Điều phối CPU

TH 106: Hệ điều hành

Khoa CNTT
ĐH KHTN

Là gì, tại sao?

Điều phối CPU là gì?

Tại sao?

Đầu tiên là để chia sẻ tài nguyên tốn kém – đa chương

Ngày nay có thể thực thi nhiều tác vụ cùng lúc vì processor rất mạnh

Giả thiết

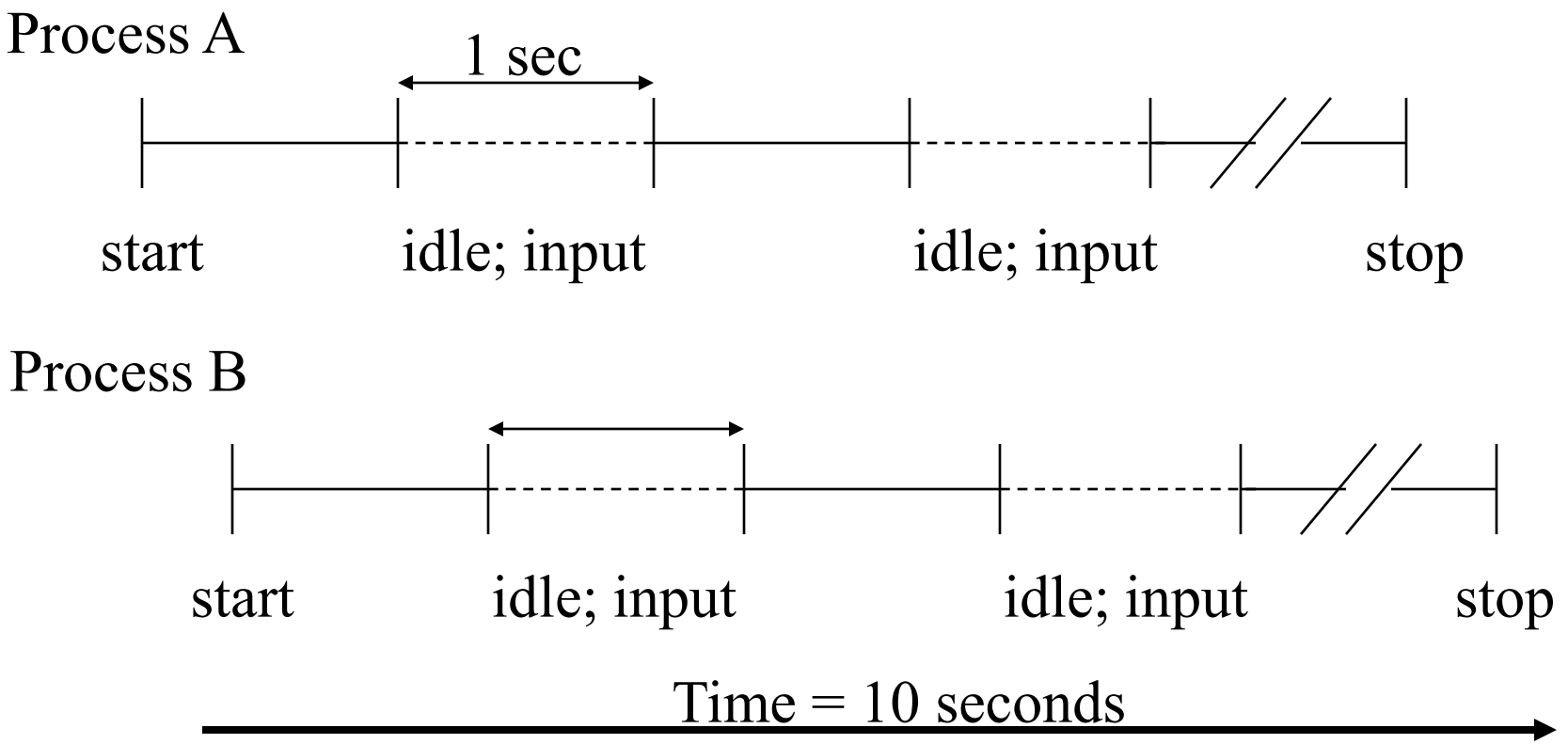
Nhiều công việc tranh giành CPU

CPU là tài nguyên khan hiếm

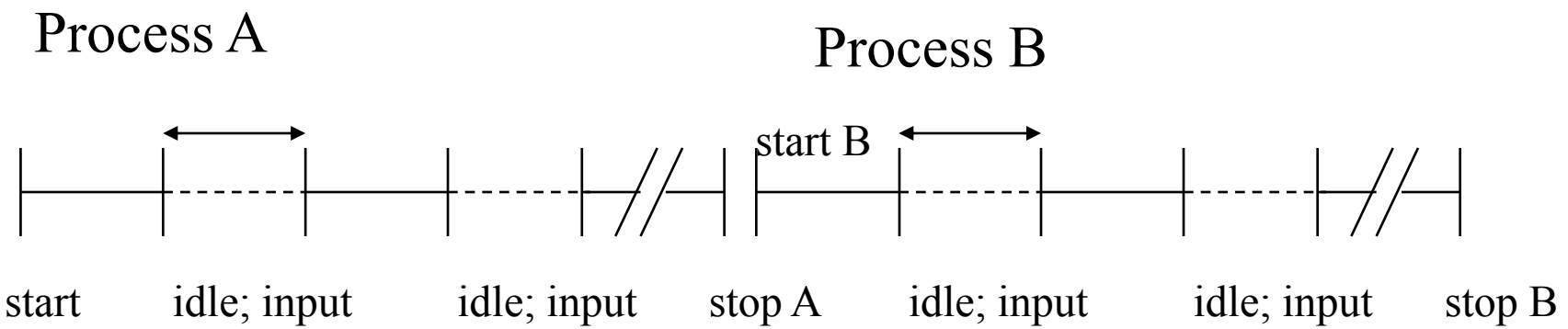
Các công việc là độc lập và tranh giành tài nguyên lẫn nhau (giả thiết này không thật sự đúng trong tất cả hệ thống/ngữ cảnh)

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)

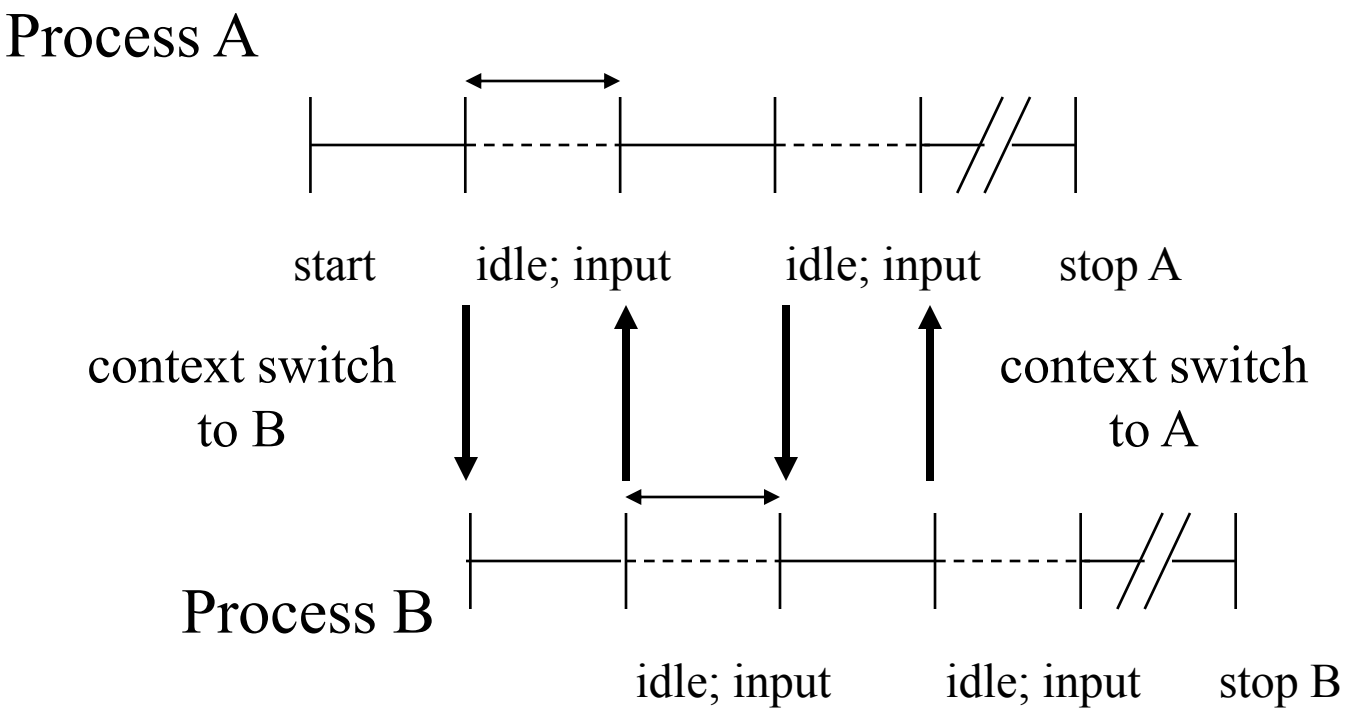


Tổng thời gian = 20 giây

Hiệu suất = 2 cv trong 20 giây = 0.1 cv/giây

Tg chờ trung bình = (0+10)/2 = 5 giây

Ví dụ đa chương trình (tt)

