ĐỊNH THỜI CPU

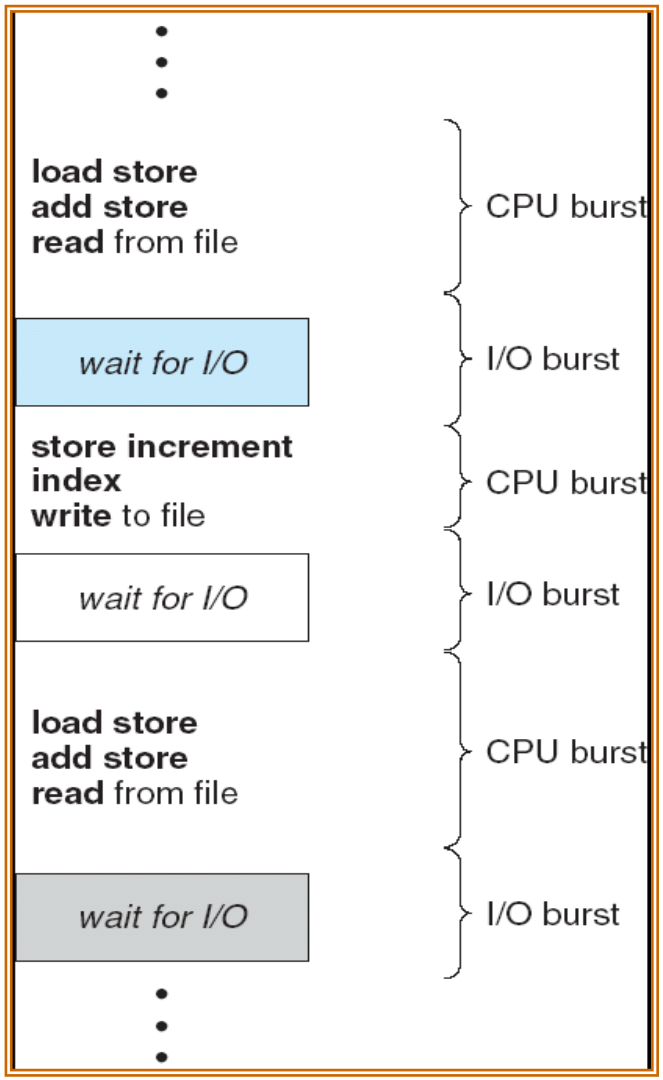
*Mục tiêu**

- Hiểu được
  - Tại sao cần phải định thời
  - Các tiêu chí định thời
  - Một số giải thuật định thời

(Ghi chú: những slide có dấu * ở tiêu đề là những slide quan trọng, những slide khác dùng để diễn giải thêm)

*Một số khái niệm cơ bản**

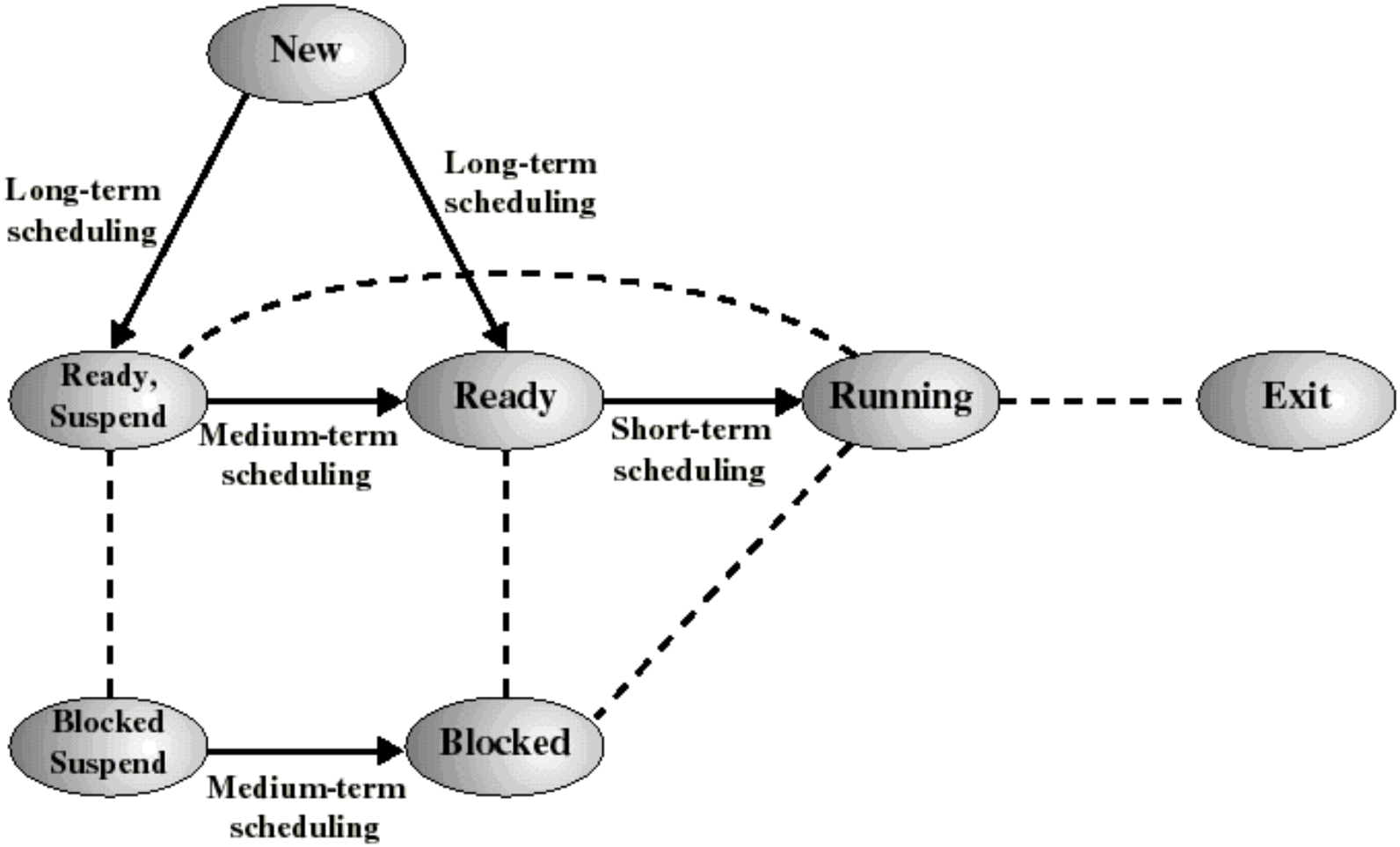
- Chu kỳ CPU-I/O
- *CPU-bound* process có thời gian sử dụng CPU nhiều hơn thời gian sử dụng I/O
- Phần lớn thời gian của *I/O-bound* process dùng để đợi I/O



*Một số khái niệm cơ bản**

- Trong các hệ thống multitasking
  - Tại một thời điểm trong bộ nhớ có nhiều process
  - Tại mỗi thời điểm chỉ có một process được thực thi
  - Do đó, cần phải giải quyết vấn đề phân chia, lựa chọn process thực thi sao cho được hiệu quả nhất. Cần có chiến lược định thời CPU

Phân loại các hoạt động định thời



Phân loại các hoạt động định thời

- *Định thời dài hạn* (long-term scheduling): process nào được chấp nhận vào hệ thống
- *Định thời trung hạn* (medium-term sched.): process nào được đưa vào (swap in), đưa ra khỏi (swap out) bộ nhớ chính
- *Định thời ngắn hạn* (short-term sched.): process nào được thực thi tiếp theo

Định thời dài hạn

- Xác định chương trình nào sẽ được đưa vào hệ thống để thực thi
- Quyết định *độ-đa-lập-trình* (degree of multiprogramming)
- Nếu càng nhiều process được đưa vào hệ thống
  - Khả năng các process bị block có xu hướng giảm
  - Sử dụng CPU hiệu quả hơn
  - Mỗi process được phân chia khoảng thời gian sử dụng CPU thấp hơn
- Thường có xu hướng đưa vào một tập lẫn lộn các CPU-bound process và I/O-bound process

Định thời trung hạn

- Quyết định về việc đưa process vào bộ nhớ chính, hay ra khỏi bộ nhớ chính phụ thuộc vào yêu cầu quản lý việc đa-lập-trình (multiprogramming)
  - Cho phép bộ định thời dài hạn chấp nhận nhiều process hơn số lượng process mà có tổng kích thước được chứa vừa trong bộ nhớ chính
  - Nhưng nếu có quá nhiều process thì sẽ làm tăng việc truy xuất đĩa, do đó cần phải lựa chọn độ-đa-lập-trình cho phù hợp
- Được thực hiện bởi phần mềm quản lý bộ nhớ

*Định thời ngắn hạn**

- Xác định process nào được thực thi tiếp theo, còn gọi là định thời CPU
- Được kích hoạt khi có một sự kiện có thể dẫn đến khả năng chọn một process để thực thi
  - Ngắt thời gian (clock interrupt)
  - Ngắt ngoại vi (I/O interrupt)
  - Lời gọi hệ thống (operating system call)
  - Signal

…chương này sẽ tập trung vào định thời ngắn hạn…

*Nội dung cần quan tâm**

- Định thời trên hệ thống có một processor (uniprocessor scheduling): quyết định việc sử dụng (một) CPU cho một tập các process trong hệ thống
- Tiêu chí nào?

