HỆ ĐIỀU HÀNH
(Operation System)

Th.S Nguyễn Hoàng Thành
Email: thanhnh@ptithcm.edu.vn
Tel: 0909 682 711

QUẢN LÝ TIẾN TRÌNH

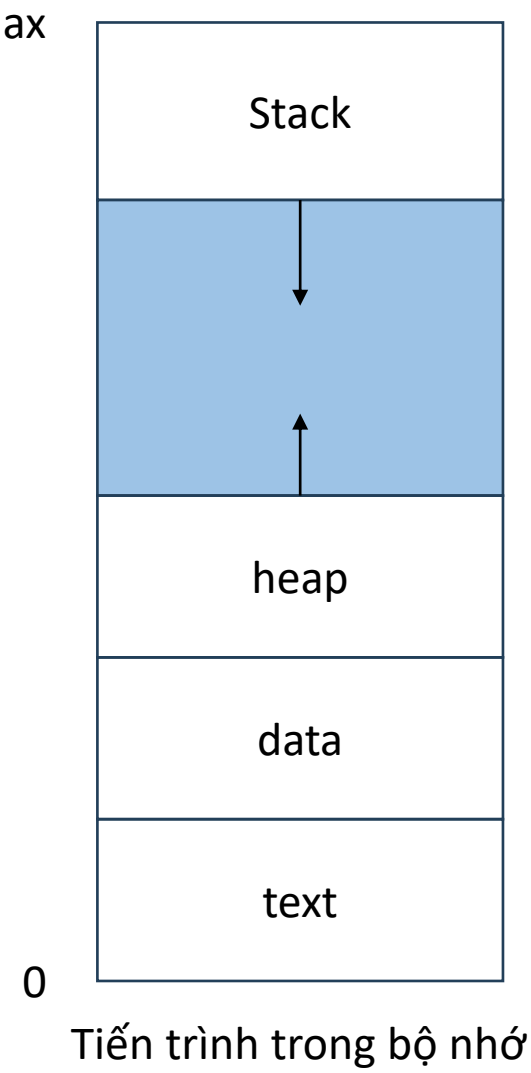
1. **Các khái niệm về tiến trình**
2. **Tiểu trình**
3. **Điều phối tiến trình**
4. **Đồng bộ các tiến trình**
5. **Tính trạng tắc nghẽn (deadlock)**

1. Khái niệm về tiến trình (process)

- Tiến trình là một chương trình đang xử lý
- Mỗi tiến trình có một không gian địa chỉ, một con trỏ lệnh, một tập các thanh ghi và stack riêng.
- Tiến trình có thể cần đến một số tài nguyên như CPU, bộ nhớ chính, các tập tin và thiết bị nhập/xuất.
- Hệ điều hành sử dụng bộ điều phối (scheduler) để điều phối việc thực thi của các tiến trình.
- Trong hệ thống có những tiến trình của hệ điều hành và tiến trình của người dùng.
- Một tiến trình bao gồm **Text section** (program code), **Data section** (chứa global variables).

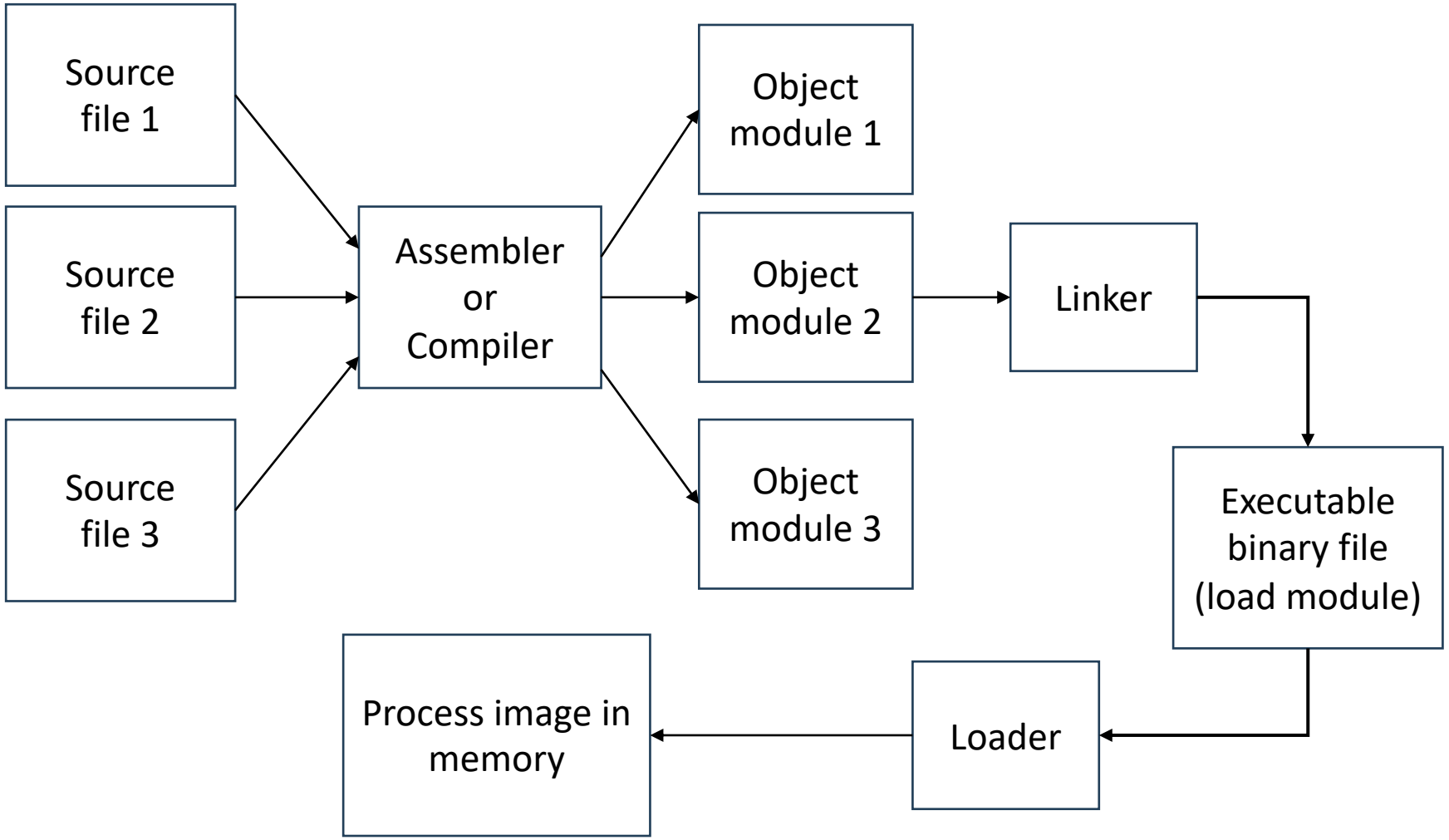
1. Khái niệm về tiến trình (process)

- Vùng code: chứa danh sách mã lệnh của CT
- Vùng static data: chứa các biến dữ liệu
- Được khai báo tường minh trong CT.
- Vùng dynamic data: chứa các vùng nhớ dữ liệu được cấp phát động này biến động theo thời gian.
- Vùng stack: phục vụ cho việc gọi hàm trong chương trình. Kích thước vùng này biến động theo thời gian.



Tiến trình trong bộ nhớ

1. Khái niệm về tiến trình (process)



Các bước nạp chương trình vào bộ nhớ

1. Khái niệm về tiến trình (process)

Các bước hệ điều hành khởi tạo tiến trình

- Cấp phát một định danh duy nhất (process number hay Process identifier, pid) cho Tiến trình
- Cấp phát không gian nhớ để nạp Tiến trình
- Khởi tạo khởi dữ liệu Process Control Block (PCB) cho Tiến trình

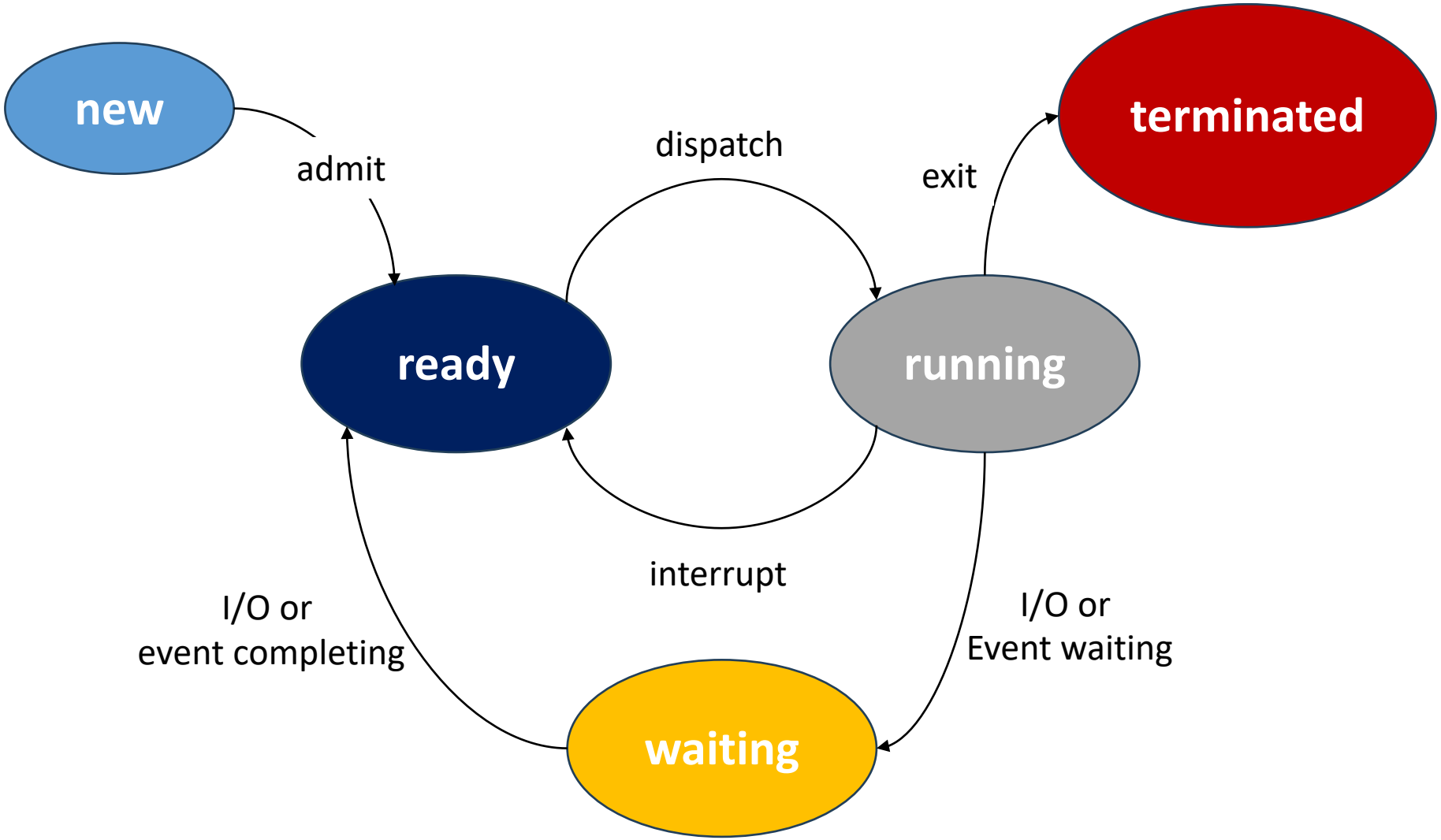
  PCB là nơi hệ điều hành lưu các thông tin về Tiến trình
- Thiết lập các mối liên hệ cần thiết (vd: sắp PCB vào hàng đợi định thời,…)

1. Khái niệm về tiến trình (process)

- **new:** Tiến trình vừa được tạo.
- **ready:** tiến trình đã đủ tài nguyên, chỉ còn cần CPU.
- **running:** các lệnh của uiến trình đang được thực thi.
- **waiting hay blocked:** tiến trình đang đợi I/O hoàn tất tín hiệu.
- **Terminated:** Tiến trình kết thúc.

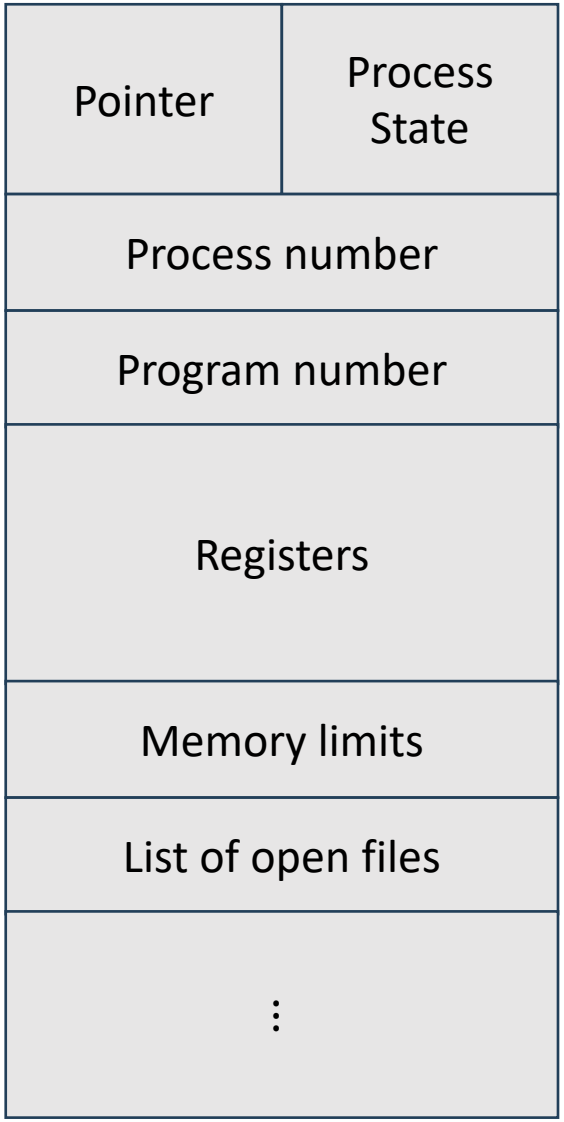
1. Khái niệm về tiến trình (process)

Chuyển đổi trạng thái giữa các tiến trình



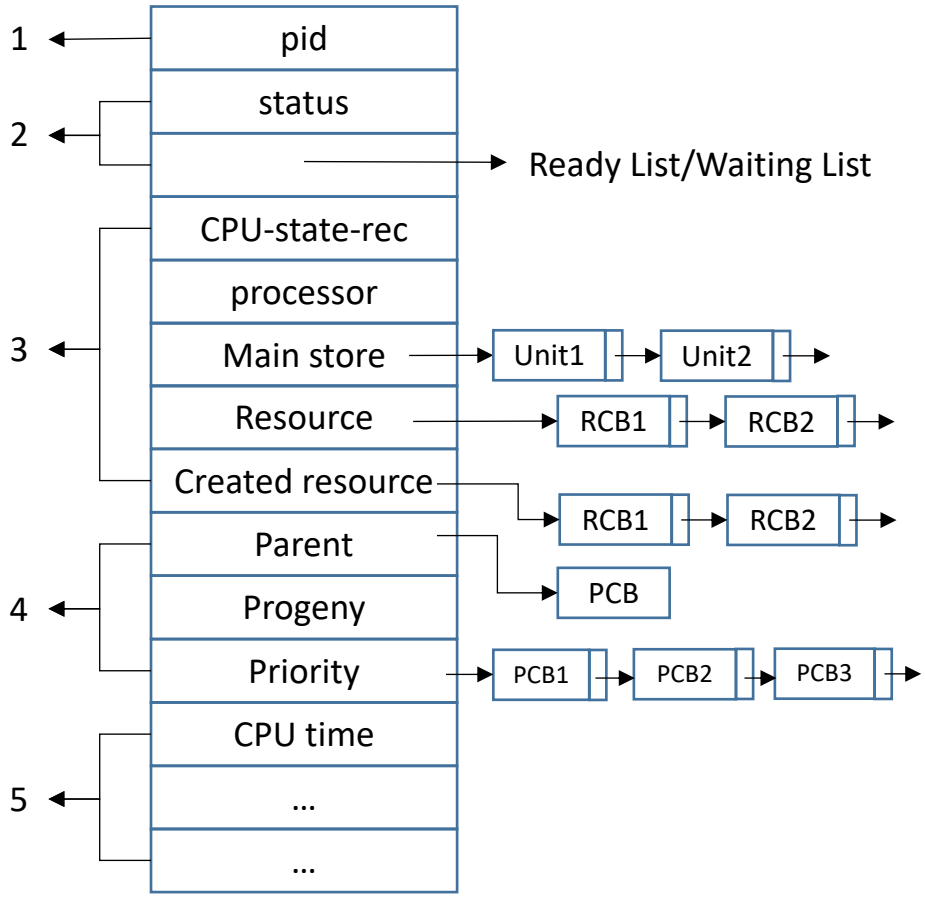
1. Khái niệm về tiến trình (process)

- Mỗi tiến trình trong hệ thống đều được cấp phát một ***Process Control Block*** (PCB)
- PCB là một trong các cấu trúc dữ liệu quan trọng nhất của hệ điều hành.



Cấu trúc dữ liệu khối quản lý tiến trình

- Khối quản lý tiến trình (PCB): là một vùng nhớ lưu trữ các thông tin mô tả cho tiến trình:
  - **Định danh của tiến trình (1)**
  - **Trạng thái tiến trình (2)**
  - **Ngữ cảnh của tiến trình (3)**
    - *Trạng thái CPU, Bộ xử lý, Bộ nhớ chính,Tài nguyên sử dụng,Tài nguyên tạo lập*
  - **Thông tin giao tiếp (4)**
    - *Tiến trình cha,Tiến trình con, Độ ưu tiên*
  - **Thông tin thống kê (5)**



Khối mô tả tiến trình

1. Khái niệm về tiến trình (process)

- **Chuyển ngữ cảnh (context switch)**
  - ***Ngữ cảnh (context)*** của một tiến trình là trạng thái của tiến trình.
  - Ngữ cảnh của tiến trình được biểu diễn trong PCB của nó.
  - ***Chuyển ngữ cảnh*** là công việc giao CPU cho tiến trình khác. Khi đó cần:
    - Lưu ngữ cảnh của tiến trình cũ vào CPU của nó.
    - Nạp ngữ cảnh từ PCB của tiến trình mới để tiến trình mới thực thi.

1. Khái niệm về tiến trình (process)

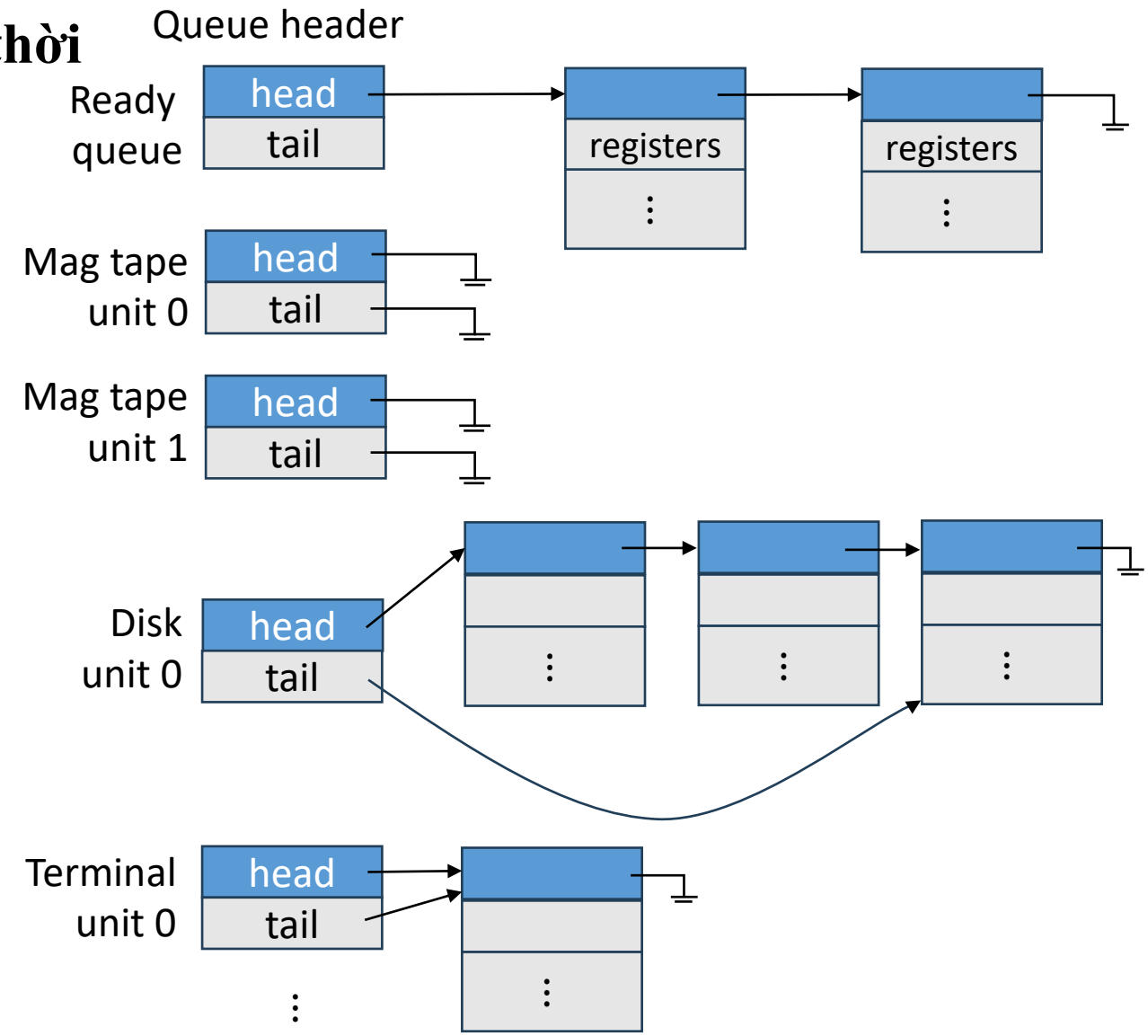
Định thời tiến trình

- Tại sao phải định thời?
  - **Multiprogramming**
    - Có **nhiều tiến trình** phải thực thi luân phiên nhau
    - Mục tiêu: **cực đại hiệu suất** sử dụng của CPU
  - **Time sharing**
    - Cho phép user tương tác với tiến trình đang thực thi
    - Mục tiêu: tối thiểu thời gian đáp ứng
- Một số khái niệm cơ bản
  - Các ***bộ định thời*** (scheduler)
  - Các ***hàng đợi định thời*** (scheduling queue)

1. Khái niệm về tiến trình (process)

Các hàng đợi định thời

- Job Queues
- Ready Queues
- Device Queues



1. Khái niệm về tiến trình (process)

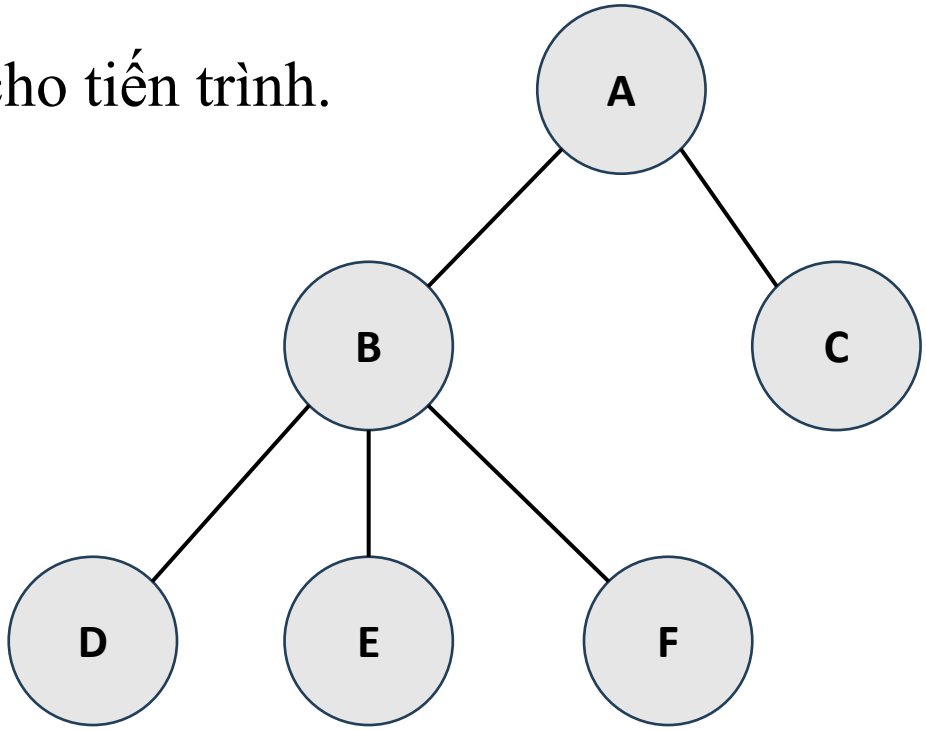
- **Thêm Medium-term scheduling**
- Đôi khi hệ điều hành (như time-sharing system) có thêm medium-term scheduling để điều **chỉnh mức độ multiprogramming** của hệ thống.
- **Medium-term scheduler**
  - ✓Chuyển tiến trình từ bộ nhớ sang đĩa (swap out)
  - ✓Chuyển tiến trình từ đĩa sang bộ nhớ (swap in)

1. Khái niệm về tiến trình (process)

- Tạo lập tiến trình (create)
- Kết thúc tiến trình (destroy)
- Tạm dừng tiến trình (suspend)
- Tái kích hoạt tiến trình (resume)
- Thay đổi độ ưu tiên tiến trình (change priority)

1. Khái niệm về tiến trình (process)

- Định danh cho tiến trình mới phát sinh
- Đưa tiến trình vào danh sách quản lý của hệ thống
- Xác định độ ưu tiên cho tiến trình
- Tạo PCB cho tiến trình
- Cấp phát các tài nguyên cho tiến trình.



1. Khái niệm về tiến trình (process)

❖Tạo tiến trình mới (process creation)

- Một tiến trình có thể tạo tiến trình mới thông qua một system call (ví dụ: hàm fork trong unix)
  - ✓Ví dụ: (Unix) Khi user đăng nhập hệ thống, một command interpreter (shell) sẽ được tạo ra cho user.
- Tiến trình được tạo là tiến trình con của tiến trình đang chạy (tiến trình cha). Quan hệ cha-con định nghĩa một *cây tiến trình*.

1. Khái niệm về tiến trình (process)

❖Tạo tiến trình mới (process creation)

- ➢Chia sẻ tài nguyên của tiến trình cha
  - ▪Tiến trình cha và con chia sẻ mọi tài nguyên
  - ▪Tiến trình con chia sẻ một phần tài nguyên của cha
- ➢Trình tự thực thi
  - ▪Tiến trình cha và con thực thi đồng thời (concurrently)
  - ▪Tiến trình cha đợi đến khi các tiến trình con kết thúc.

1. Khái niệm về tiến trình (process)

❖Kết thúc tiến trình

- ➢Tiến trình tự kết thúc
  - Tiến trình kết thúc khi thực thi lệnh cuối và gọi system routine **exit**.
- ➢Trình tự kết thúc do **tiến trình khác** (có đủ quyền, ví dụ: tiến trình cha của nó)
  - Gọi system routine abort với tham số là pid (process identifier) của tiến trình cần được kết thúc.
- ➢Hệ điều hành thu hồi tất cả các tài nguyên của tiến trình kết thúc (vùng nhớ, I/O buffer,…)

1. Khái niệm về tiến trình (process)

❖Cộng tác giữa các tiến trình

- Trong quá trình thực thi, các tiến trình có thể cộng tác (cooperate) để hoàn thành công việc.
- Các tiến trình cộng tác để
  - **Chia sẻ dữ liệu** (information sharing).
  - **Tăng tốc tính toán** (computational speedup)
  - Nếu hệ thống có nhiều CPU, chia công việc tính toán thành nhiều công việc tính toán như chạy song song.
- Thực hiện một công việc chung
  - Xây dựng một phần mềm phức tạp bằng cách chia thành các module/process hợp tác nhau.
- Sự cộng tác giữa các tiến trình yêu cầu hệ điều hành hỗ trợ **cơ chế giao tiếp** và **cơ chế đồng bộ** hoạt động của các tiến trình

2. Tiểu trình (Thread)

- Khái niệm tiến trình **truyền thống**:
  - Không gian địa chỉ (text section, data section)
  - **Một luồng thực thi duy nhất** (single thread of execution):
    - Program counter
    - Các register
    - Stack
  - Các tài nguyên khác (các open file, các quá trình con,…)

2. Tiểu trình (Thread)

❖Mở rộng khái niệm tiến trình truyền thống bằng cách hiện thực nhiều luồng thực thi trong **cùng một môi trường** của tiến trình.

