Lập trình JAVA cơ bản

Tuần 7

Giảng viên: Trần Đức Minh

Nội dung bài giảng

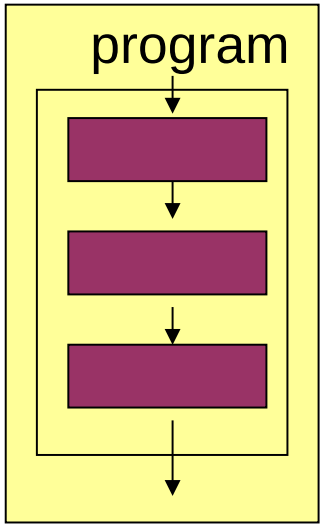
- Tiến trình, tuyến và đa nhiệm
- Tạo tuyến
- Vòng đời của một tuyến
- Đồng bộ hóa tuyến
- Tuyến ma
- Nhóm tuyến

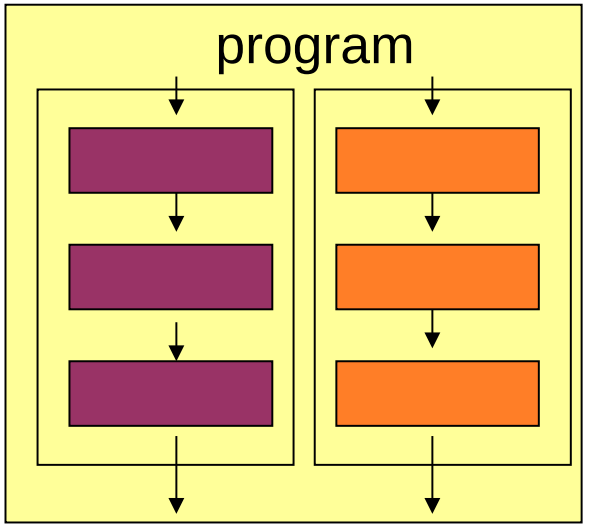
Khái niệm về Tiến trình

- **Một tiến trình** (process) là một chương trình đang được thực thi trên máy tính.
- Mỗi tiến trình có một **không gian địa chỉ của riêng** nó.
- Khi chạy một ứng dụng trên Java thì đó cũng chính là một tiến trình.

Khái niệm về Tuyến

- **Tuyến** (Thread) là một mạch thi hành độc lập của một nhiệm vụ nào đó bên trong chương trình (process).
- Một process có thể chứa nhiều tuyến và cho phép **các tuyến được phép chạy đồng thời**.
- Tất cả các tuyến trong cùng một process đều **sử dụng chung một không gian địa chỉ bộ nhớ**.





Khái niệm về đa nhiệm

- **Đa nhiệm** (Multitasking) là khả năng chạy đồng thời nhiều công việc cùng một lúc.
- Có hai kỹ thuật đa nhiệm cơ bản
  - **Đa tiến trình** (Process-based multitasking): Máy tính chạy nhiều chương trình đồng thời.
  - **Đa tuyến** (Thread-based multitasking): Một chương trình có nhiều tuyến chạy đồng thời.

Đa nhiệm

- Đa tiến trình
  - Mỗi tiến trình có một **vùng nhớ dữ liệu độc lập**.
  - **Việc giao tiếp** giữa hai tiến trình với nhau cần **xử lý khá phức tạp** do phải đăng ký việc lưu và tải bản đồ bộ nhớ, cập nhật danh sách thông tin, … ngoài ra còn bị giới hạn truy cập một số thành phần.
  - Đa tiến trình **không thuộc quyền kiểm soát của Java** mà thuộc quyền kiểm soát của hệ điều hành.

Đa nhiệm

- Đa tuyến
  - Các tuyến **sử dụng chung vùng nhớ dữ liệu** của chương trình.
  - Việc **giao tiếp giữa các tuyến với nhau dễ dàng** hơn so với các process với nhau.
  - Đa tuyến phụ thuộc vào **quyền kiểm soát của Java**.

