LẬP TRÌNH ĐỒNG THỜI & PHÂN TÁN

BÀI 2:
BÀI TOÁN LOẠI TRỪ LẪN NHAU

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

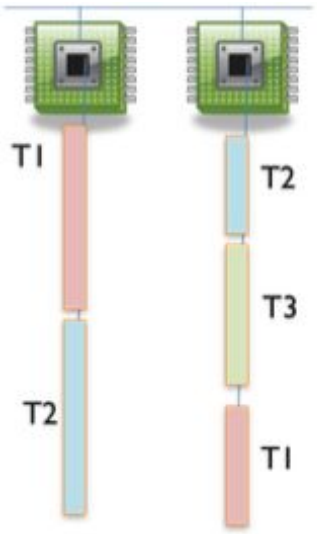
NỘI DUNG

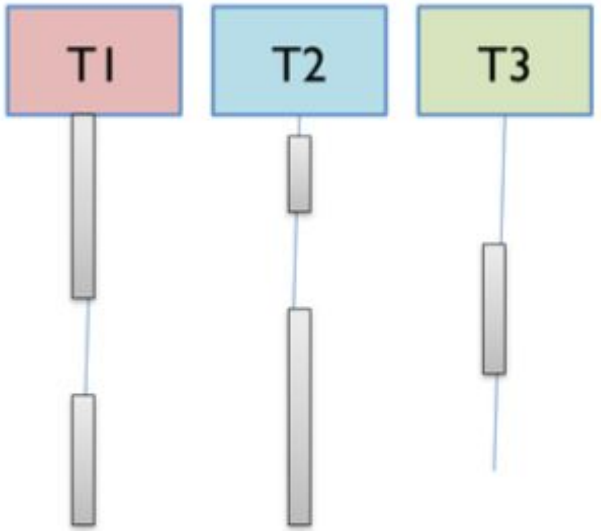
1. Bài toán loại trừ lẫn nhau trong những hệ thống chia sẻ bộ nhớ
2. Giải pháp cho bài toán loại trừ lẫn nhau

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*"

Thách thức trong các chương trình đồng thời

- Đồng bộ sự thực thi của các luồng khác nhau
- Cho phép các luồng giao tiếp với nhau thông qua bộ nhớ chia sẻ





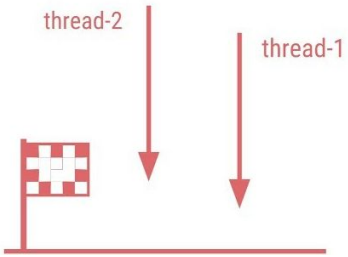
4

Phần 1. Bài toán loại trừ lẫn nhau

Mutual Exclusion Problem - Mutex

“Lost update” Problem (1)

- Nguyên nhân: Race condition



- Xét tình huống:
  - Có một biến chia sẻ *x* với giá trị ban đầu là *0*
  - Có hai luồng T_0 và T_1 đều tăng giá trị của *x* lên 1
  - **Liệu giá trị của *x* sau khi thực thi T_0 và T_1 sẽ là 2?**

"Lost update" Problem (2)

Luồng T_0	*Luồng T_1*
Đọc giá trị x vào một thanh ghi (giá trị được đọc: 0)	
Tăng thanh ghi (1)	
Ghi giá trị trong thanh ghi ngược lại x (x=1)	
	Đọc giá trị x vào một thanh ghi (giá trị được đọc: 1)
	Tăng thanh ghi (2)
	Ghi giá trị trong thanh ghi ngược lại x (x=2)

"Lost update" Problem (3)

Luồng T_0	*Luồng T_1*
Đọc giá trị x vào một thanh ghi (giá trị được đọc: 0)	
Tăng thanh ghi (1)	
	Đọc giá trị x vào một thanh ghi (giá trị được đọc: 0)
	Tăng thanh ghi (1)
Ghi giá trị trong thanh ghi ngược lại x (x=1)	
	Ghi giá trị trong thanh ghi ngược lại x (x=1)

Làm sao để tránh vấn đề mất mát dữ liệu?

