

HỆ ĐIỀU HÀNH  
CHƯƠNG 4. QUẢN LÝ TIẾN TRÌNH,  
ĐỒNG BỘ HÓA TIẾN TRÌNH & TẮC NGHẼN



## MỤC TIÊU

Giới thiệu các **khái niệm về Tiến trình** và những thao tác cơ bản trong **Quản lý Tiến trình** như tạo, định thời và kết thúc tiến trình. Các phương thức **Giao tiếp liên tiến trình** và vấn đề **Tắc nghẽn** của tiến trình cũng sẽ được trình bày.

# NỘI DUNG

TỔNG QUAN VỀ TIẾN TRÌNH

GIAO TIẾP LIÊN TIẾN TRÌNH

ĐỊNH THỜI TIẾN TRÌNH

CÁC GIẢI THUẬT ĐỊNH THỜI

ĐỒNG BỘ HÓA TIẾN TRÌNH

TẮC NGHẼN (DEADLOCK)

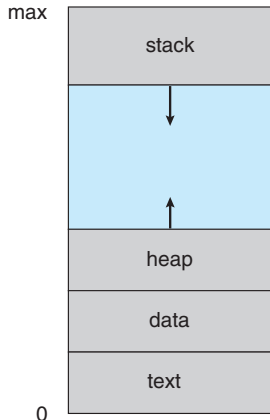
Tổng quan về tiến trình

# KHÁI NIỆM TIỀN TRÌNH

- ▶ **Tiền trình** là thể hiện (instance) của một **chương trình máy tính** trong bộ nhớ, đang thực thi hoặc chờ thực thi.
- ▶ Mỗi tiến trình thường được gán 1 **số định danh tiến trình** (process identifier, **pid**), dùng để định danh các tiến trình.
- ▶ Một tiến trình bao gồm:
  - ▶ **Mã lệnh** chương trình (program code)
  - ▶ **Bộ đếm** chương trình (program counter) và các **thanh ghi** của CPU
  - ▶ **Ngăn xếp** (stack)
  - ▶ Phần **dữ liệu** (data section)
  - ▶ Có thể gồm phần **bộ nhớ cấp phát động** khi tiến trình thực thi (heap)

# CHƯƠNG TRÌNH & TIẾN TRÌNH

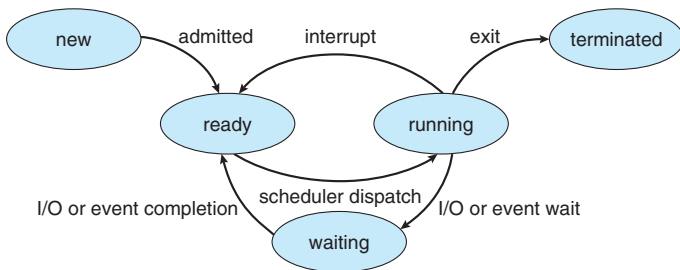
- ▶ **Chương trình** là một thực thể **bị động**, được lưu trữ trên **đĩa**.
- ▶ **Tiến trình** là một thực thể **chủ động**, lưu trú trên **bộ nhớ chính**.
- ▶ Khi một chương trình được **kích hoạt** (nhấp chuột, CLI, ...), một thể hiện của chương trình sẽ được  **nạp lên bộ nhớ, tạo ra 1 tiến trình**.
- ▶ **Một chương trình** có thể có **vài tiến trình** trong bộ nhớ.



# TRẠNG THÁI CỦA TIẾN TRÌNH (PROCESS STATE)

- ▶ Một tiến trình có thể có một trong các **trạng thái** sau:
  - ▶ **new**: tiến trình đang được khởi tạo.
  - ▶ **running**: các chỉ thị của tiến trình đang được thực thi.
  - ▶ **waiting**: tiến trình đang chờ đợi một sự kiện nào đó xảy ra (hoàn thành I/O, tín hiệu từ tiến trình khác, ...).
  - ▶ **ready**: tiến trình sẵn sàng để thực thi (đang đợi để được sử dụng CPU).
  - ▶ **terminated**: tiến trình đã kết thúc.

# SƠ ĐỒ CHUYỂN TRẠNG THÁI CỦA TIẾN TRÌNH





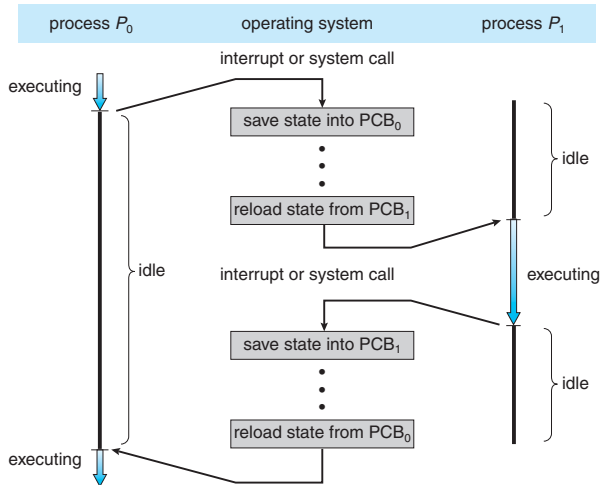
# KHOẢNG ĐIỀU KHIỂN TIỀN TRÌNH (PCB)

- ▶ Chứa thông tin của tiến trình trong Hệ điều hành:
  - ▶ **Trạng thái** của quá trình: ready, running, ...
  - ▶ **Bộ đếm** chương trình: chỉ thị kế tiếp sẽ được thực thi
  - ▶ Các **thanh ghi**: phụ thuộc vào k/trúc máy tính
  - ▶ Thông tin về **định thời sử dụng CPU**
  - ▶ Thông tin về **quản lý bộ nhớ**
  - ▶ Thông tin về **chi phí**: t/gian sử dụng CPU, pid, ...
  - ▶ Thông tin về trạng thái **nhập/xuất**: các thiết bị đang được cấp phát, danh sách tập tin đang mở, ...

process state
process number
program counter
registers
memory limits
list of open files
...

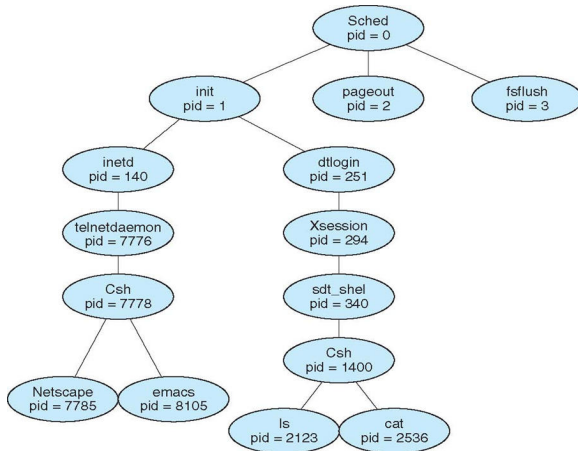
# CHUYỂN CPU GIỮA CÁC TIẾN TRÌNH

- ▶ **PCB** là nơi lưu giữ trạng thái của tiến trình
- ▶ Trạng thái của tiến trình phải được lưu trữ vào PCB khi một **interrupt** xuất hiện, nhằm cho phép tiến trình có thể tiếp tục chính xác về sau.
- ▶ Tác vụ chuyển CPU còn được gọi là **chuyển ngữ cảnh** (context switch).



# TẠO TIỀN TRÌNH

- ▶ Một tiến trình (**cha**) có thể tạo những tiến trình khác (**con**) ...
- ▶ Quan hệ “**cha-con**” của các tiến trình tạo nên **cây tiến trình**.