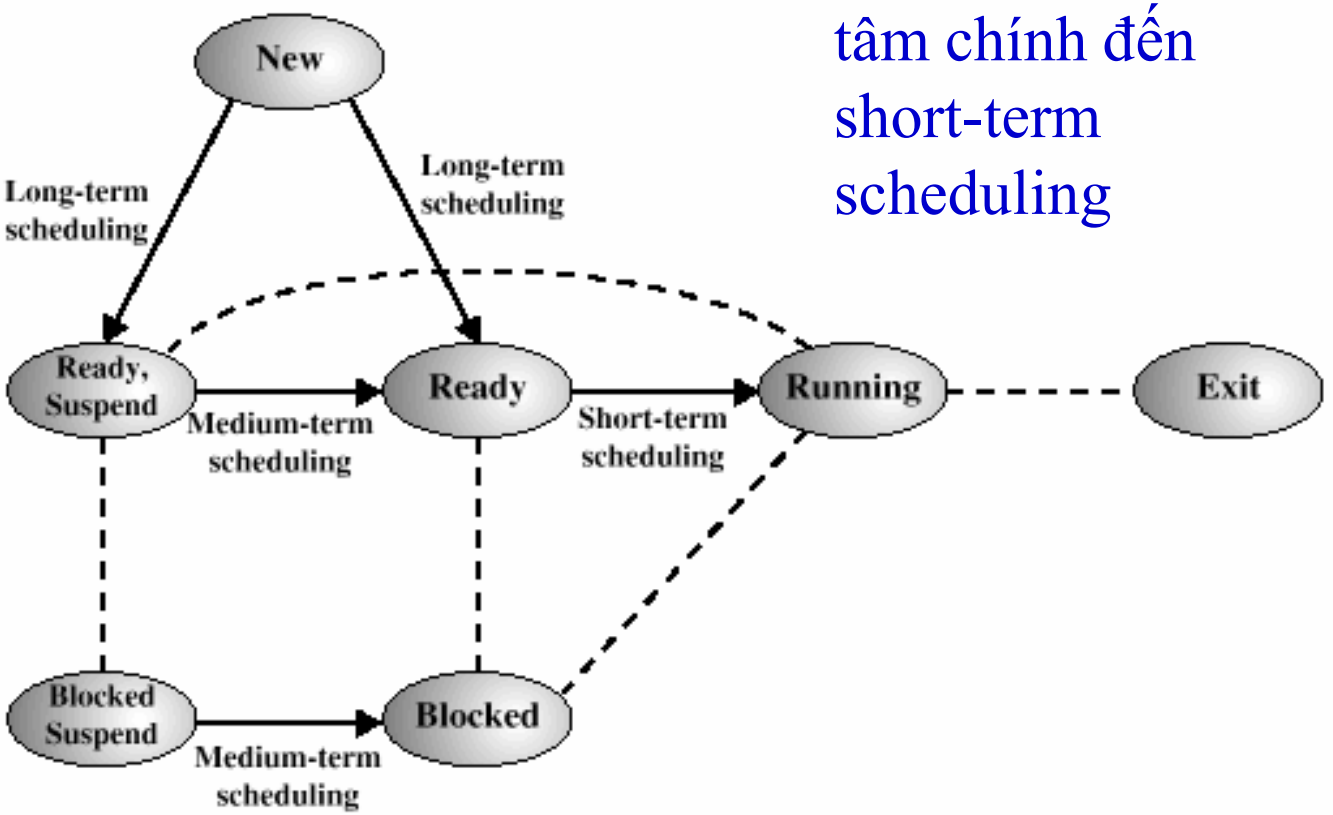


Hiệu suất = 2 cv 11 giây = 0.18 cv/giây

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

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 cv và kết thúc cv (gồm thời gian chờ). Cho cv theo lô, tuần tự

Response time: cho những cv 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

Thiết kế

Hai chiều

Chọn lựa

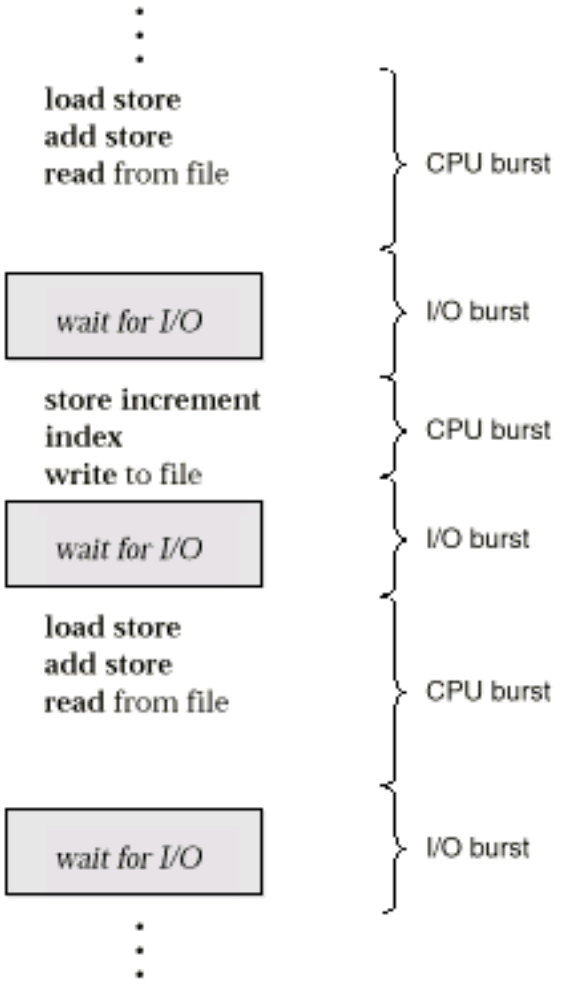
Ready job nào sẽ được thực thi kế tiếp?

Preemption

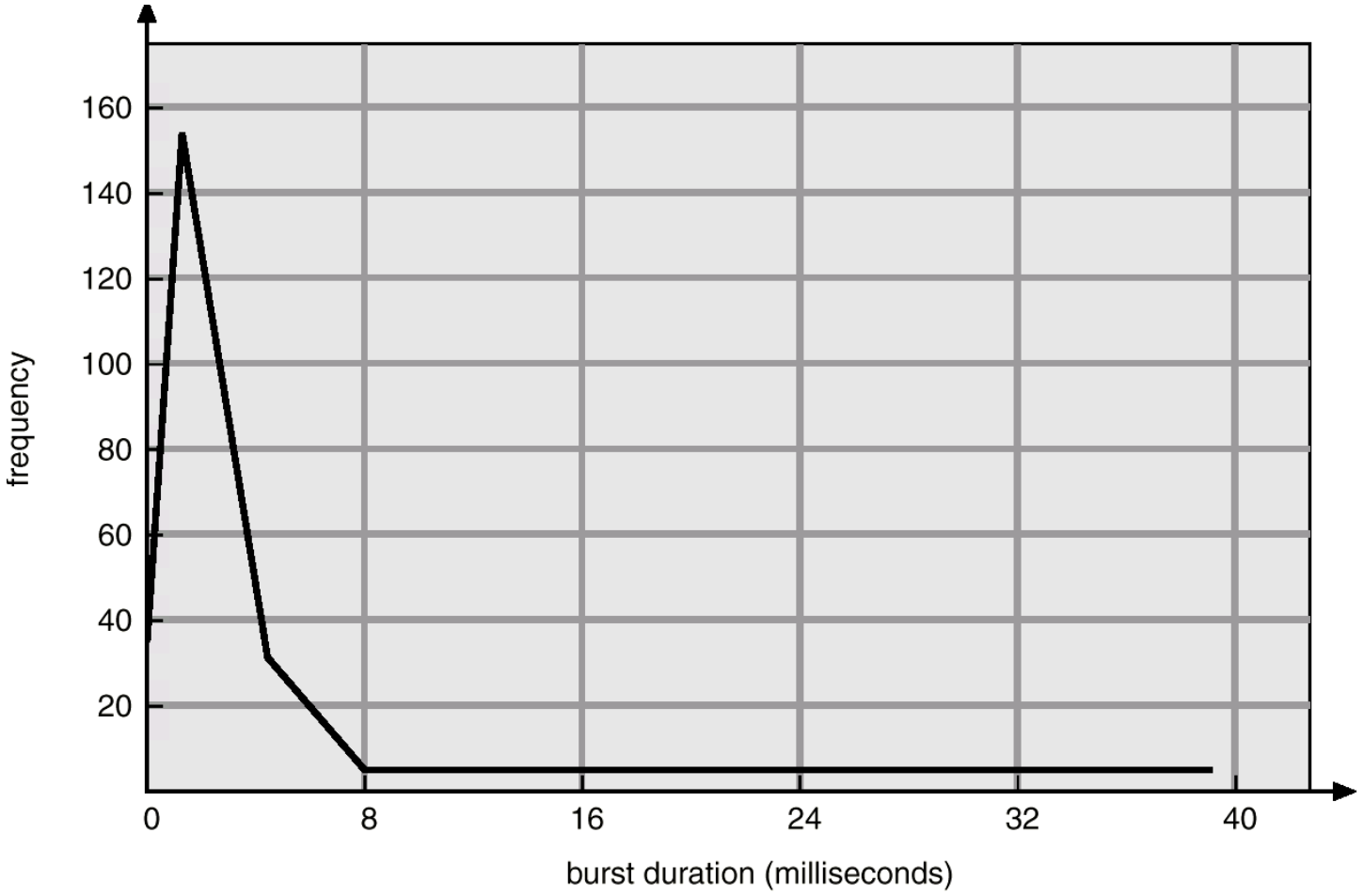
preemptive: cv đang thực thi có thể bị ngắt và chuyển vào trạng thái Ready

