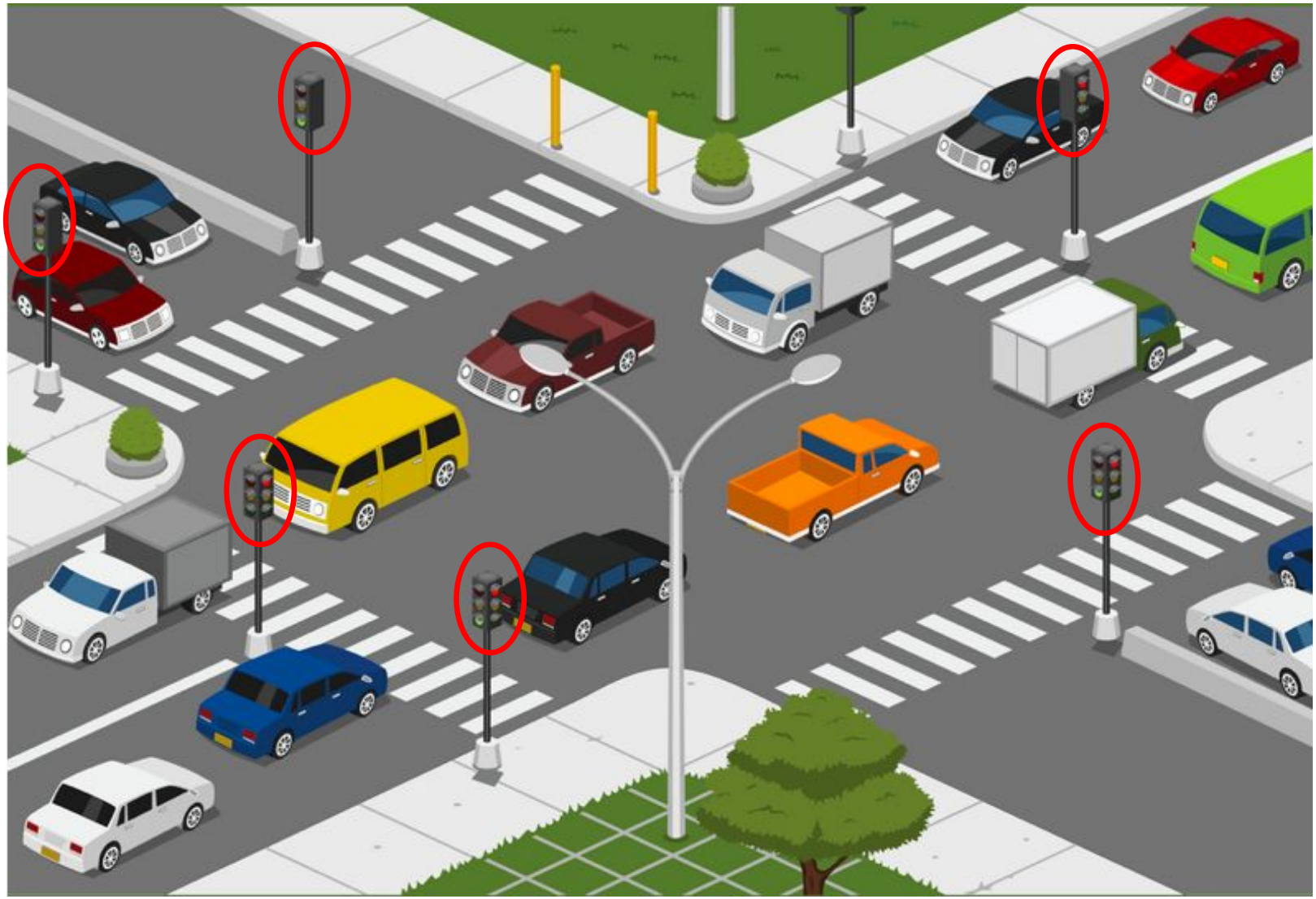
LẬP TRÌNH ĐỒNG THỜI & PHÂN TÁN

BÀI 3: NHỮNG CƠ SỞ ĐỒNG BỘ HOÁ

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

Synchonization primitives



NỘI DUNG

1. Busy-waiting problem
2. Semaphore
3. Monitor

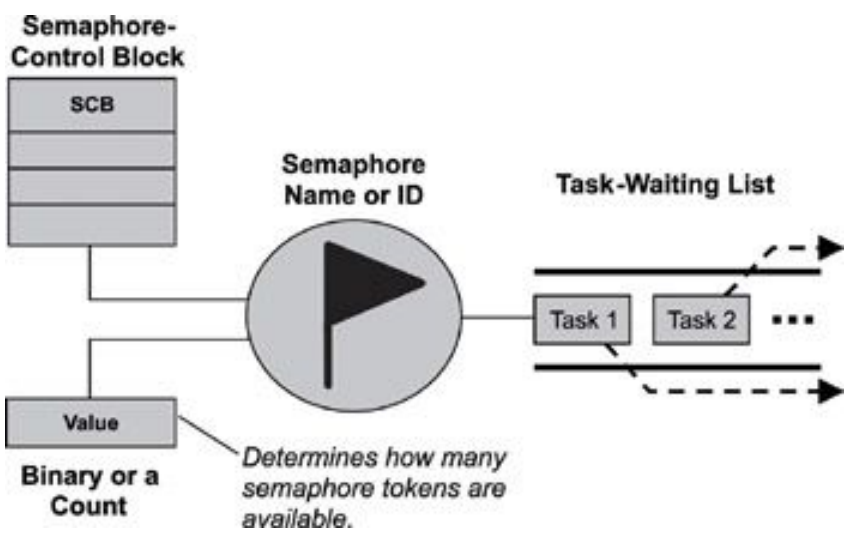
Bài giảng có sử dụng hình vẽ trong cuốn sách "***Concurrent and Distributed Computing in Java***, *Vijay K. Garg, University of Texas, John Wiley & Sons, 2005*"

Busy-waiting problem

- Những giải pháp ở bài trước gặp một vấn đề chung: *bận chờ* (*busy-wait*) khi sử dụng vòng lặp while
  - Khi một luồng không thể đi vào CS, nó sẽ ***liên lục*** kiểm tra điều kiện ở while
  - Điều này khiến luồng không thể thực hiện các công việc khác => gây lãng phí chu trình CPU
- 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

