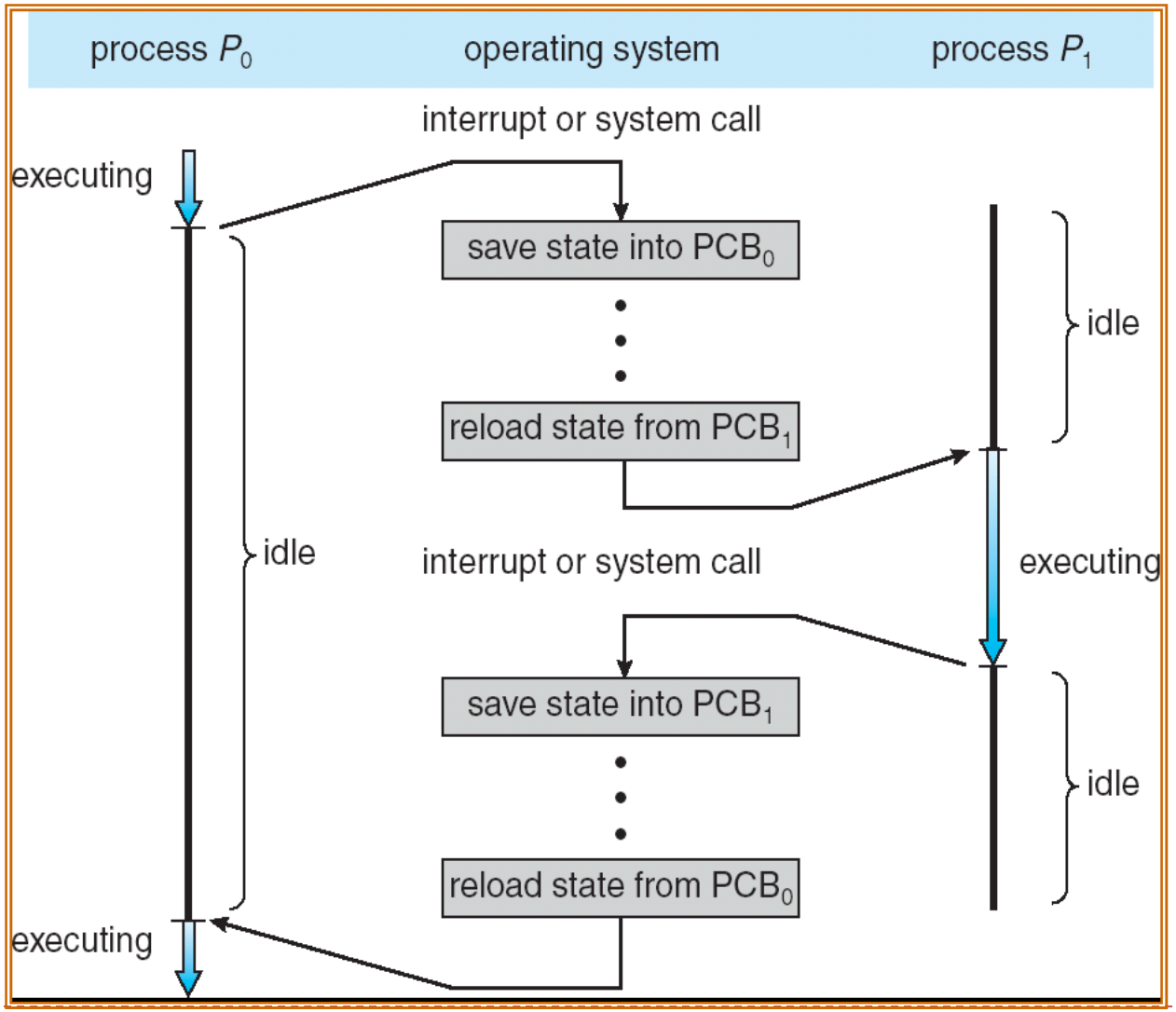
Chỉ có 1 process ở *running* trên mỗi processor tại một thời điểm

Có nhiều process chờ *ready* và *waiting* tại một thời điểm

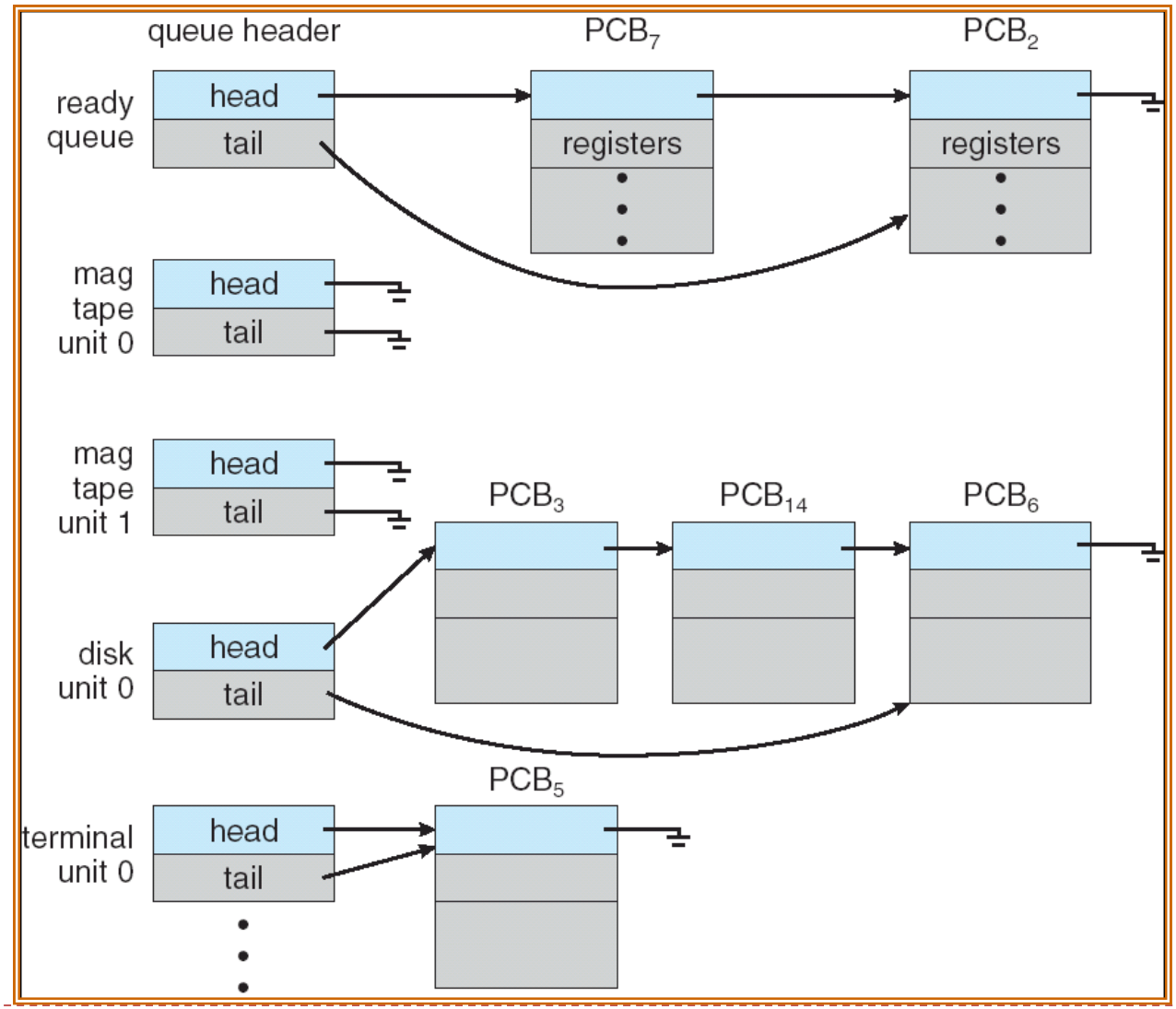
Các trạng thái của Tiến trình



CPU chuyển đổi giữa các tiến trình



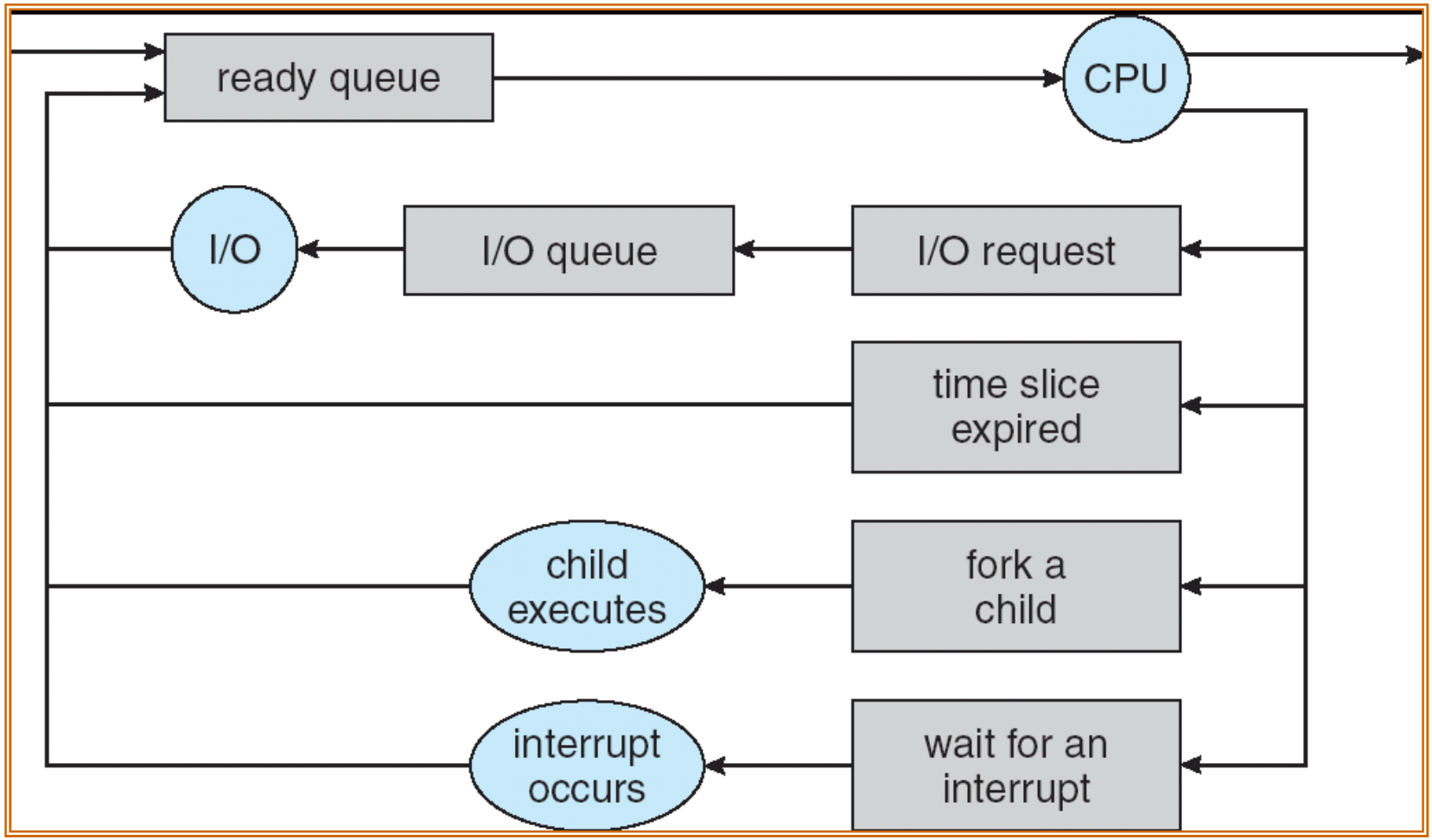
Ví dụ hàng đợi Ready



Process Scheduler

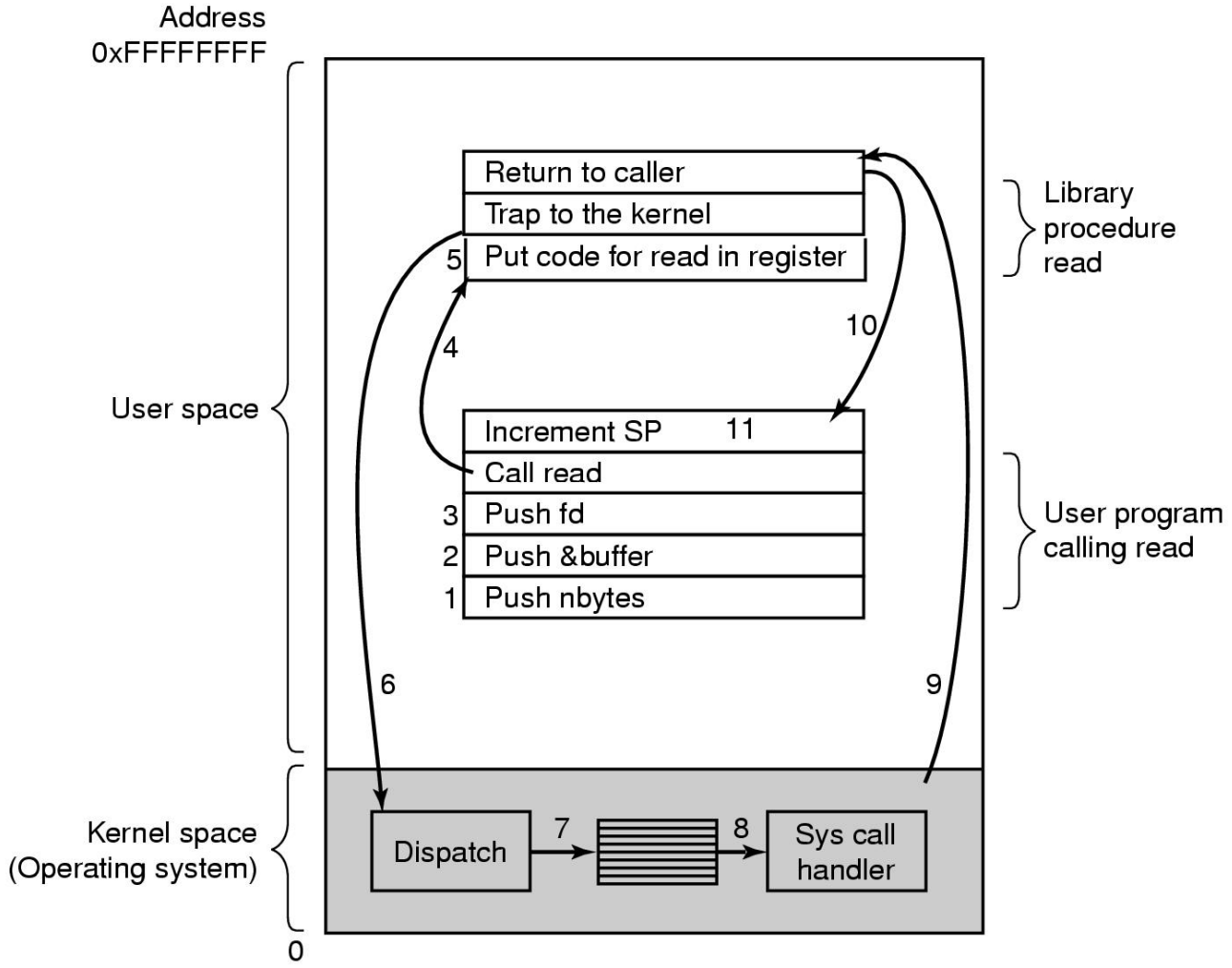
- Trong hệ multiprogramming và time-sharing, bộ lập lịch thực hiện chức năng điều phối các tiến trình
  - Chọn tiến trình được xử lý bởi CPU trong số các tiến trình đang chờ được xử lý.
  - Quản lý các hàng đợi read queue, I/O device queues
  - Chuyển ngữ cảnh

Lập lịch tiến trình (Process Scheduling)