- Câu lệnh *x = x +1* phải được thực thi *một cách nguyên tử* ***(atomically)***
- Mở rộng ra, nếu một phần mã cần được thi thực một cách nguyên tử thì phần mã đó được gọi là: *khu vực quan trọng* (*Critical Region - CR)* hay *phần quan trọng (Critical Section - CS)*
- Cho ví dụ về CS ???

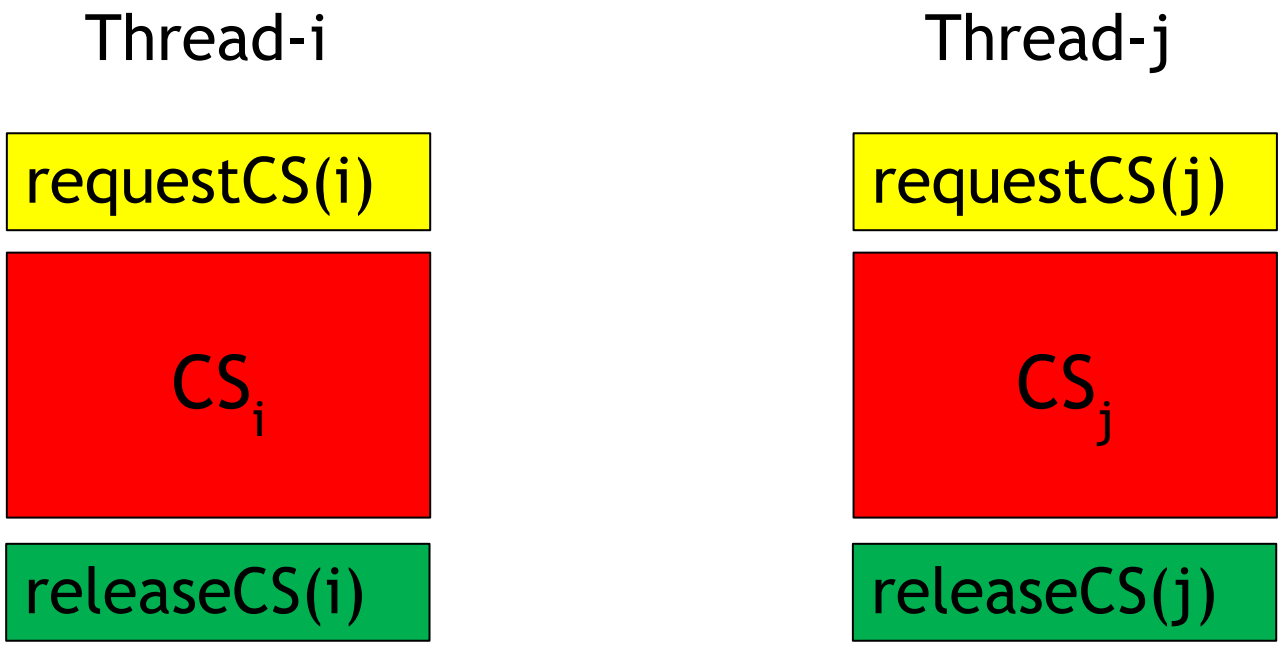
Bài toán loại trừ lẫn nhau (Mutex)

- Là bài toán nhằm đảm bảo rằng *khu vực quan trọng* (*CR*/*CS*) của một luồng phải được thực thi theo một cách nguyên tử
- Là một trong những bài toán căn bản nhất trong tính toán đồng thời

Giao diện cho Bài toán Mutex

- Định nghĩa giao diện Lock để đồng bộ việc truy cập *khu vực quan trọng* (CR/CS) của các luồng

```
public interface Lock {
    public void requestCS(int pid); //may block
    public void releaseCS(int pid);
}
```


Thread-i
Thread-j
requestCS(i)
requestCS(j)
CS_i
CS_j
releaseCS(i)
releaseCS(j)