Synchnization primitives

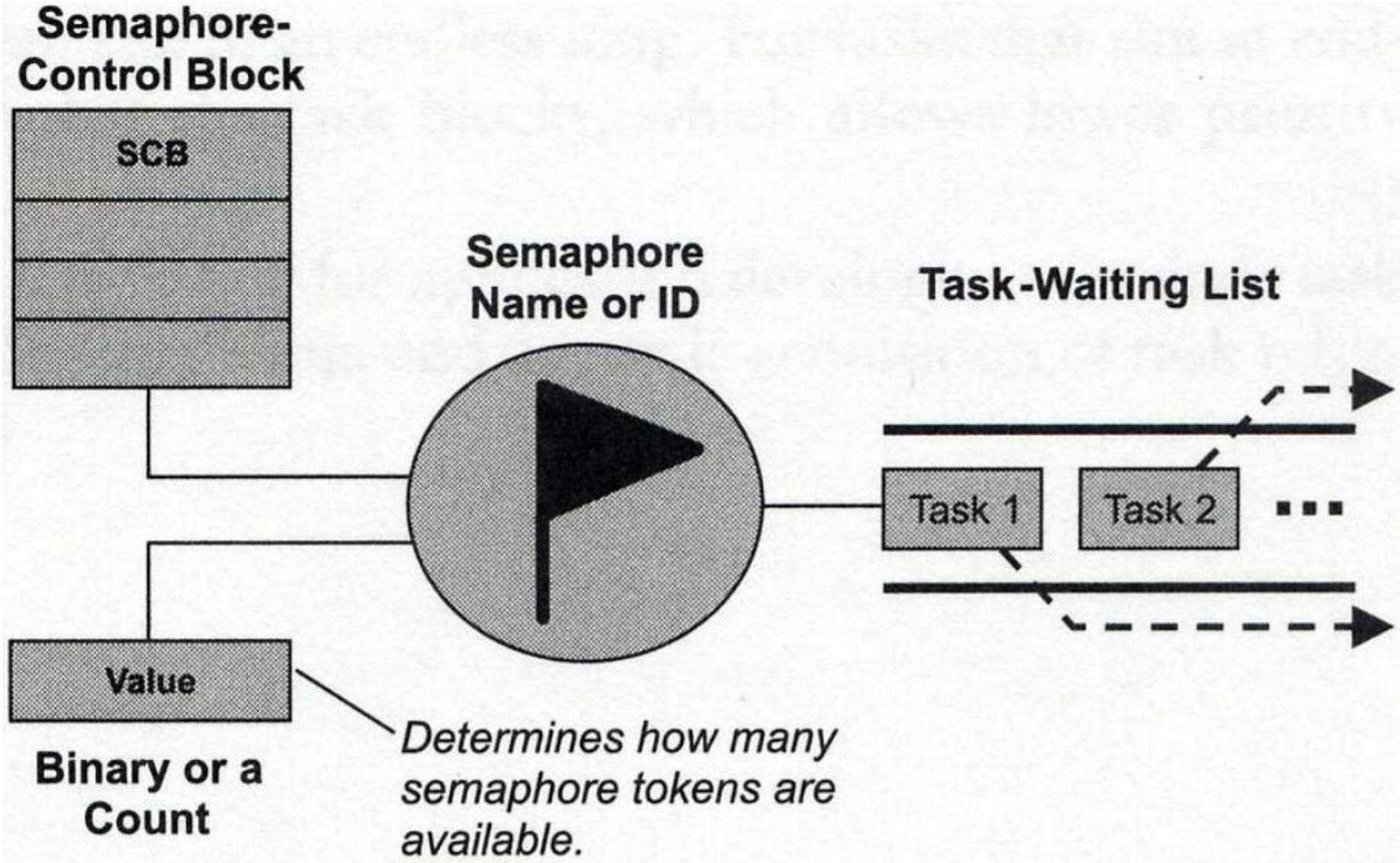
- Những cơ sở đồng bộ hóa giúp giải quyết vấn đề bận chờ
- Hai cấu trúc đồng bộ phổ biến:
  - ***Semaphore*** do Dijkstra đề xuất, năm 1968
  - ***Monitor*** được phát minh bởi P. B. Hansen và C. A. R. Hoare, năm 1972



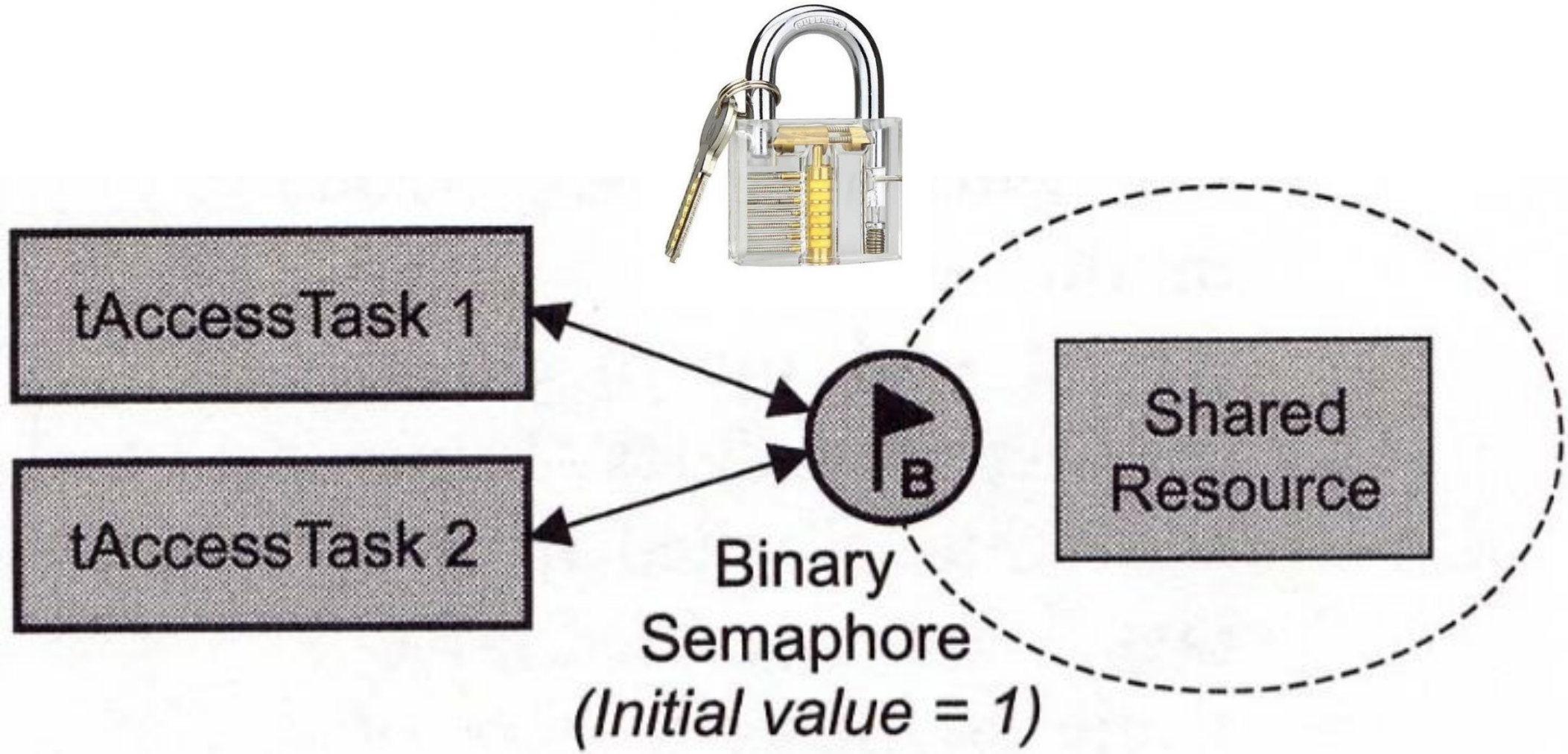
6

Phần 2. Semaphore

Semaphores were invented by Edsger Dijkstra, 1968



Source: https://www.e-reading.club/chapter.php/102147/92/Li%2C_Yao_-_Real-Time_Concepts_for_Embedded_Systems.html


tAccessTask 1
tAccessTask 2
B
Shared Resource
Binary Semaphore
(Initial value = 1)

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()