Đa nhiệm

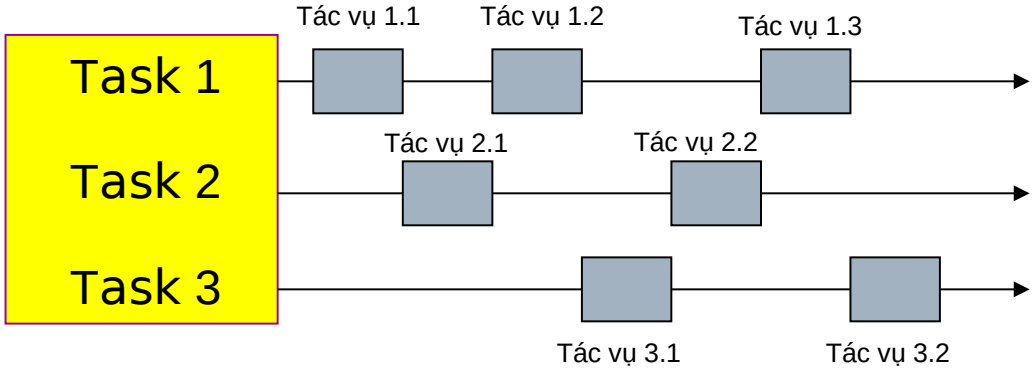
- Đa tuyến
  - Ưu điểm
    - Có thể thực hiện nhiều công việc cùng một lúc.
    - Mỗi tuyến dù dùng chung và chia sẻ tài nguyên với các tuyến khác nhưng vẫn thực hiện một cách độc lập.
    - Mỗi tuyến khi chạy là độc lập nên nó sẽ không ảnh hưởng đến các tuyến khác nếu xảy ra ngoại lệ.
    - Tiết kiệm thời gian xử lý bằng cách phối hợp hoạt động của các tuyến với nhau
      - Ví dụ: Tuyến chính chạy nhiệm vụ chính của chương trình, các tuyến phụ tính toán rồi gửi kết quả về cho tuyến chính.

Đa nhiệm

- Đa tuyến
  - Nhược điểm
    - Khi chương trình càng nhiều tuyến thì việc xử lý sẽ càng phức tạp.
    - Do các tuyến dùng chung bộ nhớ nên cần có cơ chế xử lý tranh chấp bộ nhớ và đồng bộ hóa dữ liệu.
    - Cần phát hiện để tránh trường hợp tuyến chết (deadlock - đây là một tuyến chạy bên trong ứng dụng nhưng không làm gì cả)
      - Ví dụ: Cả 2 tuyến cùng đợi nhau hoàn thành công việc rồi mới xử lý tiếp. Điều này làm cho 2 tuyến đều chạy mà không làm gì trong ứng dụng.

Đa nhiệm

- Khái niệm về **xử lý đồng thời** (concurrency)
  - Việc thực hiện nhiều nhiệm vụ hay tác vụ cùng một lúc của concurrency **chỉ là gần như cùng một thời điểm**.
  - Một thành phần tên là **CPU time-slicing** (bộ chia cắt thời gian CPU) của hệ điều hành được dùng để hỗ trợ cho mỗi nhiệm vụ sau khi thực hiện xong một tác vụ sẽ chuyển sang trạng thái chờ.
  - Khi một nhiệm vụ chuyển sang trạng thái chờ, CPU sẽ được hệ điều hành gán cho một nhiệm vụ khác để thực hiện tiếp.
  - Hệ điều hành sẽ **dựa trên độ ưu tiên của từng nhiệm vụ** để tự động phân chia CPU và tài nguyên tính toán cho hợp lý.



Nhiều công việc thi hành trên một CPU

Đa nhiệm

- Khái niệm về **xử lý song song** (parallelism)
  - Việc thực hiện nhiều nhiệm vụ hay tác vụ cùng một lúc của parallelism **là thực sự song song** do dựa vào cấu trúc đa lõi (multi-core) của CPU.
  - Mỗi nhiệm vụ được phân chia vào một lõi của CPU để thực hiện, do đó parallelism yêu cầu phần cứng phải có nhiều đơn vị xử lý.
  - Parallelism không thể thực hiện được đối với CPU chỉ có đơn lõi

Tạo tuyến

- Các tuyến trong Java được coi là các đối tượng.
- Có 2 cách để tạo lớp cho tuyến
  - **Kế thừa từ lớp java.lang.Thread**
  - **Thực hiện interface java.lang.Runnable**
    - Được sử dụng khi lớp đại điện cho tuyến cần phải kế thừa từ một lớp khác.