System Call

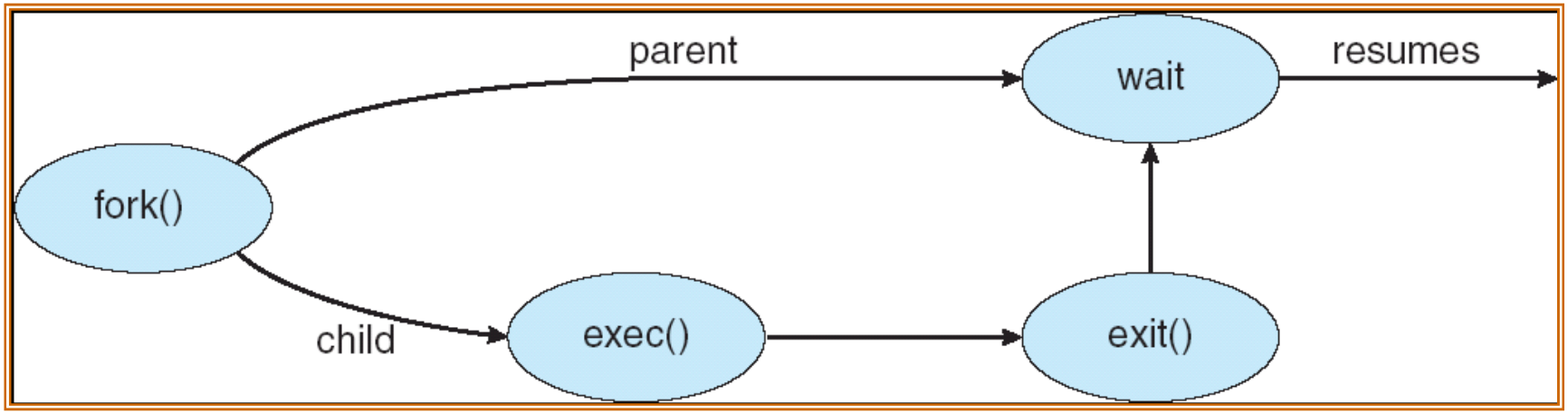
11 bước để thực hiện 1 system call **read (fd, buffer, nbytes)**



Tạo tiến trình

- Làm sao tạo một tiến trình? Sử dụng System call.
- Trong UNIX, một tiến trình có thể tạo một tiến trình khác bằng fork() (system call)
  - int pid = fork(); /* ngôn ngữ C */
- Tiến trình tạo gọi là tiến trình cha và tiến trình mới gọi là tiến trình con
- Tiến trình con là một bản sao của tiến trình cha (giống "hình dáng" và cấu trúc điều khiển tiến trình) ngoại trừ identification và trạng thái điều phối
  - Tiến trình con và cha chạy trên hai vùng bộ nhớ khác nhau
  - Mặc định là không có chia sẻ bộ nhớ
  - Chi phí tạo tiến trình là lớn vì quá trình copy
- Hàm exec() cũng để tạo một tiến trình, nhưng thường là tạo tiến trình của chương trình mới chứ không phải gọi lại chính nó

Tạo tiến trình



Ví dụ tạo tiến trình sử dụng Fork()

```
int main()
{
Pid_t  pid;
        /* fork another process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                exit(-1);
        }
        else if (pid == 0) { /* child process */
                execlp("/bin/ls", "ls", NULL);
        }
        else { /* parent process */
                /* parent will wait for the child to complete */
                wait (NULL);
                printf ("Child Complete");
                exit(0);
        }
}
```

Kết thúc tiến trình

- Một tiến trình có thể đợi một tiến trình khác hoàn thành bằng hàm wait() (system call)
  - Có thể là đợi cho tiến trình con thực thi xong như ví dụ trên
  - Cũng có thể đợi một tiến trình bất kì nào đó, phải biết PID của nó
- Để "hủy" một tiến trình khác dùng kill() (system call)
  - Điều gì xảy ra khi kill() được gọi?
  - Điều gì xảy ra nếu tiến trình "nạn nhân" vẫn chưa muốn "chết"?

Kết thúc tiến trình

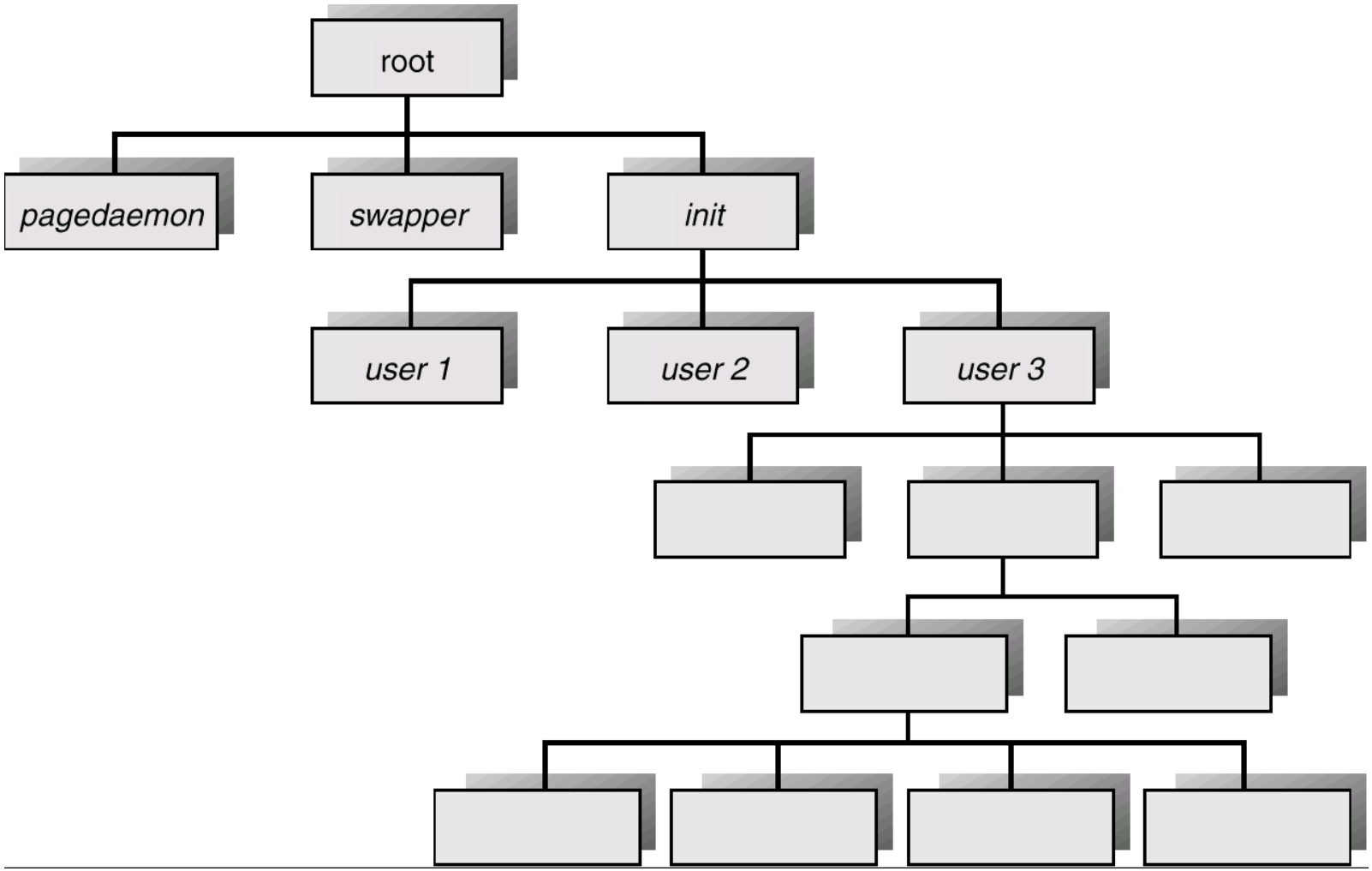
- Tiến trình kết thúc khi :
  - Normal exit (tự nguyện)
  - Error exit (tự nguyện)
  - Fatal error exit (ép buộc)
  - Được kết thúc bởi một tiến trình khác (ép buộc)

- Một system call được gửi tới HĐH yêu cầu thực hiện kết thúc một tiến trình
  - Trong Unix : exit() => kết thúc tự nguyện
    kill() => kết thúc ép buộc

Kết thúc tiến trình

- Tiến trình xử lý statement cuối và hỏi OS để xóa nó (**exit**)
  - Output data từ child sang parent (thông qua **wait**)
  - Tài nguyên tiến trình được giải phóng bởi OS
- Parent có thể kết thúc tiến trình của child (**abort**)
  - Child vượt quá tài nguyên được phân bổ
  - Nhiệm vụ gán cho child không còn được yêu cầu
  - Nếu parent kết thúc
    - Một số OS không cho phép child tiếp tục hoạt động khi parent kết thúc
      - Tất cả child bị kết thúc → **cascading termination**

Cây tiến trình trong hệ UNIX thông thường



Tóm tắt

- Khái niệm tiến trình . Phân biệt tiến trình với chương trình
- Các trạng thái của tiến trình. Sự thay đổi trạng thái.
- Khối thông tin quản lý tiến trình (PCB)
- Tạo tiến trình
- Kết thúc tiến trình

Điều phối tiến trình

Process Management

Multiprogramming và Time-sharing

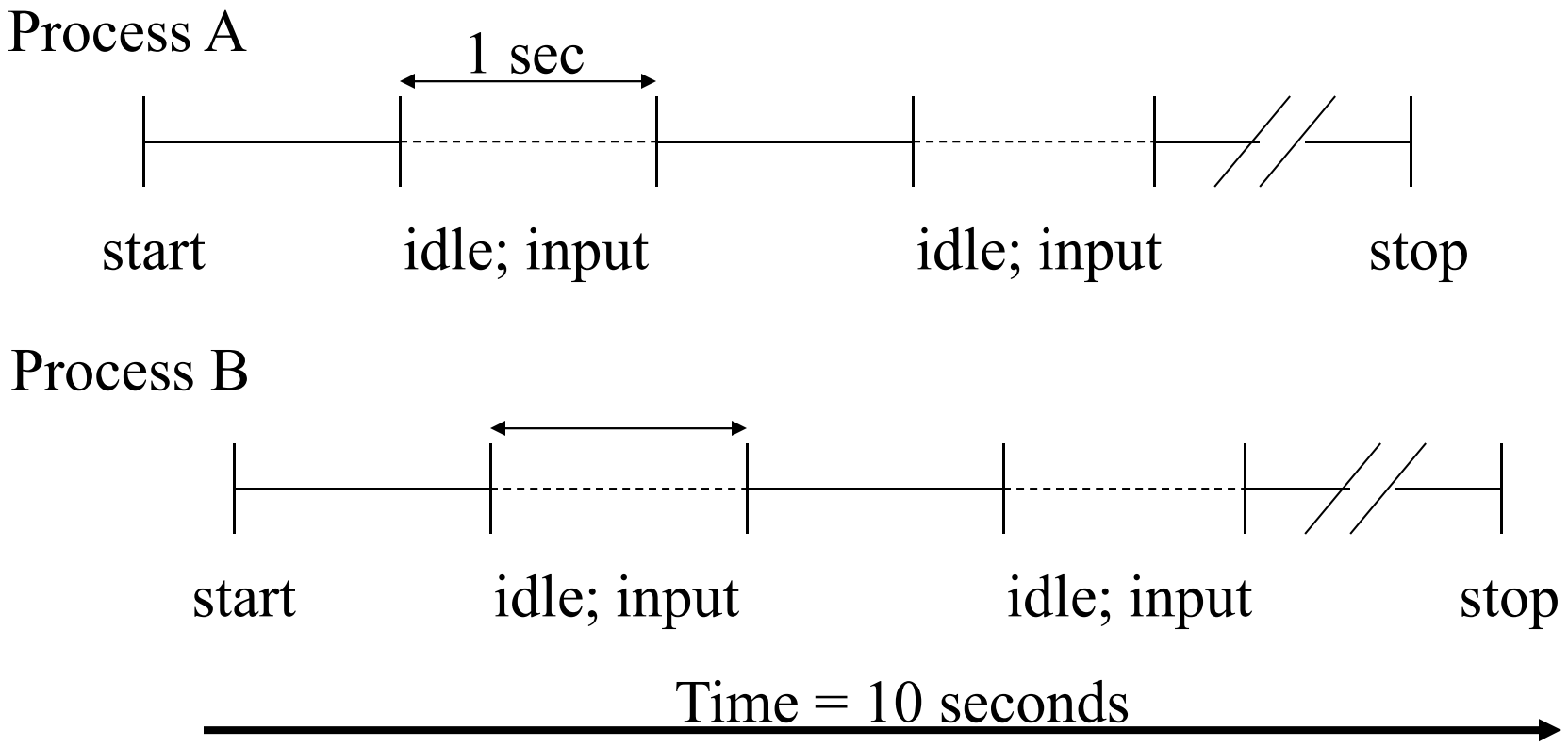
- **Đa chương** (**multiprogramming**) cho phép tối ưu hiệu quả CPU

- **Chia sẻ thời gian (time-sharing) hay đa nhiệm (multitasking)**
  - Nhiều công việc cùng được thực hiện thông qua cơ chế chuyển đổi của CPU
  - thời gian mỗi lần chuyển đổi diễn ra rất nhanh.

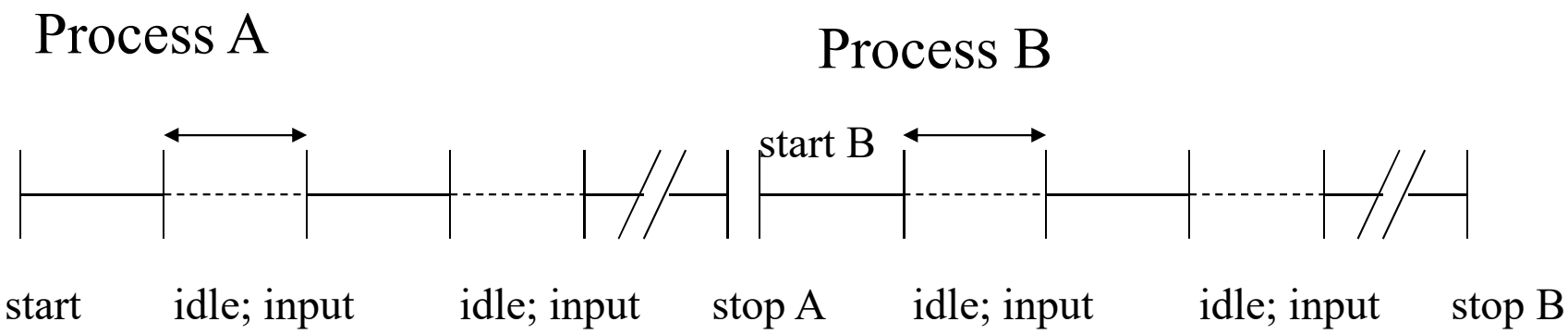
Giả thiết

- Giả thiết:
  - Chỉ có 1 CPU vật lý, và tại một thời điểm CPU chỉ xử lý **1** lệnh
  - Nhiều công việc tranh dành CPU
  - CPU là tài nguyên khan hiếm
- Các công việc là độc lập và tranh giành tài nguyên lẫn nhau (giả thiết này không thật sự đúng trong tất cả các hệ thống, ngữ cảnh)
- Người điều phối làm trung gian giữa các công việc để sao cho tối ưu hóa việc thực thi của hệ thống

Ví dụ đa chương trình



Ví dụ đa chương trình (tt)

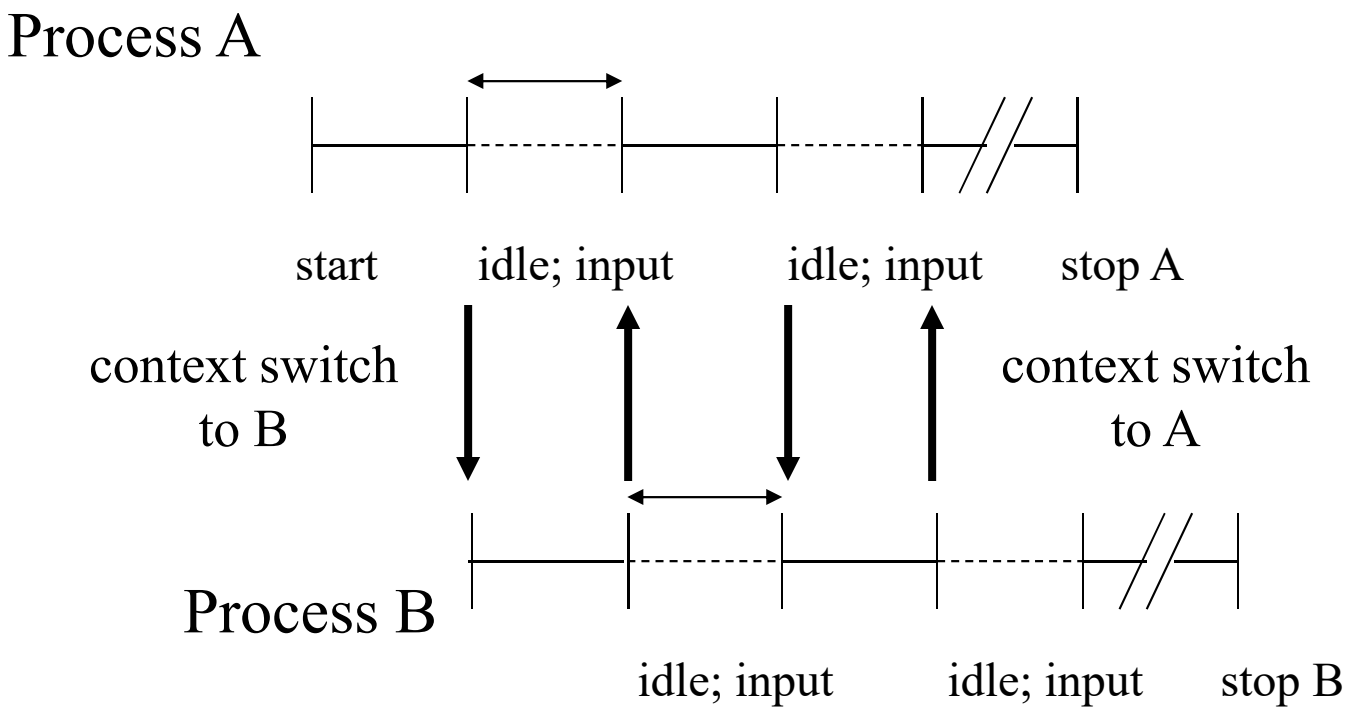


Tổng thời gian = 20 giây

Hiệu suất = 2 tiến trình trong 20 giây = 0.1 tiến trình/giây

Thời gian chờ trung bình = (0+10)/2 = 5 giây

Ví dụ đa chương trình (tt)