Text

P(): đợi

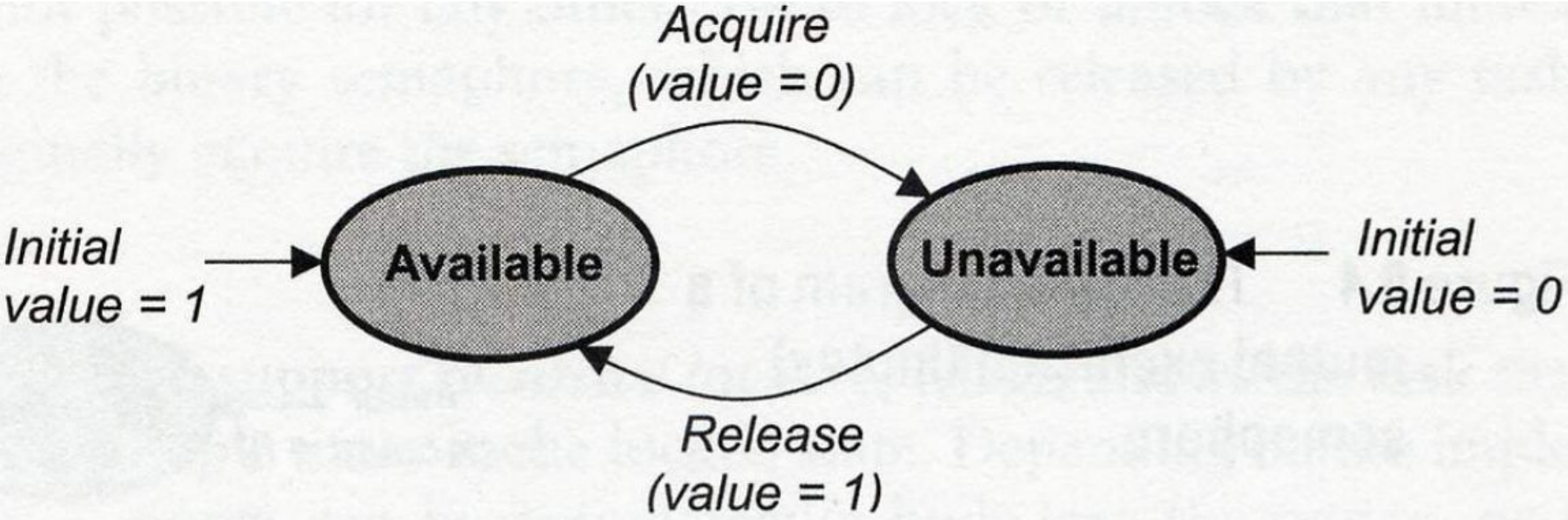
```
if (value == false) {
    Thêm bản thân luồng
  vào hàng đợi và khóa lại;
}
value = false;
```

Được thực thi nguyên tử

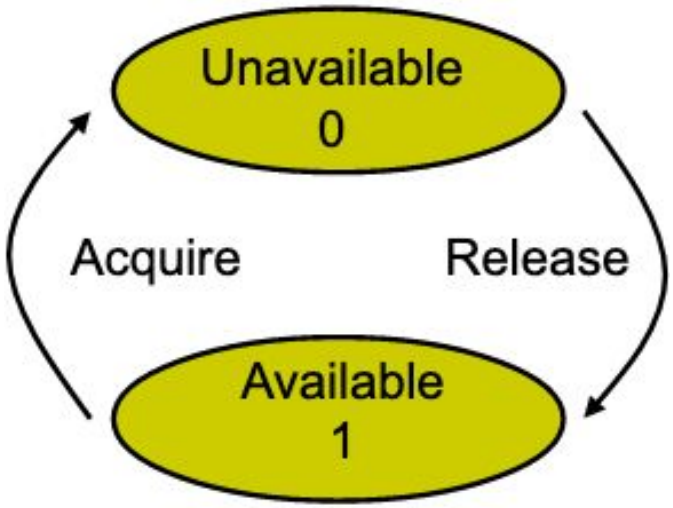
V(): đánh thức

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

Được thực thi nguyên tử


Acquire
(value =0)
Initial
value = 1
Available
Unavailable
Initial
value = 0
Release
(value = 1)

Semaphore nhị phân (2)
Ví dụ cài đặt



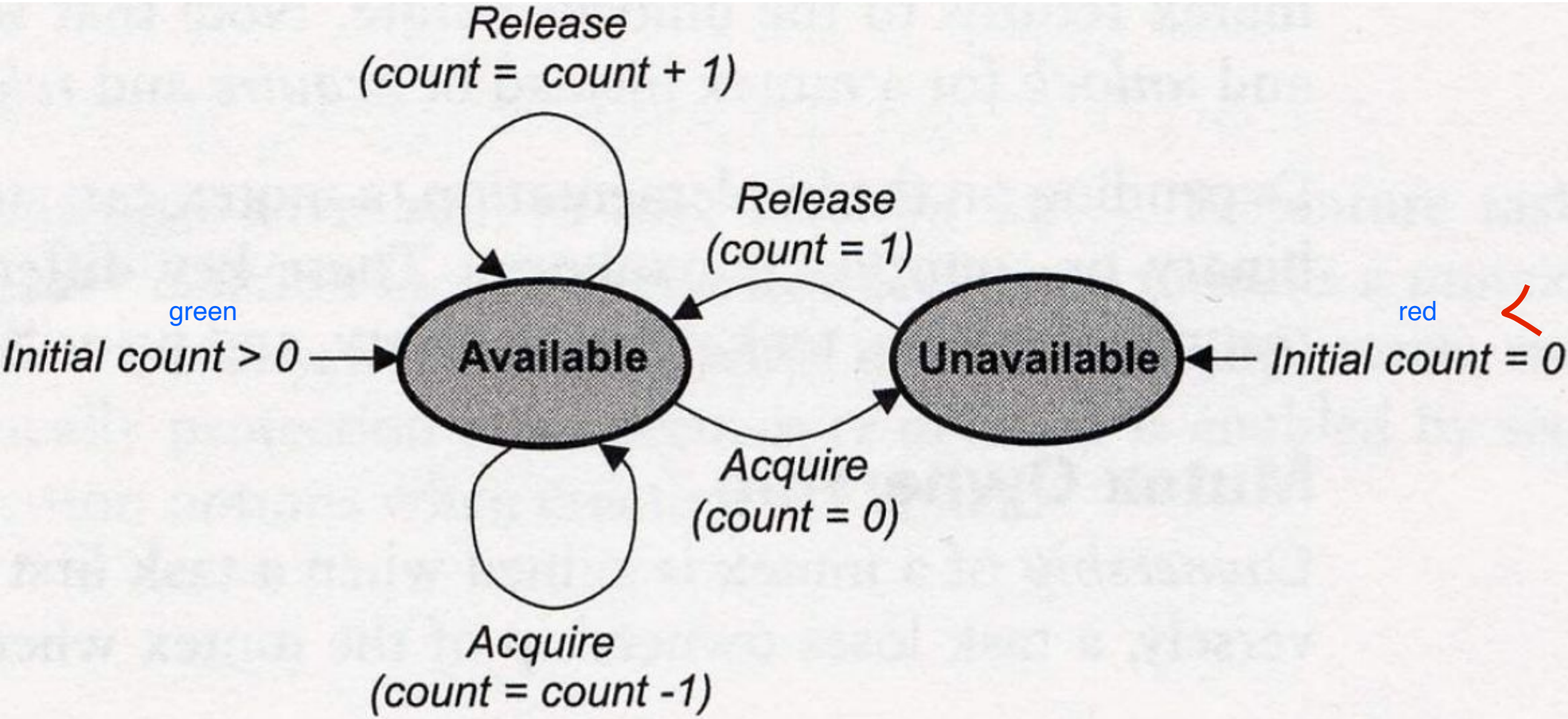

```
1  public class BinarySemaphore {
2      boolean value;
3      BinarySemaphore(boolean initValue) {
4          value = initValue;
5      }
6      public synchronized void P() {
7          if (value == false)
8              Util.myWait(this); // in queue of blocked processes
9          value = false;
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 nhị phân cho Bài toán Mutex