❖Khi đó tiến trình gồm:

- ➢Không gian địa chỉ (text section, data section)
- ➢**Một hay nhiều luồng thực thi** (thread of execution), mỗi luồng thực thi (thread) **có riêng**
  - ▪Program counter
  - ▪Các register
  - ▪Stack
- ➢Các tài nguyên khác (các open file, các tiến trình con,…)

2. Tiểu trình (Thread)

❖Mô hình đa luồng:

- ➢Các thread trong cùng một process chia sẻ code section, data section và tài nguyên khác (các file đang mở, …) của process.
- ➢Tiến trình **đa luồng** (**multithreaded** process) là tiến trình có nhiều luồng.

2. Tiểu trình (Thread)

❖Mô hình đa luồng:

➢Ưu điểm:

- **Tính đáp ứng** (responsiveness) cao cho các ứng dụng tương tác multithreaded.
- **Chia sẻ tài nguyên** (resource sharing): ví dụ memory
- **Tiết kiệm** chi phí hệ thống (lợi ích kinh tế)
  - ✓Chi phí tạo/quản lý thread nhỏ hơn so với tiến trình
  - ✓Chi phí chuyển ngữ cảnh giữa các thread nhỏ hơn so với tiến trình
- **Tận dụng kiến trúc** đa xử lý (multi processor)
- Mỗi thread chạy trên một processor riêng, do đó tăng mức độ **song song** của chương trình.

2. Tiểu trình (Thread)

User Thread

❖Một **thư viện thread** (thread library, run-time system) được hiện thực trong user space để hỗ trợ các tác vụ lên thread.

➢Thư viện thread cung cấp các hàm khởi tạo, định thời và quản lý thread như:

- **thread_create**
- **thread_exit**
- **thread_wait**
- **thread_yield**

➢Thư viện thread dùng ***Thread Control Block*** (**TCB**) để lưu trạng thái của user thread (program counter, các register, stack)

❖Kernel không biết sự có mặt của user thread

2. Tiểu trình (Thread)

Kernel Thread

Cơ chế multi-threading được hệ điều hành trực tiếp hỗ trợ

- ❖Kernel quản lý cả process và các thread
  - ➢Việc định thời CPU được kernel thực hiện trên thread
  - ➢Cơ chế multi-threading được hỗ trợ bởi kernel
- ❖Khởi tạo và quản lý các thread chậm hơn
  - ➢Tận dụng được lợi thế của kiến trúc multi-processor
  - ➢Thread bị blocked không kéo theo các thread khác bị blocked.
- ❖Mộ số hệ thống multi-threading (multi-tasking)
  - ➢Windows
  - ➢Solaris
  - ➢Linux

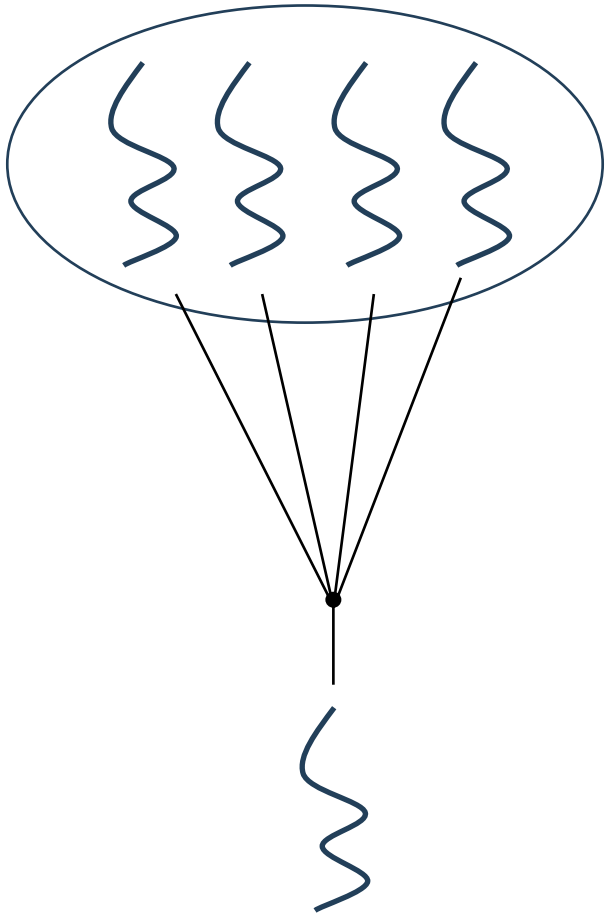
2. Tiểu trình (Thread)

- Một số mô hình hiện thực thread
  - Mô hình ***many-to-one***
  - Mô hình ***one-to-one***
  - Mô hình ***many-to-many***

2. Tiểu trình (Thread)

Mô hình many-to-one

- Nhiều user-level thread "chia sẻ" một kernel thread để thực thi
  - Việc quản lý thread được thực hiện thông qua các hàm của một thread library được gọi ở user level
  - **Blocking problem**: Khi một thread trở nên blocked thì kernel thread cũng trở nên blocked, do đó mỗi thread khác của process cũng sẽ trở nên blocked.
- Có thể được hiện thực đối với hầu hết các hệ điều hành.

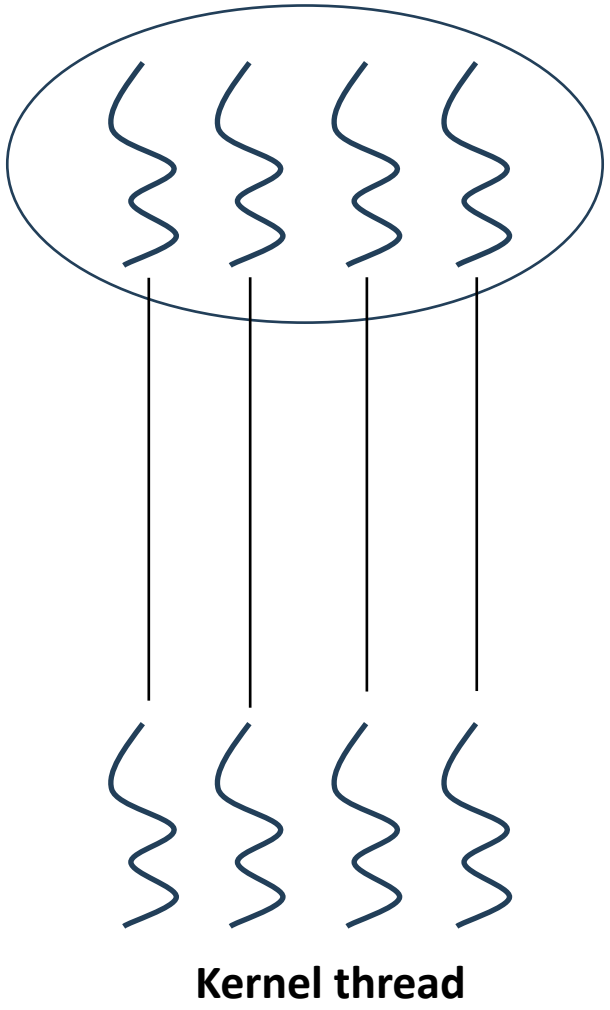


Kernel thread

2. Tiểu trình (Thread)

Mô hình one-to-one

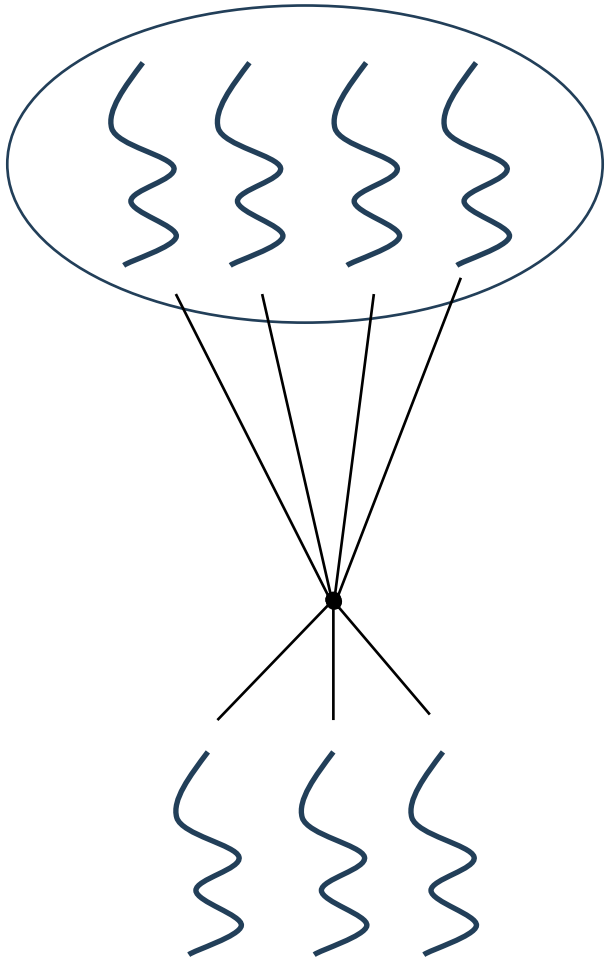
- Mỗi user-level thread thực thi thông qua một kernel thread riêng của nó
  - Mỗi khi một user thread được tạo ra thì cũng cần tạo một kernel thread tương ứng.
- Hệ điều hành phải có cơ chế cung cấp được nhiều kernel thread cho một tiến trình
- Ví dụ: Windows NT/2000



2. Tiểu trình (Thread)

Mô hình many-to-many

- Nhiều user-level thread được phân chia thực thi (multiplexed) trên một số kernel thread.
  - Tránh được một số khuyết điểm của hai mô hình many-to-one và one-to-one.
- Ví dụ
  - Solaris 2
  - Windows NT/2000 với package ThreadFiber



3. Điều phối tiến trình

Một số khái niệm cơ bản

- Chu kỳ CPU-I/O: gồm chu kỳ thực thi CPU (CPU burst) và chu kỳ chờ đợi vào ra (I/O burst).
- ***CPU-bound*** process có thời gian sử dụng CPU nhiều hơn thời gian sử dụng I/O/
- ***I/O bound process*** dùng phần lớn thời gian để đợi I/O.

3. Điều phối tiến trình

❖Trong các hệ thống **multi-tasking**

- 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**

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

- Phân loại các hoạt động điều phối
  - ***Điều phối dài hạn*** (long-term scheduling): xác định process nào được chấp nhận **vào hệ thống**.
  - ***Điều phối trung hạn*** (medium-term scheduling): xác định process nào được **đưa vào (swap in)**, **đưa ra khỏi (swap out)** bộ nhớ chính
  - ***Điều phối ngắn hạn*** (short-term scheduling): xác định process nào được **thực thi tiếp theo**

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

❖Điều phối dài hạn

- ➢Xác định chương trình nào sẽ được đưa vào hệ thống để thực thi.
- ➢Quyết định độ-đa-lập-trình (degree of multi-programming)
- ➢Nhiều process được đưa vào hệ thống:
  - Khả năng các process bị block sẽ giảm.
  - Sử dụng CPU hiệu quả hơn
  - Mỗi process được phân chia khoảng thời gian sử dụng CPU thấp hơn
- ➢Không phân biệt process là CPU-bound hay I/O bound

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

❖Điều phối trung hạn

- Quyết định việc **đưa process vào bộ nhớ chính, hay ra khỏi bộ nhớ chính**.
- Phụ thuộc vào yêu cầu quản lý multi-programming
  - ✓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**
  - ✓Quá nhiều process thì sẽ làm tăng việc truy xuất đĩa, do đó cần phải lựa chọn độ-đa-lập-trình cho phù hợp.
- Phần mềm quản lý bộ nhớ đảm nhiệm

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

❖Điều phối ngắn hạn

- Xác định process nào được thực thi tiếp theo, còn gọi là Điều phối 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

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

- **Độ lợi CPU** (CPU ultization)
  - Khoảng thời gian CPU bận, từ 0% đến 100%
  - Cần giữ cho CPU càng bận càng tốt
- **Thời gian chờ** (waiting time)
  - Thời gian chờ trong hàng đợi ready
  - Các process nên được chia sẻ việc sử dụng CPU một cách công bằng (fair share)
- **Thông năng** (Throghtput)
  - Số lượng process hoàn tất trong một đơn vị thời gian

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

- **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.
  - Thường là vấn đề với các I/O bound process
- **Thời gian quay vòng** (turn around time)
  - Thời gian để một process hoàn tất, kể từ lúc 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 turn around time)

3. Điều phối tiến trình

Các tiêu chí điều phối tiến trình

- **Hướng đến người sử dụng** (user-oriented)
  - Thời gian quay vòng (turn around time).
    - Thời gian từ lúc nạp process đến lúc process kết thúc
    - Quan tâm với các hệ thống xử lý bó (batch system)
  - Thời gian đáp ứng (response time)
    - Quan tâm các hệ thống giao tiếp (interactive system)
- **Hướng đến hệ thống** (system-oriented)
  - Độ lợi CPU (CPU utilzation)
  - Công bằng (fairness)
  - Thông năng (throughtpput): số process hoàn tất trong một đơn vị thời gian.

3. Điều phối tiến trình

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 – Không độc quyền***
    - Một process sẽ ở trạng thái running cho đến khi nó bị block hoặc nó kết thúc.
  - ***Preemptive – Độc quyền***
    - Process đang thực thi có thể bị ngắt và chuyển về trạng thái ready.
    - Tránh trường hợp một process độc chiếm (monôplizing) CPU

3. Điều phối tiến trình

Hai thành phần của chiến lược định thời

- Hàm định thời có thể được thực thi khi có tiến trình
  - Chuyển từ trạng thái running sang waiting (1)
  - Chuyển từ trạng thái running sang ready (2)
  - Chuyển từ trạng thái waiting, new sang ready (3)
  - Kết thúc thực thi (4)
- Định thời ***nonpreemtive***: chỉ thực thi hàm định thời trong trường hợp (1) và (4)
- Định thời ***preemtive***: ngoài trường hợp (1) và (4) còn thực thi thêm hàm định thời trong trường hợp (2) hoặc (3) hoặc cả hai.

3. Điều phối tiến trình

❖ Các giải thuật điều phối tiến trình

❖ First Come First Served (FCFS)

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

3. Điều phối tiến trình

❖ Các giải thuật điều phối tiến trình

❖ First Come First Served (FCFS)

Process	Burst time (ms)
P_1	21
P_2	6
P_3	6

- 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 $P_1 = 0, P_2 = 21, P_3 = 27$
- Thời gian đợi trung bình: $(0 + 21 + 27)/3 = 16$

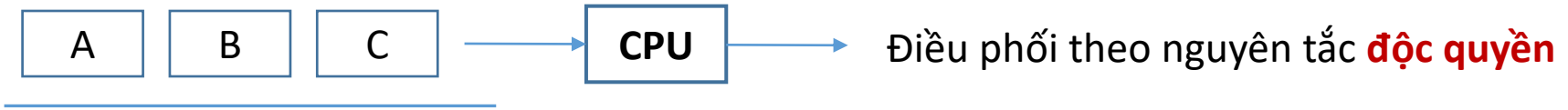
3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **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) mới được sử dụng CPU, hay FCFS "ưu tiên" các process thuộc dạng **CPU bound**.
  - FCFS có giải quyết tránh trường hợp **trì hoãn vô hạn định** (starvation hay indefinite blocking)?
  - FCFS thường được sử dụng trong các hệ thống bó (**batch system**)

3. Điều phối tiến trình - FCFS

Ready List



Điều phối theo nguyên tắc **độc quyền**

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P_1	0	24
P_2	1	3
P_3	2	3

P_1	P_2	P_3
0	24	27
		30

- Thời gian chờ đợi được xử lý là 0 đối với P1, (24 -1) với P2 và (27-2) với P3.
- Thời gian chờ trung bình là $\frac{0+23+25}{3} = 16$ miliseconds.

3. Điều phối tiến trình - FCFS

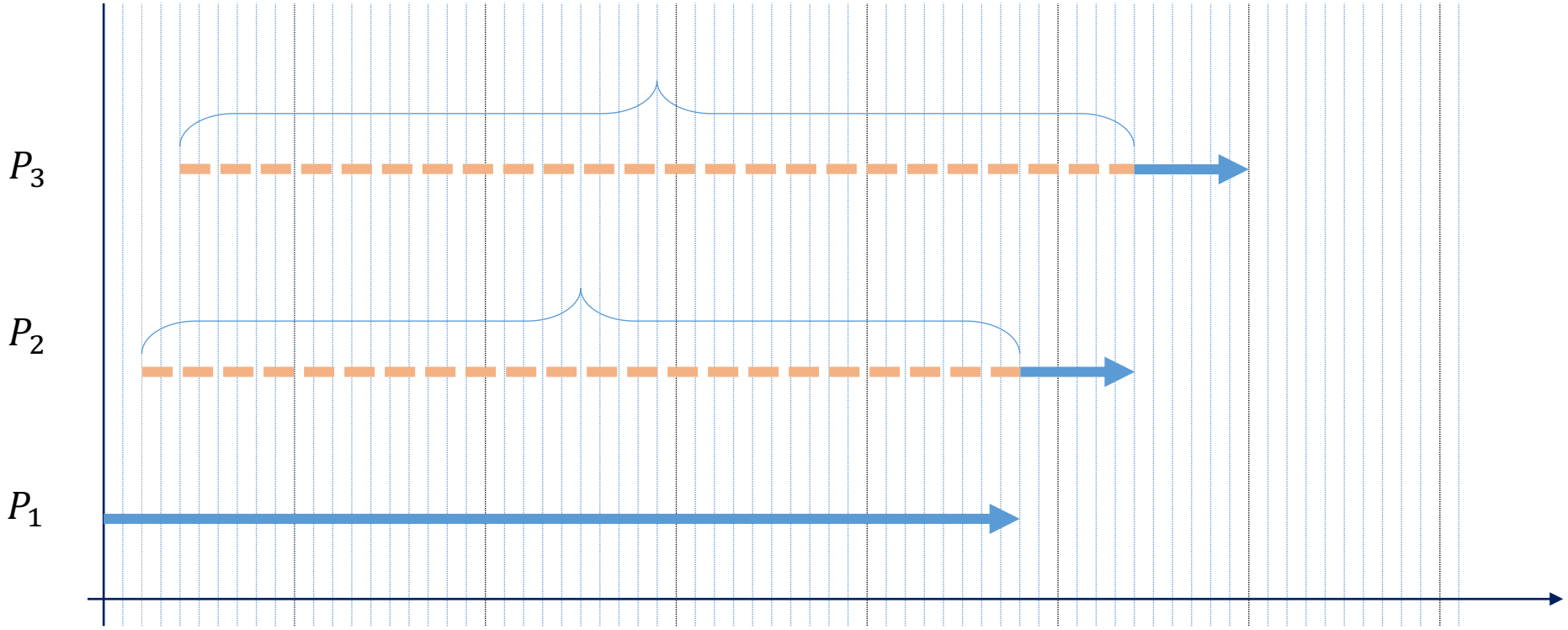
Tiến trình	Thời điểm vào RL	Thời gian xử lý
P_1	0	24
P_2	1	3
P_3	2	3

$$wait_{P_1} = 0$$

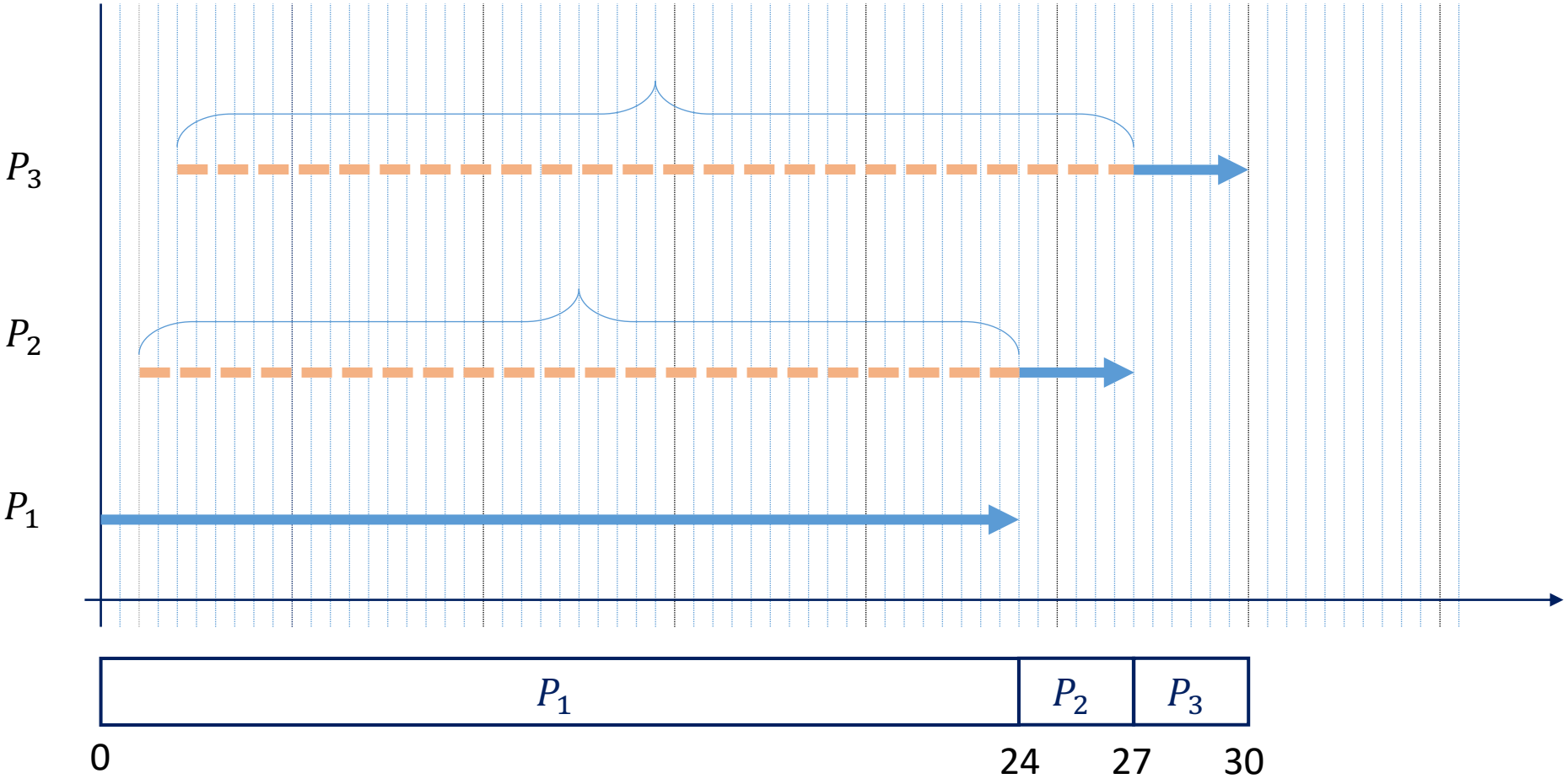
$$wait_{P_2} = 24 - 1 = 23$$

$$wait_{P_3} = (24 + 3) - 2 = 25$$

$$wait_{avg} = (0 + 23 + 25)/3 = 16$$



3. Điều phối tiến trình - FCFS



3. Điều phối tiến trình - FCFS

Tiến trình	Thời điểm vào RL (1)	Thời gian xử lý (2)	Completion Time (3)	Turn around time (4)=(3)-(1)	Thời gian chờ (5)=(4)-(2)
P_1	0	24	24	24-0=24	24-24=0
P_2	1	3	27	27-1=26	26-3=23
P_3	2	3	30	30-2=28	28-3=25

Turn Around Time: thời gian kể từ lúc đưa vào (submission) đến lúc kết thúc

Thời gian chờ trung bình là $\frac{0+23+25}{3} = 16$ miliseconds.