Hiệu suất = 2 tiến trình 11 giây = 0.18 tiến trình/giây

Thời gian chờ trung bình = (0+1)/2 = 0.5 giây

Điều phối tiến trình

- Nhu cầu thực thi nhiều tiến trình đồng thời trong các hệ thống Multiprogramming và Time-sharing

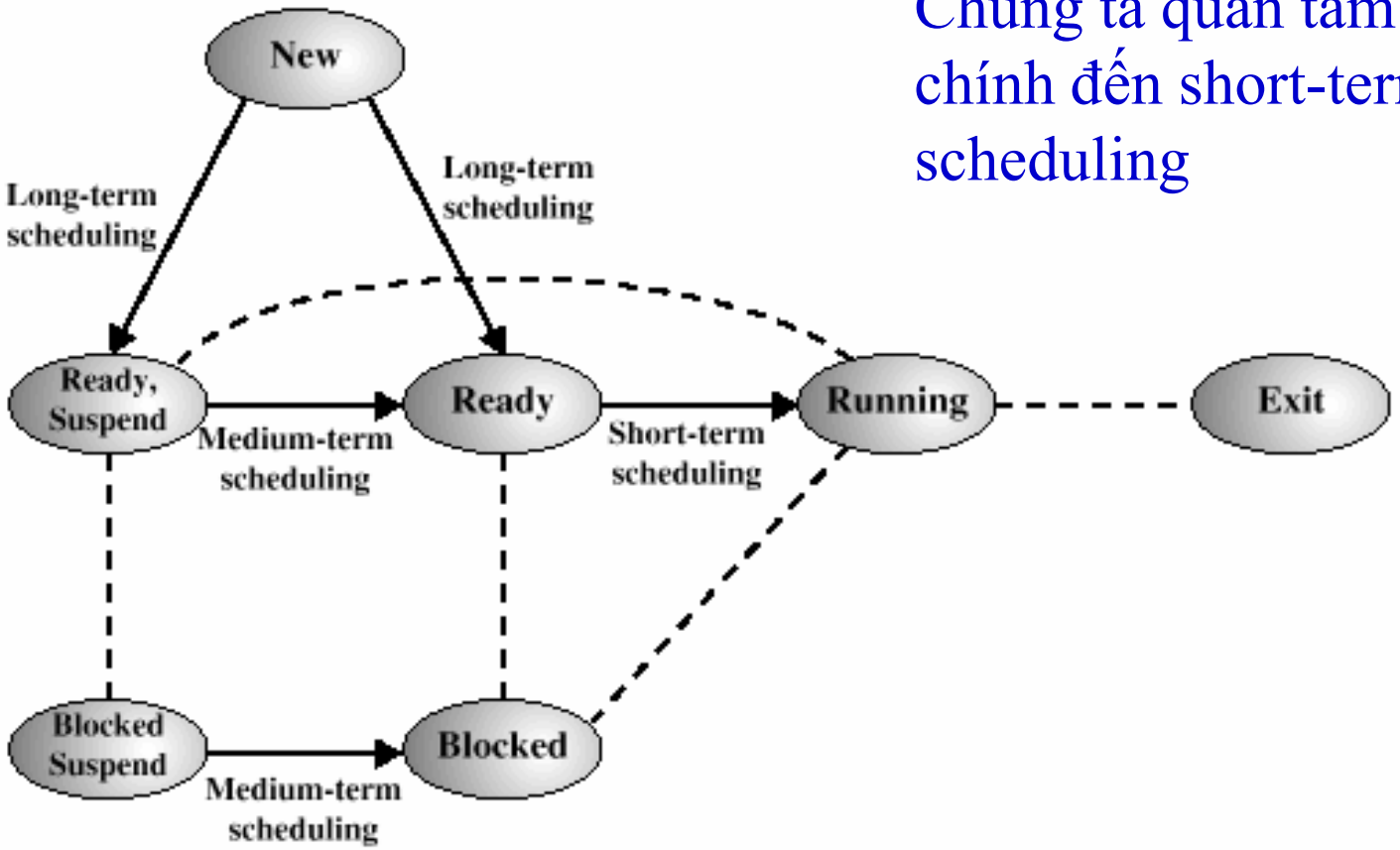
- Chức năng điều phối tiến trình được thực hiện bởi :
  - **Bộ điều phối –scheduler** : sử dụng một giải thuật điều phối thích hợp
  - **Bộ phân phối – dispatcher** : chuyển đổi ngữ cảnh và chuyển CPU cho tiến trình được chọn.

Điều phối được thực hiện khi nào ?

- Bộ điều phối cần ra quyết định vào thời điểm :
  - Khi một tiến trình kết thúc hay chuyển sang trạng thái blocked (chờ I/O , …) => chọn tiến trình nào trong hàng đợi ready ?
  - Một ngắt I/O xuất hiện từ một thiết bị I/O báo đã hoàn tất
  - Ngắt định kỳ xuất hiện từ bộ đếm thời gian

Phân loại lập lịch

Chúng ta quan tâm chính đến short-term scheduling



Hình 6.1 Sơ đồ lập lịch và chuyển trạng thái của tiến trình

Chúng ta cần tối ưu hóa những gì?

Phần hệ thống:

Tận dụng processor: phần trăm sử dụng processor

Hiệu suất: số tiến trình hoàn thành trên một đơn vị thời gian

Phần người dùng:

Turnaround time: khoảng thời gian giữa bắt đầu và kết thúc công việc (gồm thời gian chờ). Cho tiến trình theo lô, tuần tự.

Response time: cho những tiến trình tương tác, thời gian từ khi gửi yêu cầu cho đến khi nhận được phản hồi.

Deadlines: khi thời hạn thực thi của tiến trình được xác định, thì phần trăm hoàn thành đúng thời hạn phải được quan tâm.

Mục tiêu điều phối

- Đặc điểm của tiến trình
  - Tính hướng nhập/xuất hay hướng xử lý
  - Tiến trình tương tác user hay xử lý theo lô
  - Độ ưu tiên của tiến trình

- Các yếu tố đánh giá 1 chiến lược điều phối :
  - Phù hợp đặc điểm của tiến trình
  - Tính công bằng , hiệu quả

Mục tiêu điều phối

- Mục tiêu điều phối:
  - Công bằng
    - Không có tiến trình nào phải chờ vô hạn
  - Tối đa thời gian sử dụng CPU (efficiency)
  - Thông lượng tối đa (throughput)
    - tối đa số tiến trình được xử lý trong 1 đơn vị thời gian
  - Cực tiểu thời gian hoàn thành (turnaround time)
    - **Là tổng thời gian chờ trong Ready List + thời gian thực thi CPU + thời gian thực hiện I/O**
  - Cực tiểu thời gian chờ (waiting time) trong Ready List
  - Cực tiểu thời gian đáp ứng (response time)
  - Đảm bảo tính ưu tiên
  - …

Thiết kế

Hai chiều

Chọn lựa

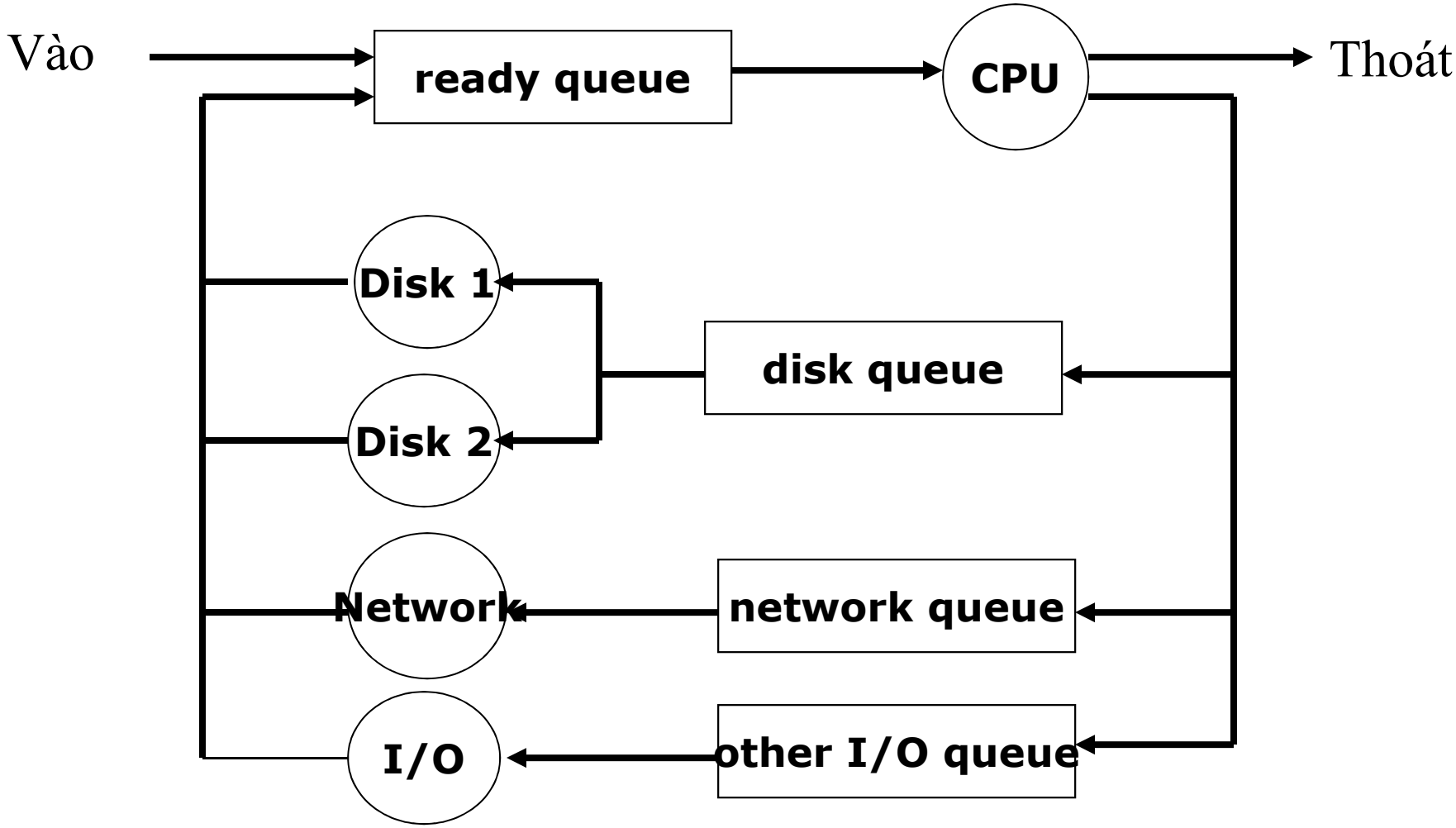
Ready job nào sẽ được thực thi kế tiếp?

Preemption

preemptive: tiến trình đang thực thi có thể bị ngắt và chuyển vào trạng thái Ready

Non-preemptive: một khi tiến trình ở trong trạng thái Running, nó sẽ tiếp tục thực thi cho đến khi kết thúc hoặc bị block vì I/O hay các dịch vụ của hệ thống

Sơ đồ hàng đợi



Mô hình hàng đợi

Vòng tròn biểu diễn servers

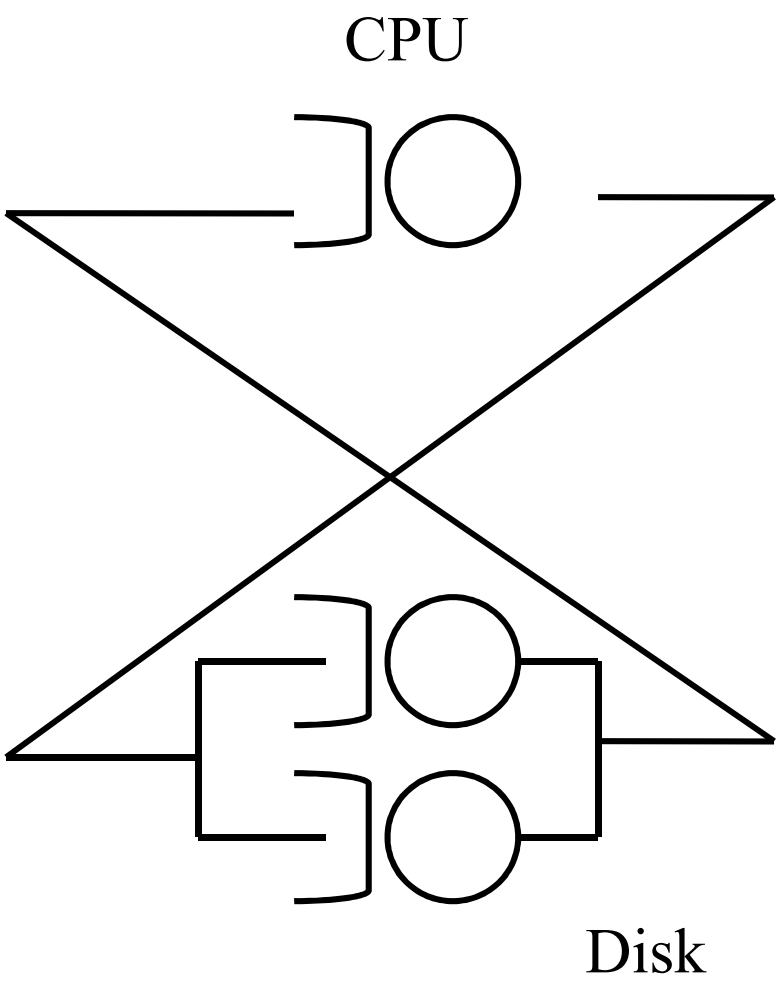
Hình chữ nhật biểu diễn hàng đợi (queues)

Công việc đến và rời khỏi hệ thống

Queuing theory(lý thuyết hàng đợi) giúp chúng ta dự đoán

- Chiều dài trung bình của hàng đợi
- Số công việc so với thời gian được phục vụ

Đặc tính công việc



I/O-bound jobs

- CV liên tục truy suất I/O
- Ít dùng đến CPU

CPU-bound jobs

- CV truy suất I/O rất ít
- Dùng CPU nhiều

Lập lịch CPU

Chọn giữa các tiến trình sẵn sàng trong bộ nhớ, cấp một CPU cho một tiến trình nào đó.

Việc lập lịch CPU được sử dụng khi một tiến trình:

1. Chuyển từ trạng thái running sang waiting.
2. Chuyển từ trạng thái running sang ready.
3. Chuyển từ waiting sang ready.
4. Kết thúc.

Lập lịch 1 và 4 là *nonpreemptive*.

Các lập lịch còn lại là *preemptive*.

