LẬP TRÌNH ĐỒNG THỜI & PHÂN TÁN

BÀI 1: NHỮNG KIẾN THỨC CƠ SỞ

Giảng viên: Lê Nguyễn Tuấn Thành
Email: thanhlnt@tlu.edu.vn

NỘI DUNG

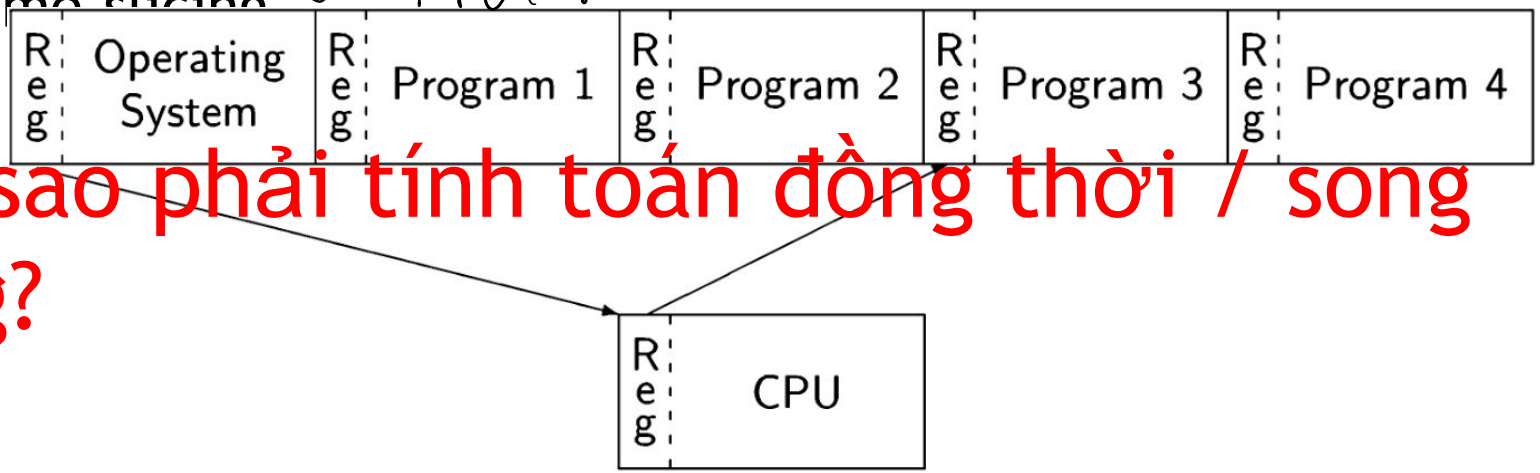
1. Thuật ngữ
2. Luồng trong Java

Bài giảng có sử dụng hình vẽ trong cuốn sách "***Concurrent and Distributed Computing in Java***, *Vijay K. Garg, University of Texas, John Wiley & Sons, 2005*"

Phần 1. Thuật ngữ

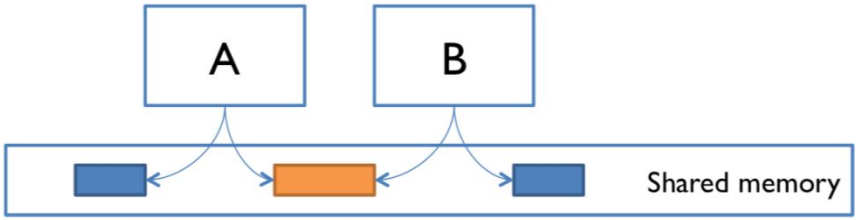
Thuật ngữ (1)

- **Tính toán tuần tự** (sequential computing)
  - Tại một thời điểm chỉ thực hiện được một tính toán
  - Chỉ có một luồng điều khiển chính
- **Hệ thống đơn nhiệm** (single-tasking systems)
- **Hệ thống đa nhiệm** (multitasking systems)
  - Time slicing



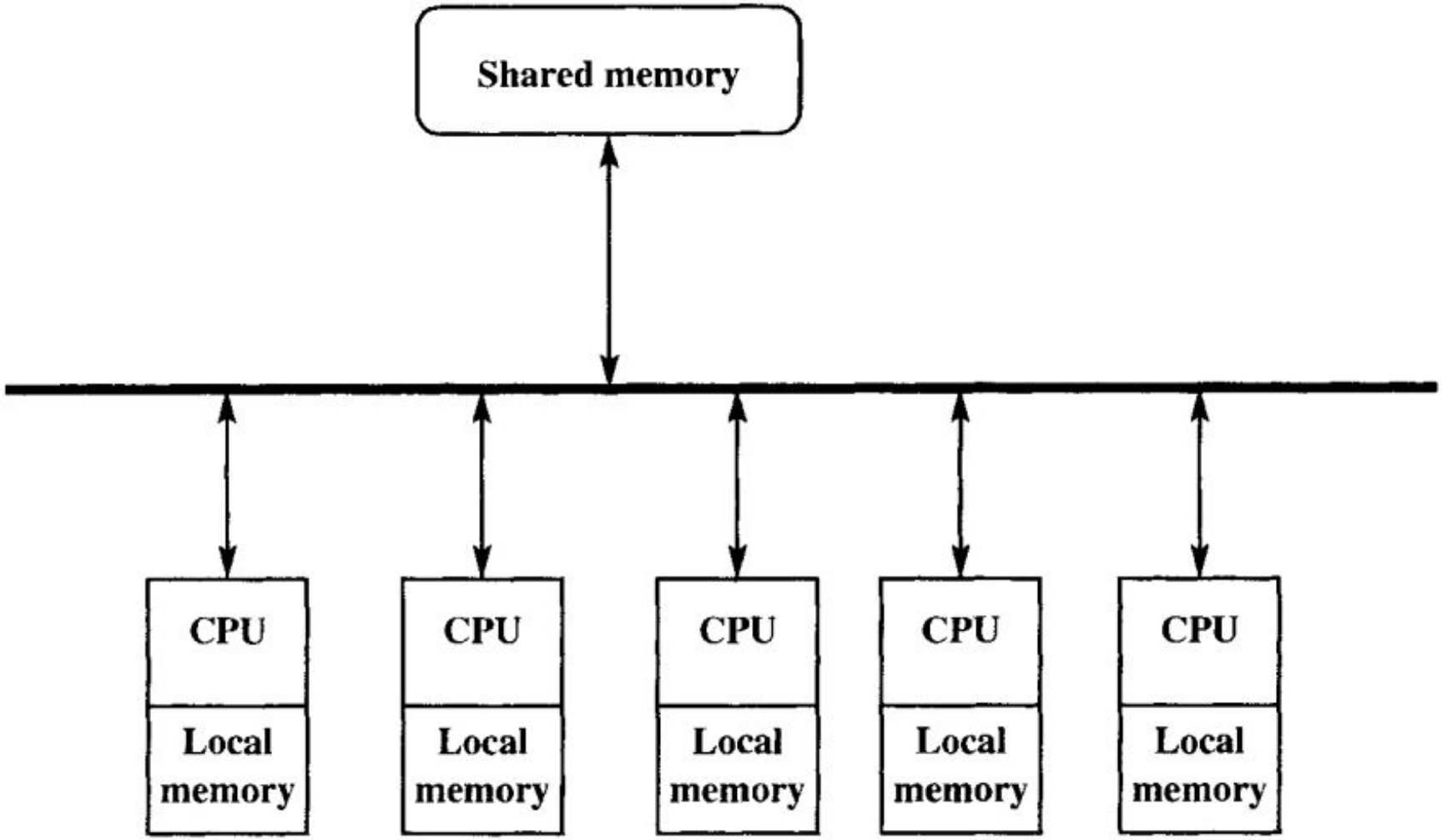
Tại sao phải tính toán đồng thời / song song?