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

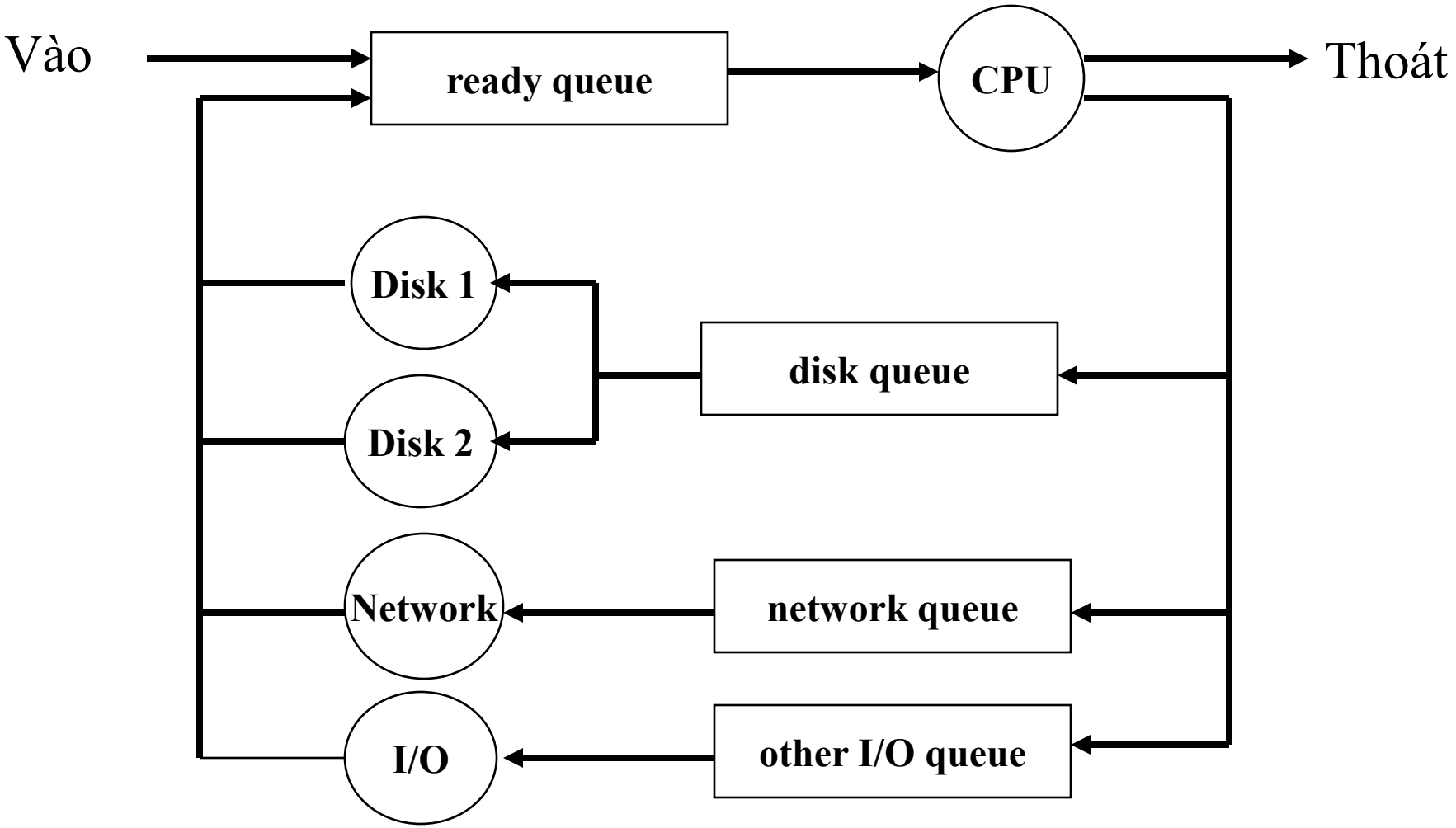
Đặc tính của công việc



Histogram của CPU-burst Times



Sơ đồ hàng đợi



Mô hình hàng đợi

Vòng tròn biểu diễn servers

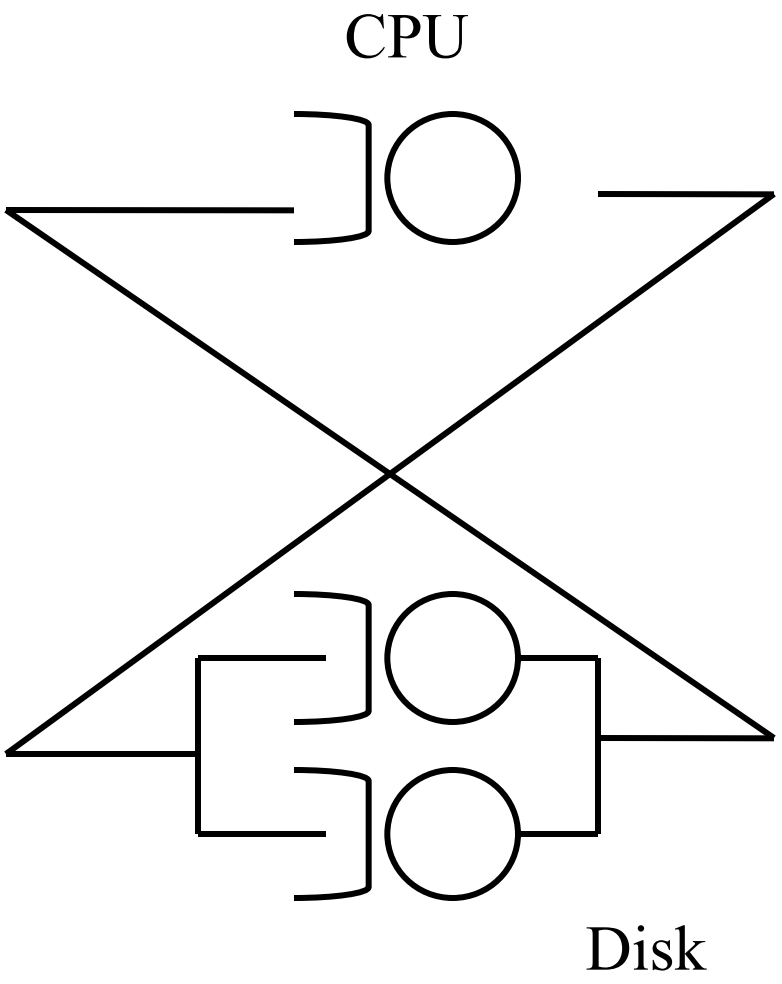
Hình chữ nhật biểu diễn hàng đợi (queues)

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 (Short-Term)

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.

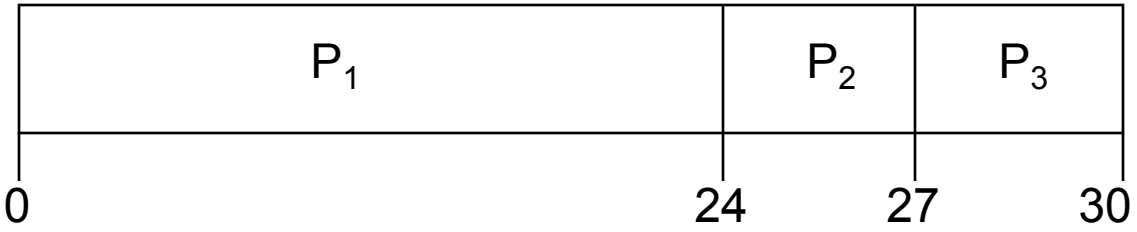
Lập lịch First-Come, First-Served (FCFS)

Ví dụ:

Tiến trình	Thời gian dùng CPU
P_1	24
P_2	3
P_3	3

Giả sử các tiến trình đến theo thứ tự: P_1 , P_2 , P_3

Gantt Chart của lập lịch như sau:



Thời gian chờ $P_1 = 0$; $P_2 = 24$; $P_3 = 27$

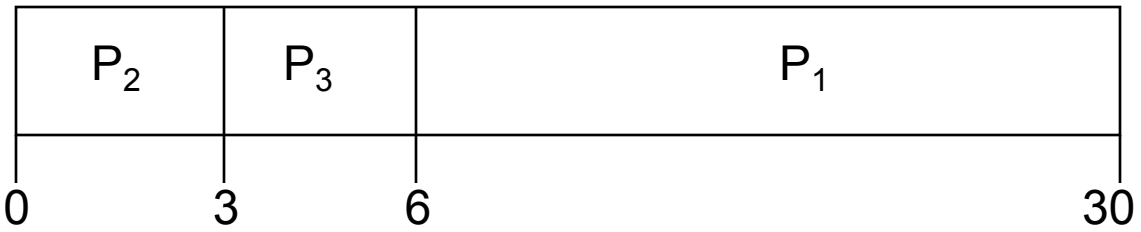
Trung bình tgian chờ: $(0 + 24 + 27)/3 = 17$

Lập lịch FCFS (tt.)