Điều phối

Bộ điều phối trao điều khiển CPU cho tiến trình được chọn bởi lập lịch short-term; quá trình này gồm:

- switching context
- Chuyển qua user mode
- Nhảy tới vị trí thích hợp trong chương trình để bắt đầu thực thi nó

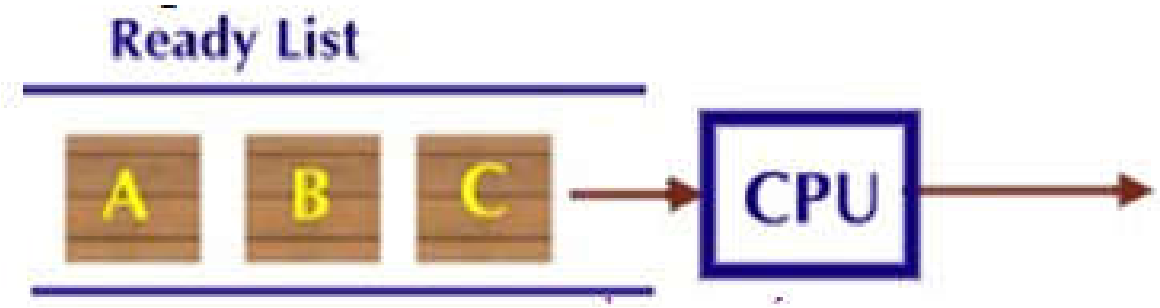
Độ trễ điều phối – thời gian để bộ điều phối dừng tiến trình này và bắt đầu một tiến trình kia.

Khảo sát các chiến lược điều phối

- FIFO hay FCFS (first come first served)
- Round Robin (phân phối xoay vòng)
- Điều phối với độ ưu tiên
- SJF (Shortest-job-first)
- Chiến lược điều phối với nhiều mức độ ưu tiên

Lập lịch FIFO – First In First Out

- Tiến trình nào vào Ready list trước → được cấp CPU trước
- CPU được giải phóng chỉ khi
  - Tiến trình kết thúc xử lý
  - Khi có một yêu cầu I/O

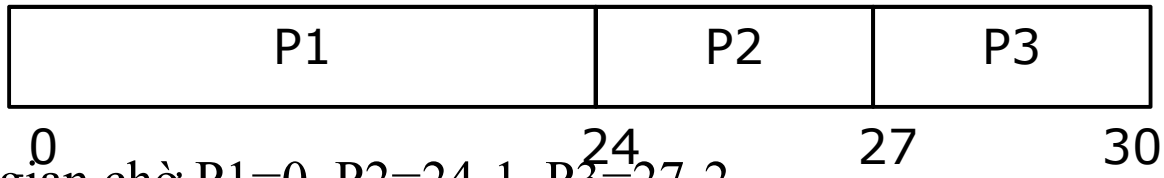


FIFO

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	0	24
P2	1	3
P3	2	3

- Thứ tự cấp phát CPU: P1, P2, P3

  Gantt chart của lập lịch như sau:



- Thời gian chờ P1=0, P2=24-1, P3=27-2
- Thời gian chờ trung bình: (0+23+25)/3=16miliseconds

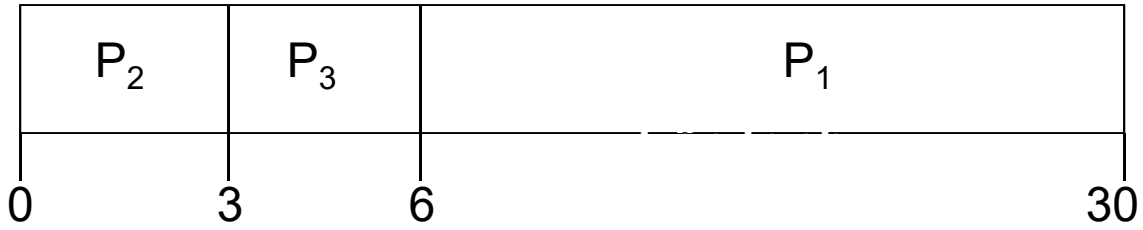
FIFO (tt.)

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	2	24
P2	0	3
P3	1	3

Giả sử tiến trình đến theo thứ tự

$$P_2 , P_3 , P_1 .$$

Gantt chart của lập lịch như sau:



Thời gian chờ P_1 = 6-2; P_2 = 0, P_3 = 3-1

Trung bình thời gian chờ: (4 + 0 + 2)/3 = 2

Tốt hơn nhiều so với trường hợp trước.

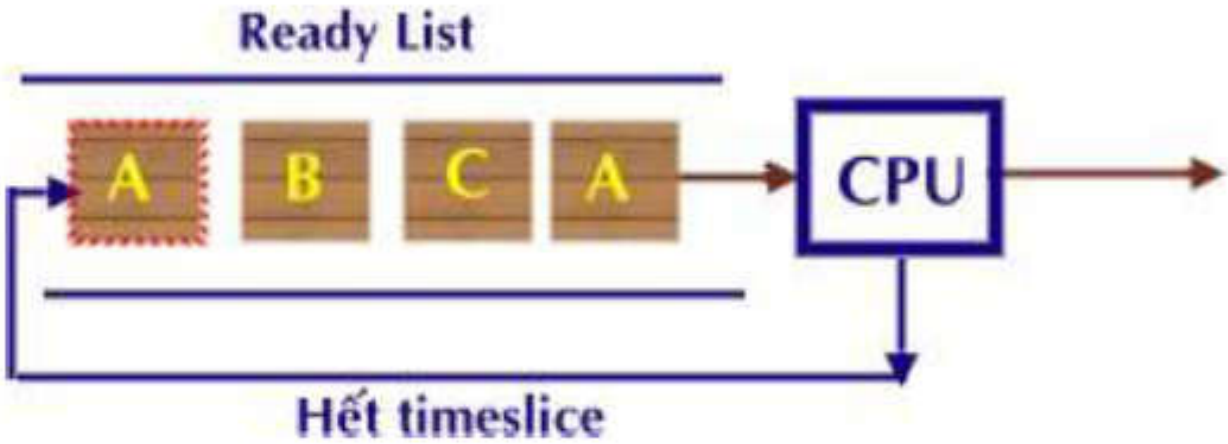
Convoy effect tiến trình ngắn có thể nằm sau tiến trình dài

FIFO (tt.)

- Nhận xét
  - Thời gian chờ trung bình:
    - Phụ thuộc vào thời gian xử lý của các process
    - Phụ thuộc vào thứ tự của các process trong RL
  - Là giải thuật điều phối theo nguyên tắc độc quyền
    - Không phù hợp với hệ time-sharing

Round Robin (Phân phối xoay vòng)

- Các process được xử lý xoay vòng
  - Một khoảng thời gian sử dụng CPU như nhau gọi là *time quantum*
  - Hết thời gian quantum dành cho process, HĐH thu hồi CPU và cấp cho process kế tiếp trong RL
  - Process được đưa trở lại vào RL chờ đến lượt



Round Robin (RR)

- Nếu có n tiến trình trong hàng đợi ready và time quantum là q, thì mỗi tiến trình sẽ nhận $1/n$ thời gian sử dụng CPU. Không tiểu trình nào phải đợi lâu hơn $(n\text{-}1)q$ đơn vị thời gian.

- Độ hiệu quả

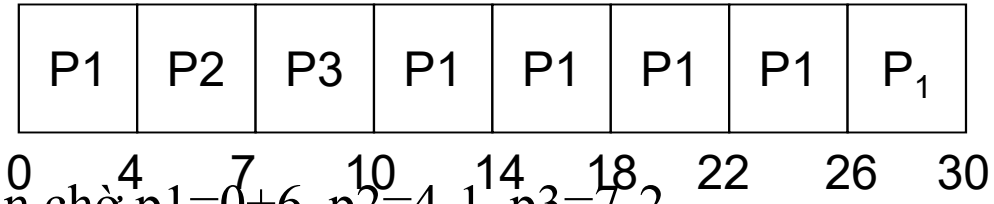
  q lớn $\Rightarrow$ FIFO

  q nhỏ $\Rightarrow$ q phải đủ lớn so với thời gian context switch, nếu không thì tổng chi phí sẽ rất cao.

Round Robin (Phân phối xoay vòng)

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P1	0	24
P2	1	3
P3	2	3

- Nếu time quantum = 4 miliseconds
- Thứ tự cấp phát CPU như sau:



- Thời gian chờ p1=0+6, p2=4-1, p3=7-2
- Thời gian chờ trung bình: (0+6+3+5)/3=4.66miliseconds

Round Robin (Phân phối xoay vòng)

- CPU được giải phóng khi
  - Hết thời gian quantum
  - Tiến trình kết thúc
  - Tiến trình bị khóa

- Nhận xét:
  - Là giải thuật điều phối không độc quyền → loại bỏ hiện tượng độc chiếm CPU
  - Độ dài quantum thế nào là hợp lý??

Điều phối với độ ưu tiên

- Một độ ưu tiên (integer) được gán vào mỗi tiến trình
- CPU được cấp cho tiến trình có độ ưu tiên cao nhất (số nhỏ nhất ≡ độ ưu tiên cao nhất).
  - Preemptive (không độc quyền)
  - Non-preemptive (độc quyền)

Điều phối với độ ưu tiên

Ví dụ : (độ ưu tiên 1 > độ ưu tiên 2> độ ưu tiên 3)

Tiến trình	Thời điểm vào RL	Độ ưu tiên	Thời gian xử lý
P1	0	3	24
P2	1	1	3
P3	2	2	3

Sử dụng thuật giải độc quyền, thứ tự cấp phát CPU như sau :

P1	P2	P3
0	‘24	27 30

Sử dụng thuật giải không độc quyền, thứ tự cấp phát CPU như sau :

P1	P2	P3	P1
0	‘1	4	7 30

Điều phối với độ ưu tiên

Minh họa độ ưu tiên (khôngđộc quyền)

P	T_{RL}	Priority	CPU burst
P1	0	2	24
P2	1	0	3
P3	2	1	3

P	TT	WT
P1	30	0+(7-1)
P2	4-1	0
P3	7-2	4-2

Avg_{WT} = (6+0+2)/3 = 2.66

0:00 P1 vào, P1 dùng CPU

0:01 P2 vào (độ ưu tiên cao hơn P1)

P2 dành quyền dùng CPU

0:02 P3 vào (độ ưu tiên thấp hơn P2)

P3 không dành được quyền dùng CPU

0:4 P2 kết thúc, P3 dùng CPU

0:7 P3 dừng, P1 dùng CPU

0:30 P1 dừng

Điều phối với độ ưu tiên

- Nhận xét:
  - Vấn đề ≡ Starvation – các tiến trình độ ưu tiên thấp có thể không bao giờ thực thi được.
  - Giải pháp ≡ Aging – tiến trình sẽ tăng độ ưu tiên theo thời gian. (sống lâu lên lão làng..)
  - SJF là một dạng điều phối theo **độ ưu tiên (độ ưu tiên: dự đoán thời gian CPU burst kế tiếp).**

Lập lịch Shortest-Job-First (SJR)

- Tùy thuộc vào thời gian sử dụng CPU của tiến trình. Sử dụng độ dài của thời gian này để lập lịch.
- Hai lược đồ:
  - Non-preemptive – một khi CPU cấp cho một tiến trình nó không thể bị chiếm cho đến khi nó hoàn thành.
  - Preemptive – nếu một tiến trình mới vào mà thời gian cần dùng CPU ít hơn thời gian còn lại cần dùng CPU của tiến trình hiện hành. Lược đồ này gọi là Shortest-Remaining-Time-First (SRTF).
- SJF là tối ưu – cho kết quả tốt nhất về trung bình thời gian chờ của một tập các tiến trình.

SJF – Shortest Job First

- Công việc ngắn nhất thực hiện trước

Là một dạng độ ưu tiên đặc biệt với độ ưu tiên

p_i = thời_gian_còn_lại($Process_i$)

➔ Có thể cài đặt độc quyền hoặc không độc quyền

SJF – Shortest Job First

Minh họa SJF (độc quyền)(1)

P	$T_{arriveRL}$	CPU burst
P1	0	24
P2	1	3
P3	2	3

P	TT	WT
P1	24	0
P2	27	24-1
P3	30	27-2

$Avg_{WT} = (23+25)/3 = 16$

0:00 P1 vào, P1 dùng CPU
0:01 P2 vào RL
0:02 P3 vào RL

0:24 P1 kết thúc, P2 dùng CPU
0:27 P2 dừng, P3 dùng CPU
0:30 P3 dừng

SJF – Shortest Job First

Minh họa SJF (độc quyền)(2)

P	$T_{arriveRL}$	CPU burst
P1	0	24
P2	1	3
P3	1	2

P	TT	WT
P1	24	0
P2	29	26-1
P3	26	24-2

$Avg_{WT} = (24+22)/3 = 15.33$

0:00 P1 vào, P1 dùng CPU
0:01 P2 vào
0:01 P3 vào

0:24 P1 kết thúc, P3 dùng CPU
0:26 P3 dừng, P2 dùng CPU
0:29 P2 dừng

SJF – Shortest Job First

Minh họa SJF (khôngđộc quyền) (1)

P	$T_{arriveRL}$	CPU burst
P1	0	24
P2	1	3
P3	2	3

P	TT	WT
P1	30	0+(7-1)
P2	4-1	0
P3	7-2	4-2

Avg_{WT} = (6+0+2)/3 = 2.66

0:00 P1 vào, P1 dùng CPU

0:01 P2 vào (độ ưu tiên cao hơn P1)

P2 dành quyền dùng CPU

0:4 P2 kết thúc, P3 dùng CPU

0:7 P3 dừng, P1 dùng CPU

0:30 P1 dừng

SJF – Shortest Job First

Minh họa SJF (khôngđộc quyền) (2)

P	$T_{arriveRL}$	CPU burst
P1	0	24
P2	1	5
P3	3	4

P	TT	WT
P1	33	0+(10-1)
P2	6	0
P3	10	6-3

$Avg_{WT} = (9+0+3)/3 = 4$

0:00 P1 vào, P1 dùng CPU

0:01 P2 vào (độ ưu tiên cao hơn P1)

P2 dành quyền dùng CPU

0:03 P3 vào (độ ưu tiên < P2)

P2 dành quyền dùng CPU

0:6 P2 kết thúc, P3 dùng CPU

0:9 P3 dừng, P1 dùng CPU

0:33 P1 dừng

Bài tập: Hãy điều phối

CPU: SJF không độc quyền. R1,R2: FIFO

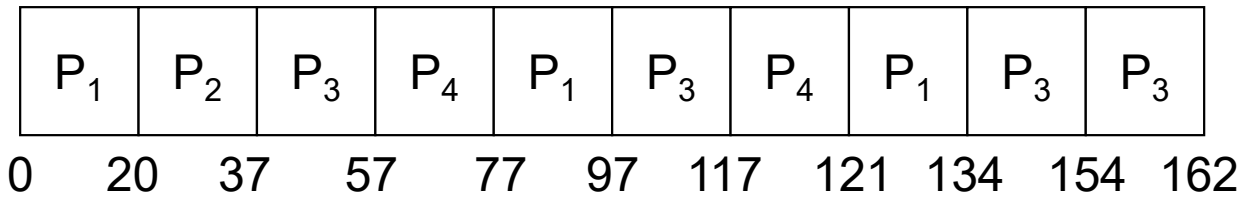
Tiến trình	Thời điểm vào Ready list	CPU1	IO lần 1		CPU2	IO lần 2	
			Thời gian	Thiết bị		Thời gian	Thiết bị
P1	0	8	5	R1	1	0	Null
P2	2	1	8	R2	2	5	R1
P3	10	6	5	R1	2	3	R2
P4	11	3	20	R2	0	0	Null

Tóm tắt

- Hệ multiprogramming và multitasking
- Bộ điều phối (scheduler)
  - Chức năng
  - Tổ chức điều phối
  - Mục tiêu đặt ra cho chiến lược điều phối
  - Nguyên tắc điều phối độc quyền và không độc quyền
- Các chiến lược điều phối : FCFS, RoundRobin, độ ưu tiên, SJF

Tiến trình	RL	Thời gian dùng CPU
P_1	*1*	53
P_2	*2*	17
P_3	*3*	68
P_4	*4*	24

- FIFO
- RR với Time Quantum = 20


P1
P2
P3
P4
P1
P3
P4
P1
P3
P3
0
20
37
57
77
97
117
121
134
154
162

CHƯƠNG 2:
QUẢN LÝ TIẾN TRÌNH – P3
ĐỒNG BỘ TIẾN TRÌNH
Process Management

Nội dung

- Các khái niệm
  - chương trình và tiến trình
  - các thao tác & trạng thái của tiến trình
  - khối điều khiển tiến trình ProcessControlBlock
- Điều phối tiến trình
- Liên lạc giữa các tiến trình
- Đồng bộ tiến trình
- Deadlock

Liên lạc giữa các tiến trình

- Các tiến trình trong hệ thống có thể độc lập hay có hợp tác với nhau
- Các tiến trình hợp tác với nhau xuất phát từ nhu cầu :
  - Chia sẻ thông tin
  - Tăng tốc độ tính toán
  - Cấu trúc module của chương trình
- Khi hợp tác , các tiến trình cần giao tiếp với nhau (**interprocess communication – IPC**)