```
BinarySemaphore mutex = new BinarySemaphore(true);
mutex.P();
criticalSection();
mutex.V();
```

Semaphore đếm (1)



Semaphore đếm (2)
Ví dụ cài đặt

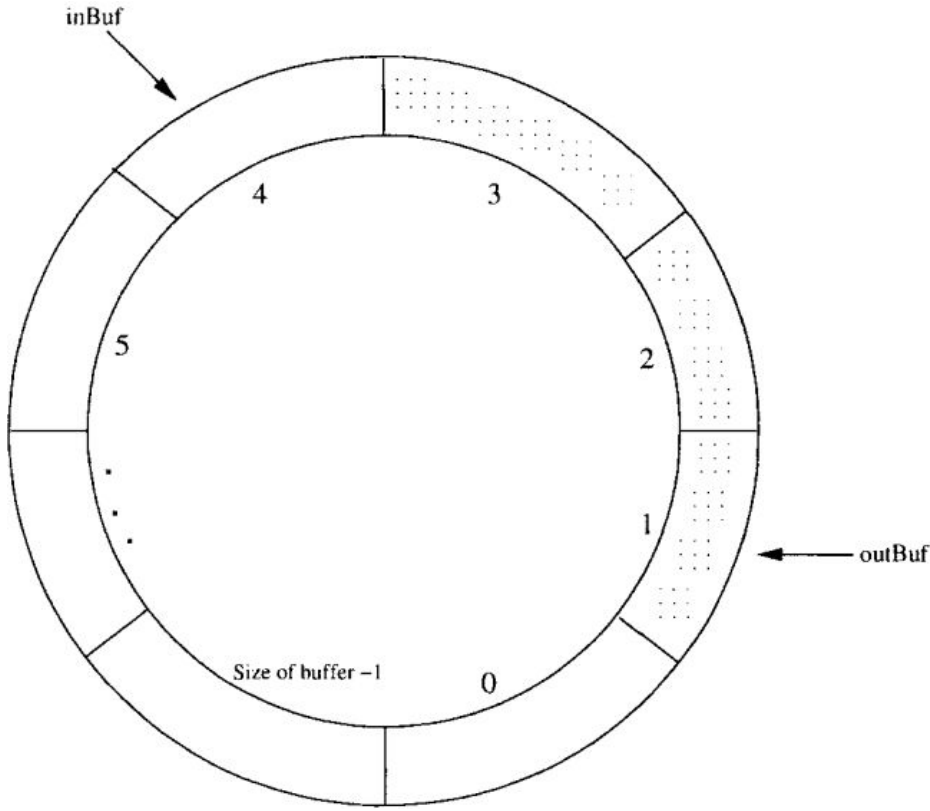
```
public class CountingSemaphore {
    int value;
    public CountingSemaphore(int initValue) {
        value = initValue;
    }
    public synchronized void P() {
        value--;
        if (value < 0) Util.myWait(this);
    }
    public synchronized void V() {
        value++;
        if (value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

Tìm hiểu: **java.util.concurrent.Semaphore**

Sử dụng Semaphore cho một số bài toán đồng bộ

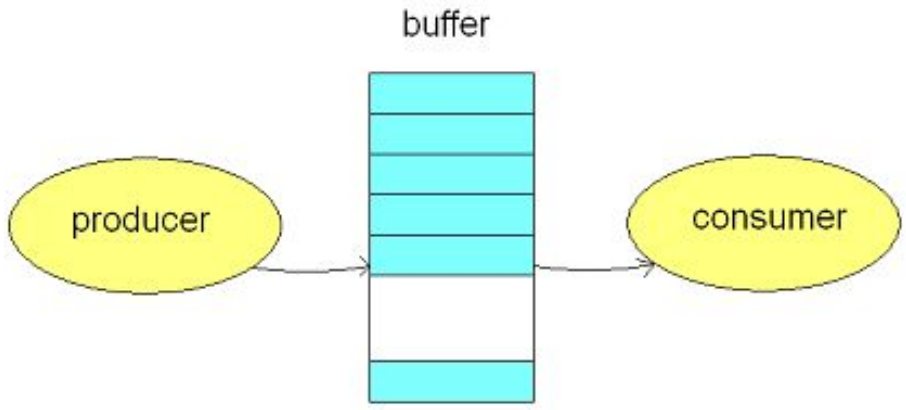
Bài toán 1: Nhà sản xuất & Người tiêu thụ (1)

- 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 … ?



```
1  class BoundedBuffer {
2      final int size = 10;
3      double[] buffer = new double[size];
4      int inBuf = 0, outBuf = 0;
5      BinarySemaphore mutex = new BinarySemaphore(true);
6      CountingSemaphore isEmpty = new CountingSemaphore(0);
7      CountingSemaphore isFull = new CountingSemaphore(size);
8
9      public void deposit(double value) {
10         isFull.P(); // wait if buffer is full
11         mutex.P(); // ensures mutual exclusion
12         buffer[inBuf] = value; // update the buffer
13         inBuf = (inBuf + 1) % size;
14         mutex.V();
15         isEmpty.V(); // notify any waiting consumer
16     }
17     public double fetch() {
18         double value;
19         isEmpty.P(); // wait if buffer is empty
20         mutex.P(); // ensures mutual exclusion
21         value = buffer[outBuf]; //read from buffer
22         outBuf = (outBuf + 1) % size;
23         mutex.V();
24         isFull.V(); // notify any waiting producer
25         return value;
26     }
27 }
```

```
import java.util.Random;
class Producer implements Runnable {
    BoundedBuffer b = null;
    public Producer(BoundedBuffer initb) {
        b = initb;
        new Thread(this).start();
    }
    public void run() {
        double item;
        Random r = new Random();
        while (true) {
            item = r.nextDouble();
            System.out.println("produced item " + item);
            b.deposit(item);
            Util.mySleep(200);
        }
    }
}
class Consumer implements Runnable {
    BoundedBuffer b = null;
    public Consumer(BoundedBuffer initb) {
        b = initb;
        new Thread(this).start();
    }
    public void run() {
        double item;
        while (true) {
            item = b.fetch();
            System.out.println("fetched item " + item);
             Util.mySleep(50);
        }
    }
}
class ProducerConsumer {
    public static void main(String[] args) {
        BoundedBuffer buffer = new BoundedBuffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);
    }
```