Tạo tuyến: Kế thừa từ lớp java.lang.Thread

Tạo lớp MyThread kế thừa từ Thread và nạp chồng phương thức run() của lớp Thread.

```
class MyThread extends Thread {
    ....
    public void run() {
        ...
    }
}
```

Tạo và thực thi tuyến.

```
Thread th1 = new MyThread();
Thread th2 = new MyThread();
th1.start();
th2.start();
```

Tạo tuyến: Kế thừa từ lớp java.lang.Thread

- **Phương thức start()** được Java xây dựng sẵn trong lớp Thread dùng để **cấp phát tài nguyên cho tuyến** trước khi thực hiện phương thức run(), do đó ta cần gọi đến phương thức start() chứ không phải phương thức run() để thực thi tuyến.
- Ta không được phép gọi phương thức start() lần thứ 2, nếu không **ngoại lệ IllegalThreadStateException** sẽ xảy ra.
- **Tuyến sẽ kết thúc** khi thực hiện hết lệnh bên trong phương thức run().

Tạo tuyến: Thực hiện interface java.lang.Runnable

Trong trường hợp lớp đã kế thừa từ một lớp khác, cần cài đặt giao tiếp Runnable để lớp có thể là một tuyến.

Runnable có duy nhất một phương thức run().

```
class MyClass extends SomeClass
    implements Runnable {
    ....
    public void run() {
            ...
    }
}
```

Tạo và thực thi tuyến.

```
Thread th1 = new Thread(new MyClass());
Thread th2 = new Thread(new MyClass());
th1.start();
th2.start();
```

Độ ưu tiên

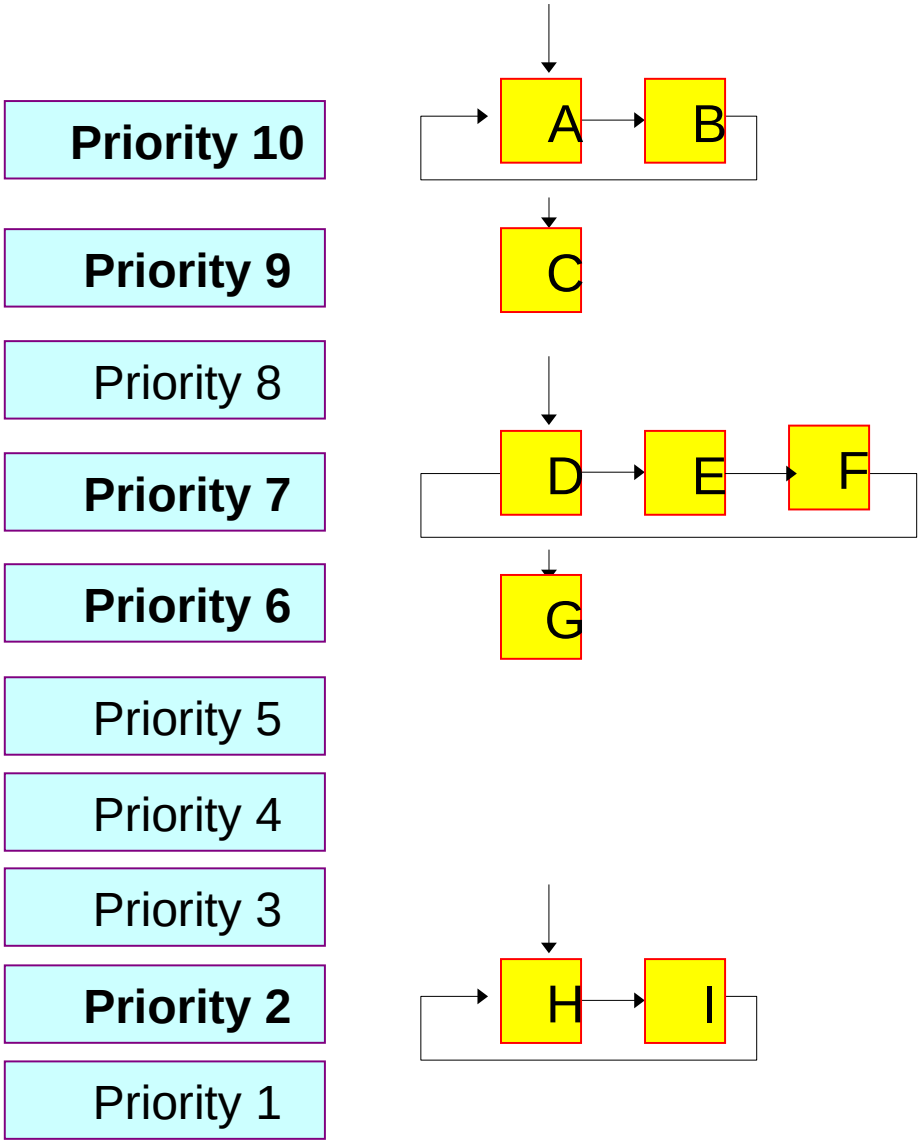
- Các tuyến trong Java có độ ưu tiên từ **Thread.MIN_PRIORITY** (giá trị 1) đến **Thread.MAX_PRIORITY** (giá trị 10)
- Tuyến có độ ưu tiên càng cao thì càng sớm được thực hiện và hoàn thành.
- Độ ưu tiên mặc định của các tuyến là **Thread.NORM_PRIORITY** (giá trị 5).
- Một tuyến mới sẽ thừa kế độ ưu tiên từ tuyến tạo ra nó.