Thuật ngữ (2)



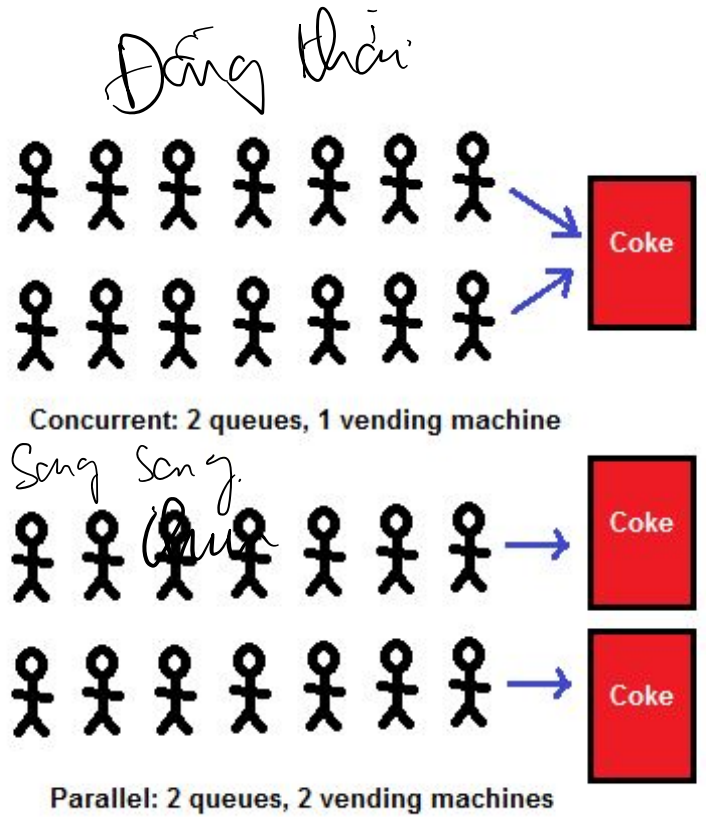
- **Tính toán đồng thời / song song** (concurrent / parallel computing): Mô hình chia sẻ bộ nhớ
  - Tại một thời điểm có thể thực hiện nhiều tính toán
  - Bao gồm nhiều "*chương trình*" chạy trên một hoặc nhiều bộ vi xử lý
  - Giao tiếp với nhau bằng cách sử dụng *bộ nhớ chia sẻ*
  - Một "*chương trình*" bất kỳ luôn biết được trạng thái toàn cục của toàn bộ hệ thống

Minh họa:
Hệ thống song song

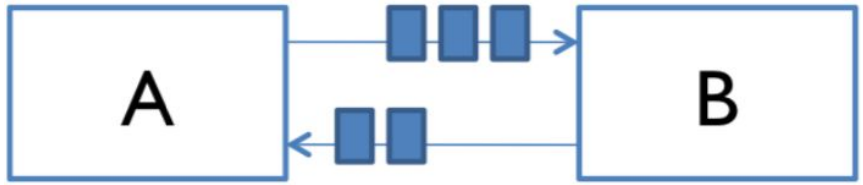


Giả sử: 1 người ≈ 1 Processor

- **Multitasking:**
  - 1 bạn: vừa làm bài tập (LT+TH) môn CSE423, vừa nghe nhạc
- **Concurrency:**
  - 1 bạn: vừa đọc phần lý thuyết, vừa code phần thực hành
- **Parallelism:**
  - 2 bạn: 1 bạn đọc phần lý thuyết, 1 bạn code phần thực hành

