LẬP TRÌNH ĐỒNG THỜI & PHÂN TÁN

BÀI 3:

NHỮNG CẤU TRÚC ĐỒNG BỘ

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

NỘI DUNG

1. Vấn đề bận-chờ
2. Semaphore
3. Monitor

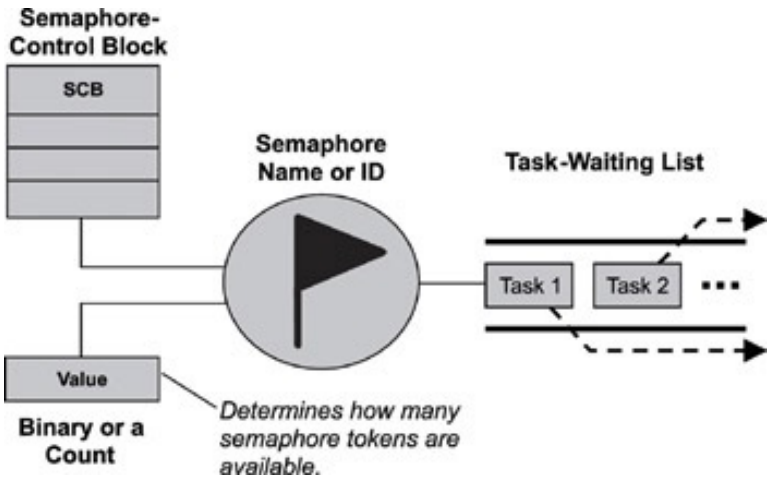
Bài giảng có sử dụng hình vẽ trong cuốn sách "***Concurrent and Distributed Computing in Java***, *Vijay K. Garg, University of Texas, John Wiley & Sons, 2005*"

Vấn đề bận-chờ

- Những giải pháp ở bài trước gặp một vấn đề chung: *bận chờ* (*busy-wait*) khi sử dụng vòng lặp while
- Thay vì phải kiểm tra liên tục điều kiện vào CS, nếu một luồng chỉ kiểm tra khi điều kiện này trở thành true thì sẽ không lãng phí chu trình CPU

Các cấu trúc đồng bộ hoá

- Những cơ sở đồng bộ hóa giúp giải quyết vấn đề bận chờ
- Hai cấu trúc đồng bộ phổ biến:
  - ***Semaphore*** do Dijkstra đề xuất, năm 1968
  - ***Monitor*** được phát minh bởi P. B. Hansen và C. A. R. Hoare, năm 1972



5

Phần 2. Semaphore

Semaphores were invented by Edsger Dijkstra, 1968

Semaphore nhị phân (1)

- Một biến *value* kiểu *boolean*
- Một *hàng đợi* các tiến trình bị khóa
- Hai thao tác nguyên tử: P() và V()

P():

```
if (value == false) {
    Thêm bản thân luồng
    vào hàng đợi và khóa lại;
}
value = false;
```

Được thực thi nguyên tử

V():

```
value = true;
if (hàng đợi không rỗng) {
        Đánh thức một luồng
        bất kỳ trong hàng đợi;
}
```

Được thực thi nguyên tử

Semaphore nhị phân (2) Ví dụ cài đặt


```
1  public class BinarySemaphore {
2      boolean value;
3      BinarySemaphore(boolean initValue) {
4          value = initValue;
5      }
6      public synchronized void P() {
7          if (value == false)
8              Util.myWait(this); // in queue of blocked processes
9          value = false;
10     }
11     public synchronized void V() {
12         value = true;
13         notify();
14     }
15 }
```

Phương thức *myWait*() sẽ khóa luồng hiện tại và chèn nó vào trong hàng đợi các luồng bị khóa

