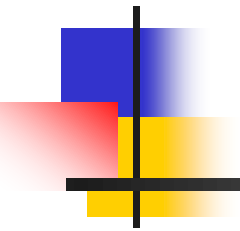
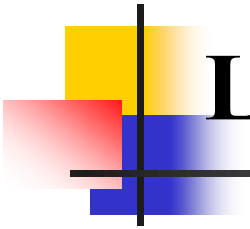


Bài 7

LẬP TRÌNH ĐA TUYẾN



Lập trình đa tuyến

- Tiến trình, đa nhiệm và đa luồng
- Xử lý đa luồng trong Java
- Mức ưu tiên của luồng
- Vấn đề đồng bộ hoá
- Bài toán tắc nghẽn

Tiến trình, đa nhiệm và đa luồng

- Tiến trình : là một chương trình chạy trên hệ điều hành và được quản lý thông qua các thẻ
- Tiểu trình : là một đơn vị xử lý cơ bản của hệ thống. Một tiến trình sở hữu nhiều tiểu trình
- Đơn nhiệm : tại một thời điểm chỉ có một tiến trình
- Đa nhiệm: ở cùng một thời điểm có nhiều hơn một tiến trình thực hiện đồng thời trên cùng một máy tính. Có hai kỹ thuật đa nhiệm:

  + Đa nhiệm dựa trên các *tiến trình*

  + Đa nhiệm dựa trên các *luồng*

Tiến trình, đa nhiệm và đa luồng

- Một tiến trình có thể bao gồm nhiều luồng. Các luồng của một tiến trình có thể chia sẻ với nhau về không gian địa chỉ chương trình, các đoạn dữ liệu và môi trường xử lý, đồng thời cũng có vùng dữ liệu riêng để thao tác.
- Trong môi trường đơn luồng, ở mỗi thời điểm chỉ cho phép một tác vụ thực thi.
- Kỹ thuật đa nhiệm cho phép tận dụng được những thời gian rỗi của CPU để thực hiện những tác vụ khác.

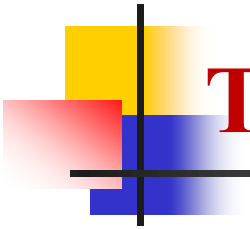
Tiến trình, đa nhiệm và đa luồng

- Đa nhiệm có thể thực hiện được theo hai cách:

  + Phụ thuộc vào hệ điều hành, nó có thể cho tạm ngừng chương trình mà không cần tham khảo các chương trình đó.

  + Các chương trình chỉ bị dừng lại khi chúng tự nguyện nhường điều khiển cho chương trình khác.

- Nhiều hệ điều hành hiện nay đã hỗ trợ đa luồng, `Java` hỗ trợ đa nhiệm dựa trên các luồng và cung cấp các đặc tính ở mức cao cho lập trình đa luồng.

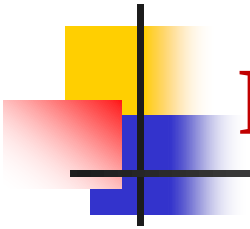


Tạo và quản lý luồng

- Khi chương trình Java thực thi hàm main() tức là luồng main được thực thi.
- Tuyến này được tạo ra một cách tự động, tại đây :

  - Các luồng con sẽ được tạo ra từ đó

  - Nó là luồng cuối cùng kết thúc việc thực thi. Ngay khi luồng main() ngừng thực thi, chương trình bị chấm dứt



Phân chia thời gian giữa các luồng

- CPU thực thi chỉ một luồng tại một thời điểm nhất định.
- Các luồng có độ ưu tiên bằng nhau thì được phân chia thời gian sử dụng bộ vi xử lý.

Lập trình đa luồng

- Với Java ta có thể xây dựng các chương trình đa luồng
- Một ứng dụng có thể bao gồm nhiều luồng, mỗi luồng được gắn công việc cụ thể và được thực thi đồng thời với các luồng khác
- Java cung cấp hai giải pháp tạo lập luồng:
  - Thiết lập lớp con của Thread
  - Cài đặt lớp xử lý luồng từ giao diện Runnable

Lập trình đa luồng

Cách thứ nhất **:**

Tạo ra một lớp kế thừa từ lớp Thread và ghi đè phương thức run của lớp Thread như sau:

```
class MyClass extends Thread
        {
        // Một số thuộc tính
        public void run(){
                // Các lệnh cần thực hiện theo luồng
                }
         // Một số hàm khác được viết đè hay được bổ sung
        }
```

Khi chương trình chạy nó sẽ gọi một hàm đặc biệt đã được khai báo trong Thread đó là start() để bắt đầu một luồng đã được tạo ra.

Lập trình đa luồng

Cách thứ hai:

+ Java giải quyết hạn chế trên bằng cách xây dựng lớp để tạo ra các luồng thực hiện trên cơ sở cài đặt giao diện hỗ trợ luồng.

+ Tạo ra một lớp triển khai từ giao diện Runnable, cài đặt phương thức run

```
class MyClass implements Runnable
{
        // Các thuộc tính
        // Nạp chồng hay viết đè một số hàm
        public void run(){
                . . .
        }
}
```

Trạng thái và các phương thức của lớp Thread

- Trạng thái:
  - born
  - ready to run
  - running
  - sleeping
  - waiting
  - ready
  - blocked
  - dead
- Phương thức:
  - start( )
  - sleep( )
  - wait( )
  - notify( )
  - run( )
  - stop( )