Giả sử tiến trình đến theo thứ tự

$$P_2 , P_3 , P_1 .$$

Gantt chart của lập lịch như sau:



Thời gian chờ $P_1 = 6; P_2 = 0, P_3 = 3$

Trung bình thời gian chờ: $(6 + 0 + 3)/3 = 3$

Tốt hơn nhiều so với trường hợp trước.

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

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.

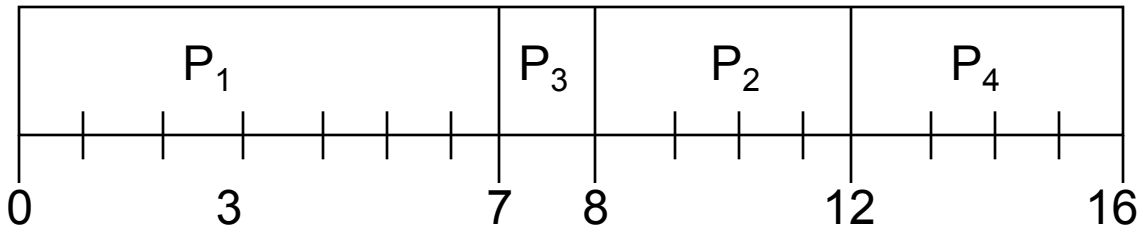
Preemptive – nếu một tiến trình mới vào mà thời gian cần dùng CPU ít hơn thời gian còn lại cần dùng CPU của tiến trình hiện hành. Lược đồ này gọi là Shortest-Remaining-Time-First (SRTF).

SJF là tối ưu – cho kết quả tốt nhất về trung bình thời gian chờ của một tập các tiến trình.

Ví dụ Non-Preemptive SJF

Tiến trình	Thời điểm đến	Thời gian dùng CPU
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (non-preemptive)

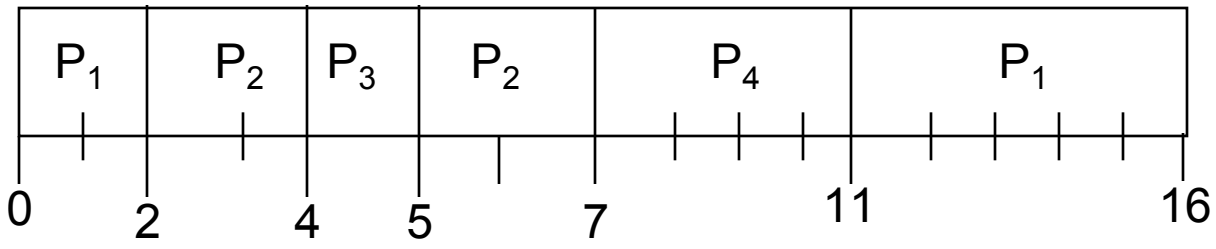


Trung bình thời gian chờ = (0 + 6 + 3 + 7)/4 = 4

Ví dụ Preemptive SJF

Tiến trình	Thời gian đến	Tgian dùng CPU
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

SJF (preemptive)



Trung bình thời gian chờ = (9 + 1 + 0 +2)/4 = 3

Xác định thời gian sử dụng CPU kế tiếp

Chỉ có thể ước lượng.

Sử dụng thời gian sử dụng CPU ngay trước, dùng qui luật trung bình giảm theo hàm mũ.

1. $t_n = \text{actual lenght of } n^{th} \text{ CPU burst}$
2. $\tau_{n+1} = \text{predicted value for the next CPU burst}$
3. $\alpha, 0 \le \alpha \le 1$
4. Define :

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\tau_n.$$

Ví dụ Exponential Averaging

$\alpha = 0$

$\tau_{n+1} = \tau_n$

Không xét các giá trị thực đã dùng CPU.

$\alpha = 1$

$\tau_{n+1} = t_n$

Chỉ dùng giá trị thực mới dùng CPU.

Nếu mở rộng công thức, ta được:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \ldots$$
$$+ (1 - \alpha)^j \alpha t_{n-1} + \ldots$$
$$+ (1 - \alpha)^{n+1} t_n \tau_0$$

Vì α và $(1 - \alpha)$ nhỏ hơn hay bằng 1,mỗi giá trị kế tiếp sẽ nhỏ dần.

Điều phối theo 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

nonpreemptive

SJF là một dạng điều phối theo độ ưu tiên (độ ưu tiên: dự đoán thời gian CPU burst kế tiếp).

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..)

Round Robin (RR)

Mỗi tiểu trình sẽ chiếm CPU trong một đơn vị thời gian (*time quantum*), thường là 10-100 milliseconds. Sau khi hết tgian này, tiến trình phải dừng và chuyển về cuối hàng đợi ready.

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

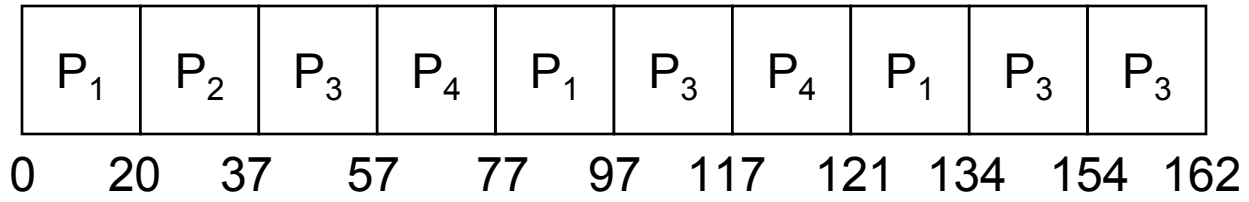
q lớn $\Rightarrow$ FIFO

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

Ví dụ: RR với Time Quantum = 20

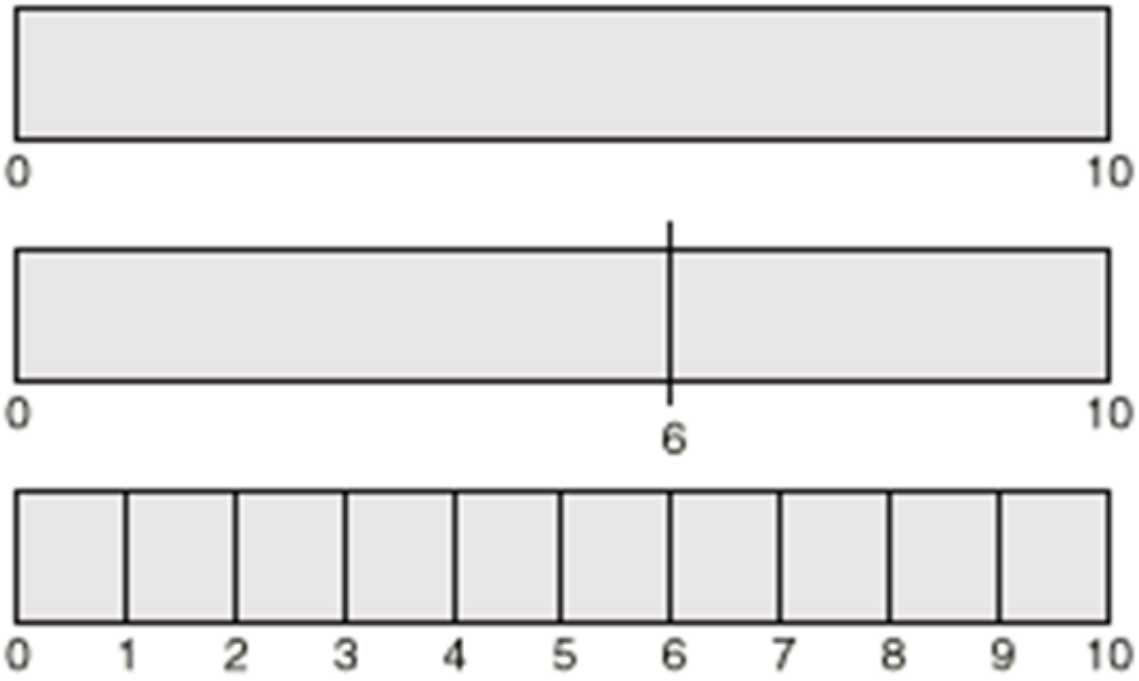
Tiến trình	Thời gian dùng CPU
P_1	53
P_2	17
P_3	68
P_4	24

Biểu đồ Gantt:



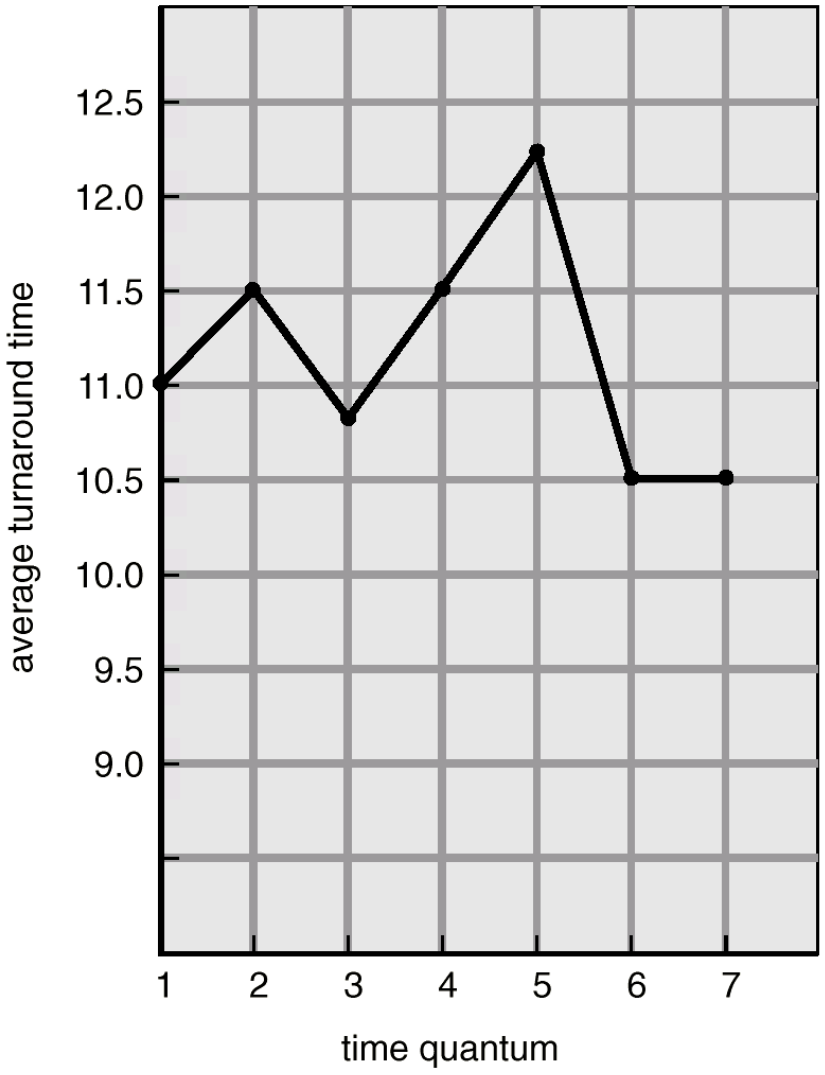
Thường là, trung bình turnaround time cao hơn SJF, nhưng sự phản hồi tốt hơn.

Time Quantum nhỏ làm tăng Context Switches ntn?



quantum	context switches
12	0
6	1
1	9

Turnaround Time thay đổi tùy theo Time Quantum



process	time
P_1	6
P_2	3
P_3	1
P_4	7

Hàng đợi đa cấp

Hàng đợi Ready được chia thành 2 hàng đợi phân biệt:
foreground (interactive)
background (batch)

Mỗi hàng đợi sử dụng thuật toán điều phối riêng,
foreground – RR
background – FCFS

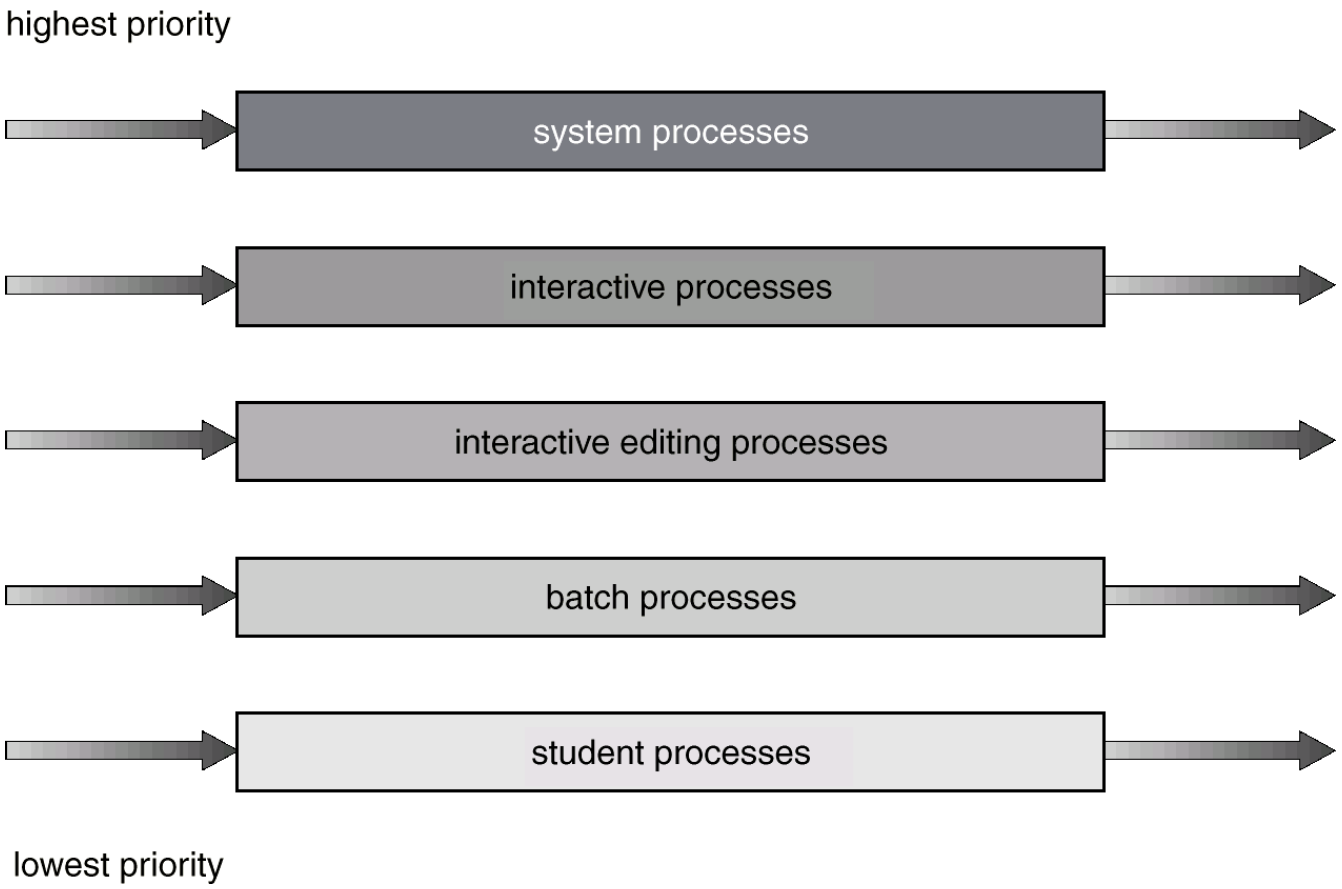
Điều phối giữa các hàng đợi.

Điều phối theo độ ưu tiên cố định; nghĩa là phục vụ tất cả tiến trình foreground rồi tới background. Có thể xảy ra starvation.

Time slice – mỗi hàng đợi nhận chiếm CPU một khoảng thời gian, và tự điều phối các tự điều phối các tiến trình của nó trong thời gian đó; nghĩa là,
80% cho foreground theo RR

20% cho background theo FCFS

Điều phối hàng đợi đa cấp



Multilevel Feedback Queue

Tiến trình có thể di chuyển giữa các hàng đợi;

Điều phối Multilevel-feedback-queue được xác định bởi các thông số sau:

Số hàng đợi

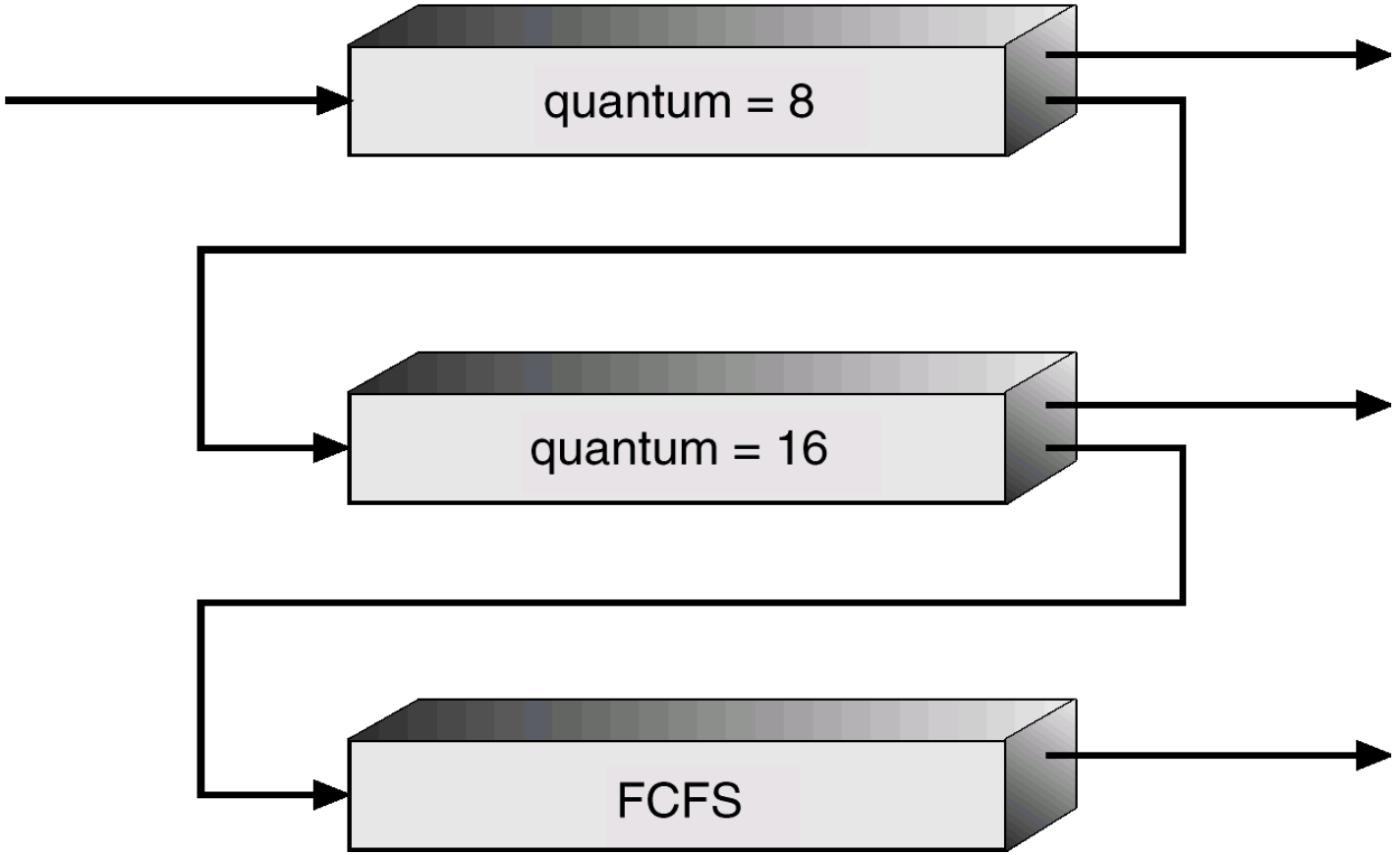
Thuật toán điều phối trong mỗi hàng đợi

Phương thức dùng để quyết định khi nào tiến trình sẽ được nâng cấp

Phương thức dùng để quyết định khi nào tiến trình sẽ bị giáng cấp

Phương thức quyết định tiến trình mới sẽ vào hàng đợi nào

Multilevel Feedback Queues



Ví dụ Multilevel Feedback Queue

Ba hàng đợi:

Q_0 – time quantum 8 mili giây

Q_1 – time quantum 16 mili giây

Q_2 – FCFS

Lập lịch

Một việc mới vào queue Q_0 nó được điều phối theo FCFS. Khi nó nhận CPU, chỉ dùng trong 8 mili giây. Nếu chưa hoàn tất trong 8 mili giây, công việc chuyển sang queue Q_1.

Tại Q_1 cv được điều phối theo FCFS và nhận CPU thêm 16 mili giây. Nếu vẫn chưa hoàn tất, nó sẽ bị đẩy qua queue Q_2.

Điều phối trong UNIX truyền thống

Multilevel feedback queues

128 Độ ưu tiên khác nhau (0-127)

1 Round Robin queue cho mỗi độ ưu tiên

Cứ mỗi sự kiện điều phối, bộ điều phối chọn hàng đợi không rỗng có độ ưu tiên thấp nhất và thực thi công việc theo round-robin

Sự kiện điều phối :

- Clock interrupt (ngắt đồng hồ)
- Tiến trình gọi một system call
- Tiến trình ngừng sử dụng CPU,v.d. thực hiện tác vụ I/O

Điều phối trong UNIX truyền thống

Tất cả các tiến trình được gán một độ ưu tiên cơ bản(baseline) dựa trên loại và trạng thái hiện tại:

swapper	0
waiting for disk	20
waiting for lock	35
user-mode execution	50

Khi có sự kiện điều phối, các độ ưu tiên được hiệu chỉnh lại dựa trên thời lượng cần dùng CPU, độ lớn của nó, và thời gian đã chờ đợi.

Đa số tiến trình đang dừng, nên rất nhiều "khoảng trống" được sử dụng để tính lại độ ưu tiên.

Tính độ ưu tiên trong UNIX

Cứ mỗi 4 ngắt đồng hồ độ ưu tiên được cập nhật:

$$P = BASELINE + \left[\frac{utilizatio\,n}{4} \right] + 2NiceFactor$$

Giá trị utilization tăng lên 1 sau mỗi ngắt đồng hồ.

niceFactor giúp điều khiển độ ưu tiên. Giá trị có thể gán từ –20 đến +20.

Các công việc sử dụng nhiều CPU thì độ ưu tiên sẽ tăng. Các công việc sử dụng ít CPU thì độ ưu tiên sẽ trở về baseline.

Tính độ ưu tiên trong UNIX

Công việc sử dụng CPU quá lâu sẽ bị "kẹt" tại độ ưu tiên cao nhất.

Hàm hủy sẽ điều chỉnh lại giá trị utilization của công việc hiện thời.

Utilization của tiến trình tại thời điểm ***t*** bị giảm mỗi giây:

$$u_t = \left[\frac{2load}{(2load + 1)} \right] + u_{(t-1)} + niceFactor$$

Load là trung bình số công việc có thể cấp CPU để thực thi trong giây vừa rồi

Giảm độ ưu tiên trong UNIX

1 cv trong CPU → Load vì vậy = 1. Giả sử niceFactor = 0.

Tính utilization tại thời điểm N:

+1 second: $$U_1 = \frac{2}{3}U_0$$

+2 seconds $$U_2 = \frac{2}{3}\left[U_1 + \frac{2}{3}U_0\right] = \frac{2}{3}U_1 + \left(\frac{2}{3}\right)^2 U_0$$

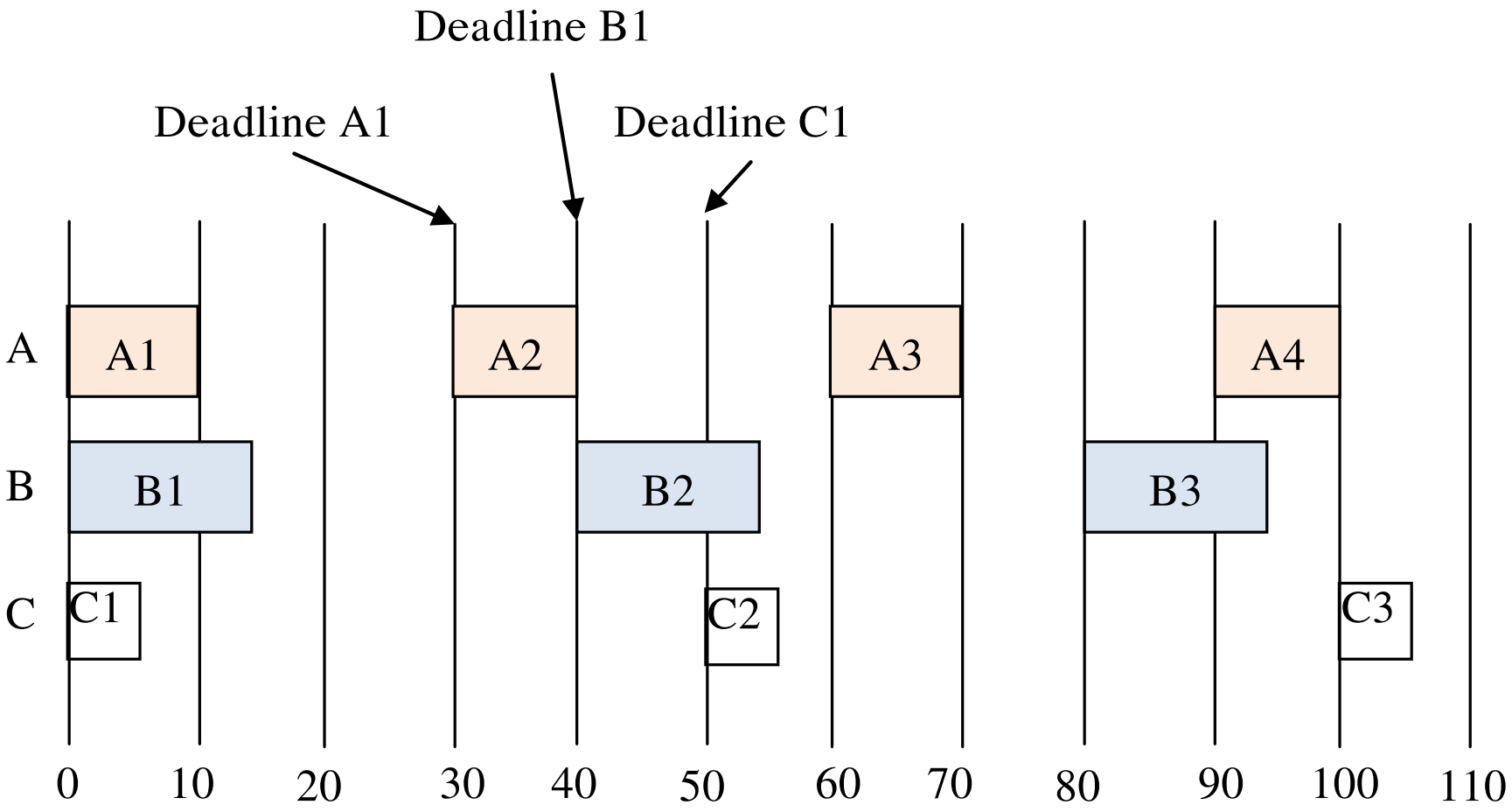
+N seconds $$U_n = \frac{2}{3}U_{n-1} + \left(\frac{2}{3}\right)^2 U_{n-2}\cdots$$