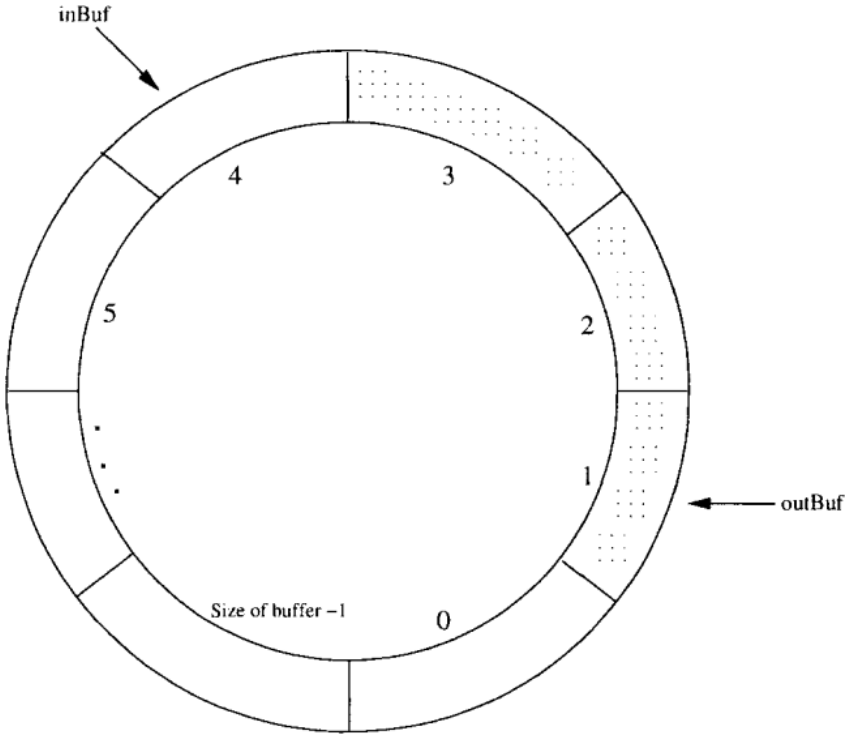
Semaphore đếm
Ví dụ cài đặt

```
public class CountingSemaphore {
    int value;
    public CountingSemaphore(int initValue) {
        value = initValue;
    }
    public synchronized void P() {
        value--;
        if (value < 0) Util.myWait(this);
    }
    public synchronized void V() {
        value++;
        if (value <= 0) notify();
    }
}
```

Sử dụng Semaphore cho một số bài toán đồng bộ

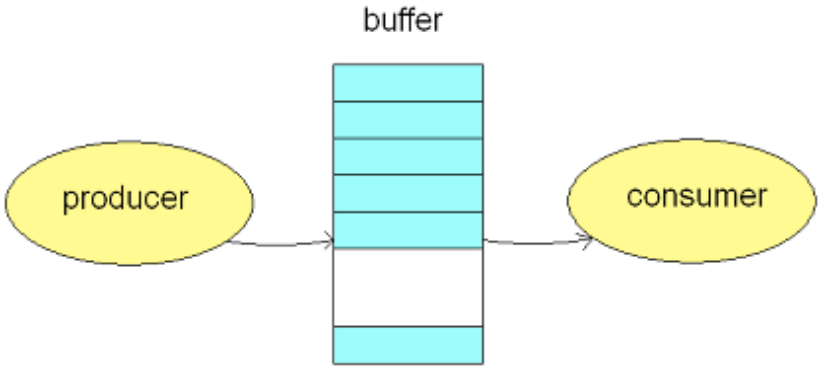
Bài toán 1: Nhà sản xuất & Người tiêu thụ (1)

- Hai luồng:
  1. Luồng 1: *Producer*
  2. Luồng 2: *Consumer*
- Bộ đệm chia sẻ là một mảng vòng tròn có kích thước *size*, gồm:
  - Hai con trỏ *inBuf* và *outBuf*
  - Biến *count* để lưu tổng số phần tử hiện tại



Bài toán 1: Nhà sản xuất & Người tiêu thụ (2)

- Ngoài việc đảm bảo loại trừ lẫn nhau, bài toán này có thêm 2 rằng buộc đồng bộ có điều kiện:
  1. Luồng sản xuất chỉ thực hiện thêm 1 phần tử vào cuối bộ đệm nếu: *bộ đệm không đầy*
  2. Luồng tiêu thụ chỉ thực hiện lấy 1 phần tử khỏi của bộ đệm nếu: *bộ đệm không rỗng*
- Bộ đệm sẽ đầy nếu *producer* thêm phần tử với tốc độ lớn hơn tốc độ lấy phần tử của *consumer*
- Bộ đệm sẽ rỗng nếu … ?