Bộ lập lịch

- Bộ lập lịch (scheduler) của Java quản lý các tuyến theo cơ chế phân chia thời gian (time slicing). Từng tuyến sẽ được cấp một khoảng thời gian ngắn (time quantum) để sử dụng CPU. Trong khi thực thi, nếu đã hết thời gian được cấp thì dù chưa kết thúc tuyến cũng phải tạm dừng để cho các tuyến khác cùng độ ưu tiên dùng CPU.
- Các tuyến cùng độ ưu tiên luân phiên sử dụng CPU theo kiểu xoay vòng (round-robin).

Bộ lập lịch

- *Ví dụ:* Tuyến A và B sẽ luân phiên nhau thực thi cho đến khi kết thúc. Tiếp theo tuyến C sẽ thực thi đến khi kết thúc. Tiếp theo tuyến D, E và F sẽ luân phiên thực thi đến khi kết thúc. Tiếp theo tuyến G thực thi đến khi kết thúc. Cuối cùng tuyến H và I luân phiên thực thi đến khi kết thúc.
- *Nhận xét:* Các tuyến có độ ưu tiên thấp sẽ có nguy cơ bị trì hoãn vô hạn định.



Ví dụ 1

- Tạo ra 3 tuyến với độ ưu tiên mặc định. Công việc của mỗi tuyến là ngủ trong một thời gian ngẫu nhiên từ 0 đến 5 giây. Sau khi ngủ xong, các tuyến sẽ thông báo ra màn hình.

Ví dụ 1

```
class PrintThread extends Thread {
    private int sleepTime;
    public PrintThread( String name ) {
        super( name );
        sleepTime = ( int ) ( Math.random() * 5000);
        System.out.println( getName() + " have sleep time: " + sleepTime);
    }
    public void run() {
        try {
            System.out.println( getName() + " starts to sleep");
            Thread.sleep( sleepTime );
        } catch ( InterruptedException e) {  // sleep()
            e.printStackTrace();
        }
        System.out.println( getName() + " done sleeping" );
    }
}
```

Ví dụ 1

```
public class ThreadTest {
    public static void main( String [ ] args ) {
        PrintThread thread1 = new PrintThread( "thread1" );
        PrintThread thread2 = new PrintThread( "thread2" );
        PrintThread thread3 = new PrintThread( "thread3" );

        System.out.println( "Starting threads" );
        thread1.start();
        thread2.start();
        thread3.start();

        System.out.println( "Threads started, main ends\n" );
    }
}
```