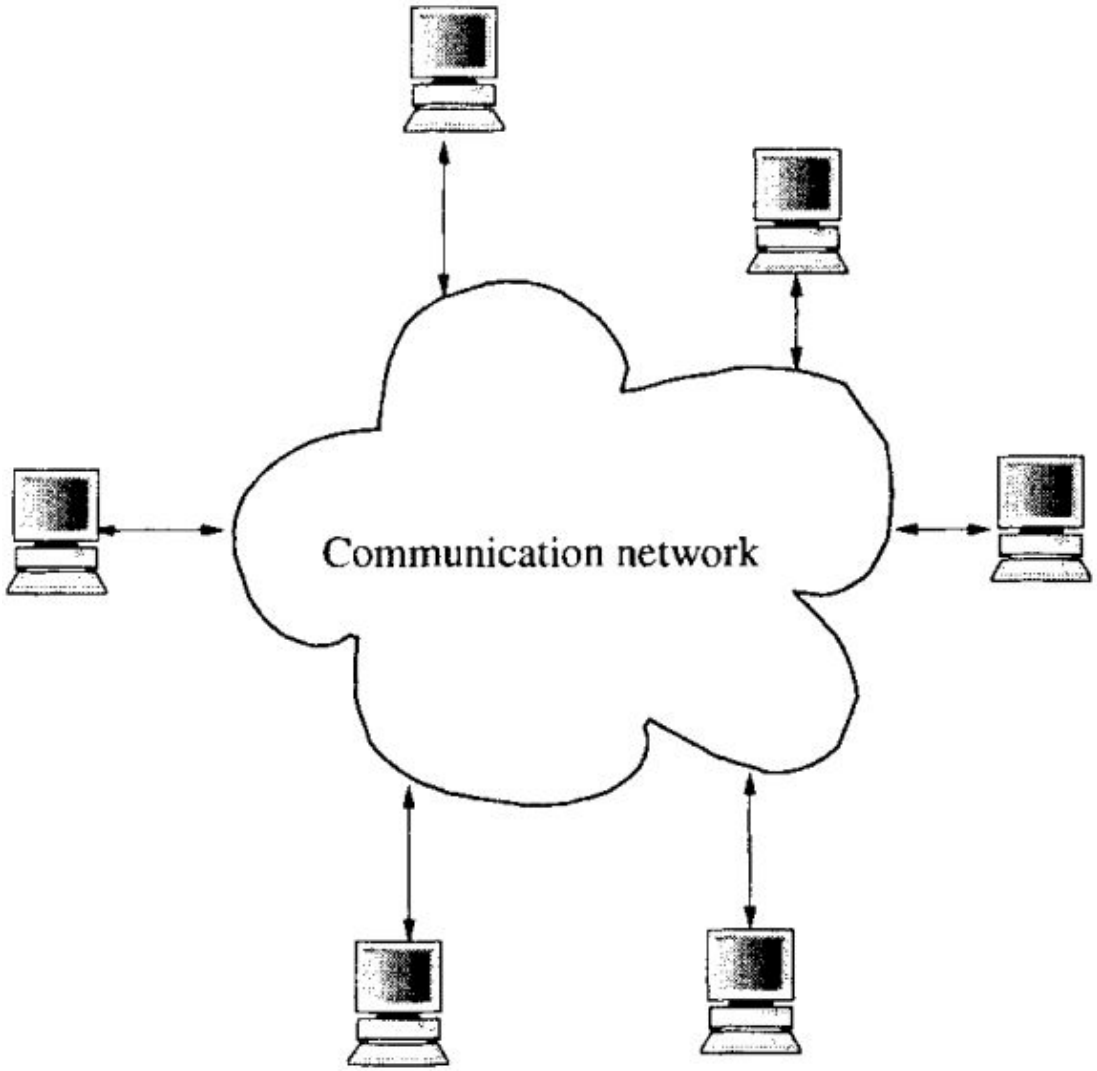


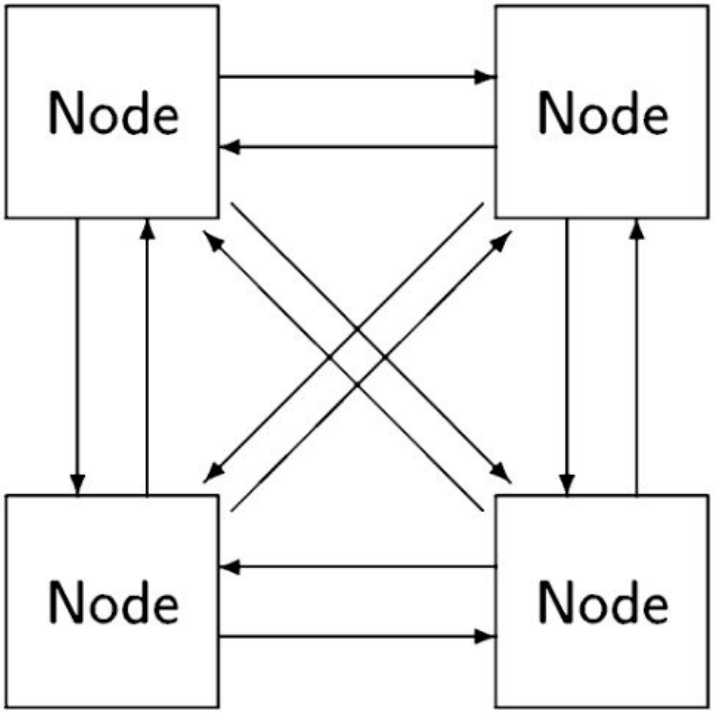
Thuật ngữ (3)

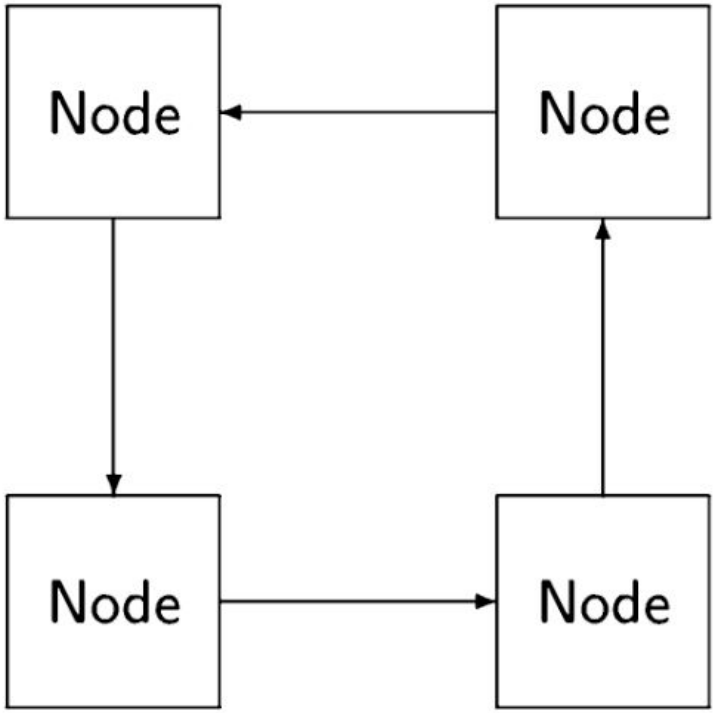


- **Tính toán phân tán** (distributed computing)
  - Hệ thống phân tán chứa nhiều bộ xử lý được kết nối với nhau bởi một mạng truyền thông
  - Các bộ vi xử lý *giao tiếp với nhau bằng cách gửi và nhận các thông điệp*, thông qua các kênh truyền thông (pipe, socket)
  - Không có bộ xử lý nào biết được trạng thái toàn cục của toàn bộ hệ thống phân tán