Phần 2. Giải pháp cho Bài toán Mutex

Busy-waiting solutions within a loop

Trường hợp 2 luồng

Thuật toán 1

- Sử dụng một biến chia sẻ giữa 2 luồng, *openDoor* kiểu boolean được khởi tạo là *true*
- **requestCS**: luồng đợi cho đến khi biến *openDoor* có giá trị *true*
  - Khi giá trị của biến này là *true*, luồng có thể đi vào CS, sau đó nó đặt lại giá trị của *openDoor* thành *false*
- **releaseCS**: luồng đặt lại giá trị của biến *openDoor* là *true*

```
class Attempt1 implements Lock {
    boolean openDoor = true;
    public void requestCS(int i) {
        while (!openDoor) ; // busy wait
        openDoor = false;
    }
    public void releaseCS(int i) {
        openDoor = true;
    }
}
```

Đánh giá thuật toán 1

- Cài đặt này không hoạt động đúng do việc kiểm tra *openDoor* và việc đặt lại giá trị biến này thành *false* không được làm *một cách nguyên tử*
- Tưởng tượng rằng một luồng có thể kiểm tra biến *openDoor* và vượt qua câu lệnh *while*
- Tuy nhiên, trước khi luồng đó có thể đặt biến *openDoor* thành *false*, luồng thứ 2 bắt đầu thực hiện
- Luồng thứ 2 lúc này kiểm tra giá trị của *openDoor* và cũng vượt qua câu lệnh *while* để đi vào CS !
- Cả hai luồng bây giờ đều có thể đặt *openDoor* thành *false* và cùng đi vào *CS*
- **Do đó, cài đặt 1 vi phạm sự loại trừ lẫn nhau !**

Thuật toán 2: Dẫn đến Deadlock

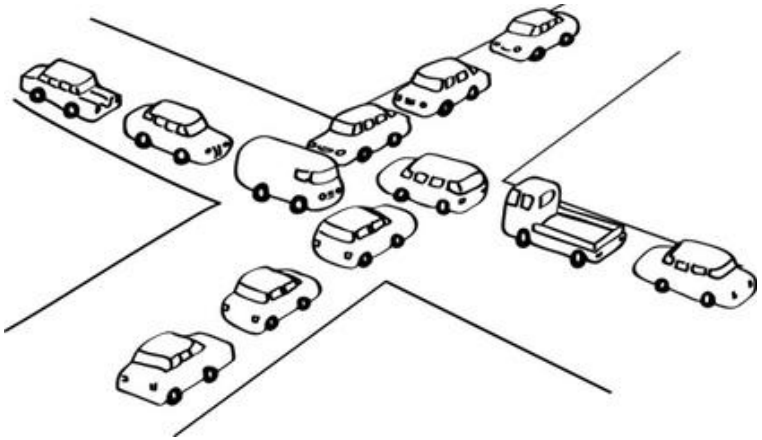
- Trong cài đặt 1, biến chia sẻ *openDoor* không lưu lại luồng nào đã cập nhật nó thành *false*
- Cài đặt 2 giải quyết vấn đề này bằng cách sử dụng 2 biến chia sẻ *wantCS [0]* và *wantCS[1]*

```
 1 class Attempt2 implements Lock {
 2     boolean wantCS[] = { false, false };
 3     public void requestCS(int i) { // entry protocol
 4         wantCS[i] = true;     //declare intent
 5         while (wantCS[1 - i]) ; // busy wait
 6     }
 7     public void releaseCS(int i) {
 8         wantCS[i] = false;
 9     }
10 }
```