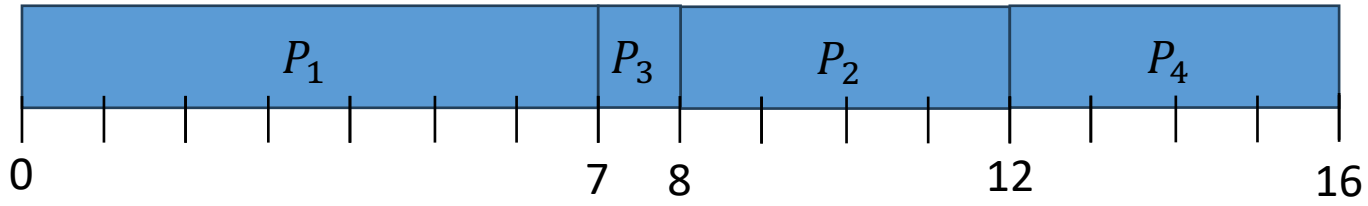
3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **Shortest Job First (SJF)**

Process	Thời điểm đến	Burst time (ms)
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Giản đồ Gantt khi định thời theo SJF



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

3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **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 trong việc giảm thời gian đợi trung bình.
  - Nhược điểm: Cần phải **ước lượng thời gian** cần CPU tiếp theo của process

3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **Shortest Job First (SJF)**
  - t: thời gian xử lý mà tiến trình còn yêu cầu
  - Độ ưu tiên p = 1/t

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P_1	0	6
P_2	1	8
P_3	2	4
P_4	3	2

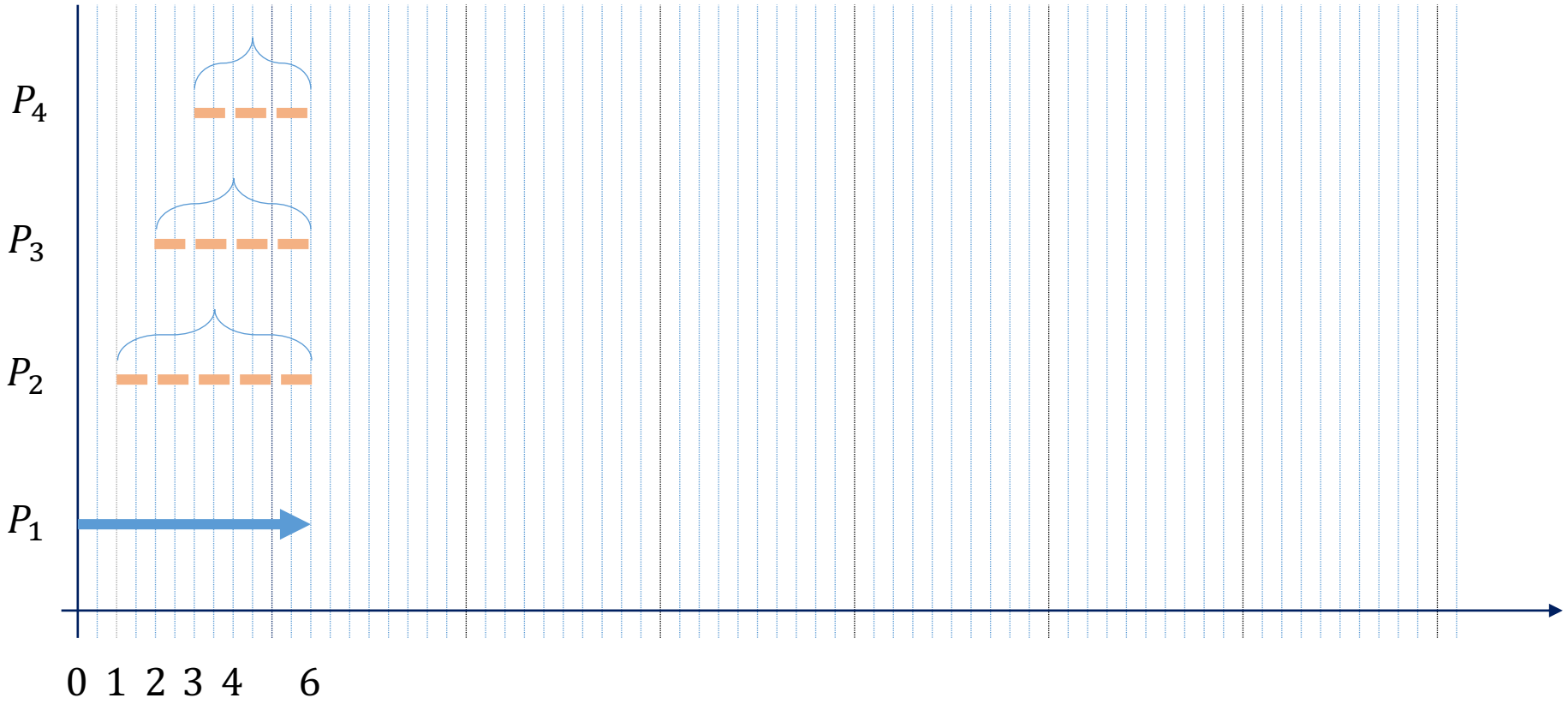
Thuật giải SJF độc quyền

Thuật giải SJF không độc quyền

Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6	1/6
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

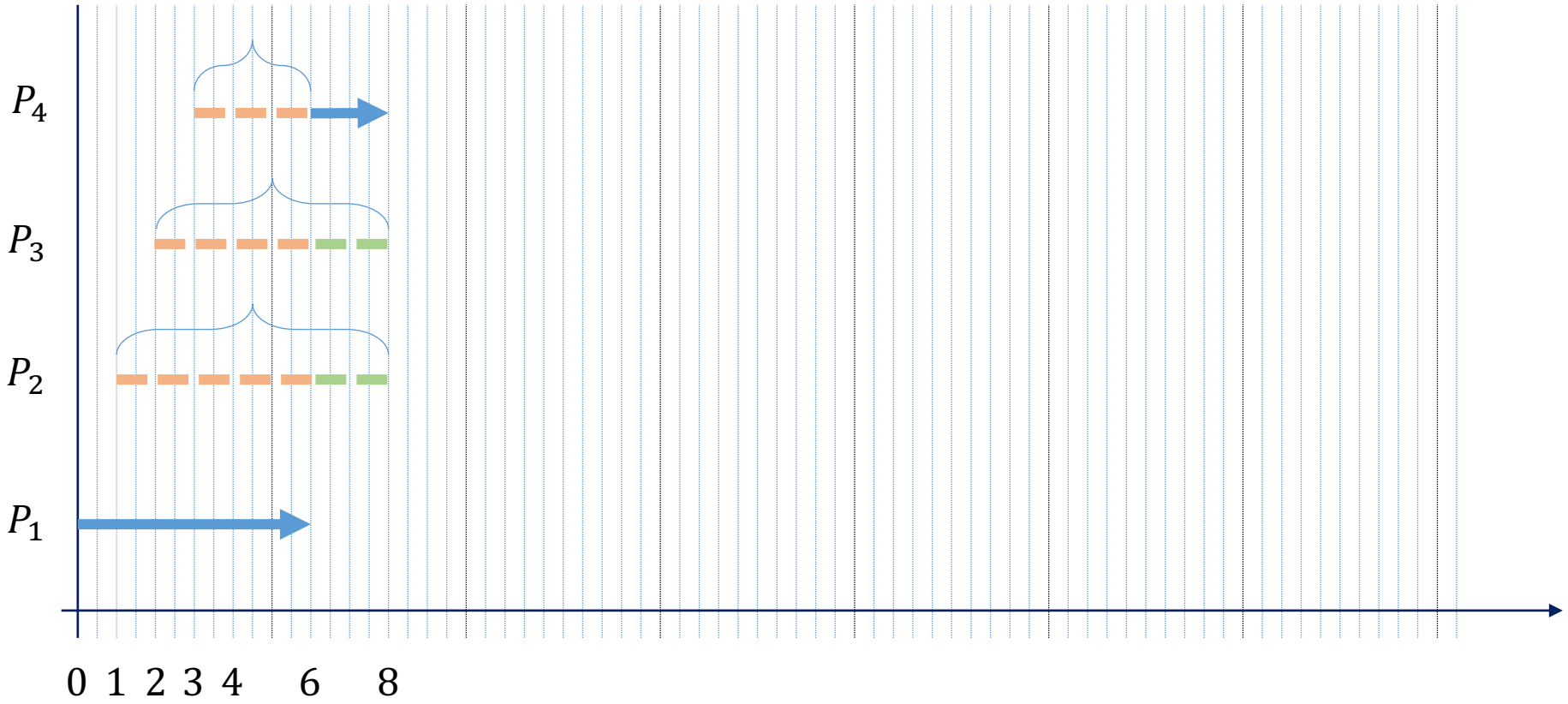
Thuật giải SJF **độc quyền**



Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6	1/6
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

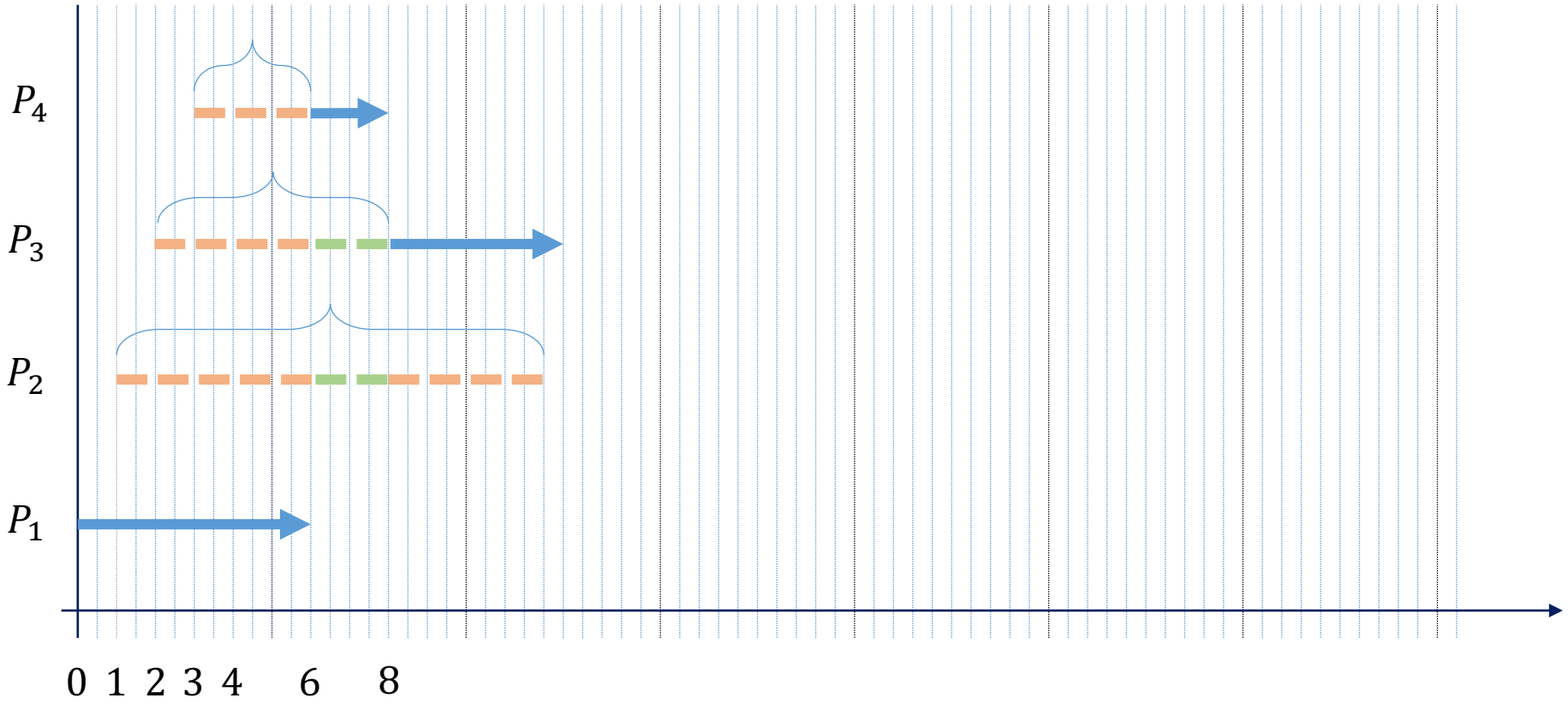
Thuật giải SJF **độc quyền**



Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6	1/6
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

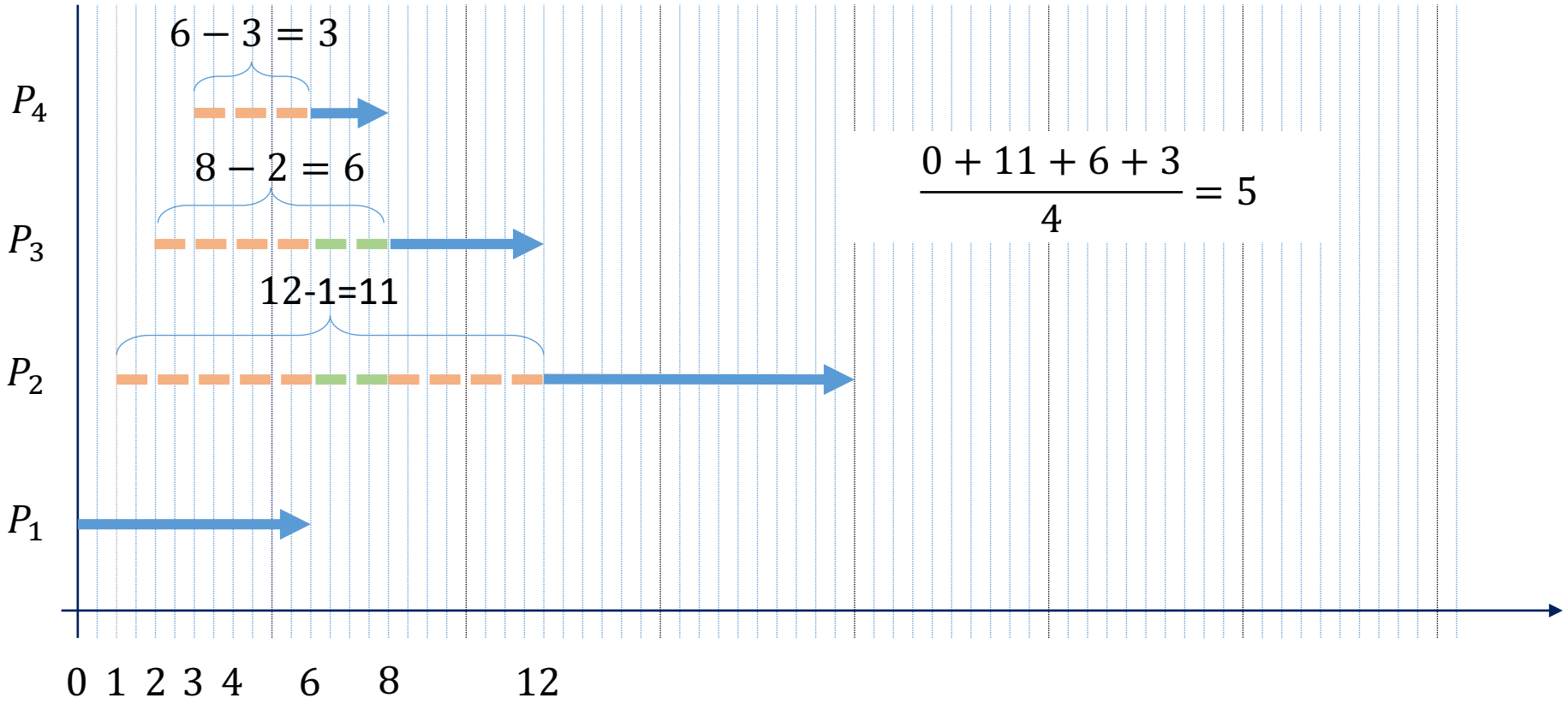
Thuật giải SJF **độc quyền**



Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

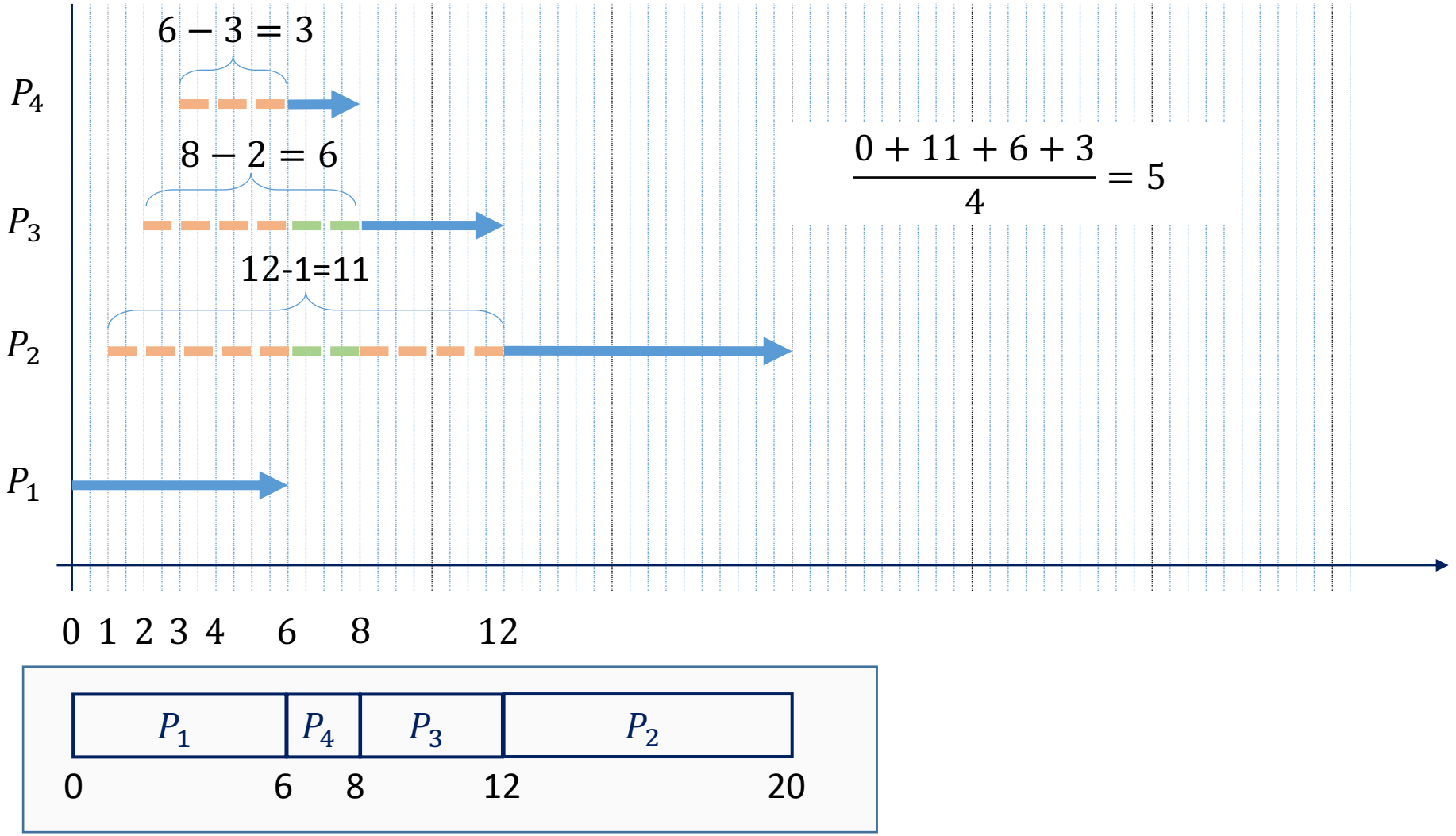
Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6	1/6
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

Thuật giải SJF **độc quyền**



Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

Thuật giải SJF **độc quyền**



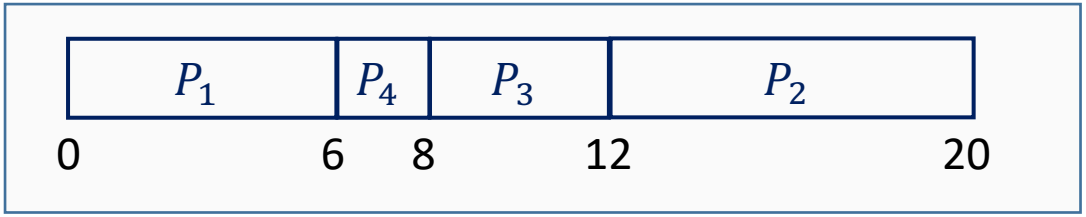
Thuật toán công việc ngắn nhất (Shortest-job-first SJF)

Thuật giải SJF **độc quyền**

Tiến trình	Thời điểm vào RL (1)	Thời gian xử lý (2)	Complete time (3)	Turn around time (4)=(3)-(1)	Thời gian chờ (5)=(4)-(2)
P_1	0	6	6	6-0=6	6-6=0
P_2	1	8	20	20-1=19	19-8=11
P_3	2	4	12	12-2=10	10-4=6
P_4	3	2	8	8-3=5	5-2=3

Turn Around Time: thời gian kể từ lúc đưa vào (submission) đến lúc kết thúc

Thời gian chờ trung bình là $\frac{0+11+6+3}{4} = \frac{20}{4} = 5$ miliseconds.



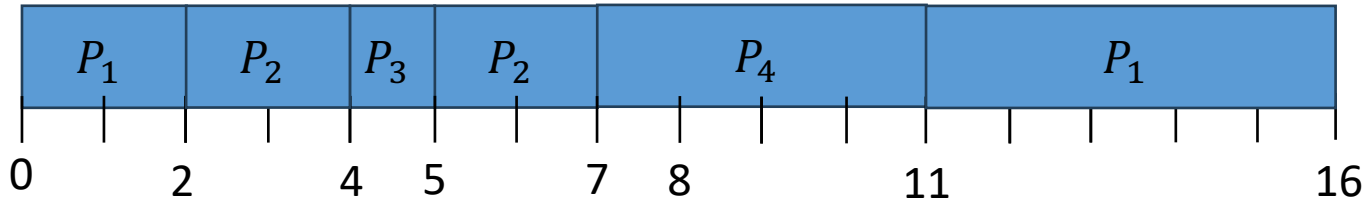
3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **Shortest Remaining Time First (SRTF)**

Process	Thời điểm đến	Burst time (ms)
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

- Giản đồ Gantt khi định thời theo SJF



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

3. Điều phối tiến trình

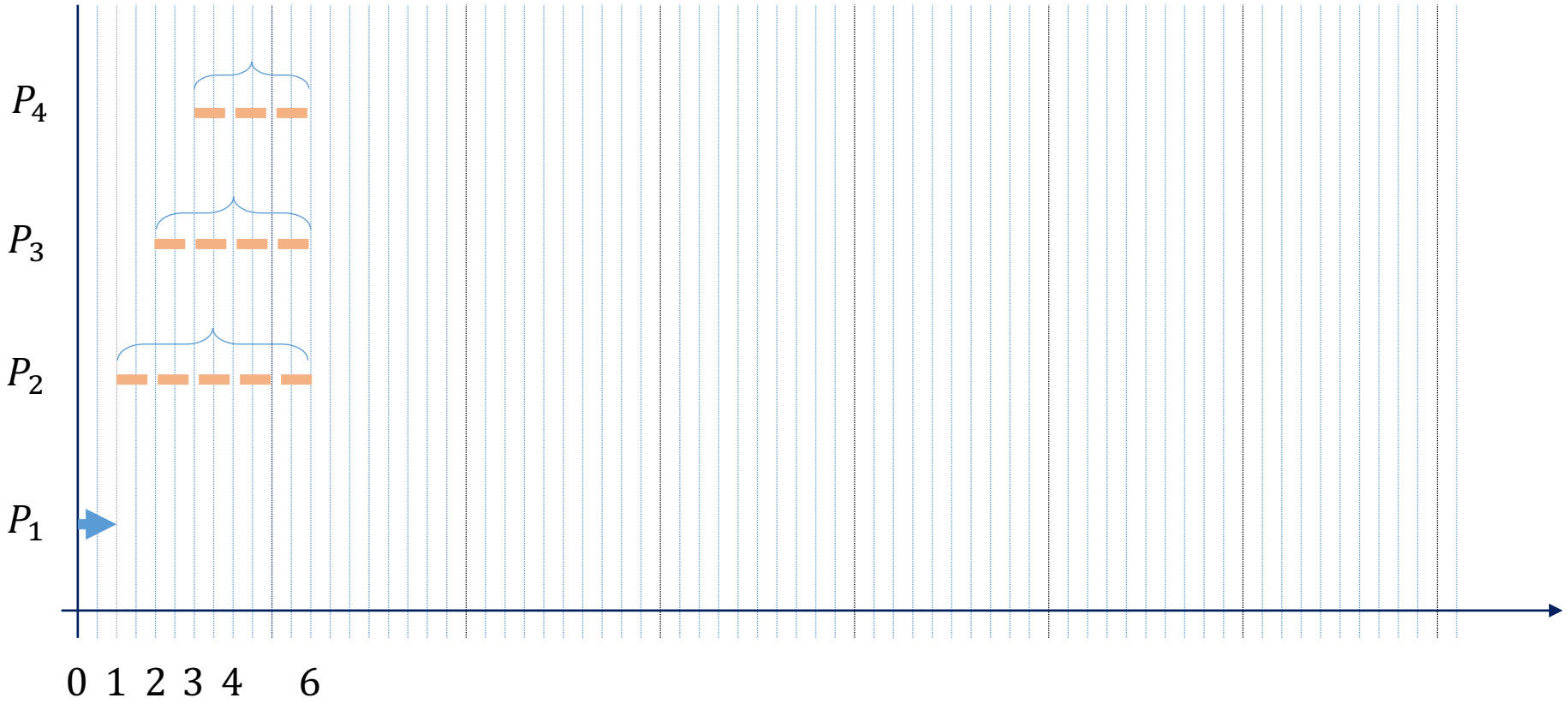
Các giải thuật điều phối tiến trình

- **Shortest Remaining Time First (SRTF)**
  - **Tránh** trường hợp **độc chiếm CPU** của các process có thời gian thực thi dài
  - 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**.

3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6	1/6
P_2	1	8	
P_3	2	4	
P_4	3	2	

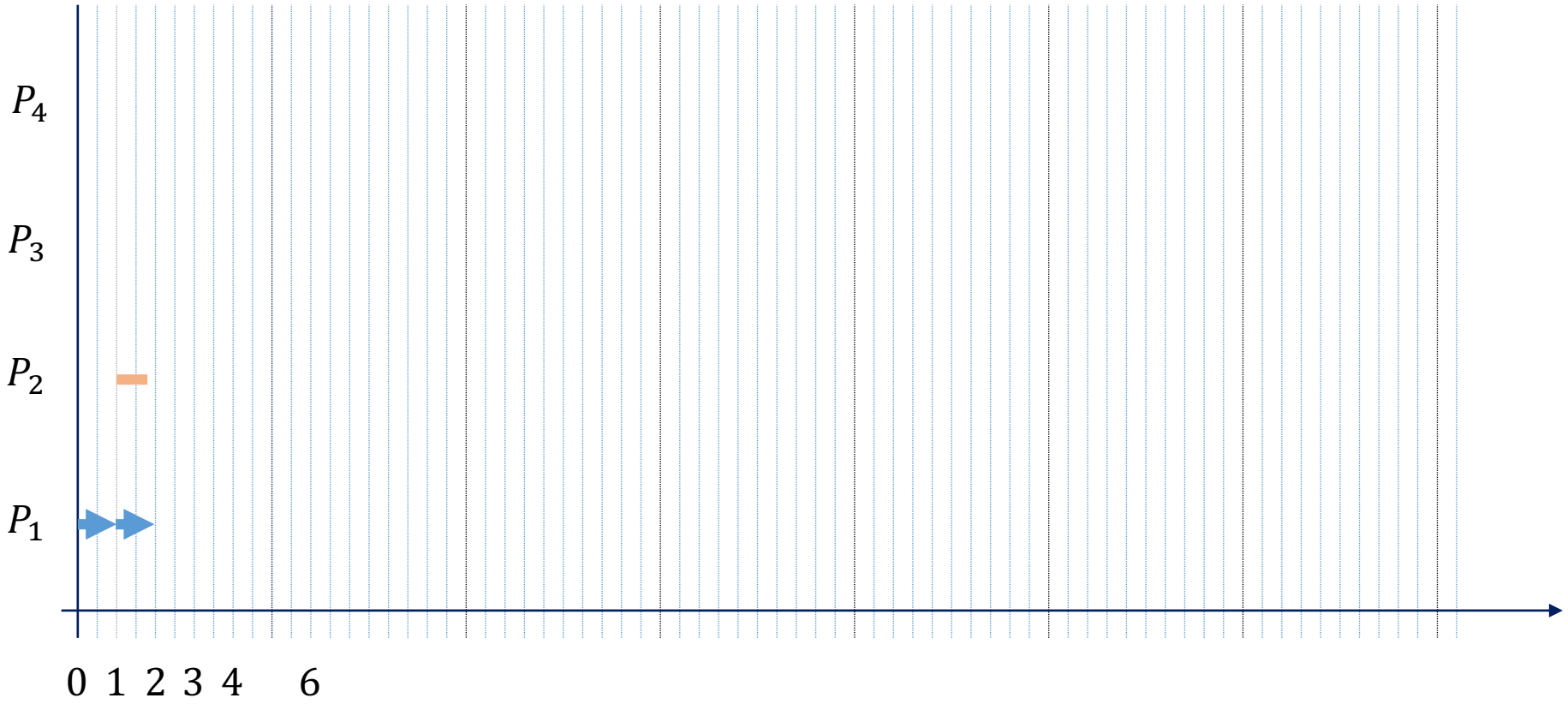
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	6⟶5	1/5
P_2	1	8	1/8
P_3	2	4	
P_4	3	2	