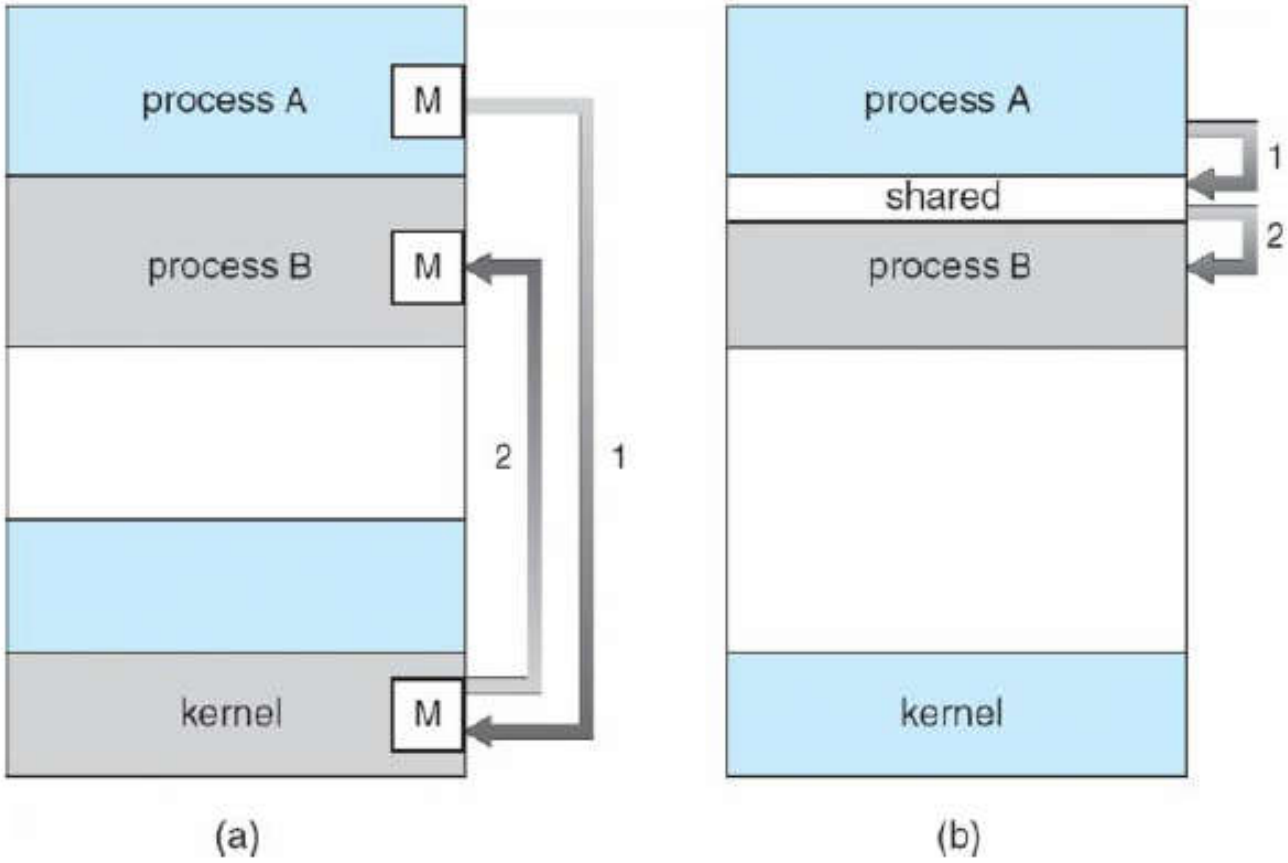
Liên lạc giữa các tiến trình

- Do mỗi tiến trình sở hữu một không gian địa chỉ riêng => HĐH phải cung cấp cơ chế liên lạc giữa các tiến trình
- Các vấn đề nảy sinh trong liên lạc giữa các tiến trình :
  - Liên kết tường minh hay tiềm ẩn
  - Liên lạc theo chế độ đồng bộ hay bất đồng bộ
  - Liên lạc giữa các tiến trình trong 1 máy tính khác biệt với liên lạc giữa các tiến trình giữa các máy tính khác nhau

Các cơ chế liên lạc

- Tín hiệu – signal
- Pipe
- Vùng nhớ chia sẻ
- Trao đổi thông điệp
- Socket

Communication models



Tham khao : IPC share memory (http://www.cs.cf.ac.uk/Dave/C/node27.html)

ĐỒNG BỘ TIẾN TRÌNH

Khái niệm

- Sự cần thiết của đồng bộ hóa tiến trình trong hệ đa nhiệm ?
  - Yêu cầu độc quyền truy suất
    - Phát sinh do truy suất tài nguyên không thể chia sẻ
    - Kết quả tác động đến tài nguyên làm ảnh hưởng lẫn nhau
  - Yêu cầu phối hợp
    - Có những tình huống, các tiến trình cần phối hợp hoạt động

- Việc xem xét sự đồng bộ giữa các tiến trình dựa trên giả định các tiến trình có khả năng liên lạc với nhau
  - vùng nhớ chung

Ví dụ 1

- Ví dụ 1 : hai tiến trình cùng truy suất 1 biến chung

  characters = characters + 1;

- Lệnh máy tương đương

read characters into register r1

increment r1

write register r1 to characters

Ví dụ 1

- 2 tiến trình cùng thực hiện đếm số ký tự theo đoạn code trên:
  - Ban đầu characters = 100

P1 :
read characters into register r1 //r1 =100

increment r1 //r1 = 101
write register r1 to characters
//characters = 101

P2 :

read characters into register r2 //r2 =100
increment r2 //r2 = 101
write register r2 to characters
//characters = 101

Ví dụ 2

- Vd2: Taikhoan = 800, P1 muốn rút 500, P2 muốn rút 400

P1

```
If ( taikhoan  >= tienrut )
      taikhoan = taikhoan  -
tienrut ;
Else
      error (“khong the rut !”);
```

P2

```
If ( taikhoan  >= tienrut )
      taikhoan = taikhoan  -
tienrut ;
Else
      error (“khong the rut !”);
```

Ví dụ 2

Taikhoan = 800

P1

```
If ( taikhoan >= tienrut )




     taikhoan = taikhoan  - tienrut ;
else
     error (“hettien , ko the rut !”);
```

P2

```
If ( taikhoan >= tienrut )
    taikhoan = taikhoan  - tienrut ;
Else
     error (“hettien , ko the rut !”);
```

Từ khóa

- Critical section (miền găng): đoạn mã nguồn đọc/ghi dữ liệu lên vùng nhớ chia sẻ
- Race condition (tranh đoạt điều khiển): có tiềm năng bị xen kẻ lệnh giữa các tiểu trình khác nhau trong miền găng
  - Kết quả không xác định
- Mutual exclusion: cơ chế đồng bộ đảm bảo chỉ có một tiểu trình duy nhất được thực hiện trong miền găng tại một thời điểm
- Deadlock: tình trạng các tiểu trình bị khóa mãi mãi
- Starvation: đang thực thi nhưng không có tiến triển.

Tranh đoạt điều khiển

Tranh đoạt điều khiển là gì? (Race condition)

```
if (taikhoan – tien_rut >= 0)
        taikhoan = taikhoan – tien_rut;
else
        cout << "Không đủ tiền trong tài khoản";
```

Là tình huống mà kết quả của chương trình phụ thuộc vào sự điều phối của hệ thống

Tranh đoạt điều khiển và Phương pháp giải quyết

- ***Miền găng***: là đoạn mã nguồn truy cập tới vùng nhớ dùng chung

  …

  //bắt đầu miền găng

  If(taikhoan – tienrut >= 0) taikhoan = taikhoan – tienrut;

  Elsecout<<"Không đủ tiền trong tài khoản"

  //kết thúc miền găng

- ***Cách xử lý tranh chấp:***
  - Đảm bảo tính "độc quyền truy xuất" (mutual exclusion) cho miền găng
  - Sự phối hợp giữa các tiến trình phải thực hiện theo một kịch bản định trước

Bài toán đồng bộ hóa

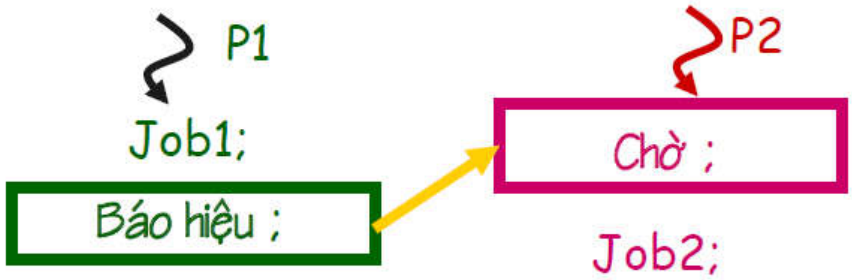
- Mô hình đảm bảo độc quyền truy xuất

Kiểm tra và dành quyền vào CS

CS:

Từ bỏ quyền sử dụng CS

- Mô hình tổ chức phối hợp giữa 2 tiến trình



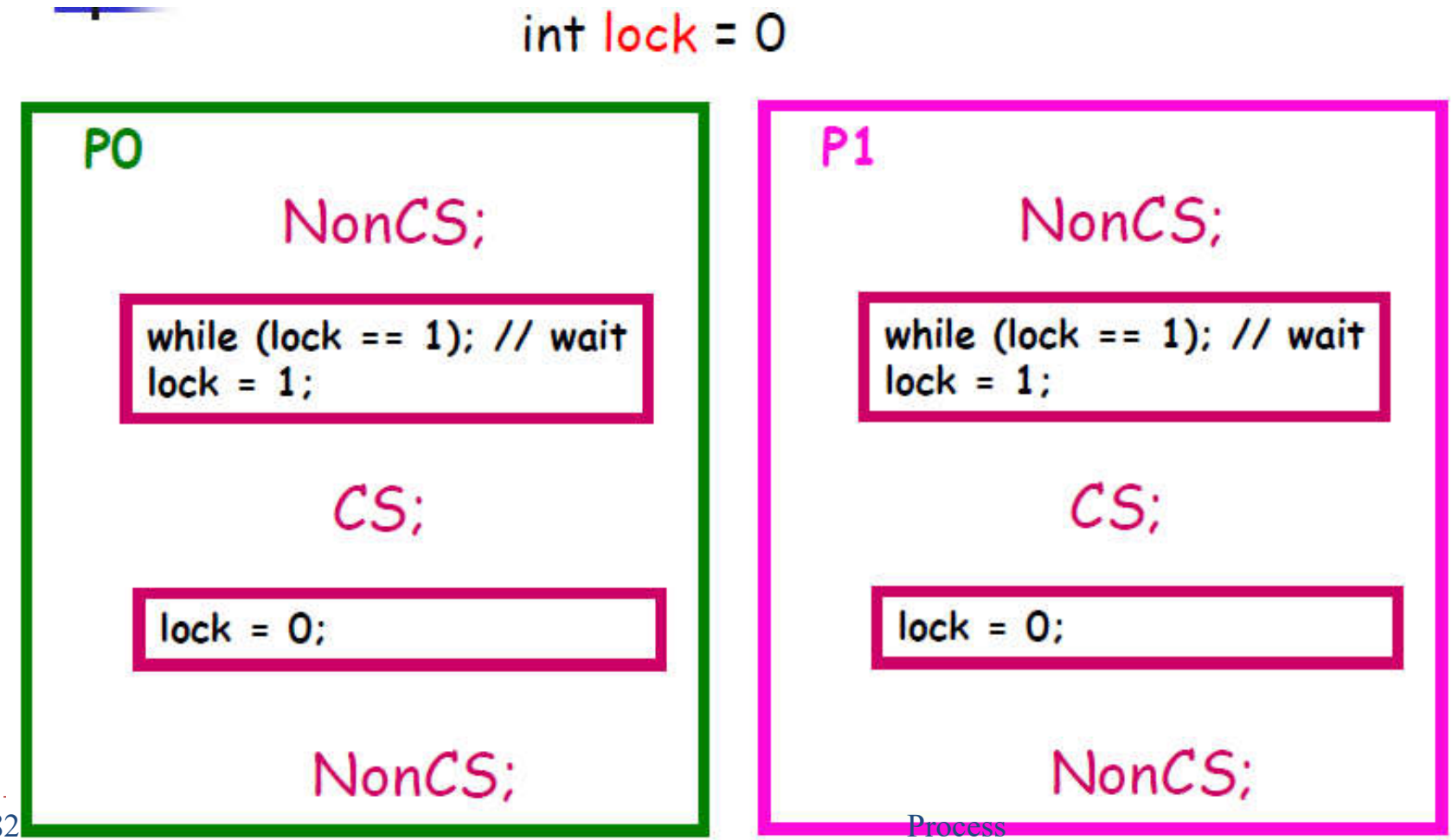
Bài toán đồng bộ hóa

- Các mô hình đưa ra cần đảm bảo
  1. Chỉ một tiến trình được thực thi trong miền găng tại một thời điểm
  2. Không có giả thiết về tốc độ CPU, số lượng CPU
  3. Không có tiến trình ở ngoài miền găng có thể khóa không cho tiến trình khác vào miền găng
  4. Không có tiến trình nào chờ đợi mãi mãi để vào miền găng

- Nhóm giải pháp Busy Waiting
  - Giải pháp Phần mềm
    - Sử dụng các biến cờ hiệu
    - Sử dụng việc kiểm tra luân phiên
    - Giải pháp của Peterson
  - Giải pháp Phần cứng
    - Cấm (Vô hiệu hóa) ngắt
    - Chỉ thị TSL (Test-and-Set)
- Nhóm giải pháp Sleep & Wakeup
  - Semaphore
  - Monitor
  - Message