Bài toán 1: Sản xuất & Tiêu thụ (3)

BinarySemaphore **mutex**(true);
CountingSemaphore **isFull**(size), **isEmpty**(0)

Producer	Consumer

void **deposit**() {
  isFull.P();
  mutex.P();
  <Ghi dữ liệu vào bộ đệm>
  mutex.V();
  isEmpty.V();
}
	void **fetch**() {
	  isEmpty.P();
	  mutex.P();
	  <Lấy dữ liệu ra khỏi bộ đệm>
	  mutex.V();
	  isFull.V();
	}

Bài toán 2: Người đọc & Người ghi

- Phối hợp truy cập tới một cơ sở dữ liệu chia sẻ giữa nhiều người đọc và nhiều người ghi
- Các rằng buộc đồng bộ:
  1. **Rằng buộc đọc-ghi**
  2. **Rằng buộc ghi-ghi**
  3. **Nhiều người đọc có thể đồng thời truy cập CSDL chia sẻ**

```
class ReaderWriter {
    int numReaders = 0;
    BinarySemaphore mutex = new BinarySemaphore(true);
    BinarySemaphore wlock = new BinarySemaphore(true);
    public void startRead() {
        mutex.P();
        numReaders++;
        if (numReaders == 1) wlock.P();
        mutex.V();
    }
    public void endRead() {
        mutex.P();
        numReaders--;
        if (numReaders == 0) wlock.V();
        mutex.V();
    }
    public void startWrite() {
        wlock.P();
    }
    public void endWrite() {
        wlock.V();
    }
}
```

Luồng đọc - R_i

startRead()

Đọc từ CSDL

endRead()

Luồng ghi - W_j

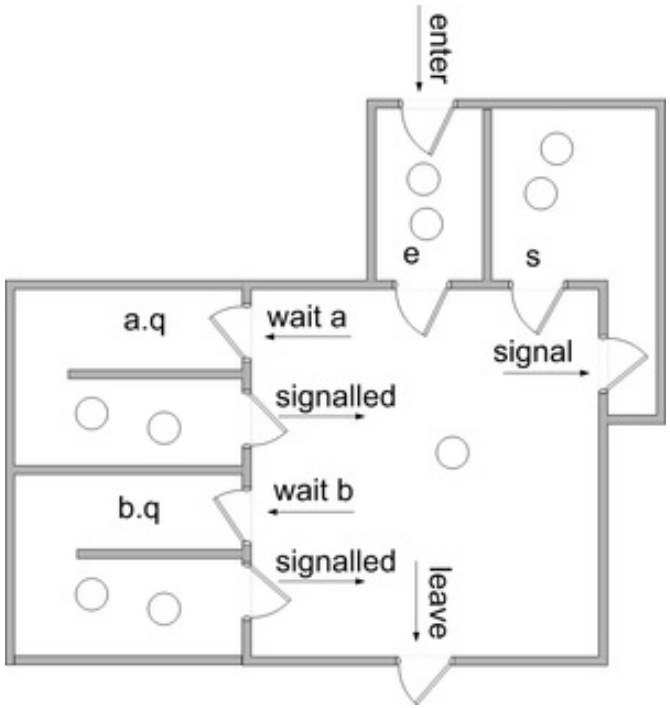
startWrite()