*Tiêu chí định thời**

- *Độ lợi CPU* (CPU utilization)
  - Khoảng thời gian CPU bận, từ 0% đến 100%
  - Cần giữ cho CPU càng bận càng tốt
- *Thời gian chờ* (waiting time)
  - Tổng thời gian chờ trong hàng đợi ready
  - Các process nên được chia sẻ việc sử dụng CPU một cách công bằng (fair share)

*Tiêu chí định thời**

- *Thông lượng* (throughput)
  - Số lượng process hoàn tất trong một đơn vị thời gian
- *Thời gian đáp ứng* (response time)
  - Thời gian từ lúc có yêu cầu của người dùng (user request) đến khi có đáp ứng đầu tiên (lưu ý: đáp ứng đầu tiên, chứ không phải output)
  - Thường là vấn đề với các I/O-bound process

*Tiêu chí định thời**

- *Thời gian quay vòng* (turnaround time)
  - Thời gian để một process hoàn tất, kể từ lúc nạp vào hệ thống (submission) đến lúc kết thúc (termination)
  - Là một trị đặc trưng cần quan tâm với các process thuộc dạng CPU-bound
- *Thời gian quay vòng trung bình* (average turnaround time)

*Tiêu chí định thời **

- Độ lợi CPU – giữ CPU càng bận càng tốt
  - Tối đa hóa
- Thông lượng – số lượng process kết thúc việc thực thi trong một đơn vị thời gian
  - Tối đa hóa
- Turnaround time – tổng thời gian kể từ lúc bắt đầu đưa vào (submission) đến lúc kết thúc
  - Tối thiểu hóa
- Thời gian chờ – tổng thời gian một process chờ trong hàng đợi ready
  - Tối thiểu hóa
- Thời gian đáp ứng – thời gian từ khi đưa yêu cầu đến khi có đáp ứng đầu tiên
  - Tối thiểu hóa

*Có thể làm được?**

- Tất cả các tiêu chí không thể được tối ưu đồng thời vì có một số tiêu chí loại trừ lẫn nhau

*Tiêu chí định thời từ các góc nhìn**

- Hướng đến người sử dụng (user-oriented)
  - Thời gian quay vòng (turnaround time)
    - Thời gian từ lúc nạp process đến lúc process kết thúc
    - Cần quan tâm với các hệ thống xử lý bó (batch system)
  - Thời gian đáp ứng (response time)
    - Cần quan tâm với các hệ thống giao tiếp (interactive system)

*Tiêu chí định thời từ các góc nhìn**

- Hướng đến hệ thống (system-oriented)
  - Độ lợi CPU (CPU utilization)
  - Công bằng (fairness)
  - Thông lượng (throughput): số process hoàn tất trong một đơn vị thời gian

*Hai thành phần của chiến lược định thời**

- *Hàm lựa chọn* (selection function)
  - Xác định process nào trong ready queue sẽ được thực thi tiếp theo. Thường theo các tiêu chí như
    - w = tổng thời gian đợi trong hệ thống
    - e = thời gian đã được phục vụ
    - s = tổng thời gian thực thi của process (bao gồm cả trị e)

*Hai thành phần của chiến lược định thời**

- *Chế độ quyết định* (decision mode)
  - Chọn thời điểm hàm lựa chọn định thời thực thi
  - *Nonpreemptive*
    - Một process sẽ ở trạng thái running cho đến khi nó bị block hoặc nó kết thúc
  - *Preemptive*
    - Process đang thực thi có thể bị ngắt và chuyển về trạng thái ready
    - Tránh trường hợp một process độc chiếm (monopolizing) CPU

Nonpreemptive và preemptive

- Hàm định thời được thực hiện khi
  (1) Chuyển từ trạng thái running sang waiting
  (2) Chuyển từ trạng thái running sang ready
  (3) Chuyển từ trạng thái waiting, new sang ready
  (4) Kết thúc thực thi
- Trường hợp 1, 4 được gọi là định thời *nonpreemptive*
- Trường hợp 2, 3 được gọi là định thời *preemptive*

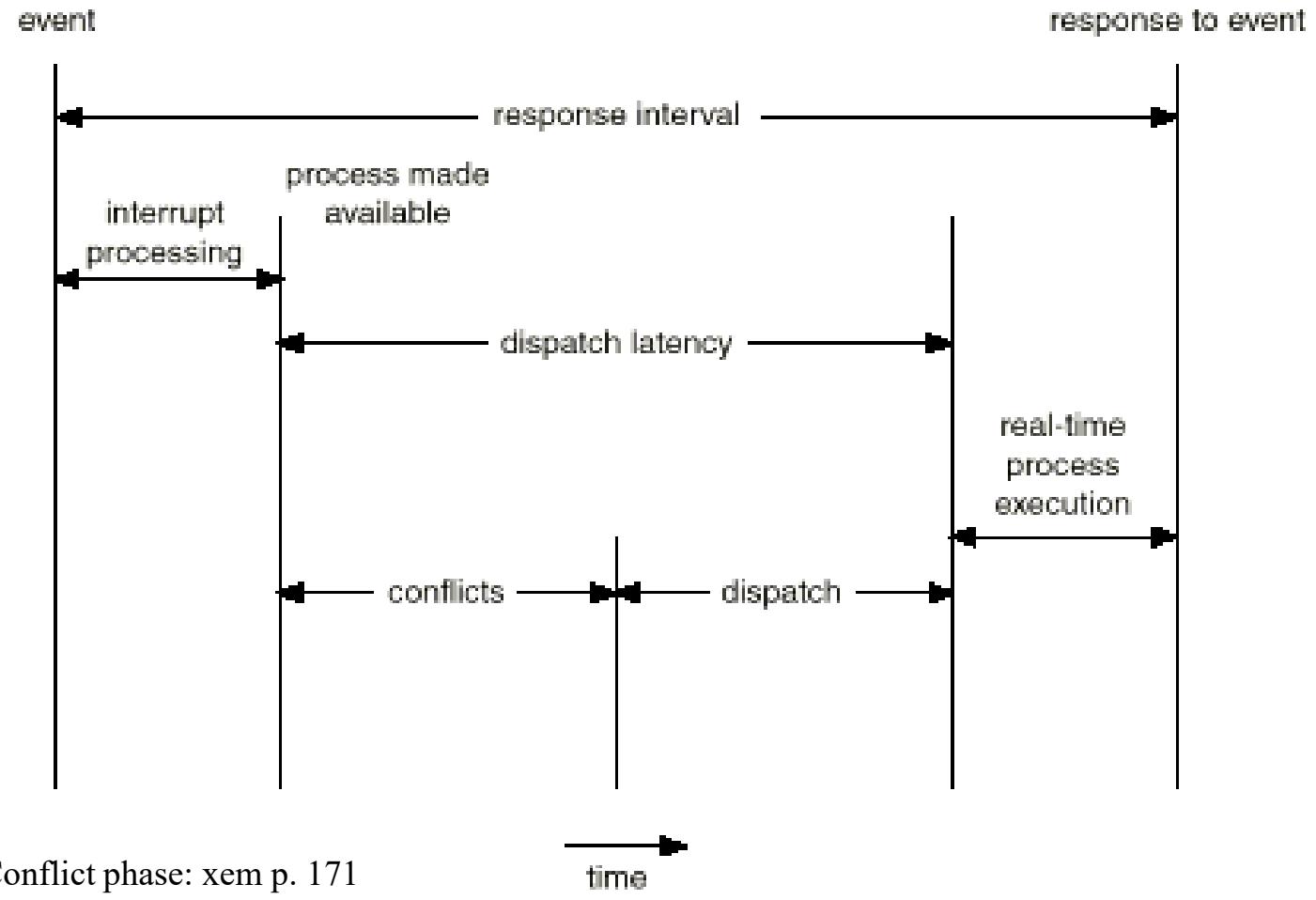
Nonpreemptive và preemptive (2)

- Hiện thực cơ chế nào khó hơn? Tại sao?
- Preemptive scheduling hiện thực khó hơn: cần phải duy trì sự nhất quán của:
  - Dữ liệu được chia sẻ giữa các process, và quan trọng hơn là
  - Các dữ liệu trong kernel (ví dụ các hàng đợi I/O)
- Ví dụ: trường hợp xảy ra preemption khi kernel đang thực thi một lời gọi hệ thống (do ứng dụng yêu cầu)
  - Rất nhiều hệ điều hành chờ cho các lời gọi hàm hệ thống kết thúc rồi mới preemption

*Dispatcher**

- Dispatcher sẽ chuyển quyền điều khiển CPU về cho process được chọn bởi bộ định thời ngắn hạn
- Bao gồm:
  - Chuyển ngữ cảnh (sử dụng thông tin ngữ cảnh trong PCB)
  - Chuyển về user mode
  - Nhảy đến vị trí thích hợp trong chương trình ứng dụng để khởi động lại chương trình (chính là program counter trong PCB)
- Công việc này gây ra phí tổn
  - *Dispatch latency*: thời gian mà dispatcher dừng một process và khởi động một process khác