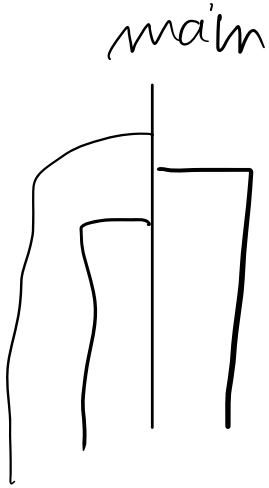
Minh họa:
Hệ thống phân tán




Node
Node
Node
Node


Node
Node
Node
Node

Thuật ngữ (4)



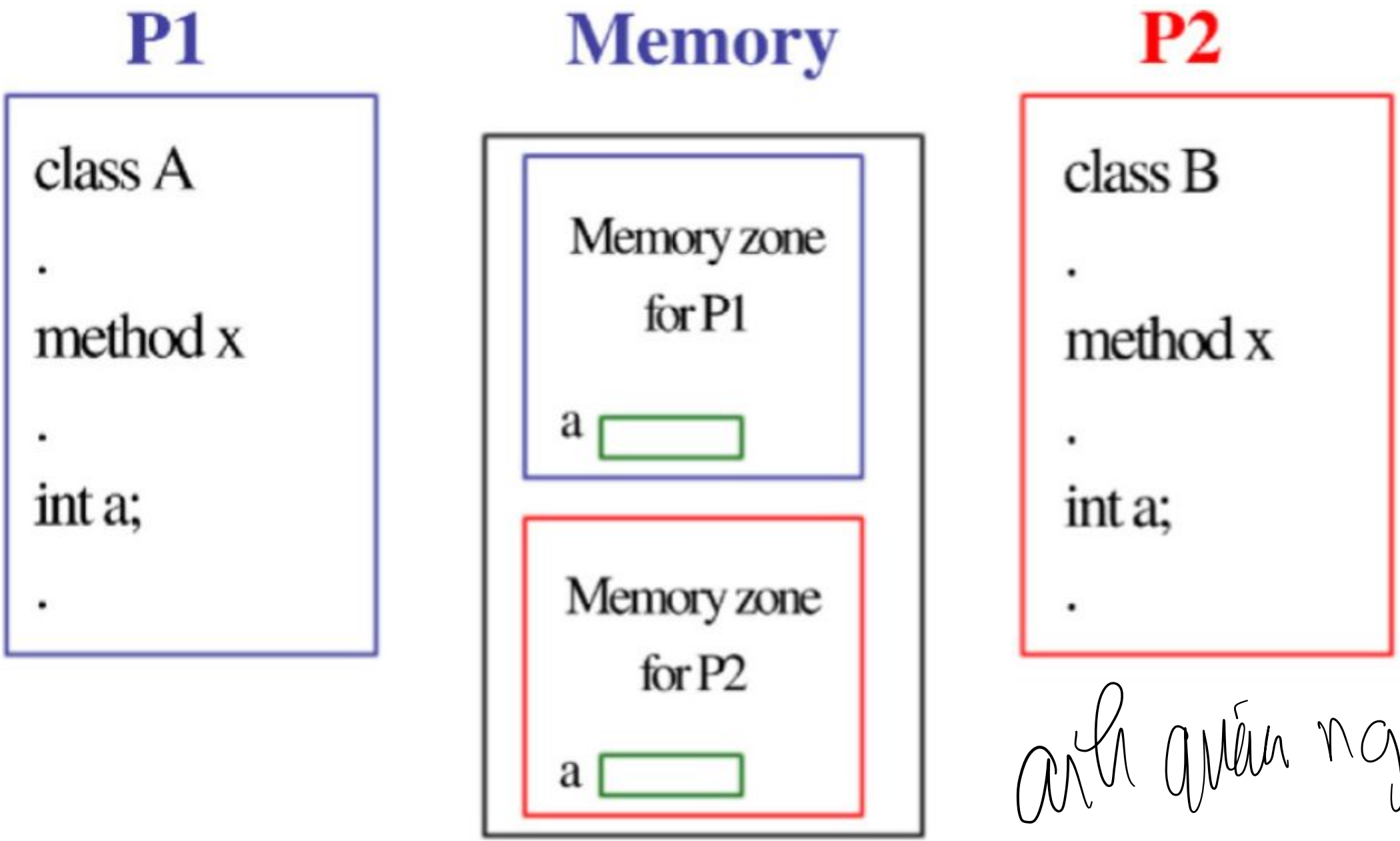
Chương trình (program): một tập các chỉ lệnh bằng ngôn ngữ lập trình

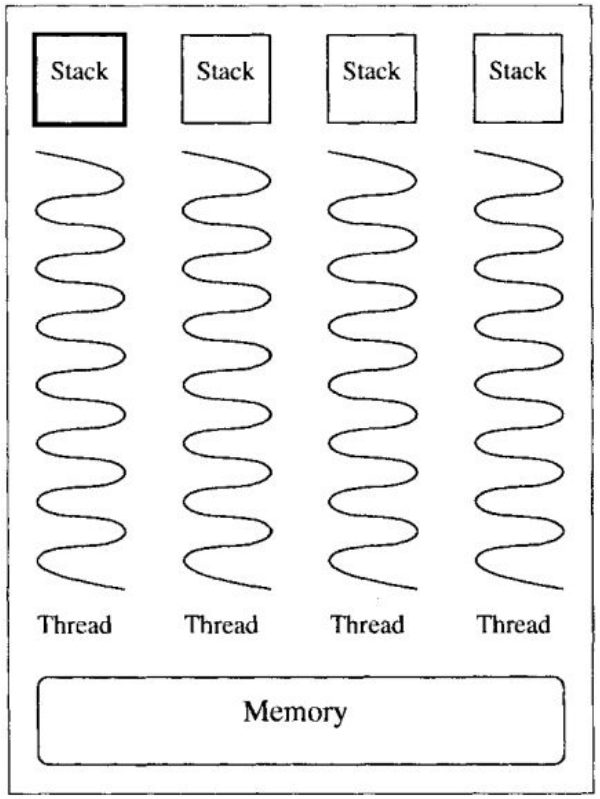
- *Chương trình tuần tự*: thực hiện trong một "*tiến trình*" duy nhất
- *Chương trình đồng thời*: nhiều "*tiến trình*"