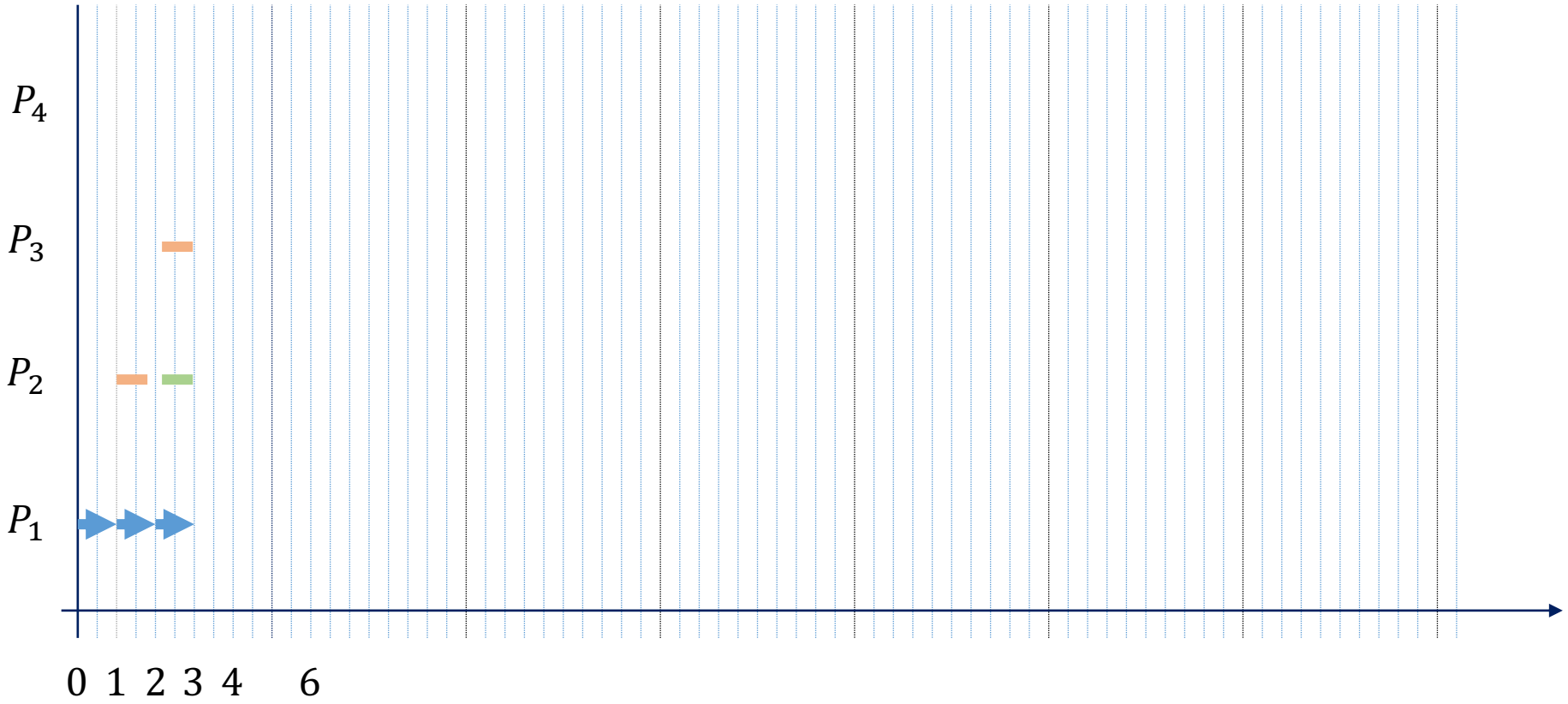
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	5⟶ 4	1/4
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	

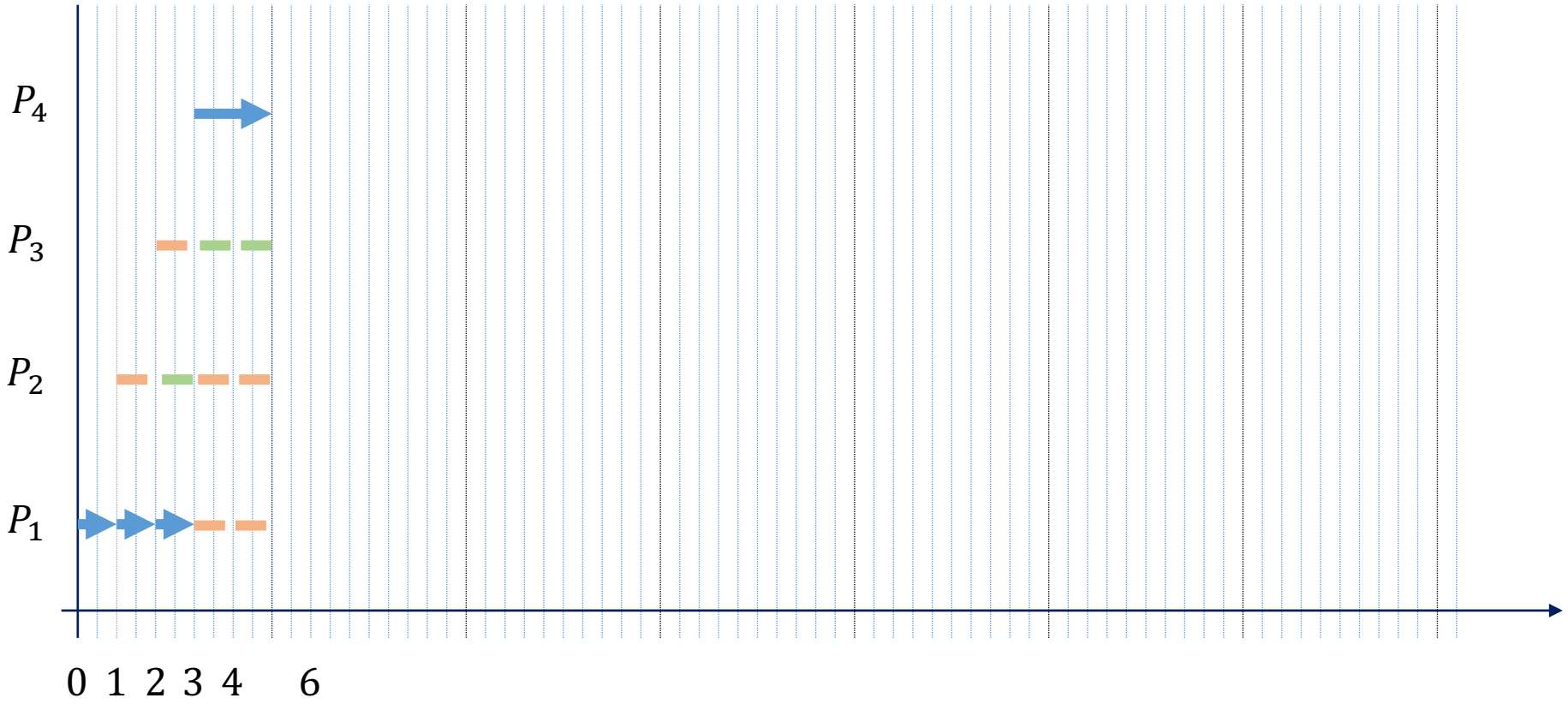
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	4⟶ 3	1/3
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

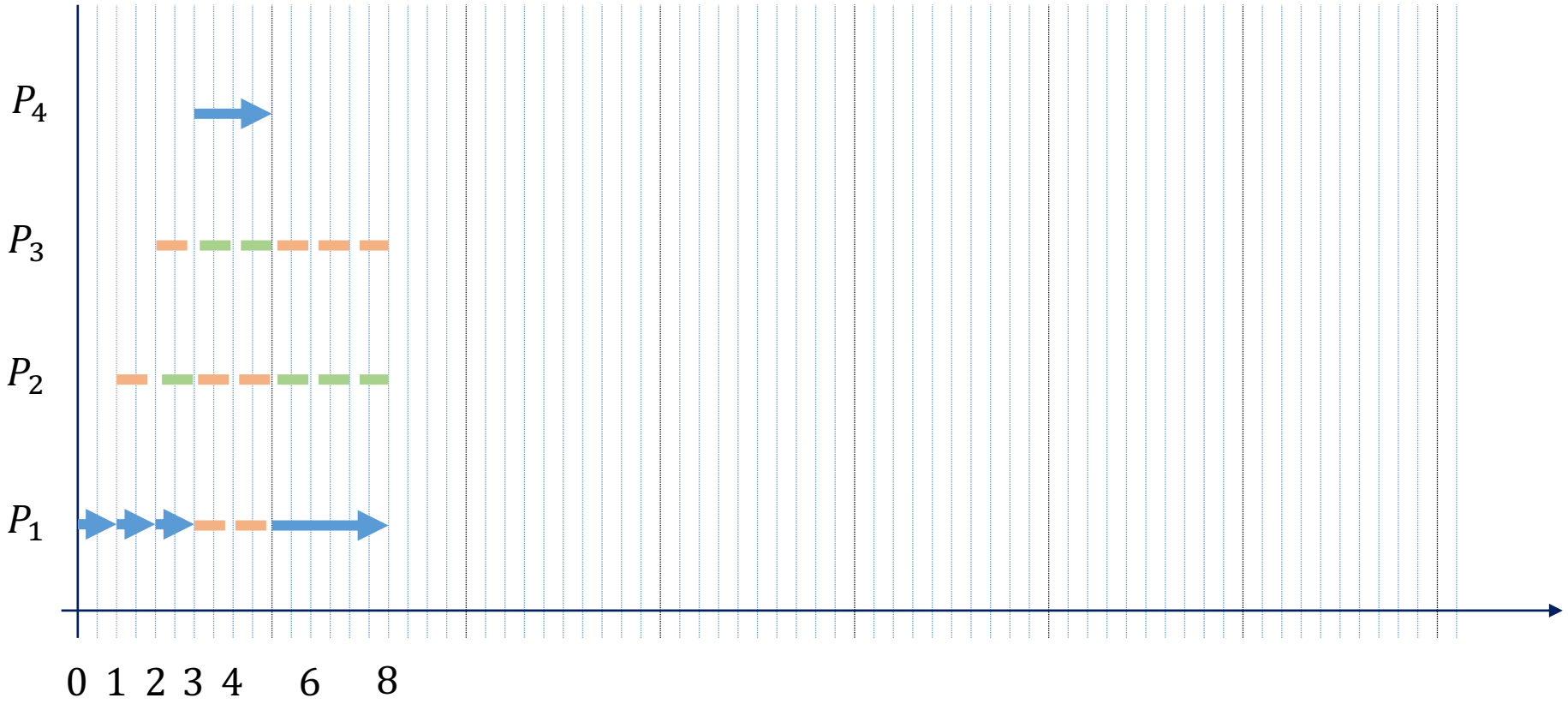
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	4⟶ 3	1/3
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

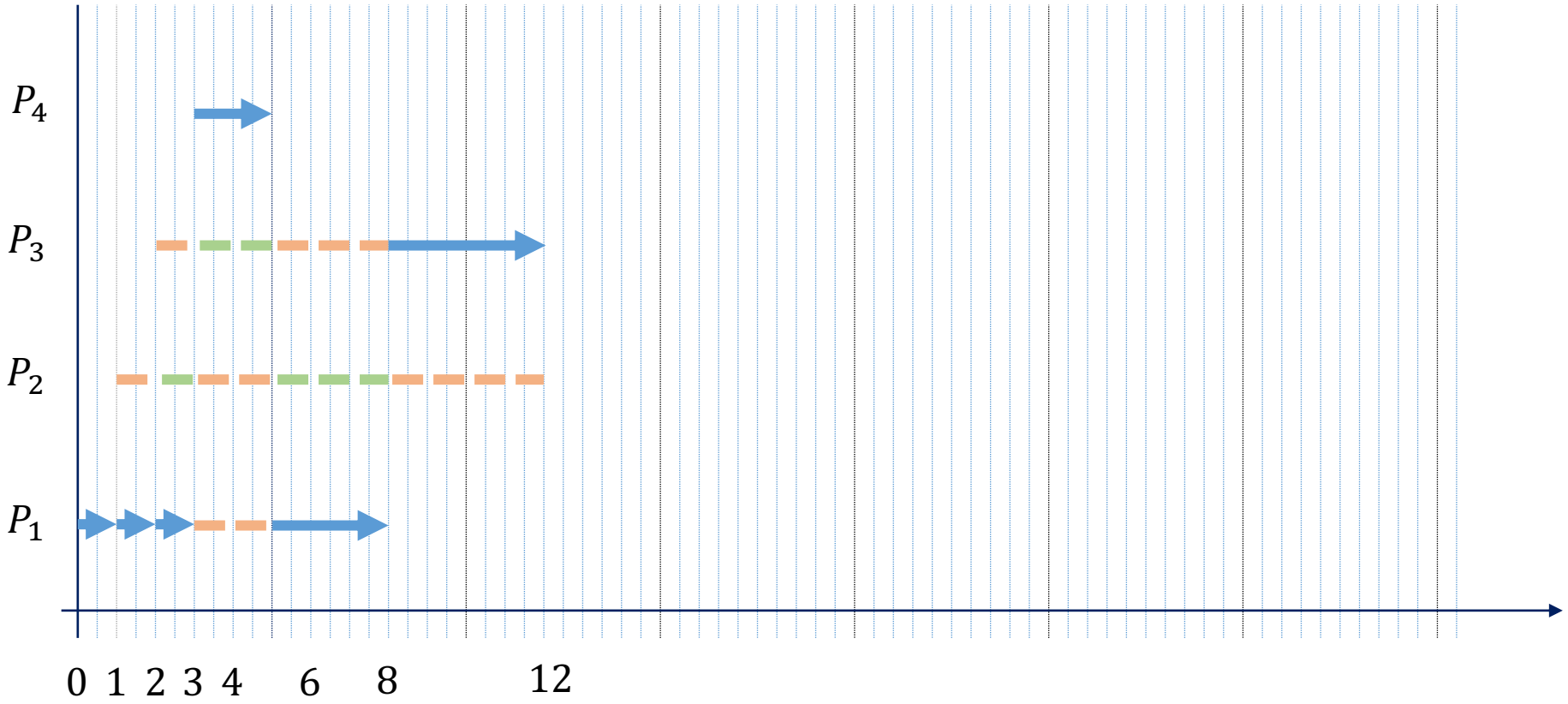
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	4⟶ 3	1/3
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

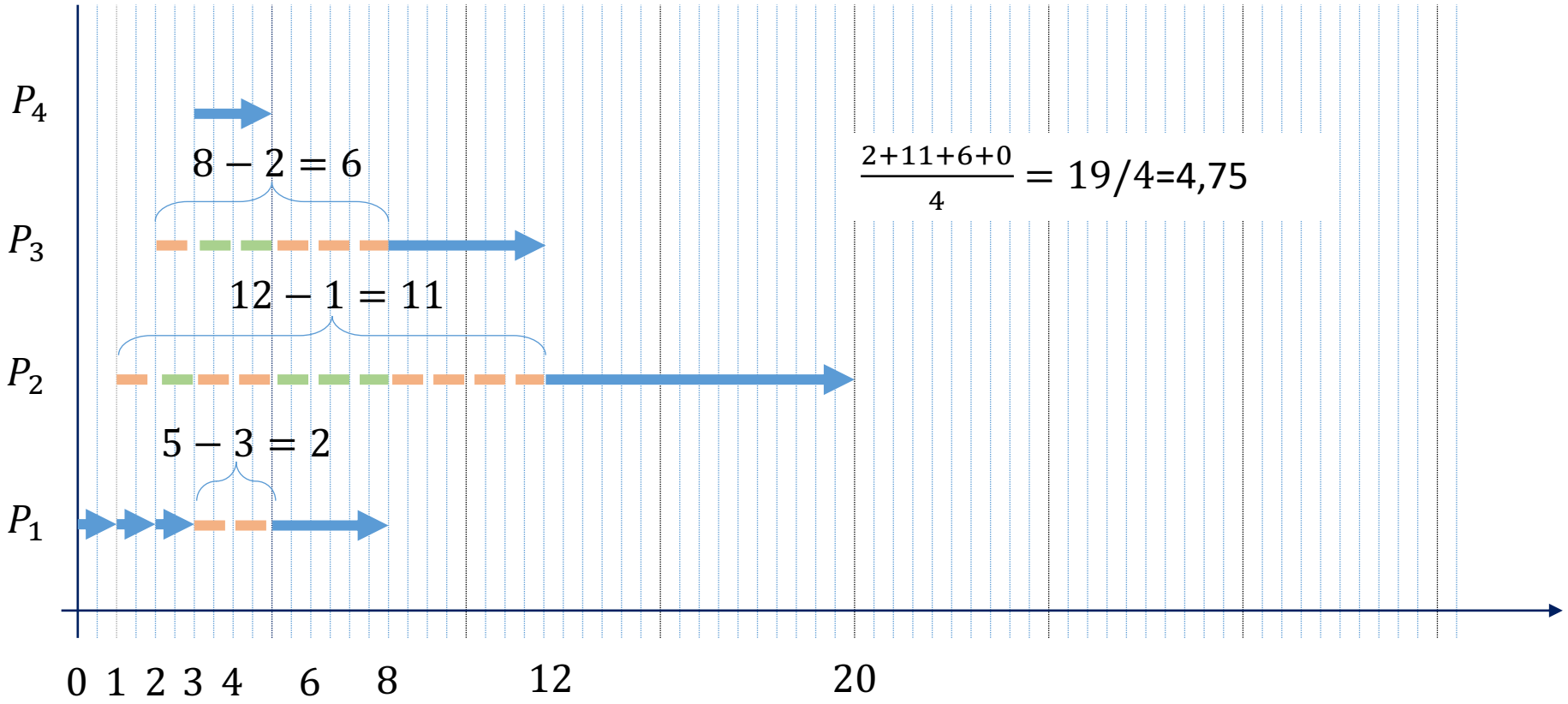
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

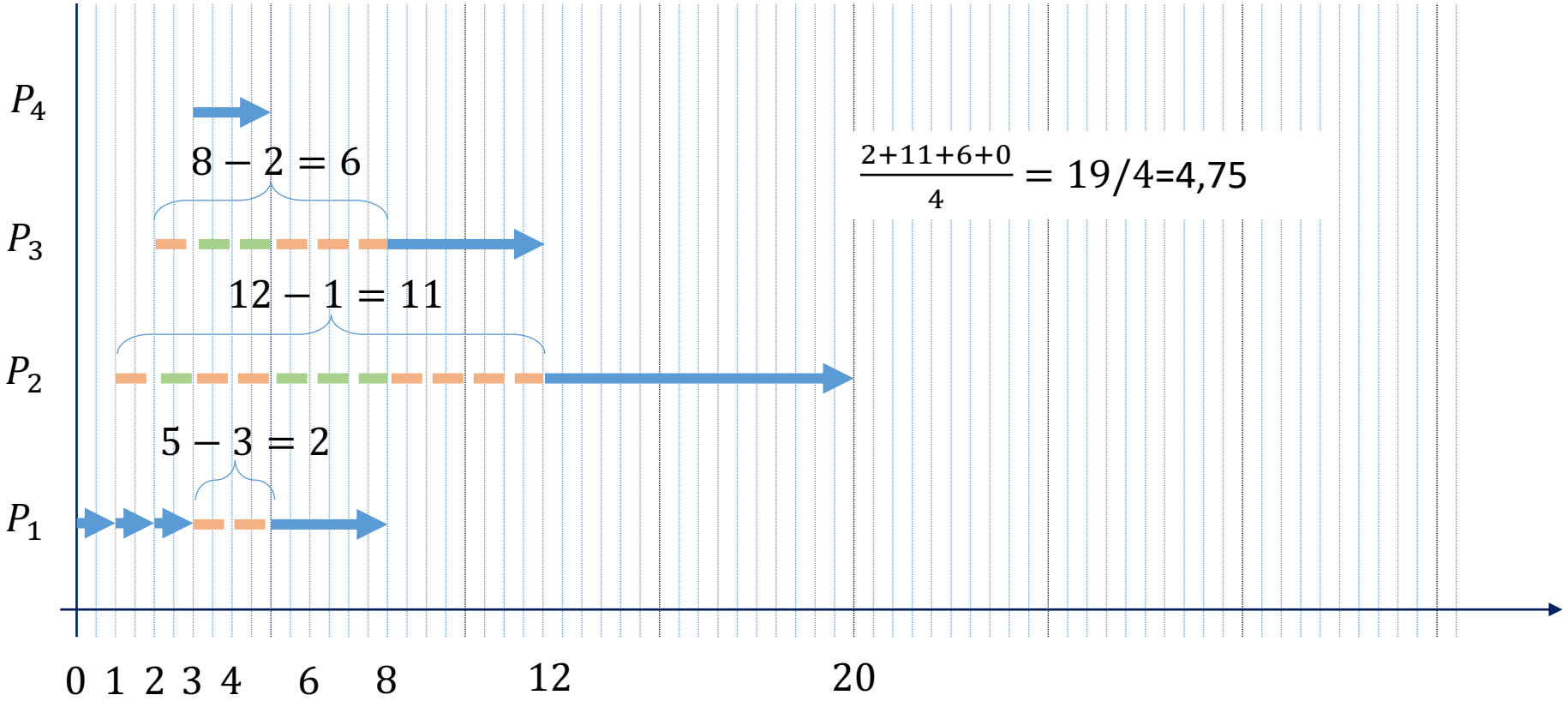
Tiến trình	Thời điểm vào RL	Thời gian xử lý	Độ ưu tiên
P_1	0	4⟶ 3	1/3
P_2	1	8	1/8
P_3	2	4	1/4
P_4	3	2	1/2

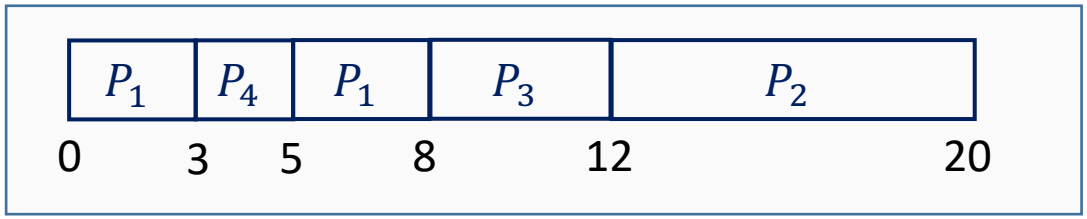
Thuật giải SJF không độc quyền



3. Điều phối tiến trình

Thuật giải SJF không độc quyền





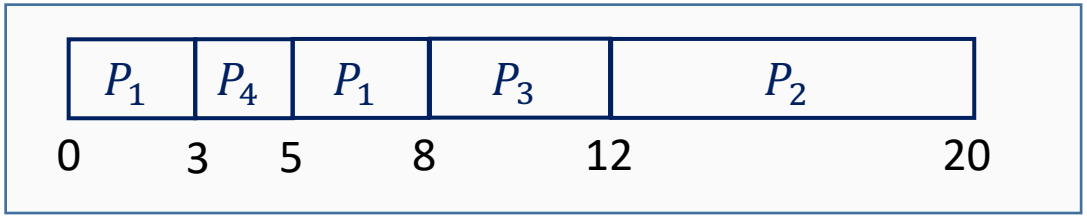
3. Điều phối tiến trình

Thuật giải SJF không độc quyền

Tiến trình	Thời điểm vào RL	Thời gian xử lý	Complete time (3)	Turn around time (4)=(3)-(1)	Thời gian chờ (5)=(4)-(2)
P_1	0	6	8	8-0=8	8-6=2
P_2	1	8	20	20-1=19	19-8=11
P_3	2	4	12	12-2=10	10-4=6
P_4	3	2	5	5-3=2	2-2=0

Turn Around Time: thời gian kể từ lúc đưa vào (submission) đến lúc kết thúc

Thời gian chờ trung bình là $\frac{2+11+6+0}{4} = \frac{19}{4} = 4{,}75$ miliseconds.



3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **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**
    - **Nonpreemptive**

3. Điều phối tiến trình

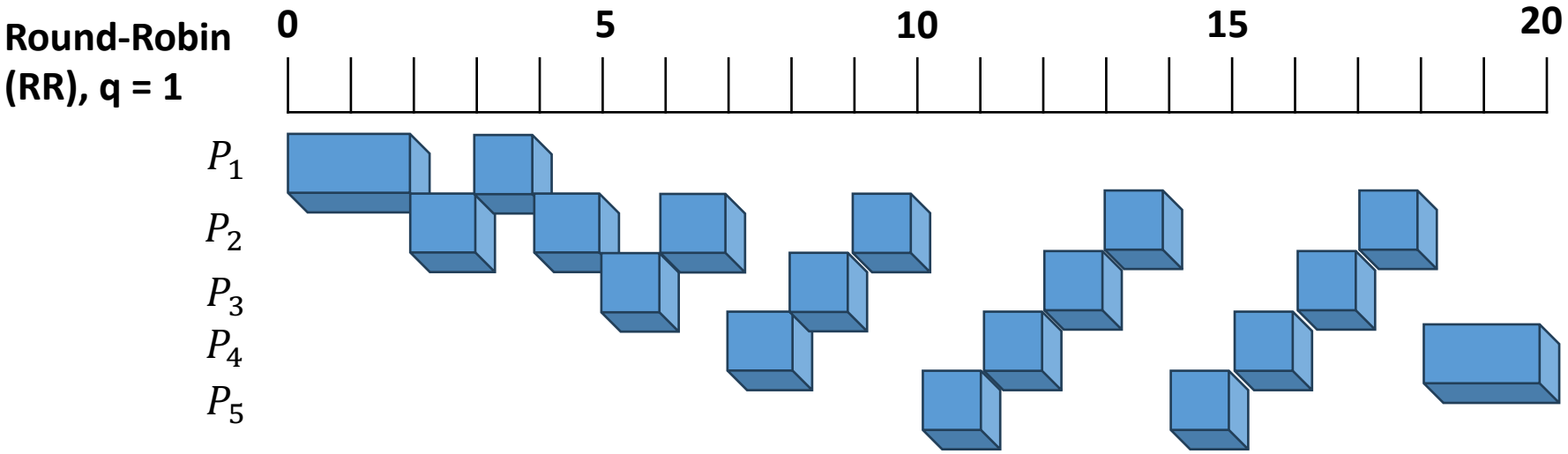
Các giải thuật điều phối tiến trình

- **Priority Scheduling**
  - 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 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.
  - 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.

3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

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



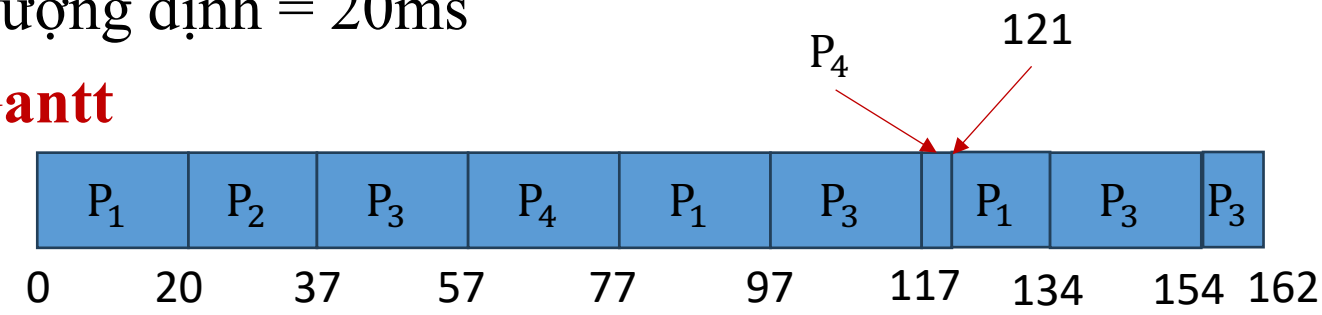
3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

- **Round Robin (RR)**

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

- Thời gian lượng định = 20ms
- **Giản đồ Gantt**



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

3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

❖ Round Robin (RR)

- Khi thực hiện chuyển process 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
- Hiệu năng tuỳ thuộc vào kích thước của thời gian lượng định (còn gọi là time slice). Hàm phụ thuộc này không đơn giản.
- Time slice ngắn thì đáp ứng nhanh
  - 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

3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

❖ Round Robin (RR)

- Quantum time và thời gian chuyển process:
- Nếu quantum time = 20 ms và thời gian chuyển process = 5 ms, thì phí tổn OS overhead chiếm 5/25 = 20%
- Nếu quantum = 500 ms, thì phí tổn chỉ còn khoảng 1%
  - Nếu có nhiều người sử dụng trên hệ thống và thuộc loại interactive thì sẽ thấy đáp ứng rất chậm
- Tuỳ tập công việc mà chọn quantum time phù hợp
- Time slice nên lớn trong tương quan so sánh với thời gian chuyển process

  Ví dụ: với 4.3 BSD UNIX, time slice là 1 giây.