# KẾT THÚC TIỀN TRÌNH

- ▶ T/trình thực thi câu lệnh cuối cùng và yêu cầu HĐH xóa nó (`exit()`)
  - ▶ Truyền dữ liệu từ tiến trình con lên tiến trình cha (`wait()`).
  - ▶ Thu hồi tài nguyên đã được cấp phát cho tiến trình.
- ▶ Tiến trình con kết thúc trước khi t/trình cha gọi `wait()`  $\Rightarrow$  zombie
- ▶ Tiến trình con còn thực thi khi t/trình cha đã kết thúc  $\Rightarrow$  orphan
- ▶ Tiến trình cha có thể kết thúc tiến trình con (`abort()`):
  - ▶ Tiến trình con đã có vượt quá số tài nguyên được cấp.
  - ▶ Công việc giao cho tiến trình con làm nay không còn cần thiết nữa.
  - ▶ Tiến trình cha đang thoát: một vài HĐH không cho phép orphan.

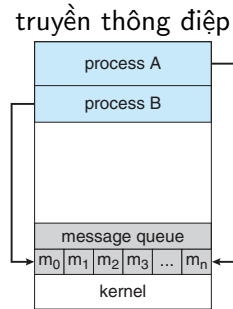
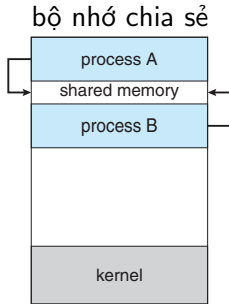
Giao tiếp liên tiến trình

# HỢP TÁC TIỀN TRÌNH (COOPERATING PROCESS)

- ▶ **Tiến trình độc lập**: không thể ảnh hưởng hoặc không bị ảnh hưởng bởi sự thực thi của quá trình khác.
- ▶ **Hợp tác tiến trình**: có thể ảnh hưởng hoặc bị ảnh hưởng bởi sự thực thi của quá trình khác.
- ▶ **Thuận lợi** của sự hợp tác quá trình:
  - ▶ Chia sẻ thông tin
  - ▶ Gia tăng tốc độ tính toán (nếu máy có nhiều CPU)
  - ▶ Module hóa
  - ▶ Tiện dụng

# GIAO TIẾP LIÊN TIẾN TRÌNH

- Các tiến trình muốn **trao đổi dữ liệu** với nhau cần sử dụng **cơ chế giao tiếp liên tiến trình** (interprocess communication, IPC):



## BỘ NHỚ CHIA SẺ (SHARED-MEMORY)

- ▶ Một vùng đệm (buffer) được dùng để chia sẻ dữ liệu giữa các t/trình:
  - ▶ kích thước không giới hạn (unbounded buffer): tiến trình đọc có thể chờ, tiến trình ghi không bao giờ chờ.
  - ▶ kích thước có giới hạn (bounded buffer): cả tiến trình đọc và ghi có thể chờ.
- ▶ Ví dụ kinh điển “Nhà sản xuất – Người tiêu thụ”: tiến trình Nhà sản xuất sinh dữ liệu, được sử dụng bởi tiến trình Người tiêu thụ.
  - ▶ Tạo 1 vùng nhớ đệm (buffer) chung.
  - ▶ Tiến trình Nhà sản xuất ghi dữ liệu lên buffer.
  - ▶ Tiến trình Người tiêu thụ lấy dữ liệu từ buffer.



## TẠO VÙNG ĐỆM (BUFFERING)

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];

int in_item = 0;
int out_item = 0;
```

## NHÀ SẢN XUẤT (PRODUCER)

```
item next_produced;

while (true) {
    /* produce an item in next produced */

    while (((in_item + 1) % BUFFER_SIZE) == out_item)
        ; /* do nothing */

    buffer[in_item] = next_produced;
    in_item = (in_item + 1) % BUFFER_SIZE;
    counter++;
}
```

## NGƯỜI TIÊU DÙNG (CONSUMER)

```
item next_consumed;

while(true){
    while(in_item == out_item)
        ; /* do nothing */

    next_consumed = buffer[out_item];
    out_item = (out_item + 1) % BUFFER_SIZE;