Một số phương thức của Thread

- **boolean isAlive()**: Kiểm tra xem tuyến còn active hay không.
- **void yield()**
  - Mỗi tuyến sẽ được cấp CPU trong 1 khoảng thời gian nhất định, sau đó trả lại CPU để HĐH cấp CPU cho tuyến khác. Các tuyến sẽ nằm chờ trong **hàng đợi Ready** để nhận CPU theo thứ tự.
  - Khi gọi yield(), tuyến sẽ bị ngừng cấp CPU và nhường cho tuyến tiếp theo trong hàng đợi Ready.
  - Luồng không phải ngưng cấp CPU hẳn mà chỉ ngưng cấp trong lần nhận CPU đó mà thôi.

Một số phương thức của Thread

- **void sleep(long value)**: tạm dừng tuyến trong một khoảng thời gian value milli giây.
- **void join()**: Chờ tuyến gọi phương thức join() hoàn thành rồi tuyến cha mới được tiếp tục chạy.
- **void join(long value)**: Tuyến cha cần phải đợi **value milli giây** mới được tiếp tục chạy, kể từ lúc gọi join(long). Nếu tham số value = 0 nghĩa là phải đợi cho đến khi tuyến gọi phương thức join() kết thúc.

Một số phương thức của Thread

- **String getName()**: Trả về tên của tuyến.
- **void setName(String name)**: Thay đổi tên của tuyến.
- **long getId()**: Trả về id của tuyến.
- **Thread.State getState()**: trả về trạng thái của tuyến.
- **Thread currentThread()**: Trả về tham chiếu của tuyến đang được thi hành.
- **int getPriority()**: Trả về mức độ ưu tiên của tuyến.
- **void setPriority(int)**: Thay đổi mức độ ưu tiên của tuyến.

Ví dụ 2

- **Bài 1**: Chạy 2 tuyến t1 và t2 và thực hiện nhiệm vụ sau:
  - Tuyến t1 chạy xong mới được thực hiện tuyến t2
- **Bài 2**: Chạy 2 tuyến t1 và t2 và thực hiện nhiệm vụ sau:
  - Tuyến t1 chạy bình thường ngay sau khi start.
  - Tuyến t2 chỉ được thực hiện sau khi start 500 milli giây.

Ví dụ 2

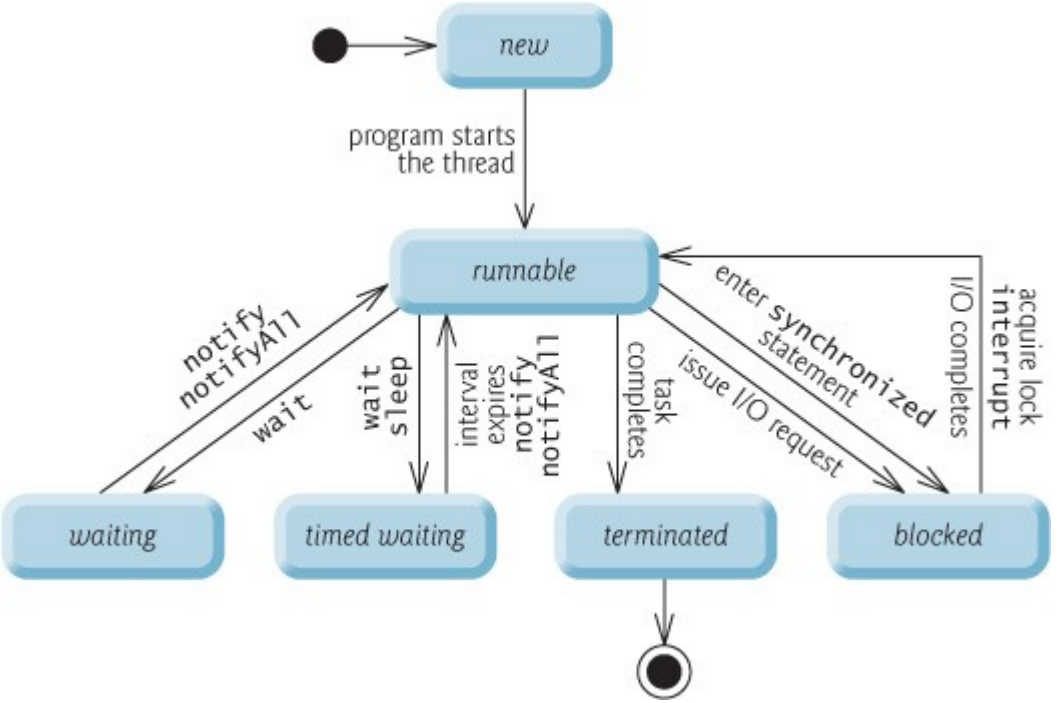
```
public class MainClass extends Thread {
    public MainClass(String name) {
        super(name);
    }
    public void run() {
        System.out.println(getName());
        for (int i = 1; i <= 5; i++) {
            try {
                System.out.print(i + " ");
                Thread.sleep(300);
            } catch (InterruptedException ie) {
                System.out.println(ie.toString());
            }
        }
        System.out.println();
    }
```