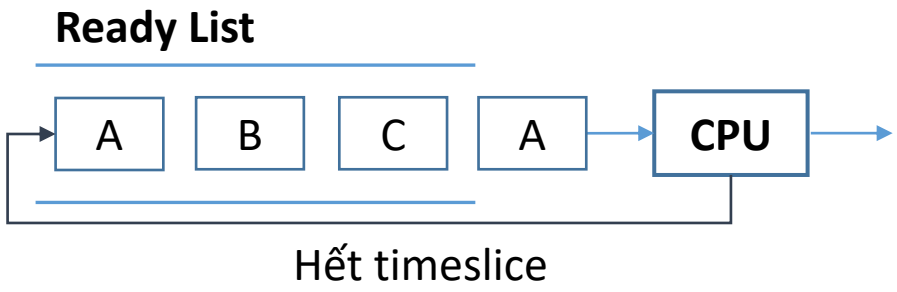
3. Điều phối tiến trình

Các giải thuật điều phối tiến trình

❖ Round Robin (RR)

- Nếu có ***n*** process trong hàng đợi ready, và quantum time là ***q***, thì mỗi process sẽ lấy ***1/n*** thời gian CPU theo từng khối có kích thước lớn nhất là.
  - 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
  - RR không phù hợp với hệ thống xác định độ ưu tiên khác nhau cho mỗi process
- Các process CPU-bound vẫn còn được "ưu tiên"

Thuật toán phân phối xoay vòng (Round Robin)



Tiến trình	Thời điểm vào RL	Thời gian xử lý
P_1	0	24
P_2	1	3
P_3	2	3

P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	
0	4	7	10	14	18	22	26	30

- Thời gian chờ đợi trung bình sẽ là $\frac{(6+3+5)}{3} = 4.66$ milisecondes.

Thuật toán phân phối xoay vòng (Round Robin)

Tiến trình	Thời điểm vào RL	Thời gian xử lý
P_1	0	24
P_2	1	3
P_3	2	3

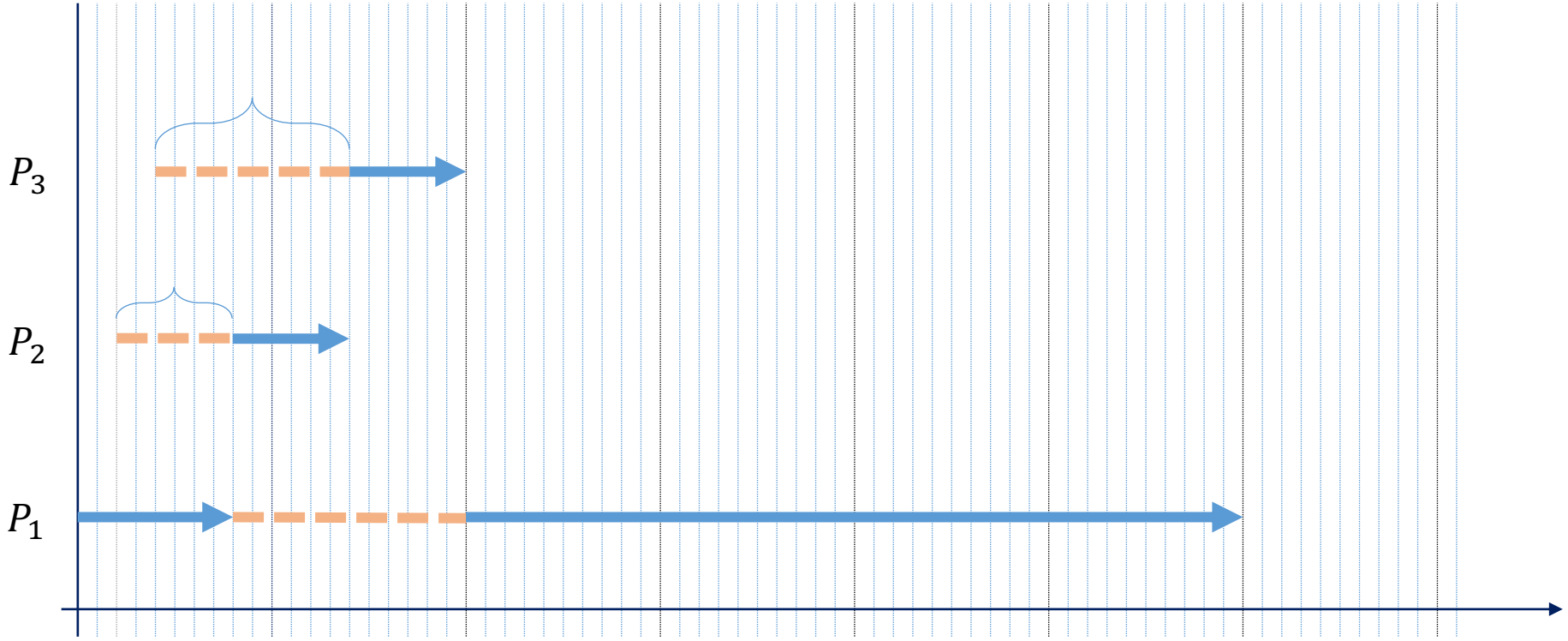
$$wait_{P_1} = 3 + 3 = 6$$

$$wait_{P_2} = 3$$

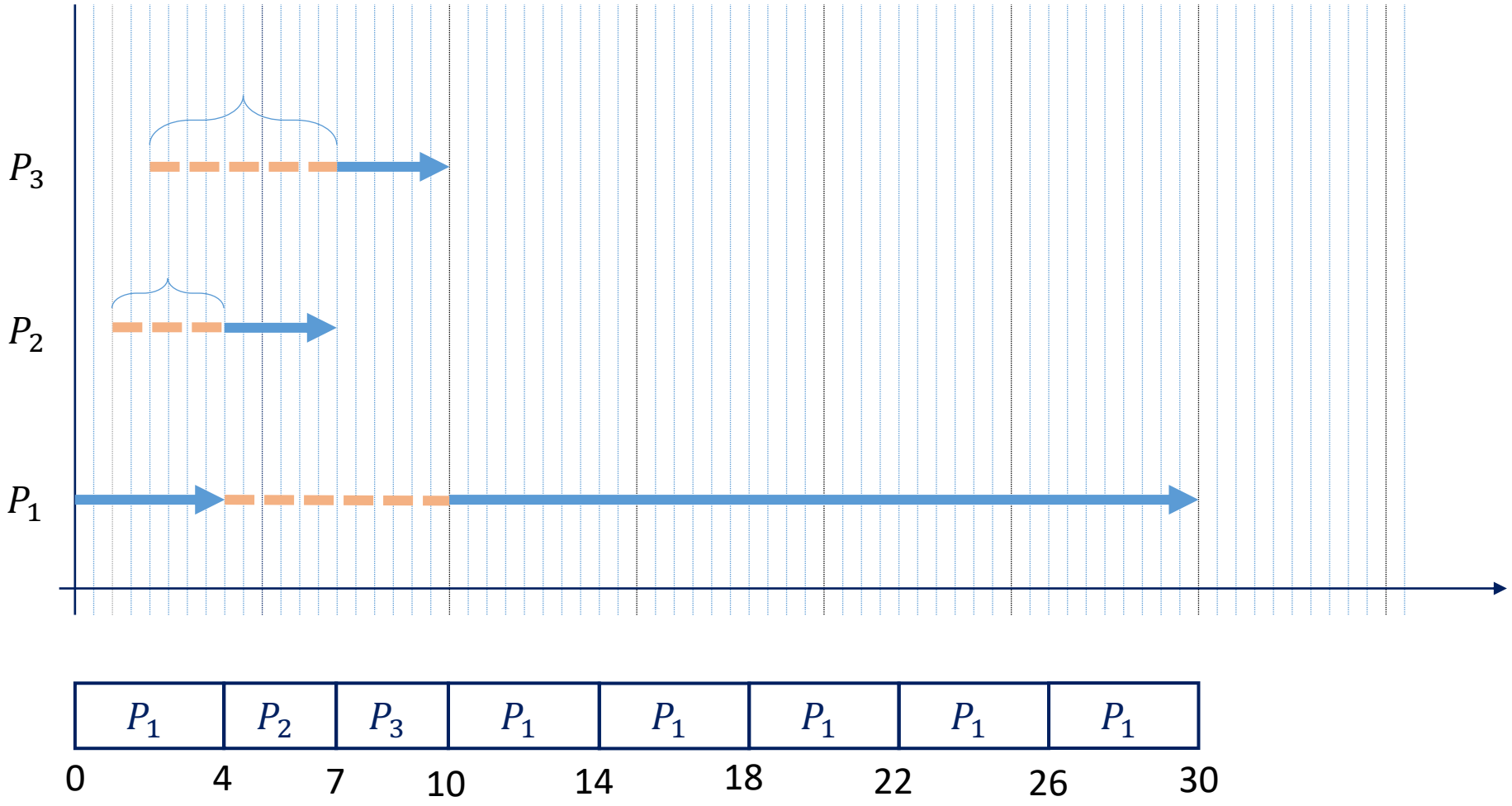
$$wait_{P_3} = 2 + 3 = 5$$

$$wait_{avg} = (6 + 3 + 5)/3 = 14/3$$

quantum là 4 miliseconds



Thuật toán phân phối xoay vòng (Round Robin)

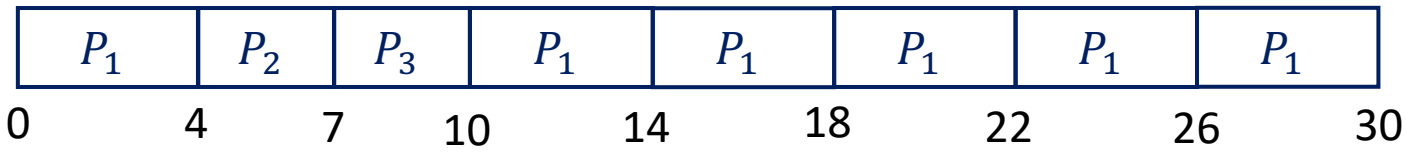


Thuật toán phân phối xoay vòng (Round Robin)

Tiến trình	Thời điểm vào RL (1)	Thời gian xử lý (2)	Complete time (3)	Turn around time (4)=(3)-(1)	Thời gian chờ (5)=(4)-(2)
P_1	0	24	30	30-0=30	30-24=6
P_2	1	3	7	7-1=6	6-3=3
P_3	2	3	10	10-2=8	8-3=5

Turn Around Time: thời gian kể từ lúc đưa vào (submission) đến lúc kết thúc

Thời gian chờ trung bình là $\frac{6+3+5}{3} = \frac{14}{3}$ miliseconds.



Thuật toán độ ưu tiên

- Độ ưu tiên $P_2 > P_3 > P_1$

Tiến trình	Thời điểm vào RL	Độ ưu tiên	Thời gian xử lý
P_1	0	3	24
P_2	1	1	3
P_3	2	2	3

Thuật giải độ ưu tiên độc quyền

P_1	P_2	P_3
0	24	27
		30

Thời gian chờ đợi trung bình sẽ là:
(0+23+25)/3=16 miliseconds

Thuật giải độ ưu tiên không độc quyền

P_1	P_2	P_3	P_1	
0	1	4	7	30

Thời gian chờ đợi trung bình sẽ là:
(6+0+2)/3=2.7 miliseconds

Thuật toán độ ưu tiên

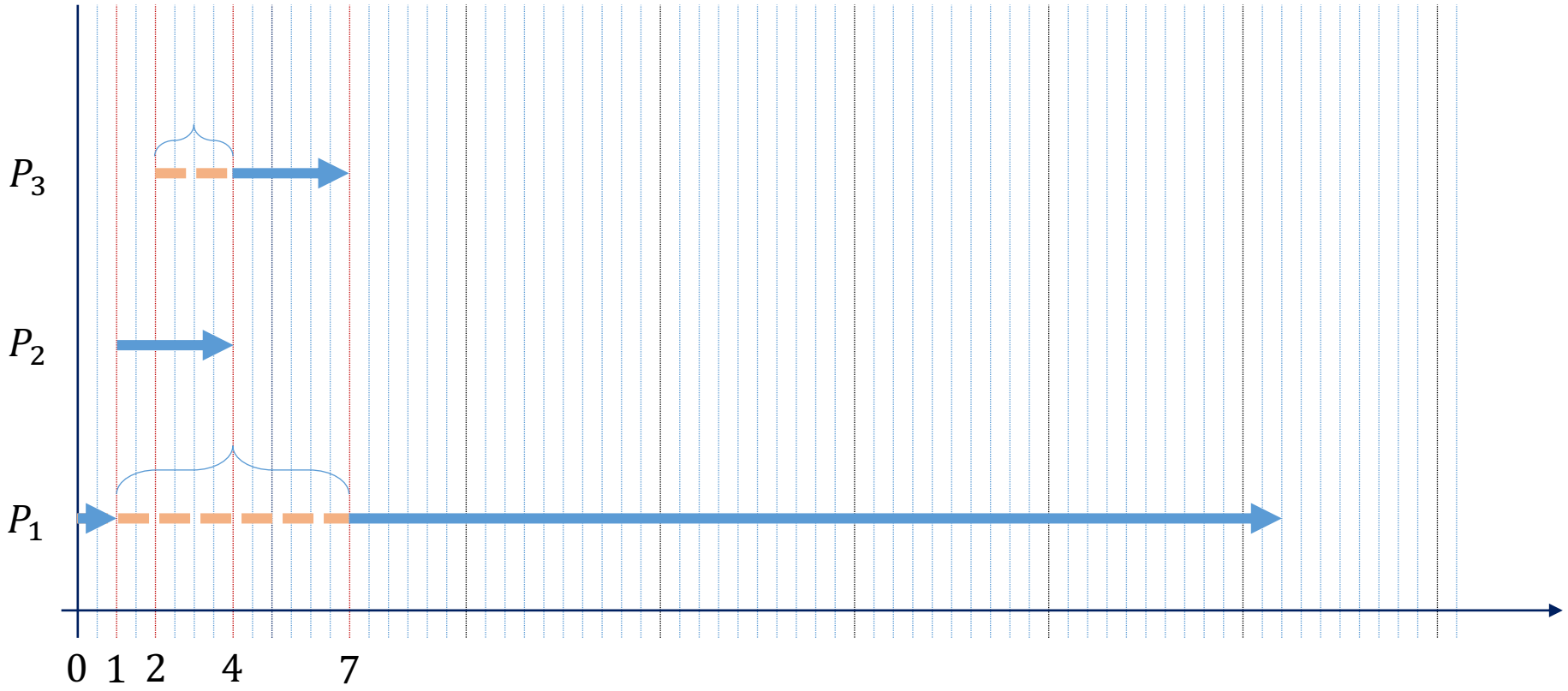
Tiến trình	Thời điểm vào RL	Độ ưu tiên	Thời gian xử lý
P_1	0	3	24
P_2	1	1	3
P_3	2	2	3

$$wait_{P_1} = 3 + 3 = 6$$

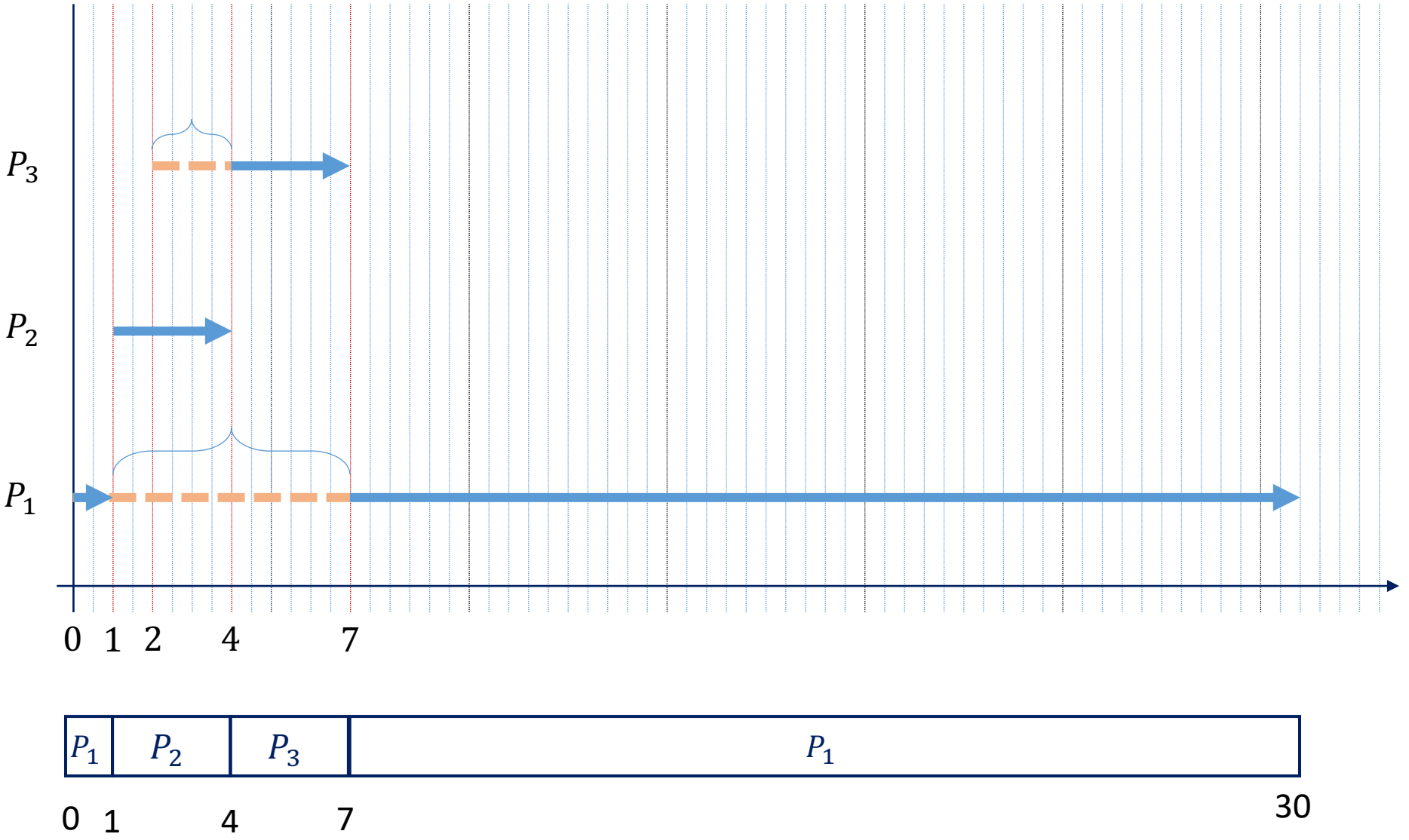
$$wait_{P_2} = 0$$

$$wait_{P_3} = 2$$

$$wait_{avg} = (6 + 0 + 2)/3 = 8/3$$



Thuật toán độ ưu tiên



Thuật toán độ ưu tiên

Tiến trình	Thời điểm vào RL (1)	Thời gian xử lý (2)	Complete time (3)	Turn around time (4)=(3)-(1)	Thời gian chờ (5)=(4)-(2)
P_1	0	24	30	30-0=30	30-24=6
P_2	1	3	4	4-1=3	3-3=0
P_3	2	3	7	7-2=5	5-3=2

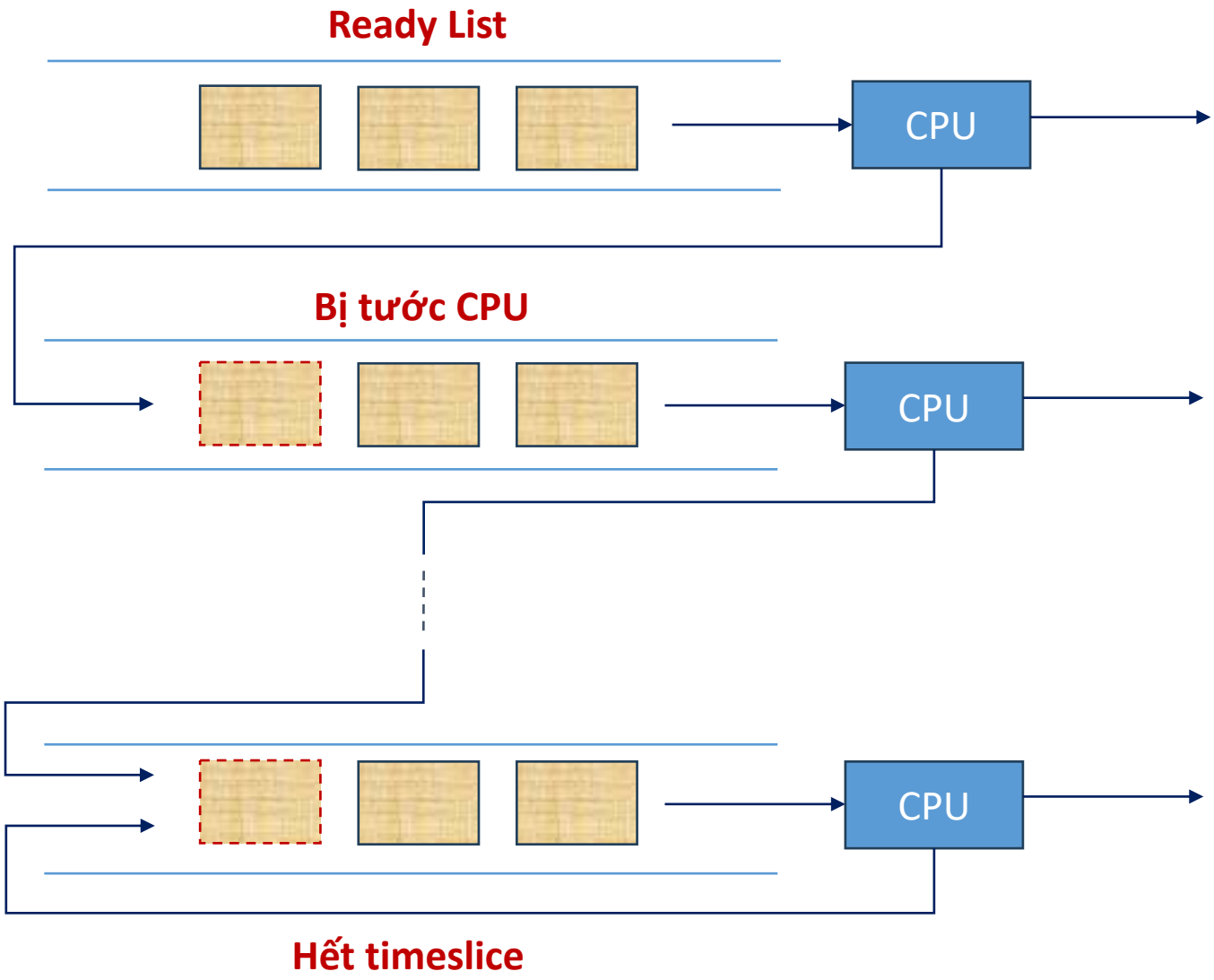
Turn Around Time: thời gian kể từ lúc đưa vào (submission) đến lúc kết thúc

Thời gian chờ trung bình là $\frac{6+0+2}{3} = \frac{8}{3}$ miliseconds.

Thuật toán nhiều mức độ ưu tiên

- Danh sách sẵn sàng được chia thành nhiều danh sách.
- Mỗi danh sách gồm các tiến trình có cùng độ ưu tiên và được áp dụng một giải thuật điều phối riêng

Điều phối theo nhiều mức ưu tiên xoay vòng (Multilevel Feedback)



3. Điều phối tiến trình

- **Chiến lược điều phối xổ số (Lottery)**
  - Mỗi tiến trình được cấp một "vé số".
  - HĐH chọn 1 vé "trúng giải", tiến trình nào sở hữu vé này sẽ được nhận CPU.
  - Là giải thuật độc quyền.
  - Đơn giản, chi phí thấp, bảo đảm tính công bằng cho các tiến trình.

4. Đồng bộ tiến trình

- **Liên lạc giữa các tiến trình**
  - **Mục đích:**
    - để chia sẻ thông tin như dùng chung file, bộ nhớ,…
    - hoặc hợp tác hoàn thành công việc
  - **Các cơ chế:**
    - Liên lạc bằng **tín hiệu** (Signal)
    - Liên lạc bằng **đường ống** (Pipe)
    - Liên lạc qua **vùng nhớ chia sẻ** (shared memory)
    - Liên lạc bằng **thông điệp** (Message)
    - Liên lạc qua **socket**

4. Đồng bộ tiến trình

Liên lạc bằng tín hiệu (Signal)

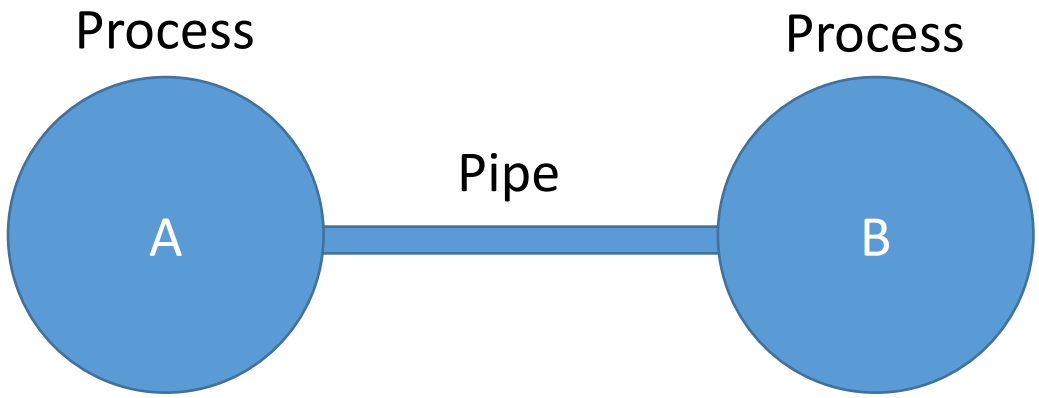
Tín hiệu	Mô tả
SIGINT	Người dùng nhấn phím Ctl-C để ngắt xử lý tiến trình
SIGILL	Tiến trình xử lý một chỉ thị bất hợp lệ
SIGKILL	Yêu cầu kết thúc một tiến trình
SIGFPT	Lỗi chia cho 0
SIGSEGV	Tiến trình truy xuất đến một địa chỉ bất hợp lệ
SIGCLD	Tiến trình con kết thúc

Tín hiệu được gởi đi bởi:
- Phần cứng
- Hệ điều hành:
- Tiến trình:
- Người sử dụng:

Khi tiến trình nhận tín hiệu:
- Gọi hàm xử lý tín hiệu.
- Xử lý theo cách riêng của tiến trình.
- Bỏ qua tín hiệu.

4. Đồng bộ tiến trình

Liên lạc bằng đường ống (Pipe)



- Dữ liệu truyền: dòng các byte (FIFO)
- Tiến trình đọc pipe sẽ bị khóa nếu pipe trống, và đợi đến khi pipe có dữ liệu mới được truy xuất.
- Tiến trình ghi pipe sẽ bị khóa nếu pipe đầy, và đợi đến khi pipe có chỗ trống để chứa dữ liệu.