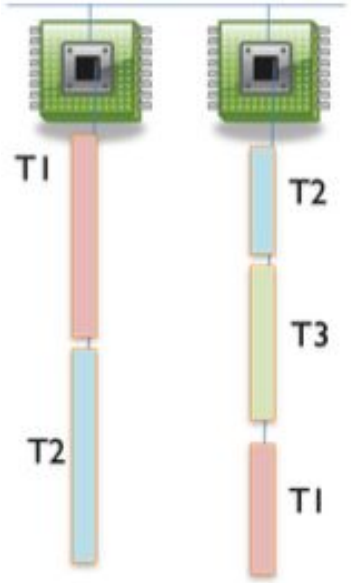
Tiến trình (process): một instance của một chương trình đang chạy, có không gian bộ nhớ riêng, gồm:

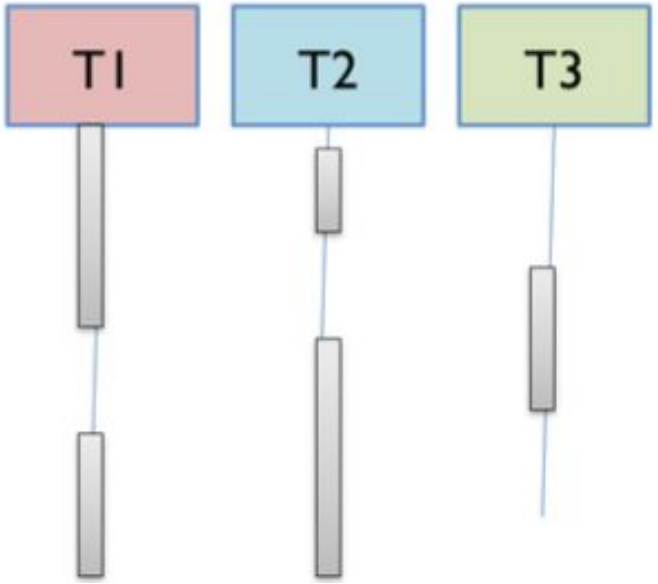
- *Mã* chương trình: những chỉ lệnh máy trong bộ nhớ mà tiến trình thực thi
- *Dữ liệu* gồm bộ nhớ được sử dụng bởi các biến toàn cục tĩnh và bộ nhớ được cấp phát trong thời gian chạy
- *Ngăn xếp* gồm các biến địa phương và các bản ghi kích hoạt lời gọi hàm

Luồng (*threads*): một tiến trình gồm một hay nhiều luồng.
Các luồng trong cùng một tiến trình chia sẻ tài nguyên (bộ nhớ, files,...)
Luồng "*gọn nhẹ*" hơn so với tiến trình và tốn ít phụ phí hơn để tạo và huỷ luồng so với khởi động một tiến trình mới.


P1
Memory
P2
class A
.
method x
.
int a;
.
Memory zone
for P1
a
Memory zone
for P2
a
class B
.
method x
.
int a;
.







anh quân nguyễn

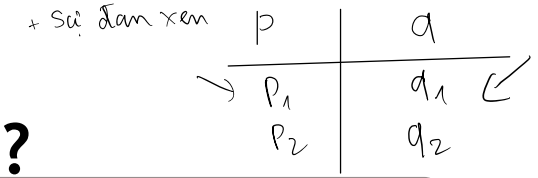
Minh hoạt luồng

Thách thức của các chương trình đồng thời

Làm sao để đồng bộ việc thực thi của các tiến trình/luồng khác nhau và cho phép chúng giao tiếp với nhau ?

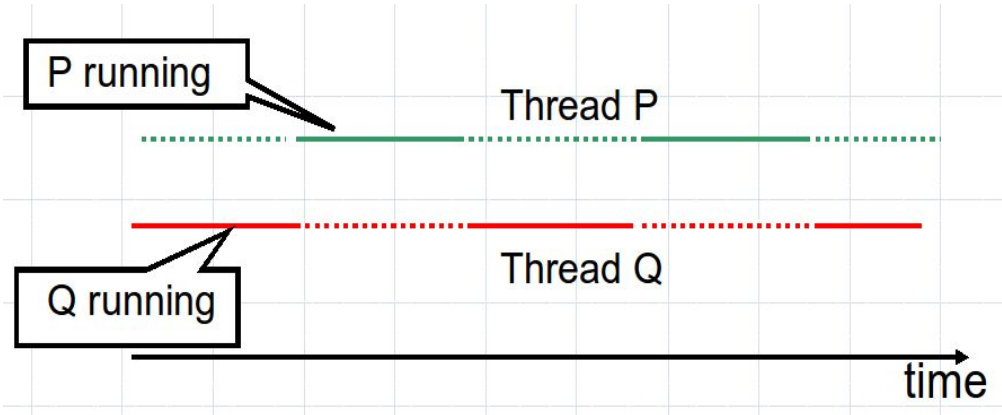
Interleaving

- Giả sử chương trình có 2 luồng:
  1. Luồng **P** bao gồm 2 câu lệnh p_1, được theo sau bởi p_2
  2. Luồng **Q** bao gồm 2 câu lệnh q_1, được theo sau bởi q_2
- Hai luồng bắt đầu thực thi tại vị trí của con trỏ điều kiển (control pointer), lúc đầu trỏ tới p_1 và q_1
- Giả sử các câu lệnh không thực hiện việc chuyển điều khiển khi đang thực thi



- **Các kịch bản có thể xảy ra ???**
  1. $p_1 \rightarrow q_1 \rightarrow p_2 \rightarrow q_2$
  2. $p_1 \rightarrow q_1 \rightarrow q_2 \rightarrow p_2$
  3. $p_1 \rightarrow p_2 \rightarrow q_1 \rightarrow q_2$
  4. $q_1 \rightarrow p_1 \rightarrow q_2 \rightarrow p_2$
  5. $q_1 \rightarrow p_1 \rightarrow p_2 \rightarrow q_2$
  6. $q_1 \rightarrow q_2 \rightarrow p_1 \rightarrow p_2$

Interleaving



- **$p_2 \rightarrow p_1 \rightarrow q_1 \rightarrow q_2$ có phải là một kịch bản không?**
  - KHÔNG !
  - Tôn trọng *sự thực thi tuần tự của mỗi tiến trình*
  - Do đó p2 không thể thực thi trước p1 !