Ví dụ 2

```
    public static void main(String[] args) throws
    InterruptedException {
        Thread t1 = new MainClass("Thread 1");
        Thread t2 = new MainClass("Thread 2");
        t1.start();
        t1.join();
        t2.start();
        System.out.println("Main Thread Finished");
    }
}
```

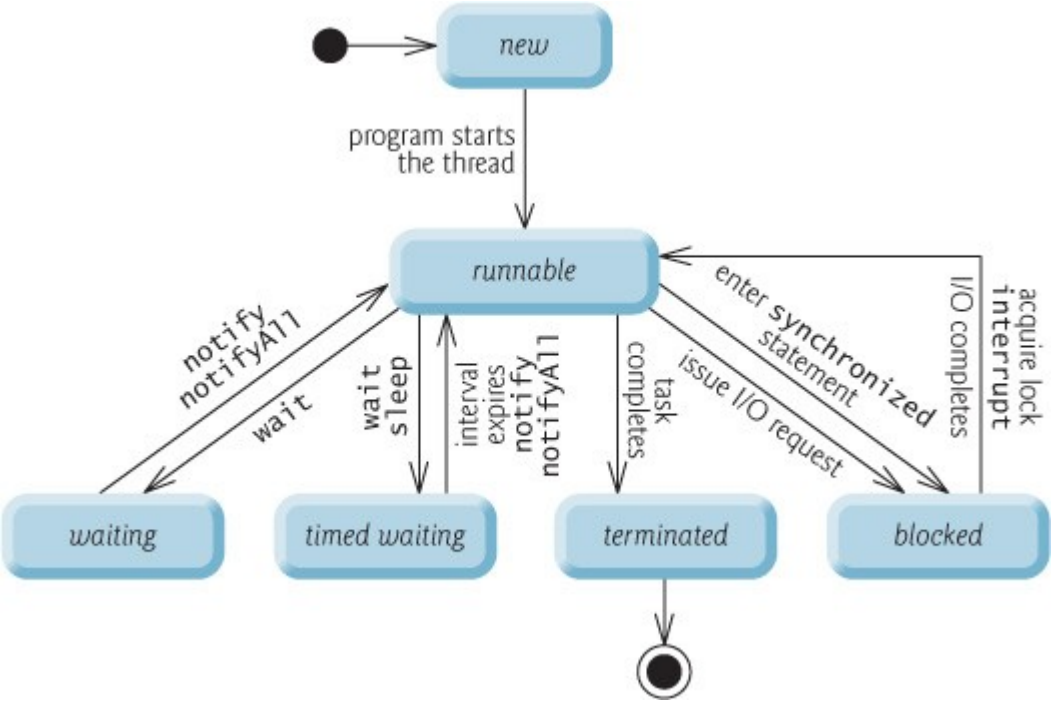
Vòng đời của một tuyến

- **Vòng đời của tuyến** trong Java **được kiểm soát bởi Java Virtual Machine**. Java định nghĩa các trạng thái của tuyến trong các thuộc tính static của **Thread.State**
  - **NEW**: Đây là trạng thái khi tuyến vừa được khởi tạo nhưng chưa được start(). Ở trạng thái này, tuyến được tạo ra nhưng chưa được cấp phát tài nguyên và cũng chưa chạy.
  - **RUNNABLE**: Sau khi gọi phương thức start() thì tuyến đã được cấp phát tài nguyên và các lịch điều phối CPU cho tuyến cũng bắt đầu có hiệu lực.