    /* consume the item in next consumed */
}
```

# TRUYỀN THÔNG ĐIỆN (MESSAGE PASSING)

- ▶ Giao tiếp giữa các tiến trình **không cần dùng bộ nhớ chia sẻ**  
⇒ hữu ích trong môi trường phân tán, giao tiếp qua mạng.
- ▶ Cần hai thao tác: **send(msg)** và **receive(msg)**.
- ▶ Tiến trình P và Q muốn giao tiếp với nhau:
  - ▶ **Tạo một nối kết giao tiếp** (communication link)
  - ▶ **Trao đổi thông điệp** thông qua send/receive
- ▶ **Phương thức cài đặt** nối kết giao tiếp (mức luận lý):
  - ▶ Giao tiếp trực tiếp hay gián tiếp
  - ▶ Đồng bộ hay bất đồng bộ
  - ▶ Kích thước thông điệp cố định hay biến đổi

# GIAO TIẾP TRỰC TIẾP (DIRECT COMMUNICATION)

- ▶ Các quá trình phải được **đặt tên** rõ ràng:
  - ▶  $\text{Send}(P, \text{msg})$ : gửi thông điệp tới quá trình P.
  - ▶  $\text{Receive}(Q, \text{msg})$ : nhận thông điệp từ quá trình Q.
- ▶ **Các thuộc tính** của nối kết giao tiếp:
  - ▶ Các nối kết được **thiết lập tự động**.
  - ▶ Một nối kết kết hợp với **chính xác một cặp** quá trình.
  - ▶ Giữa mỗi cặp quá trình tồn tại **chính xác một nối kết**.
  - ▶ Nối kết có thể **một hướng**, nhưng thường là **hai hướng**.
  - ▶ Giao tiếp **bất đối xứng**:  $\text{Send}(P, \text{msg})$ ,  $\text{Receive}(id, \text{msg})$ .

# GIAO TIẾP GIÁN TIẾP (INDIRECT COMMUNICATION)

- ▶ Các thông điệp được gửi và nhận thông qua **mailbox** hay **port**.
  - ▶ Mỗi mailbox có một **định danh** (id) duy nhất.
  - ▶ Các quá trình chỉ có thể giao tiếp nếu chúng **dùng chung mailbox**.
    - ▶ `Send/Receive(A, msg)`: gửi/nhận thông điệp tới/từ hộp thư A.
- ▶ **Các thuộc tính** của nối kết gián tiếp:
  - ▶ Nối kết chỉ được thiết lập nếu các quá trình **chia sẻ một hộp thư chung**.
  - ▶ Một nối kết **có thể kết hợp với nhiều quá trình**.
  - ▶ Mỗi cặp quá trình **có thể dùng chung nhiều nối kết** giao tiếp.
  - ▶ Nối kết có thể **một hướng hay hai hướng**.

# CÁC TÁC VỤ TRONG GIAO TIẾP GIÁN TIẾP

- ▶ **Các tác vụ cơ bản:** tạo mailbox mới, gửi và nhận thông điệp qua mailbox, và xóa mailbox.
- ▶ **Chia sẻ mailbox:**
  - ▶ Các tiến trình có thể chia sẻ cùng 1 mailbox.
  - ▶ **Vấn đề:** nếu 1 tiến trình gửi thì tiến trình nào sẽ nhận?
- ▶ **Giải pháp cho việc chia sẻ mailbox:**
  - ▶ Một liên kết chỉ tương ứng với 2 tiến trình.
  - ▶ Chỉ cho phép 1 tiến trình thực hiện thao tác nhận tại 1 thời điểm.
  - ▶ HDH chỉ định tiến trình nhận (1 tiến trình), và thông báo cho tiến trình gửi biết người nhận.

# ĐỒNG BỘ HÓA (SYNCHRONISATION)

- ▶ Truyền thông điệp có thể **chặn** (blocking) hay **không chặn** (non-blocking).
- ▶ **Blocking** được xem là **đồng bộ** (synchronous):
  - ▶ Blocking send: tiến trình gửi chờ cho đến khi thông điệp được nhận.
  - ▶ Blocking receive : tiến trình nhận chờ cho đến khi thông điệp sẵn sàng .
- ▶ **Non-blocking** được xem là **bất đồng bộ** (asynchronous):
  - ▶ Non-blocking send: gửi thông điệp và tiếp tục thực hiện công việc khác.
  - ▶ Non-blocking receive: nhận một thông điệp hay rỗng.



## TẠO VÙNG ĐỆM (BUFFERING)

- ▶ Vùng đệm dùng để chứa các thông điệp của 1 nối kết.
- ▶ Ba cách cài đặt:
  - ▶ Sức chứa là 0 (zero capacity): tiến trình gửi bị chặn đến khi thông điệp được nhận (no buffering!?).
  - ▶ Sức chứa giới hạn (bounded capacity): kích thước vùng đệm giới hạn n thông điệp. Tiến trình gửi bị chặn khi vùng đệm bị đầy.
  - ▶ Sức chứa không giới hạn (unbounded capacity): kích thước không giới hạn. Tiến trình gửi không bao giờ bị chặn.

Định thời tiến trình

- ▶ **Định thời biểu CPU** là một chức năng cơ bản và quan trọng của các HĐH đa chương.
- ▶ **Chức năng: phân bổ thời gian/thời điểm sử dụng CPU** cho các tiến trình trong hệ thống, nhằm:
  - ▶ **tăng hiệu năng** (CPU utilisation) sử dụng CPU
  - ▶ **giảm thời gian đáp ứng** (response time) của hệ thống
- ▶ **Ý tưởng cơ bản:** phân bổ thời gian rảnh rỗi của CPU (khi chờ đợi tiến trình đang thực thi thực hiện các thao tác nhập xuất) cho các tiến trình khác trong hệ thống.

# CHU KỲ CPU-I/O (CPU-I/O BURST)

## ▶ Chu kỳ CPU-I/O:

- ▶ Sự thực thi của tiến trình bao gồm nhiều chu kỳ CPU-I/O.
- ▶ Một chu kỳ CPU-I/O bao gồm **chu kỳ thực thi CPU** (CPU burst) và **chu kỳ chờ đợi vào/ra** (I/O burst).

## ▶ Sự phân bổ sử dụng CPU:

- ▶ Chương trình **hướng nhập xuất** (I/O-bound) thường có nhiều chu kỳ CPU ngắn.
- ▶ Chương trình **hướng xử lý** (CPU-bound) thường có nhiều chu kỳ CPU dài.

## VÍ DỤ VỀ CHU KỲ CPU-I/O

...

**load store**  
**add store**  
**read** from file

CPU burst

*wait for I/O*

I/O burst

**store increment**  
**index**  
**write** to file

CPU burst

*wait for I/O*

I/O burst

**load store**  
**add store**  
**read** from file

CPU burst

*wait for I/O*

I/O burst

...

# BỘ ĐỊNH THỜI CPU (CPU SCHEDULER)

- ▶ Còn gọi là **bộ định thời ngắn kỳ**, chọn một trong các tiến trình trong hàng đợi sẵn sàng và cấp phát CPU cho nó thực thi.
- ▶ Quyết định định thời xảy ra khi một tiến trình:
  1. chuyển từ trạng thái đang chạy sang trạng thái chờ đợi
  2. chuyển từ trạng thái đang chạy sang trạng thái sẵn sàng
  3. chuyển từ trạng thái chờ đợi sang trạng thái sẵn sàng
  4. kết thúc

# ĐỊNH THỜI TRƯNG DỤNG & KHÔNG TRƯNG DỤNG

- ▶ **Định thời không trưng dụng** (nonpreemptive scheduling):
  - ▶ Tiến trình được phân CPU có quyền sử dụng CPU **đến khi sử dụng xong** (k/thúc hoặc chuyển sang trạng thái chờ, như trường hợp 1 và 4).
- ▶ **Định thời trưng dụng** (preemptive scheduling):
  - ▶ Bộ định thời có thể thu hồi CPU của tiến trình **bất kỳ lúc nào** để phân cho tiến trình khác (trường hợp 2 và 3).
  - ▶ **Phức tạp** hơn định thời không trưng dụng vì nó phải giải quyết:
    - ▶ sự cạnh tranh dữ liệu giữa các tiến trình.
    - ▶ sự trưng dụng khi tiến trình đang thực thi trong chế độ kernel.
    - ▶ dàn xếp giữa sự trưng dụng và xử lý các ngắt của hệ thống.