Các trạng thái của Thread

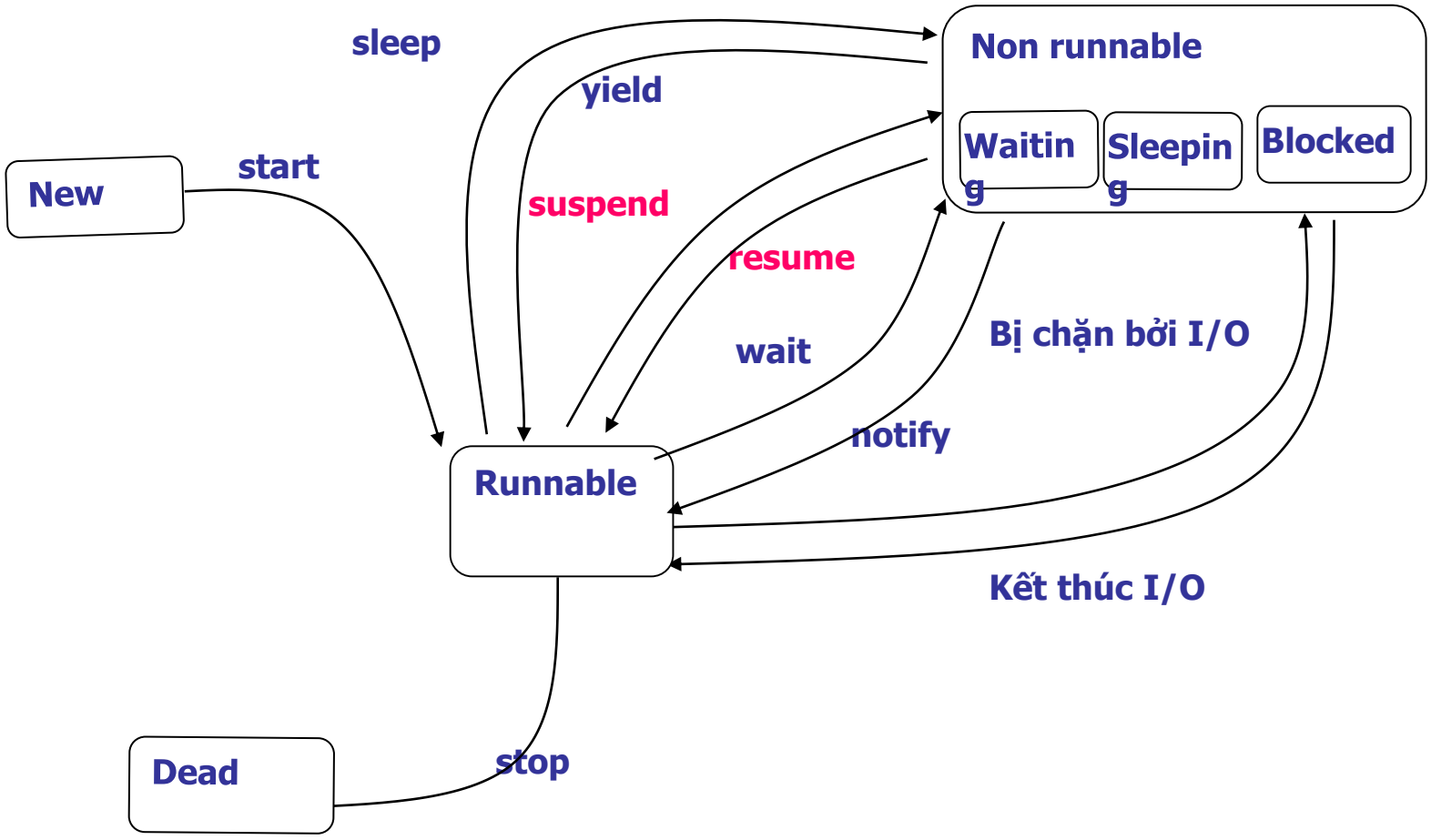
Một luồng có thể ở một trong các trạng thái sau:

+ New: Khi một luồng mới được tạo ra với toán tử new() và sẵn sàng hoạt động.

+ Runnable: Trạng thái mà luồng đang chiếm CPU để thực hiện, khi bắt đầu thì nó gọi hàm start(). Bộ lập lịch phân luồng của hệ điều hành sẽ quyết định luồng nào sẽ được chuyển về trạng thái Runnable và hoạt động. Cũng cần lưu ý rằng ở một thời điểm, một luồng ở trạng thái Runnable có thể hoặc không thể thực hiện.

+ Non runnable (blocked): Từ trạng thái runnable chuyển sang trạng thái ngừng thực hiện ("bị chặn") khi gọi một trong các hàm: sleep(), suspend(), wait(), *hay bị chặn lại ở* Input/output.

Các trạng thái của Thread



Các trạng thái của Thread

Một luồng có thể ở một trong các trạng thái sau:

+ Waiting: khi ở trạng thái Runnable, một luồng thực hiện hàm wait() thì nó sẽ chuyển sang trạng thái chờ đợi (Waiting).

+ Sleeping: khi ở trạng thái Runnable, một luồng thực hiện hàm sleep() thì nó sẽ chuyển sang trạng thái ngủ (Sleeping).

+ Blocked: khi ở trạng thái Runnable, một luồng bị chặn lại bởi những yêu cầu về tài nguyên, như yêu cầu vào/ra (*I/O*), thì nó sẽ chuyển sang trạng bị chặn (Blocked).

Các trạng thái của Thread

Mỗi luồng phải thoát ra khỏi trạng thái Blocked để quay về trạng thái Runnable, khi:

+ Nếu một luồng đã được cho đi "ngủ" (sleep) sau khoảng thời gian bằng số micro giây *n* đã được truyền vào tham số của hàm sleep(n).

+ Nếu một luồng bị chặn lại vì vào/ra và quá trình này đã kết thúc.

+ Nếu luồng bị chặn lại khi gọi hàm wait(), sau đó được thông báo tiếp tục bằng cách gọi hàm notify() hoặc notifyAll().

+ Nếu một luồng bị chặn lại để chờ monitor của đối tượng đang bị chiếm giữ bởi luồng khác, khi monitor đó được giải phóng thì luồng bị chặn này có thể tiếp tục thực hiện (khái niệm monitor được đề cập ở phần sau).

Các trạng thái của Thread

+ Nếu một luồng bị chặn lại bởi lời gọi hàm suspend(), muốn thực hiện thì trước đó phải gọi hàm resume().

+ Hàm suspend() có tác dụng tạm ngừng tuyến, ít được dùng do không nhả tài nguyên của hệ thống, dễ dẫn đến deaklock.

Nếu ta gọi các hàm không phù hợp đối với các luồng thì JVM sẽ phát sinh ra ngoại lệ IllegalThreadStateException.