Vòng đời của một tuyến

- **Vòng đời của tuyến** trong Java **được kiểm soát bởi Java Virtual Machine**. Java định nghĩa các trạng thái của tuyến trong các thuộc tính static của **Thread.State**
  - **WAITING**: Tuyến chờ không giới hạn cho đến khi một tuyến khác đánh thức nó.
  - **TIMED WAITING**: Tuyến chờ trong một thời gian nhất định, hoặc là có một tuyến khác đánh thức nó.
  - **BLOCKED**: Đây là 1 dạng của trạng thái "Not Runnable", là trạng thái khi tuyến vẫn còn sống, nhưng hiện tại không được chọn để chạy. Tuyến chờ unlock một đối tượng mà nó cần.
  - **TERMINATED**: Một tuyến ở trong trạng thái terminated hoặc dead khi phương thức run() của nó bị thoát.



Đồng bộ hóa tuyến

- Do việc các tuyến trong cùng một chương trình có thể sử dụng chung vùng nhớ nên việc các tuyến cùng truy cập vào một đối tượng có thể sẽ đem lại kết quả không như mong muốn.
  - Ví dụ: Tuyến A có nhiệm vụ cập nhật đối tượng X và tuyến B có nhiệm vụ đọc dữ liệu từ X. Rất có thể xảy ra sự cố tuyến B đọc dữ liệu khi tuyến A chưa cập nhật dữ liệu cho đối tượng X.
- **Đồng bộ hoá tuyến** (thread synchronization) giúp cho tại mỗi thời điểm chỉ có một tuyến có thể truy nhập vào đối tượng còn các tuyến khác phải đợi.
  - Ví dụ: Trong khi tuyến A cập nhật X thì tuyến B chưa được đọc và ngược lại khi tuyến B đang đọc X thì tuyến A không được cập nhật.

Đồng bộ hóa tuyến

- Dùng từ khoá **synchronized** trên các phương thức cần thực hiện đồng bộ hoá của **lớp đại diện cho đối tượng**.
- Đối tượng khai báo phương thức synchronized sẽ có một **bộ giám sát** (monitor). Bộ giám sát đảm bảo tại mỗi thời điểm chỉ có một tuyến được gọi phương thức synchronized.
- Khi một tuyến gọi phương thức synchronized, phương thức synchronized sẽ **bị khoá**. Khi tuyến đó thực hiện xong phương thức, phương thức sẽ được **mở khoá**.

Đồng bộ hóa tuyến

- Trong phương thức synchronized, một tuyến có thể gọi wait() để chuyển sang trạng thái chờ cho đến khi một điều kiện nào đó xảy ra. Khi tuyến đang chờ, phương thức synchronized sẽ không bị khoá.
- Khi thực hiện xong công việc trên đối tượng, một tuyến cũng có thể thông báo (notify) cho các tuyến khác đang chờ để truy nhập đối tượng.
- Cần phải xử lý để không xảy ra trường hợp deadlock (Tuyến A chờ tuyến B và tuyến B cũng chờ tuyến A)

Ví dụ 3

- Giả sử có 2 tuyến: Tuyến **Producer** dùng để ghi dữ liệu vào một buffer (bộ đệm) và tuyến **Consumer** dùng để đọc dữ liệu từ buffer
  - Cần có sự đồng bộ hoá nếu không dữ liệu có thể bị Producer ghi đè trước khi Consumer đọc được hoặc Consumer có thể đọc một dữ liệu nhiều lần khi Producer chưa sản xuất kịp.