Nhóm giải pháp Busy Waiting

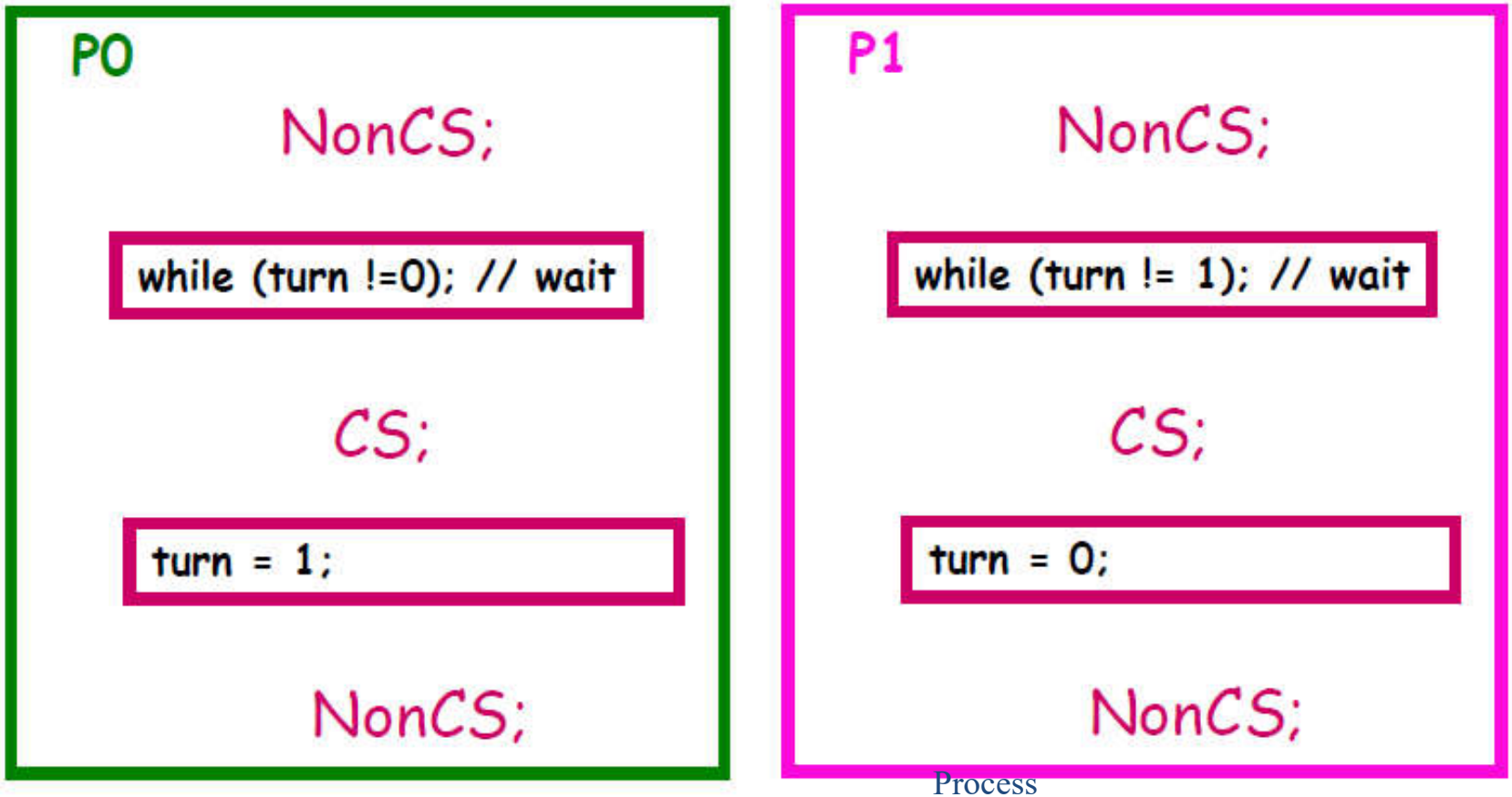
Dùng biến cờ hiệu



Nhóm giải pháp Busy Waiting

Kiểm tra luân phiên

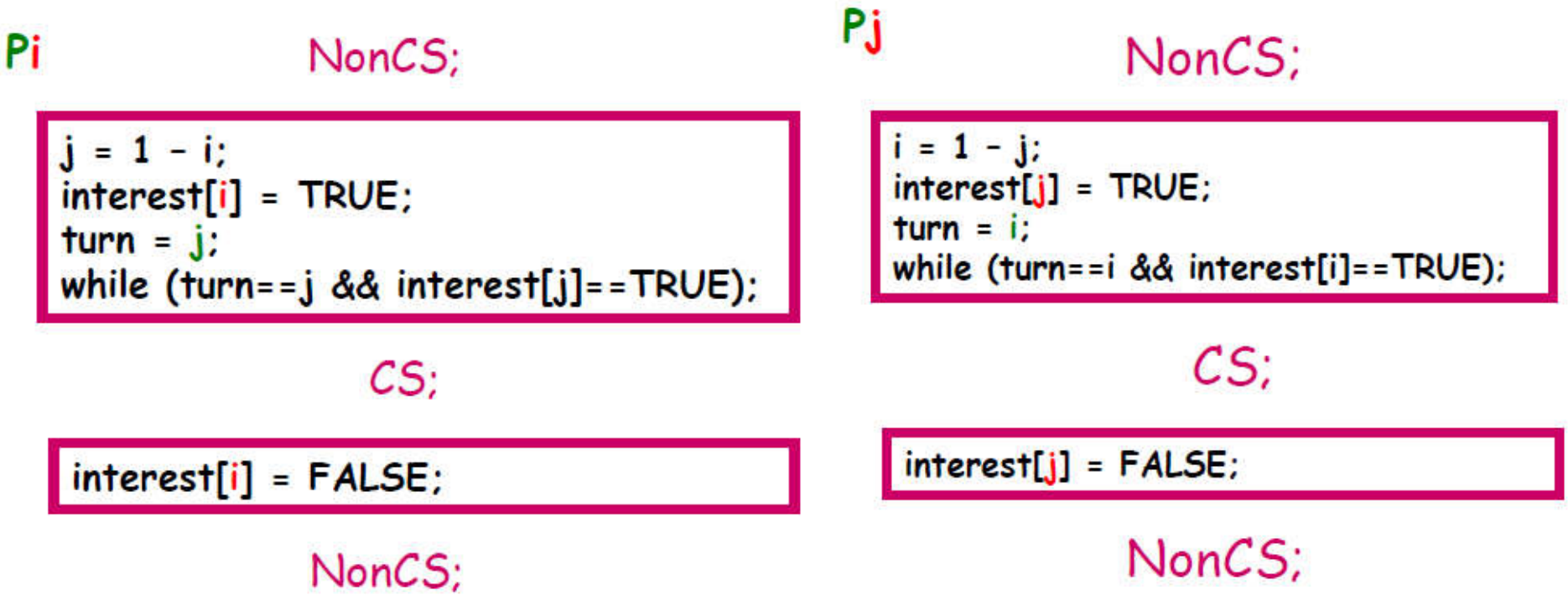
- Cả hai tiến trình đều ở ngoài miền găng, và *turn = 0*.



Nhóm giải pháp Busy Waiting

Giải pháp của Peterson

- Kết hợp ý tưởng của 1&2, các tiến trình chia sẻ
  - int turn; //đến phiên ai
  - int interest[2] = FALSE; //interest[i] = T:Pi muốn vào CS



Nhóm giải pháp Busy Waiting

Vô hiệu hóa ngắt

- Giải pháp đơn giản là khi một tiến trình vào miền găng thì nó vô hiệu hóa hết tất cả các ngắt
- → Liệu cho phép các tiến trình người dùng có thể vô hiệu hết ngắt là khả thi!

Nhóm giải pháp Busy Waiting

Chỉ thị TSL (Test-and-Set)

```
enter_region:
    TSL RX, LOCK | chép giá trị lock vào RX và gán lock = 1
    CMP RX, #0                    | so sánh với 0
    JNE enter_region              | jump nếu khác 0
    RET                           | vào miền găng

leave_region:
    MOVE   LOCK, #0               | gán lock = 0
    RET                           | trả về
```

- **Giải pháp Peterson và TSL đều đúng, tuy nhiên chiếm thời gian CPU vì vòng lặp kiểm tra liên tục**
- **→ giải pháp sleep and wakeup**

Ý tưởng chính

```
while (TRUE) {
if (busy){
        blocked = blocked + 1;
        sleep();
    }
else busy = 1;
critical-section ();
busy = 0;
if(blocked){
    wakeup(process);
    blocked = blocked - 1;
    }
noncritical-section ();
}
```

Nhóm giải pháp Sleep & Wakeup

Semaphore

- Semaphore
  - Semaphore **là một biến nguyên, ngoài thao tác khởi tạo chỉ có 2 thao tác là wait và signal**
  - **Down (Wait hay P)**: giảm S đi một, và kiểm tra xem process có bị blocked hay tiếp tục run?
  - **Up (Signal hay V):** tăng S lên một, và kiểm tra xem có 1 process đang blocked thì đánh thức nó
  - Hai thao tác này có tính nguyên tố, nghĩa là KHÔNG bị xen kẽ - bị ngắt giữa chừng

Nhóm giải pháp Sleep & Wakeup

Semaphore

- **Cài đặt Semaphore**
  - Semaphore có thể khởi tạo 1 hoặc 0

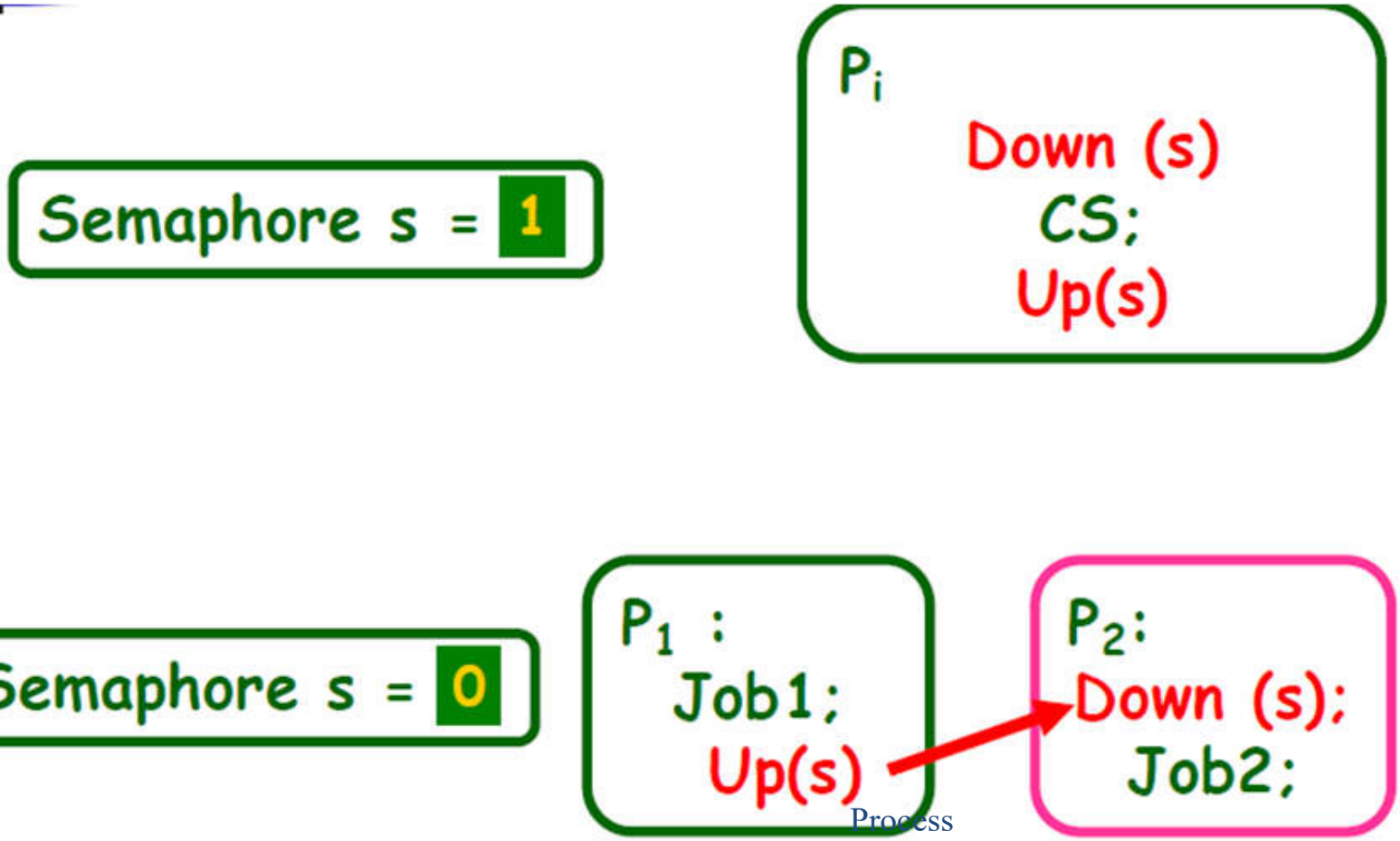
```
Down (S)
{
   S.value --;
   if S.value < 0
   {
      Add(P,S.L);
      Sleep();
   }
}
```

```
Up(S)
{
   S.value ++;
   if S.value ≤ 0
   {
      Remove(P,S.L);
      Wakeup(P);
   }
}
```

Nhóm giải pháp Sleep & Wakeup

Semaphore

- **Sử dụng Semaphore**
  - Tổ chức độc quyền truy xuất



Nhóm giải pháp Sleep & Wakeup

Semaphore

- Sử dụng Semaphore
  - **(1) Sử dụng semaphore để đảm bảo độc quyền truy suất :**
    - Sử dụng semaphore như **lock (khóa)**
    - Semaphore dạng này gọi là **Binary semaphore**

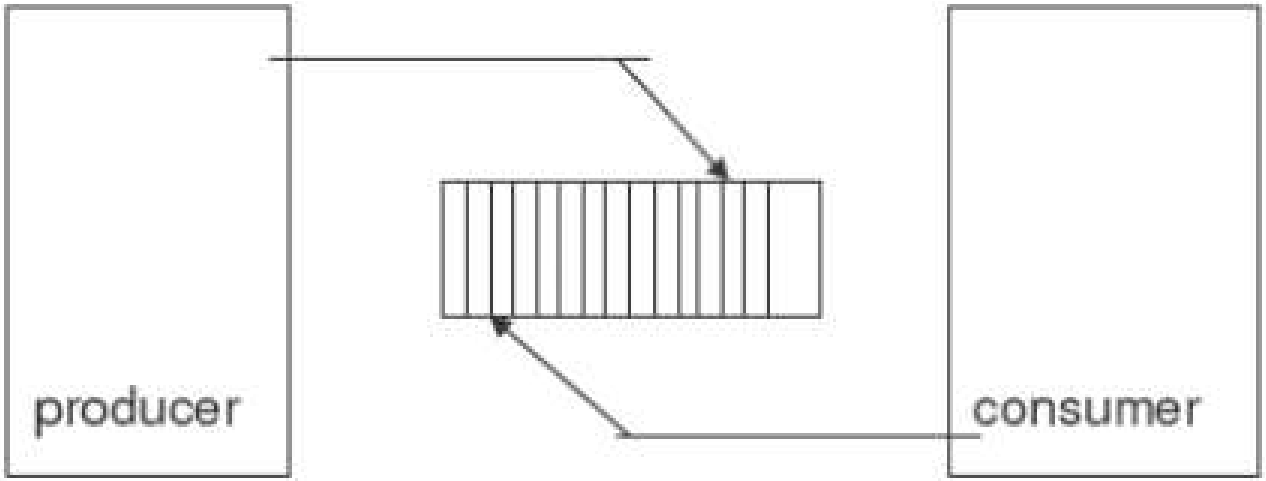
      (Khởi tạo **1**)
    - **Nguyên tắc** :
      - trước khi vào miền găng => khóa,
      - sau khi vào miền găng => mở khóa ,
      - kết quả, khi P1 nằm trong miền găng thì ko có tt nào được vào miền găng => độc quyền truy suất .

Nhóm giải pháp Sleep & Wakeup

Semaphore

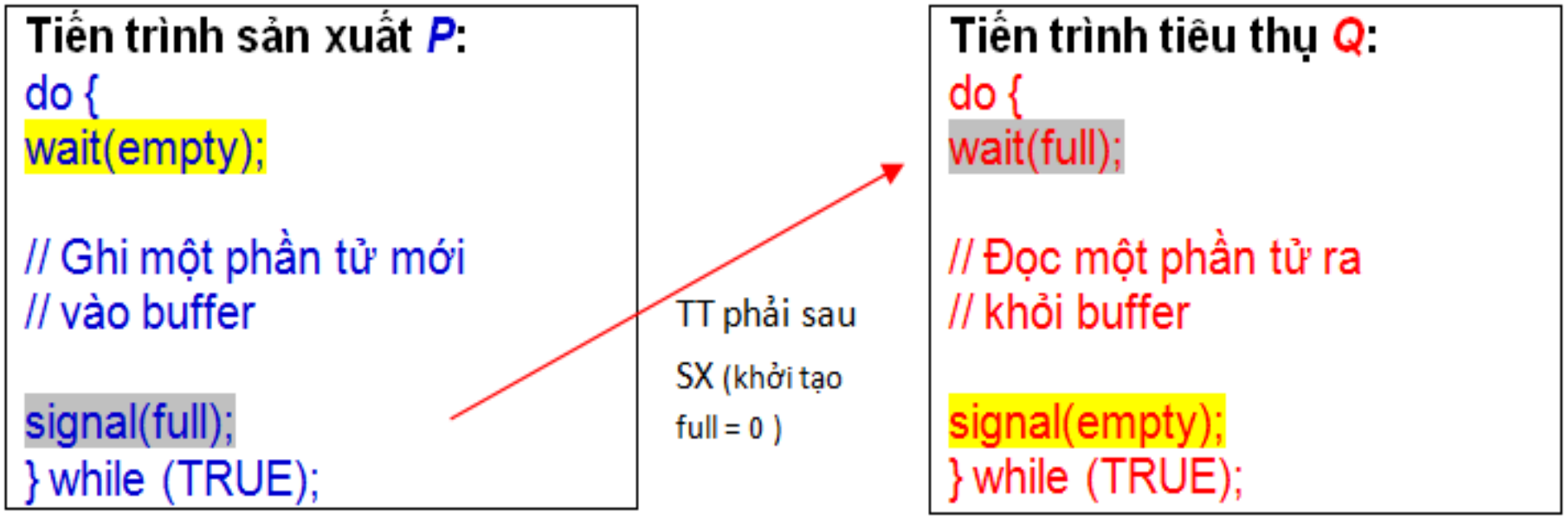
- Sử dụng Semaphore
  - **(2) Sử dụng semaphore để phối hợp hoạt động giữa các tiến trình :**
    - Hai tiến trình : P1 cần thực hiện job1(), P2 cần thực hiện job2(). Trình tự thực hiện là : job2() chỉ được thực hiện sau khi job1() hoàn tất.
    - Hai tiến trình cần chia sẻ một **semaphore s**, giá trị khởi tạo **là 0**

Bài toán sản xuất – tiêu thụ
(hay Vùng đệm có giới hạn)



Bài toán sản xuất – tiêu thụ
(hay Vùng đệm có giới hạn)

- **Cấu trúc 1 : đảm bảo đồng bộ (phối hợp) giữa 2 tiến trình**



đảm bảo đồng bộ giữa 2 tiến trình = đảm bảo 2 ràng buộc

empty=0 thì SX phải block
full=0 thì TT phải block

Bài toán sản xuất – tiêu thụ
(hay Vùng đệm có giới hạn)