Race condition

Trạng thái đua tranh

Algorithm 2.1: Trivial concurrent program	
integer n ← 0 (biến chia sẻ)	
p	**q**
integer k1 ← 1 (khai báo biến cục bộ)	integer k2 ← 2
p1: n ← k1 (gán)	q1: n ← k2

chạy sinh ra 6 kịch bản

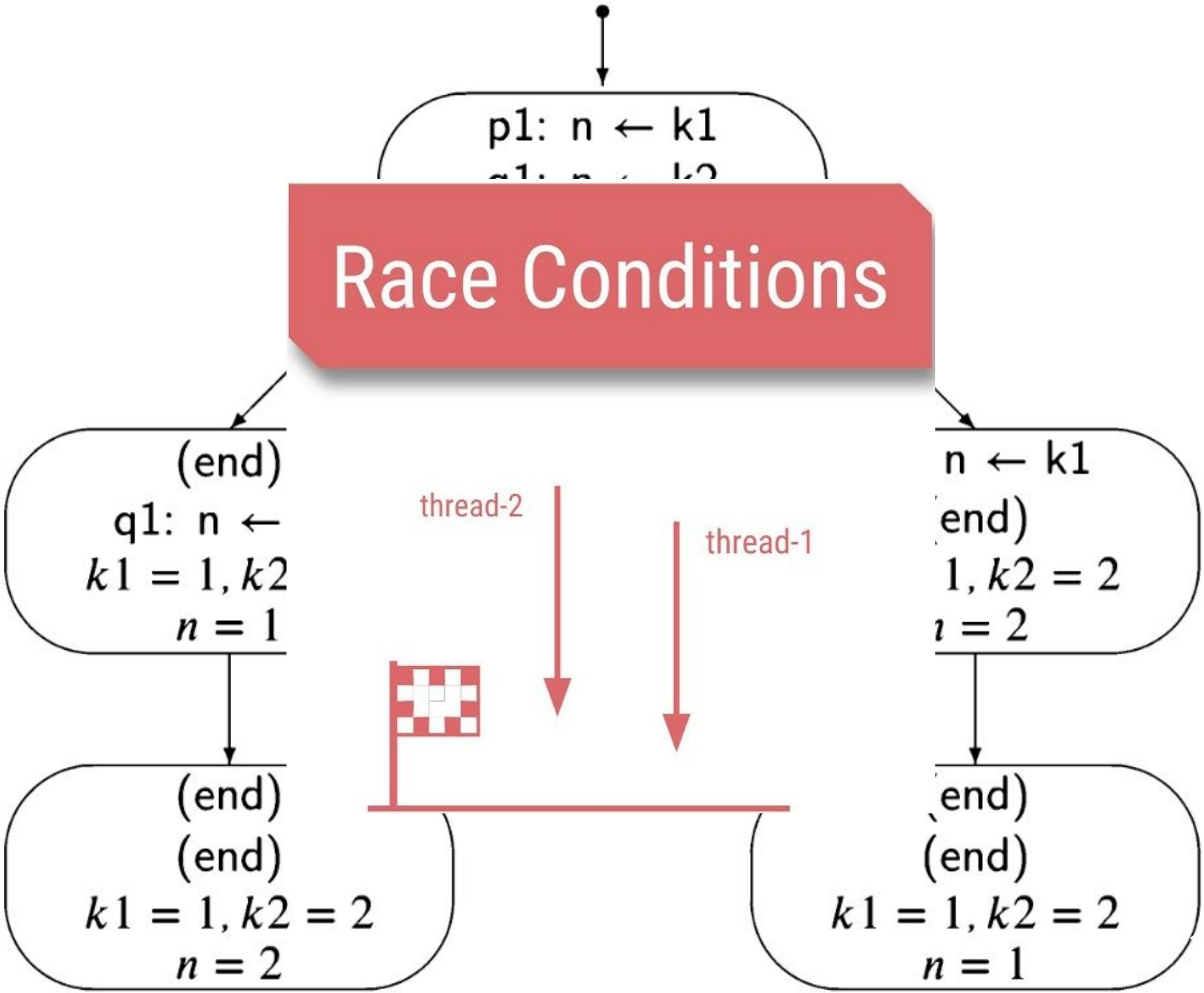
nếu k1 thực hiện cuối cùng n = 1

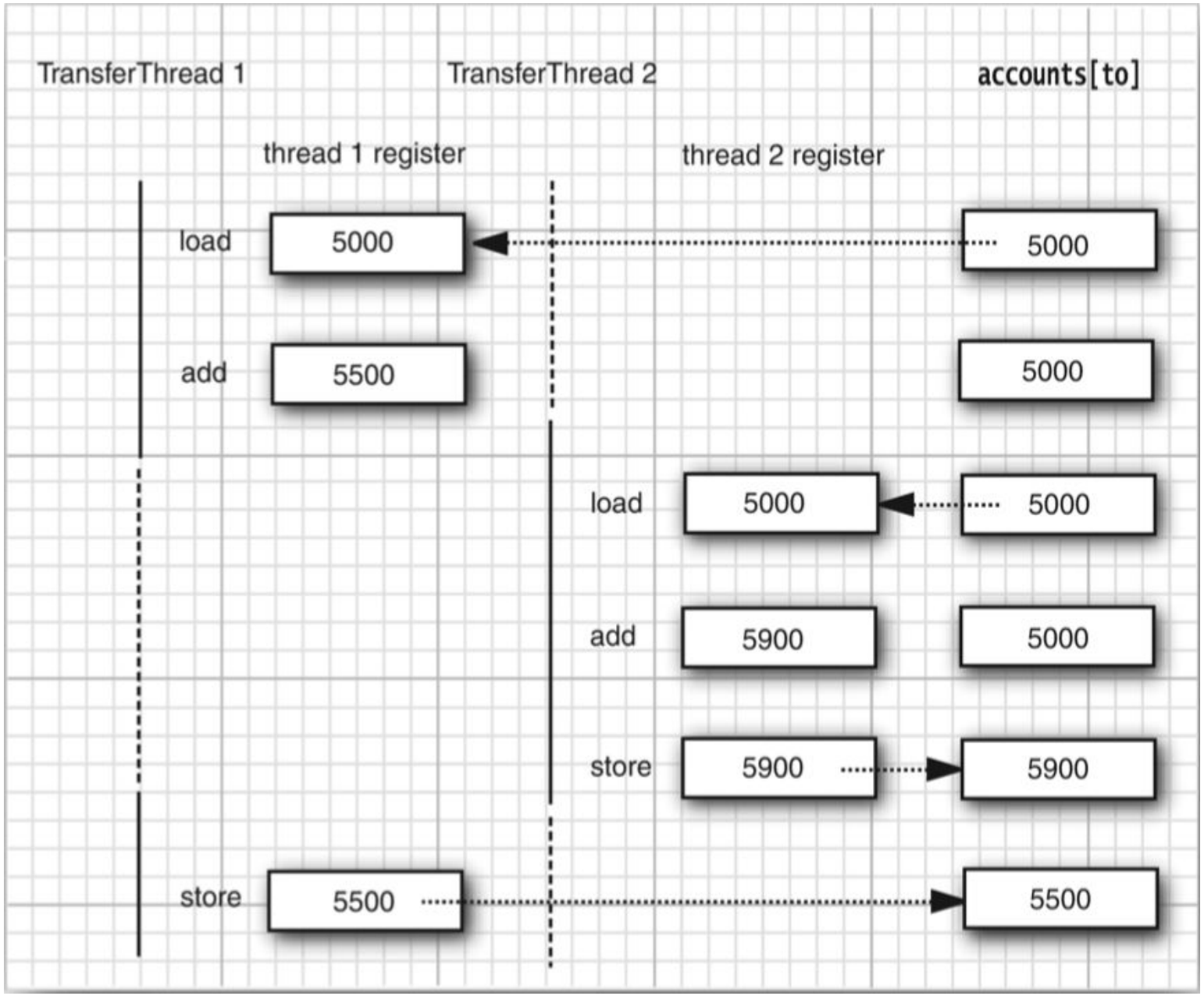
Nếu k2 thực hiện cuối n = 2

Giá trị của n là bao nhiêu khi p, q thực thi xong ?

= 1 hoặc 2 tùy vào kịch bản

Kết thúc = 0


p1: n ← k1
Race Conditions
(end)
q1: n ←
k1 = 1, k2
n = 1
n ← k1
end)
1, k2 = 2
= 2
thread-2
thread-1
(end)
(end)
k1 = 1, k2 = 2
n = 2
end)
(end)
k1 = 1, k2 = 2
n = 1



Có hai cơ chế để bảo vệ một khối mã lệnh khỏi việc truy cập đồng thời

- Từ khoá *synchronized*
- Lớp *ReentrantLock* (từ Java SE 5.0)

Concurrency is Hard to Test and Debug (1)

- It's very hard to discover race conditions using testing
  - Each time you run a program containing a race condition, you may get different behavior !
- Interleaving of instructions or messages depends on the relative timing of events that are strongly influenced by the environment
- Delays can be caused by other running programs, other network traffic, operating system scheduling decisions, variations in processor clock speed, etc.

Concurrency is Hard to Test and Debug (2)

- Two kinds of bugs:
  1. *heisenbugs*, which are nondeterministic and hard to reproduce,
  2. *bohrbug*, which shows up repeatedly whenever you look at it.
- Almost all bugs in sequential programming are *bohrbugs*
- A *heisenbug* may even disappear when you try to look at it with *println* or *debugger* !
  - The reason is that printing and debugging are so much slower than other operations, often 100-1000x slower, that they dramatically change the timing of operations, and the interleaving.

Phần 2. Luồng trong Java

```
public class HelloWorldThread extends Thread {
    public void run() {
        System.out.println("Hello World");
    }
    public static void main(String[] args) {
        HelloWorldThread t = new HelloWorldThread();
        t.start();
    }
}
```