Ví dụ 3: Không đồng bộ

```
class Buffer {
    private int buffer = -1;

    public void set( int value ) {
        buffer = value;
    }

    public int get() {
        return buffer;
    }
}
```

Ví dụ 3: Không đồng bộ

```
class Producer extends Thread {
    private Buffer sharedBuffer;
    public Producer( Buffer shared ) {
        super( "Producer" );
        sharedBuffer = shared;
    }
    public void run() {
        for ( int count = 1; count <= 5; count++ ) {
            try {
                Thread.sleep( ( int ) ( Math.random() * 3000 ) );
                sharedBuffer.set( count );
                System.out.println( "Producer writes " + count);
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
        }
        System.out.println( getName() + " finished.");
    }
}
```

Ví dụ 3: Không đồng bộ

```
class Consumer extends Thread {
    private Buffer sharedBuffer;
    public Consumer( Buffer shared ) {
        super( "Consumer" );
        sharedBuffer = shared;
    }
    public void run() {
        for ( int count = 1; count <= 5; count++ ) {
            try {
                Thread.sleep( ( int ) ( Math.random() * 3000 ) );
                System.out.println( "Consumer reads " + sharedBuffer.get());
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
        }
        System.out.println( getName() + " finished.");
    }
}
```

Ví dụ 3: Không đồng bộ

```
public class MainClass {
    public static void main( String[] args ) {
        Buffer sharedBuffer = new Buffer();

        Producer producer = new Producer( sharedBuffer );
        Consumer consumer = new Consumer( sharedBuffer );

        producer.start();
        consumer.start();
    }
}
```

Ví dụ 3: Kết quả khi không đồng bộ

```
Producer writes 1
Producer writes 2
Consumer reads 2
Producer writes 3
Producer writes 4
Consumer reads 4
Producer writes 5
Producer finished.
Consumer reads 5
Consumer reads 5
Consumer reads 5
Consumer finished.
```

Ví dụ 3: Đồng bộ

- Giải pháp đồng bộ hoá:
  - Trước khi tiếp tục sinh dữ liệu và đưa vào buffer, Producer phải chờ (wait) Consumer đọc xong dữ liệu từ buffer.
  - Khi Consumer đọc xong dữ liệu, nó sẽ thông báo (notify) cho Producer biết để tiếp tục sinh dữ liệu.
  - Nếu Consumer thấy trong buffer không có dữ liệu hoặc dữ liệu đó đã được đọc rồi, nó sẽ chờ (wait) cho tới khi nhận được thông báo có dữ liệu mới.
  - Khi Producer sản xuất xong dữ liệu, nó thông báo (notify) cho Consumer biết.

Ví dụ 3: Đồng bộ

```
// Thiết kế lại lớp Buffer
class Buffer {
    private int buffer = -1;
    private boolean writable = true;

    public synchronized void set( int value ) {
        while ( ! writable ) {
            try {
                wait();
            } catch ( InterruptedException e ) {
                e.printStackTrace();
            }
        }
        buffer = value;
        writable = false;
        notify();
    }
```

Ví dụ 3: Đồng bộ

```
public synchronized int get() {
    while ( writable ) {
        try {
            wait();
        } catch ( InterruptedException e ) {
            e.printStackTrace();
        }
    }
    writable = true;
    notify();
    return buffer;
}
}
```

Tuyến ma

- Tuyến ma (daemon thread) thường là tuyến hỗ trợ môi trường thực thi của các tuyến khác.
  - Ví dụ: garbage collector của Java là một tuyến ma.
- Tuyến ma kết thúc chỉ kết thúc khi chương trình kết thúc.
- Các phương thức với tuyến ma:
  - void setDaemon(boolean isDaemon); // đặt tuyến trở thành tuyến ma
  - boolean isDaemon(); // kiểm tra tuyến có phải tuyến ma không

Nhóm tuyến (Thread group)

- Các tuyến có thể được đưa vào trong cùng một nhóm thông qua **lớp ThreadGroup**.
  - Ví dụ: nhóm tuyến tìm kiếm dữ liệu trên các tập dữ liệu khác nhau.
- Một nhóm tuyến chỉ có thể xử lý trên các tuyến trong nhóm
  - Ví dụ: ngắt tất cả các tuyến.
- Có thể tạo ra các nhóm tuyến là nhóm con của một nhóm tuyến khác.
- Nhóm tuyến đặc biệt: System, main

Hết Tuần 7

Cảm ơn các bạn đã chú ý lắng nghe !!!