Ví dụ

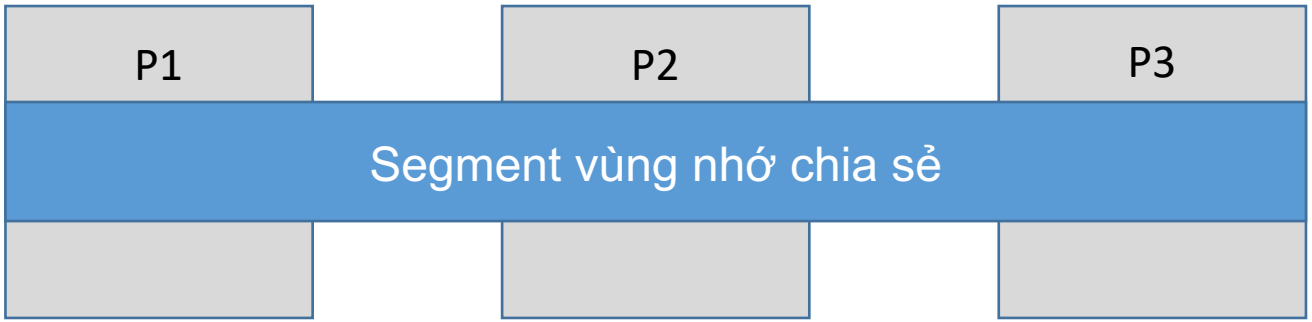
```
ctfptit@ctftime:~/codec$ ps aux > aux_demo.txt
ctfptit@ctftime:~/codec$ grep -i ssh aux_demo.txt
root        5946  0.0  0.0  12020  8192 ?        Ss   Sep05   0:00 sshd: /usr/sbin/sshd -D
[listener] 0 of 10-100 startups
root       63028  0.0  0.0  14964  8048 ?        Ss   15:46   0:00 sshd: ctfptit [priv]
ctfptit    63109  0.0  0.0  15124  6960 ?        S    15:47   0:00 sshd: ctfptit@pts/0
```

```
ctfptit@ctftime:~/codec$ ps aux | grep ssh
root        5946  0.0  0.0  12020  8192 ?        Ss   Sep05   0:00 sshd: /usr/sbin/sshd -D
[listener] 0 of 10-100 startups
root       63028  0.0  0.0  14964  8048 ?        Ss   15:46   0:00 sshd: ctfptit [priv]
ctfptit    63109  0.0  0.0  15124  6960 ?        S    15:47   0:00 sshd: ctfptit@pts/0
ctfptit    63490  0.0  0.0   7076  2048 pts/0    S+   17:26   0:00 grep --color=auto ssh
```

4. Đồng bộ tiến trình

❖Liên lạc qua vùng nhớ chia sẻ (shared memory)

- Vùng nhớ chia sẻ độc lập với các tiến trình
- Tiến trình phải gắn kết vùng nhớ chung vào không gian địa chỉ riêng của tiến trình

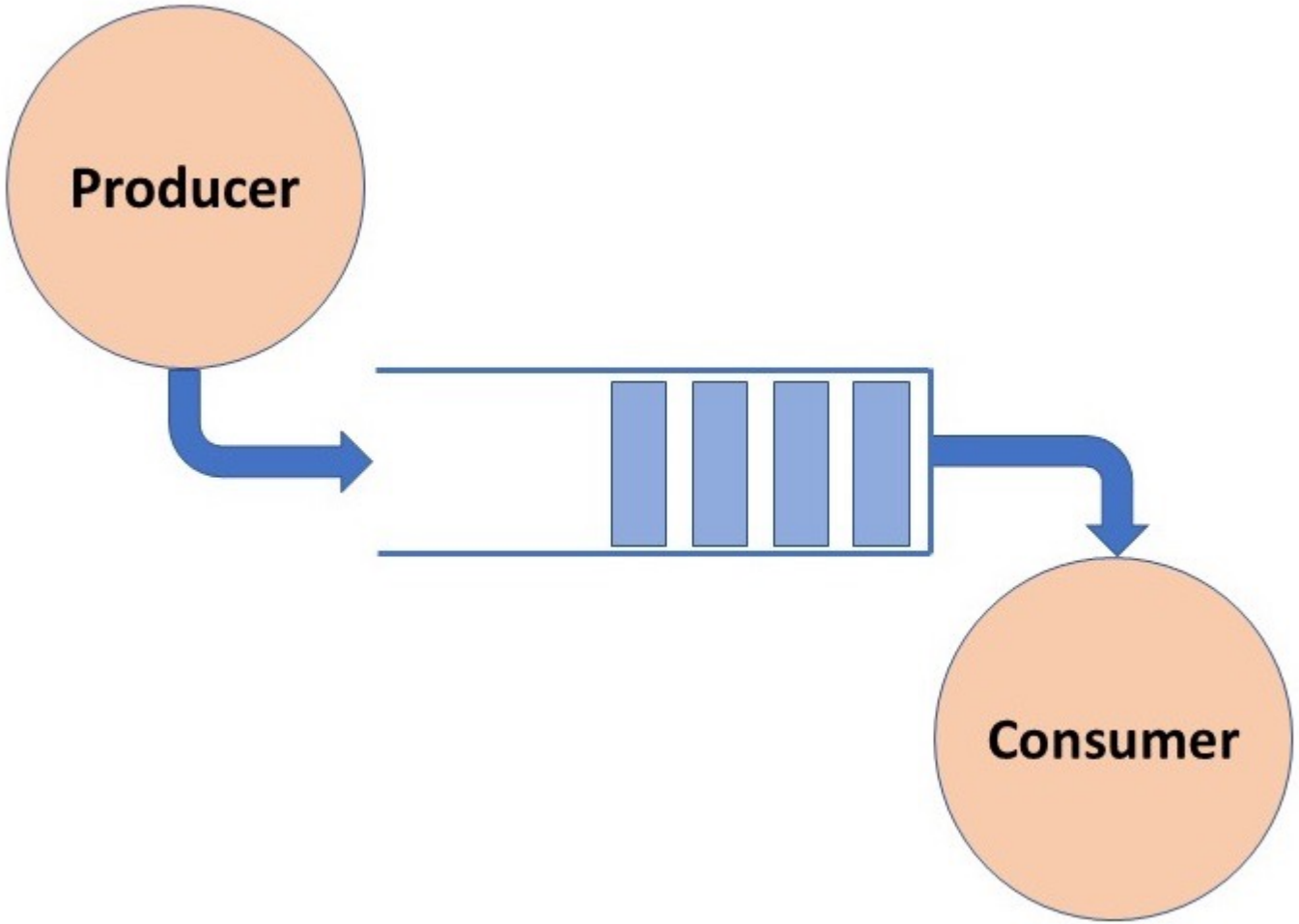


- Vùng nhớ chia sẻ là:
  - phương pháp nhanh nhất để trao đổi dữ liệu giữa các tiến trình.
  - cần được bảo vệ bằng những cơ chế đồng bộ hóa.
  - không thể áp dụng hiệu quả trong các hệ phân tán

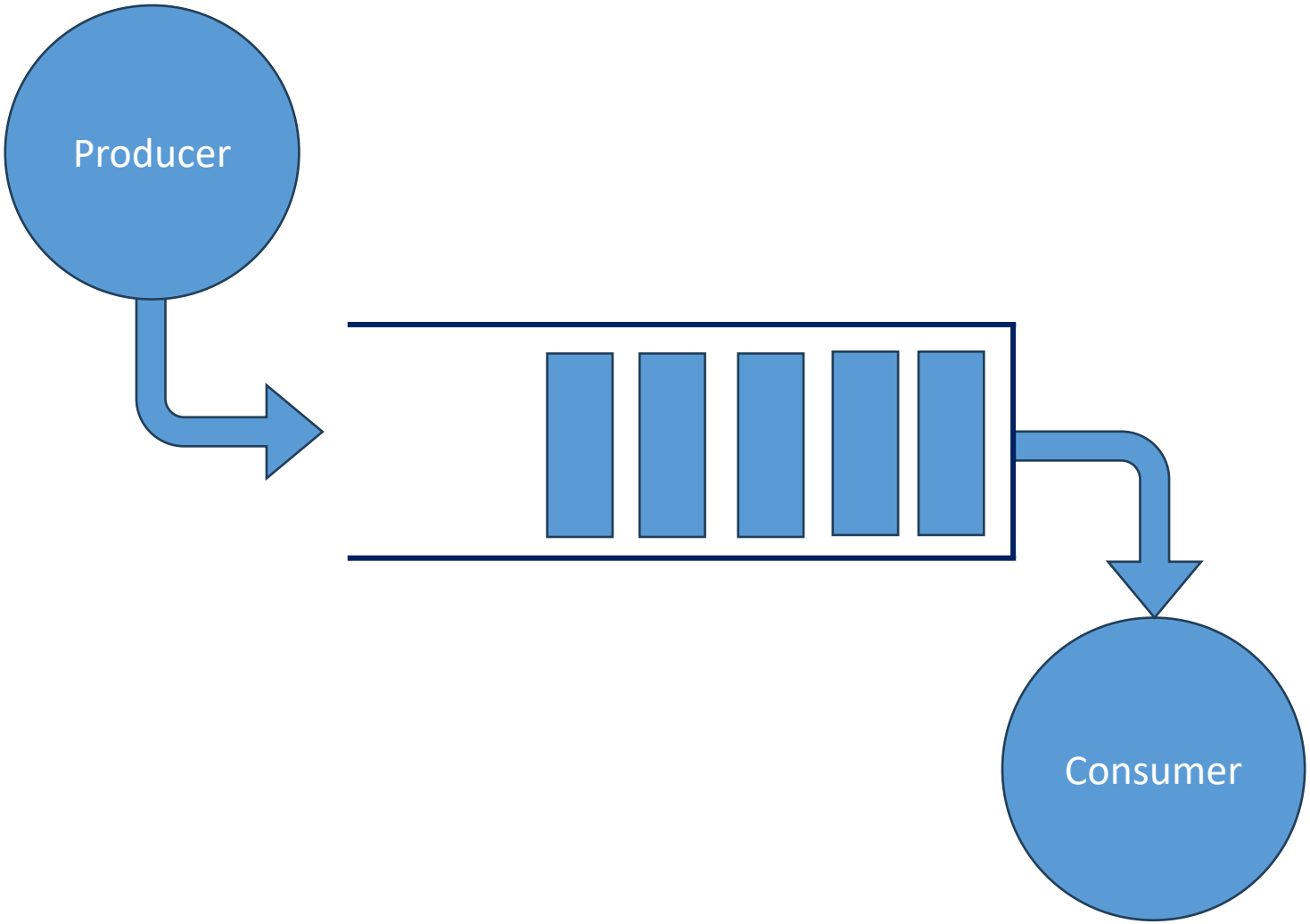
4. Đồng bộ tiến trình

- **Liên lạc bằng thông điệp (Message)**
  - Thiết lập một mối liên kết giữa hai tiến trình
  - sử dụng các hàm send, receive do hệ điều hành cung cấp để trao đổi thông điệp
  - **Cách liên lạc bằng thông điệp:**
    - **Liên lạc gián tiếp (indirect communication)**
      - Send(A, message): gởi một thông điệp tới port A
      - Receive(A, message): nhận một thông điệp từ port A
    - **Liên lạc trực tiếp (direct communication)**
      - Send(P, message): gởi một thông điệp đến tiến trình P
      - Receive(Q,message) : nhận một thông điệp từ tiến trình Q

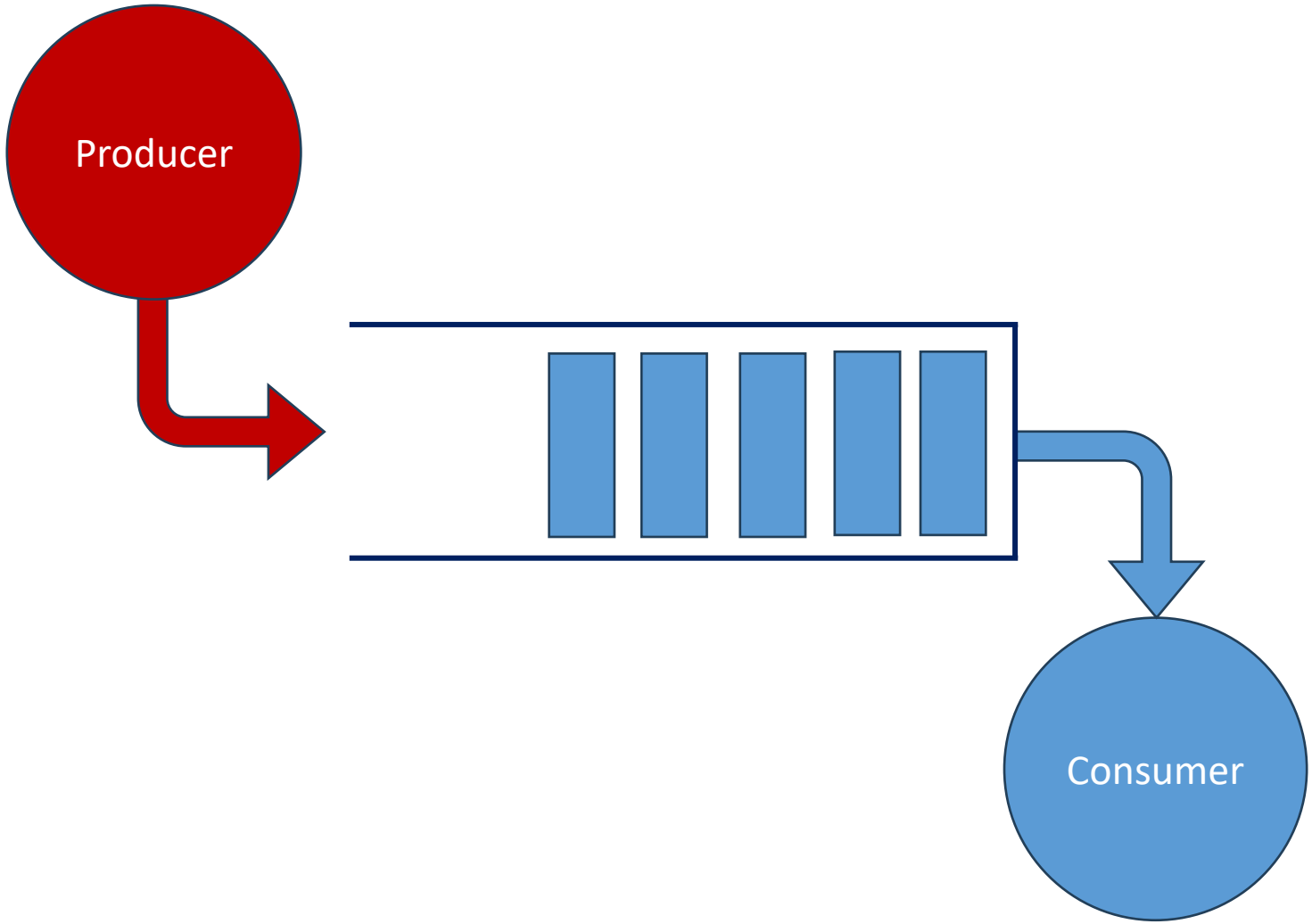
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



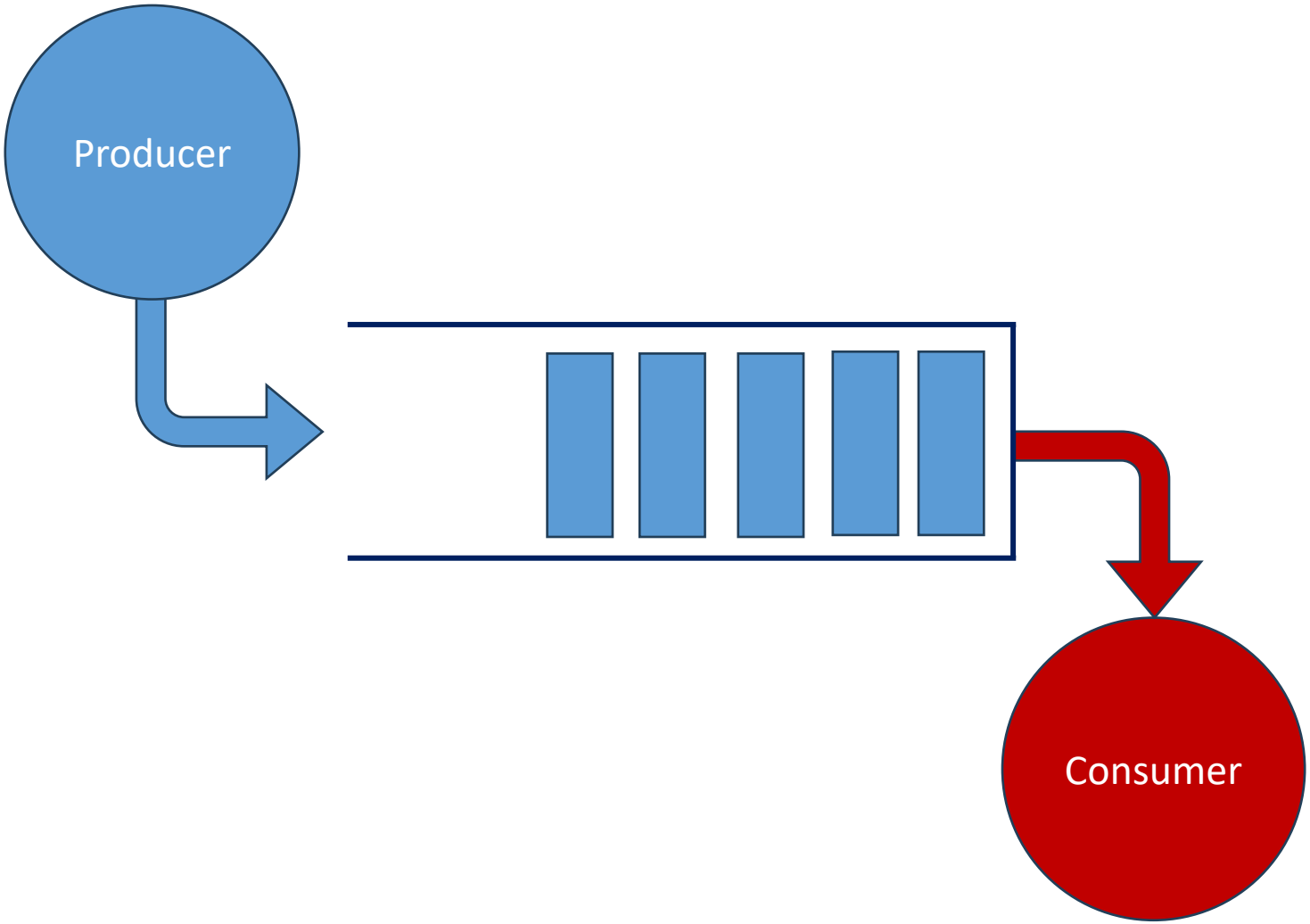
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



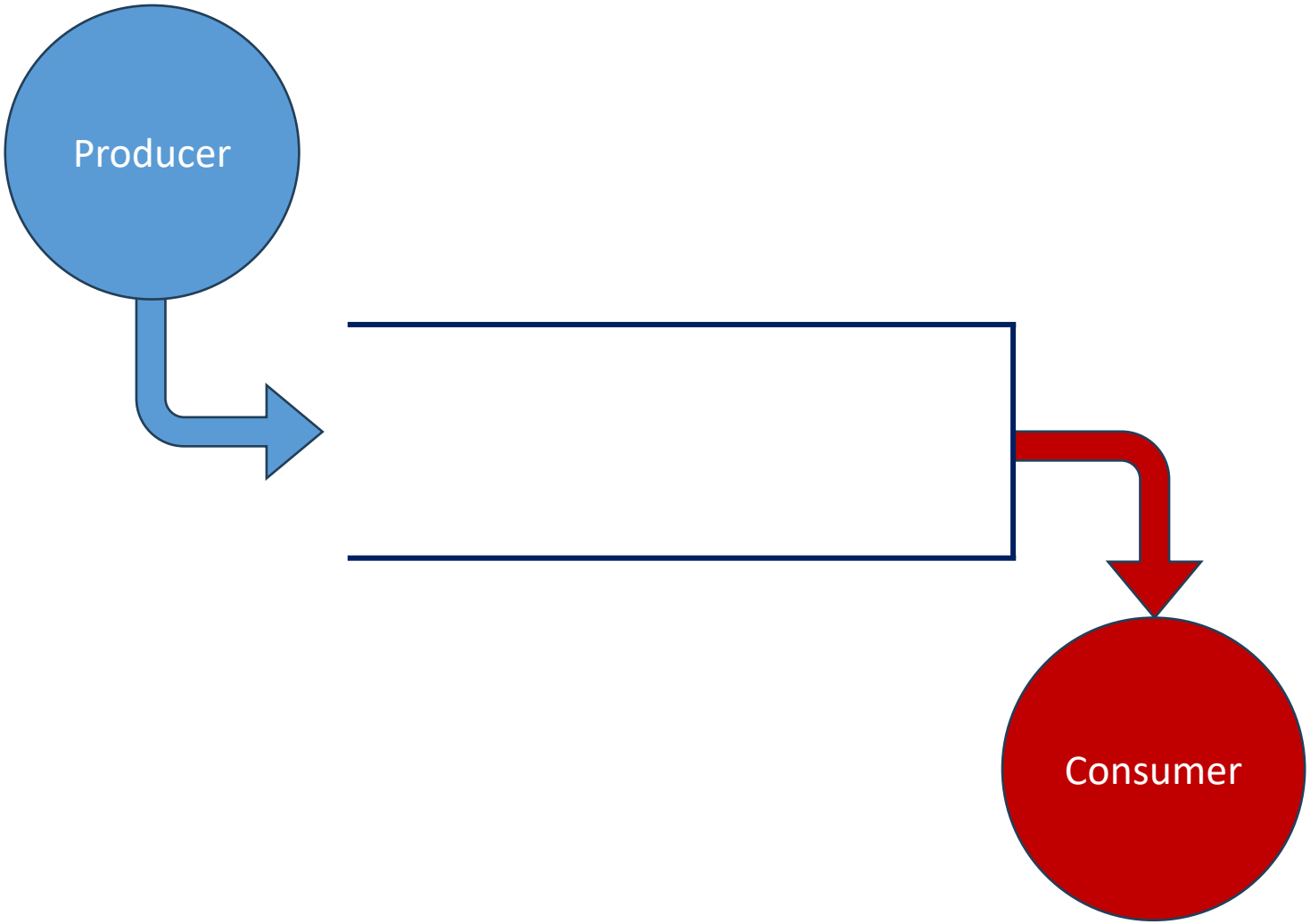
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



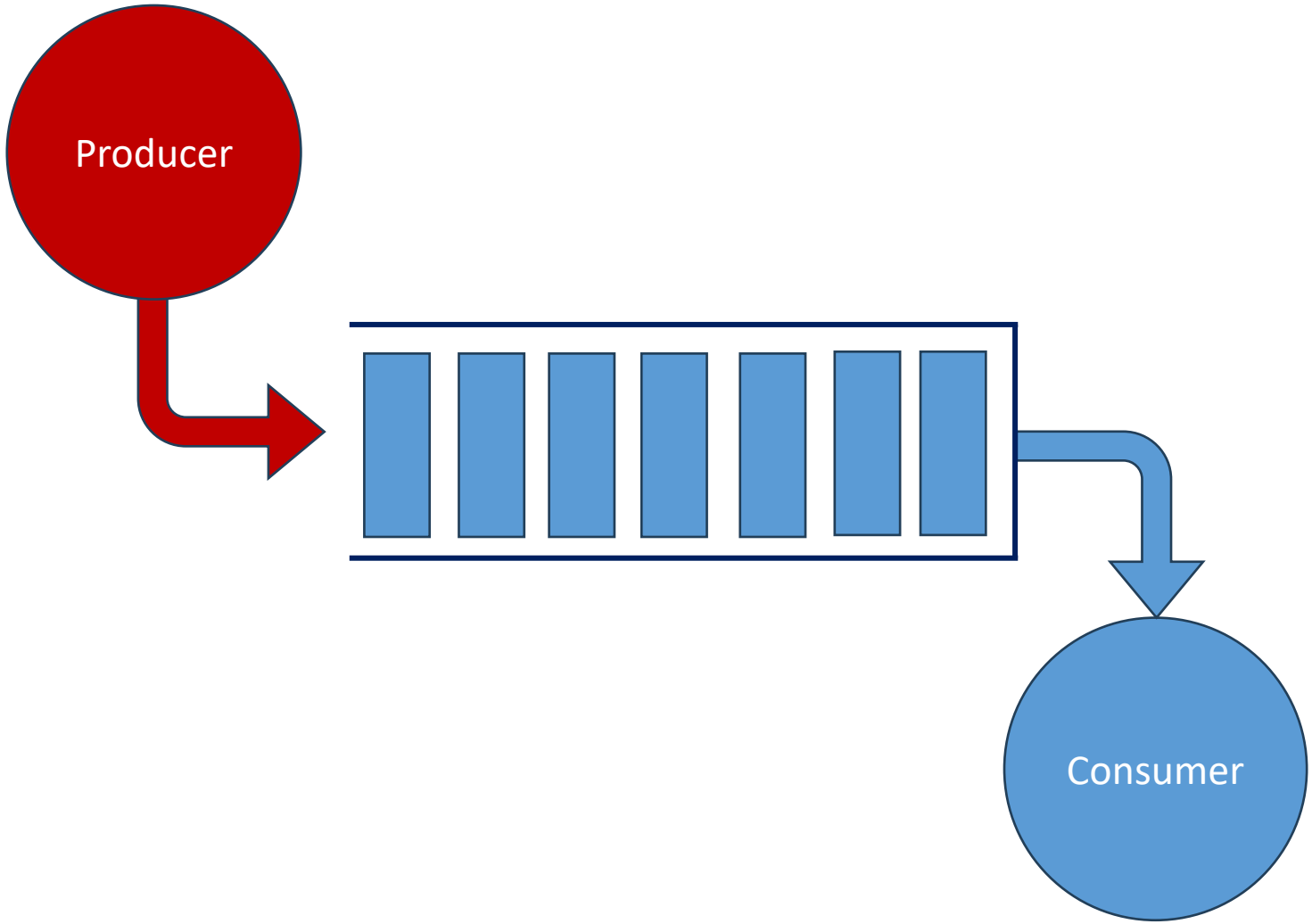
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



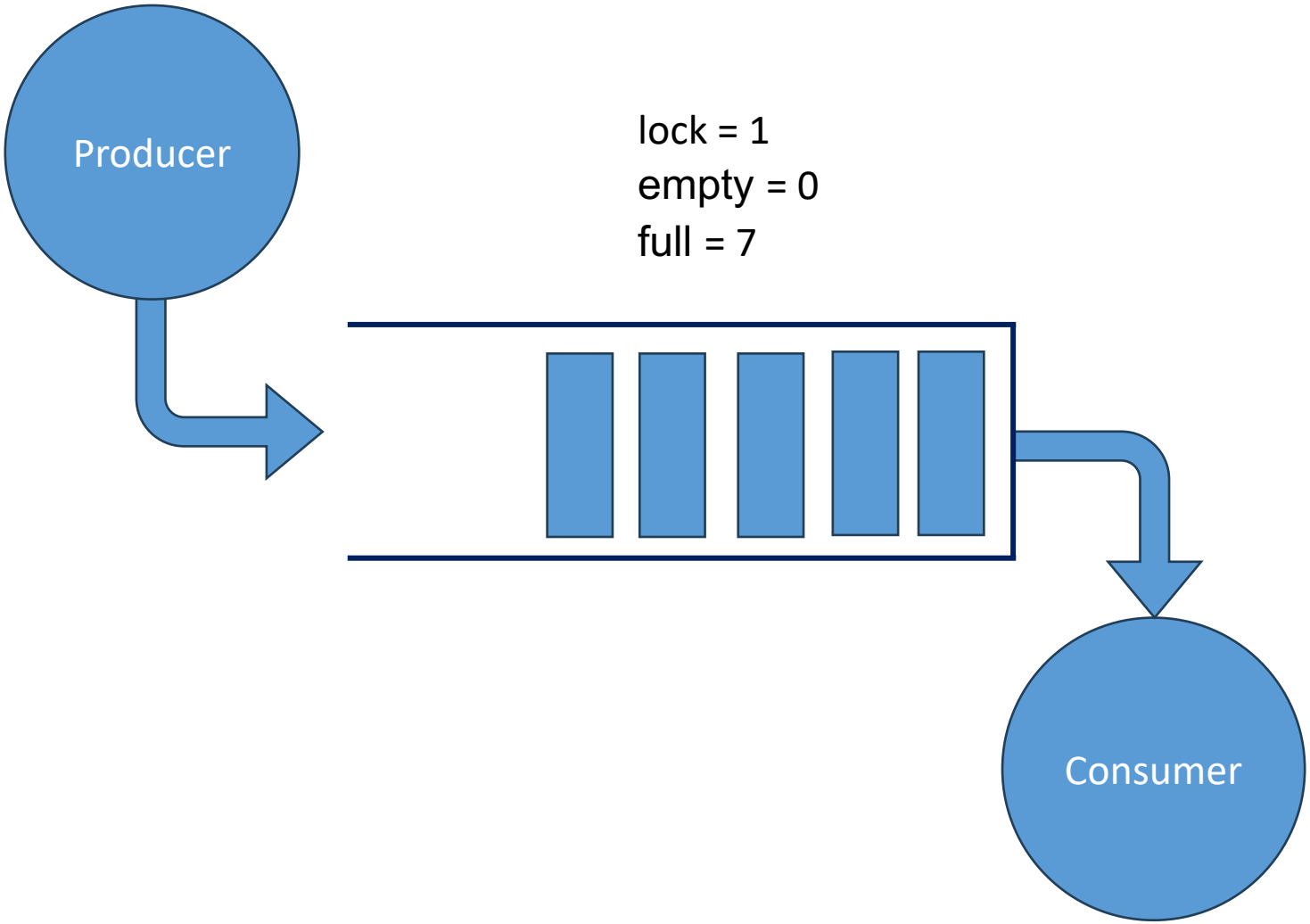
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