Điều phối trong hệ thống thời gian thực

- Giả sử hệ thống có m tiến trình.
- Mỗi tiến trình thực thi công việc trong mỗi định kỳ **Pi**,
- Và cần thời gian **Ci** để hoàn thành công việc.
- Điều kiện cần để tất cả các tiến trình hoàn thành đúng hạn

$$\sum_{i=0}^{m} \frac{Ci}{Pi} \leq 1$$

Ví dụ tiến trình định thời

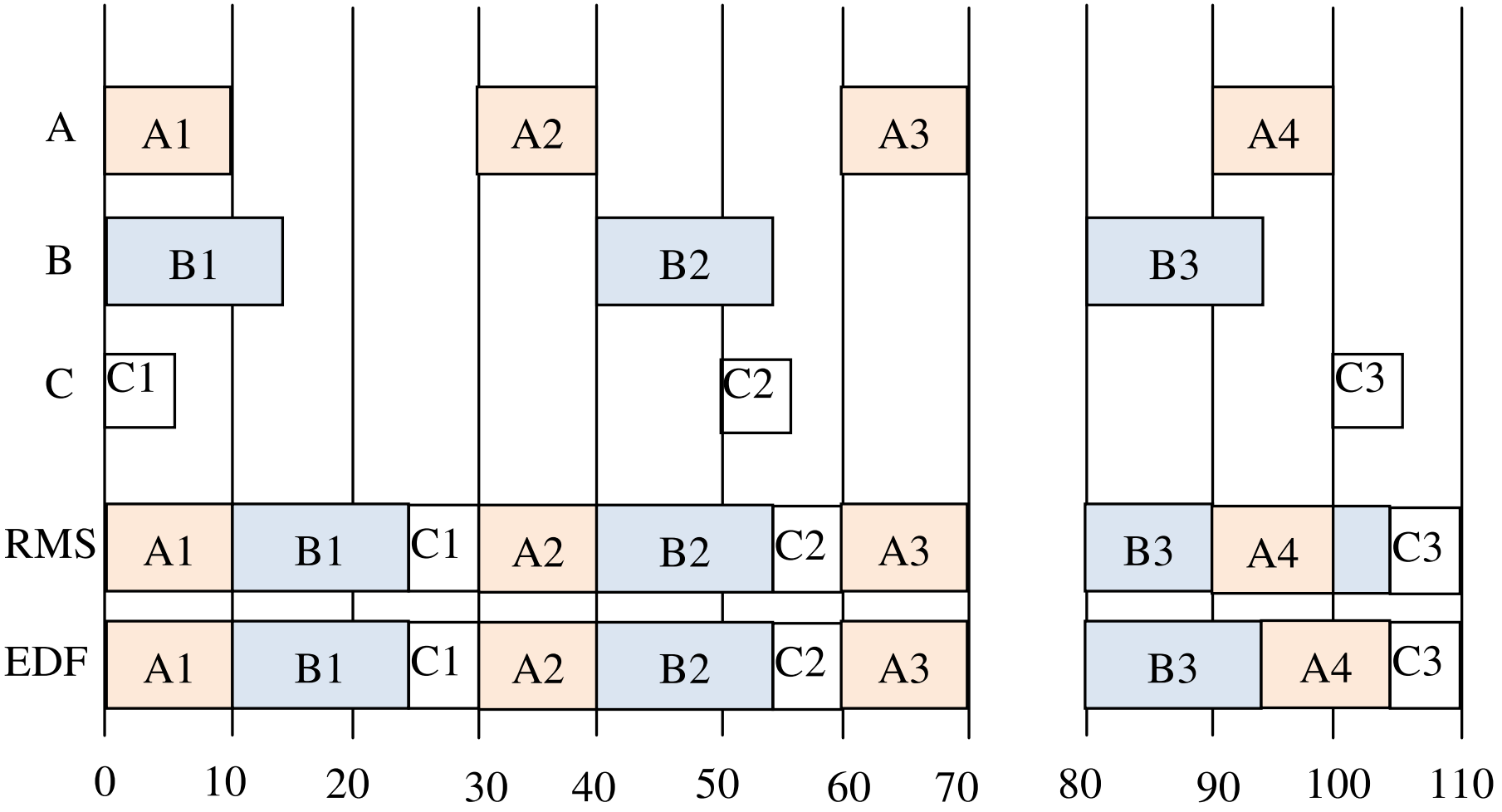


Điều phối dựa vào tần suất (RMS rate monotonic scheduling)

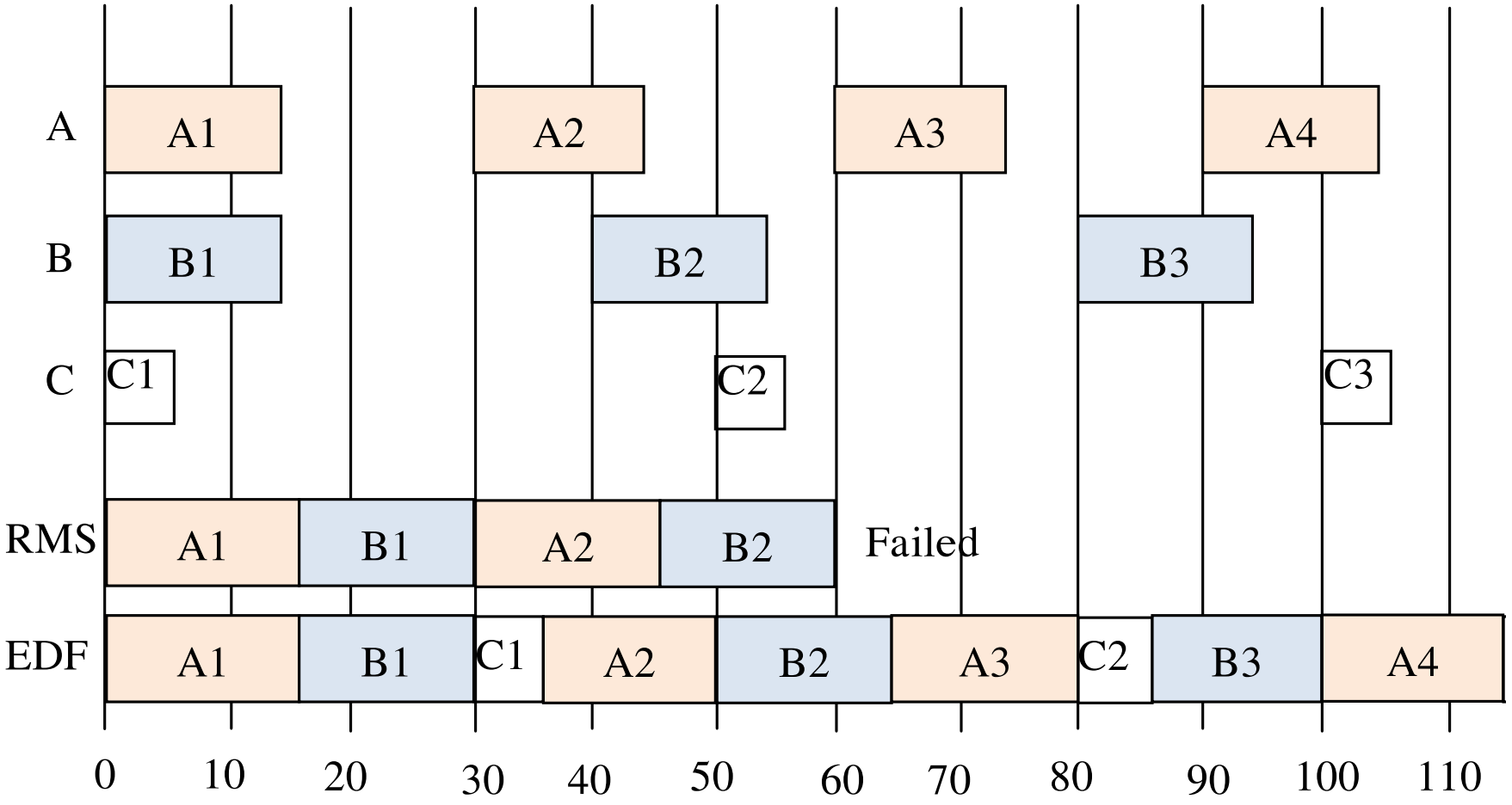
- RMS (rate monotonic scheduling) do Liu và Layland đề xuất 1973.
- Điều kiện bài toán:

1. **Mỗi tiến trình định kỳ phải hoàn thành trong khoảng hạn định của nó**
2. **Các tiến trình không phụ thuộc lẫn nhau**
3. **Mỗi tiến trình sử dụng CPU một khoảng thời gian bằng nhau khi nó được cấp CPU**
4. **Các tiến trình không theo định kỳ thì không có deadline.**
5. **Thời gian thay đổi tiến trình chiếm giữ CPU là không đáng kể**

RMS (rate monotonic scheduling)



Điều phối theo deadline gần nhất (EDF Earliest Deadline First)



- Liu và Layland chứng minh rằng, trong các hệ thống có m tiến trình định kỳ, nếu