+ Dead: Luồng chuyển sang trạng thái "chết" khi nó kết thúc hoạt động bình thường, hoặc gặp phải ngoại lệ không thực hiện tiếp được.

+ Trong trường hợp đặc biệt, bạn có thể gọi hàm stop() để kết thúc ("giết chết") một luồng.

Mức ưu tiên của các luồng

+ Trong Java, mỗi luồng có một mức ưu tiên thực hiện nhất định.

+ Khi chương trình chính thực hiện sẽ tạo ra luồng chính, luồng cha. Luồng này sẽ tạo ra các luồng con, và cứ thế tiếp tục.

+ Theo mặc định, một luồng sẽ kế thừa mức ưu tiên của luồng cha của nó. Bạn có thể tăng hay giảm mức ưu tiên của luồng bằng cách sử dụng hàm setPriority().

+ Mức ưu tiên của các luồng có thể đặt lại trong khoảng từ MIN_PRIORITY (Trong lớp Thread được mặc định bằng 1) và MAX_PRIORITY (mặc định bằng 10), hoặc NORM_PRIORITY (mặc định là 5).

Mức ưu tiên của các luồng

+ Luồng có mức ưu tiên cao nhất tiếp tục thực hiện cho đến khi:

- Nó nhường quyền điều khiển cho luồng khác bằng cách gọi hàm yield()
- Nó dừng thực hiện (bị "dead" hoặc chuyển sang trạng thái bị chặn).

Mức ưu tiên của các luồng

+ Vấn đề nảy sinh là chọn luồng nào để thực hiện khi có nhiều hơn một luồng sẵn sàng thực hiện và có cùng một mức ưu tiên cao nhất? Nói chung, một số cơ sở sử dụng bộ lập lịch lựa chọn ngẫu nhiên, hoặc lựa chọn chúng để thực hiện theo thứ tự xuất hiện.

***Ví dụ*:**

Chúng ta hãy xét chương trình hiển thị các quả bóng màu xanh hoặc đỏ nảy (chuyền) theo những đường nhất định.

Mỗi khi nhấn nút "Blue ball" thì có 5 luồng được tạo ra với mức ưu tiên thông thường (mức 5) để hiển thị và di chuyển các quả bóng xanh.

Khi nhấn nút "Red ball" thì cũng có 5 luồng được tạo ra với mức ưu tiên (mức 7) cao hơn mức thông thường để hiển thị và di chuyển các quả bóng đỏ.

Để kết thúc trò chơi bạn nhấn nút "Close".

Mức ưu tiên của các luồng

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
public class Bounce{
        public static void main(String arg[]){
            JFrame fr = new BounceFrame();
            fr.show();
        }
}
class BounceFrame extends JFrame{
        public BounceFrame(){
            setSize(300, 200);
            setTitle("Bong chuyen");
            addWindowListener(new WindowAdapter(){
                        public void windowClosing(WindowEvent e){
                                System.exit(0);
                        }});
```

Mức ưu tiên của các luồng

```
Container contentPane = getContentPane();
canvas = new JPanel();
contentPane.add(canvas, "Center");
JPanel p = new JPanel();
addButton(p, "Blue ball", new ActionListener(){
        public void actionPerformed(ActionEvent evt){
                for(int i = 0; i < 5; i++){
                        Ball b = new Ball(canvas, Color.blue);
                        b.setPriority(Thread.NORM_PRIORITY);
                        b.start();  }}});
addButton(p, "Red ball", new ActionListener(){
        public void actionPerformed(ActionEvent evt){
                for(int i = 0; i < 5; i++){
                        Ball b = new Ball(canvas, Color.red);
                        b.setPriority(Thread.NORM_PRIORITY + 2);
                        b.start();
                }}});
```

Mức ưu tiên của các luồng

```
        addButton(p, "Close", new ActionListener(){
                public void actionPerformed(ActionEvent evt){
                        canvas.setVisible(false);
                        System.exit(0);
                }
        });
        contentPane.add(p, "South");
        }
        public void addButton(Container c, String title, ActionListener a){
                JButton b = new JButton(title);
                c.add(b);
                b.addActionListener(a);
        }
        private JPanel canvas;
}
```

Mức ưu tiên của các luồng

```
class Ball extends Thread{
        public Ball(JPanel b, Color c){
            box = b; color = c;
        }
        public void draw(){
            Graphics g = box.getGraphics(); g.setColor(color);
            g.fillOval(x, y, XSIZE, YSIZE); g.dispose();
        }
        public void move(){
            if(!box.isVisible()) return;
            Graphics g = box.getGraphics();
```

Mức ưu tiên của các luồng

```
g.setXORMode(box.getBackground());
g.setColor(color);
g.fillOval(x, y, XSIZE, YSIZE);
x += dx;          y += dy;
Dimension d = box.getSize();
if(x < 0){
        x = 0; dx = -dx;
}
if(x + XSIZE >= d.width){
        x = d.width - XSIZE; dx = -dx;
}
if(y < 0){
        y = 0; dy = -dy; }
```

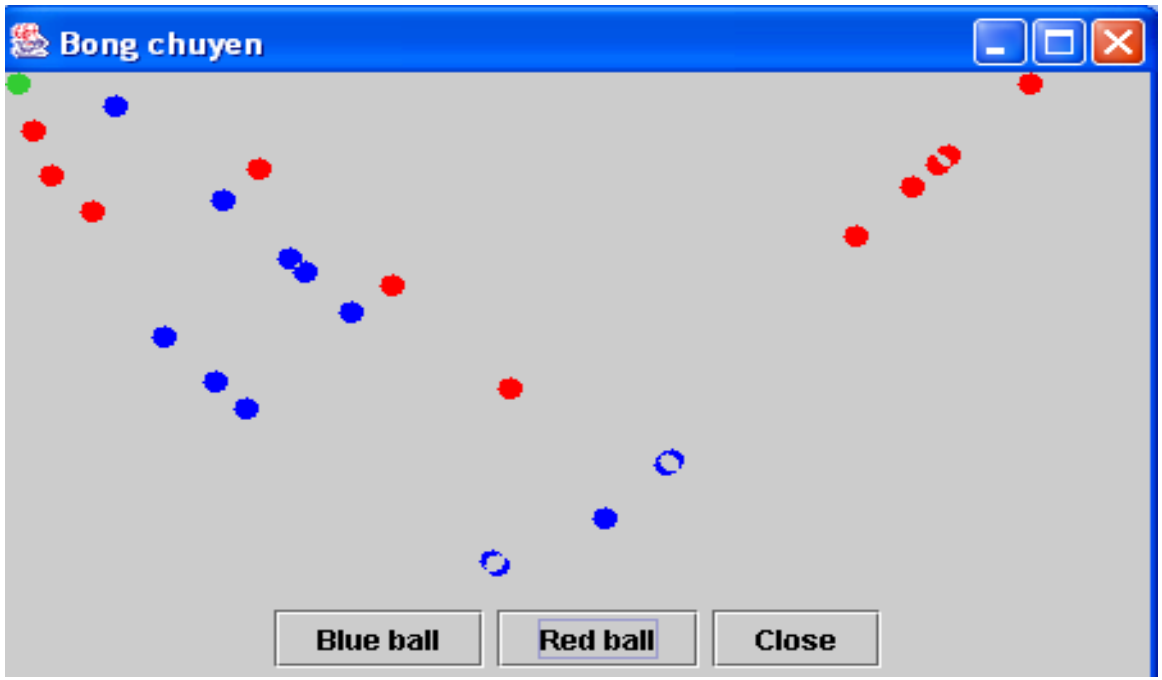
Mức ưu tiên của các luồng