Dispatch latency



Conflict phase: xem p. 171

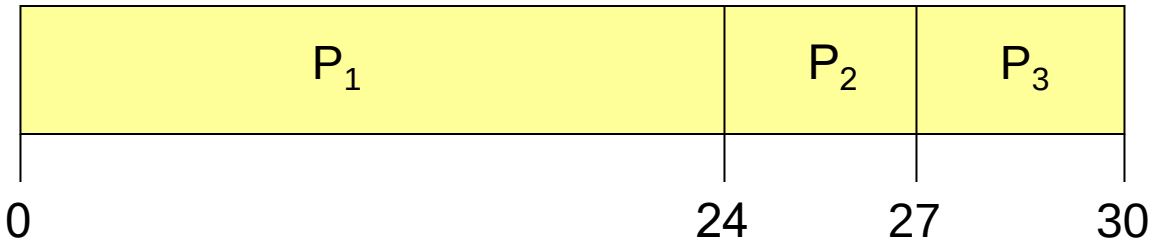
*First Come First Served (FCFS)**

- Hàm lựa chọn: chọn process đợi trong hàng đợi ready lâu nhất
- Chế độ quyết định: nonpreemptive
  - Một process sẽ được thực thi cho đến khi nó bị block hoặc kết thúc
- FCFS thường được quản lý bằng một FIFO queue

First Come First Served (FCFS)*

Process	Burst time (ms)
P_1	24
P_2	3
P_3	3

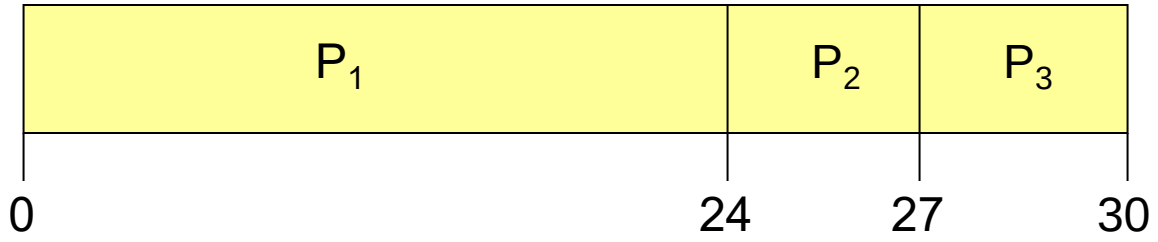
- Giả sử các process đến theo thứ tự P_1 , P_2 , P_3
- *Giản đồ Gantt* cho việc định thời là:



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

*First Come First Served (FCFS)**

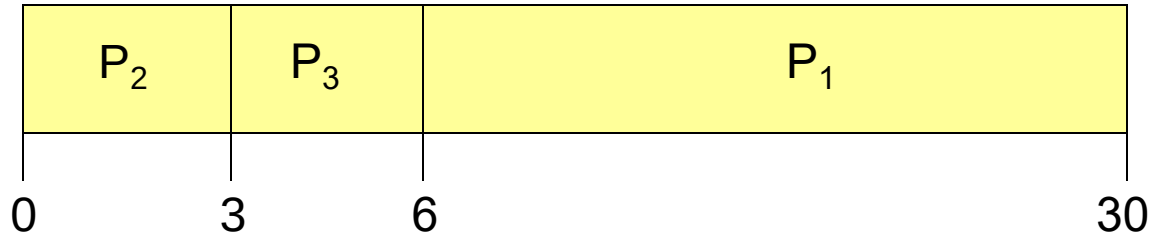
- Thời gian phục vụ trung bình =
- Thông lượng =
- Thời gian quay vòng=
  - Kiểm tra lại: Thời gian đợi = (thời gian quay vòng – thời gian phục vụ – dispatch latency)



First Come First Served (FCFS)*

- Giả sử các process đến theo thứ tự:

  P_2 , P_3 , P_1
- Giản đồ Gantt cho việc định thời là:



- Thời gian đợi cho $P_1 = 6, P_2 = 0, P_3 = 3$
- Thời gian đợi trung bình là: $(6 + 0 + 3)/3 = 3$
  - Tốt hơn rất nhiều so với trường hợp trước

*First Come First Served (FCFS)**

- FCFS không công bằng với các process có CPU burst ngắn. Các process này phải chờ trong thời gian dài (so với thời gian mà nó cần phục vụ) thì mới được sử dụng CPU. Điều này đồng nghĩa với việc FCFS "ưu tiên" các process thuộc dạng CPU bound.

- Câu hỏi: Liệu có xảy ra trường hợp *trì hoãn vô hạn định* (starvation hay indefinite blocking) với giải thuật FCFS?

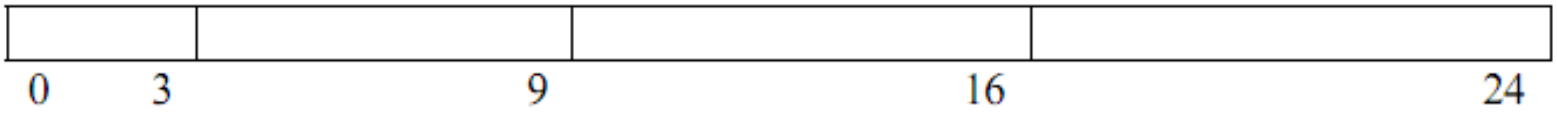
- FCFS thường được sử dụng trong các hệ thống bó (batch system)

Ví dụ thực tế

- Việc phục vụ khách trong nhà hàng
  - Thực khách sẽ đến và gọi món ăn cho mình
  - Mỗi món ăn cần thời gian chuẩn bị khác nhau
- Mục tiêu:
  - Giảm thời gian đợi trung bình của các thực khách
- Cách làm nào sẽ phù hợp?
  - Thông thường các nhà hàng sẽ phục vụ theo kiểu FCFS (!)

*Shortest Job First (SJF)**

Process	Burst time
P1	6
P2	8
P3	7
P4	3



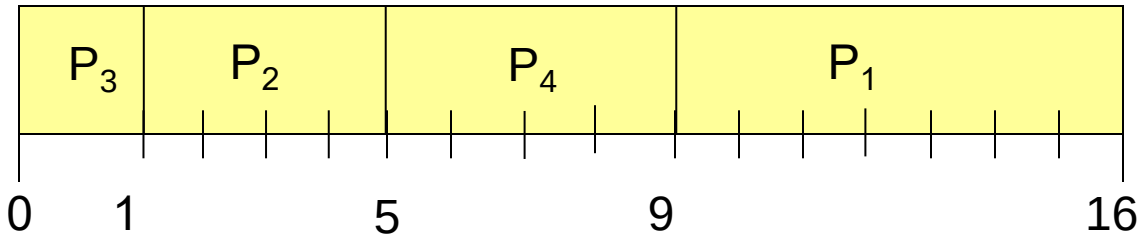
- Thời gian chờ đợi là 3 mili giây cho quá trình P1, 16 mili giây cho quá trình P2, 9 mili giây cho quá trình P3, và 0 mili giây cho quá trình P4.

  => thời gian chờ đợi trung bình là (3+16+9+0)/4 = 7 mili giây.
- Nếu dùng FCFS thì thời gian chờ đợi trung bình là 10.23 mili giây.

*Shortest Job First (SJF)**

Process	Burst time (ms)
P_1	7
P_2	4
P_3	1
P_4	4

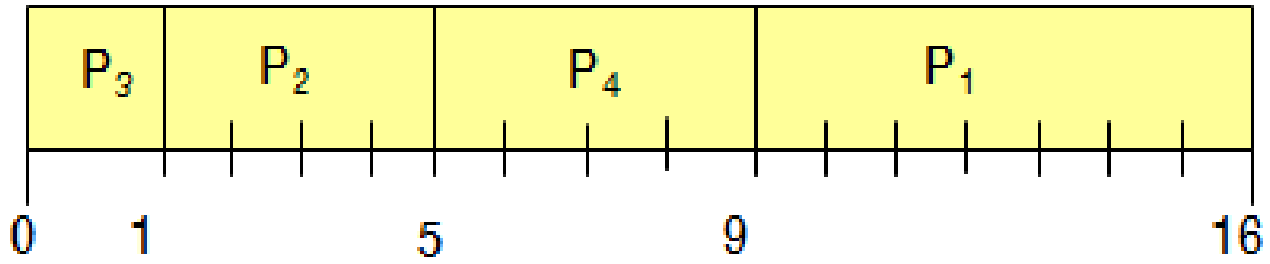
- Giản đồ Gantt khi định thời theo SJF (nonpreemptive)



- Thời gian đợi trung bình = (9 + 1 + 0 + 5)/4 = 3.75

*Shortest Job First (SJF)**

- Thời gian phục vụ trung bình =
- Thông lượng =
- Thời gian quay vòng =
  - Kiểm tra lại: Thời gian đợi = (thời gian quay vòng – thời gian phục vụ – dispatch latency)