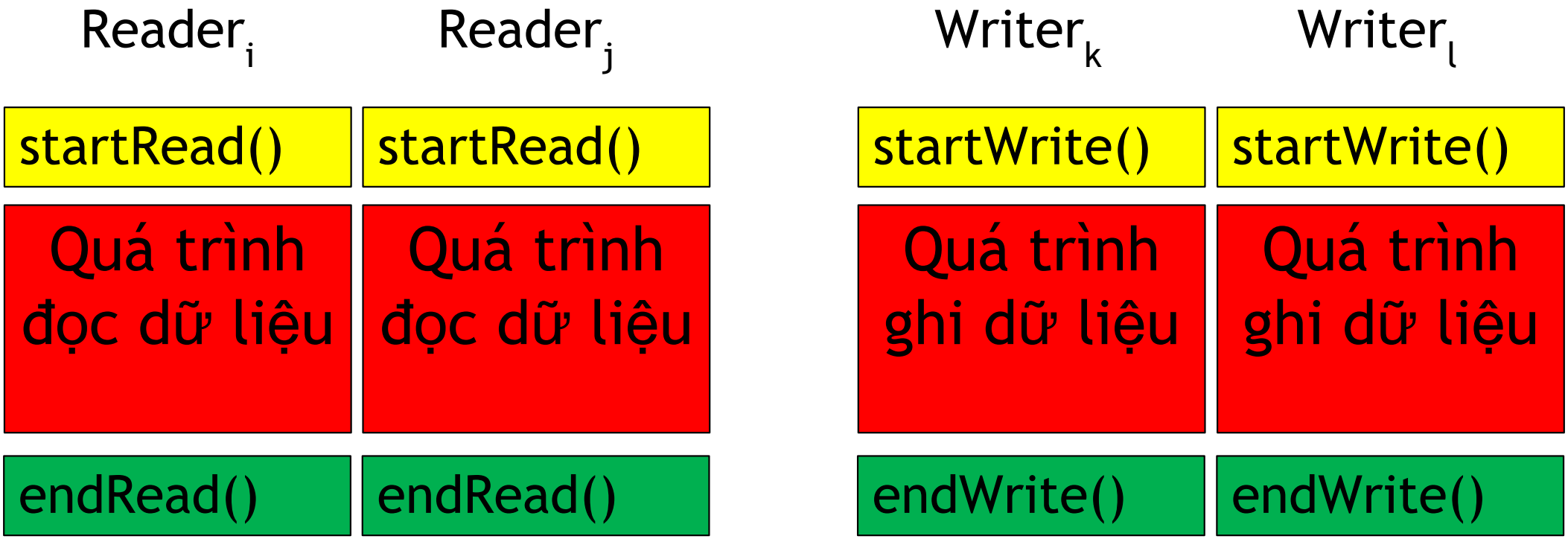
Bài toán 2: Người đọc & Người ghi

- 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**: Một người đọc và một người ghi không được truy cập đồng thời vào CSDL chia sẻ
  2. **Rằng buộc ghi-ghi**: Hai người ghi không được truy cập đồng thời vào CSDL chia sẻ
  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();
    }
}
```

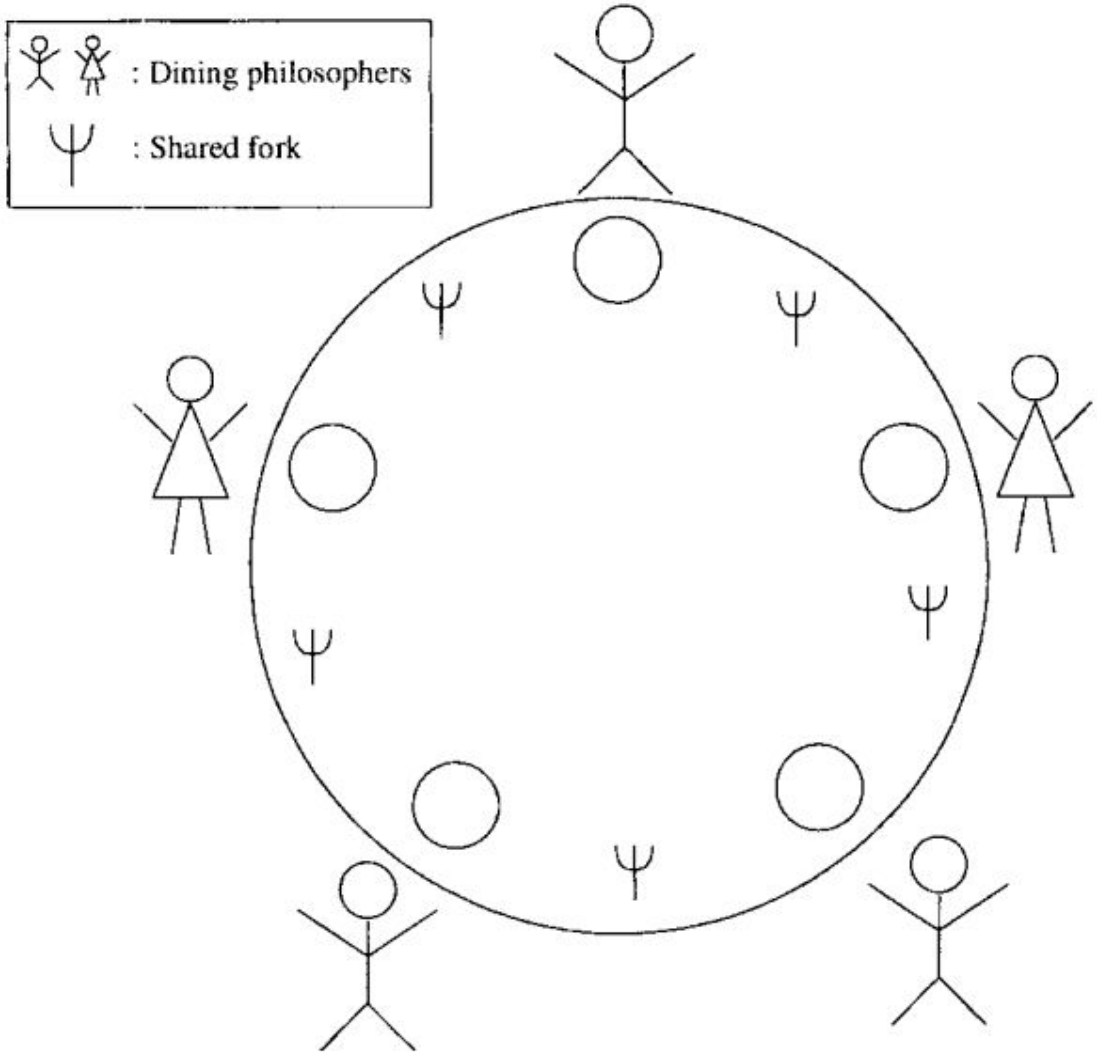
chuyển trạng thái không sẵn sàng đẩy vào hành động

nếu dãy cuối cùng đã thực hiện xong việc đọc CSDL => đánh thức luồng ghi bất kỳ đang bị khoá đẻ thực hiện tiếp



Starvation of a writer ?

Bài toán 3: Bữa tối của Triết gia (1)



Bài toán 3: Bữa tối của Triết gia (2)

- N triết gia:
  - *Nghĩ, Đói & Ăn*
- N cái nĩa (fork)
- N semaphore
  - ***fork [1..N]***
  - *Một semapore cho một cái nĩa*

Triết gia thứ i

```
while (true) {
    // suy nghĩ trong một thời gian, bắt đầu đói
    fork[i].P();
    fork[(i+1) % 5].P();
    //ăn; (vùng quan trọng - CS)
    fork[i].V();
    fork[(i+1) % 5].V();
}
```