## BỘ ĐIỀU PHỐI (DISPATCHER)

- ▶ Có nhiệm vụ thực thi việc trao quyền sử dụng CPU cho tiến trình được cấp phát CPU bởi bộ định thời.
- ▶ Bao gồm các tác vụ:
  - ▶ Chuyển ngữ cảnh
  - ▶ Chuyển sang chế độ người dùng
  - ▶ Nhảy tới vị trí thích hợp trong chương trình người dùng để khởi động lại chương trình đó.
- ▶ **Độ trễ điều phối** (dispatcher latency): thời gian dispatcher cần để ngưng một tiến trình và khởi động một tiến trình khác.



## TIÊU CHÍ CHO VIỆC ĐỊNH THỜI

1. **Hiệu suất sử dụng CPU:** tỷ lệ giữa thời gian CPU được sử dụng trên tổng thời gian hoạt động của hệ thống.
2. **Thời gian đáp ứng (response time):** lượng thời gian từ lúc một yêu cầu được đệ trình cho đến khi **bắt đầu được đáp ứng**.
3. **Thời gian chờ đợi (waiting time):** tổng thời gian 1 tiến trình nằm trong hàng đợi sẵn sàng (ready queue).
4. **Thời gian xoay vòng (turnaround time):** tổng thời gian để thực thi một t/trình, bao gồm các khoảng t/gian: thực thi, chờ I/O, chờ trong ready queue (= t/điểm kết thúc – t/điểm bắt đầu vào ready queue).
5. **Thông lượng (throughput):** số lượng tiến trình hoàn thành trên một đơn vị thời gian.

# TỐI ƯU HÓA CÁC TIÊU CHÍ

Các giải thuật định thời được đánh giá thông qua khả năng **tối ưu hóa** các **tiêu chí định thời** của nó:

1. Hiệu suất sử dụng CPU: càng lớn càng tốt
2. Thông lượng: càng lớn càng tốt
3. Thời gian xoay vòng: càng nhỏ càng tốt
4. Thời gian chờ đợi: càng nhỏ càng tốt
5. Thời gian đáp ứng: càng nhỏ càng tốt

## CÁC GIẢI THUẬT ĐỊNH THỜI

1. First-come, first-served (FCFS): đến trước được phục vụ trước.
2. Shortest-job-rirst (SJF): công việc ngắn nhất trước.
3. Priority: dựa trên độ ưu tiên.
4. Round-robin (RR): xoay vòng.
5. Multilevel scheduling: hàng đợi đa cấp.
6. Multilevel feedback-queue scheduling: hàng đợi phản hồi đa cấp.

# FIRST-COME, FIRST SERVED (FCFS)

- ▶ Là giải thuật định thời đơn giản nhất, dựa trên nguyên tắc **đến trước, được phục vụ trước**.
- ▶ **Cài đặt**: phương pháp đơn giản nhất là dùng **hàng đợi FIFO**.
- ▶ **Ưu điểm**: cài đặt dễ dàng, đơn giản và dễ hiểu.
- ▶ **Nhược điểm**:
  - ▶ **Thời gian chờ đợi trung bình** thường là dài.
  - ▶ Không thích hợp cho hệ thống phân chia thời gian do đây là giải thuật **định thời không trưng dụng** (nonpreemptive).

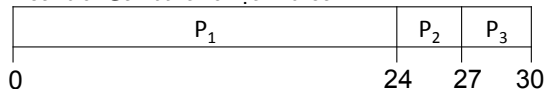
## FCFS – Ví Dụ 1

- Cho các tiến trình với thời gian thực thi và thứ tự xuất hiện như sau:

Process	TG sử dụng CPU	Thứ tự xuất hiện
$P_1$	24	1
$P_2$	3	2
$P_3$	3	3

(g/s tgian xuất hiện là  $t = 0$ )

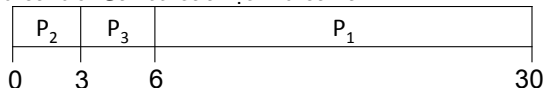
- Biểu đồ Gantt cho lịch biểu:



- Thời gian chờ đợi:  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Thời gian chờ đợi trung bình:  $(0 + 24 + 27)/3 = 17$

## FCFS – Ví Dụ 2

- Giả sử các tiến trình trong ví dụ 1 xuất hiện theo thứ tự  $P_2, P_3, P_1$ ; biểu đồ Gantt của lịch biểu là:



- Thời gian chờ đợi:  $P_1 = 6, P_2 = 0, P_3 = 3$
- Thời gian chờ đợi trung bình:  $(6 + 0 + 3)/3 = 3$   
 $\Rightarrow$  tốt hơn nhiều so với ví dụ 1 (17)
- Tình trạng thời gian chờ đợi dài do tiến trình ngắn nằm sau tiến trình dài được gọi là “hiệu ứng nổi đuôi” (convoy effect).

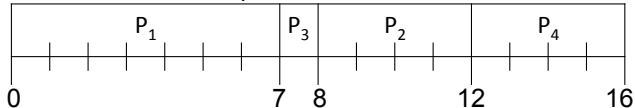
## SHORTEST-JOB-FIRST (SJF)

- ▶ **Ý tưởng cơ bản:** phân phối CPU cho tiến trình nào có thời gian thực thi CPU (CPU burst) kế tiếp nhỏ nhất (*shortest-next-CPU-burst* alg.)
- ▶ Mỗi tiến trình sẽ được gán 1 độ dài thời gian của lần sử dụng CPU kế tiếp (dự đoán).
- ▶ Có 2 cách tiếp cận cho việc phân bổ CPU:
  - ▶ **Không trưng dụng:** tiến trình được giao CPU sẽ chiếm giữ CPU đến khi nó thực thi xong CPU burst.
  - ▶ **Trưng dụng:** nếu 1 tiến trình mới đến có CPU burst ngắn hơn thời gian thực thi còn lại của tiến trình đang thực thi, CPU sẽ được lấy lại để giao cho tiến trình mới (*shortest-remaining-time-first* algorithm, SRTF)
- ▶ SJF cho thời gian chờ đợi trung bình tối ưu (ngắn nhất).

## SJF KHÔNG TRUNG DỤNG – VÍ DỤ

Process	TG sử dụng CPU kế tiếp	Thời gian xuất hiện
P <sub>1</sub>	7	0
P <sub>2</sub>	4	2
P <sub>3</sub>	1	4
P <sub>4</sub>	4	5

- ▶ Biểu đồ Gantt cho lịch biểu:



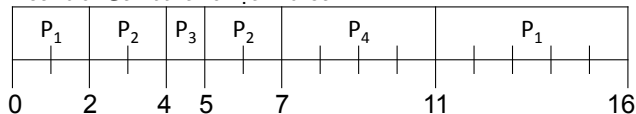
- ▶ Thời gian chờ đợi trung bình:  $(0 + 6 + 3 + 7)/4 = 4$



## SJF TRƯNG DỤNG – VÍ DỤ

Process	TG sử dụng CPU kế tiếp	Thời gian xuất hiện
P <sub>1</sub>	7	0
P <sub>2</sub>	4	2
P <sub>3</sub>	1	4
P <sub>4</sub>	4	5

- ▶ Biểu đồ Gantt cho lịch biểu:



- ▶ Thời gian chờ đợi trung bình:  $(9 + 1 + 0 + 2)/4 = 3$

## THỜI GIAN SỬ DỤNG CPU LẦN KẾ TIẾP

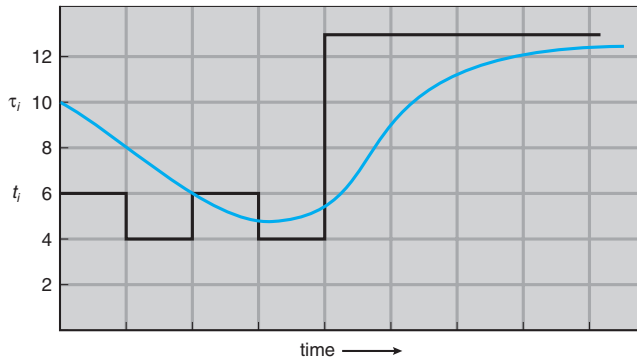
- ▶ Chỉ có thể **ước lượng**, dựa vào **lịch sử** của những lần **sử dụng CPU** trước đó.
- ▶ **Thời gian sử dụng CPU kế tiếp** (công thức trung bình mũ):

$$T_{n+1} = \alpha t_n + (1 - \alpha) T_n$$

- ▶  $T_{n+1}$ : **ước lượng** thời gian sử dụng CPU lần  $n + 1$
- ▶  $t_n$ : thời gian sử dụng CPU **thực tế** lần thứ  $n$
- ▶  $\alpha \in [0, 1]$ : hệ số trung bình mũ, dùng để điều chỉnh **trọng số** cho các **giá trị lịch sử** (thông thường được gán giá trị  $1/2$ )

# THỜI GIAN SỬ DỤNG CPU LẦN KẾ TIẾP

- Ví dụ: ước lượng thời gian sử dụng CPU lần kế tiếp, với  $\alpha = 1/2$ ,  $T_0 = 10$



CPU burst ( $t_i$ )	6	4	6	4	13	13	13	...	
"guess" ( $\tau_i$ )	10	8	6	6	5	9	11	12	...

## TÙY BIẾN HỆ SỐ TRUNG BÌNH MŨ

▶  $\alpha = 0 \Rightarrow T_{n+1} = T_n = \dots = T_0$

$\Rightarrow$  không tính đến lịch sử: tình trạng/sự thực thi “hiện thời” được coi như nhất thời, không có ý nghĩa.

▶  $\alpha = 1 \Rightarrow T_{n+1} = t_n$

$\Rightarrow$  chỉ tính đến thời gian sử dụng CPU thực tế gần nhất.

▶  $\alpha = 1/2$ : các giá trị lịch sử thực tế và dự đoán có trọng số tương đương.

▶ Nếu mở rộng công thức, ta có:

$$T_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \dots + (1 - \alpha)^j \alpha t_{n-j} + \dots (1 - \alpha)^{n+1} T_0$$

▶ Vì  $\alpha$  và  $(1 - \alpha) \leq 1$ , trọng số của giá trị lịch sử càng xa thì càng nhỏ.