```
    if(y + YSIZE >= d.height){
            y = d.height - YSIZE; dy = -dy;
    }
    g.fillOval(x, y, XSIZE, YSIZE);
    g.dispose();
}
public void run(){
    try{
            for(int i = 1; i <= 1000; i++){
                    move();          sleep(5);
            }
    }catch(InterruptedException e){
    }
}
```

Mức ưu tiên của các luồng

```
    private JPanel box;
    private static final int XSIZE = 10;
    private static final int YSIZE = 10;
    private int x = 0;
    private int y = 0;
    private int dx = 2;
    private int dy = 2;
    private Color color;
}
```

Mức ưu tiên của các luồng



Chạy chương trình trên chúng ta nhận thấy hình như những quả bóng đỏ nảy nhanh hơn vì các luồng thực hiện chúng có mức ưu tiên cao hơn.

Mức ưu tiên của các luồng

+ Các luồng có mức ưu tiên thấp hơn sẽ không có cơ hội thực hiện nếu những luồng cao hơn không nhường, hoặc nhường bằng hàm `yield()`.

+ Nếu có những luồng đang ở trạng thái `Runnable` mà có mức ưu tiên ít nhất bằng mức ưu tiên của luồng vừa nhường thì một trong số chúng được xếp lịch để thực hiện.

+ Bộ lập lịch thường xuyên tính lại mức ưu tiên của các luồng đang thực hiện

+ Tìm luồng có mức ưu tiên cao nhất để thực hiện.

Đa tuyến với Applets

- Trong Java ta có thể tạo ra các tuyến thi hành song song bằng cách triển khai giao diện Runnable
- Java không hỗ trợ kế thừa bội
- Muốn kế thừa từ một lớp nào đó mà lại muốn đa tuyến thì bắt buộc sử dụng giao diện Runnable
- Các chương trình Java dựa trên Applet thường sử dụng nhiều hơn một tuyến

Đa tuyến với Applets

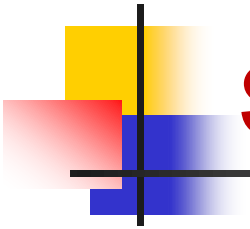
- Trong đa tuyến với Applets, Lớp 'java.applet.Applet' là lớp con được tạo ra một Applet người sử dụng đã định nghĩa
- Lớp con của Applet không thể dẫn xuất được trực tiếp từ lớp Thread.
- Cách để lớp con Applet là tuyến:
  - implements Runnable
  - Truyền đối tượng Runnable vào hàm constructor của Thread.

Sự đồng bộ

- Khi nhiều tuyến truy cập tài nguyên dùng chung
- Tài nguyên không thể chia xẻ, khi đó tài nguyên có thể bị phá hỏng

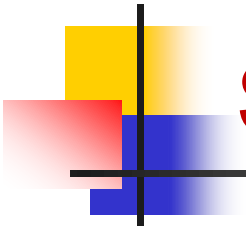
Ví dụ : Một luồng đọc dữ liệu, trong khi luồng khác lại thay đổi

- Cần cho phép một luồng hoàn thành tác vụ của nó, rồi cho phép luồng kế tiếp thực thi



Sự đồng bộ

- Thâm nhập các tài nguyên/dữ liệu bởi nhiều tuyến
- Sự đồng bộ (Synchronization)
- Sự quan sát (Monitor)



Sự đồng bộ

- Để thâm nhập sự quan sát của một đối tượng, lập trình viên sử dụng từ khóa 'synchronized' khi khai báo phương thức.
- Mỗi một đối tượng sẽ có một bộ quản lý khóa, chỉ cho một phương thức "synchronized" của đối tượng đó chạy tại một thời điểm
- Khi một tuyến đang được thực thi trong phạm vi một phương thức đồng bộ (synchronized), bất kỳ tuyến khác hoặc phương thức đồng bộ khác mà cố gắng gọi nó trong thời gian đó sẽ phải đợi

Sự đồng bộ

- Các luồng chia sẻ với nhau cùng một không gian bộ nhớ, nghĩa là chúng có thể chia sẻ với nhau các tài nguyên.
- Khi có nhiều hơn một luồng cùng muốn sử dụng một tài nguyên sẽ xuất hiện tình trạng căng thẳng, ở đó chỉ cho phép một luồng được quyền truy cập.
- Để cho các luồng chia sẻ với nhau được các tài nguyên và hoạt động hiệu quả, luôn đảm bảo nhất quán dữ liệu thì phải có cơ chế đồng bộ chúng.

Sự đồng bộ

- Mấu chốt của sự đồng bộ là khái niệm "monitor" (giám sát) hay còn gọi là "semaphore" (cờ hiệu)
- Khái niệm "semaphore" thường được sử dụng để điều khiển đồng bộ các hoạt động truy cập vào những tài nguyên dùng chung.
- Một luồng muốn truy cập vào một tài nguyên dùng chung (như biến dữ liệu) thì trước tiên nó phải yêu cầu để có được monitor riêng.
- Khi có được monitor thì luồng như có được "chìa khoá" để "mở cửa" vào miền "tranh chấp" để sử dụng những tài nguyên đó.

Sự đồng bộ

- Cơ chế monitor thực hiện hai nguyên tắc đồng bộ chính:

  + Không một luồng nào khác được phân monitor khi có một luồng đã yêu cầu và đang chiếm giữ. Những luồng có yêu cầu monitor sẽ phải chờ cho đến khi monitor được giải phóng.

  + Khi có một luồng giải phóng (ra khỏi) monitor, một luồng đang chờ *monitor* có thể truy cập vào tài nguyên dùng chung tương ứng với monitor đó.

- Mọi đối tượng trong Java đều có monitor, mỗi đối tượng có thể được sử dụng như một khoá loại trừ nhau, cung cấp khả năng để đồng bộ truy cập vào những tài nguyên chia sẻ.
- Trong lập trình có hai cách để thực hiện đồng bộ:

  + **Các hàm được đồng bộ**

  + **Các khối được đồng bộ**

Sự đồng bộ

- Hàm của một lớp chỉ cho phép một luồng được thực hiện ở một thời điểm thì nó phải khai báo synchronized, được gọi là *hàm đồng bộ*.
- Một luồng muốn thực hiện hàm đồng bộ thì nó phải chờ để có được monitor của đối tượng có hàm đó.
- Trong khi một luồng đang thực hiện hàm đồng bộ thì tất cả các luồng khác muốn thực hiện hàm này của cùng một đối tượng, đều phải chờ cho đến khi luồng đó thực hiện xong và được giải phóng.
- Bằng cách đó, những hàm được đồng bộ sẽ không bao giờ bị tắc nghẽn.

Sự đồng bộ

- Những hàm không được đồng bộ của đối tượng có thể được gọi thực hiện mọi lúc bởi bất kỳ đối tượng nào.
- Khi chạy, chương trình sẽ thi hành tuần tự các lệnh cho đến khi kết thúc chương trình.
- Trong Java, hàm đồng bộ có thể khai báo static. Các lớp cũng có thể có các monitor tương tự như đối với các đối tượng.
- Một luồng yêu cầu monitor của lớp trước khi nó có thể thực hiện với một hàm được đồng bộ tĩnh (static) nào đó trong lớp, đồng thời các luồng khác muốn thực hiện những hàm như thế của cùng một lớp thì bị chặn lại.

Sự đồng bộ

***Ví dụ*: Hệ thống ngân hàng có 10 tài khoản, trong đó có các giao dịch chuyển tiền giữa các tài khoản với nhau một cách ngẫu nhiên. Chương trình tạo ra 10 luồng cho 10 tài khoản. Mỗi giao dịch được một luồng phục vụ sẽ chuyển một lượng tiền ngẫu nhiên từ một tài khoản này sang tài khoản khác.**

- Nếu chương trình thực hiện với 10 luồng hoạt động không đồng bộ để chuyển tiền giữa các tài khoản trong ngân hàng.
- Vấn đề sẽ nảy sinh khi có hai luồng đồng thời muốn chuyển tiền vào cùng một tài khoản. Giả sử hai luồng cùng thực hiện:

```
accounts[to] += amount;
```

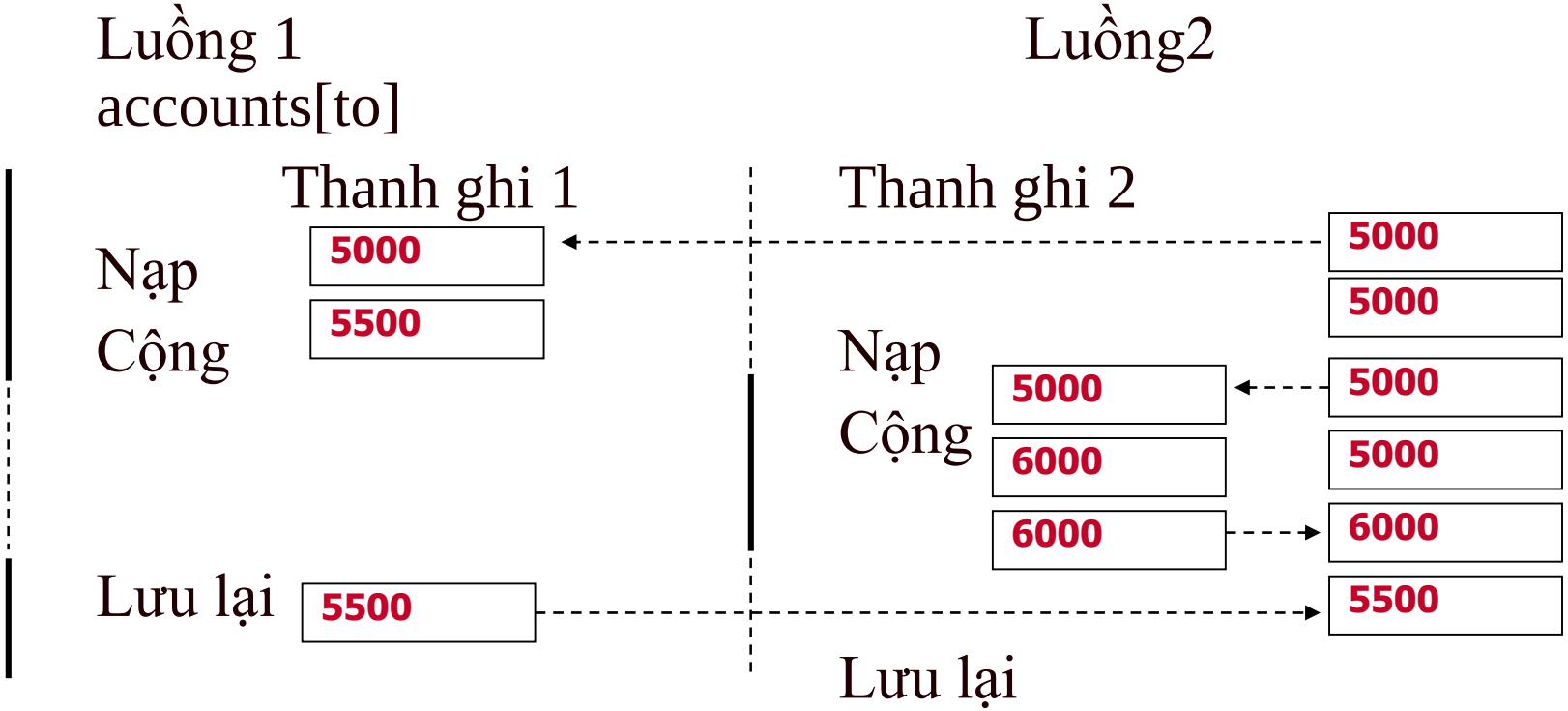
Câu lệnh này được thực hiện như sau:

1. Nạp accounts[to] vào thanh ghi.
2. Cộng số tiền trong tài khoản accounts với amount
3. Lưu lại kết quả cho accounts[to].

Sự đồng bộ

- Chúng ta có thể giả thiết luồng thứ nhất thực hiện bước 1 và 2 với amount = 500, sau đó nó bị ngắt.
- Luồng thứ hai có thể thực hiện trọn vẹn cả ba bước trên với amount = 1000, sau đó luồng thứ nhất kết thúc việc cập nhật bằng cách thực hiện nốt bước 3. Quá trình này được mô tả như ở hình sau :

Sự đồng bộ



- Kết thúc luồng thứ nhất, `accounts[to]` có 6000, nhưng ngay sau đó luồng thứ hai kết thúc thì cũng chính tài khoản đó chưa chuyển tiền đi đâu cả, nhưng lại chỉ còn 5500. Đúng ra nó phải là 6500

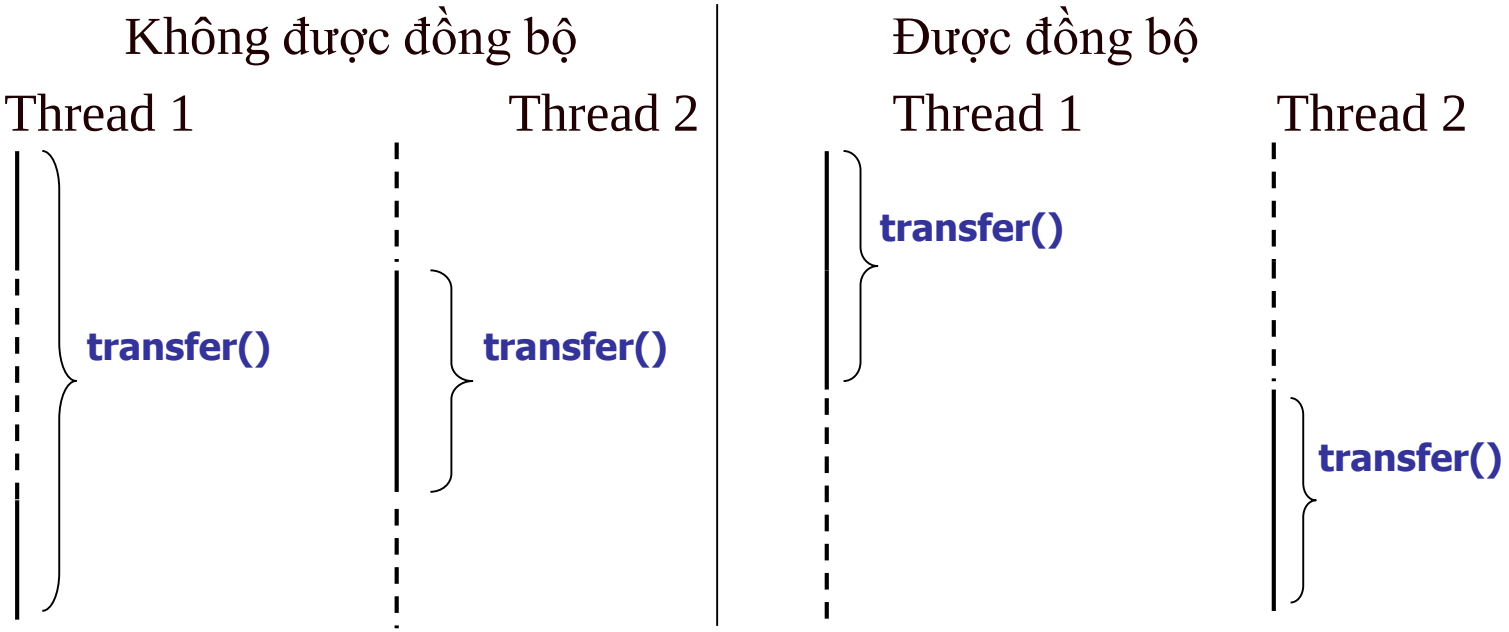
Sự đồng bộ

- Java sử dụng cơ chế đồng bộ khá hiệu quả là monitor. Một hàm sẽ không bị ngắt nếu bạn khai báo nó là synchronized.