```
1  class DiningPhilosopher implements Resource {
2      int n = 0;
3      BinarySemaphore [] fork = null;
4      public DiningPhilosopher(int initN) {
5          n = initN;
6          fork = new BinarySemaphore[n];
7          for (int i = 0; i < n; i++) {
8              fork[i] = new BinarySemaphore(true);
9          }
10     }
11     public void acquire(int i) {
12         fork[i].P();
13         fork[(i + 1) % n].P();
14     }
15     public void release(int i) {
16         fork[i].V();
17         fork[(i + 1) % n].V();
18     }
19     public static void main(String [] args) {
20         DiningPhilosopher dp = new DiningPhilosopher(5);
21         for (int i = 0; i < 5; i++)
22             new Philosopher(i, dp);
23     }
24 }
```

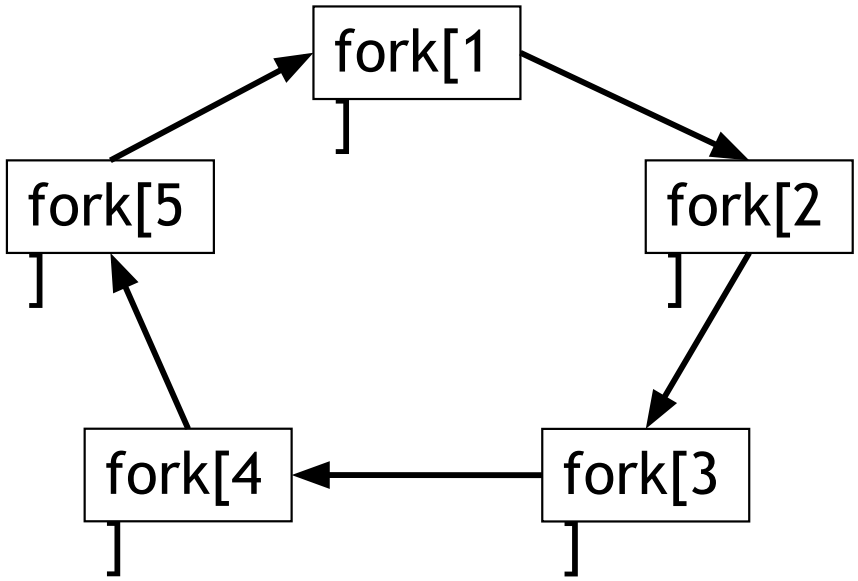
```
class Philosopher implements Runnable {
    int id = 0;
    Resource r = null;
    public Philosopher(int initId, Resource initr) {
        id = initId;
        r = initr;
        new Thread(this).start();
    }
    public void run() {
        while (true) {
            try {
                System.out.println("Phil " + id + " thinking");
                Thread.sleep(30);
                System.out.println("Phil " + id + " hungry");
                r.acquire(id);
                System.out.println("Phil " + id + " eating");
                Thread.sleep(40);
                r.release(id);
            } catch (InterruptedException e) {
                return;
            }
        }
    }
}