# GIẢI THUẬT ĐỊNH THỜI CÓ ƯU TIÊN (PRIORITY)

## ► Ý tưởng:

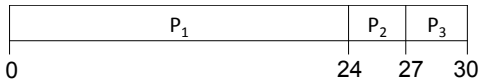
- Mỗi tiến trình sẽ được gán một **chỉ số ưu tiên** (priority number, int) .
- CPU sẽ được cấp phát cho tiến trình có **chỉ số ưu tiên cao nhất**, thông thường là nhỏ nhất.
- **SJF là trường hợp đặc biệt** của giải thuật này, trong đó thời gian thực thi CPU kế tiếp đóng vai trò là chỉ số ưu tiên.
- Có thể cài đặt theo phương pháp **trưng dụng** hay **không trưng dụng**.
- Có thể xảy ra **tình trạng “chết đói”** (starvation): các tiến trình độ ưu tiên thấp không bao giờ được thực thi.  
⇒ Giải pháp: dùng **sự “lão hóa”** (aging) – các tiến trình đang chờ đợi trong hệ thống sẽ được **tăng dần độ ưu tiên** theo thời gian chờ đợi.

## Ví Dụ

Process	Thời điểm xuất hiện	Độ ưu tiên	Thời gian xử lý
$P_1$	0	3	24
$P_2$	1	1	3
$P_3$	2	2	3

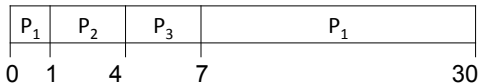
- **Không trưng dụng:**  $T/\text{gian chờ đợi trung bình} = (0 + 23 + 25)/3 = 16$

Biểu đồ Gantt:



- **Trương dụng:**  $T/\text{gian chờ đợi trung bình} = (6 + 0 + 2)/3 = 2.7$

Biểu đồ Gantt:

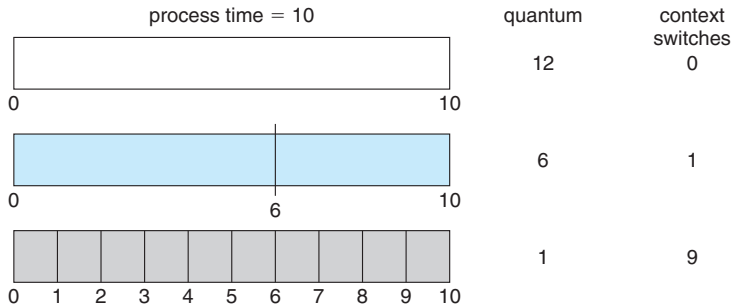


# GIẢI THUẬT ĐỊNH THỜI LUÂN PHIÊN

- ▶ Bộ điều phối **cấp phát xoay vòng** cho mỗi tiến trình trong hàng đợi sẵn sàng một đơn vị thời gian, gọi là định mức thời gian (*time quantum*, thường khoảng 10–100ms).
  - ▶ Sau khi sử dụng hết t/gian được cấp, **CPU bị thu hồi** để cấp cho tiến trình khác, tiến trình bị thu hồi CPU sẽ chuyển vào **hàng đợi sẵn sàng**.
  - ▶ **Bộ đếm thời gian (timer)** sẽ phát ra các ngắt sau mỗi **định mức thời gian** để **xoay vòng cấp phát CPU**.
- ▶ Nếu hàng đợi sẵn sàng có  $n$  tiến trình, định mức thời gian là  $q$ :
  - ▶ mỗi tiến trình sẽ nhận được  $1/n$  **tổng thời gian CPU**, trong đó thời gian mỗi lần sử dụng tối đa là  $q$
  - ▶ không có tiến trình nào **chờ đợi** quá lượng thời gian  $(n - 1) \times q$

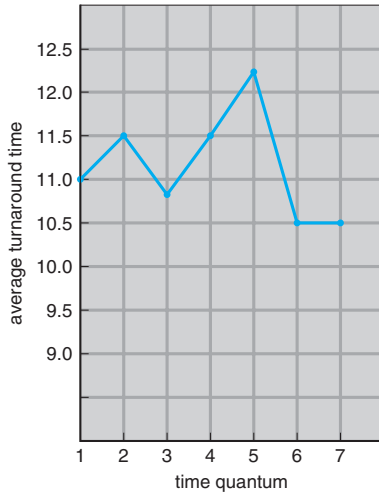
## CÁC TÙY BIẾN & HIỆU NĂNG

- ▶  **$q$  lớn:** RR trở thành giải thuật FCFS (FIFO)
- ▶  **$q$  nhỏ:**  $q$  phải đủ lớn so với thời gian chuyển ngữ cảnh, nếu không, **hao phí chuyển ngữ cảnh** sẽ rất cao.





# CÁC TÙY BIẾN & HIỆU NĂNG



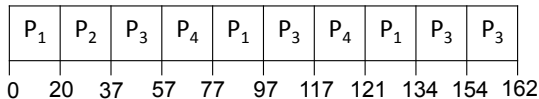
process	time
$P_1$	6
$P_2$	3
$P_3$	1
$P_4$	7

## Ví Dụ

Process	Thời gian sử dụng CPU
P <sub>1</sub>	53
P <sub>2</sub>	17
P <sub>3</sub>	68
P <sub>4</sub>	24

- ▶ Với time quantum = 20

- ▶ Biểu đồ Gantt:



- ▶ Thông thường, RR có thời gian chờ đợi trung bình lớn hơn SJF, nhưng đảm bảo thời gian đáp ứng tốt hơn.

# GIẢI THUẬT HÀNG ĐỢI ĐA CẤP

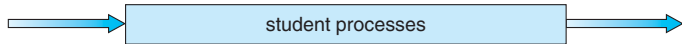
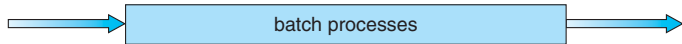
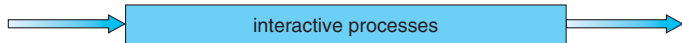
- ▶ Chia hàng đợi s/sàng ra thành nhiều hàng đợi với độ **ưu tiên khác nhau**, ví dụ:
  - ▶ **foreground**: tương tác, cần ưu tiên cao hơn.
  - ▶ **background**: bó, cần ít ưu tiên hơn.
- ▶ Các tiến trình sẽ được **phân phối** vào các hàng đợi dựa trên các đặc tính như loại tiến trình (foreground/background), độ ưu tiên, ...
- ▶ Mỗi hàng đợi sẽ được áp dụng một giải thuật định thời riêng, tùy vào tính chất của hàng đợi; ví dụ:
  - ▶ **foreground (tương tác)**: cần thời gian đáp ứng nhanh hơn  $\Rightarrow$  RR
  - ▶ **background (bó)**: có thể đáp ứng chậm hơn  $\Rightarrow$  FCFS

# CHIẾN LƯỢC ĐỊNH THỜI GIỮA CÁC HÀNG ĐỢI

- ▶ Định thời với **độ ưu tiên cố định**:
  - ▶ Phục vụ tất cả các t/trình trong hàng đợi ưu tiên cao (vd: foreground) rồi mới đến hàng đợi có độ ưu tiên thấp hơn (vd: background).
  - ▶ Có khả năng dẫn đến tình trạng “chết đói” (starvation) CPU.
- ▶ Định thời với **phân chia thời gian** (time-slice):
  - ▶ Mỗi hàng đợi sẽ nhận được một khoảng thời gian nào đó của CPU để định thời cho các tiến trình nằm trong đó.
  - ▶ Ví dụ: 80% cho foreground với RR, và 20% cho background với FCFS.

# VÍ DỤ HÀNG ĐỢI ĐA CẤP

highest priority



lowest priority

# GIẢI THUẬT HÀNG ĐỢI PHẢN HỒI ĐA CẤP

- ▶ Hàng đợi sẵn sàng cũng tổ chức giống như trong giải thuật Hàng đợi đa cấp.
- ▶ Một tiến trình có thể di chuyển giữa các hàng đợi khác nhau.
- ▶ Cơ chế định thời có thể được cài đặt theo cách:
  - ▶ Nếu tiến trình dùng quá nhiều thời gian CPU, nó sẽ được di chuyển vào hàng đợi có độ ưu tiên thấp hơn.
  - ▶ Nếu tiến trình đã chờ quá lâu trong 1 hàng đợi với độ ưu tiên thấp, nó sẽ được chuyển sang hàng đợi có độ ưu tiên cao hơn (cơ chế “sự lão hóa”).

## THAM SỐ CỦA BỘ ĐỊNH THỜI

- ▶ Bộ định thời đa cấp có phản hồi có thể được định nghĩa bằng những tham số sau:
  - ▶ Số lượng hàng đợi
  - ▶ Giải thuật định thời cho từng hàng đợi
  - ▶ Phương thức dùng để quyết định khi nào thì nâng cấp một tiến trình.
  - ▶ Phương thức dùng để quyết định khi nào thì hạ cấp một tiến trình
  - ▶ Phương thức dùng để quyết định là nên đặt tiến trình vào hàng đợi nào khi tiến trình này cần được phục vụ.

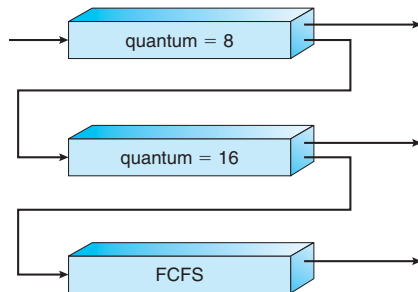
# VÍ DỤ VỀ HÀNG ĐỢI PHẢN HỒI ĐA CẤP

## ► Các hàng đợi:

- $Q_0$ : FCFS + quantum-time 8ms
- $Q_1$ : FCFS + quantum-time 16ms
- $Q_2$ : original FCFS

## ► Định thời:

- Một tiến trình mới  $P$  sẽ được phân vào  $Q_0$  với giải thuật định thời FCFS. Khi có được CPU, nó sẽ được sử dụng tối đa 8ms.
- Nếu sau 8ms,  $P$  chưa hoàn thành thì CPU sẽ bị thu hồi để phân phối cho tiến trình khác và  $P$  sẽ được chuyển sang  $Q_1$ .
- Tại  $Q_1$ , việc định thời diễn ra tương tự, với quantum-time là 16ms. Nếu  $P$  vẫn chưa hoàn thành thì nó sẽ được chuyển sang  $Q_2$  với giải thuật FCFS.





# ĐỊNH THỜI TRONG WINDOWS

- ▶ Đơn vị cấp phát CPU trong các HĐH hiện nay là **luồng (thread)** thay vì tiến trình.
- ▶ HĐH Windows dùng giải thuật định thời dựa trên **độ ưu tiên** (32 mức) và **định mức thời gian** (RR), sử dụng chiến lược **trưng dụng**.
- ▶ Mỗi độ ưu tiên sẽ có 1 hàng đợi riêng.