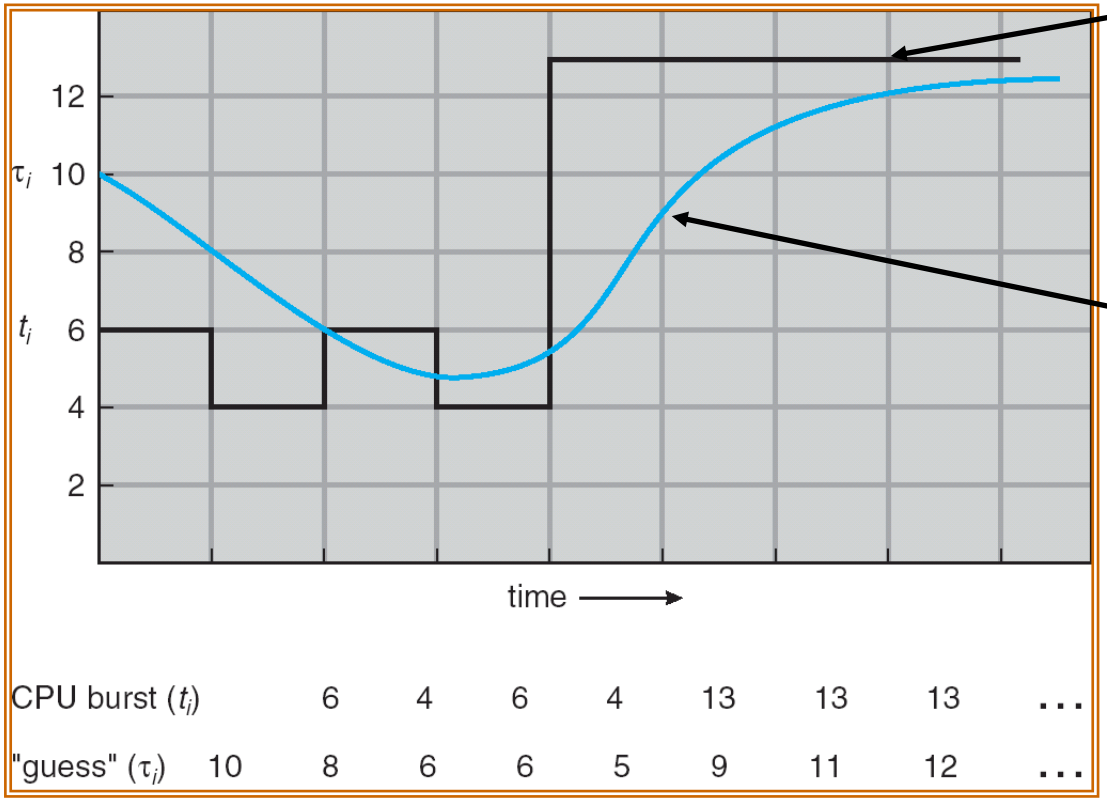
*Shortest Job First (SJF)**

- Tương ứng với mỗi process cần có độ dài của CPU burst tiếp theo
- Hàm lựa chọn: chọn process có độ dài CPU burst nhỏ nhất
- Chứng minh được: SJF tối ưu trong việc giảm thời gian đợi trung bình
- Nhược điểm: Cần phải ước lượng thời gian cần CPU tiếp theo của process
- Giải pháp cho vấn đề này?

*Dự đoán thời gian sử dụng CPU**

- (Thời gian sử dụng CPU chính là độ dài của CPU burst)
- Trung bình tất cả các CPU burst đo được trong quá khứ
- Nhưng thông thường những CPU burst càng mới càng phản ánh đúng hành vi của process trong tương lai
- Một kỹ thuật thường dùng là sử dụng *trung bình hàm mũ* (exponential averaging)
  - $\tau_{n+1} = \alpha\, t_n + (1 - \alpha)\, \tau_n$, $0 \le \alpha \le 1$
  - $\tau_{n+1} = \alpha\, t_n + (1- \alpha)\, \alpha\, t_{n-1} + \ldots + (1- \alpha)^j \alpha \tau_{n-j} + \ldots + (1- \alpha)^{n+1} \alpha \tau_0$
  - Nếu chọn $\alpha = ½$ thì có nghĩa là trị đo được t_n và trị dự đoán τ_n được xem quan trọng như nhau.

Dự đoán thời gian sử dụng CPU



Độ dài CPU burst đo được

Độ dài CPU burst dự đoán, với $\alpha = ½$ và $\tau_0 = 10$

*Shortest Job First (SJF)**

- SJF sử dụng ưu tiên ngầm định: công việc ngắn nhất được ưu tiên trước
  - Những công việc thuộc loại I/O bound thường có CPU burst ngắn
- Process có thời gian thực thi dài có thể bị trì hoãn vô hạn định nếu các process có thời gian thực thi ngắn liên tục vào
- Không thích hợp cho môi trường time-sharing khi không dùng preemption
  - Dù các CPU bound process có "độ ưu tiên" thấp
  - Nhưng một process không thực hiện I/O có thể độc chiếm hệ thống nếu nó là process đầu tiên vào hệ thống

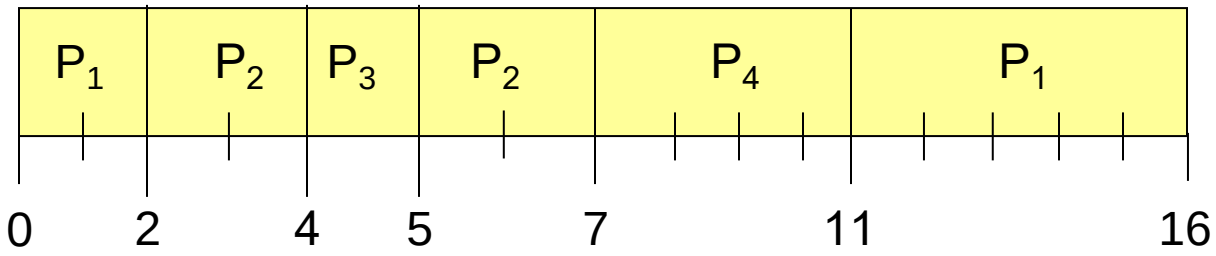
*Shortest Job First (SJF)**

- Chế độ quyết định: nonpreemptive
- Phiên bản preemptive của SJF:
  - Nếu một process mới đến mà có độ dài CPU burst nhỏ hơn thời gian cần CPU còn lại của process đang thực thi, thì thực hiện preempt process đang thực thi
  - Cách làm này còn được gọi là *Shortest-Remaining-Time-First* (SRTF)

Shortest Remaining Time First (SRTF)*

Process	Thời điểm đến	Burst time (ms)
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

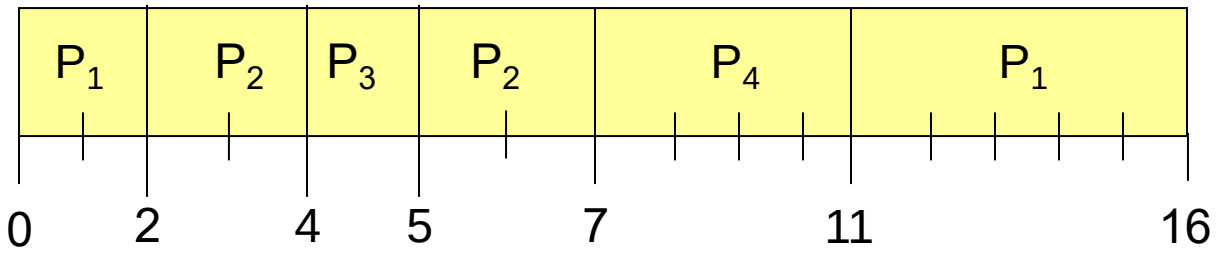
- Giản đồ Gantt khi định thời theo SRTF



- Thời gian đợi trung bình = (9 + 1 + 0 + 2)/4 = 3
  - Tốt hơn giải thuật nonpreemptive SJF

*Shortest Remaining Time First (SRTF)**

- Thời gian phục vụ trung bình =
- Thông lượng =
- Thời gian quay vòng =
  - Kiểm tra lại: Thời gian đợi = (thời gian quay vòng – thời gian phục vụ – dispatch latency)



*Shortest Remaining Time First (SRTF)**

- Tránh trường hợp các process có thời gian thực thi dài độc chiếm CPU
- Cần phải quản lý thời gian thực thi còn lại của các process
- Có thời gian quay vòng tốt hơn SJF
- Process có thời gian thực thi ngắn có độ ưu tiên cao

*Priority Scheduling**

- Mỗi process sẽ được gán một độ ưu tiên
- CPU sẽ được cấp cho process có độ ưu tiên cao nhất
- Định thời sử dụng độ ưu tiên có thể:
  - Preemptive hoặc
  - Nonpreemptive

*Gán độ ưu tiên**

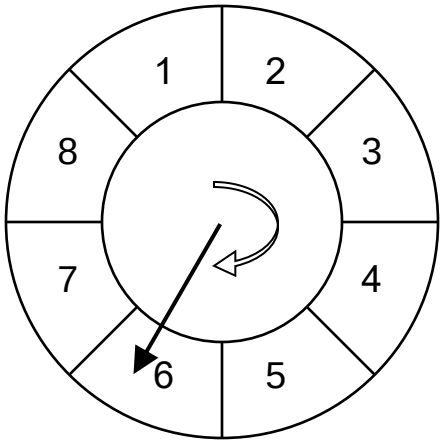
- SJF là một giải thuật định thời sử dụng độ ưu tiên với độ ưu tiên là thời-gian-sử-dụng-CPU-dự-đoán
- Gán độ ưu tiên còn dựa vào:
  - Yêu cầu về bộ nhớ
  - Số lượng file được mở
  - Tỉ lệ thời gian dùng cho I/O trên thời gian sử dụng CPU
  - Các yêu cầu bên ngoài ví dụ như: số tiền người dùng trả khi thực thi công việc