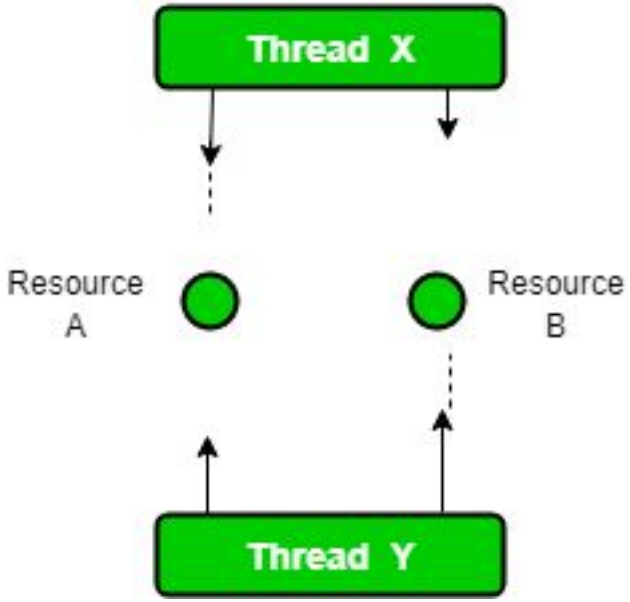
Đánh giá thuật toán 2

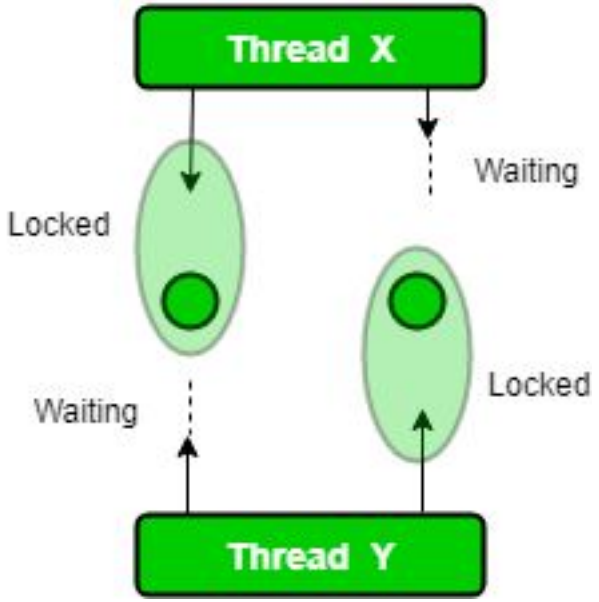
- Cài đặt này cũng không làm việc đúng!
- Cả hai luồng có thể CÙNG LÚC đặt bit *wantCS* của nó thành *true* và rơi vào vòng lặp đợi vô hạn do đều chờ luồng kia đặt bit *wantCS* thành *false*!
- DEADLOCK !

Deadlock



- Thread 1: locks resource A, waits for resource B
- Thread 2: locks resource B, waits for resource A





Thuật toán 3: Luân phiên chặt chẽ

- Cài đặt 3 cũng khắc phục được vấn đề lưu vết luồng đã thực hiện sự thay đổi trên biến chia sẻ, xảy ra ở cài đặt 1
- Cài đặt này được dựa trên việc kiểm tra giá trị của biến chia sẻ *turn*
- Một luồng sẽ đợi đến lượt nó để đi vào *CS*. Khi thoát ra khỏi CS, nó đặt lại giá trị biến *turn* thành $1-i$

```
class Attempt3 implements Lock {
    int turn = 0;
    public void requestCS(int i) {
        while (turn == 1 - i) ;
    }
    public void releaseCS(int i) {
        turn = 1 - i;
    }
}
```

Đánh giá thuật toán 3

- Cài đặt 3 bảo đảm *sự loại trừ lẫn nhau* !
- Cài đặt này cũng đảm bảo rằng nếu cả hai luồng đang cố gắng để đi vào CS, thì một trong số hai luồng sẽ đi vào CS thành công
- Tuy nhiên, cài đặt này phát sinh một vấn đề khác
  - Cả 2 luồng phải luân phiên nhau để đi vào CS !
  - Do đó, sau khi luồng T_0 thoát khỏi CS, T_0 không thể đi vào CS nữa cho đến khi luồng T_1 đi vào CS và thay đổi lại giá trị của biến *turn* !
  - Nếu T_1 không quan tâm tới việc đi vào CS, thì T_0 sẽ bị tắc lại vô hạn do phải đợi P_1 !

Thuật toán Peterson (1)

- Kết hợp 2 cách tiếp cận trước để giải quyết bài toán mutex trong một hệ thống có 2 luồng hoạt động đồng thời
- Trong thuật toán này, chúng ta lưu giữ 2 cờ/bit, *wantCS[0]* và *wantCS[1]*, như cài đặt 2, và một biến *turn* như trong cài đặt 3

```
 1 class PetersonAlgorithm implements Lock {
 2     boolean wantCS[] = { false, false };
 3     int turn = 1;
 4     public void requestCS(int i) {
 5         int j = 1 - i;
 6         wantCS[i] = true;
 7         turn = j;
 8         while (wantCS[j] && (turn == j)) ;
 9     }
10     public void releaseCS(int i) {
11         wantCS[i] = false;
12     }
13 }
```