- **Cấu trúc 2 : đảm bảo đồng bộ (phối hợp) giữa 2 tiến trình + độc quyền truy suất**

Tiến trình sản xuất *P*:

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

Tiến trình tiêu thụ *Q*:

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

CHƯƠNG 2:
QUẢN LÝ TIẾN TRÌNH – P4
DEADLOCK (TẮC NGHẼN)

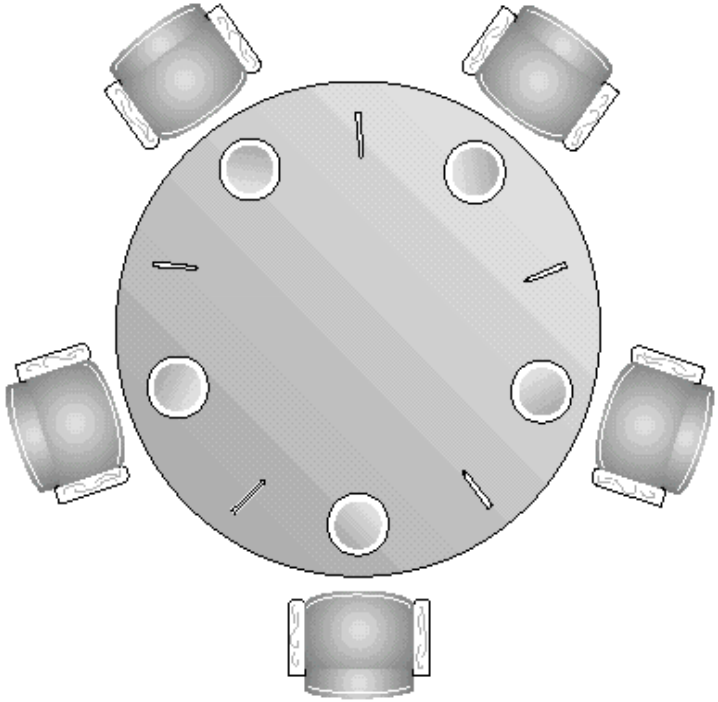
Process Management

Bài giảng 5 : Deadlock

- Định nghĩa Deadlock
- Mô hình hệ thống
- Điều kiện phát sinh Deadlock
- Xử lý Deadlock

Dining Philosophers

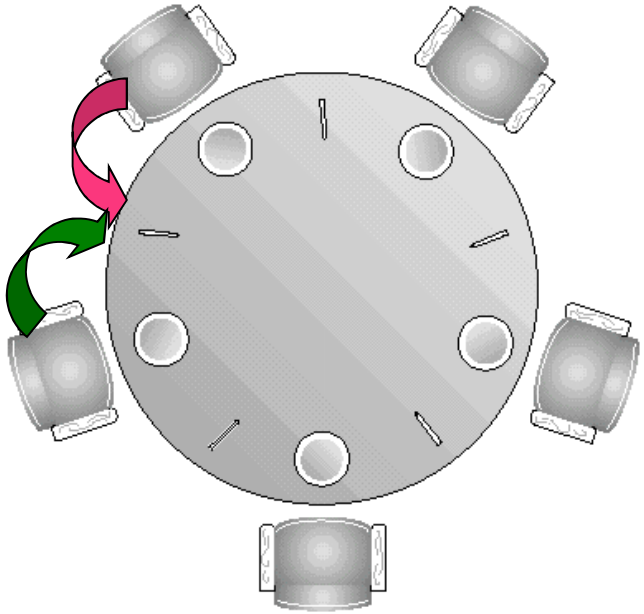
- Năm triết gia ngồi chung quanh bàn ăn món spaghetti (yum..yum)
  - Trên bàn có 5 cái nĩa được đặt giữa 5 cái đĩa (xem hình)
  - Để ăn món spaghetti mỗi người cần có 2 cái nĩa
- Triết gia thứ i:
  - Thinking...
  - Eating...

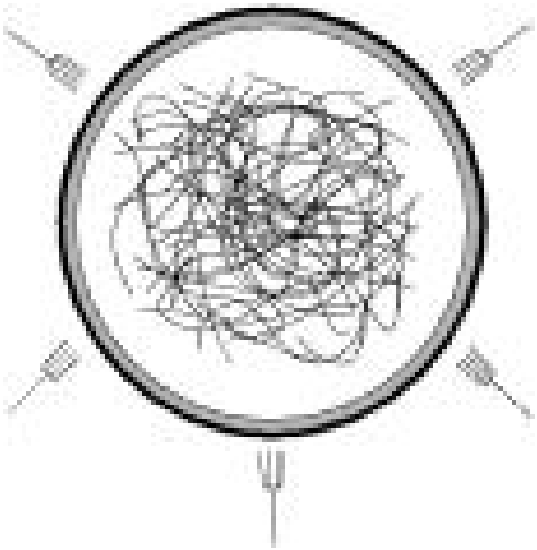


Chuyeän gì coù theå xaûy ra ?

Dining Philosophers : Tình huống nguy hiểm

- 2 triết gia "giành giật" cùng 1 cái nĩa
  - Tranh chấp
- Cần đồng bộ hoá hoạt động của các triết gia

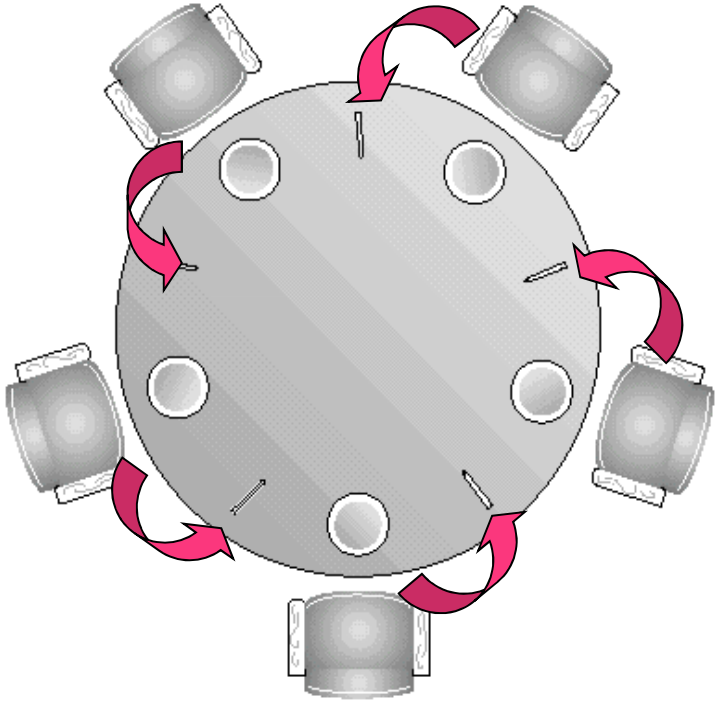




Dining Philosophers : Giải pháp đồng bộ

```
semaphore fork[5] = 1;

Philosopher (i)
{
  while(true)
   {
        down(fork[i]);
        down(fork[i+1 mod 5])
        eat;
        up(fork[i]);
        up(fork[i+1 mod 5]);
        think;
   }
```



Deadlock

Định nghĩa Deadlock

- Deadlock :
  - Chờ đợi một sự kiện không bao giờ xảy ra
  - Các tiến trình trong tập hợp chờ đợi lẫn nhau
- Starvation
  - Chờ đợi không có giới hạn một sự kiện mà chưa thấy xảy ra
- Deadlock kéo theo Starvation
  - Điều ngược lại không chắc

Mô hình hệ thống

- Hệ thống bao gồm một số xác định các loại tài nguyên sẽ được chia sẻ cho các tiến trình có nhu cầu
- Mỗi loại tài nguyên có thể có nhiều thể hiện
- Mỗi tiến trình sử dụng tài nguyên theo trình tự
  - Request : yêu cầu tài nguyên, nếu yêu cầu không được thoã mãn nay, tiến trình phải đợi
  - Use : sử dụng tài nguyên được cấp phát
  - Release : giải phóng tài nguyên

Các điều kiện xảy ra Deadlock

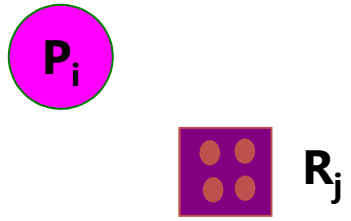
- Coffman, Elphick và Shoshani (1971) đã đưa ra 4 điều kiện cần có thể làm xuất hiện tắc nghẽn:
  - Mutual exclusion (loại trừ lẫn nhau): một tài nguyên bị chiếm giữ bởi 1 tiến trình, và khơng tiến trình nào khác cĩ thể sử dụng tài nguyên này
  - Wait for (hold and wait – giữ & chờ) : một tiến trình giữ ít nhất 1 tài nguyên và chờ một số tài nguyên khác rỗi để sử dụng. Các tài nguyên này đang bị một tiến trình khác chiếm giữ
  - No preemption (khơng cĩ đặc quyền): tài nguyên bị chiếm giữ chỉ cĩ thể rỗi khi tiến trình tự nguyện giải phĩng tài nguyên đĩ sau khi sử dụng xong
  - Circular wait (chờ vịng): Tồn tại một chu kỳ trong đồ thị cấp phát tài nguyên .
  - Hội đủ 4 điều kiện trên đây : Deadlock có thể xảy ra

Đồ thị cấp phát tài nguyên

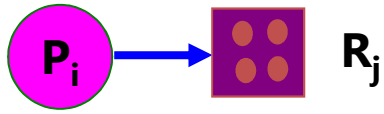
- 2 loại nodes:
  - P = {P1, P2, ..., Pn}, tập các tiến trình
  - R = {R1, R2, ..., Rm}, tập các loại tài nguyên
- Tiến trình yêu cầu tài nguyên :
  - Pi → Rj
- Tài nguyên được cấp phát cho tiến trình :
  - Rj → Pi

Ví dụ đồ thị cấp phát tài nguyên

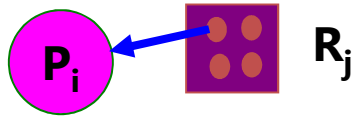
- Process
- Một loại tài nguyên với 4 thể hiện



- Pi yêu cầu 1 thể hiện của Rj

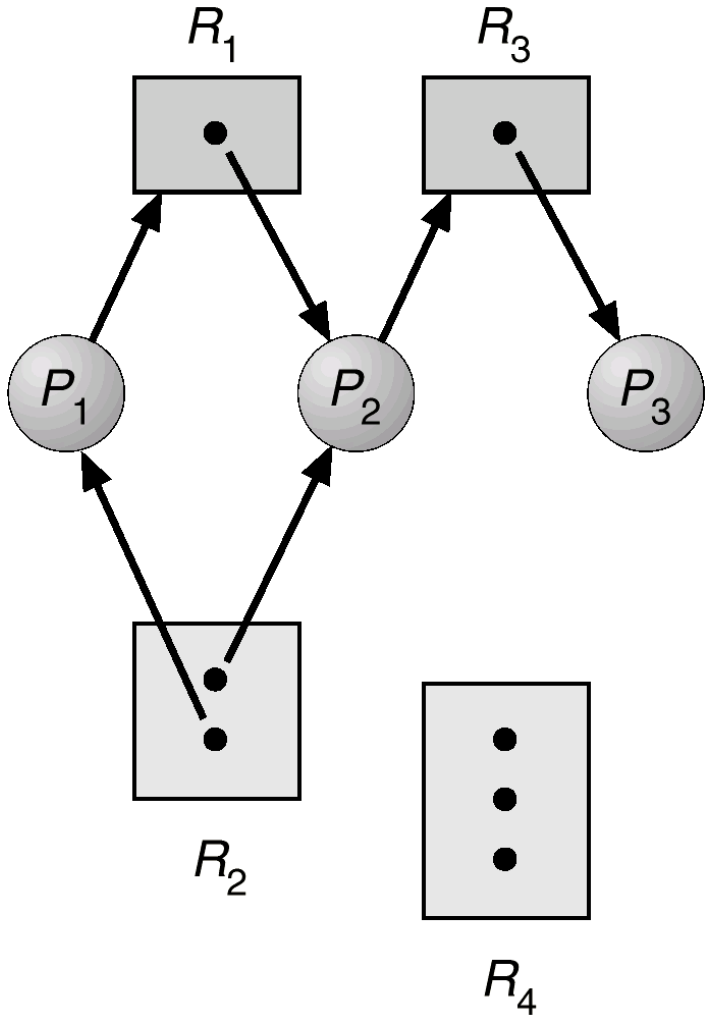


- Pi đang giữ 1 thể hiện của Rj



Ví dụ đồ thị cấp phát tài nguyên

Example:

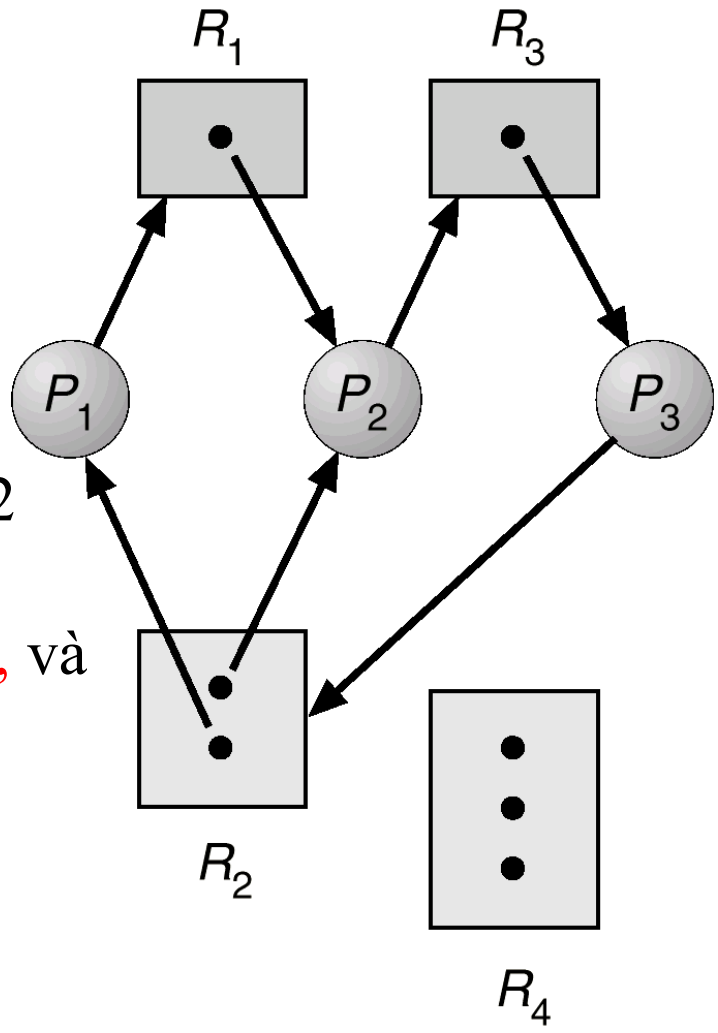


Nhận xét cơ bản

- Nếu đồ thị không có chu trình ⇒ no deadlock.
- Nếu đồ thị có 1 chu trình ⇒
  - Nếu mỗi tài nguyên chỉ có 1 thể hiện ⇒ deadlock.
  - Nếu mỗi tài nguyên có nhiều thể hiện ⇒ có thể có deadlock.