*Priority Scheduling**

- Vấn đề: trì hoãn vô hạn định – process có độ ưu tiên thấp có thể không bao giờ được thực thi
- Giải pháp: *aging* – độ ưu tiên của process sẽ tăng theo thời gian

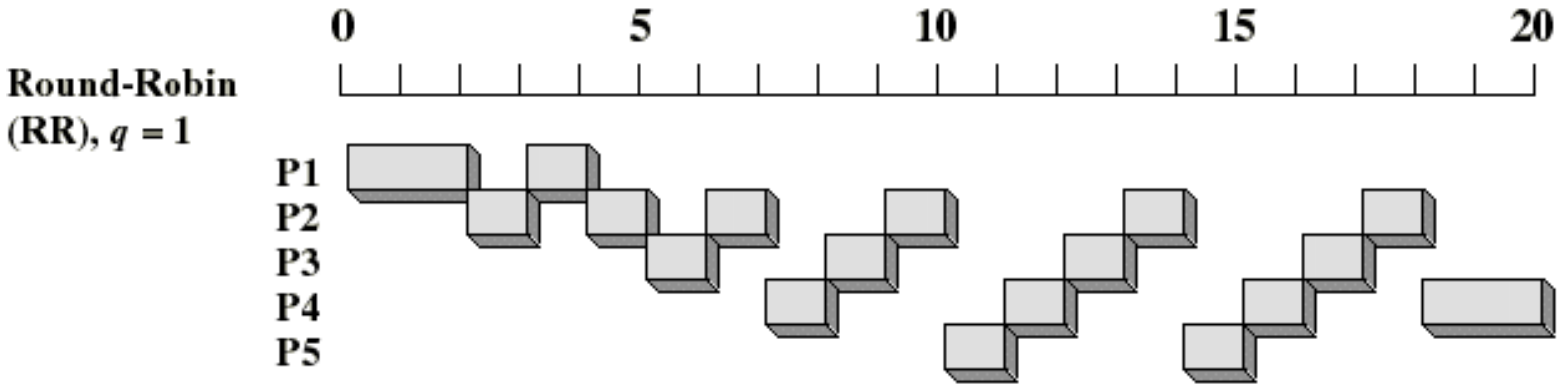
*Round Robin (RR)**

- Hàm lựa chọn: giống FCFS



*Round Robin (RR)**

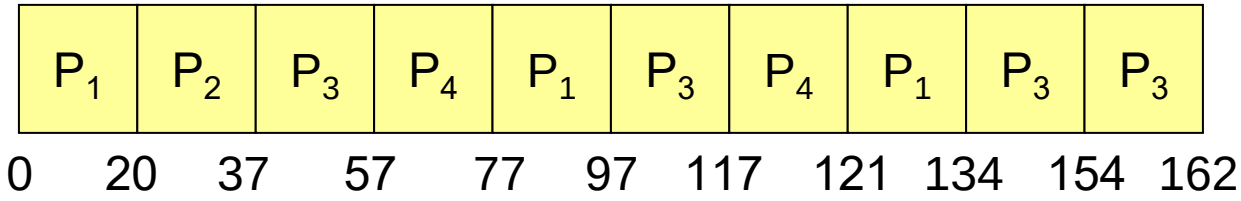
- Chế độ quyết định: preemptive
  - Khoảng thời gian tối đa cho phép (thường 10 - 100 ms) được đảm bảo bằng việc sử dụng timer interrupt
  - Process đang chạy hết thời gian sẽ được chuyển về cuối của hàng đợi ready



*Round Robin (RR)**

Process	Burst time (ms)
P_1	53
P_2	17
P_3	68
P_4	24

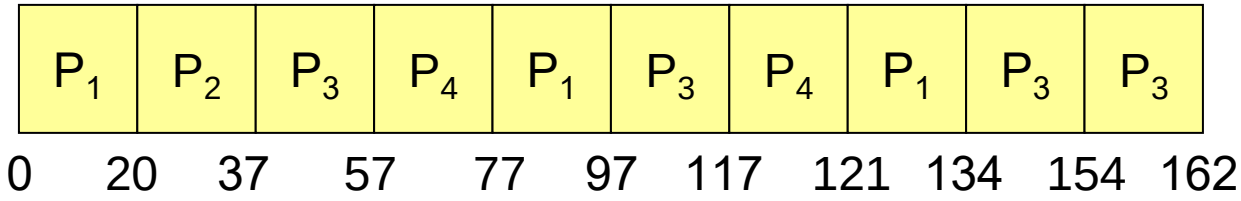
- Quantum time = 20 ms
- Giản đồ Gantt:



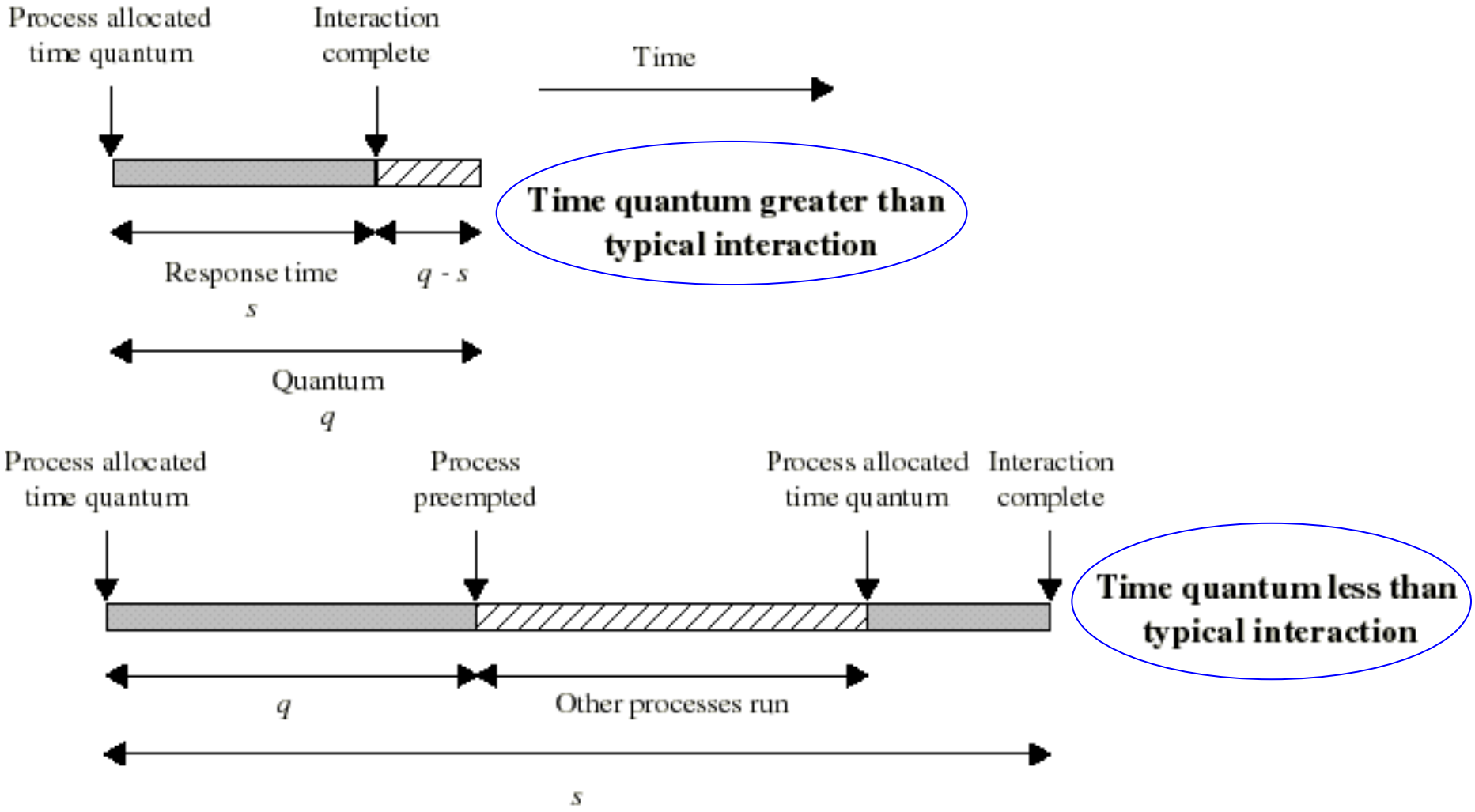
- Thường có thời gian quay vòng cao hơn SJF, nhưng lại có thời gian đáp ứng tốt hơn

*Round Robin (RR)**

- Thời gian phục vụ trung bình =
- Thông lượng =
- Thời gian quay vòng =
  - Kiểm tra lại: Thời gian đợi = (thời gian quay vòng – thời gian phục vụ – dispatch latency)