Đánh giá thuật toán Peterson

1. ***Loại trừ lẫn nhau*** *(mutual exclusion)*
   - Cả hai luồng không thể ở trong CS tại cùng một thời điểm
2. ***Tiến độ*** *(progress)*
   - Nếu một hoặc hai luồng đang cố gắng đi vào CS và không có luồng nào bên trong CS, thì ít nhất một luồng sẽ đi vào CS thành công
3. ***Không chết đói*** *(starvation-freedom)*
   - Nếu một luồng đang cố gắng đi vào CS, thì cuối cùng nó phải được đi vào CS

Trường hợp N luồng (N>2)

Thuật toán Bakery của Lamport (1)

- Ý tưởng tương tự như cách các tiệm bánh phục vụ khách hàng
- Mỗi khách hàng khi đến tiệm bánh được phát cho *một số hiệu duy nhất*
- Tại một thời điểm, tiệm bánh sẽ phục vụ khách hàng đang giữ *số nhỏ nhất hiện tại*

Thuật toán Bakery của Lamport (2)

- Một luồng T_i phải đi qua 2 bước chính trước khi có thể đi vào CS

1. **Bước 1**: được gọi là *doorway*
   - T_i được yêu cầu chọn một số
   - T_i đọc số của tất cả những luồng khác và chọn ra một số lớn hơn số lớn nhất mà nó đọc được
2. **Bước 2**: kiểm tra 2 điều kiện để đi vào CS
   - Với mỗi luồng T_j khác, T_i kiểm tra liệu T_j có đang ở trong *doorway* không. Nếu T_j đang ở trong *doorway*, thì T_i phải đợi cho T_j ra khỏi *doorway*
   - T_i phải đợi cho *number[j]* bằng *0* hoặc ***(number[i], i) < (number[j], j)***

```
1  class Bakery implements Lock {
2      int N;
3      boolean[] choosing; // inside doorway
4      int[] number;
5      public Bakery(int numProc) {
6          N = numProc;
7          choosing = new boolean[N];
8          number = new int[N];
9          for (int j = 0; j < N; j++) {
10             choosing[j] = false;
11             number[j] = 0;
12         }
13     }
14     public void requestCS(int i) {
15         // step 1: doorway: choose a number
16         choosing[i] = true;
17         for (int j = 0; j < N; j++)
18             if (number[j] > number[i])
19                 number[i] = number[j];
20         number[i]++;
21         choosing[i] = false;
22
23         // step 2: check if my number is the smallest
24         for (int j = 0; j < N; j++) {
25             while (choosing[j]) ; // process j in doorway
26             while ((number[j] != 0) &&
27                    ((number[j] < number[i]) ||
28                    ((number[j] == number[i]) && j < i)))
29                 ; // busy wait
30         }
31     }
32     public void releaseCS(int i) { // exit protocol
33         number[i] = 0;
34     }
35 }
```

Tại sao phải kiểm tra:
number[j] == number[i]
&& j < i ?

Đánh giá thuật toán Bakery

- Thuật toán thỏa mãn *sự loại trừ lẫn nhau*
- Thuật toán cũng thỏa mãn điều kiện *không chết đói*
  - Do bất kỳ luồng nào đang đợi để đi vào CS thì cuối cùng nó cũng sẽ có giữ số nhỏ nhất khác 0 tại một thời điểm nào đó
  - Khi đó, luồng này sẽ đi vào CS thành công !

Đánh giá thuật toán Bakery: Nhược điểm

- Thuật toán luôn đòi hỏi thời gian *O(N)* cho mỗi luồng khi muốn lấy được khóa (lock) mặc dù có thể không có tranh chấp
- Thuật toán đòi hỏi mỗi luồng sử dụng dấu thời gian (timestamps), i.e. số id, với giá trị không bị giới hạn

Lớp ReentrantLock

```
myLock.lock(); // Một đối tượng ReentrantLock
try
{
    <khu vực quan trọng (critical section)>
}
finally
{
    myLock.unlock(); // Đảm bảo lock được khoá lại ngay cả
                     khi một ngoại lệ được  ném ra
}
```

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