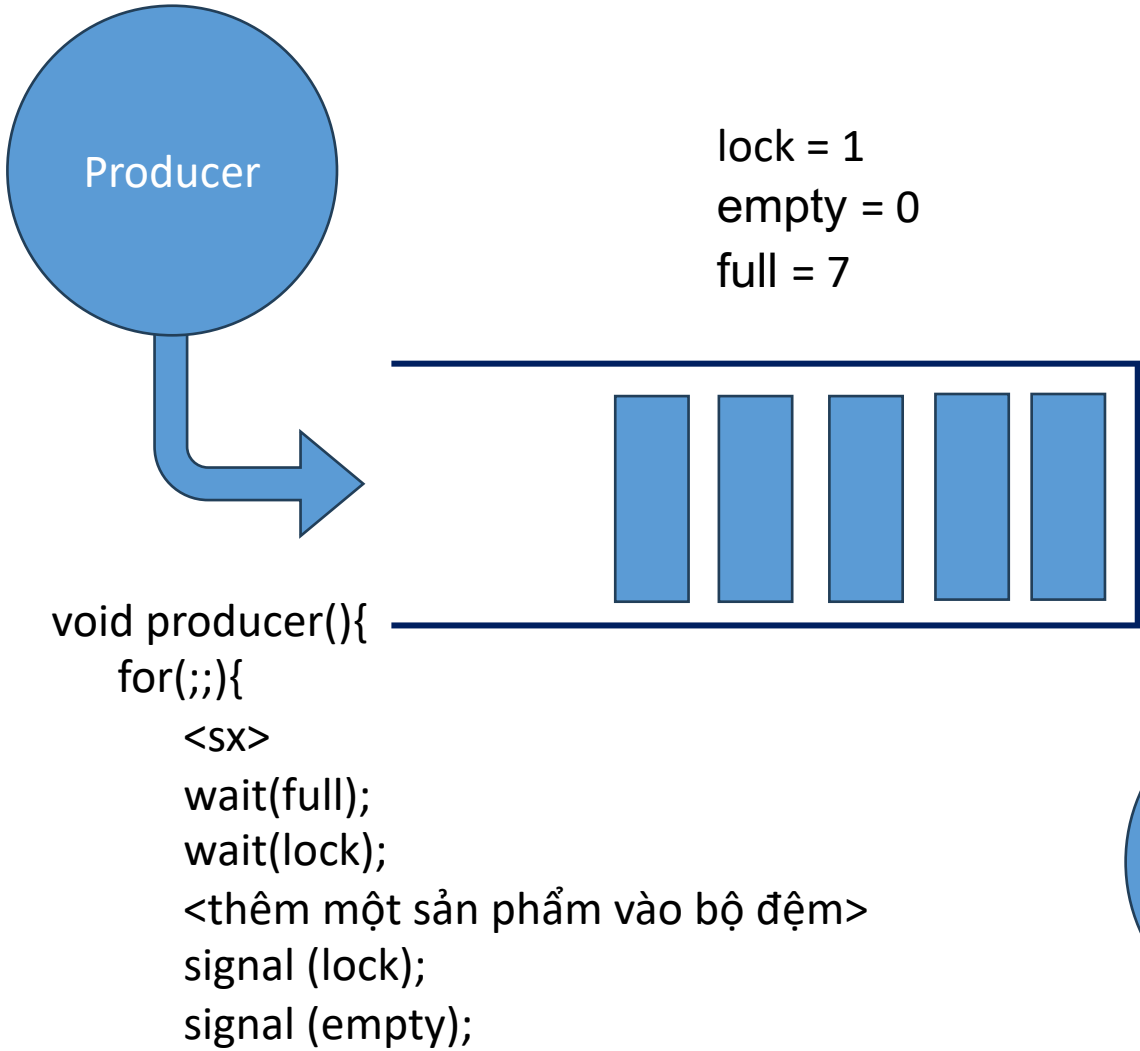
Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)



Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)




```
void producer(){
    for(;;){
        <sx>
        wait(full);
        wait(lock);
        <thêm một sản phẩm vào bộ đệm>
        signal (lock);
        signal (empty);
    }
}
```

```
void consumer(){
    for(;;){
        wait(empty);
        wait(lock);
        <lấy một sản phẩm từ bộ đệm>
        signal (lock);
        signal (full);
        <tiêu dùng>
    }
}
```

Ví dụ: Bài toán nhà sản xuất - người tiêu thụ (producer-consumer)

```
void nsx()
{       while(1)
        {       tạo_sp();
                send(ntt,sp); //gởi sp cho ntt
        }
}
void ntt()
{       while(1)
        {       receive(nsx,sp); //ntt chờ nhận sp
                tiêu_thụ(sp);
        }
}
```

4. Đồng bộ tiến trình

Liên lạc qua socket

- Mỗi tiến trình cần tạo một socket riêng
- Mỗi socket được kết buộc với một cổng khác nhau.
- Các thao tác đọc/ghi lên socket chính là sự trao đổi dữ liệu giữa hai tiến trình.
- Cách liên lạc qua socket
  - Liên lạc kiểu thư tín (socket đóng vai trò bưu cục)
    - "tiến trình gởi" ghi dữ liệu vào socket của mình, dữ liệu sẽ được chuyển cho socket của "tiến trình nhận"
    - "tiến trình nhận" sẽ nhận dữ liệu bằng cách đọc dữ liệu từ socket của "tiến trình nhận"

4. Đồng bộ tiến trình

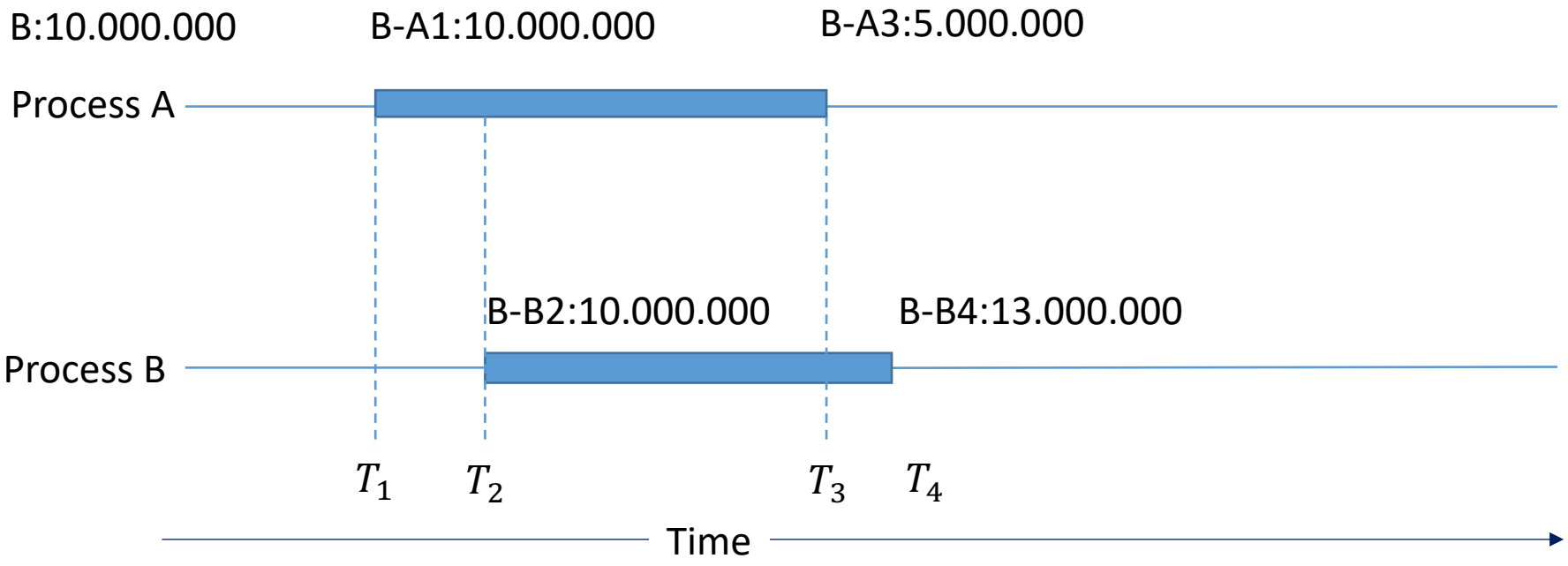
Liên lạc qua socket

- Cách liên lạc qua socket
  - Liên lạc kiểu điện thoại (socket đóng vai trò tổng đài)
    - Hai tiến trình cần kết nối trước khi truyền/nhận dữ liệu và kết nối được duy trì suốt quá trình truyền nhận dữ liệu

4. Đồng bộ tiến trình

- Bảo đảm các tiến trình xử lý song song không tác động sai lệch đến nhau.
- **Yêu cầu độc quyền truy xuất (Mutual exclusion):**
  - tại một thời điểm, chỉ có một tiến trình được quyền truy xuất một tài nguyên không thể chia sẻ.
- **Yêu cầu phối hợp (Synchronization):**
  - các tiến trình cần hợp tác với nhau để hoàn thành công việc.
- **Hai "bài toán đồng bộ" cần giải quyết:**
  - bài toán "độc quyền truy xuất" ("**bài toán miền găng**")
  - bài toán "phối hợp thực hiện".

Miền găng (critical section)

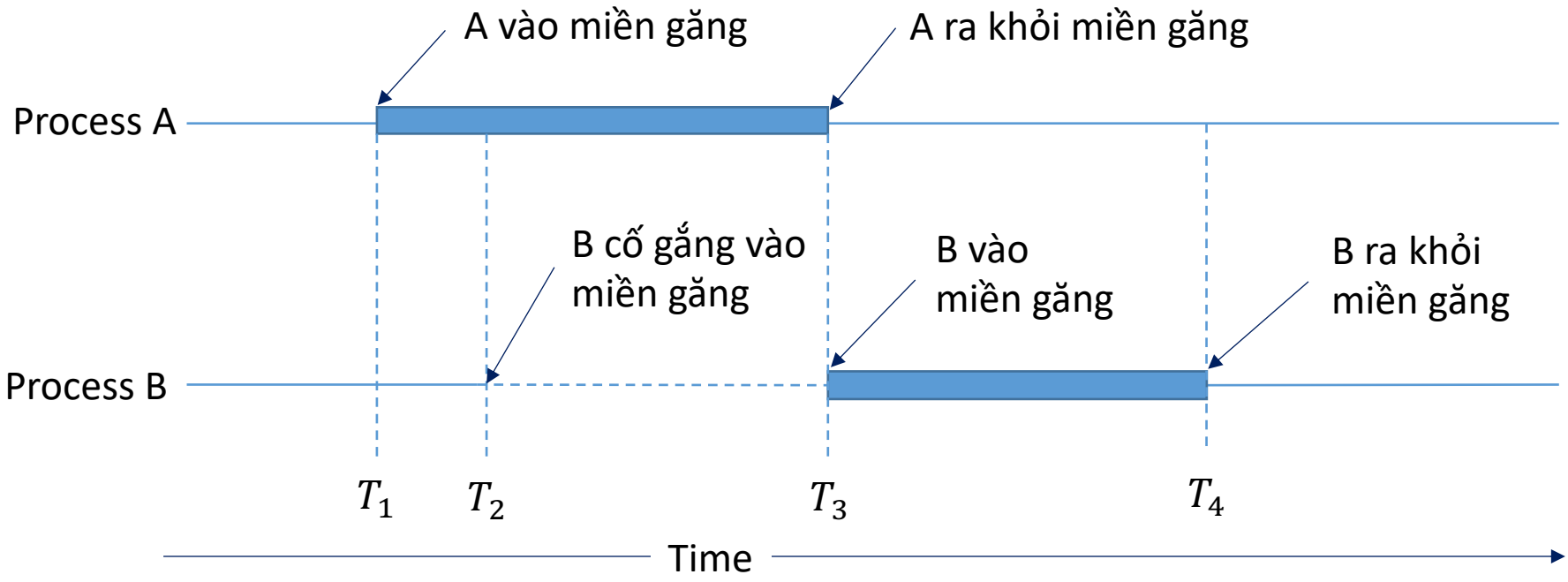


Số tiền ban đầu: 10.000.000
Process A: Rút 5.000.000 VND
Process B: Gửi 3.000.000 VND

Miền găng (critical section)

- Đoạn mã của một tiến trình có khả năng xảy ra lỗi khi truy xuất tài nguyên dùng chung (biến, tập tin,…).
- Ví dụ:

```
if (taikhoan >= tienrut)    taikhoan = taikhoan - tienrut;
        else Thong bao “khong the rut tien !”;
```



4. Đồng bộ tiến trình

❖ **Các điều kiện cần khi giải quyết bài toán miền găng**

- Không có giả thiết về tốc độ của các tiến trình, cũng như về số lượng bộ xử lý.
- Không có hai tiến trình cùng ở trong miền găng cùng lúc.
- Một tiến trình bên ngoài miền găng không được ngăn cản các tiến trình khác vào miền găng.
- Không có tiến trình nào phải chờ vô hạn để được vào miền găng

4. Đồng bộ tiến trình

- **Các nhóm giải pháp đồng bộ**
  - Busy Waiting
  - Sleep And Wakeup
    - Semaphore
    - Monitor
  - Message.

4. Đồng bộ tiến trình

- **Busy Waiting (bận thì đợi)**
  - **Giải pháp phần mềm**
    - Thuật toán sử dụng biến cờ hiệu
    - Thuật toán sử dụng biến luân phiên
    - **Thuật toán Peterson**
  - **Giải pháp phần cứng**
    - Cấm ngắt
    - Sử dụng lệnh TSL (Test and Set Lock)

4. Đồng bộ tiến trình

- Thuật toán sử dụng biến cờ hiệu *(dùng cho nhiều tiến trình)*
  - lock=0 là không có tiến trình trong miền găng.
  - lock=1 là có một tiến trình trong miền găng.

```
lock=0;
while (1)
{
        while (lock == 1);

        lock = 1;
        critical-section ();.
        lock = 0;

        noncritical-section();
}
```

Vi phạm: "Hai tiến trình có thể cùng ở trong miền găng tại một thời điểm".

4. Đồng bộ tiến trình

- Thuật toán sử dụng biến luân phiên (dùng cho 2 tiến trình)
  - Hai tiến trình A, B sử dụng chung biến turn:
    - turn = 0, tiến trình A được vào miền găng
    - turn=1 thì B được vào miền găng

```
// Tiến trình A
while(1){
    while(turn == 1);
    critical-section();
    turn = 1;
    nritical-section();
}
```

```
// Tiến trình B
while(1){
    while(turn == 0);
    critical-section();
    turn = 0;
    nritical-section();
}
```

  - Hai tiến trình chắc chắn không thể vào miền găng cùng lúc, vì tại một thời điểm turn chỉ có 1 giá trị.
  - Vi phạm: 1 tiến trình có thể bị ngăn chặn vào miền găng bởi một tiến trình khác không ở trong miền găng.

4. Đồng bộ tiến trình

Thuật toán Peterson (dùng cho 2 tiến trình)

- Dùng chung hai biến turn và flag[2] (kiểu int).
  - flag[0] = flag[1] = FALSE
  - turn được khởi động là 0 hay 1.
- Nếu flag[i] = TRUE (i=0,1)→ P_i muốn vào miền găng và turn = i là đến lượt P_i
- Để có thể được vào miền găng:
  - P_i đặt giá trị flag[i] TRUE để thông báo nó muốn vào miền găng.
  - Đặt turn=j để thử đề nghị tiến trình P_j vào miền găng.

4. Đồng bộ tiến trình

- Nếu tiến trình P_j không quan tâm đến việc vào miền găng (flag[j] = FALSE), thì P_i có thể vào miền găng.
- Nếu flag[j] = TRUE thì P_i phải chờ đến khi flag[j] = FALSE.
- Khi tiến trình P_i ra khỏi miền găng, nó đặt lại giá trị flag[i] = FALSE.

4. Đồng bộ tiến trình

- **Thuật toán Peterson**

```
//tiến trình P0 (i=0)
while(TRUE){
        flag[0] = TRUE;
        turn = 1;
        while( turn == 1 && flag[1] == TRUE);
        critical_section();
        flag[0] = FALSE;
        non_critical_section();
}
```

```
//tiến trình P1 (i=1)
while(TRUE){
        flag[1] = TRUE;
        turn = 0;
        while( turn == 0 && flag[0] == TRUE);
        critical_section();
        flag[1] = FALSE;
        non_critical_section();
}
```

4. Đồng bộ tiến trình

- Cấm ngắt
  - Tiến trình cấm tất cả các ngắt trước khi vào miền găng, và phục hồi ngắt khi ra khỏi miền găng.
    - Không an toàn cho hệ thống
    - Không tác dụng trên hệ thống có nhiều bộ xử lý.

4. Đồng bộ tiến trình

- **Sử dụng TSL (Test and Set Lock)**

```
booean Test_And_Set_Lock(boolean lock){
    boolean temp = lock;
    lock = TRUE;
    return temp;//trả về giá trị ban đầu của biến lock
}
```

Lệnh TSL cho phép kiểm tra và cập nhật một vùng nhớ trong một thao tác độc quyền

```
boolean lock = FALSE; //biến dùng chung
while (TRUE) {
    while (Test_And_Set_Lock(lock));
    critical_section ();
    lock = FALSE;
    noncritical_section ();
}
```

Hoạt động trên hệ thống có nhiều bộ xử lý

4. Đồng bộ tiến trình

❖ **Nhóm giải pháp "SLEEP and WAKEUP" (ngủ và đánh thức)**

1. Sử dụng lệnh SLEEP và WAKEUP
2. Sử dụng cấu trúc Semaphore
3. Sử dụng cấu trúc Monitors
4. Sử dụng thông điệp

4. Đồng bộ tiến trình

❖ Sử dụng lệnh SLEEP và WAKEUP

- SLEEP → "danh sách sẵn sàng", lấy lại CPU cấp cho Tiến trình khác.
- WAKEUP → HĐH chọn một tiến trình trong ready list, cho thực hiện tiếp.
- Tiến trình chưa đủ điều kiện vào miền găng → gọi SLEEP để tự khoá, đến khi có Tiến trình khác gọi WAKEUP để giải phóng cho nó.
- Một tiến trình gọi WAKEUP khi ra khỏi miền găng để đánh thức một tiến trình đang chờ, tạo cơ hội cho. tiến trình này vào miền găng.

4. Đồng bộ tiến trình

❖ Sử dụng lệnh SLEEP và WAKEUP

```
1   int busy = FALSE;
2   int blocked = 0;
3   while (TRUE){
4           if (busy) {
5                   blocked = blocked + 1;
6                   sleep();
7           else busy = TRUE;
8           critical-section ();
9           busy = FALSE;
10          if (blocked > 0){
11                  wakeup();//đánh thức một tiến trình đang chờ
12                  blocked = blocked -1;
13          }
14          noncritical-section();
15  }
```

4. Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

- Biến semaphore s có các thuộc tính:
  - Một giá trị nguyên dương e;
  - Một hàng đợi f: lưu danh sách các tiến trình đang chờ trên semaphore s.
- Hai thao tác trên semaphore s:
  - Down(s): e = e – 1
    - ✓ Nếu e < 0 thì tiến trình phải chờ trong f (sleep), ngược lại tiến trình tiếp tục.
  - Up(s): e = e + 1
    - ✓ Nếu e ≤ 0 thì chọn một tiến trình trong f cho tiếp tục thực hiện (đánh thức)

4. Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

```
1   Down (s) {
2           e = e – 1;
3           if (e < 0){
4                   status (P) = blocked; //khoá P (trạng thái chờ)
5                   enter(P, f); //cho P vào hàng đợi f
6           }
7   }
8   Up (s) {
9           e = e + 1
10          if (e <= 0) {
11                  exit (Q, f); //lấy một tiến trình Q ra khỏi hàng đợi f
12                  status (Q) = ready; //chuyển Q sang trạng thái sẵn sàng
13                  enter (Q, ready-list); //đưa Q vào danh sách sẵn sàng
14          }
15  }
```

4. Đồng bộ tiến trình

❖ Sử dụng cấu trúc Semaphore

- Hệ điều hành cần cài đặt các thao tác Down, Up là độc quyền.
- Cấu trúc của semaphore:

```
class semaphore {
        int e;
        PCB *f; danh sách riêng của semaphore
        public:
        down ();
        up ();
}
```

- |e| = số tiến trình đang chờ trên f

4. Đồng bộ tiến trình

❖ **Giải quyết bài toán miền găng bằng Semaphore**

- Dùng một semaphore s, e được khởi tạo gán là 1.
- Tất cả các tiến trình áp dụng cùng cấu trúc chương trình sau:

```
1  semaphore s – 1; //nghĩa là e của s = 1
2  while (1)
3  {
4      Down (s);
5      critical-section ();
6      Up (s);
7      Noncritical-section ();
8  }
```

4. Đồng bộ tiến trình

❖ Giải quyết bài toán đồng bộ bằng Semaphore

- Hai tiến trình đồng hành P1 và P2, P1 thực hiện công việc 1, P2 thực hiện công việc 2.
- Công việc 1 làm trước rồi mới làm công việc 2, cho P1 và P2 dùng chung một semaphore s, khởi gán e(s) = 0:

```
P1:{
        job1();
        Up(s); //đánh thức P2
}
P2:{
        Down(s); //chờ P1 đánh thức
        job2();
}
```

4. Đồng bộ tiến trình

❖ **Vấn đề khi sử dụng semaphore**

- Tiến trình quên gọi Up(s), và kết quả là khi ra khỏi miền găng nó sẽ không cho tiến trình khác vào miền găng.

```
e (s) = 1;
while (1)
{
        Down(s)
        critical-section ();
        Noncritical-section();
}
```

4. Đồng bộ tiến trình

❖ Vấn đề khi sử dụng semaphore

- Sử dụng semaphore có thể gây ra tình trạng tắc nghẽn.

```
P1:{
        down(s1); down(s2);
        …
        up(s1); up(s2);
}
P2:{
        down(s2); down(s1);
        …
        up(s2); up(s1);
}
```

Hai tiến trình P1, P2 sử dụng chung 2 semaphore s1=s2=1

Nếu thứ tự thực hiện như sau:
P1: down(s1), P2: down(s2),
P1: down(s2), P2: down(s1)
Khi đó s1=s2=-1 nên P1, P2 đều chờ mãi

4. Đồng bộ tiến trình

❖ Sử dụng cấu trúc Monitors

- Hoare (1974) và Brinch & Hansen (1975) đã đề nghị một cơ chế cao hơn gọi là monitor được cung cấp bởi một số ngôn ngữ lập trình
- Monitor là một cấu trúc đặc biệt (lớp)
  - Các phương thức độc quyền (critical-section)
  - Các biến (được dùng chung cho các tiến trình)
    - Các biến trong monitor chỉ có thể được truy xuất bởi các phương thức trong monitor
  - Tại một thời điểm, chỉ có một tiến trình duy nhất được hoạt động bên trong một monitor.
  - Biến điều kiện c
    - Dùng để đồng bộ việc sử dụng các biến trong monitor
    - wait(c) và Signal(z:)

4. Đồng bộ tiến trình

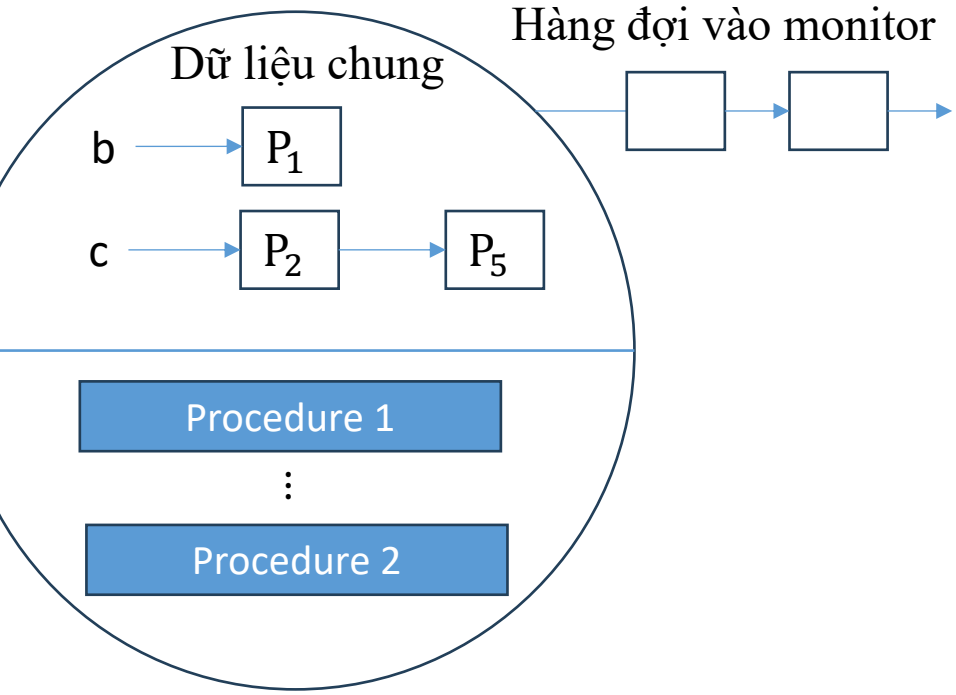
❖ Sử dụng cấu trúc Monitors

```
Wait (c) {
        status(P) = blocked;
        enter(P, f(c) );
}
Signal (c) {
        if (f(c) != NULL) {
                exit(Q, f(c) );
                status (Q) = ready;
                enter (Q, ready-list);
        }
}
```

Wait(c): chuyển trạng thái tiến trình gọi sang chờ (blocked) và đặt tiến trình này vào hàng đợi trên biến điều kiện c.

Signal(c): khi có một tiến trình đang chờ trong hàng đợi c, kích hoạt tiến trình đó và tiến trình gọi sẽ rời khỏi monitor.