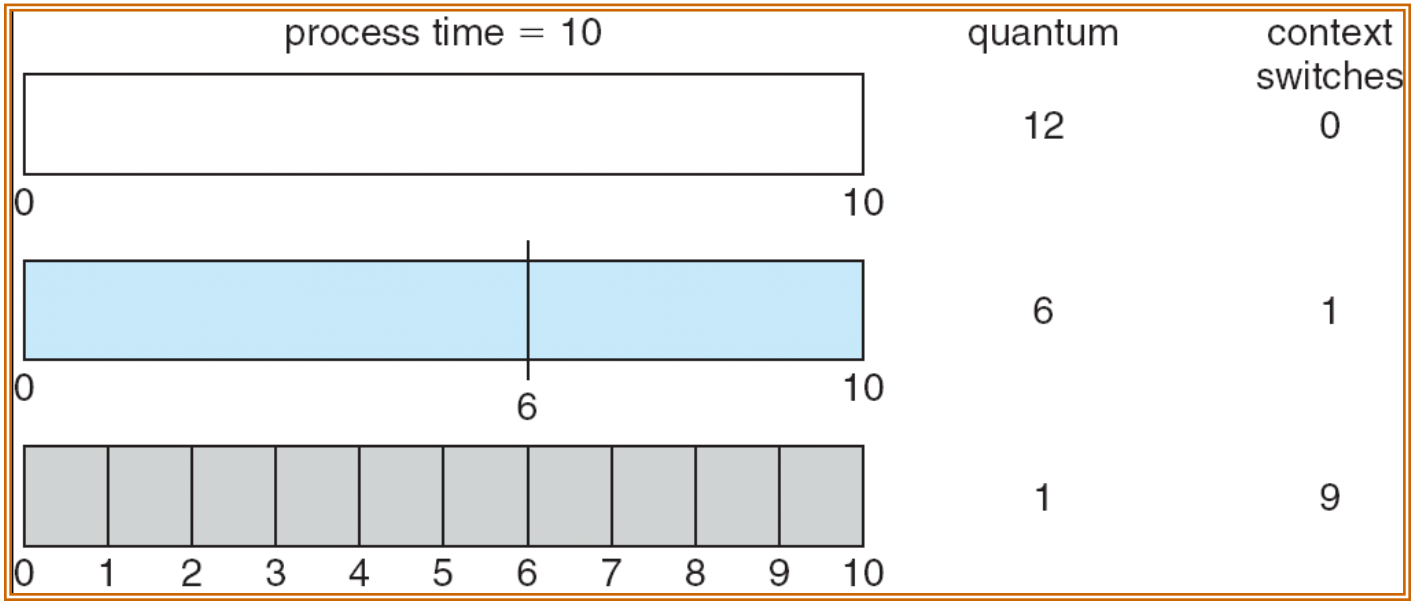
Quantum time cho Round Robin



Quantum time và chuyển ngữ cảnh

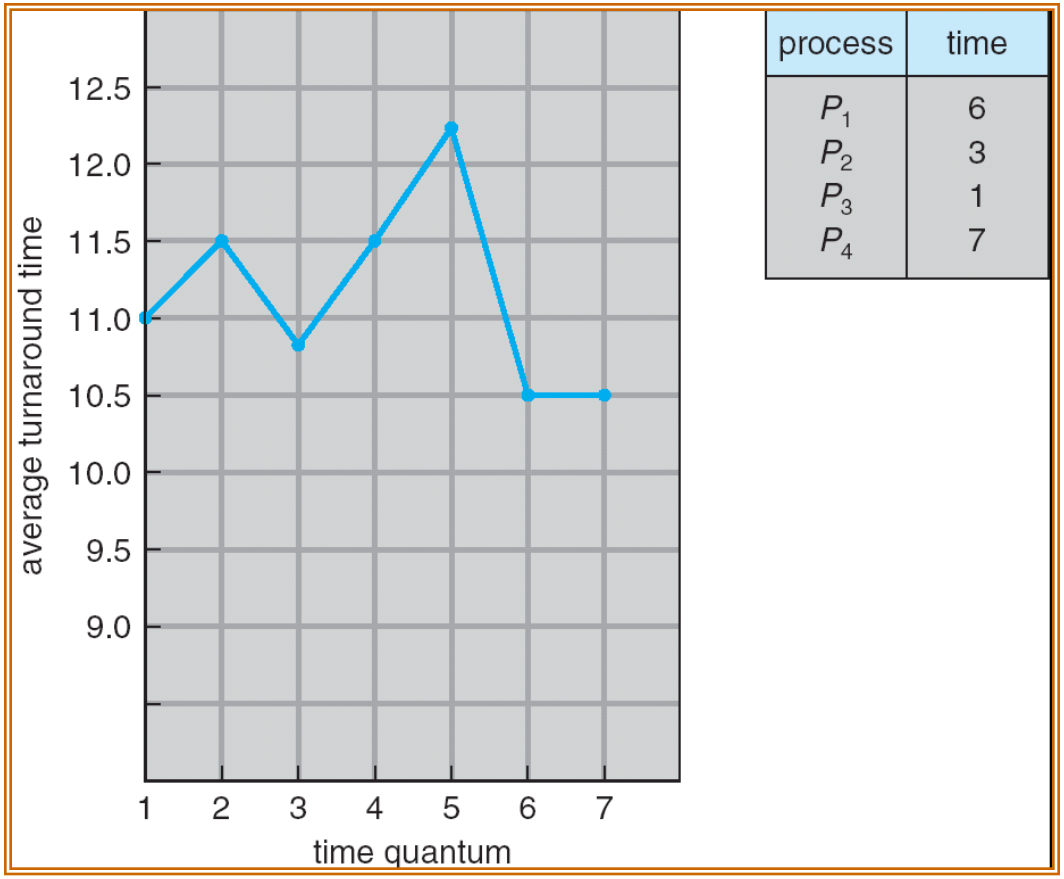
- Quantum time càng nhỏ thì càng có nhiều lần chuyển ngữ cảnh (context switch)

Số lần ngưng/tiếp tục tiến trình



Thời gian quay vòng và quantum time

- Thời gian quay vòng trung bình (average turnaround time) không chắc sẽ được cải thiện khi quantum lớn



*Quantum time cho Round Robin**

- Khi thực hiện process switch thì OS sẽ sử dụng CPU chứ không phải process của người dùng (*OS overhead*)
  - Dừng thực thi, lưu tất cả thông tin, nạp thông tin của process sắp thực thi
- Performance tùy thuộc vào kích thước của quantum time (còn gọi là time slice), và hàm phụ thuộc này không đơn giản
- Time slice ngắn thì đáp ứng nhanh
  - Vấn đề: có nhiều chuyển ngữ cảnh. Phí tổn sẽ cao.
- Time slice dài hơn thì throughput tốt hơn (do giảm phí tổn OS overhead) nhưng thời gian đáp ứng lớn
  - Nếu time slice quá lớn, RR trở thành FCFS.

Quantum time cho Round Robin

- Quantum time và thời gian cho process switch:
  - Nếu quantum time = 20 ms và thời gian cho process switch = 5 ms, như vậy phí tổn OS overhead chiếm 5/25 = 20%
  - Nếu quantum = 500 ms, thì phí tổn chỉ còn ≈ 1%
    - Nhưng nếu có nhiều người sử dụng trên hệ thống và thuộc loại interactive thì sẽ thấy đáp ứng rất chậm
  - Tùy thuộc vào tập công việc mà lựa chọn quantum time
  - Time slice nên lớn trong tương quan so sánh với thời gian cho process switch
  - Ví dụ với 4.3 BSD UNIX, time slice là 1 giây

Round Robin

- Nếu có *n* process trong hàng đợi ready, và quantum time là *q*, như vậy mỗi process sẽ lấy 1/*n* thời gian CPU theo từng khối có kích thước lớn nhất là *q*
  - Sẽ không có process nào chờ lâu hơn (*n* - 1)*q* đơn vị thời gian
- RR sử dụng một giả thiết ngầm là tất cả các process đều có tầm quan trọng ngang nhau
  - Không thể sử dụng RR nếu muốn các process khác nhau có độ ưu tiên khác nhau

Round Robin: nhược điểm

- Các process dạng CPU-bound vẫn còn được "ưu tiên"
  - Ví dụ:
    - Một I/O-bound process sử dụng CPU trong thời gian ngắn hơn quantum time và bị blocked để đợi I/O. Và
    - Một CPU-bound process chạy hết time slice và lại quay trở về hàng đợi ready queue (ở phía trước các process đã bị blocked)

*Multilevel Queue Scheduling**

Trường hợp các tiến trình có thể được phân thành nhóm (ví dụ: interactive và batch)

- Hàng đợi ready sẽ được chia thành nhiều hàng đợi riêng rẽ. Ví dụ:
  - *foreground* (cho công việc cần giao tiếp - interactive)
  - *background* (cho công việc dạng bó - batch)
- Mỗi hàng đợi sẽ có giải thuật định thời riêng. Ví dụ:
  - foreground: dùng RR
  - background: dùng FCFS