Ví dụ đồ thị cấp phát tài nguyên

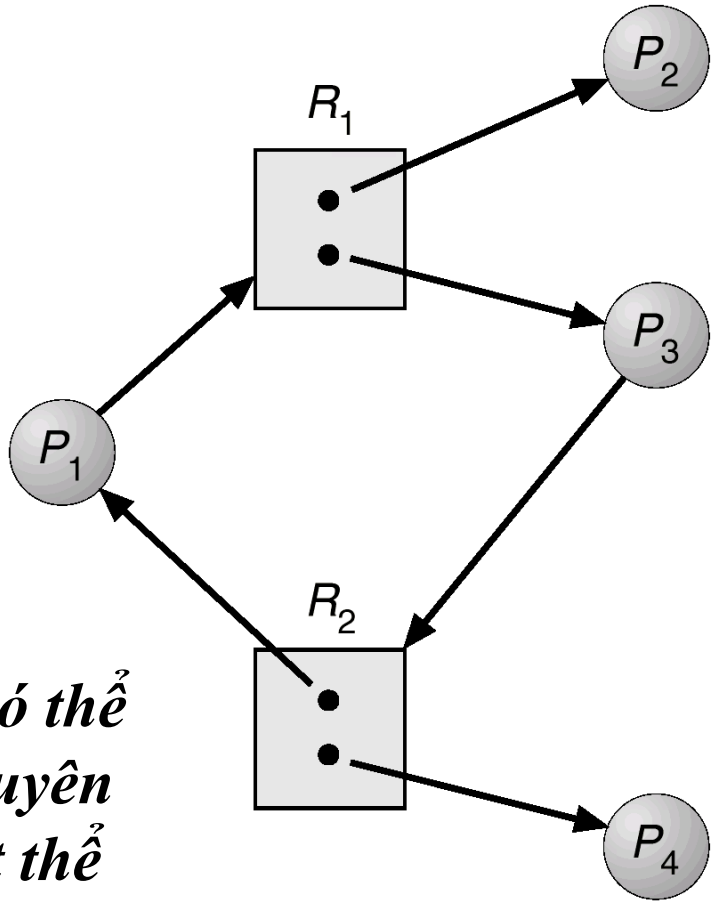
Deadlocked:



- Giả sử P3 yêu cầu 1 thể hiện của R2
- Khi đó có 2 chu trình xuất hiện:
  P1→R1→P2→R3→P3→R2→P1, và
  P2→R3→P3→R2→P2
- Khi đó các tiến trình P1,P2,P3 bị deadlock

Ví dụ đồ thị cấp phát tài nguyên

With Cycle but
No Deadlock:



- Chu trình xuất hiện:
  P1→R1→P3→R2→P1
- ***Deadlock không xảy ra vì P4 có thể giải phóng một thể hiện tài nguyên R2 và P3 sẽ được cấp phát một thể hiện của R2***

Deadlock Prevention

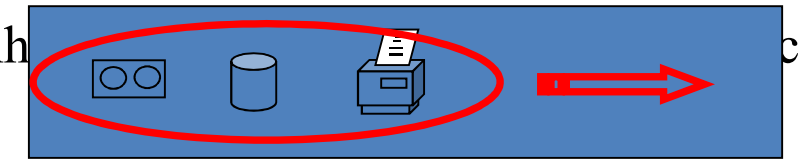
- Đảm bảo Deadlock không thể xảy ra
  - Tìm cách loại bỏ ít nhất 1 trong 4 điều kiện cần để xảy ra Deadlock
  - Nhắc lại 4 điều kiện cần
    - Mutual Exclusion
    - Hold and Wait
    - No preemption
    - Circular wait
- Để xem...

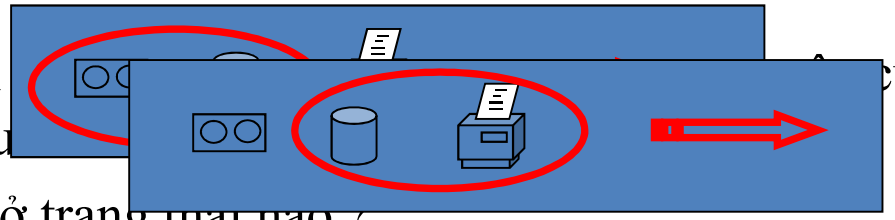
Deadlock Prevention

- Mutual Exclusion
  - Không cần đảm bảo độc quyền truy xuất tài nguyên ?
  - Thinking...
  - Thinking...
  - Thinking...
  - Với các shareable resources : dĩ nhiên !
  - Với các non-shareable resource : “Mission Impossible” ???
    - Làm gì : virtualize, spooling...
    - Rất hạn chế, đặc biệt đối với tài nguyên phần mềm
- **Kết luận: KHÓ/KHÔNG THỂ loại bỏ vì các hệ thống luôn có các tài nguyên không thể sử dụng chung được**

Deadlock Prevention

- Hold and Wait
  - Không cho vừa chiếm giữ vừa yêu cầu thêm tài nguyên
  - No Wait
    - Cấp cho tiến trình c khi bắt đầu xử lý
    - Làm sao biết ?
  - No Hold
    - Tiến trình không xin cũ, và chờ xin lại lần sau
    - "Trả lại" tài nguyên ở trạng thái nào ?
  - Sử dụng tài nguyên kém hiệu quả, có thể starvation
  - **Kết luận: Thật sự khó khả thi → không thể loại bỏ**





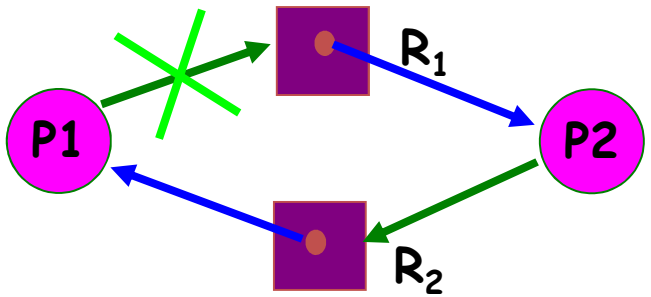
Deadlock Prevention

- No Preemption
  - Hệ điều hành chủ động thu hồi các tài nguyên của tiến trình blocked
    - Tài nguyên nào có thể thu hồi ?
      - CPU :OK
      - Printer : Hu hu
  - Các tài nguyên bị thu hồi sẽ được bổ sung vào danh sách tài nguyên tiến trình cần xin lại. Tiến trình chỉ có thể tiếp tục xử lý khi xin lại đủ các tài nguyên này (cũ và mới)
  - **Kết luận: thật sự khó loại bỏ hoàn toàn**

Deadlock Prevention

- Circular Wait
  - Không để xảy ra chu trình trong đồ thị cấp phát tài nguyên
    - Cấp phát tài nguyên theo 1 trật tự nhất định
  - Gọi R = {R1, R2,...,Rm} là tập các loại tài nguyên.
  - Các loại tài nguyên được phân cấp theo độ ưu tiên F(R) thuộc [1-N]
  - Ví dụ : F(đĩa) = 2, F(máy in) = 12
  - Các tiến trình khi yêu cầu tài nguyên phải tuân thủ quy định : khi tiến trình đang chiếm giữ tài nguyên Ri thì chỉ có thể yêu cầu các tài nguyên Rj nếu F(Rj) > F(Ri).

Trật tự nào ?



F(R1) = 1; F(R2) = 2;

Deadlock Prevention

- Đảm bảo Deadlock không thể xảy ra
  - Không thể loại bỏ ít nhất 1 trong 4 điều kiện cần để xảy ra Deadlock
- Quá khắt khe, không khả thi...

Deadlock Avoidance

- Là phương pháp sử dụng thêm thông tin về các phương thức yêu cầu cấp phát tài nguyên để đưa ra quyết định cấp phát tài nguyên sao cho deadlock 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 deadlock → kiểm tra trạng thái cấp phát tài nguyên một cách "động" để đảm bảo điều kiện vòng chờ (circular wait) không xảy ra
- Trạng thái cấp phát tài nguyên được xác định bởi số lượng tài nguyên rỗi, số lượng tài nguyên đã cấp và số lượng lớn nhất các yêu cầu cấp phát tài nguyên của các tiến trình

Deadlock Avoidance

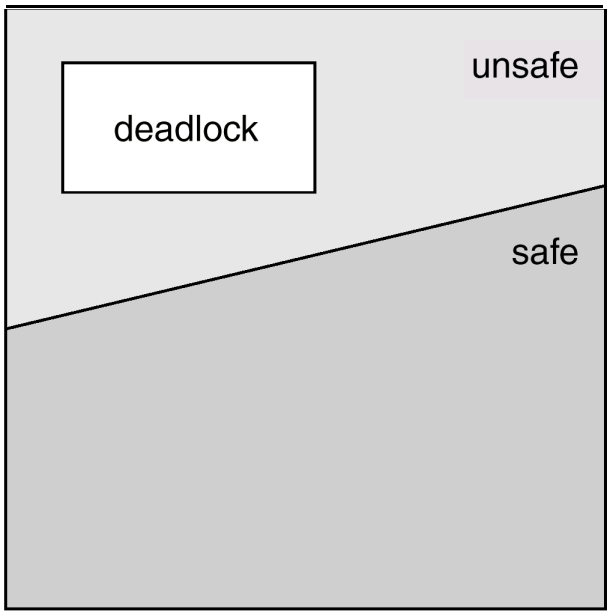
- Một số định nghĩa cơ bản
  - Trạng thái an toàn (Safe): hệ thống có thể thỏa mãn các nhu cầu tài nguyên (cho đến tối đa) của tất cả các tiến trình theo một thứ tự nào đó mà không dẫn đến tắc nghẽn.
    - Việc cấp phát tài nguyên không hình thành chu trình nào trong đồ thị cấp phát
  - Một chuỗi cấp phát an toàn: một thứ tự kết thúc các tiến trình <P1, P2,...,Pn> sao cho với mỗi tiến trình Pi nhu cầu tài nguyên của Pi có thể được thỏa mãn với các tài nguyên còn tự do của hệ thống, cộng với các tài nguyên đang bị chiếm giữ bởi các tiến trình Pj khác, với j<i.

Deadlock Avoidance

- Thay đổi chiến lược cấp phát tài nguyên để đảm bảo không dẫn dắt hệ thống vào tình trạng deadlock
  - Chỉ thỏa mãn yêu cầu tài nguyên của tiến trình khi trạng thái kết quả là an toàn
  - Đòi hỏi phải biết trước một số thông tin về nhu cầu sử dụng tài nguyên của tiến trình

Nhận xét

- Hệ thống ở safe state ⇒ không deadlocks.
- Nếu hệ thống ở unsafe state ⇒ có khả năng deadlock.
- Avoidance ⇒ bảo đảm hệ thống không bao giờ đi vào trạng thái unsafe.



Deadlock Avoidance : thông tin cần biết

- Giả sử hệ thống được mô tả với các thông tin sau :
  - int Available[NumResources];
    - Available[r]= số lượng các thể hiện còn tự do của tài nguyên r
  - int Max[NumProcs, NumResources];
    - Max[p,r]= nhu cầu tối đa của tiến trình p về tài nguyên r
  - int Allocation[NumProcs, NumResources];
    - Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p
  - int Need[NumProcs, NumResources];
    - Need[p,r] = Max[p,r] - Allocation[p,r]

Giải thuật cấp phát tài nguyên kiểu cũ

Pi xin k thể hiện của Rj

1 : if (k <= Need[i,j]) goto 2;

else Error();

2: if (k <= Available[j]) Allocate(i,j,k); //cấp cho Pi k thể hiện Rj

else MakeWait(Pi);

Deadlock Avoidance : Banker's Algorithm

Pi xin k thể hiện của Rj

```
1 :  if (k <= Need[i,j])   goto 2;
     else Error();
2:  if (k <= Available[j]) goto 3;
     else  MakeWait(Pi);
3:  /* Giả sử hệ thống đã cấp phát cho Pi các tài nguyên mà nó yêu cầu và
     cập nhật tình trạng hệ thống như sau:*/
          Available[j] = Available[j] - k;
          Allocation[i,j]= Allocation[i,j]+ k;
          Need[i,j] = Need[i,j] - k;
        Result = TestSafe();
        if (Result == Safe) Allocate(i,j,k); //cấp cho Pi k thể hiện Rj
          else  MakeWait(Pi);
```

TestSafe ()

1. Giả sử có các mảng

   int Work[NumResources] = Available[NumResources];

   int Finish[NumProcs] = false;

2. Tìm i sao cho

   Finish[i] == false & Need[i,j] <= Work[i,j], □j <=NumRes

   Nếu không có i như thế, đến bước 4.

3. Work = Work + Allocation[i];

   Finish[i] = true;

   Đến bước 2

4. Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái an toàn.

Ví dụ 1

Giả sử tình trạng hệ thống được mô tả như sau :

	Max			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	3	2	2	1	0	0	4	1	2
P2	6	1	3	2	1	1			
P3	3	1	4	2	1	1			
P4	4	2	2	0	0	2			

Ví dụ 1

Giả sử tình trạng hệ thống được mô tả như sau :

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	4	1	2
P2	4	0	2	2	1	1			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

3 – 1

Ví dụ 1

P2 yêu cầu 4 cho R1, 1 cho R3 : cân nhắc

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	2	2	2	1	0	0	0	1	1
P2	0	0	1	6	1	2			
P3	1	0	3	2	1	1			
P4	4	2	0	0	0	2			

Ví duï 1

P2 yêu cầu 4 cho R1, 1 cho R3 : cân nhắc

	Need			Allocation			Available		
	R1	R2	R3	R1	R2	R3	R1	R2	R3
P1	0	0	0	0	0	0	9	3	6
P2	0	0	0	0	0	0			
P3	0	0	0	0	0	0			
P4	0	0	0	0	0	0			

P2 yêu cầu 4 cho R1, 1 cho R3 : chấp nhận

Deadlock Detection

- Chấp nhận hệ thống rơi vào trạng thái deadlock
- Hệ thống nên cung cấp:
  - Một giải thuật kiểm tra và phát hiện deadlock có xảy ra trong hệ thống hay không
  - Một giải thuật để hiệu chỉnh, phục hồi hệ thống về trạng thái trước khi deadlock xảy ra.
- Cần tốn kém chi phí để :
  - Lưu trữ, cập nhật các thông tin cần thiết
  - Xử lý giải thuật phát hiện deadlock
  - Chấp nhận khả năng mất mát khi phục hồi.

Giải thuật phát hiện deadlock

- Cần sử dụng các cấu trúc dữ liệu sau :
  - int Available[NumResources];
    - // Available[r]= số lượng các thể hiện còn tự do của tài nguyên r
  - int Allocation[NumProcs, NumResources];
    - // Allocation[p,r] = số lượng tài nguyên r thực sự cấp phát cho p
  - int Request[NumProcs, NumResources];
    - // Request[p,r] = số lượng tài nguyên r tiến trình p yêu cầu thêm

Giải thuật phát hiện deadlock

1. Giả sử có các mảng

int Work[NumResources] = Available;

int Finish[NumProcs] ;

for (i = 0; i < NumProcs; i++)

Finish[i] = (Allocation[i] == 0);

2. Tìm i sao cho

Finish[i] == false & Request[i,j] <= Work[i,j], □j <=NumRes

Nếu không có i như thế, đến bước 4.

3. Work = Work + Allocation[i];

Finish[i] = true;

Đến bước 2

4. Nếu Finish[i] == true với mọi i, thì hệ thống ở trạng thái không có deadlock

Ngược lại, các tiến trình Pi, Finish[i] == false sẽ ở trong tình trạng deadlock

Sử dụng giải thuật phát hiện deadlock

- Khi nào, và mức độ thường xuyên cần kích hoạt giải thuật phát hiện deadlock ? Phụ thuộc vào
  - Tần suất xảy ra deadlock?
  - Số lượng các tiến trình liên quan, cần "rolled back"?
    - 1 cho mỗi chu tri(nh rời nhau
- Một cách cẩn thận tối đa, kích hoạt giải thuật phát hiện mỗi khi có một yêu cầu cấp phát bị từ chối

- Tiết kiệm chi phí : kích hoạt giải thuật phát hiện deadlock sau những chu kỳ định trước
  - Khuyết điểm ?

Deadlock Recovery: Hủy bỏ tiến trình

- Hủy tất cả các tiến trình liên quan deadlock
  - Thiệt hại đáng kể...
- Hủy từng tiến trình liên quan cho đến khi giải toả được chu trình deadlock
  - Tốn chi phí thực hiện giải thuật phát hiện deadlock
  - Bắt đầu từ tiến trình nào ? Sau đó là ai ?...
    - Priority of the process.
    - How long process has computed, and how much longer to completion.
    - Resources the process has used.
    - Resources process needs to complete.
    - How many processes will need to be terminated.
    - Is process interactive or batch?
  - Sao cho thiệt hại ít nhất

Deadlock Recovery: Thu hồi tài nguyên

- Lần lượt thu hồi một số tài nguyên từ các tiến trình và cấp phát các tài nguyên này cho những tiến trình khác đến khi giải toả được chu trình deadlock
- 3 vấn đề cần quan tâm :
  - Chọn lựa "nạn nhân" : – Thu hồi tài nguyên nào ? Của ai ?
    - Tiến trình nắm giữ nhiều tài nguyên
    - Tiến trình có thời gian đã xử lý không cao
    - ...?
  - Rollback : quay lại trạng thái an toàn (nào?), khởi động lại tiến trình từ trạng thái an toàn đó.
    - Phải lưu vết hoạt động của hệ thống ->Chi phí cao
  - Starvation: có thể một tiến trình nào đó luôn luôn bị chọn làm "nạn nhân", không bao giờ có đủ tài nguyên để tiến triển xử lý.

Deadlock Ignorance : Ostrich's algorithm

- Hệ thống : "Deadlock hả ? What, what, what ???"
  - Don't see
  - Don't know
  - Don't care
- Có thể chấp nhận không ?
  - Cân nhắc giữa tần suất xảy ra deadlock và chi phí giải quyết deadlock
  - Là giải pháp của hầu hết HĐH hiện nay