Ghi vào CSDL

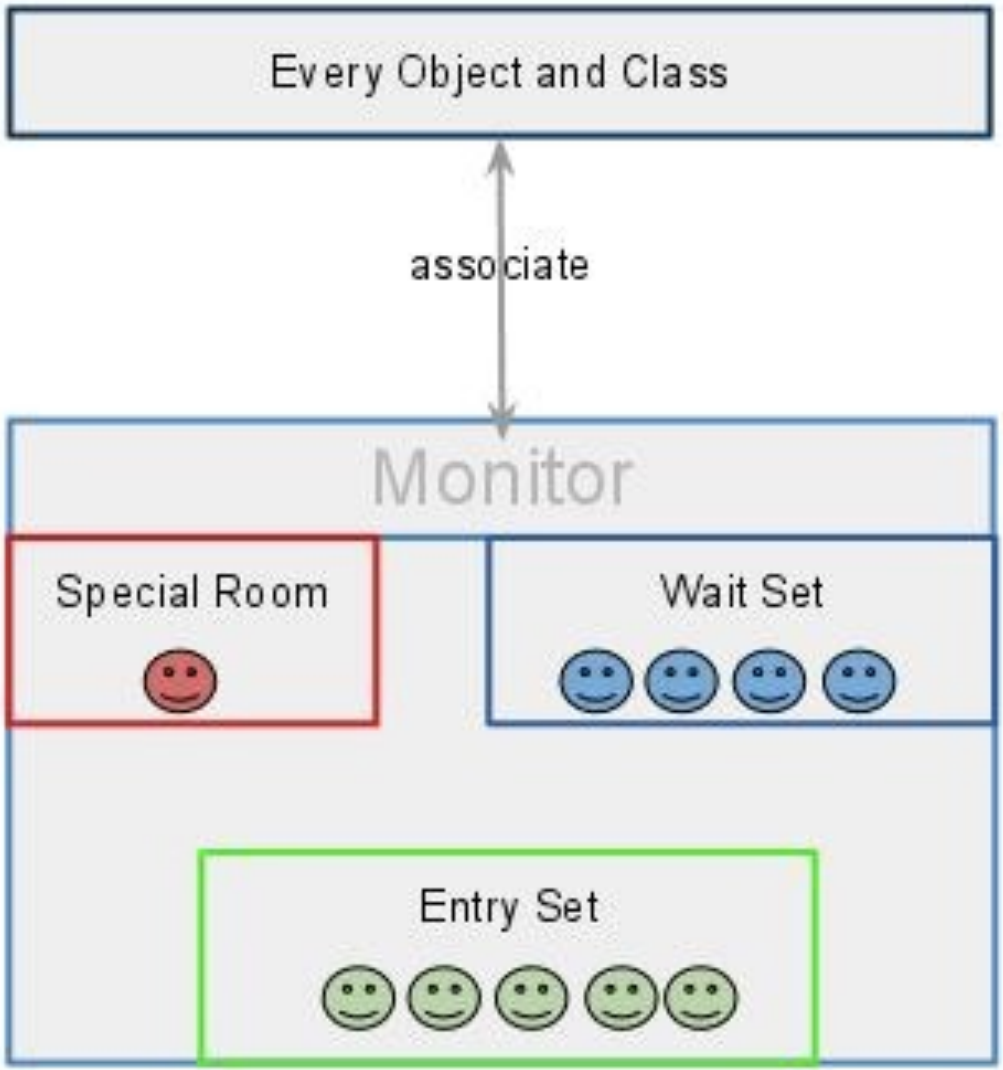
endWrite()