```
    public synchronized void transfer(int from, int to, int
amount){
    if(accounts[from] < amount) return;
    accounts[from] -= amount; accounts[to] += amount;
    numTransacts++;
    if(numTransacts % NTEST == 0) test();
}
    public synchronized void test(){
    int sum = 0;
    for(int i = 0; i < accounts.length; i++) sum +=
accounts[i];
    System.out.println("Giao dich: " + numTransacts
          + " tong so: " + sum);
}
```

Sự đồng bộ

- Khi có một luồng gọi hàm được đồng bộ thì nó được đảm bảo rằng hàm này phải thực hiện xong thì luồng khác mới được sử dụng đối với cùng một đối tượng.
- Hoạt động của các luồng không đồng bộ và đồng bộ của hai luồng thực hiện gọi hàm `transfer()`



Các khoá đối tượng

- Khi một luồng gọi một hàm được đồng bộ thì đối tượng của nó bị "khoá", giống như khoá cửa phòng.
- Như vậy, khi một luồng khác muốn gọi hàm được đồng bộ của cùng đối tượng đó thì sẽ không mở được.
- Sau khi thực hiện xong, luồng ở bên trong giải phóng hàm được đồng bộ vừa sử dụng, ra khỏi đối tượng và đưa chìa khoá ra ngoài bậc cửa để những luồng khác có thể tiếp tục công việc của mình.

Các khoá đối tượng

- Một luồng có thể giữ nhiều khoá đối tượng ở cùng một thời điểm, như trong khi đang thực hiện một lời gọi hàm đồng bộ của một đối tượng, nó lại gọi tiếp hàm đồng bộ của đối tượng khác.
- Nhưng, tại mỗi thời điểm, mỗi khoá đối tượng chỉ được một luồng sở hữu.
- Chúng ta hãy phân tích chi tiết hơn hoạt động của hệ thống ngân hàng. Một giao dịch chuyển tiền sẽ không thực hiện được nếu không còn đủ tiền. Phải chờ cho đến các tài khoản khác chuyển tiền đến và khi có đủ thì mới thực hiện được giao dịch đó.

Các khoá đối tượng