- ▶ Một luồng được chọn thực thi bởi bộ điều phối sẽ t/thi cho đến khi:
  - ▶ **bị trưng dụng** bởi luồng có mức ưu tiên cao hơn, hoặc
  - ▶ kết thúc (terminate), hoặc
  - ▶ hết định mức thời gian (time quantum), hoặc
  - ▶ thực hiện lời gọi **I/O**.

# ĐỊNH THỜI TRONG WINDOWS

- ▶ Độ và nhóm ưu tiên của các luồng được chia như sau:

	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

- ▶ Ngoài ra, một số cơ chế để **nâng/hạ độ ưu tiên** cho cá luồng cũng được sử dụng (tương tự ý tưởng của g/thuật hàng đợi phản hồi đa cấp)

# CHỌN LỰA TIỀN TRÌNH TRONG WINDOWS

- ▶ Bộ định thời duyệt qua các hàng đợi theo **độ ưu tiên từ cao đến thấp**.
- ▶ Nếu không có tiến trình nào sẵn sàng, bộ định thời khởi động 1 tiến trình đặc biệt gọi là **idle process**.
- ▶ **Độ ưu tiên** của t/trình bị trưng dụng **có thể bị thay đổi**.
  - ▶ Nếu sử dụng **hết time quantum**: độ ưu tiên giảm, nhưng không dưới mức “normal”.
  - ▶ Chuyển **từ trạng thái chờ** (waiting): độ ưu tiên tăng, mức độ tăng phụ thuộc vào t/trình đã chờ cái gì: đĩa – tăng nhiều, I/O – tăng ít hơn.

Đồng bộ hóa tiến trình

## CẠNH TRANH VÀ SỰ NHẤT QUAN DỮ LIỆU

- ▶ Các tiến trình thực thi đồng thời, chia sẻ dữ liệu dùng chung có thể dẫn đến tình trạng không nhất quán (inconsistency) của dữ liệu.
- ▶ Nhất quán = đúng đắn và chính xác; tùy thuộc vào ngữ cảnh, giao dịch.
- ▶ Có 2 lý do chính để thực hiện đồng thời (cạnh tranh) các tiến trình:
  - ▶ Tăng hiệu suất sử dụng tài nguyên hệ thống.
  - ▶ Giảm thời gian đáp ứng trung bình của hệ thống.
- ▶ Việc duy trì sự nhất quán của dữ liệu yêu cầu một cơ chế để đảm bảo sự thực thi một cách có thứ tự của các tiến trình có hợp tác với nhau.

## VÍ DỤ 1 – GIAO DỊCH CẠNH TRANH

► Cho hai giao dịch:

- T1: A **mua hàng** trị giá **50\$** của P  
(50\$:  $A \rightarrow P$ )
- T2: B **mua hàng** trị giá **100\$** của P  
(100\$:  $B \rightarrow P$ )

► **Khởi tạo** ban đầu:  $A=500$ ;  $B=500$ ;  $P=1000$

T1	T2
R(A)	R(B)
$A = A - 50$	$B = B - 100$
W(A)	W(B)
R(P)	R(P)
$P = P + 50$	$P = P + 100$
W(P)	W(P)

► Yêu cầu về tính **nhất quán**:  $(A + B + P)$  không đổi; cụ thể hơn:

- giá trị  $A, B, P$  **sau khi thực hiện** T1, T2 là:  $A=450$ ;  $B=400$ ;  $P=1150$

► **Nhận xét**:

- Nếu thực hiện **tuần tự**  $T1 \rightarrow T2$  hoặc  $T2 \rightarrow T1$ , dữ liệu sẽ **nhất quán**.
- Nếu thực hiện **cạnh tranh** (đồng thời), dữ liệu sẽ **nhất quán???**

## VÍ DỤ 1 – GIAO DỊCH CẠNH TRANH

T1	T2	A/B	P
R(A) A = A - 50 W(A)	R(B) B = B - 100 W(B)	450  400	
R(P) P = P + 50	R(P)		1050
W(P)	P = P + 100 W(P)		1100
<b>Schedule 1</b>			

T1	T2	A/B	P
R(A) A = A - 50	R(B) B = B - 100 W(B)	  400 450	
W(A) R(P) P = P + 50 W(P)	R(P) P = P + 100 W(P)		1050
			1150
<b>Schedule 2</b>			

## TÌNH TRẠNG “TRANH ĐUA” (RACE CONDITION)

- ▶ Là tình trạng mà nhiều tiến trình cùng truy cập và thay đổi lên dữ liệu được chia sẻ, và giá trị cuối cùng của dữ liệu chia sẻ phụ thuộc vào tiến trình hoàn thành sau cùng.
  - ▶ giá trị của  $P$  trong ví dụ 1
  - ▶ hoặc giá trị biến `counter` trong ví dụ 2
- ▶ Tình trạng tranh đua có thể dẫn đến tình trạng không nhất quán.
- ▶ Để ngăn chặn tình trạng tranh đua, các tiến trình cạnh tranh cần phải được đồng bộ hóa (synchronize).



# VẤN ĐỀ MIỀN TƯƠNG TRỰC (CSP)

- ▶ Xét 1 hệ thống có  $n$  tiến trình đang cạnh tranh  $\{P_0, P_1, \dots, P_{n-1}\}$
- ▶ **Miền tương trực (critical section):** là một đoạn mã lệnh của các tiến trình có chứa các hành động truy cập dữ liệu được chia sẻ như: thay đổi các biến dùng chung, cập nhật CSDL, ghi tập tin, ...  
  
⇒ Để tránh tình trạng tranh đua, các hệ thống phải đảm bảo khi một tiến trình đang trong miền tương trực, không có một tiến trình nào khác được phép chạy trong miền tương trực của nó.
- ▶ **Vấn đề miền tương trực (critical-section problem):** Thiết kế các giao thức để các tiến trình có thể sử dụng để hợp tác/cạnh tranh với nhau.

# VẤN ĐỀ MIỀN TƯƠNG TRỰC (CSP)

- ▶ Cần phải xác định được phần **entry section** và **exit section**.  
do {
- ▶ Mỗi tiến trình phải **xin phép** để được vào miền tương trực (đi qua vùng **entry section**), và sau đó thoát khỏi miền tương trực (đi qua vùng **exit section**) và thực hiện phần còn lại (remainder section).  

*entry section*

  
critical section  

*exit section*

  
remainder section
- ▶ **Giải pháp** cho vấn đề miền tương trực tương đối **phức tạp** với với các hệ thống định thời **trưng dụng**.  
} while (true);

## YÊU CẦU ĐỐI VỚI CÁC GIẢI PHÁP CHO CSP

- ▶ Một giải pháp cho vấn đề miền tương trực phải thỏa 3 yêu cầu:
  1. **Loại trừ lẫn nhau** (mutual exclusion): Nếu 1 t/trình đang thực thi trong miền tương trực, không một tiến trình nào khác được đi vào miền tương trực của chúng.
  2. **Tiến triển** (progress): Nếu không có tiến trình nào đang thực thi trong miền tương trực và tồn tại tiến trình đang chờ được thực thi trong miền tương trực của chúng, thì việc lựa chọn cho một tiến trình bước vào miền tương trực không thể bị trì hoãn vô hạn.
  3. **Chờ đợi hữu hạn** (bounded wait): Mỗi t/trình chỉ phải chờ để được vào miền tương trực trong một khoảng t/gian có hạn định (không xảy ra tình trạng “chết đói” – starvation).

# PHÂN LOẠI CÁC GIẢI PHÁP

- ▶ **Các giải pháp phần mềm:** dựa trên các **giải thuật phần mềm**, như:
  - ▶ Cho trường hợp chỉ có **2 tiến trình cạnh tranh**:
    - ▶ Giải thuật Peterson (Peterson's algorithm)
  - ▶ Cho trường hợp có  **$n \geq 2$  tiến trình cạnh tranh**:
    - ▶ Giải thuật Bakery
- ▶ **Các giải pháp phần cứng:**
  - ▶ Lệnh vô hiệu hóa ngắt (disable interrupt)
  - ▶ Lệnh máy đặc biệt: TestAndSet

# GIẢI THUẬT PETERSON

- ▶ Kết hợp cả biến khóa chia sẻ (GT1) và biến khóa riêng (GT2):
  - ▶ `int turn; //turn = i:  $P_i$  được phép vào miền tương trực.`
  - ▶ `boolean flag [2]; //flag[i] = true:  $P_i$  sẵn sàng vào miền tương trực.`
- ▶ Tổ chức đoạn mã của 1 tiến trình  $P_i$ :