Tạo luồng bằng cách kế thừa lớp Thread

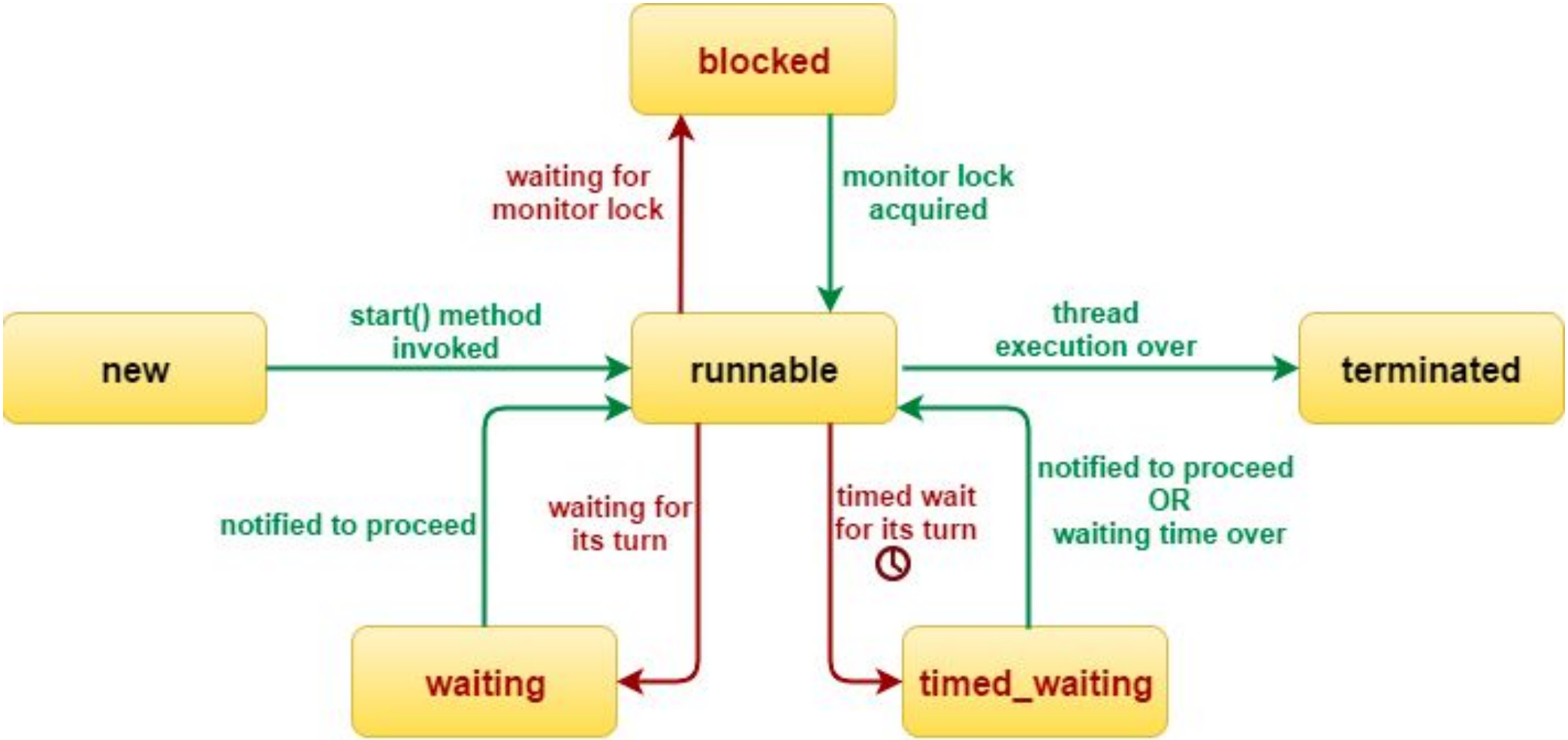
```
public class MyRunnable implements Runnable {

        public void run() {
                System.out.println("Executing thread "+Thread.currentThread().
                    getName());
        }

        public static void main(String[] args) throws InterruptedException {
                Thread myThread = new Thread(new MyRunnable(), "myRunnable");
                myThread.start();
        }
}
```

Tạo luồng bằng cách cài đặt giao diện Runnable

Các trạng thái của luồng trong Java



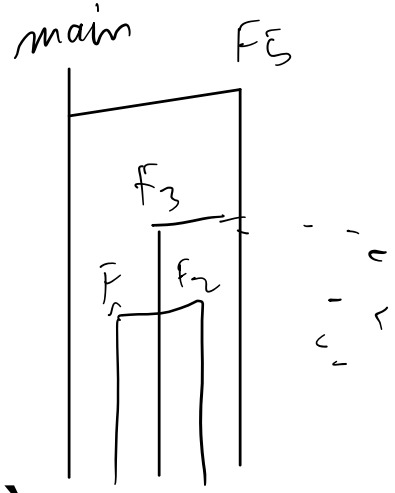
Cơ chế Join (1)

- Cho phép một luồng đợi một luồng khác hoàn thành việc thực thi
- Ví dụ:
  - Viết chương trình sử dụng luồng trong Java để tính số Fibonacci thứ n (F_n) sử dụng công thức: $F_n = F_{n-1} + F_{n-2}$ với $n \geq 2$
  - Các trường hợp cơ sở: $F_0 = 1$, $F_1 = 1$

Cơ chế Join (2)

```
public class Fibonacci extends Thread {
    int n;
    int result;
    public Fibonacci(int n) {
        this.n = n;
    }
    public void run() {
        if ((n == 0)||(n == 1)) result = 1;
        else {
            Fibonacci f1 = new Fibonacci(n-1);
            Fibonacci f2 = new Fibonacci(n-2);
            f1.start();
            f2.start();
            try {
                f1.join();
                f2.join();
            } catch (InterruptedException e){};
            result = f1.getResult() + f2.getResult();
        }
    }
    public int getResult(){
        return result;
    }
    public static void main(String[] args) {
        Fibonacci f1 = new Fibonacci(Integer.parseInt(args[0]));
        f1.start();
        try {
            f1.join();
        } catch (InterruptedException e){};
        System.out.println("Answer is " + f1.getResult());
    }
}
```

Lập lịch trình luồng



- Nếu cả hai luồng đều có thể chạy, luồng nào sẽ được chọn để chạy bởi hệ thống?
  - Phụ thuộc vào độ ưu tiên và chính sách lập lịch của hệ thống
  - Thay đổi độ ưu tiên của luồng sử dụng *setPriority* và lấy ra độ ưu tiên hiện tại sử dụng *getPriority*
  - MIN_PRIORITY (1), MAX_PRIORITY (10), NORM_PRIORITY (5): 3 hằng số nguyên được định nghĩa trong lớp Thread
- Daemon thread: luồng chạy ngầm

Tài liệu tham khảo

- *Concurrent and Distributed Computing in Java*, Vijay K. Garg, University of Texas, John Wiley & Sons, 2005
- Tham khảo:
  - *Principles of Concurrent and Distributed Programming*, M. Ben-Ari, Second edition, 2006
  - *Foundations of Multithreaded, Parallel, and Distributed Programming*, Gregory R. Andrews, University of Arizona, Addison-Wesley, 2000
  - *The SR Programming Language: Concurrency in Practice*, Benjamin/Cummings, 1993
  - *Xử lý song song và phân tán*, Đoàn văn Ban, Nguyễn Mậu Hân, Nhà xuất bản Khoa học và Kỹ thuật, 2009

$x = 0$

$y = 0$

T_1	T_2
$x = 1$	$y = 2$
return y	return 3*x

Kb1: 1 $x = 1$, $y = 2$ return 3* x = 3 return y = 2

2 $x = 1$, $y = 2$ return $y = 2$ return 3* x = 3

3 $x = 1$ return $y = 0$ ($y = 0$ return 3* x = 3

4 $y = 2$ $x = 1$ return 3* x = 3 return $y = 2$

5

6