4. Đồng bộ tiến trình



critical-section

```
monitor <tên monitor> //khai báo monitor dùng chung cho các tiến trình
{
        <các biến dùng chung>;
        <các biến điều kiện>;
        <các phương thức độc quyền>;
}
//tiến trình Pi:
while(1) {//cấu trúc tiến trình thứ i
        Noncritical-section();
        <tên monitor>.Phương_thức_i;
        //thực hiện công việc độc quyền thứ i
        Noncritical-section ();
}
```

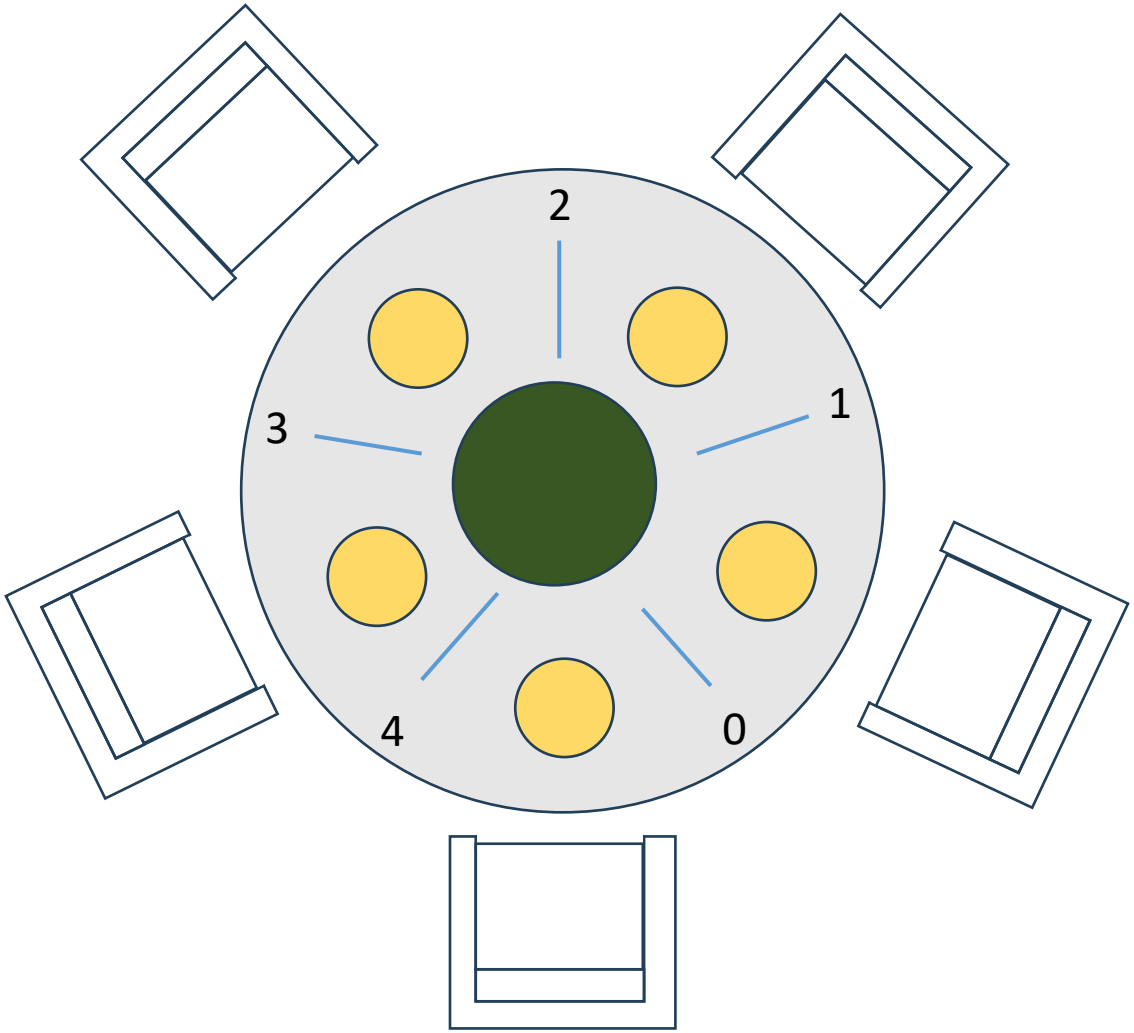
4. Đồng bộ tiến trình

❖ **Sử dụng cấu trúc Monitors**

- Nguy cơ thực hiện đồng bộ hoá sai giảm rất nhiều.
- Ít có ngôn ngữ hỗ trợ cấu trúc monitor.

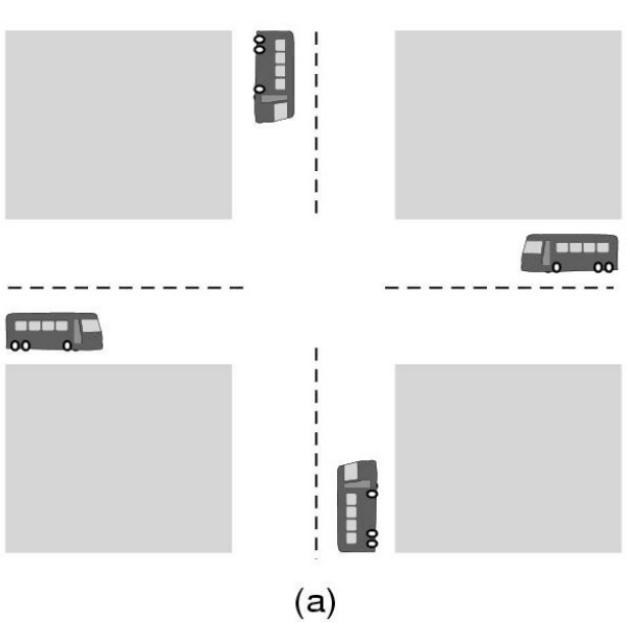
4. Đồng bộ tiến trình

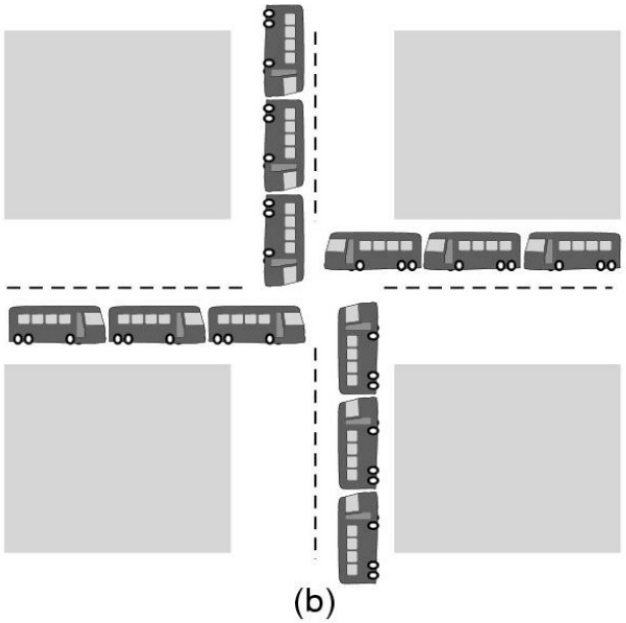
❖ **Monitor và bài toán 5 triết gia ăn tối**



5. Tình trạng tắc nghẽn

- Một tập hợp các tiến trình gọi là ở tình trạng tắc nghẽn nếu mỗi tiến trình trong tập hợp đều chờ đợi tài nguyên mà tiến trình khác trong tập hợp đang chiếm giữ.





5. Tình trạng tắc nghẽn (deadlock)

- **Điều kiện xuất hiện tắc nghẽn**
  - Điều kiện 1: Có sử dụng tài nguyên không thể chia sẻ
  - Điều kiện 2: Sự chiếm giữ và yêu cầu thêm tài nguyên không thể chia sẻ.
  - Điều kiện 3: Không thu hồi tài nguyên từ tiến trình đang giữ chúng.
  - Điều kiện 4: Tồn tại một chu trình trong đồ thị cấp phát tài nguyên.

5. Tình trạng tắc nghẽn (deadlock)

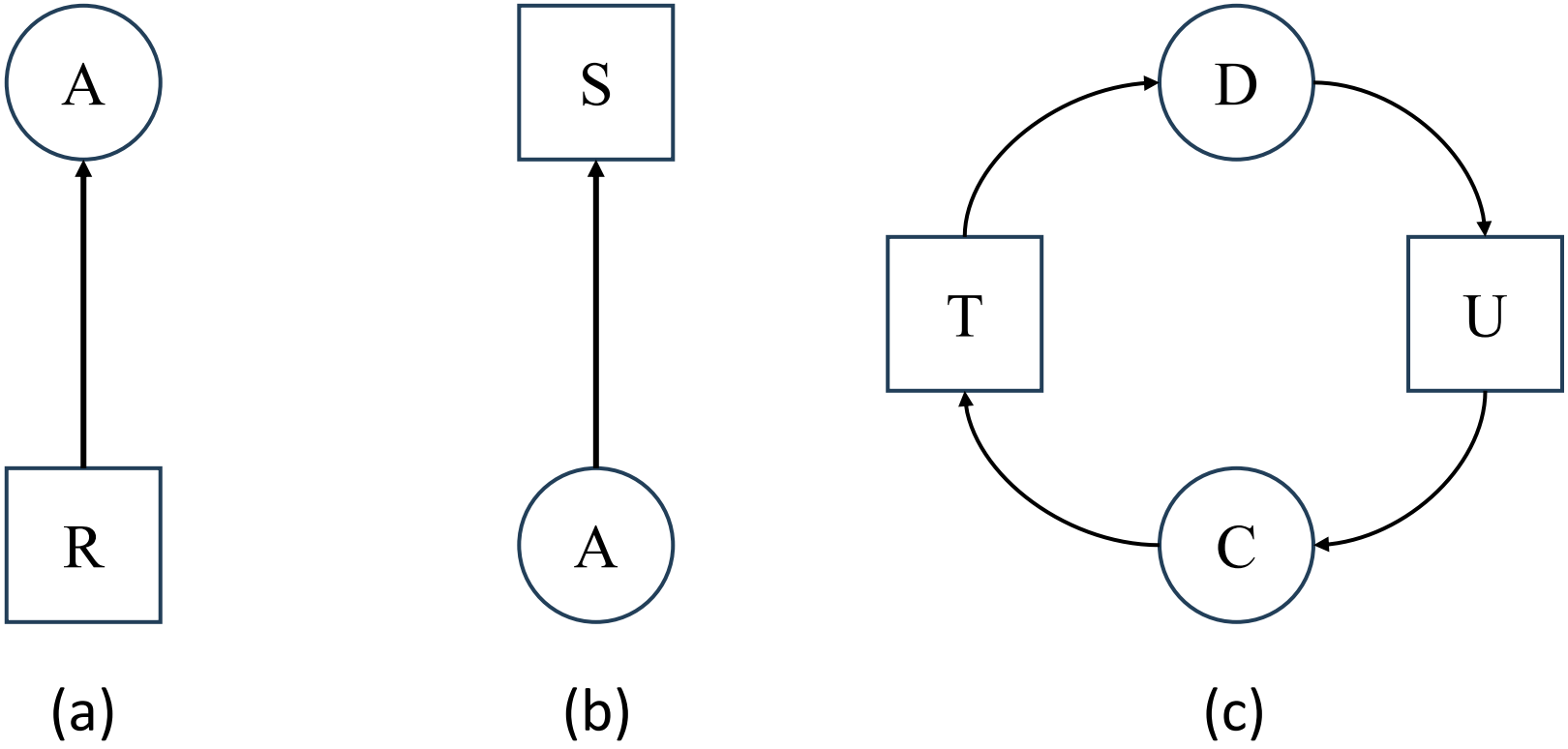
- **Đồ thị cấp phát tài nguyên**

(a) A request R

(b) Release S

(c) C request U

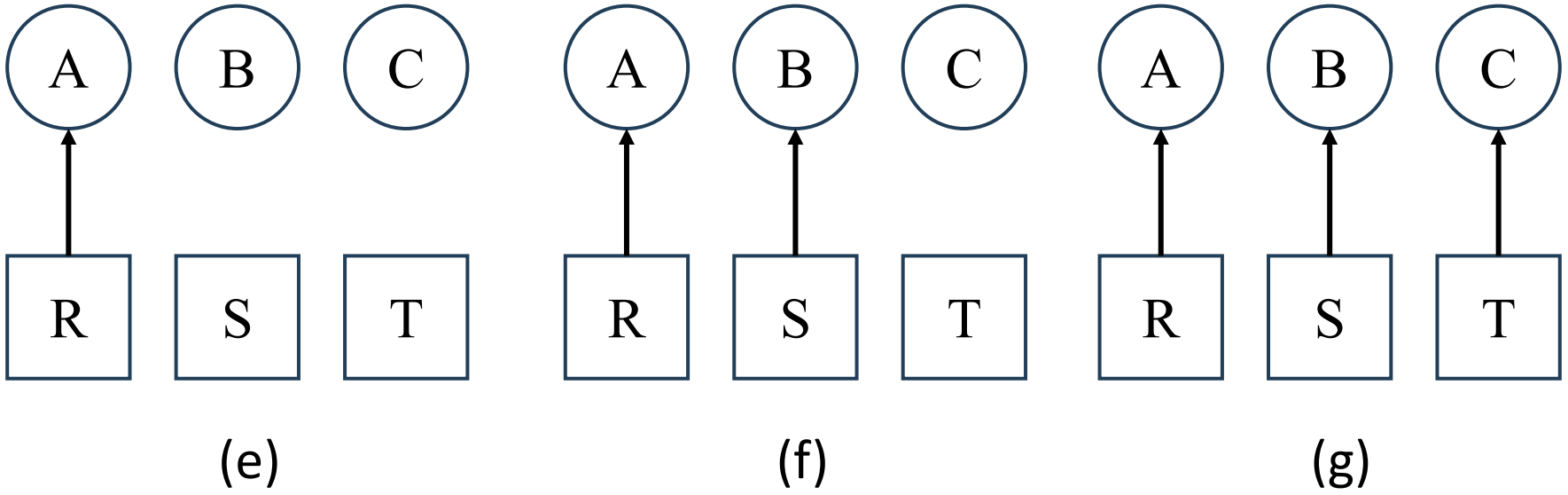
D request T



(a) (b) (c)

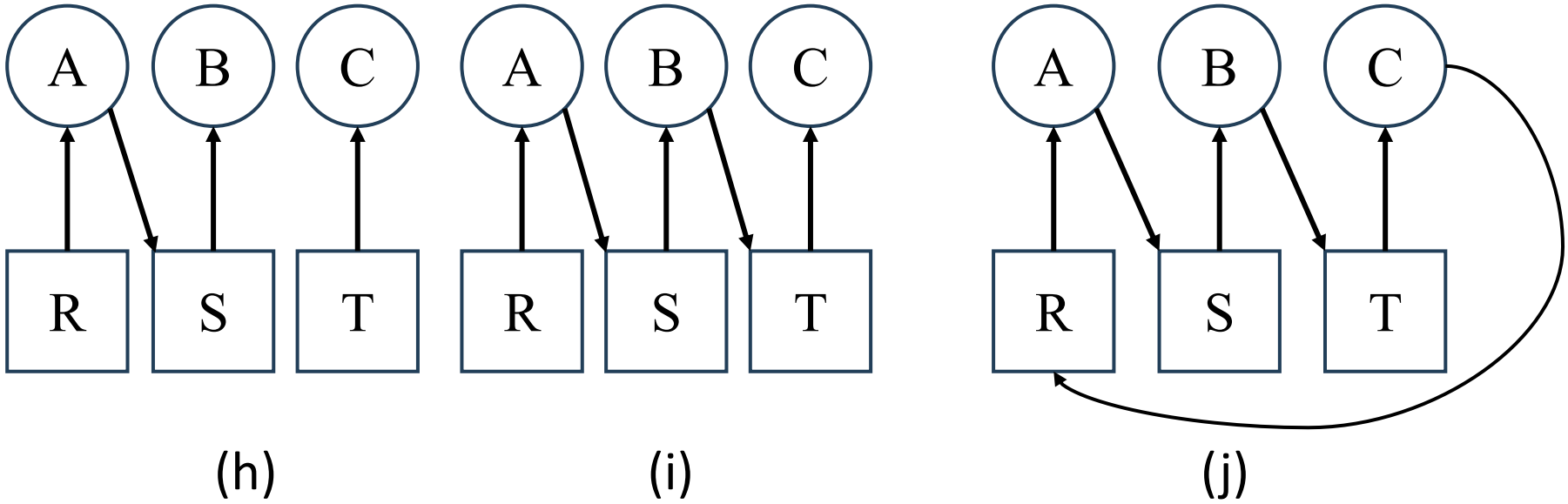
5. Tình trạng tắc nghẽn (deadlock)

- **Đồ thị cấp phát tài nguyên**



5. Tình trạng tắc nghẽn (deadlock)

- **Đồ thị cấp phát tài nguyên**



(h) (i) (j)

5. Tình trạng tắc nghẽn (deadlock)

❖ **Các phương pháp xử lý tắc nghẽn và ngăn chặn tắc nghẽn.**

- Sử dụng một thuật toán cấp phát tài nguyên → không bao giờ xảy ra tắc nghẽn.
- Cho phép xảy ra tắc nghẽn → tìm cách sữa chữa tắc nghẽn.
- Bỏ qua việc xử lý tắc nghẽn, xem như hệ thống không bao giờ xảy ra tắc nghẽn.

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ngăn chặn tắc nghẽn.**

- **Điều kiện 1** gần như không thể tránh được.
- **Để điều kiện 2** không xảy ra:
  - ➢ Tiến trình phải yêu cầu tất cả các tài nguyên cần thiết trước khi cho bắt đầu xử lý.
  - ➢ Khi tiến trình yêu cầu một tài nguyên mới và bị từ chối.
    - Giải phóng các tài nguyên đang chiếm giữ.
    - Tài nguyên cũ được cấp lại cùng với tài nguyên mới.
- **Để điều kiện 3** không xảy ra:
  - ➢ Thu hồi tài nguyên từ các tiến trình bị khoá và cấp phát trở lại cho tiến trình khi nó thoát khỏi tình trạng bị khoá.
- **Để điều kiện 4** không xảy ra:
  - ➢ Khi tiến trình đang chiếm giữ tài nguyên R_i thì chỉ có thể yêu cầu các tài nguyên $\boldsymbol{R_j}$ ***nếu*** $\boldsymbol{F(R_j) > F(R_i)}$

5. Tình trạng tắc nghẽn (deadlock)

❖ **Giải thuật cấp phát tài nguyên tránh tắc nghẽn.**

- Giải thuật xác định trạng thái an toàn
- Giải thuật Banker

5. Tình trạng tắc nghẽn (deadlock)

- **Giải thuật xác định trạng thái an toàn**

int **NumResources**; *// số tài nguyên*

int **NumProcs**; *// số tiến trình trong hệ thống*

int **Available**[NumResources]; *// số lượng tài nguyên còn tự do*

int **Max**[NumProcs, NumResources];

//Max[p, r] là nhu cầu tối đa của tiến trình p về tài nguyên r

int **Allocation**[NumProcs, NumResources];

//Allocation[p, r] là số tài nguyên r đã cấp phát cho tiến trình p

int Need[NumProcs, NumResources];

//Need[p, r] = Max[p,r] – Allocation[p, r] là số tài nguyên r mà tiến trình p còn cần sử dụng

int Finish[NumProcs] = false; *//= true nếu process P thực thi xong*

5. Tình trạng tắc nghẽn (deadlock)

❖ Giải thuật xác định trạng thái an toàn

Bước 1.

if ∃i:

Finish[i] = false; //tiến trình i chưa thực thi xong

Need[i, j] ≤ Available[j], ∀j //Yêu cầu của process i được thoả

Do **Bước 2** else goto **Bước 3**

Bước 2. Cấp phát mọi tài nguyên mà tiến trình i cần

Allocation[i, j] = Allocation[i, j] + Need[i, j]; ∀j

Need[i, j] = 0; ∀j

Available[j] = Available[j] – Need[i, j];

Finish[i] = true;

Available[j] = Available[j] – Allocation[i, j];

goto **Bước 1**

Bước 3. ∀j if Finish[i] = true → <hệ thống ở trạng thái an toàn>,

else <không an toàn>

5. Tình trạng tắc nghẽn (deadlock)

❖ **Giải thuật Banker**: Pi yêu cầu kr thể hiện của tài nguyên r

Bước 1.		
	If kr ≤ Need[i, r] ∀r goto **Bước 2.**	
		else <Lỗi>
Bước 2.		
	If kr ≤ Available[r] ∀r, goto **Bước 3**	
		else Pi <wait>
Bước 3.		
	∀r:	
		Available[r] = Available[r] – kr;
	Allocation[i, r] = Allocation[i, r] + kr;	
	Need[i, r] = Need[i, r] – kr;	
Bước 4:		
	Kiểm tra trạng thái an toàn của hệ thống	

5. Tình trạng tắc nghẽn (deadlock)

❖ Ví dụ cấp phát tài nguyên tránh tắc nghẽn.

	Max			**Allocation**			**Available**		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	3	2	2	1	0	0	4	1	2
P_2	6	1	3	2	1	1			
P_3	3	1	4	2	1	1			
P_4	4	2	2	0	0	2			

❖Nếu tiến trình P_2 yêu cầu 4 R_1, 1 R_3, hãy cho biết yêu cầu này có thể đáp ứng mà không xảy ra **deadlock** hay không?

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

- Bước 0: Tính Need là nhu cầu còn lại về mỗi tài nguyên j của mỗi tiến trình i:
- Need[i, j] = Max[i, j] – Allocation[i, j]

	Need			**Allocation**			**Available**		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	2	1	0	0	4	1	2
P_2	4	0	2	2	1	1			
P_3	1	0	3	2	1	1			
P_4	4	2	0	0	0	2			

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

- Bước 1+B2: Yêu cầu tài nguyên của P_2 thoả điều kiện ở Bước 1, Bước 2.
- Bước 3: Thử cấp phát cho P_2, cập nhật tình trạng hệ thống

	Need			Allocation			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	2	1	0	0	0	1	1
P_2	0	0	1	6	1	2			
P_3	1	0	3	2	1	1			
P_4	4	2	0	0	0	2			

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

- Bước 4: Kiểm tra trạng thái an toàn của hệ thống.
- Lần lượt chọn tiến trình để thử cấp phát:
  - Chọn P_2, thử cấp phát, giả sử P_2 thực thi xong thu hồi
  - Available[j] = Available[j] + Allocation[i, j];

	Need			**Allocation**			**Available**		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	2	2	2	1	0	0	6	2	3
P_2	0	0	0	0	0	0			
P_3	1	0	3	2	1	1			
P_4	4	2	0	0	0	2			

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

Chọn P_1									
	Need			Allocation			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	0	0	0	0	7	2	3
P_2	0	0	0	0	0	0			
P_3	1	0	3	2	1	1			
P_4	4	2	0	0	0	2			

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

Chọn P_3									
	Need			Allocation			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	0	0	0	0	9	3	4
P_2	0	0	0	0	0	0			
P_3	0	0	0	0	0	0			
P_4	4	2	0	0	0	2			

5. Tình trạng tắc nghẽn (deadlock)

❖ **Ví dụ cấp phát tài nguyên tránh tắc nghẽn.**

Chọn P_4									
	Need			Allocation			Available		
	R_1	R_2	R_3	R_1	R_2	R_3	R_1	R_2	R_3
P_1	0	0	0	0	0	0	9	3	4
P_2	0	0	0	0	0	0			
P_3	0	0	0	0	0	0			
P_4	4	2	0	0	0	2			

- **Mọi tiến trình đã được cấp phát tài nguyên với yêu cầu cao nhất, nên trạng thái của hệ thống là an toàn, do đó có thể cấp phát các tài nguyên theo yêu cầu của P2.**

5. Tình trạng tắc nghẽn (deadlock)

- Giải thuật phát hiện tắc nghẽn.
  - Đối với **Tài nguyên chỉ có một thể hiện**
  - Đối với **Tài nguyên có nhiều thể hiện.**

5. Tình trạng tắc nghẽn (deadlock)

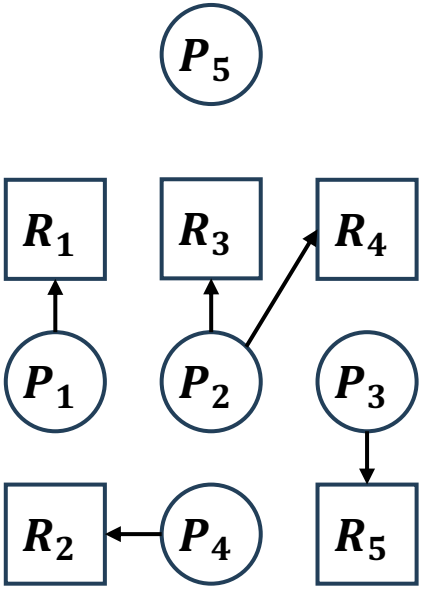
❖ **Đối với Tài nguyên chỉ có một thể hiện.**

- Dùng đồ thị đợi tài nguyên (wait-for grapth)
  - Được xây dựng từ đồ thị cấp phát tài nguyên (resource-allocation graph) bằng cách bỏ những đỉnh biểu diễn loại tài nguyên.
  - Cạnh từ Pi tới Pj: Pj đang đợi Pi giải phòng một tài nguyên mà Pj cần.
- Hệ thống bị tắc nghẽn nếu và chỉ nếu đồ thị đợi tài nguyên có chu trình.

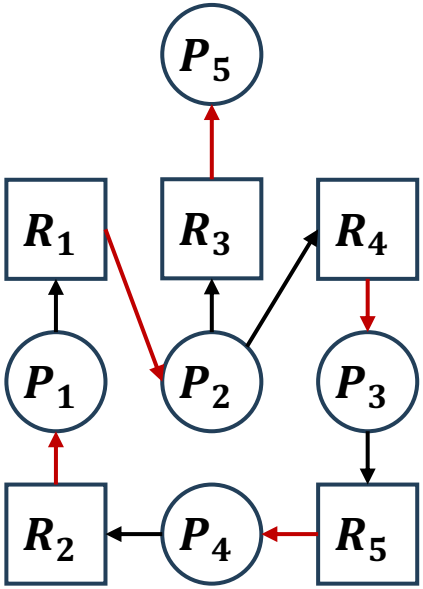
5. Tình trạng tắc nghẽn (deadlock)

❖ **Đối với Tài nguyên chỉ có một thể hiện.**

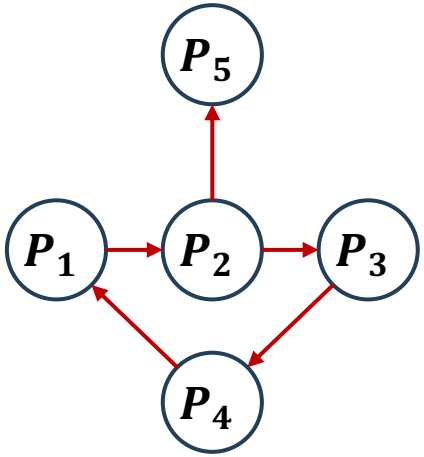
Tiến trình	Yêu cầu	Chiếm giữ
P_1	R_1	R_2
P_2	R_3, R_4	R_1
P_3	R_5	R_4
P_4	R_2	R_5
P_5		R_3



Đồ thị cấp phát tài nguyên



Đồ thị cấp phát tài nguyên
Theo yêu cầu



Đồ thị đợi tài nguyên

5. Tình trạng tắc nghẽn (deadlock)

❖ **Đối với Tài nguyên có nhiều thể hiện.**

- **Bước 1:** Chọn Pi đầu tiên sao cho có yêu cầu tài nguyên có thể được đáp ứng, nếu không có thì hệ thống bị tắc nghẽn.
- **Bước 2:** Thử cấp phát tài nguyên cho Pi và kiểm tra trạng thái hệ thống, nếu hệ thống an toàn thì tới Bước 3, ngược lại thì quay về Bước 1 tìm Pi kế tiếp.
- **Bước 3:** Cấp phát tài nguyên cho Pi. Nếu tất cả Pi được đáp ứng thì hệ thống không bị tắc nghẽn, ngược lại quay lại Bước 1.

5. Tình trạng tắc nghẽn (deadlock)

❖ Hiệu chỉnh tắc nghẽn

- Huỷ tiến trình trong tình trạng tắc nghẽn.
  - Huỷ đến khi không còn chu trình gây tắc nghẽn.
  - Dựa vào các yếu tố như độ ưu tiên, thời gian đã xử lý, số tài nguyên đang chiếm giữ, hoặc yêu cầu thêm…
- Thu hồi tài nguyên
  - **Chọn lựa một nạn nhân**: tiến trình nào sẽ bị thu hồi tài nguyên? Và thu hồi những tài nguyên nào?
  - **Trở lại trạng thái trước tắc nghẽn (rollback)**: khi thu hồi tài nguyên, cần phục hồi trạng thái của tiến trình về trạng thái gần nhất khi chưa bị tắc nghẽn.
  - **Tình trạng <đói tài nguyên>**: cần bảo đảm không có tiến trình nào luôn bị thu hồi tài nguyên.