```
do {  
    flag[i] := true  
    turn := j;  
    while (flag[j] && turn=j) ;  


critical section

  
    flag[i] = false;  


remainder section

  
} while (true);
```

- ▶ **Nhận xét:**

- ▶ Loại trừ lẫn nhau: ✓
- ▶ Tiến triển: ✓
- ▶ Chờ đợi hữu hạn: ✓

# GIẢI THUẬT BAKERY

- ▶ Miền tương trực cho  $n$  tiến trình:
  - ▶ Mỗi t/trình sẽ nhận được 1 **số** trước khi vào miền tương trực.
  - ▶ Tiến trình có số **nhỏ nhất** sẽ có quyền **ưu tiên cao nhất**.
  - ▶ Nếu hai tiến trình  $P_i$  và  $P_j$  nhận được cùng một số, nếu  $i < j$  thì  $P_i$  được phục vụ trước.
  - ▶ Bộ sinh số luôn sinh các số **theo thứ tự tăng**, ví dụ: 1, 2, 3, 3, 3, 4, ...
- ▶ Dữ liệu chia sẻ:  
    boolean choosing[n]  
    int number[n]
  - ▶ Khởi tạo: tất cả các phần tử của **choosing** = **false** và **number** = 0.

## GIẢI THUẬT BAKERY CHO TIỀN TRÌNH $P_i$