interface Resource {
    public void acquire(int i);
    public void release(int i);
}
```

Bài toán 3: Tình huống Deadlock

Triết gia 1	*Triết gia 2*	*Triết gia 3*	*Triết gia 4*	*Triết gia 5*
fork[1].P(); fork[2].P();	fork[2].P(); fork[3].P();	fork[3].P(); fork[4].P();	fork[4].P(); fork[5].P();	fork[5].P(); fork[1].P();

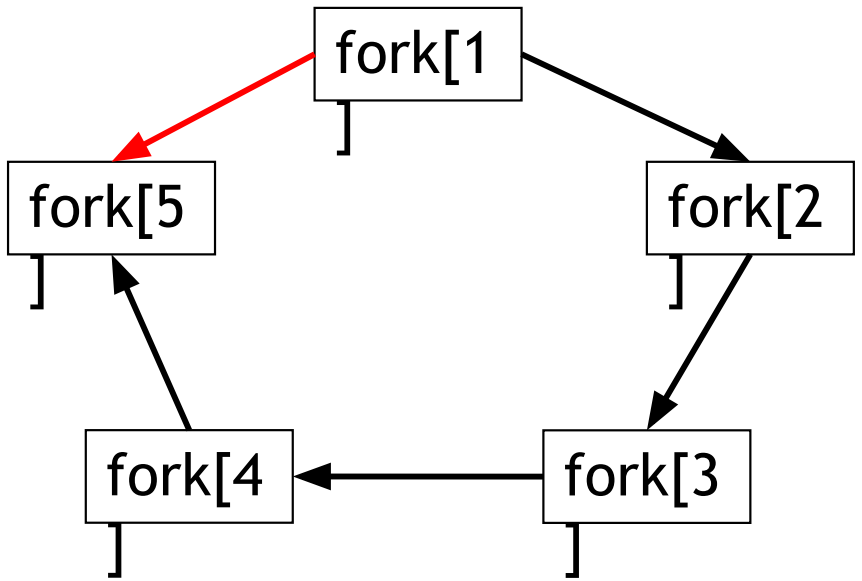
Có khả năng mọi luồng đều bị tắc nghẽn: ***Deadlock***



Bài toán 3: Giải pháp tránh Deadlock

Triết gia 1	*Triết gia 2*	*Triết gia 3*	*Triết gia 4*	*Triết gia 5*
fork[1].P(); fork[2].P();	fork[2].P(); fork[3].P();	fork[3].P(); fork[4].P();	fork[4].P(); fork[5].P();	fork[1].P(); fork[5].P();

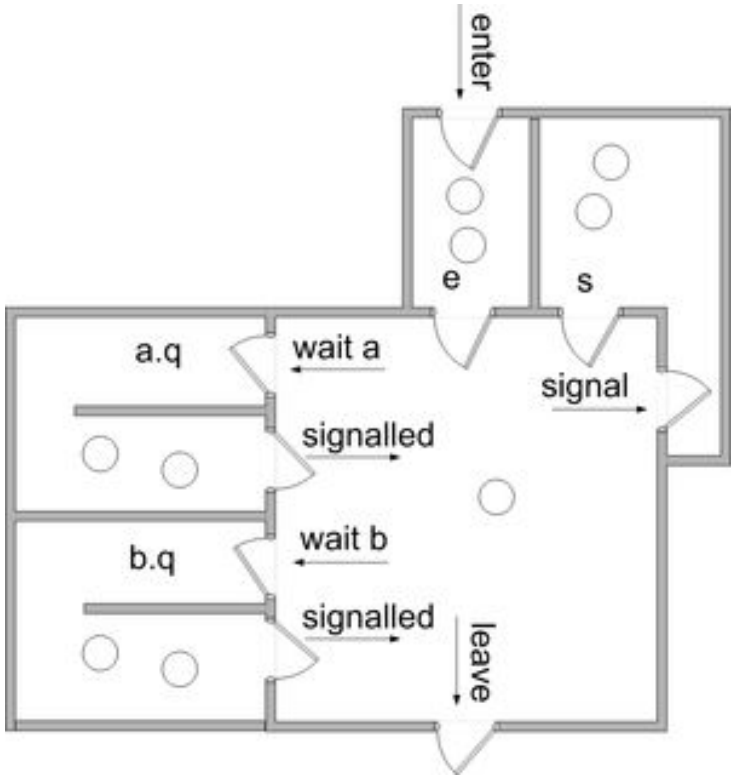
Tránh chu trình



tại 1 thời điểm bất kỳ chỉ có 1 luồng được thực hiện

29

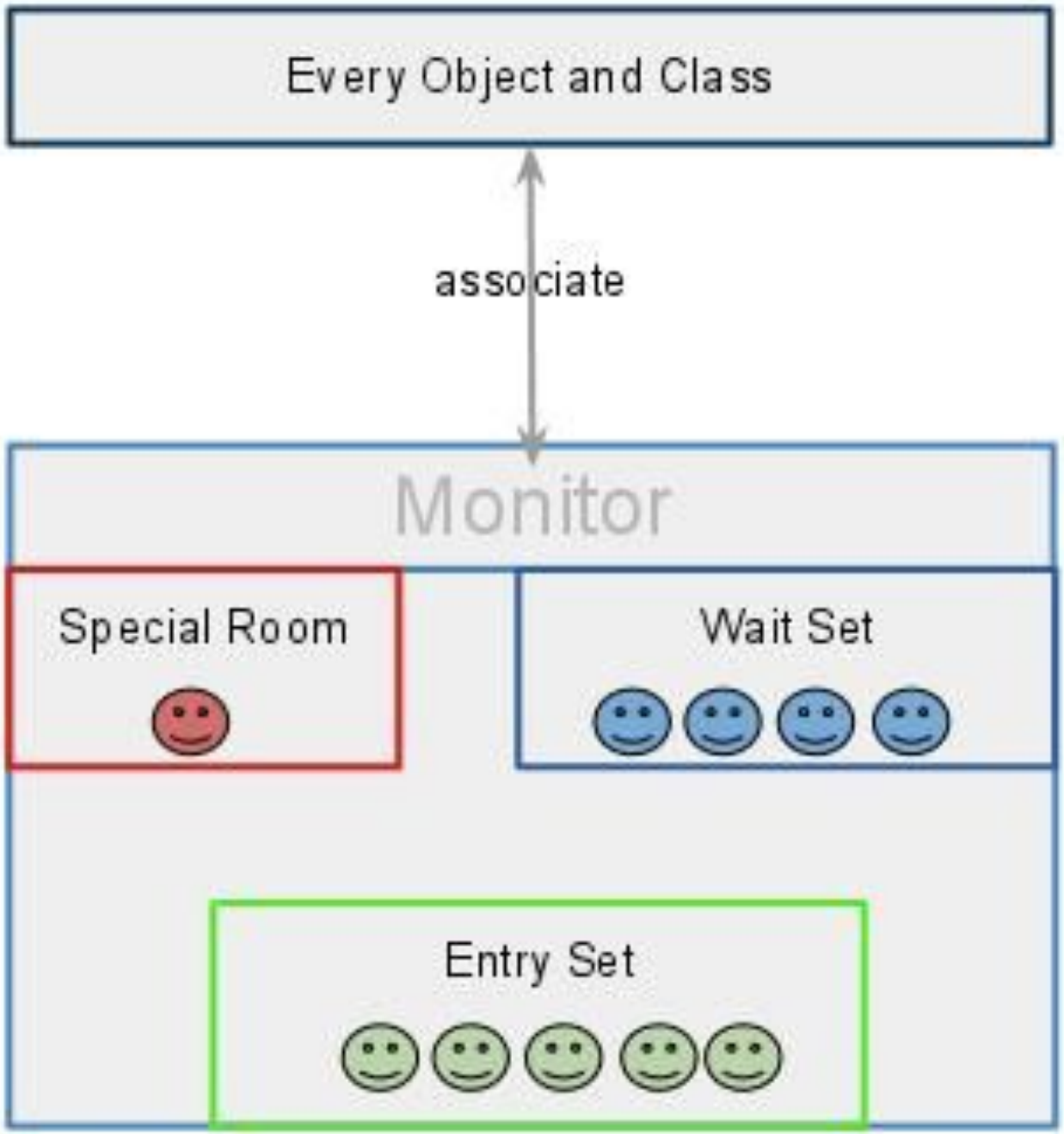
Phần 2. Monitor



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

Monitor (1)

- Semaphore là một công cụ mạnh cho bài toán đồng bộ luồng ở mức thấp, nhưng không thật sự hướng đối tượng
- Monitor ở mức cao hơn, hướng đối tượng và dễ sử dụng hơn
  - Có thể sử dụng Semaphore để cài đặt Monitor và ngược lại


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

Monitor (2)

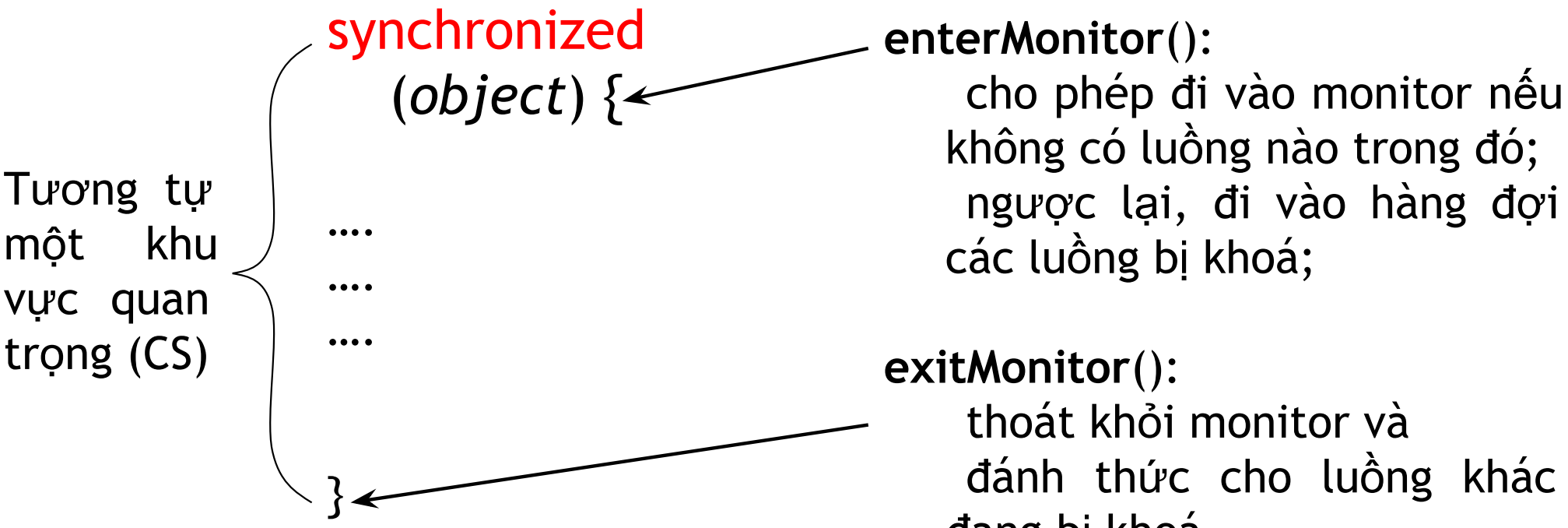
- Monitor có thể được xem như một lớp trong lập trình đồng thời gồm: *dữ liệu* và *các thao tác* trên dữ liệu đó
- Monitor hỗ trợ khái niệm *phương thức entry* để đảm bảo loại trừ lẫn nhau
  - Tại một thời điểm, chỉ có nhiều nhất 1 luồng có thể thực thi trong một phương thức entry
- Monitor đi kèm với một hàng đợi chứa những luồng đang đợi để vào monitor

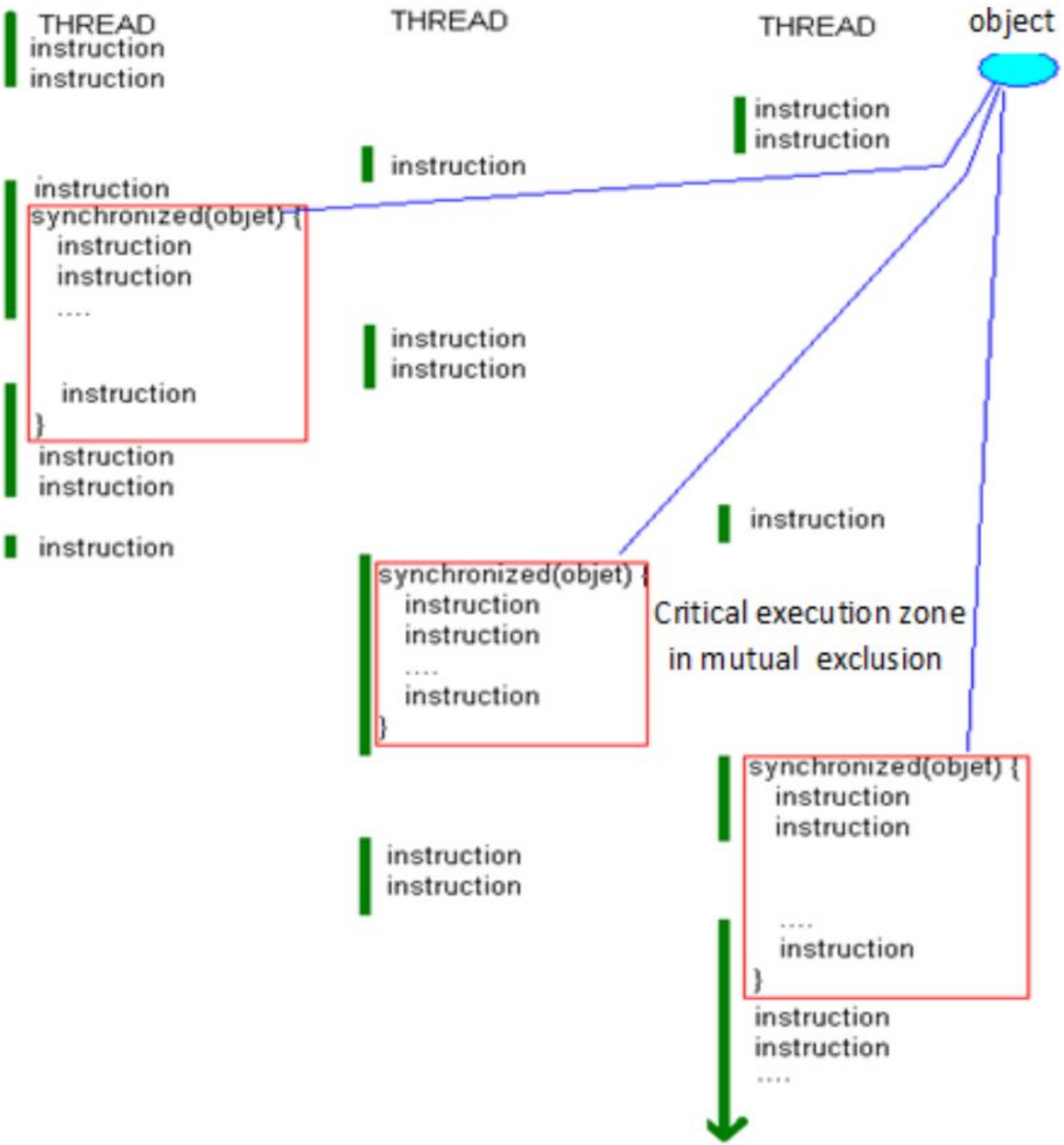
Monitor (3)

- Monitor hỗ trợ khái niệm *biến điều kiện* cho trường hợp yêu cầu sự đồng bộ có điều kiện
  - Một luồng phải đợi một điều kiện nào đó trở thành *đúng*
- 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




THREAD
instruction
instruction
instruction
synchronized(objet) {
instruction
instruction
....
instruction
}
instruction
instruction
instruction
THREAD
instruction
instruction
instruction
synchronized(objet) {
instruction
instruction
....
instruction
}
instruction
instruction
THREAD
instruction
instruction
instruction
Critical execution zone
in mutual exclusion
synchronized(objet) {
instruction
instruction
....
instruction
}
instruction
instruction
object

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

- horse signal
- mesa (java)

Luồng 0	*Luồng 1*
synchronized (object) { …. object.wait(); gọi hàm wave để T0 đi vào hàng đợi	Nếu đánh thức T0 gọi notify

	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 (1)
Blocking condition variables

Luồng 0	*Luồng 1*

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

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

…. }	ra khỏi Monitor, nhường lại cho T0
	…. }

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

Hoare-style Monitor (2)
Blocking condition variables

Luồng 0	*Luồng 1*

synchronized (object) {
if (x != 1)
object.wait();	

	synchronized (object) {
	if (x==1)
	object.notify();

assert(x==1); // x phải bằng 1
x++;
}	

	// x có thể không còn bằng 1 nữa ở đây }

Luồng 0 không phải lo lắng về sự thay đổi trạng thái của môi trường, biến điều kiện, so với ban

Mesa-style Monitor (1)
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

Mesa-style Monitor (2)
Nonblocking condition variables

Luồng 0	*Luồng 1*

synchronized (object) {
~~if (x != 1)~~
object.wait();	

	synchronized (object) {
	if (x==1)
	object.notify();

	assert(x == 1); //phải bằng 1
	}

// x có thể khác 1 ở đây }	

Trạng thái của môi trường khi luồng 0 được phép đi vào monitor có thể đã thay đổi so với ban đầu

Mesa-style Monitor (3)
Nonblocking condition variables

Luồng 0	*Luồng 1*

synchronized (object) {
while (x != 1)
object.wait();	

	synchronized (object) {
	x=1;
	object.notify();

	assert(x == 1); //phải bằng 1
	}

assert(x == 1); //phải bằng 1
}	

Luồng 0 phải sử dụng while để đảm bảo trạng thái của môi trường phù hợp

(Lý do: khi luồng 0 được phép đi vào monitor, thức dậy, để tiếp tục chạy, nó vẫn đang ở trong vòng

Hai cấu trúc sử dụng Monitor trong Java