```
public synchronized void transfer(int from, int to,
int amount){
        while(accounts[from] < amount)
            wait();
        // Chuyển tiền
}
```

Các khoá đối tượng

- Chúng ta sẽ làm gì khi trong tài khoản không có đủ tiền? Tất nhiên là phải chờ cho đến khi có đủ tiền trong tài khoản. Nhưng transfer() là hàm được đồng bộ.
- Do đó, khi một luồng đã chiếm được khoá đối tượng thì luồng khác sẽ không có cơ hội sở hữu trừ nó, cho đến khi luồng trước giải phóng khoá đó.
- Khi wait() được gọi ở trong hàm được đồng bộ (như ở transfer()), luồng hiện thời sẽ bị chặn lại và trao lại khoá đối tượng cho luồng khác.
- Có sự khác nhau thực sự giữa luồng đang chờ để sử dụng hàm đồng bộ với hàm bị chặn lại bởi hàm wait().

Các khoá đối tượng

- Khi một luồng gọi wait() thì nó được đưa vào danh sách hàng đợi.

  + Cho đến khi các luồng chưa được đưa ra khỏi danh sách hàng đợi thì bộ lập lịch sẽ bỏ qua và do vậy chúng không thể tiếp tục được.

  + Để đưa một luồng ra khỏi danh sách hàng đợi thì phải có một luồng khác gọi notify() hoặc notifyAll() trên *cùng một đối tượng*.

  + *notify*() đưa một luồng bất kỳ ra khỏi danh sách hàng đợi.

  + *notifyAll*() đưa tất cả các luồng ra khỏi danh sách hàng đợi.

Các khoá đối tượng

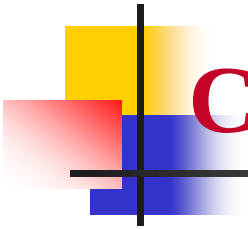
- Những luồng đưa ra khỏi danh sách hàng đợi sẽ được bộ lập lịch kích hoạt chúng. Ngay tức khắc luồng nào chiếm được khoá đối tượng thì sẽ bắt đầu thực hiện.
- Như vậy, trong hàm transfer() chúng ta gọi notifyAll() khi kết thúc việc chuyển tiền để một trong các luồng có thể được tiếp tục thực hiện và tránh bế tắc.
- Cuối cùng chương trình sử dụng cơ chế đồng bộ được viết lại như sau:

Các khoá đối tượng

```
public class SynBankTransfer{
    public static void main(String arg[]){
        Bank b = new Bank(NACCOUNTS, INI_BALANCE);
        for(int i = 0; i < NACCOUNTS; i++){
            TransferThread t = new TransferThread(b, i,
    INI_BALANCE);
            t.setPriority(Thread.NORM_PRIORITY + i % 2);
            t.start();
        }
    }
    public static final int NACCOUNTS = 10;
    public static final int INI_BALANCE = 10000;
}
```

Các khoá đối tượng

```
class Bank{
    public static final int NTEST = 1000;
    private int[] accounts;
    private long numTransacts = 0;
    public Bank(int n, int initBalance){
        accounts = new int[n];
        for(int i = 0; i < accounts.length; i++)
            accounts[i] = initBalance;
        numTransacts = 0;
    }
```



Các khoá đối tượng

```
public void transfer(int from, int to, int amount){
    while(accounts[from] < amount) wait();
    accounts[from] -= amount;
    accounts[to] += amount;
    numTransacts++;
    notifyAll();
    if(numTransacts % NTEST == 0) test();
}
```

Các khoá đối tượng

```
public synchronized void test(){
        int sum = 0;
        for(int i = 0; i < accounts.length; i++)
                sum += accounts[i];
        System.out.println("Giao dich: " + numTransacts + " tong so:
    " + sum);
    }

    public int size(){
        return accounts.length;
    }
}
```

Các khoá đối tượng

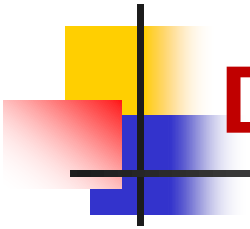
```
class TransferThread extends Thread{
    private Bank bank;
    private int fromAcc;
    private int maxAmount;
    public TransferThread(Bank b, int from, int max){
        bank = b;
        fromAcc = from;    maxAmount = max;
    }
```

Các khoá đối tượng