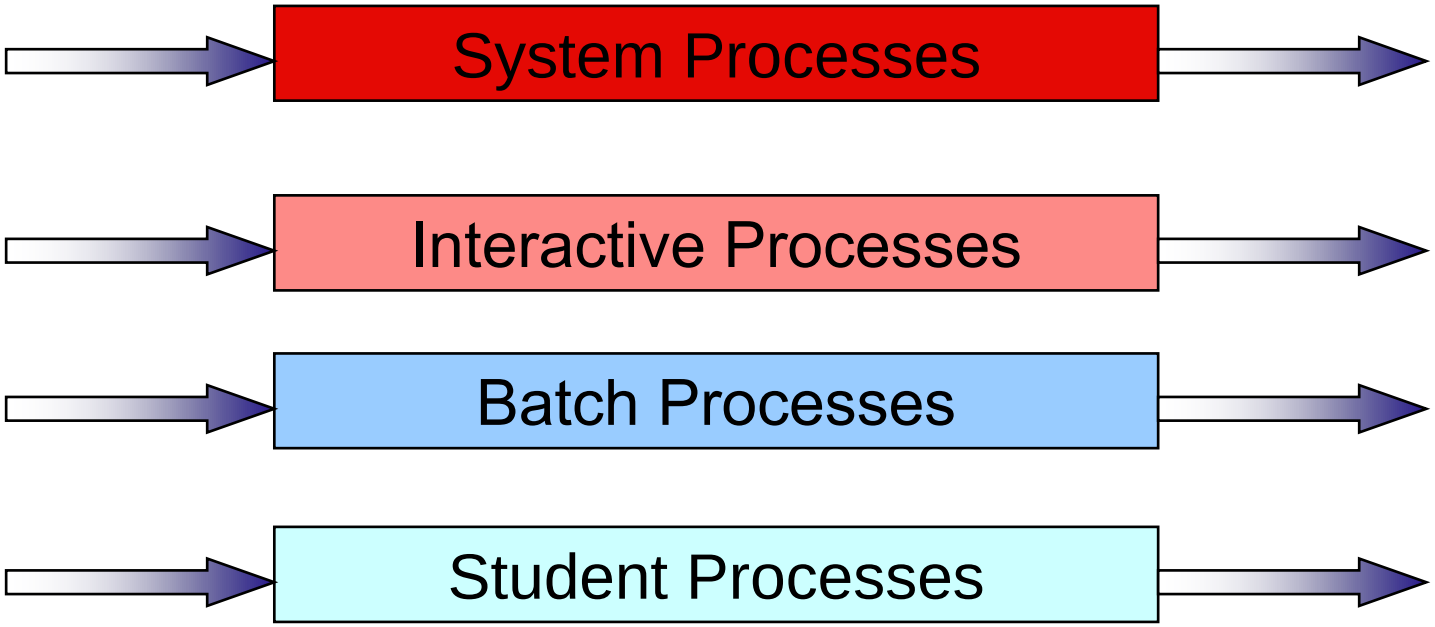
*Multilevel Queue Scheduling**

- Định thời cần phải thực hiện giữa các hàng đợi với nhau
  - Theo cách cố định (fixed priority scheduling) – ví dụ: phục vụ tất cả các process của foreground rồi mới đến background
    - Có khả năng xảy ra trì hoãn vô hạn định (starvation)
  - Chia thời gian (time slice) – mỗi hàng đợi sẽ được lấy một khoảng sử dụng CPU nhất định để định thời cho các process của mình. Ví dụ:
    - 80% cho foreground (dùng RR)
    - 20% cho background (dùng FCFS)

*Multilevel Queue Scheduling**

- Ví dụ phân nhóm các tiến trình

Độ ưu tiên cao nhất



Độ ưu tiên thấp nhất

*Multilevel Feedback Queue**

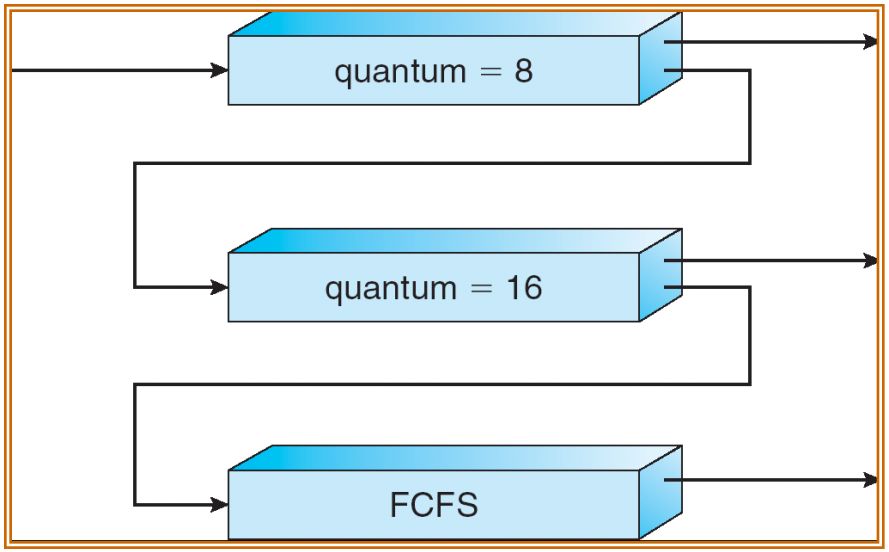
- Process có thể di chuyển giữa các queue tùy theo đặc tính của nó.

  Ví dụ:

  - Nếu một process sử dụng CPU quá lâu, nó sẽ bị di chuyển sang một hàng đợi có độ ưu tiên thấp hơn
  - Nếu một process chờ quá lâu trong một hàng đợi có độ ưu tiên thấp, nó sẽ được di chuyển lên hàng đợi có độ ưu tiên cao hơn (*aging*, giúp tránh starvation)

Multilevel Feedback Queue*

- Ví dụ: Có 3 hàng đợi
  - Q_0 , dùng RR với quantum 8 ms
  - Q_1 , dùng RR với quantum 16 ms
  - Q_2 , dùng FCFS
- Giải thuật
  - Công việc mới sẽ vào hàng đợi Q_0. Khi đến lượt mình, công việc sẽ được một khoảng thời gian là 8 milli giây. Nếu không kết thúc được trong 8 milli giây, công việc sẽ được đưa xuống hàng đợi Q_1
  - Tại Q_1, tương tự công việc sau khi chờ sẽ được cho một khoảng thời gian thực thi là 16 milli giây. Nếu hết thời gian này vẫn chưa kết thúc sẽ bị chuyển sang Q_2



Multilevel Feedback Queue

- Multilevel Feedback Queue được xác định bởi các thông số
  - Có bao nhiêu hàng đợi?
  - Với mỗi queue sử dụng giải thuật định thời nào?
  - Xác định thời điểm thăng cấp cho một process?
  - Làm sao để xác định thời điểm giáng cấp một process?
  - Xác định được hàng đợi nào process sẽ vào khi process đó cần thực thi?

Policy và Mechanism

Rất quan trọng trong định thời và phân phối tài nguyên

- *Policy*
  - Điều gì (what) nên (hay cần) làm
- *Mechanism*
  - Làm sao (how) để làm điều đó
- Ví dụ
  - Policy: tất cả người dùng cần được công bằng
  - Mechanism: sử dụng round robin
  - Policy: công việc được trả tiền cao có độ ưu tiên cao
  - Mechanism: sử dụng các giải thuật có preemptive

Định thời trên hệ thống multiprocessor

- Nếu có nhiều CPU thì có thể thực hiện việc chia tải
  - Phức tạp hơn so với định thời trên một processor
- Làm sao để chia tải?
  - Asymmetric multiprocessor
    - Một master processor sẽ thực hiện định thời cho tất cả các processor còn lại
  - Symmetric multiprocessor (SMP)
    - Hoặc mỗi processor có một hàng đợi ready riêng và bộ định thời riêng
    - Hoặc có một hàng đợi ready chung cho tất cả processors
      - Một processor được chọn làm scheduler cho các processor khác
      - Hoặc mỗi processor có bộ định thời riêng và tự chọn process từ hàng đợi chung để thực thi

Processor affinity

- Khi một process chạy trên một processor, có một số dữ liêu được cache trên bộ nhớ cache của processor
- Khi một process được di dời sang một processor khác
  - Cache của processor mới phải được **repopulated**
  - Cache của processor cũ phải được **invalidated**

  $\Rightarrow$ vấn đề phí tổn

Cân bằng tải

- Một processor có quá nhiều tải, trong khi những bộ xử lý khác thì lại rảnh
- Cân bằng tải sử dụng:
  - **Push migration**: một task đặc biệt sẽ định kỳ kiểm tra tải trên tất cả các processors và công việc sẽ được đẩy đến processor rảnh
  - **Pull migration**: processor rảnh sẽ lấy công việc từ processor đang bận
  - Một số hệ thống (ví dụ Linux) hiện thực cả hai
- Cần phải có sự cân bằng giữa load balancing và processor affinity

Đánh giá giải thuật định thời CPU

- *Deterministic modeling*
  - Định nghĩa trước một tập tải (workload) và khảo sát performance của các giải thuật trên cùng tập tải đó
  - Không tổng quát
- *Queuing models*
  - Sử dụng queuing theory để phân tích giải thuật
  - Sử dụng nhiều giả thiết để phục vụ việc phân tích
  - Không sát thực tế
- *Mô phỏng* (simulation)
  - Xây dựng bộ mô phỏng và chạy thử
    - Với tập tải giả (thường được sinh tự động)
    - Hoặc tập tải được ghi nhận từ thực tế
- *Hiện thực*
  - Viết mã của giải thuật và test nó trong hệ thống thực

*Tổng kết**

- Sự thực thi của một process
- Bộ định thời chọn một process từ hàng đợi ready
  - Dispatcher thực hiện switching
- Các tiêu chí định thời (thường xung đột nhau)
  - Độ lợi CPU, thời gian chờ, thời gian đáp ứng, thông lượng…
- Các giải thuật định thời
  - FCFS, SJF, Priority, RR, Multilevel Feedback Queue,…
- Định thời trên hệ thống multiprocessor (đọc thêm)
  - Processor affinity và cân bằng tải
- Đánh giá giải thuật (đọc thêm)
  - Mô hình, mô phỏng, hiện thực

Một số vấn đề bàn thêm

- Cách làm tốt nhất là adaptive
- Để thực hiện tối ưu hoàn toàn thì cần phải tiên đoán đúng tương lai (!)
  - Thực tế thì đa số các giải thuật lại cho kết quả gán độ ưu tiên cao nhất cho các process có nhu cầu ít nhất
  - Vấn đề định thời có xu hướng chuyển sang "tweak and see"

- Các tiêu chí nào nên tối ưu?
  - Có rất nhiều, tùy vào hệ thống, ngữ cảnh mà chọn lựa

Tham khảo

- Operating System Concepts. Sixth Edition. John Wiley & Sons, Inc. 2002. Silberschatz, Galvin, Gagne.
- Modern Operating Systems. Second Edition. Prentice Hall. 2001. Andrew S. Tanenbaum.