```
public synchronized void
    myMethod () {
        ....
        ....
        ....

}
```

Khu vực quan trọng
ngôn ngữ thực thi
1 cách nguyên tử

=

```
public void myMethod () {
    synchronized (this) {
            ....
            ....
            ....
    }
}
```

Các phương thức tĩnh cũng có thể được đồng bộ hóa

Sử dụng Monitor cho một số bài toán đồng bộ

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

object sharedBuffer;

cho vào

```
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();
    }
}
```

lấy ra

```
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();
    }
}
```

```
1  class BoundedBufferMonitor {
2      final int sizeBuf = 10;
3      double[] buffer = new double[sizeBuf];
4      int inBuf = 0, outBuf = 0, count = 0;
5      public synchronized void deposit(double value) {
6          while (count == sizeBuf) // buffer full
7              Util.myWait(this);
8          buffer[inBuf] = value;
9          inBuf = (inBuf + 1) % sizeBuf;
10         count++;
11         if (count == 1) // items available for fetch
12             notify();
13     }
14     public synchronized double fetch() {
15         double value;
16         while (count == 0) // buffer empty
17             Util.myWait(this);
18         value = buffer[outBuf];
19         outBuf = (outBuf + 1) % sizeBuf;
20         count--;
21         if (count == sizeBuf - 1) // empty slots available
22             notify();
23         return value;
24     }
25 }
```

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();
    }
}
```

Luồng được đánh thức phải là một người ghi hay người đọc? Chứng minh

Bài toán 3: Bữa tối của Triết gia

```
public synchronized void acquire(int i) {
        state[i] = hungry;
        checkStartEating(i);
        while (state[i] != eating)
            Util.myWait(this);
}
public synchronized void release(int i) {
        state[i] = thinking;
        checkStartEating(left(i));
        checkStartEating(right(i));
}
void checkStartEating(int i) {
        if ( (state[left(i)] != eating) && (state[right(i)] !=
eating) && (state[i] == hungry) ) {
            state[i] = eating;
            notifyAll();
        }
}
```

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