15

Phần 3. Monitor



Invented by P. B. Hansen and C. A. R. Hoare, 1972

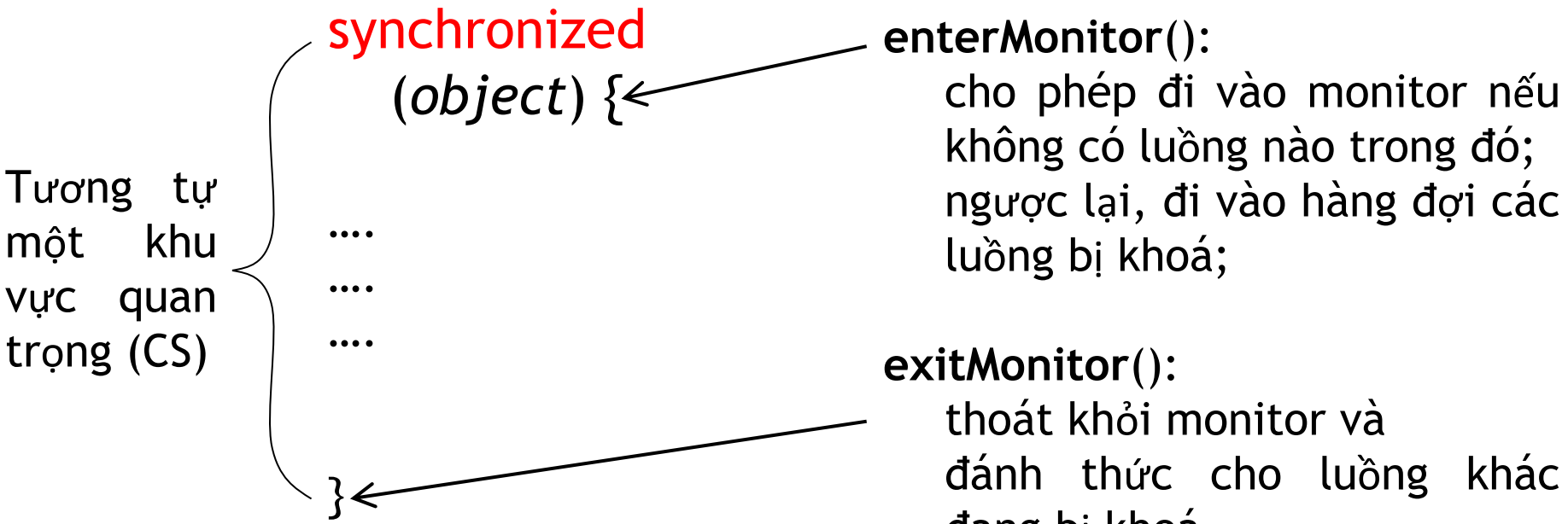

Every Object and Class
associate
Monitor
Special Room
Wait Set
Entry Set

Monitor

- Hỗ trợ khái niệm *biến điều kiện* cho trường hợp yêu cầu sự đồng bộ có điều kiện
- Mỗi biến điều kiện x định nghĩa 2 thao tác:
  - ***wait***: khoá luồng gọi và đưa vào hàng đợi của x
  - ***notify*** hoặc ***signal***: loại một luồng khỏi hàng đợi của x và và chèn nó vào hàng đợi *sẵn-sàng-thực-thi*

Ngữ nghĩa của Monitor (1)

- Trong Java, sử dụng từ khoá synchronized để quy định một đối tượng là một Monitor



Ngữ nghĩa của Monitor (2)

Các phương thức sử dụng bên trong Monitor

```
synchronized (object)
{
    ....
    object.wait();
    ....
    object.notify();
    ....
    object.notifyAll();
    ....
}
```

Dừng thực thi luồng hiện tại, thả khoá, đặt luồng này vào hàng đợi của object và chờ cho đến khi luồng khác đánh thức

Nếu hàng đợi không rỗng, chọn một luồng tùy ý trong hàng đợi và đánh thức nó

Nếu hàng đợi không rỗng, đánh thức tất cả luồng trong hàng đợi

Hai kiểu Monitor

Luồng 0	*Luồng 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();

Luồng nào nên tiếp tục thực hiện vào thời điểm này?	
…. }	…. }

Do chỉ có một luồng có thể được ở bên trong Monitor tại một thời điểm.

Hai khả năng …

Hoare-style Monitor
Blocking condition variables

Luồng 0	*Luồng 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();

…. }	
	…. }

*Luồng 0 sẽ đi vào monitor **ngay lập tức***
sau khi luồng 1 gọi hàm notify()

Mesa-style Monitor
Nonblocking condition variables

Luồng 0	*Luồng 1*

synchronized (object) {
….
object.wait();	

	synchronized (object) {
	….
	object.notify();

	…. }
…. }	

Luồng 1 tiếp tục thực hiện sau khi gọi hàm notify()

Sau khi luồng 1 ra khỏi monitor, luồng 0 có thể đi vào monitor

Sử dụng Monitor cho một số bài toán đồng bộ

Bài toán 1: Nhà sản xuất & Người tiêu thụ (1)

object sharedBuffer;

```
void produce() { // or deposit
    synchronized (sharedBuffer) {
        while (sharedBuffer đầy)
            sharedBuffer.wait();
        Thêm 1 phần tử vào bộ đệm;
        if (bộ đệm không rỗng)
            sharedBuffer.notify();
    }
}
```

```
void consume() { // or fetch
    synchronized (sharedBuffer) {
        while (bộ đệm rỗng)
            sharedBuffer.wait();
        Lấy 1 phần tử  khỏi bộ đệm;
        if (bộ đệm không đầy)
            sharedBuffer.notify();
    }
}
```

Bài toán 2: Người đọc – Người ghi

int numReader, numWriter; Object object;

```
void writeDB() {
    synchronized (object) {
        while (numReader > 0 ||
                numWriter > 0)
            object.wait();
        numWriter = 1;
    }
    // ghi dữ liệu vào DB (không
cần phải ở trong monitor);
    synchronized (object) {
        numWriter = 0;
        object.notifyAll();
    }
}
```

```
void readDB() {
 synchronized (object) {
        while (numWriter > 0)
            object.wait();
        numReader++;
    }
    // đọc dữ liệu từ DB (không
cần phải ở trong monitor);
    synchronized (object) {
        numReader--;
        object.notify();
    }
}
```

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