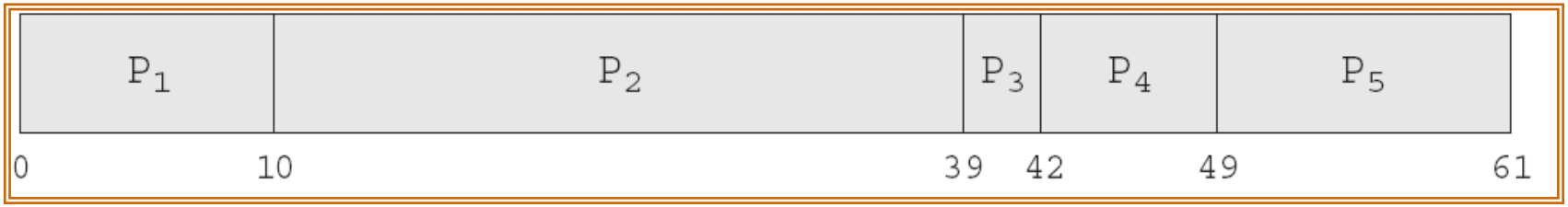
Bài tập (1)

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

- Tất cả đều đến ở thời điểm 0
- Xét các giải thuật FCFS, SJF, và RR với quantum time = 10
- Giải thuật nào cho
  - Thời gian đợi trung bình nhỏ nhất?
  - Thông lượng cao nhất?
  - Thời gian quay vòng trung bình của process nhỏ nhất?

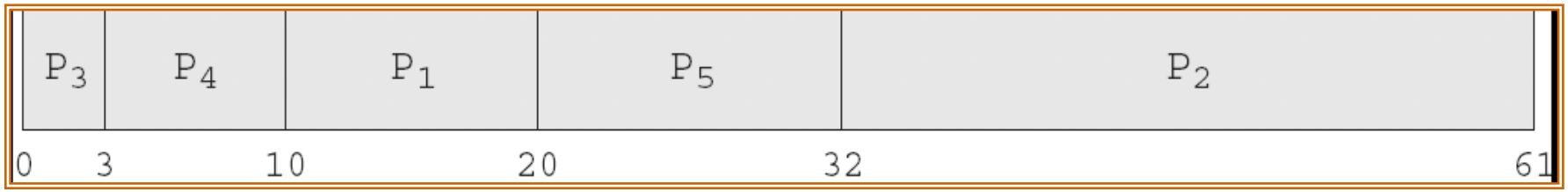
Bài tập (2)

- FCFS: thời gian đợi trung bình là 28 milli giây, hãy tính các thông số khác



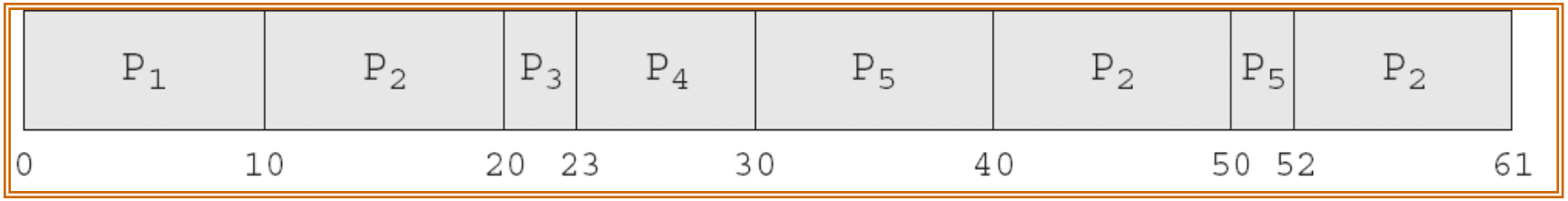
Bài tập (3)

- SJF (nonpreemptive): thời gian đợi trung bình là 13 milli giây, hãy tính các thông số khác

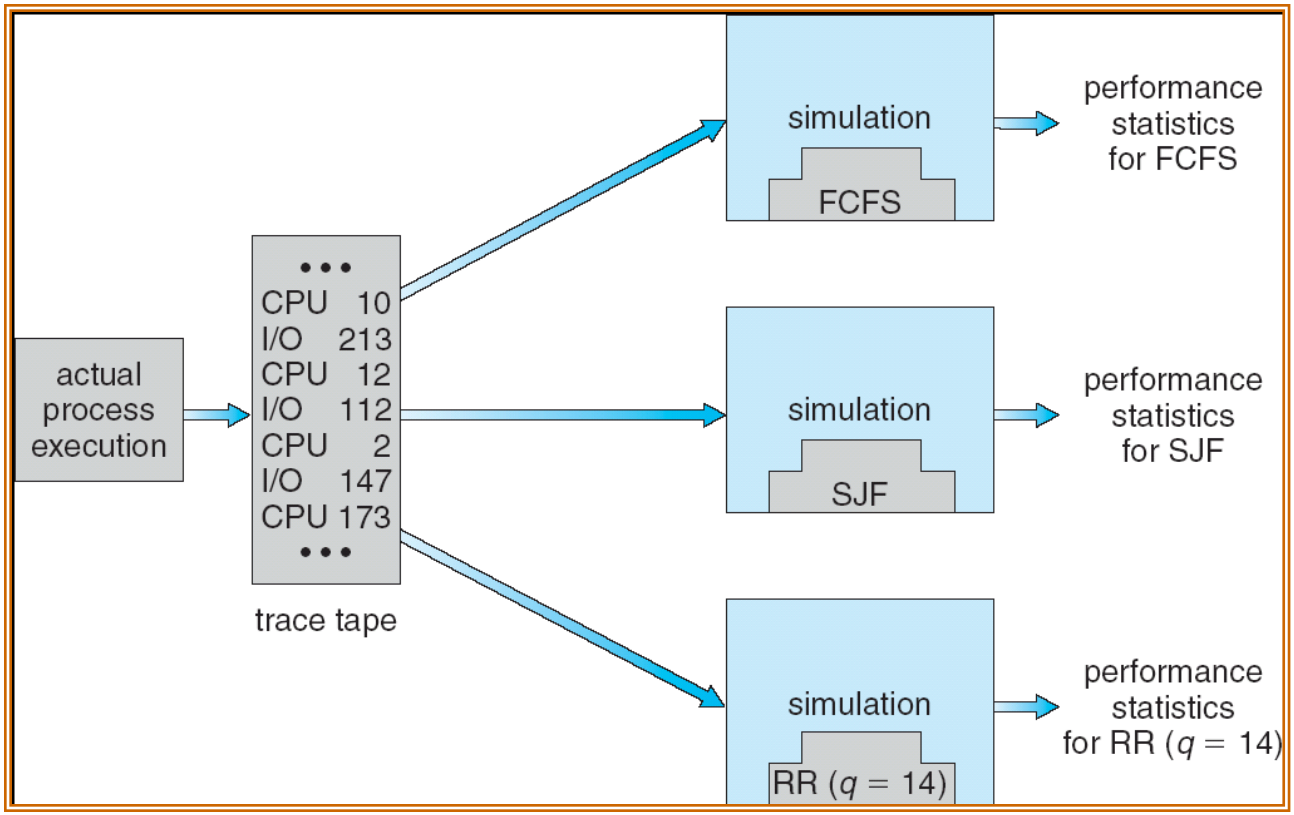


Bài tập (4)

- RR: thời gian đợi trung bình là 23 milli giây, hãy tính các thông số khác



Giả lập bộ định thời CPU



Định thời trong Linux

- Hai giải thuật: chia sẻ thời gian (time-sharing) và thời gian thực (real-time)
- Chia sẻ thời gian (time-sharing)
  - Ưu tiên dựa trên credit: process có nhiều credit nhất sẽ được chọn để thực thi
  - Khi có một ngắt thời gian credit sẽ bị giảm
  - Khi credit = 0, process khác sẽ được chọn
  - Khi tất cả các process có credit = 0, thực hiện recrediting dựa vào các yếu tố như độ ưu tiên và lịch sử (history)

Định thời trong Linux

- Thời gian thực (real-time)
  - Thời gian thực mềm (soft real-time)
  - Tương thích Posix.1b compliant – có 2 lớp
    - FCFS và RR
    - Process có độ ưu tiên cao nhất được thực thi trước tiên