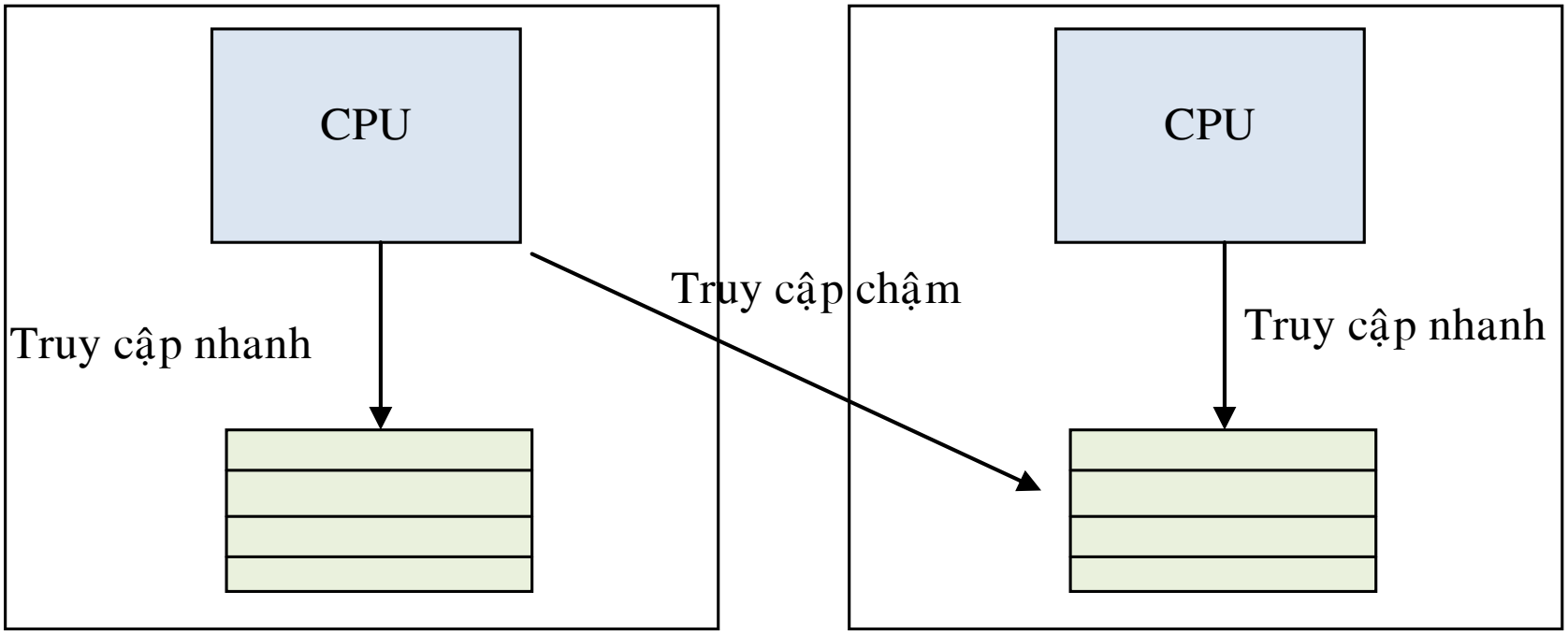
$$\sum_{i=0}^{m} \frac{Ci}{Pi} \leq m(2^{\frac{1}{m}} - 1)$$

thì RMS sẽ điều phối thành công.

ĐIỀU PHỐI TRONG HỆ THỐNG NHIỀU BỘ XỬ LÝ

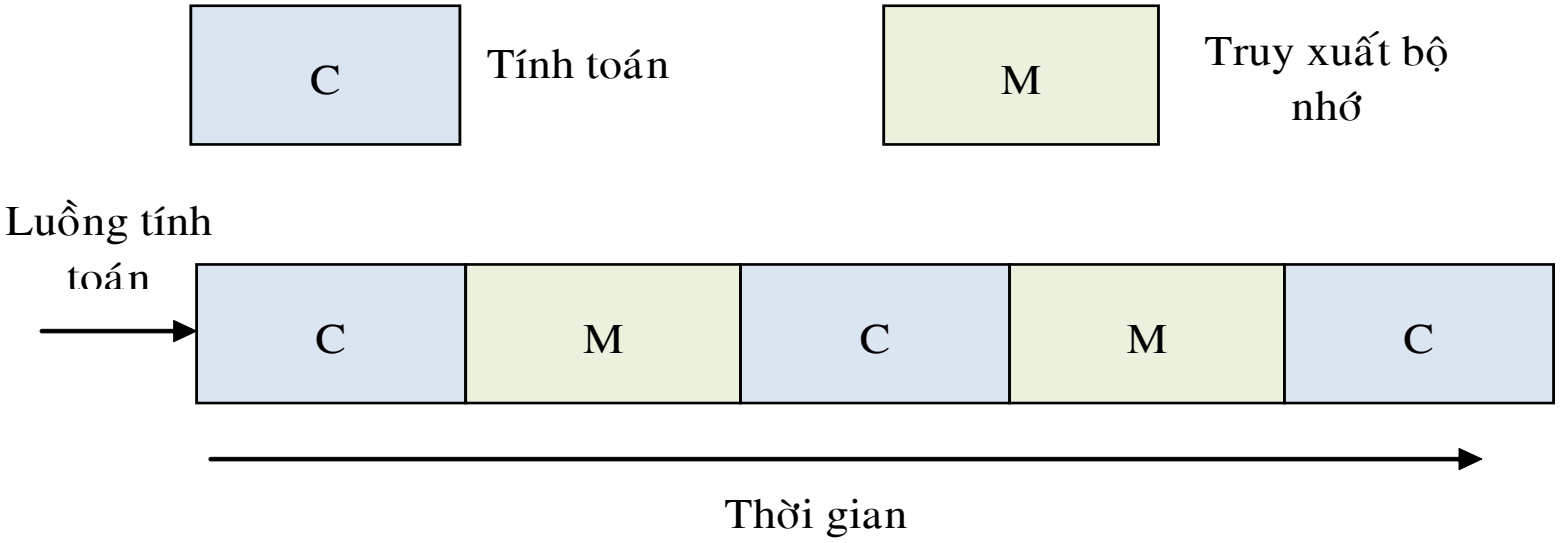
- Điều phối **đa xử lý bất cân xứng**
  - **Dùng 1 CPU làm Master server**
  - **Mọi điều phối do CPU này quyết định**
- Điều phối **đa bộ xử lý cân xứng (symmetric multiprocessing - SMP)**
  - **Các CPU ngang hàng nhau. Mỗi CPU 1 hàng đợi**
  - **Cân bằng tải: làm sao để các hàng đợi ngang bằng nhau**
- Gần như tất cả các hệ điều hành hỗ trợ SMP, gồm có Windows XP, 2000, Solaris, Linux, Mac OS X.

Ưu ái bộ xử lý

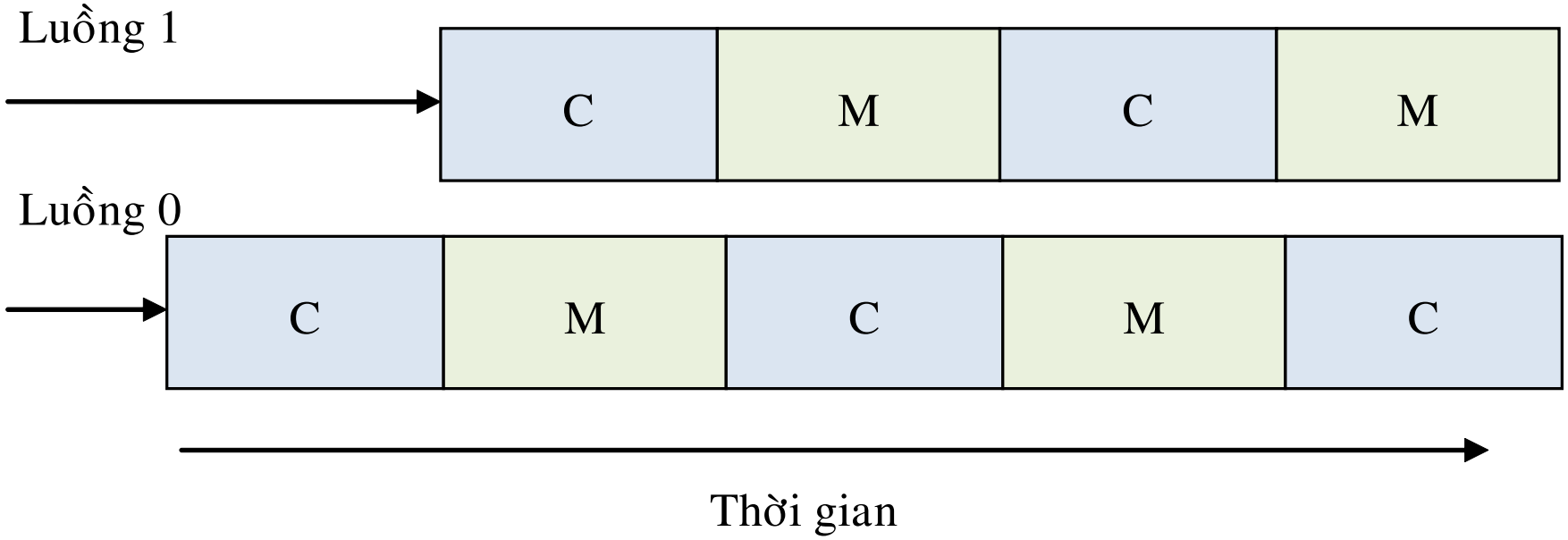


NUMA – nonuniform memory access

- Người ta phát hiện ra rằng, khi bộ xử lý truy cập bộ nhớ, nó cần một khoảng thời gian lớn để dữ liệu trên bộ nhớ chính sẵn sàng.
- Điều này, gọi là **trì trệ truy xuất bộ nhớ (memory stall)**, lý do: dữ liệu không còn trên vùng nhớ cache.



Multithreaded processor



- Hôm sau: đồng bộ, đọc phần Synchronization.