```
    public void run(){
        try{
          while(!interrupted()){
                int toAcc = (int)(bank.size() * Math.random());
                int amount = (int)(maxAmount * Math.random());
                bank.transfer(fromAcc, toAcc, amount);
                sleep(1);
           }
        }catch(InterruptedException e){
        }
    }
}
```

Các khoá đối tượng

- Nếu bạn chạy chương trình với các hàm transfer(), test() được đồng bộ thì mọi việc sẽ thực hiện chính xác đúng theo yêu cầu.
- Tuy nhiên, bạn cũng có thể nhận thấy chương trình sẽ chạy chậm hơn chút ít bởi vì phải trả giá cho cơ chế đồng bộ nhằm đảm bảo cho hệ thống hoạt động chính xác, đảm bảo nhất quán dữ liệu, hoặc tránh gây ra tắc nghẽn.



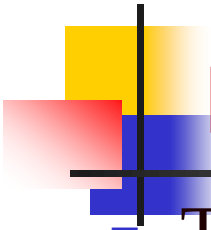
Deadlock

- Một “deadlock” xảy ra khi hai tuyến có một phụ thuộc vòng quanh trên một cặp đối tượng đồng bộ
- Cơ chế đồng bộ trong Java là rất tiện lợi, khá mạnh, nhưng không giải quyết được mọi vấn đề nảy sinh trong quá trình xử lý đa luồng.

Deadlock

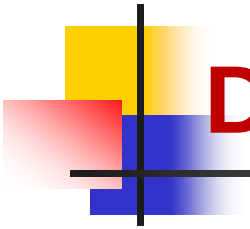
Ví dụ : ở Account 1 có 2000$, Account 2 có 3000$ và Thread 1 cần chuyển 3000$ từ Account 1 sang Account 2, ngược lại Thread 2 cần chuyển 3500$ từ Account 2 sang Account 1.

- Khi đó, Thread 1 và Thread 2 rơi vào tình trạng *chết tắc* hoặc *tắc nghẽn* vì chúng chặn lẫn nhau.
- Một hệ thống mà tất cả các luồng (tiến trình) bị chặn lại để chờ lẫn nhau và không một luồng (tiến trình) nào thực hiện tiếp thì được gọi là hệ thống bị chết tắc (tắc nghẽn).
- Trong tình huống ở trên, cả hai luồng đều phải gọi wait() xử lý hai tài khoản đều không đủ số tiền để chuyển.



Deadlock

- Trong chương trình `SynBankTransfer.java`, hiện tượng tắc nghẽn không xuất hiện bởi một lý do đơn giản. Mỗi giao dịch chuyển tiền nhiều nhất là 10000$.
- Có 10 tài khoản với tổng số tiền là 100000$. Do đó, ở mọi thời điểm đều có ít nhất một tài khoản có không ít hơn 10000$, nghĩa là luồng phụ trách tài khoản đó được phép thực hiện.
- Tuy nhiên, khi lập trình ta có thể gây ra tình huống khác có thể làm xuất hiện tắc nghẽn

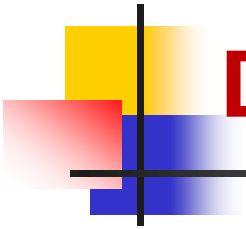


Deadlock

- Trong `SynBankTransfer.java` thay vì gọi `notifyAll()` ta gọi `notify()`.
- Như ở trên đã phân tích, `notifyAll()` thông báo cho tất cả các luồng đang chờ để có đủ tiền chuyển đi có thể tiếp tục thực hiện, còn `notify()` chỉ báo cho một luồng được tiếp tục.
- Khi đó, nếu luồng được thông báo lại không thể thực hiện, vì không đủ tiền để chuyển chẳng hạn, thì tất cả các luồng khác cũng sẽ bị chặn lại.

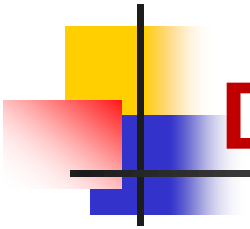
Deadlock

- Chúng ta hãy xét ví dụ sau:
  + `Account 1`: 19000$
  + Tất cả các `Account` còn lại đều có 9000$
  + `Thread 1`: chuyển 9500$ từ `Account 1` sang `Account 2`
  + Tất cả các luồng khác đều chuyển sang tài khoản khác một lượng tiền là 9100$.
- Chỉ có `Thread 1` đủ tiền để chuyển còn các luồng khác bị chặn lại. `Thread 1` thực hiện chuyển tiền xong ta có:
  + `Account 1`: 9500$
  + `Account 2`: 18500$
  + Tất cả các `Account` còn lại đều có 9000$



Deadlock

- Giả sử `Thread 1` gọi `notify()`. Hàm này chỉ thông báo cho một luồng ngẫu nhiên để nó có thể tiếp tục thực hiện. Giả sử đó là `Thread 3`. Nhưng luồng này cũng không chuyển được vì không đủ tiền ở tài khoản `Account 3`, nên phải chờ (gọi `wait()`).
- `Thread 1` vẫn tiếp tục thực hiện. Một giao dịch mới ngẫu nhiên lại được tạo ra.
- `Chẳng hạn Thread 1` chuyển 9600$ từ `Account 1` sang `Account 2`. Bây giờ `Thread` lại gọi `wait()`, và như vậy tất cả các luồng đều rơi vào tình trạng tắc nghẽn.

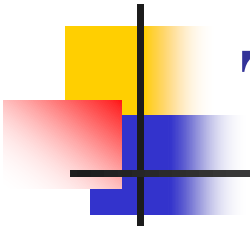


Deadlock

- Qua ví dụ trên cho thấy, một *ngôn ngữ lập trình có cơ chế hỗ trợ đồng bộ là chưa đủ để giải quyết vấn đề tắc nghẽn*.
- Quan trọng là khi thiết kế chương trình, ta phải đảm bảo rằng ở mọi thời điểm có ít nhất một luồng (tiến trình) tiếp tục thực hiện.

Phương thức finalize()

- Java cung cấp một cách để làm sạch một tiến trình trước khi điều khiển trở lại hệ điều hành
- Phương thức finalize(), nếu hiện diện sẽ được thực thi trên mỗi đối tượng, trước khi sự dọn rác
- Câu lệnh của phương thức finalize() như sau :
  - **protected void finalize( ) throws Throwable**
- Tham chiếu không phải là sự dọn rác; chỉ các đối tượng mới được dọn rác



Tổng kết

- Đa luồng
- Tạo và quản lý luồng
- Vòng đời của luồng
- Phạm vi luồng và các phương thức
- Đồng bộ luồng
- Deadlock