```
do {  
    choosing[i] = true;  
    number[i] = max(number[0], number[1], ..., number[n-1]);  
    choosing[i] = false;  
    for (j = 0; j < n; j++) {  
        while (choosing[j]) ;  
        while ((number[j] != 0) && ((number[j], j) < (number[i], i))) ;  
    } //for  
    $\framebox{critical section}$  
    number[i] = 0;  
    $\framebox{remainder section}$  
} while (true);
```

- ✱  $(\text{number } \#1, i) < (\text{number } \#2, j)$  nếu  $(\text{number } \#1 < \text{number } \#2)$   
hoặc  $(\text{number } \#1 = \text{number } \#2) \text{ AND } (i < j)$

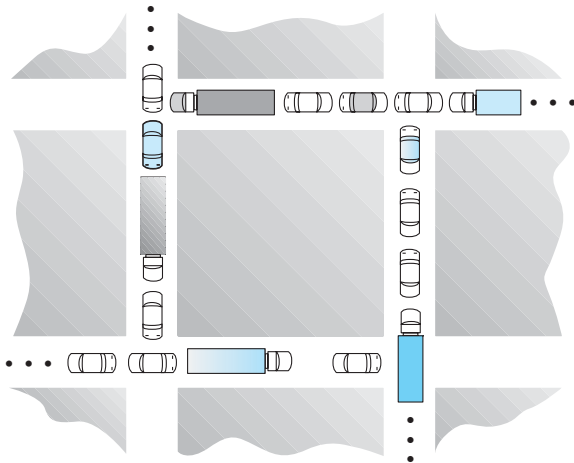
Tắc nghẽn (Deadlock)



# DEADLOCK LÀ GÌ?

- ▶ **Deadlock** là một trạng thái của hệ thống trong đó:
  - ▶ một tập hợp các tiến trình **đang bị nghẽn**
  - ▶ mỗi tiến trình đang **giữ một tài nguyên** và cũng đang **chờ một tài nguyên đang bị giữ** bởi một tiến trình khác trong tập các tiến trình đang bị nghẽn.
- ▶ Ví dụ 1:
  - ▶ Giả sử 1 hệ thống có 2 tiến trình P và Q và F1, F2 là 2 tập tin.
  - ▶ Tiến trình P đang giữ F1 và cần truy xuất thêm F2.
  - ▶ Tiến trình Q đang giữ F2 và cần truy xuất thêm F1.

## VÍ DỤ 2 – TRAFFIC DEADLOCK



## ĐIỀU KIỆN PHÁT SINH DEADLOCK

1. **Loại trừ hồ tương:** ít nhất một tài nguyên được giữ ở chế độ không chia sẻ (nonsharable mode).
2. **Giữ và chờ:** một tiến trình đang giữ ít nhất một tài nguyên và đợi thêm tài nguyên đang bị giữ bởi tiến trình khác.
3. **Không trưng dụng tài nguyên:** không trưng dụng tài nguyên cấp phát tiến trình, trừ khi tiến trình tự hoàn trả.
4. **Chờ đợi vòng tròn:** tồn tại một tập các tiến trình  $\{P_0, P_1, \dots, P_n\}$  đang chờ đợi như sau:  $P_0$  đợi một tài nguyên  $P_1$  đang giữ,  $P_1$  đợi một tài nguyên  $P_2$  đang giữ,  $\dots$ ,  $P_n$  đang đợi một tài nguyên  $P_0$  đang giữ.

# MÔ HÌNH HÓA HỆ THỐNG


- ▶ Hệ thống gồm một tập **các loại tài nguyên**, kí hiệu  $R_1, R_2, \dots, R_m$ 
  - ▶ Ví dụ: CPU cycles, memory space, I/O devices, ...
- ▶ Mỗi loại tài nguyên  $R_i$  có một **tập các thể hiện** (instances)  $W_i$
- ▶ Tiến trình **sử dụng tài nguyên** theo các bước:
  1. Yêu cầu (request) – phải chờ nếu không được đáp ứng ngay.
  2. Sử dụng (use)
  3. Giải phóng (release)
- ▶ Các tác vụ yêu cầu và hoàn trả được thực hiện bằng các lời gọi hệ thống.


# ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN – RAG


- ▶ Là đồ thị có hướng, với tập đỉnh  $V$  và tập cạnh  $E$
- ▶ Tập đỉnh  $V$  gồm 2 loại:
  - ▶ Tập  $P = \{P_1, P_2, \dots, P_n\}$ : tập các tiến trình trong hệ thống
  - ▶ Tập  $R = \{R_1, R_2, \dots, R_m\}$ : tập các tài nguyên của hệ thống
- ▶ Tập cạnh cũng gồm 2 loại:
  - ▶ Cạnh yêu cầu (request edge): có hướng từ  $P_i$  đến  $R_j$
  - ▶ Cạnh cấp phát (assignment edge): có hướng từ  $R_j$  đến  $P_i$

# KÝ HIỆU

► Tiến trình: 

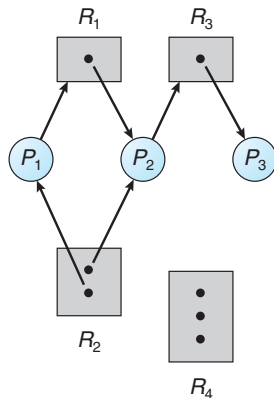
► Loại tài nguyên (với 4 thể hiện): 

►  $P_i$  yêu cầu 1 thể hiện của  $R_j$ : 

►  $P_i$  đang giữ 1 thể hiện của  $R_j$ : 

# ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN – VÍ DỤ 1

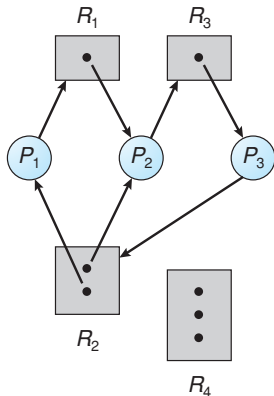
- ▶ Đồ thị cấp phát tài nguyên (không có chu trình và không deadlock):



- ▶  $P = \{P_1, P_2, P_3\}$
- ▶  $R = \{R_1, R_2, R_3, R_4\}$
- ▶  $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$
- ▶ **Thể hiện** của các loại tài nguyên:  
 $R_1 : 1; R_2 : 2; R_3 : 1; R_4 : 3.$
- ▶ **Trạng thái** của các tiến trình:
  - ▶  $P_1$ : giữ  $(R_2, 1)$ ; chờ  $(R_1, 1)$
  - ▶  $P_2$ : giữ  $(R_1, 1), (R_2, 1)$ ; chờ  $(R_3, 1)$
  - ▶  $P_3$ : giữ  $(R_3, 1)$

## ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN – VÍ DỤ 2

- ▶ Đồ thị cấp phát tài nguyên với chu trình và deadlock:

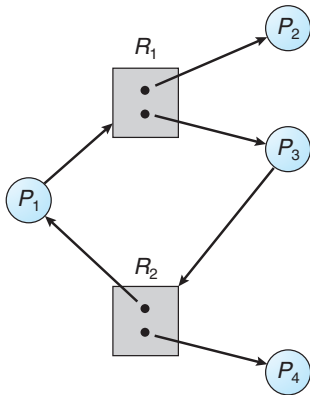


- ▶ **Trạng thái** của các tiến trình:
  - ▶  $P_1$ : giữ ( $R_2, 1$ ); chờ ( $R_1, 1$ )
  - ▶  $P_2$ : giữ ( $R_1, 1$ ), ( $R_2, 1$ ); chờ ( $R_3, 1$ )
  - ▶  $P_3$ : giữ ( $R_3, 1$ ); chờ ( $R_2, 1$ )
- ▶ 2 chu trình nhỏ nhất (minimal cycles):
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$
- ▶ **Deadlock**: cả 3 tiến trình  $P_1, P_2, P_3$



## ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN – VÍ DỤ 3

- ▶ Đồ thị cấp phát tài nguyên có chu trình nhưng không deadlock:




- ▶ Chu trình:  
 $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
- ▶ Tại sao không deadlock?

# ĐỒ THỊ CẤP PHÁT TÀI NGUYÊN & DEADLOCK

- ▶ Nếu đồ thị không có chu trình: **chắc chắn không có** deadlock
- ▶ Nếu đồ thị có chu trình:
  - ▶ Nếu mỗi loại tài nguyên chỉ có một thể hiện: **chắc chắn có** deadlock.
  - ▶ Nếu mỗi loại tài nguyên có nhiều thể hiện: **có thể có** deadlock.

# CÁC CÁCH TIẾP CẬN ĐỐI VỚI VẤN ĐỀ DEADLOCK

1. Đề ra các giao thức để đảm bảo cho hệ thống không bao giờ rơi vào trạng thái deadlock.
2. Cho phép hệ thống rơi vào trạng thái deadlock, sau đó sử dụng các giải thuật để phát hiện deadlock và phục hồi.
3. Không quan tâm và không xử lý vấn đề deadlock trong hệ thống 
  - ▶ Khá nhiều hệ điều hành sử dụng phương pháp này.
  - ▶ Tuy nhiên, nếu deadlock không được phát hiện và xử lý sẽ dẫn đến việc giảm hiệu suất của hệ thống. Cuối cùng, hệ thống có thể ngưng hoạt động và phải khởi động lại.

# NGĂN CHẶN VÀ TRÁNH DEADLOCK

- ▶ Có **hai biện pháp** để đảm bảo hệ thống không rơi vào trạng thái deadlock:
  1. **Ngăn chặn deadlock** (deadlock prevention): không cho phép (ít nhất) một trong bốn **điều kiện cần cho deadlock** xảy ra.
  2. **Tránh deadlock** (deadlock avoidance): các tiến trình cần cung cấp **thông tin về tài nguyên** nó cần để hệ thống cấp phát tài nguyên một cách thích hợp.

# PHÁT HIỆN DEADLOCK

- ▶ Trong cách tiếp cận “phát hiện và phục hồi” đối với vấn đề deadlock:
  - ▶ Chấp nhận cho deadlock xảy ra trong hệ thống.
  - ▶ Sử dụng các giải thuật để phát hiện deadlock.
  - ▶ Nếu có deadlock thì sẽ tiến hành phục hồi hệ thống, dùng 1 **sơ đồ phục hồi** thích hợp.
- ▶ Các giải thuật phát hiện deadlock thường sử dụng **RAG**.
- ▶ Có 2 loại giải thuật:
  - ▶ Cho trường hợp mỗi loại tài nguyên chỉ có 1 thể hiện.
  - ▶ Cho trường hợp mỗi loại tài nguyên có nhiều thể hiện.

# TỔNG KẾT

TỔNG QUAN VỀ TIẾN TRÌNH

GIAO TIẾP LIÊN TIẾN TRÌNH

ĐỊNH THỜI TIẾN TRÌNH

CÁC GIẢI THUẬT ĐỊNH THỜI

ĐỒNG BỘ HÓA TIẾN TRÌNH

TẮC NGHẼN (DEADLOCK)